The Farsite Project: A Retrospective

William J. Bolosky, John R. Douceur, Jon Howell
Microsoft Research Redmond

johndo@microsoft.com

ABSTRACT

The Farsite file system is a storage service that runs on the desktop computers of a large organization and provides the semantics of a central NTFS file server. The motivation behind the Farsite project was to harness the unused storage and network resources of desktop computers to provide a service that is reliable, available, and secure despite the fact that it runs on machines that are unreliable, often unavailable, and of limited security. A main premise of the project has been that building a scalable system requires more than scalable algorithms: To be scalable in a practical sense, a distributed system targeting 10^5 nodes must tolerate a significant (and never-zero) rate of machine failure, a small number of malicious participants, and a substantial number of opportunistic participants. It also must automatically adapt to the arrival and departure of machines and changes in machine availability, and it must be able to autonomically repartition its data and metadata as necessary to balance load and alleviate hotspots. We describe the history of the project, including its multiple versions of major system components, the unique programming style and software-engineering environment we created to facilitate development, our distributed debugging framework, and our experiences with formal system specification. We also report on the lessons we learned during this development.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management – d*istributed file systems*.

General Terms

Design, Documentation, Management.

Keywords

Serverless distributed file system, project management, system design iteration, software engineering, distributed debugging, formal system specification, tech transfer.

1. INTRODUCTION

From 1999 through 2005, Microsoft Research's Farsite project strove to build a scalable, serverless distributed file system. This system functions as a centralized file server, but its physical realization is dispersed among a network of incompletely trusted desktop workstations. The Farsite system was intended to provide both the benefits of a central file server (a central namespace, location-transparent access, and reliable storage) and the benefits of local desktop file systems (low cost, privacy, and resistance to localized faults). Farsite replaces the physical security of a server in a locked room with the virtual security of cryptography, randomized replication, and Byzantine fault-tolerant replicated state machines (RSMs) [6]. Farsite was designed to support desktop workloads in academic and corporate environments.

Simply stated, the goal of the Farsite project was to build a serverless distributed file system that is truly scalable, particularly to the scale of 10^5 machines. In the systems community, the term "scalable" typically refers to scalable protocols, algorithms, and distributed data structures. However, we chose to regard the term in a broader sense, arguing that practical scalability entails three other requirements: tolerance of failed machines, security against compromised machines, and automatic administration.

Strong fault tolerance is critical because, in a network of 10^5 machines, partial infrastructure failure is not merely the common case; it's the only case. There will never be a time when all machines are up and working at the same time. This precluded any fault-recovery solution that relied on waiting for a time when the infrastructure is fault-free, because no such time will ever come to pass.

Security is just as critical, not only against outside threats but also against inside ones. Virtually every large corporation includes at least one disgruntled employee [26], and universities are filled with curious, inventive, and occasionally antisocial students. Since we were proposing to run an organization's file services on the desktop computers of its constituents, the very infrastructure of the system could not be fully trusted.

Administration is already challenging. Large data centers simplify their administration with uniform hardware selection and standard machine configuration, which are not available options when running a system on extant desktop computers that have been arbitrarily configured by their immediate users. As a practical matter, any system whose manual administration load increases with the size of the system will run up against the limits of what the administrative staff can reasonably support. Thus, Farsite must adapt to the arrival and departure of machines and changes in machine availability, autonomically repartitioning its data and metadata to balance load and alleviate hotspots.

We were very upfront about our non-goals. We found it quite striking that as soon as we'd tell people that we were developing a new distributed file system with the above well-defined goals, they would immediately suggest other goals we could pursue as well. These other goals included large-scale write sharing, high-throughput parallel file I/O, transactionality, integration of file-system stores with database stores, new models of file-system structure (such as attribute-based systems), and weak consistency for offline access. In an effort to keep an already challenging

problem from becoming unmanageable, we explicitly disclaimed these additional goals.

The next section outlines the project's history. Section 3 details the multiple versions we wrote of three main system components: the file-system driver, the directory service, and the RSM substrate. Section 4 describes our programming models and how they evolved over the course of the project. Section 5 describes our distributed debugging framework, and Section 6 reports on our experiences with formal system specification. Section 7 describes a couple of key lessons we take away from the project.

2. PROJECT HISTORY

In early 1999, while ramping down other projects, we* started thinking seriously about Farsite as our next big systems project. In our spare time, we did a lot of brainstorming throughout the spring, and then over the summer, we did a feasibility study [4] to determine whether the idea of a serverless distributed file system running on desktop machines was even plausible. Buoyed by our results, we soon began designing the system architecture.

At the beginning of 2000, we started writing user-mode code, and by the spring, we started writing kernel-mode code for a file-system driver based on the Windows Single Instance Store filter driver (§3.1.1). We spent the remainder of 2000 doing two things: designing many aspects of the system that were both near-term and far-term, and implementing much of the near-term design. We designed the security model, the naming and certification architecture, the distributed duplicate-file-coalescing system, the quota-control architecture, the on-disk file format, a strategy for data durability, replica placement algorithms, and a scheme for directory encryption [12]. We implemented the secure messaging infrastructure, kernel/user communication code, the local cache manager, initial parts of the replicated state machine substrate (§3.3.1), lots of crypto stuff, and an epidemic upgrade system. By the end of 2000, we were stress testing the system on 40 machines in our newly set-up lab.

2001 was the year of rewrites. We rewrote the file-system driver based on the FastFAT file system (§3.1.2), and we then re-rewrote it as a mini-RDR (§3.1.3). We rewrote the directory service (§3.2.1) to support persistence. We rewrote our programming-model infrastructure (§4) to allow CTM-style code [2] in addition to event-driven code. It wasn't all rewrites; we also developed an encrypted key cache to improve crypto performance, worked out details of a metadata-hint caching strategy, and added MACs to the messaging system. We developed, analyzed, and simulated file-placement algorithms [7,8,9,10] and the distributed duplicate-coalescing system [11]. By the end of 2001, the code was functional and stable enough that we had a large benchmark – a modified version of the Andrew benchmark – working for very long runs.

In 2002, we started thinking seriously about distributing the directory service, and we concluded that we would need to start a new design (§3.2.2); we began with informal design and decided that for a problem of this complexity we would be better off trying formal specification (§6). On the implementation side, we got file replication working, got the directory state and local file cache stored persistently, implemented caching file data in the driver for performance, started implementing the distributed duplicate-coalescing system, and replaced our earlier file-based metadata storage system with a real database, namely SQL Server. We built a trace replayer, which helped to expose a raft of bugs that we spent a lot of time fixing, until we got the system stable enough to run a full day's replayed traces. In the process, we discovered that the high loads resulting from initializing the trace state could cause significant overloads in the system; it would take much of the following year to fully address this problem. At the end of 2002, we published our main system paper on Farsite [1].

In 2003, tech-transfer was very much on our minds. We worked with several prospective recipients for the Farsite technology, and we spent considerable time addressing issues that these groups considered important. This included getting Farsite's files to work with Windows Single Instance Store [5], checkpointing and restarting the directory service, supporting quotas, replacing the Byzantine-fault-tolerant RSM substrate (§3.3.1) with a fail-stop-fault-tolerant one (§3.3.2), and significant performance turning.

By 2004, aside from the ongoing design work on the distributed directory service (§3.2.2), all of our efforts were focused on tech-transfer. Major components included client-side crash recovery and checkpoint/restart for the centralized directory service. After nearly two years of effort in formal specification, we coded the distributed directory service this year. Among other changes, this involved switching the service's persistent store from SQL Server to the store in our atomic-action substrate (§4.3).

By 2005, most of our tech-transfer prospects had disappeared, at least those that had the potential of transferring a fully functional distributed file system. We refocused our efforts on transferring subsystems and components of the Farsite system, most notably our fail-stop-fault-tolerant RSM substrate (§3.3.2). During this year, most of the team began transitions to other projects. We continued developing the distributed directory service, mainly as an academic exercise targeted at publication [15].

To the extent that research projects ever have a formal end, Farsite came to an end in early 2006.

3. DESIGN ITERATIONS

Over the course of the Farsite project, we developed three implementations of the file-system driver, two implementations of the directory service, and two implementations of the RSM substrate. Although these cases seem superficially parallel, their reasons for re-implementation differ significantly. In particular, with the benefit of hindsight, we would have developed only one file-system driver, but we would still have developed both directory services and both RSM substrates.

3.1 File-System Drivers

We created the three versions of the file-system driver as our knowledge of the proper way to build Windows-based file systems improved. We started with a design that was loosely based on some previous kernel-mode work some project members had done, primarily because of familiarity with the design rather than because it the best way to structure the driver. After some

* Because this paper is a retrospective, covering work that was performed over almost seven years, the term "we" herein refers not merely to the present authors but rather to the entire project team.

experience with that problems of that approach, we tried a new design based on the existing Windows FastFAT file system on the theory that FastFAT was proven to work properly. However, we discovered that while it was an appropriate design for a purely local file system, it was at best awkward to do some operations necessary for a network file system, such as forcing the system cache to purge all of the data for a file so that it can be used remotely. This led to the final design as a mini-Redirector using Windows' (at the time) new functionality for implementing network file systems.

3.1.1 Driver built based on Windows SIS driver

Our initial attempt at a file-system driver was based on the Windows Single Instance Store (SIS) driver [5]. We chose this design because a number of us had just finished the SIS project, and so were very familiar with the code and its interaction with the system. SIS is implemented as a file-system filter driver, which sits above a local or remote file system on the IO stack, and is able to inspect, alter or directly implement any calls that are destined for the underlying file system. While this might seem an odd design for a service that is a file system itself rather than an added service for some existing file system, in point of fact Farsite (in all three incarnations) never implemented the on-disk portion of the file system and instead relied on NTFS to do that. So, one view of what Farsite does is to provide a (very involved) filter on top of a local NTFS implementation.

After a relatively small time, we discovered that while in principle Farsite could be viewed as a filter on NTFS, in practice we had to reimplement much of NTFS's functionality, including the very complicated pathname parsing and lookup code. Dealing with whether Farsite or NTFS owned the file object associated with handles was at best awkward and never worked quite right, and synchronizing access to files among the filter, NTFS and the rest of the NT kernel led to difficult-to-fix deadlocks.

While we abandoned this design early on, some of the code that we developed turned out to be useful and made it into the two subsequent versions. In particular, the code that implemented convergent encryption and Merkle-tree based content verification, and the code for passing messages between the driver and the user-mode daemon survived largely unchanged.

3.1.2 Driver derived from FastFAT

The experience with the SIS-derived driver led us to start over based on an existing file-system driver. We chose FastFAT rather than NTFS not for the obvious reasons (that it is published, documented, and publicly available), but rather because its code structure is much simpler. We removed the portion of the driver that dealt with on-disk structures and replaced it with calls into NTFS. There was no confusion about ownership of file objects, and the deadlocks largely disappeared. Furthermore, we were able to get a mostly-working implementation in fairly short order.

However, the FastFAT model didn't provide an easy way to evict data from the cache as is required to maintain consistency when a file is accessed writably by multiple nodes, it didn't have code dealing with access control lists (all files in FastFAT provide full access to all users), and it wasn't really designed to stop while calls are made up to user level during certain operations (particularly file open).

For a second time, we abandoned the driver and started more-or-less from scratch.

3.1.3 Driver built as mini-redirector

The final version of the driver is a mini-redirector written in terms of the Windows Redirected-Drive Buffering SubSystem (RDBSS), part of the Windows Installable File System Kit [23]. This is the way that Microsoft recommends building network file systems. At the time we started it, no documentation was available. However, because we were inside Microsoft, we were able to obtain a (preliminary) copy of the WebDAV mini-redirector to use as a template.

WebDAV seemed to be particularly apropos, because, like Farsite, it stored files in NTFS and implemented much of its functionality in user-mode, and so had to make calls between the kernel and user-mode components. When we merged it in with the existing user-mode component and the kernel-mode code we retained from the FastFAT implementation, however, we chose to keep two parallel user-to-kernel communication channels, the old Farsite one and the WebDAV one. They were very different in their design and never lived comfortably together. However, absent a good reason to get rid of one or the other, they co-existed from 2001 to 2004. In 2004, we were working on enabling the system to losslessly recover from a crash and restart of the user-mode component. Doing this meant keeping careful track of the set of calls that were outstanding, and the set of updates that the kernel had sent to the user-mode component. It turned out that the WebDAV communication channel was not well-suited to either task, and so we finally removed it in favor of the older Farsite channel.

The mini-RDR/RDBSS structure turned out to be a good match to Farsite's needs, even though it wasn't intended to be used with a file system that supplied all its data from the local disk[*]. RDBSS's function of handling most of the necessary interface with the system's virtual memory and caching components made coding significantly easier, eliminated several potential sources of deadlocks and race conditions, and did not remove any flexibility that we needed to get our driver working properly. We should have started with this design (and had it been documented when we started, we like to think we would have); we wasted far too much time on the first two implementations, and learned relatively little from them, beyond the fact that they were the wrong choice.

3.2 Directory Services

We built two separate versions of the directory service, one that is centralized and one that is distributed. Although this was not really intentional, in hindsight we believe it was a good thing to do. The centralized directory service provided an expedient path to getting a working system without the significant complexity of distributing file-system metadata. This was particularly important since it took two people nearly three years to design and build the distributed directory service.

---

[*] While the file data in fact go off-machine, all off-machine operations are performed in user-mode, so from the kernel's point of view file data are solely local.

3.2.1 Centralized directory service

We did not plan for the centralized directory service to remain centralized. Our intent had been to turn it into a distributed service, and we put some substantial effort into thinking about how to do that. However, we focused the bulk of our efforts on building a working service, irrespective of the eventual need for distribution.

Code structure – The centralized directory service was structured as a deterministic state machine, so that it could run on an RSM substrate (§3.3). Writing a sizeable and complex piece of software as a deterministic state machine was harder than we had expected. Not only is the state-machine model an unintuitive and unfamiliar way to structure systems code, but we had to eliminate every potential source of nondeterminism to prevent the state-machine replicas from diverging (§7.1). The service was originally written in an event-driven style (§4.1) but later evolved to a mix of event-driven and cooperative-task-management (§4.2) styles.

Leases – This centralized directory service temporarily loans authority over portions of file-system metadata to clients via leases. There are four classes of leases: content leases, name leases, mode leases, and access leases.

Content leases govern which client machines currently have control of a file's content. They can be read/write or read-only, and they follow single-writer/multi-reader semantics.

Name leases govern which client machine currently has control over a name in the directory namespace. Name leases are always read/write, and they transitively extend to all unused child names. Thus, when a client uses a name lease to create a new directory, it can immediately create files and subdirectories in that directory.

Mode leases govern which clients have a file open for Windows' various access and sharing modes [16], which provide explicit control over file-sharing semantics. There are six types of mode leases: read, write, and delete mode leases are used to open a file for read access, write access, and delete access, respectively. The other three, exclude-read, exclude-write, and exclude-delete mode leases are used to open a file *without* read sharing, write sharing, or delete sharing, respectively. The sense of these latter modes is inverted to preserve standard lease-conflict rules.

Access leases govern which clients can perform operations that have bearing on the deletion of a file. In Windows, a file is deleted by opening it, marking it for deletion, and closing it. The file is not truly deleted until the last handle is closed on a deletion-marked file. While the file is marked for deletion, no new handles may be opened on the file. There are three types of access leases: public, protected, and private. As one might expect, a public lease grants shared access and a private lease grants exclusive access. A protected lease grants shared access and the guarantee of a callback to the lease holder before any other client is granted access. Opening a file, closing a file, and marking a file for deletion all require access leases, selected in a combination to provide Windows' deletion semantics.

Access control – The centralized directory service enforces write-access control directly, by checking a user's cryptographically established identity against an access control list for the file or directory in question. Because directory service modifies state via a Byzantine-fault-tolerant protocol (3.3.1), we trust the service to apply only correct updates. By contrast, since a single faulty machine can inappropriately leak information, the service does not directly enforce read-access control. Instead, file content is encrypted so that it is only readable by clients whose users have an appropriate decryption key.

Distribution (conceptual) – We had developed several ideas for partitioning and distributing file-system metadata among multiple RSM groups. For purposes of discussion, we regard each RSM group as a *server*. Our intent had been to partition file metadata among servers according to file path names. Each client would maintain a cache of mappings from path names to their managing servers, similar to a Sprite prefix table [28]. The client could verify the authority of the server over the path name by evaluating a chain of delegation certificates extending back to the root server. To diffuse metadata hotspots, servers would issue stale snapshots instead of leases when the client load got too high, and servers would lazily propagate the result of rename operations throughout the name space.

Integration with driver –The directory service is all user-level code, but the client-side code that communicates with the service delegates some of its leased authority to the file-system driver. This enables many operations to be performed and logged directly in the driver instead of requiring an upcall to Farsite's user-level code, which is important for performance. For this reason, the driver understands much of the directory-service lease structure.

3.2.2 Distributed directory service

In attempting to turn the centralized directory service into a distributed service, we learned that many of the ideas we had for how to do this were problematic. In particular, the centralized service had assumed that metadata partitioning would eventually be partitioned according to file path name; however, this turns out to complicate rename operations that span partitions, so we opted to instead partition according to file identifier. In the absence of path-name delegation, name-based prefix tables are inappropriate. Similarly, if partitioning is not based on names, consistently resolving a path name requires access to metadata from all files along a path, so delegation certificates are unhelpful for scalability. Our ideas about stale snapshots and lazy rename propagation would allow the name space to become inconsistent, which can in turn cause orphaned loops in the namespace [15]. We thus built the distributed directory service from scratch, using an entirely different code structure, lease arrangement, and metadata distribution scheme.

Code structure – Like the centralized directory service, the distributed directory service was structured as a deterministic state machine; however, we never completed the integration of the distributed directory service with an RSM substrate. The service was written in an atomic-action coding style (§4.3). Nearly half of the application code is data-structure definitions and support routines that were mechanically extracted from a formal TLA+ specification of the directory service (§6.1).

Leases – The distributed directory service replaces the four classes of leases in the centralized service with two classes: shared-value leases and disjunctive leases. We observed that the complex lease classes in the centralized service were conflations of metadata with protections over that metadata. In the distributed

service, we separated these notions into specific metadata fields and comparatively simple leases that protect those fields.

Shared-value leases are conventional single-writer/multi-reader leases over fields of a file; these leases are used for a file's content field (really a hash of the content, since the actual content is stored separate from the directory service), child name fields, and various metadata fields, including the file's deletion disposition. As an important efficiency enhancement, the service has a special shorthand representation for an infinitely large set of child name fields; this representation is used to grant access to all child names of a file except an explicitly enumerated exclusion set.

Disjunctive leases are used to protect seven metadata fields: one field represents the set of clients who have handles open on the file, and six other fields each represent the set of clients who have the file open for each access or sharing mode. It would be inefficient for clients to have to obtain a read/write shared-value lease over one of these sets, merely to add itself to the set when opening a handle or to remove itself from the set when closing the handle. So instead, each client has a Boolean *self* value that it can write and a Boolean *other* value that it can read. The other value for each client x is defined as:

$$other_x = \sum_{y \neq x} self_y$$

where the summation symbol indicates a logical OR. Clients set their self values when opening handles and clear their self values when closing handles. Self values are protected by write leases, and other values are protected by read leases.

Access control (conceptual) – We have neither designed nor implemented access control in the distributed directory service. However, we believe we could extend the use of shared-value leases for the purpose of access control. The basic idea is that each file would have an access-control metadata field for each principal; the value of the field would indicate that principal's access rights. To access a file, a client would obtain a read lease over the requesting principal's access-control field, the value of which would let the client know whether to fail the operation or proceed. Like name leases, we would have a shorthand representation for leasing an infinitely large set of access-control fields; this could be granted to a client whose principal is authorized to change the access-control list. As in the centralized service, we would still use cryptography to protect the actual data against leaks.

Distribution – In contrast to our ideas for distributing the centralized directory service, the distributed directory service does not partition its metadata according to file path name. Instead, it partitions according to file identifier, so as to avoid implicitly coupling the logistical issue of which server manages which metadata with the operational issue of correctly implementing directory rename. The file identifiers have a tree structure that stays approximately aligned with the tree structure of the name space, so files can be efficiently partitioned with arbitrary granularity while making few cuts in the name space. One consequence of this tree structure is that file identifiers have variable length; this is not a problem in practice, because (1) the size of identifiers tends to remain quite manageable – generally smaller than an MD5 hash – and (2) the variability is encapsulated in a small class, so it is unseen by the rest of the system.

Since file-system metadata is distributed, it is necessary to provide a means for obtaining consistent access to file path names. In the absence of such means, two concurrent rename operations can produce an orphaned loop in the namespace [15]. The service provides such a means in the form of a recursive path lease, which is a read-only lease on the chain of file identifiers of all files on the path up to the file-system root. Path leases are recursively issued to the child files of the file whose path is being leased, so the path-lease load on any given file is bounded by the number of children it has.

There are two file-system operations that can span multiple servers: (1) renaming a file and (2) closing the last handle on a deletion-marked file, which unlinks the file. Rename can span three servers: the server that owns the source parent directory, the server that owns the destination parent directory, and the server that owns the file being moved. Close-and-unlink can span two servers: the server that owns the file being closed and the server that owns the directory from which the file is being unlinked. For these two operations, the servers coordinate their updates to ensure atomicity. In particular, the servers use two-phase locking, wherein one server acts as the leader and the other servers lock the relevant metadata fields of their files while they wait for the leader to coordinate the operation update.

Integration with driver (conceptual) – We have not integrated the distributed directory service with the file-system driver. There are two challenges to doing so: First, the driver's permission model is based on the leases used by the centralized directory service. We would have to either modify the driver's internal representation of operational permission or attempt to shim the distributed directory service's leases into a representation that could be understood by the driver. In practice, we would likely follow some combination of these approaches. Second, the driver uses fixed-size file identifiers, unlike the variable-size identifiers used by the distributed directory service. The most expedient way to address this mismatch is to provide a translation table at the interface between the components.

3.3 Replicated-State-Machine Substrates

We built two separate RSM substrates, one that tolerates Byzantine faults and one that tolerates only stopping faults. The Byzantine-fault-tolerant RSM was intended to address the original Farsite vision of running exclusively on client desktop machines. However, we found some potential product-group interest in running a Farsite-like system on trusted machines inside a data center, so we built a second RSM substrate that more efficiently supports the weaker faults expected in such an environment.

3.3.1 Byzantine-fault-tolerant RSM

We envisioned Farsite being deployed on the desktop computers in a university, wherein it would have to continue operating despite the curious tinkering of our hypothetical attacker, the SUSSCRAM (Smart Undergraduate Student with Source Code and Root Access to a Machine). This is an attack model that is ideally suited for Byzantine fault tolerance (BFT), because BFT assumes that malicious machine failures are independent, which they will generally **not** be if they are caused by software bugs or viruses.

Our BFT RSM substrate was based on the work of Castro and Liskov [6]. It ensures both safety and liveness as long as strictly

fewer than one third of all replicas are faulty. Although it requires weak synchrony assumptions to provide liveness, it needs no synchrony assumptions to guarantee safety. It executes read-only operations with a single round trip and read/write operations with two round trips. It avoids the expense of public-key cryptography in the common case, relying instead on symmetric-key message authentication codes, which are significantly faster to compute.

3.3.2 Paxos-based RSM

Inside a data center, the fault assumptions underlying BFT become less applicable. It is still possible for a machine to exhibit Byzantine behavior due to software bugs or viruses, but such behavior would not manifest with independent probability among machines. By contrast, stopping failures may well occur with independent probability if simple steps are taken to remove the most common correlating factors, such as co-locating machines of a replica group in a single rack, on a single power supply, or with a single cooling unit.

When stopping failures are the main concern, there are more efficient replica-coordination strategies than BFT. In particular, the Paxos algorithm [18, 19] ensures both safety and liveness as long as strictly fewer than half of all replicas are faulty. Our RSM substrate [22] allows not merely replication and coordination but also migration of the service to a new set of machines, which we call changing the *configuration* of the service. To accomplish this goal without excessively complicating the protocol, we introduced the idea of *configuration-specific replicas*, wherein each replica is associated with one and only one configuration. Multiple replicas for different configurations can execute concurrently on a single machine, but for simplicity they remain logically separated, although they share execution modules for efficiency.

4. PROGRAMMING MODELS

Based on our prior experience in building systems, we were well aware that the most challenging and frustrating bugs tend to arise from concurrency issues, yielding faulty behavior that is often difficult to reproduce reliably, let alone to diagnose and correct. We thus decided that the primary determinant for a programming model should be the prevention of concurrency bugs. This decision led us through three successive of programming models, each of which built upon the previous one. Code written in all three models runs side-by-side in the system, interacting across programming-model boundaries using shims and wrappers.

4.1 Event-Driven Programming

Initially, we followed an event-driven programming model, wherein we divided our code into uninterruptable regions that we encapsulated in continuations. Before a continuation ends, it often schedules one or more other continuations to execute, either at a later time or after some time-consuming non-computational task – such as a disk read – completes. This model reduces opportunities for race conditions and deadlocks, relative to the more common approach of programming with multiple execution threads [24, 27]. Event-driven programming avoids the key concurrency problem with standard multithreaded programming, namely the interruption of an executing task by another task that touches shared state. The event-driven model is significantly less complex and error-prone than carefully crafting locks and state-access policies, which when too liberal admit race conditions and when too conservative can cause deadlocks.

One challenge in writing event-driven code is maintaining a task's context across a set of event handlers that collectively implement the task, particularly as the code evolves and a single event handler is split into multiple handlers whenever a new I/O call is introduced. This context is trivial to maintain in a multithreaded model, because a task is typically performed by a dedicated thread, whose context is maintained by a stack that is preserved across I/O calls.

4.2 Cooperative Task Management

The difficulty of managing context in an event-driven program drove us to back to storing a task's state on a stack, while retaining the cooperative scheduling aspect of event-driven programming. In this cooperative task management (CTM) model [2], a task is – rather than a collection of event handlers that bridge between I/O calls – a single block of sequential code with well-defined yield points at each I/O call. The code between I/O calls runs without interruption by other tasks, much like an event handler would run. However, when an I/O call completes, rather than reconstructing the task's state from a manually pickled continuation, the state is already available on the task's stack.

Compared with multithreading, CTM reduces the opportunities for a task's state to be perturbed by another task; however, it does not completely eliminate these opportunities. In particular, when a task yields for an I/O call, other tasks may execute, and these other tasks might access or modify shared state. When the interrupted task then resumes, it must be prepared for the possibility that any shared state it had accessed prior to the I/O call has since changed.

To deal with this situation, we developed a programming idiom we called the pinning pattern [2§5], in which a task is divided into two phases. The first phase includes all of the task's read I/O calls (disk reads and network RPCs) inside a loop. If any I/O operation yields, the loop is restarted, because the task can no longer be certain that any value it has read still reflects the global state. Disk reads and RPC results are cached in memory, so this I/O read is unlikely to yield again on the next pass through the loop. The loop exits only when every read I/O operation executes without yielding, so at the end of the first phase, the task has a consistent representation of the portion of system state it cares about. In the second phase, the task performs its computation, and it records any state updates and outgoing network messages in an in-memory buffer. The contents of this buffer are subsequently written back to the disk or transmitted on the network by a separate housekeeping thread. Thus, after possibly looping multiple times through read I/O calls that yield, the task ultimately executes as an atomic block: The final iteration of the first phase does not yield, and the second phase never yields because it contains no I/O operations.

Although a strict adherence to the pinning pattern results in correct code, it does not lend itself to modularity. For example, a subroutine that performs both reads and writes cannot be called from either phase of a task, since the first phase must contain no writes and the second phase must contain no reads. The pinning pattern also demands significant discipline from the programmer, and it is quite unforgiving if the task departs from the pattern's strictures in any way.

4.3 Atomic Actions

The difficulties of restricted modularity and stringent coding constraints led us one step further. Our programming model had progressively evolved toward writing tasks as atomic blocks, so we finally decided to implement our tasks as full-blown atomic actions [21]. Our intent was twofold: First, by addressing the issue of state consistency at a single place in the code, we hoped to eliminate the class of consistency errors we had experienced due to the difficulty of honoring the pinning pattern consistently. Second, we wanted to write our application code in a more modular, readable, and maintainable style than the pinning pattern would allow.

We built an application-generic atomic-action substrate, on top of which application-specific code is written using an action for each sequential task. The substrate isolates each action's effects by mediating access to state, time, and messages.

State – For ease of implementation, we used an explicit state interface rather than a transparent memory interface. This decision prevents the application from reusing old data-structure code to organize its data. Therefore, instead of a simple address space, the substrate provides an interface of single-key dictionaries with custom keys. This is nearly as easy to implement as a linear address space, but it supports sorting, efficient indexing, and range queries, which largely makes up for the inability to use standard data structures in the application code.

Atomic state is implemented using redo logs. A *redo log* is a buffer of ⟨address,value⟩ pairs written by an action. By buffering the writes, the redo log isolates the effects from other actions. Redo logs are chained, each using the next as its backing store. A read that cannot be satisfied by any entry in a redo log is passed on to that log's backing store. An action is atomically committed by simply referring to its redo log as the new current state of the system. An action is atomically aborted by simply discarding its redo log.

Time – The natural interface to time is to provide the value of the clock. For example, a host might evaluate whether a lease has expired by evaluating the expression:

```
GetCurrentTime() > lease.expiration
```

An alternate interface is to let the code make Boolean queries of the time:

```
IsNowLaterThan(lease.expiration)
```

The latter approach constrains how much information about time flows into the application code, which gives the substrate more freedom and can result in fewer action aborts. Specifically, when an action queries a time predicate, the substrate evaluates and records the constraint enforced by the predicate before returning the result to the application. At the end of the action, if the current time violates any of the predicate constraints, the action aborts. Thus, the effective time of each action is the time it commits, which trivially enforces the temporal consistency of commit order.

Messages – When a message arrives, it is stored into the shared state, and optionally an action is started to process it. When an action sends a message, the message is held in a buffer. If and when the action commits, all buffered messages are transmitted. If the action aborts, the messages are discarded.

5. DISTRIBUTED DEBUGGING

System components built on the atomic-action programming model can be deterministically replayed in the distributed system. We have used this facility to isolate several would-be Heisenbugs.

Although deterministically replaying a distributed system is an old idea [25], it can be difficult to achieve because nondeterminism enters a system any number of ways, which makes it challenging to capture all of its sources and constrain a later run to obey the observed behavior.

The atomic action model provides ideal support for a replay system, because isolation requires mediating all of an action's access to the outside world. Likewise, action commits are the only way that a host's state can change. Therefore, a host is a state machine whose evolution is completely determined by the sequence of actions it commits. Each action can be completely characterized by the action's identity, argument values, and the value of time observed by the action.

The programmer must explicitly cooperate with this discipline. It is forbidden to record state that lives beyond a transaction outside of the shared state interface, since changes to that state are not serialized or rolled back upon action abort. As an example, our pseudorandom-number generator object uses the state interface, rather than conventional heap storage, to store its state.

The bane of distributed systems implementation is the difficulty of debugging a system in which data is widely dispersed among machines. There is no single thread of control to break, and a crash is not particularly likely to be reproducible because the run that led to it cannot be deterministically reproduced. Divergence can arise from the innate entropy of the distributed environment or from the perturbations of monitoring code.

The ability to deterministically replay the system facilitates debugging by allowing us to probe the system while ensuring that it continues to exhibit the broken behavior. Probing may involve using a debug build with extensive assert checking or printf logging. It may involve modifying the executable to introduce a new sanity check, or even to repair the behavior, although a dramatic repair may make the rerun diverge from the logged run. Probing may also involve remapping the hosts in the distributed system onto different physical machines. For example, we have replayed a multiple-machine deployment on a single physical machine, in a single process, with a debugger attached. Reconfiguration cannot involve changing the number of logical hosts in the distributed system, however, because such a change would lead to a different set of logs and different behavior.

During replay, we keep the hosts causally synchronized. When a host's log indicates it should evaluate an action that depends on the receipt of a message, the scheduler waits until that message is actually received from the sending host before proceeding. This ensures that events and debugging messages occur in a sensible order in the replayed system.

6. FORMAL SYSTEM SPECIFICATION

When we began developing the distributed directory service described in section 3.2, we started with the approach we had always used for distributed-system design: informal specification using textual description, pseudo-code, and block diagrams. We quickly found ourselves getting quite lost in the details of the

design, largely because the distributed directory service is a highly constrained design problem: The service must be scalable, strongly consistent, and resistant to Byzantine faults. Initially, we were not even clear on what "resistant to Byzantine faults" meant, but we were concerned that, as the number of BFT groups in the system grows, so too grows the probability that at least one group will have more machine failures than it can handle, and we did not want to let a random faulty group take out the entire system [14].

To tackle these compound challenges, we – slowly and quite reluctantly – began using formal system specification for the distributed directory service. We found that a formal specification can serve as an *abstract prototype* that calls attention to errors in the design before implementation begins. This is valuable not only because the high level of abstraction makes it easy to reason about the design without getting bogged down in implementation details, but also because radical changes to the specification tend to require far less effort than comparable changes to an implemented system. Although we do not conclude that formal specification is an appropriate tool for most system designs, we do believe it can have applicability for subtle and highly constrained solution spaces, wherein correctness is difficult to reason about.

The benefits of formal specification accrue from two components: a formal mathematical syntax and the concept of refinement.

6.1 Formal Syntax

For specifying the distributed directory service, we used the TLA+ language [20], which provides well-defined syntax for set theory and first-order logic, syntactic shorthand for defining systems as state machines, and temporal logic for reasoning about liveness. The set theory is easy to use by anyone with a basic mathematics background. We found the state machine syntax quite natural once we thought a little bit about how invariants are maintained inductively. We used very little of the temporal logic, because we focused on safety properties rather than liveness properties. Although the language looks intimidating at first, it is quite accessible systems builders.

Formal syntax provides three benefits relative to informal specification: unambiguity, decoupling abstraction from precision, and explicit indication of dependencies.

Unambiguity – Because math and sets are well-defined, formal syntax does not admit the ambiguity that can creep into prose specifications. For example, in our informal specs, we had written, "If a client holds a name lease on name *E* in directory *D*, then the client implicitly holds a name lease on all nonexistent children of *E*." This rule turns out to be ambiguous, because a server may have stale information about whether a client has created a child of *E*, in which case the server and client interpret the meaning of an outstanding lease differently. Formal syntax forces the designer to settle on some unambiguous interpretation of the abstract idea, even if it is a straw man. As later components of the design make use of the idea, there is no ambiguity about what the current specification means; either it satisfies the needed properties, or it must be changed.

Decoupled abstraction and precision – Text and pseudo-code tend to couple abstraction and precision. Raising the level of abstraction is commonly achieved by (in text) leaving out details or (in pseudo-code) by stubbing out subroutines. High-level prose is generally imprecise about its meaning, and stubbed-out pseudo-code is similarly imprecise except in the rare cases in which an omitted subroutine is well-defined at an abstract level (e.g., stable sort by case-insensitive Unicode primary key). By contrast, formal syntax can be employed at any level of abstraction. For instance, we at first specified our distributed clock by formally writing down the properties it achieved [13], using a few lines of TLA+. Later, we went back and replaced this with a formal description of the messages exchanged between machines and the corresponding state updates. At either level of abstraction, the specification was precise about its meaning.

Explicit dependencies – When working with prose specification documents, it is not immediately obvious how changes to one part of the specification affect other parts. When using a formal syntax, one cannot refer to a component of the design except by explicit reference to the symbol that defines it. This allows the designer, when making specification changes, to grep for other definitions that should be inspected to ensure they remain compatible with the new definition.

6.2 Refinement

Whereas formal syntax helps find errors during the design process, refinement is a technique that guides the design process. The technique focuses the process on which ideas are necessary and helps the designer know when the job is complete. In the context of distributed systems, the refinement technique involves constructing three artifacts: a semantic spec, a distributed-systems spec, and a refinement.

Semantic spec – The semantic specification describes the intended behavior of the system from the viewpoint of the systems users. Farsite logically functions as a centralized file server, so Farsite's semantic spec defines the behavior of a centralized file server, namely the file-system operations open, close, read, write, create, delete, and move/rename. To address the requirement of resisting Byzantine faults, the semantic spec also specifies how faults can manifest to the users. A significant challenge was finding a semantic spec that was neither unrealistically strong nor uselessly weak.

Distributed-system spec – The distributed-system specification describes how a set of machines and BFT groups interact: receiving file-system requests, sending messages, receiving messages, modifying local-machine state, and returning results of file-system requests. The distributed-system spec can be regarded as the main product of the refinement process, insofar as it precisely describes the behavior of the constituent machines in the distributed system. Turning a distributed-system spec into a working system is merely a matter of writing an implementation for the single-machine components of the system, which can be done without any further thought about the distributed-system aspects of the problem.

Refinement – The refinement is a formal correspondence between the semantic spec and the distributed-system spec. The semantic spec describes an abstract structure, and the refinement describes how the distributed, asynchronously updated structures spread out across the distributed system can be interpreted as the abstract structure of the semantic spec. Constructing the refinement guides the construction of the distributed system. When faced with a problem, we would brainstorm a possible solution in the distributed system and then ask, "How does this solution refine to

the semantic spec?" That simple question consistently led us to immediately understand which invariants we needed to maintain.

No matter how knotty the distributed data structures in our system become, our refinement tells us how to interpret them. In particular, a key concept in the distributed directory service is *authority*, which indicates which item of distributed state should be regarded as the value of a particular semantic datum. The distributed system must guarantee that its messages and state updates always preserve the invariant that a single host is authoritative over any semantic datum, and furthermore that authority is transferred among hosts in a reasonable way. In this context, "reasonable" also includes the concept of delegating authority in a way that restricts the influence of Byzantine faulty hosts in the distributed system.

6.3 Anecdotal experience

The value of this methodology is highlighted by our experience in developing the procedure for the move/rename operation. Although the rename operation is semantically straightforward, at the distributed-system level it involves up to four hosts interacting to perform an atomic operation. Moreover, any subset of these machines may be Byzantine-faulty, and our requirement for restricting Byzantine faults forbids us from allowing the state of the non-faulty machines to become polluted. As we developed a rename procedure, we recorded the distributed-system behavior in TLA+. Once we had a formally precise description, we could reason through the behavior, and it turned out that our first rename procedure was flawed, so we started over with a different procedure. We repeated this process **19 times**, in several cases fundamentally changing the distributed-system state schema, until we achieved a distributed-system spec that refined to a reasonable semantic spec.

If we had gone directly to an implementation without first writing a distributed-system spec, it would have been far more costly to make the necessary changes. Furthermore, it would have been more difficult to understand the distributed-system aspects of the problem without getting mired in implementation details.

7. LESSONS

Over the course of the Farsite project, we have learned many lessons, most of which we have presented in earlier papers [1, 15]. Overall, however, two lessons stand out: First, in a real system, determinism is harder to achieve than you would expect. Second, a system that first seemed to be addressing a disk problem turned out to be addressing a network problem.

7.1 Determinism Is Harder Than Expected

Running a service in a replicated state machine requires that the service be deterministic. Theoretically, this sounds fairly simple, but in practice, we found that non-determinism creeps into code from many sources. A striking anecdote illustrates the point:

After much debugging effort, we once tracked down a replica-divergence bug that turned out to be calls to the system quicksort function *qsort* producing different outputs from identical inputs. Because quicksort is not a stable sort, the function has freedom to produce different orderings when not all keys are unique. At first we suspected that qsort does something like using rand() to select a pivot, thereby causing different invocations with the same inputs to behave differently; however, investigation revealed that this is not the case. It turns out that in our test configuration, one machine in the RSM group was running Windows 2000, and another was running Windows XP. The implementations of qsort in Win2K and WinXP differ in a way that can cause them to produce different results given identical inputs if the inputs have elements with equal keys. Because the qsort routine is dynamically linked, the Farsite code picks up different versions of it on different machines.

The main lesson from this is that code boundaries are not as clear as one might wish, and any call to any routine that is outside the controlled code is a potential entry point for non-determinism.

7.2 A Network Problem, Not a Disk Problem

A number of the Farsite team members had previously worked on Tiger [3], a scalable video file server built from a collection of personal computers and a network switch. When we started that project, we thought that that hard issue would be getting the video data from the disks. After designing and implementing Tiger, it turned out that once we had the initial idea of how to schedule disk accesses, the disks weren't the problem. Instead, managing the network in terms of overloads, failures and getting the protocols right consumed nearly all of our time and mental effort. That is, we concluded that the video server problem was more about the network than the disk.

We had an analogous experience with Farsite. In the beginning, we thought that the hardest issue would be finding enough disk space in order to make sufficient replicas to ensure reasonable file availability. In fact, the first Farsite publication [4] was a feasibility study that considered this question, and concluded that sufficient disk space was available; a second early publication [11] addressed how to find duplicate files to coalesce to save space. By the end of the project, it was clear that the hardest problems involved the directory service, lease protocol, and consistent distributed crash recovery. That is, Farsite, like Tiger before it, really was more of a network problem than a disk/file system problem.

As with many things, this seems obvious only in retrospect. The difficulty in solving network problems is really a more complicated version of the local concurrency problem that we were trying to ameliorate by adopting the single threaded programming models described in §4. We couldn't adopt such a simple strategy across nodes because we needed the concurrency and we also had to worry about (possibly Byzantine) failures. The design decision to treat even BFT groups as potentially malicious extended this problem into the directory service, and gave rise to much of the complexity there. We were able to solve the file availability problem using simple statistical models and some assumptions about usage based on measurements of Microsoft machines. The protocol design problem required TLA+ and years of careful thought.

After twice making the same error about identifying where the difficulty in a system design lies, we hope that we and our readers can learn from where we went wrong and find new and more interesting mistakes to make in the future.

8. ACKNOWLEDGMENTS

The Farsite project has benefited from the talents of many people. In addition to the authors, the project team included Atul Adya,

Miguel Castro, Gerry Cermak, Ronnie Chaiken, Jon Howell, Jacob Lorch, Marvin Theimer, and Roger Wattenhofer. Project interns have included Kaustuv Chaudhuri, David Ely, Thanos Papathanasiou, Patrick Reynolds, Rodrigo Rodrigues, Brandon Salmon, and Dmitrii Zagorodnov. We also thank the many others who have made contributions and shared illuminating discussions, including Dimitris Achlioptas, Josh Benaloh, Yuqun Chen, Dinei Florencio, Tim Harris, Eric Horvitz, David Hovel, Simon Peyton-Jones, Rajeev Rajan, Balan Sethu Raman, and Dan Simon.

9. REFERENCES

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, R. P. Wattenhofer. "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," *5th OSDI*, Dec 2002.

[2] A. Adya, J. Howell, M. Theimer, B. Bolosky, J. Douceur. "Cooperative Task Management without Manual Stack Management." *USENIX Annual Technical Conference*, 2002.

[3] W. J. Bolosky, J. S. Barrera III, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhrvold, and R. F. Rashid. "The Tiger Video Fileserver," in NOSSDAV '96, April, 1996.

[4] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs." *SIGMETRICS 2000*, Jun 2000.

[5] W. J. Bolosky, S. Corbin, D. Goebel, J. R. Douceur. "Single Instance Storage in Windows 2000." *4th Usenix Windows System Symposium*, Aug 2000.

[6] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", *3rd OSDI*, USENIX, Feb 1999.

[7] J. R. Douceur and R. P. Wattenhofer. "Large-Scale Simulation of a Replica Placement Algorithms for a Serverless Distributed File System." *9th MASCOTS*, IEEE, Aug 2001.

[8] J. R. Douceur and R. P. Wattenhofer, "Modeling Replica Placement in a Distributed File System: Narrowing the Gap between Competitive Analysis and Simulation", *ESA 2001*, Aug 2001.

[9] J. R. Douceur and R. P. Wattenhofer, "Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System", *15th DISC*, Oct 2001.

[10] J. R. Douceur and R. P. Wattenhofer, "Optimizing File Availability in a Secure Serverless Distributed File System", *20th SRDS*, IEEE, Oct 2001.

[11] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, M. Theimer, "Reclaiming Space from duplicate Files in a Serverless Distributed File System", *ICDCS*, Jul 2002.

[12] J. R. Douceur, A. Adya, J. Benaloh, W. J. Bolosky, G. Yuval. "A Secure Directory Service based on Exclusive Encryption." *18th ACSAC*, 2002.

[13] J. R. Douceur; J. Howell. "Scalable Byzantine-Fault-Quantifying Clock Synchronization." *Microsoft Research tech report MSR-TR-2003-67*, 2003.

[14] J. R. Douceur, J. Howell. "Byzantine Fault Isolation in the Farsite Distributed File System." *5th IPTPS*, 2006.

[15] J. R. Douceur, J. Howell. "Distributed Directory Service in the Farsite File System." *7th OSDI*, 2006.

[16] J. M. Hart. *Win32 System Programming: A Windows(R) 2000 Application Developer's Guide, Second Edition*, Addison-Wesley, 2000.

[17] J. Kistler, M. Satyanarayanan. "Disconnected operation in the Coda File System." *TOCS* 10(1), Feb 1992.

[18] L. Lamport. "The part-time parliament." *TOCS*, 16(2):133–169, May 1998.

[19] L. Lamport. "Paxos made simple." ACM *SIGACT News*, 32(4):18–25, Dec. 2001.

[20] L. Lamport. *Specifying Systems*. Addison-Wesley, 2003.

[21] D. B. Lomet. "Process structuring, synchronization, and recovery using atomic actions." *ACM Conference on Language Design for Reliable Software*, SIGPLAN Notices 12(3), pp. 128-137, 1977.

[22] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, J. Howell. "The SMART Way to Migrate Replicated Stateful Services." *EuroSys* 2006.

[23] Microsoft Corporation. "IFS Kit - Installable File System Kit." http://www.microsoft.com/whdc/DevTools/IFSKit/default.mspx

[24] J. K. Ousterhout. "Why Threads Are a Bad Idea (for most purposes)." *USENIX Annual Technical Conference*, 1996.

[25.] M. Ronsse, K. De Bosschere, J. C. de Kergommeaux. "Execution replay and debugging." *Automated and Algorithmic Debugging*, pp. 5-18. 2000.

[26] S. T. Shafer, "The Enemy Within", *Red Herring*, Jan 2002.

[27] R. von Behren, J. Condit, E. Brewer. "Why events are a bad idea (for high-concurrency servers)." *HotOS IX*. May 2003.

[28] B. Welch, J. Ousterhout. "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System," *6th ICDCS*, 1986.