Experience with a Distributed File System Implementation

Randolph Y. Wang and Thomas E. Anderson

University of California at Berkeley

Michael D. Dahlin

University of Texas at Austin

Abstract

This paper highlights some of the lessons learned during the course of implementing xFS, a fully distributed file system. xFS is an interesting case study for two reasons. First, xFS's *serverless* architecture leads to more complex distributed programming issues than are faced by traditional client-server operating system services. Second, xFS implements a complex, multithreaded service that is tightly coupled with the underlying operating system. This combination turned out to be quite challenging. On one hand, the complexity of the system forced us to turn to distributed programming tools based on formal methods to verify the correctness of our distributed algorithms; on the other hand the complex interactions with the operating system on individual nodes violated some of the tools' assumptions, making it difficult to use them in this environment. Furthermore, the xFS system tested the limits of abstractions such as threads, RPC, and vnodes that have traditionally been used in building distributed file systems. Based on our experience, we suggest several strategies that should be followed by those wishing to build distributed operating systems services, and we also indicate several areas where programming tools and operating system abstractions might be improved.

1 Introduction

The recent emergence of high-performance local area networks [4, 7] and cluster technology [2, 50] has resulted in a renewed interest in distributed operating systems services. Relative to the client-server programs of the previous generation, the new peer-to-peer distributed systems enabled by low latency, high bandwidth communication are more complex due to their performance, scalability, and availability requirements. This growing complexity has outpaced our understanding of how to engineer these systems.

xFS, a network file system described in a previous paper [3], is an example of such a *serverless* distributed system. It distributes its cache, secondary storage, and metadata management over closely coupled workstations. The decentralized nature of the system, while offering superior performance, scalability, and availability compared to traditional client-server file systems, also increases its complexity. Based on these observations, we have implemented a new version of xFS. We believe our experience may offer insight for future system builders and encourage the development of new tools and interfaces that can ease their jobs. Specifically, implementors of future distributed operating systems services should consider the following observations:

- *Formal verification tools can significantly simplify the development and debugging of complex distributed algorithms.* Although formal tools are becoming widely used in the distributed shared memory (DSM) and network protocol design community, operating systems designers have been less quick to adopt them. The increasing complexity needed to meet the performance, scalability, and reliability demands of distributed operating systems services means that designers ignore these tools at their own peril.

- *Single-threaded event loops can reduce the difficulty of composing multithreaded subsystems.* As heavily multithreaded software becomes prevalent, designers of services that interact with multiple multithreaded subsystems will find it increasingly difficult to reason about the behavior of their systems. Our experience bears out the hypothesis that the judicious use of single-threaded event loops can be effective in managing this complexity [37].

Many of the difficulties we encountered were rooted in mismatches between the service we were constructing and the tools and interfaces on which we built. Programmers should be aware of these mismatches so that they can structure their applications to minimize this dissonance or avoid using unsuitable interfaces. These mismatches also suggest areas where the tools and interfaces should be improved. In particular, improvements in the following areas would greatly benefit the development of other services like xFS:

- *Formal verification tools should provide more support for multithreaded applications and applications that interact with complex or blocking operating system functions.* A great deal of the beauty and strength of these tools is that they abstract complex problems into a form that can be analyzed. Unfortunately, this often means assuming a simple environment that does not match today's operating systems. In our case, we built the xFS coherence protocol using a tool that had been designed for DSM coherence. Although the tool was invaluable for designing the protocol itself, it was challenging to integrate the resulting protocol with multithreaded daemons and with operating system calls (particularly those that might block).

- *Although the RPC abstraction is useful, to better support distributed operating systems RPC should be extended to support peer-to-peer communications patterns, and it should be reimplemented to provide better performance.* In particular, RPC's request-response paradigm does not match the requirements of serverless designs that do not rely on centralized servers to satisfy client requests; we found it unnatural to use RPC to synthesize the multi-party communication needed by xFS. A continuation-passing extension to RPC — whereby one node can pass the right to respond to an RPC to another node — would, for example, better match a serverless system's needs. Furthermore, the overhead imposed by current RPC implementations' layered architectures result in overheads that are unacceptable on new, low-latency networks. We found that programming directly on top of Active Message [51, 33] provided acceptable semantics and excellent performance, but a high-performance, continuation-passing RPC system would be simpler to program than raw Active Messages.

- *Kernel vnode layers should provide more support for cache coherence, and they should be reimplemented to reduce cache miss overhead.* The weak consistency models provided by traditional distributed file systems such as NFS [43] and Andrew [23] do not satisfy the more strict consistency models assumed by many distributed applications. The vnode layer should provide a better coherence interface to allow file systems to support the strong consistency needed by these applications. Also, today's fast networks pressure the vnode layer to provide better performance; these networks are sufficiently fast that the vnode layer can increase the latency of accessing data from a cache on a remote node by an order of magnitude. To get correct behavior and acceptable performance, xFS had to modify these aspects of the operating system that were not exported via the vnode layer.

Many of these issues have been observed to varying degrees in other systems. We found them to be particularly challenging in xFS because our system combined complex distributed algorithms arising from the system's serverless architecture with complicated individual node behavior arising from the system's interaction with the host operating system. Moreover, this style of interaction is becoming

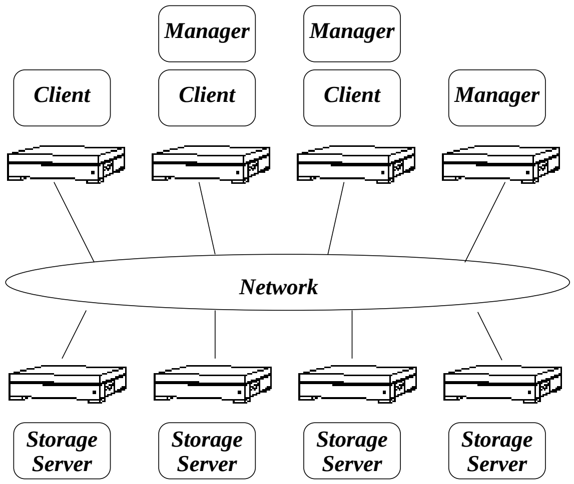


Figure 1: A sample xFS configuration. Clients, managers, and storage servers provide a global memory cache, a distributed metadata manager, and a striped network disk respectively.

more common; many experimental systems including Vesta [10], Zebra [20], Treadmarks [27], SAM [45], GMS [15], CRL [25], Inktomi [53], and Petal [31] have elements similar to those found in xFS. Thus, we believe these lessons are widely applicable to this new generation of peer-to-peer distributed systems.

After we provide a brief overview of xFS in section 2, section 3 describes the use of a formal verification system in the xFS cache coherence engine. Next, in section 4, we shift our focus to the components on a single client. We report on our experience with threads. Then in section 5, we turn our attention to communication; we discuss the use of RPC and Active Message in the context of non-traditional communication patterns. And finally in section 6, we focus on the lowest level of the system. We describe kernel support (or lack thereof) for a high-performance distributed file system. We summarize the lessons in section 7. When applicable, we also draw on our experiences from other systems.

2 xFS Overview

The main design philosophy of xFS is the elimination of any centralized bottleneck and the efficient use of all resources in a network of workstations. The three main components of the systems are the *clients*, the *managers*, and the *storage servers*. Under the xFS architecture, any machine can be responsible for caching, managing, or storing of any piece of data or metadata by instantiating one or more of these subsystems. Figure 1 shows a sample xFS installation.

Each of the three subsystems implements a specific interface. A client accepts file system requests from users, sends data to storage servers on writes, forwards reads to managers on cache misses, and receives replies from storage servers or other clients. It also answers cooperative cache forwarding requests from the manager by sending data to other clients. The job of the metadata manager is tracking locations of file data blocks, and forwarding requests from clients to the appropriate destinations. Its functionality is similar to the directory in traditional DSM systems such as DASH [32] and Alewife [30]. Finally, the storage servers collectively provide the illusion of a striped network disk. They receive striped writes from clients. They also react to requests from managers by supplying data to the clients which have initiated the I/O operations.

Our prototype runs on Sun SPARC and UltraSPARC workstations connected by the Myrinet [7] network. In order to ease development and take advantage of pre-existing user level software modules, the bulk of the client code executes at user level. We use a loadable kernel module to implement the Solaris kernel *vnode* interface and redirect I/O requests to the user level daemon. All network communications occur at user level, using low overhead Active Messages [51, 33].

Although the architecture is fundamentally sound and has demonstrated excellent scalability, the peer-to-peer serverless architecture entails many implementation challenges. For example, one I/O request by a user can potentially involve five different machines, demanding more formal approaches when reasoning about the correctness of interactions in such a system. The communication pattern among the components is richer than a simple client-server dialogue. At a lower level, concurrency management in the various user-level components and the kernel module can be challenging, and the demands placed on the vnode layer are non-traditional.

We will explore each of these areas in greater detail in the following sections. We start each section with the xFS features or problems that complicate the implementation, proceed to the conventional wisdom on how to approach these problems in traditional operating systems and explain why traditional methods are inadequate, and conclude each section with the approach we have taken.

3 The Use of Formal Verification in the Cache Coherence Engine

One of the most important features of xFS is its separation of data storage from data management. This separation requires a more sophisticated cache coherence protocol. In addition, other aspects of the cluster file system — such as multi-level storage and reliability constraints — further complicate the system compared to more traditional client-server and DSM coherence protocols. Due to these aspects of the design, we found it difficult to implement a correct protocol with traditional methods. Despite our initial skepticism of formal methods, we have found that the use of a formal protocol specification and verification tool has resulted in clearer abstraction levels, increased system confidence, and reduced complexity in the implementation of cache coherence in xFS. At the same time, there are significant differences between xFS and the original applications which the tool was designed to support. These differences have revealed some shortcomings of the tool.

3.1 Caching in xFS

xFS employs a directory-based invalidate cache coherence protocol. A client must obtain a read token in order to read a file block and must obtain a write token in order to overwrite or modify a block. The managers maintain the lists of current cachers of a block, and in response to client token requests they send invalidate messages or forwarding requests to other clients.

There are some important differences between the xFS protocol and previous cache coherence protocols. These differences are the result of balancing performance, functionality, and complexity. To better understand the design tradeoffs involved when we move from a traditional client-server architecture to a peer-to-peer architecture, consider the difference between the Sprite file system cache coherence protocol [35] and that of xFS. In Sprite, the server acts as both the token manager and the home of data and clients only exchange network messages with the server. Furthermore, consistency actions are only invoked at file open time, and clients are allowed to cache a file only when the file is not being concurrently write shared. Although these design decisions simplify the implementation, they contribute to limitations of the Sprite file system in terms of scalability, performance, and availability.

As we address these limitations in the cache coherence mechanism of the xFS peer-to-peer architecture, we encounter issues that are not present in the client-server model. First, xFS separates data management from data storage. Although this separation allows better locality and more flexible configuration, it splits atomic operations into different phases that are more prone to races and deadlocks. Second, more aggressive caching in xFS means that we cannot solve cache coherence problems by disabling caching and must maintain coherence at a finer grain than a per-file basis. Finally, client-to-client data transfers in xFS, while more efficient than passing all data through the server, introduce potential circular dependencies.

The cache coherence protocol in xFS is similar to those seen in hardware DSM systems such as DASH [32] and Alewife [30]. But even minor modifications to these protocols can lead to subtle bugs [9]. Also, aspects of the cluster file system require protocol modifications that do not apply to DSM systems. For example, xFS must maintain reliable data storage in the face of node failures. A client must therefore write its dirty data to storage servers before it can forward the data to another client. Another example of the differences between xFS and DSM systems is that xFS manages more storage levels. It must maintain the coherence of the kernel caches, write-ahead logs, and secondary storage. Recognizing the error-prone nature of distributed state machines, many hardware designers have adopted formal methods, but these methods have not been a common practice in the operating system community. We will next examine some of the difficulties involved in protocol development that necessitate more formal approaches.

3.2 Implementation Challenges

Cache coherence protocol designers often face issues of the proliferation of intermediate states and race conditions. These issues are more challenging than usual for xFS because of the nature of the underlying high speed switched network fabric and our peer-to-peer architecture.

3.2.1 Unexpected Messages and Network Reordering

In the course of executing the finite state machine representing the coherence protocol, an xFS node can receive messages that cannot be processed right away in its current state. For example, if a data block is in the process of being modified, then a read request for the block must wait until the modification finishes. Disabling message reception is not an option because the Active Message layer demands that the network must be constantly drained; otherwise deadlocks result in the network fabric. Although this is also a problem in some DSM coherence systems, it is particularly pervasive in xFS because xFS separates data storage and control and thereby makes it difficult to serialize data transfer messages and control messages with one another: data transfer messages pass between clients and storage servers or between clients and clients while control messages pass between clients and managers or storage servers and managers.

The possible solutions are queueing the unexpected message for later processing, sending a negative acknowledgement, or encoding the unexpected message more more protocol states. Each of these approaches has its own problems. Queueing can lead to deadlocks; negative acknowledgements can lead to deadlocks or livelocks; and encoding the message with states leads to proliferation of states and/or ad hoc modification of book-keeping data structures.

Out-of-order message delivery also complicates distributed protocols. In a switched network or a network that can lose and retransmit messages, messages sent by one host can be received out of order by other hosts. Furthermore, in order to deliver the maximum possible performance, the low level communication layers on these networks typically do not enforce message ordering, and many high performance file systems, in their eagerness to take advantage of the fast networks, have chosen to program directly on top of these communication layers. Compounding the problem, distributed operating system designers often allow multiple outstanding messages in the network to improve performance. For example, the Sprite file system used a customized fast kernel RPC which did not guarantee in order delivery. In an early version of that system, granting of a read token could be overtaken by a subsequent revoke of the same token and this led to a subtle race condition [36]. Similarly, the Active Message layer used by xFS does not enforce ordering between pairs of hosts, and the xFS protocol allows multiple outstanding requests. For example, when an xFS manager instructs a client to forward data to another client, the manager continues processing protocol requests immediately; thus a subsequent invalidate message can potentially reach the same client out of order. A formal protocol verifier can be particularly valuable in these situations where it can uncover subtle race conditions resulting from reordered messages, and furthermore, answer the more fundamental question of whether the amount of complexity in the protocol can justify the decision of using a communication layer that does not enforce message ordering.

3.2.2 Software Development Complexity

One engineering challenge we faced in building xFS was managing the large number of states needed to implement the xFS state machine. Although intuitively, each block can be in one of only four states – *Read Shared*, *Private Clean*, *Private Dirty*, or *Invalid* – the system must, in fact, use various transient states to mark progress during communication with the operating system and the network. The alternative, using blocking communication, can easily lead to deadlocks. This significantly increases the state space. Dealing with unexpected or out of order messages, handling the separation between data storage and data management, maintaining multiple levels of storage hierarchy, and ordering events to ensure reliable data storage all increase the number of transient states needed to handle xFS events. Even a simplified view of the xFS coherence engine contains twenty-two states. One needs a systematic approach when dealing with this large state space.

We initially tried to engineer the cache coherence engine with traditional methods. As we were implementing the cache protocol, it became clear that the C language was too general. Despite our best intentions, aspects of implementations that were not related to protocol specification were mixed in. The result was less modular, less general, harder to debug, and harder to maintain than the version that resulted from using a formal tool. Similarly, we have found that in CRL (a software DSM system), protocol specification is mixed with many low level details of communication. Although the xFS protocol is similar to many other DSM protocols (and we would have liked to have reused this earlier work), we have found it non-trivial to reuse or modify existing codes due to their ties to their native environments.

Testing also presents challenges. Due to the timing-dependent nature of many of the bugs and the complexity of the system, it was not practical to debug the system by running the xFS prototype on a collection of nodes and then testing all combinations of states and events. Furthermore, when timing-dependent errors do occur, it is extremely difficult to reconstruct the sequence of events leading to the error. The next subsection provides an example of such a case.

3.3 Implementing Cache Coherence with Formal Methods

After several unsuccessful attempts of completing the cache coherence protocol using traditional development methods, we decided to rewrite the system using Teapot [9], a tool for writing memory coherence protocols. Our experience with this more formal approach has been positive. In particular, the close ties between Teapot and the Murfi verification system have provided us with an effective testing tool for attacking the problem of unexpected event ordering; many of the bugs we found and corrected would have been difficult to isolate through field testing alone. Furthermore, several aspects of the Teapot language have simplified some of the engineering complexity in our system.

3.3.1 Teapot Overview

Teapot provides three main components: a protocol specifier, a protocol verifier, and an implementation generator. The protocol specifier provides a concise language that allows one to construct abstract state machines. A Teapot program consists of a set of states; each state specifies a set of message types and the actions to be taken on receipt of each message, should it arrive for a cache block in that state. The language also provides a mechanism that is similar to a "call-with-current-continuation" of functional programming languages. This mechanism is used to provide a blocking primitive inside a handler to relinquish the processor while preserving its context. The use of such continuations allows us to avoid having to manually decompose a handler into atomically executable pieces and then sequence them with state transitions. The protocol verifier is based on the Murfi system [13]. It systematically checks for protocol bugs such as invariant violations and deadlocks by performing an exhaustive state space exploration. Effectively, Murfi generates an exhaustive test vector for the distributed protocol; it reduces the length of this test vector by exploiting symmetries in the protocol. Finally, the implementation generator outputs C++ code.

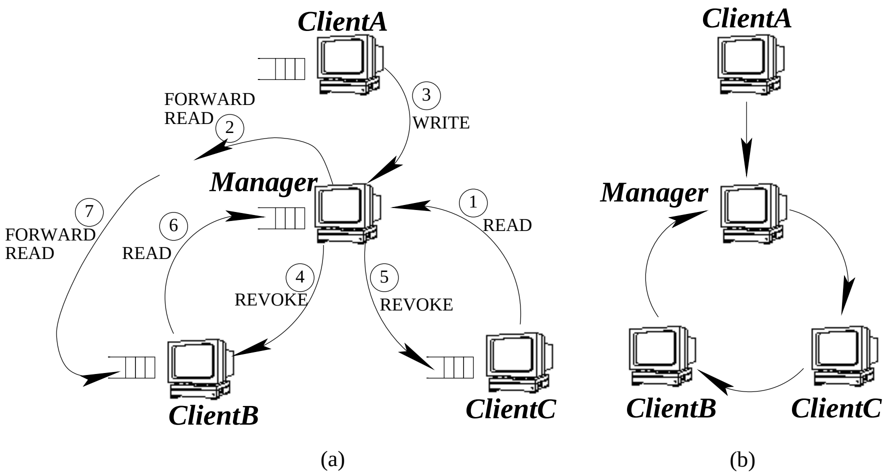


Figure 2: A sample deadlock discovered by the protocol verifier. The hosts labeled *ClientA*, *ClientB*, and *ClientC* are xFS clients and the host labeled *Manager* is an xFS metadata manager. In Figure (a), arrows denote the directions of the messages. The numbers denote the logical times at which messages are sent and/or received. Shown to the left of each host is a message queue, which holds the requests that are waiting to be processed. Messages that are not queued are processed immediately. In Figure (b), arrows denote the wait-for relationship, and the presence of a cycle indicates a deadlock.

3.3.2 Testing for Unexpected Event Orderings

Although arriving at the correct protocol remains an iterative process, the ability to compile a protocol specification to both verification code and implementation code dramatically shortens the turnaround time. The verification system can often provide the protocol designer with instant feedback that allows her to progressively refine the protocol. Most of the bugs discovered by the verifier are trivial ones. Because brute force testing is no longer in the critical path, these simple bugs can be quickly identified and fixed.

The tool also found more subtle timing bugs. Figure 2 shows an example of a bug in an early version of the xFS protocol that would have been difficult to isolate with field testing alone but which Murfi easily discovered. In this faulty version of the protocol, we saw no need for the manager to maintain sequence numbers for its outgoing messages. If a node received a request from a manager but was not ready to act upon it, the receiver simply queued it for later processing. Murfi found the following deadlock bug in this approach:

Initially, *ClientB* is the sole cacher of a clean block. 1) *ClientC* sends a read request to the *Manager*. 2) The *Manager* detects that *ClientB* has a cached copy of the file block; on a fast network, *ClientB*'s memory is faster to access than disk. Thus, the *Manager* forwards the request to *ClientB*. To indicate that *ClientB* should send the data directly to *ClientC*, the *Manager* also updates its state to indicate that both *ClientB* and *ClientC* are caching the data. 3) Meanwhile, *ClientA* sends a write request to the *Manager*. 4) The *Manager* sends a revoke request to *ClientB*, which arrives at *ClientB* before the previous forwarding message, invalidating its data. 5) The *Manager* sends a second revoke request to *ClientC*, which *ClientC* queues, because its requested data has not arrived. 6) *ClientB* sends a read request to the *Manager*, which the *Manager* queues, because its earlier revoke message has not been acknowledged. 7) The delayed message from step 2 finally arrives, which *ClientB* queues, because its earlier request to the *Manager* has not been satisfied. Now we have finally reached a deadlock: *ClientA* is waiting for the *Manager* to complete the revoke operations for its write; the *Manager* is waiting for *ClientC* to acknowledge the revoke request; *ClientC* is waiting for *ClientB* to supply the desired data; and *ClientB* is waiting for the *Manager* to process its read request. We solved this problem by using sequence numbers to order the outgoing messages from the manager, so the sequence of events seen by any client is consistent with the view of the manager. More importantly, the exercise with the protocol verifier has led us to conclude that the initial decision of using a communication layer that enforces no message ordering was unwise. Although in-order delivery can not eliminate all unexpected messages, it can simplify the protocol by eliminating a class of bugs resulting from reordered messages, without sacrificing significant performance.

3.3.3 Reduced Software Development Complexity

In addition to automatically uncovering protocol bugs, the use of Teapot also illustrates a number of other advantages of using formal methods. First, it demonstrates the importance of using a suitable notation. Teapot's continuations significantly reduce the number of states needed by the xFS protocol by combining each set of similar transient states into a single continuation state. Also, the language is more restrictive and the specifications are written in a fairly stylized way; making the specification easier to read and understand. Second, formal methods force one to concentrate on the problem at hand and separate it from other implementation details. In our case, the use of Teapot has resulted in modular and general-purpose code that is well isolated from the rest of the file system. Finally, formal methods encourage software reuse by isolating features that are common to the class of problems that they are designed to solve. In our case, we were able to inherit many support structures such as message queues and state tables from other protocols supplied with the Teapot release, further reducing complexity and the chance of errors.

3.4 Teapot Shortcomings

Teapot was designed and is best suited for DSM environments in which the primitives available to protocol handler writers are limited and simple. The xFS coherence engine, on the other hand, must interact with other components of the system such as the kernel and the Active Message subsystem using more powerful operations. Some examples are system calls and thread synchronizations. These primitives frequently lead to blocking operations on the local node and recursive invocations of the state machine. This difference in terms of power and expressiveness of the handler primitives have revealed some shortcomings of Teapot that have not become apparent in its original application domain.

The first shortcoming is the lack of support for multithreading. In order to support concurrent users and react to concurrent requests from the network, an xFS client is a heavily multithreaded system. The cache coherence engine generated by Teapot, unfortunately, has a large amount of global state that is difficult to make thread-safe. Section 4 discusses in detail the alternatives we have explored to enable the cache coherence engine to interact with other multithreaded components.

The second shortcoming concerns blocking operations on the local node, which occur frequently in xFS coherence handlers. For example, when an xFS client needs to invalidate a file data block it caches, it must make a system call to invalidate the data cached in the kernel. The complication arises when this system call might block, recursively waiting for some other event that requires the attention of the coherence engine. Although Teapot provides good support for blocking operations that wait for remote messages, using the same mechanism to handle local blocking operations is tedious. It also inflates the state space and prolongs the protocol verification process. In the above example, one must split the synchronous system call into asynchronous phases, invent a new node to represent the kernel, invent new states for the kernel node, invent new messages that the kernel must accept and generate, and write a number of handlers to tie all these elements together. Automating these mechanical chores and doing so efficiently can significantly ease the xFS protocol implementation.

The third shortcoming concerns some restrictions of Teapot's programming model. One example is Teapot's lack of support for operations that affect blocks other than the block on which the current message arrives. The problem arises, for example, when servicing the read fault of one block by an xFS client requires the eviction of a different block. Other implementation details such as the inability to add new arguments to handlers by a Teapot user also make the tool less flexible.

In this section, we have seen that the use of a formal protocol specifier has simplified the task of implementing the cache coherence engine. Unfortunately, the formal tool has been developed in isolation and does not compose well with the operating system and other complex subsystems. In the next section, we examine in greater detail the issue of composing the multithreaded subsystems.

4 Experience with Threads

Because of concurrent user, kernel, and network events, many xFS subsystems are multithreaded. Our first instinct was to compose these subsystems using threads. This appears natural because each thread simply makes procedure calls to enter other subsystems. Our experience indicates that composing complex multithreaded subsystems in this way leads to many concurrency control bugs, and that the "natural" ordering of events frequently leads to races and deadlocks.

Ideally, we would like to use formal methods (such as the ones reported in [44]) to verify the correctness of the synchronization behavior of our system. Unfortunately, the state of art of these technologies has not advanced sufficiently for them to be applied to the kernel, or to modules or libraries written in unsupported source languages (such as Teapot), or to subsystems for which we do not have source code. In this section, we discuss some practical engineering heuristics.

The key observation is that the xFS client software architecture is analogous to that of a network protocol stack [24] or a graphical user interface application [37]. Using events as the communication mechanism among different modules that are single-threaded event loops can eliminate the source of many synchronization bugs. As thread programming is entering the mainstream [18], and support for reusable software components matures [48], we believe our experience is applicable for a larger audience.

4.1 Client Software Architecture

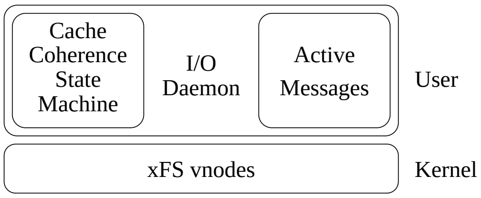


Figure 3: Software architecture of an xFS client.

Figure 3 shows the software architecture of an xFS client. The kernel vnode module manages a directory cache, interacts with the existing kernel file cache, and initiates upcalls to the user space for cache misses. The cache coherence state machine, as described in the previous section, reacts to events generated by the local system or the network and calls into the other components on state transitions.

The communication module implements the Active Message protocol. It accepts messages from the local system for asynchronous transmission and invokes message handlers on incoming messages. The I/O daemon coordinates all these components.

With the exception of the I/O daemon, the xFS development team has only limited influence over the internals of the modules. The kernel module must adhere to the conventions and interfaces available to the vnode layer. The cache coherence state machine is largely generated automatically by the protocol specification system. And the communication module is developed by yet another team. These modules should appear as black boxes to the xFS development team.

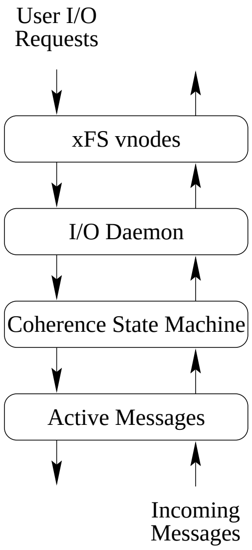


Figure 4: Opposite flow of events on an xFS client.

Events on an xFS client naturally flow in opposite directions through these black boxes, as shown in Figure 4. When a user initiates an I/O operation on the client, the request is redirected by the kernel module to the user space. The I/O daemon, which services the upcall, generates an event for the coherence state machine. During the state transition, the state machine prepares a message for the communication module, which eventually transmits the message. When an incoming message (not necessarily a reply) reaches the client, the exact opposite sequence of events occur. As we shall see in the next subsection, these opposite flow of events through software components, whose internals should not be exposed, can become a fertile ground for subtle synchronization bugs in a multithreaded system.

4.2 Difficulty with Concurrency Control

A seemingly natural way of structuring xFS is using threads. Unfortunately, the different sequence of events as discussed in the previous subsection, force the threads through different modules in different

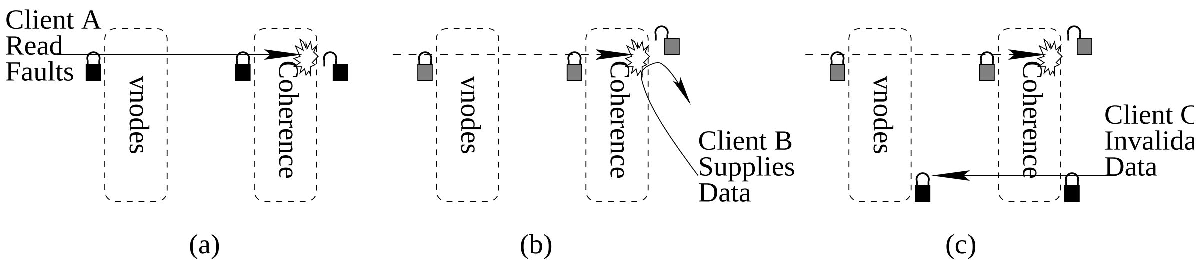


Figure 5: Deadlock of multiple threads when responding to local and remote events on a single client. In Figure (a), the thread that services client A's read fault holds a read lock in the vnode layer before sleeping in the cache coherence state machine monitor. In Figure (b), client B supplies the requested data and signals client A's sleeping thread. In Figure (c), client C steals the state machine monitor lock before client A reacquires it. It enters into a deadlock when this third thread attempts to acquire the vnode lock in order to invalidate the data.

orders. As we will see in a later example, this breaks abstractions and leads to deadlocks. The fact that some of these modules are not under our development control exacerbates the problem. Figure 5 shows one such case.

Figure 5 concerns two software modules, the kernel vnode module and the cache coherence state machine. The kernel vnode module protects the state of a data block with a lock. The cache coherence state machine is implemented as a monitor and is protected by a monitor lock. 1) Client A incurs a read miss. It marks the vnode state as read-in-progress, effectively holding a vnode read lock on the data block. Then the thread continues into the state machine monitor, releasing the state machine monitor lock prior to going to sleep. 2) Client B supplies the data and signals the sleeping thread. 3) Unfortunately, the waken thread does not reacquire the monitor lock immediately. Meanwhile, under the assumption that client A has acquired a read copy of the data, client C steals the monitor lock and proceeds to invalidate client A's data. The result is a deadlock. Client A is waiting to reacquire the monitor lock (held by client C) so that it can finish its read operation, while client C is waiting for client A to finish its read.

Deadlocks in a hierarchy of components that must support events flowing in opposite directions are by no means unique to xFS. In the JavaBeans Development Kit (BDK) [48], for example, events normally flow bottom-up towards the enclosing beans, while normal program flow travels top-down towards the enclosed beans. If the bean methods in question are protected by monitor locks, one runs the risk of deadlocks when multiple threads execute concurrently in opposite directions. The BDK, unfortunately, provides no satisfactory solution. (The BDK recommends not using locks at the risk of inconsistency in certain situations to avoid deadlocks.)

4.3 Use of Event Loops

The client architecture shown in Figure 4 is analogous to a network protocol stack in which each protocol component has a message interface that allows it to send packets to and receive packets from an adjacent protocol [24]. In one possible implementation strategy of this architecture, the absence of multiple threads executing concurrently in a protocol component eliminates the need for concurrency control. Similarly, in [37], Ousterhout observes that single-threaded event loops have many advantages over threads for most graphical user interface applications.

The same arguments can apply to several of the components in xFS. For example, instead of allowing multiple threads to execute concurrently in the cache coherence state machine, we simply enqueue a request event for the state machine. The state machine employs a single thread that continuously services its event queue in a loop. When the state machine requires service from other modules, it generates more events in a similar fashion.

The most important advantage of a single-threaded event loop over a multithreaded approach is that it eliminates unnecessary synchronizations that can lead to bugs. Consider the example in Figure 5. The two locks involved in the deadlock are actually quite different. The read lock in the vnode layer marks the file block as being read. It is meaningful to the file system implementor and she is responsible for reasoning about it. The monitor lock on the cache coherence state machine, on the other hand, has no meaning to the file system. It is an implementation artifact that is used to protect the global state of the state machine and the user of the state machine need not be aware of it. By turning the state machine from a multithreaded monitor into a single-threaded event loop that interacts with the other modules with asynchronous events, we eliminate the need for the monitor lock, and as a result, the deadlock illustrated in Figure 5 is no longer possible. It is important to point out that we are not declaring event loops to be a panacea for curing all synchronization woes. It is still possible for the user of a state machine to deadlock herself if she creates circular dependencies at the interface level.

Several improvements can make the event loop approach even more useful. In a pure event loop, since the event handler must run to completion, it is difficult to carry local state from the handling of one event to that of another. This, in turn, makes it difficult to simulate a procedure call from one module to another using events, since the request event and the reply event are decoupled, and the execution context of the request event is not available while handling the reply event. One solution to this problem is to use a continuation [14] which captures the execution context before sending the request event and restores the execution context when the reply event arrives. Another useful extension is to allow multiple threads to interact with an event loop. It is not possible or even desirable to eliminate all uses of multithreading. We have modified the continuation mechanism provided by Teapot so that when the event loop finishes servicing a request, instead of generating a reply event, it wakes up the blocked requesting thread.

A third improvement is increasing concurrency. A single-threaded event loop allows no CPU concurrency, so it cannot take advantage of multiple processors. There are well-known solutions that progressively incorporate more elements of multithreading to increase concurrency. A simple approach is to run different event loops on different processors. Another is to split a single event loop into multiple event loops that do not need to synchronize with each other. For example, one can split the coherence state machine into multiple event loops that manage disjoint portions of the file block space. These event loops can run on different processors without having to synchronize with each other. Even more concurrency can be achieved by allowing multiple threads that execute in the same event loop and may need to synchronize with each other. This is analogous to the thread-per-packet solution in a network protocol stack. One can use static checkers to prove the correctness of these local synchronizations [34].

To summarize, we have found that managing concurrency with threads has been a difficult problem in the xFS implementation, especially when threads enter off-the-shelf software modules over which we do not have full control of the implementation. The judicious use of single-threaded event loops, when applicable, can reduce the complexity. A different approach to the thread deadlock problem is to use wait-free synchronization [19]. While event loops reduce unnecessary concurrency, wait-free synchronization preserves concurrency and retries the operations if conflicts are detected. One problem with wait-free synchronization is that it is difficult to achieve good performance without operating system and hardware support.

5 Communication

The communication pattern in xFS represents a significant departure from a traditional client-server system. In this section, we first report our experience with traditional RPC. RPC, a popular communication abstraction for traditional distributed systems, has never been adopted by DSM builders. xFS's communication needs are more similar to those of DSM's than those of client-server systems. We shall see that RPC can neither satisfy the performance demands of the serverless architecture, which frequently use multiple small peer-to-peer control messages for each client request, nor does it match the requirements of xFS's communication pattern. To cope with these issues, we reimplemented our communications layer using Active Messages, a communication layer originally designed for supercomputing applications [11] but which has been enhanced to support cluster applications [33]. We found that Active Messages provided excellent performance, and by clearing away RPC's inappropriate abstractions it provided better semantics for supporting our communications patterns. Nonetheless, the Active Message interface exposes low-level details that we would prefer not to manage at the application level. At the end of this section, we discuss the aspects of RPC and Active Messages that might be combined to provide good support for high performance peer-to-peer distributed systems.

5.1 RPC

RPC communication [6] provides easy-to-understand semantics and has been the tool of choice for many distributed operating system builders. By providing a communication facility which is almost as easy to use as local procedure calls, the designers of RPC sought to remove the unnecessary difficulties of network communication so that system builders can concentrate on the higher level issues. They were largely successful in this goal for simple client-server systems.

However, traditional RPC provides neither the semantics nor the performance required by applications like xFS. First, the serverless architecture requires the use of many small control messages, whose efficient transmission is crucial to the performance of xFS. Traditional RPC implementations layered on top of heavy weight protocols (such as TCP and UDP) [26] cannot satisfy this performance requirement. Second, instead of using simple two-party request/reply exchanges, xFS transactions can require the cooperation of several machines. Unfortunately, when we synthesize multi-party communication using RPC, we cannot benefit from the semantic advantages that RPC was designed to provide.

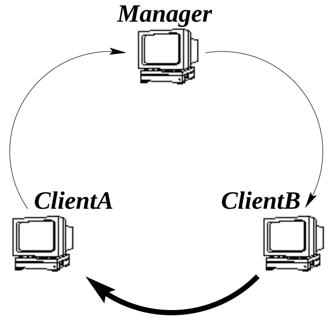
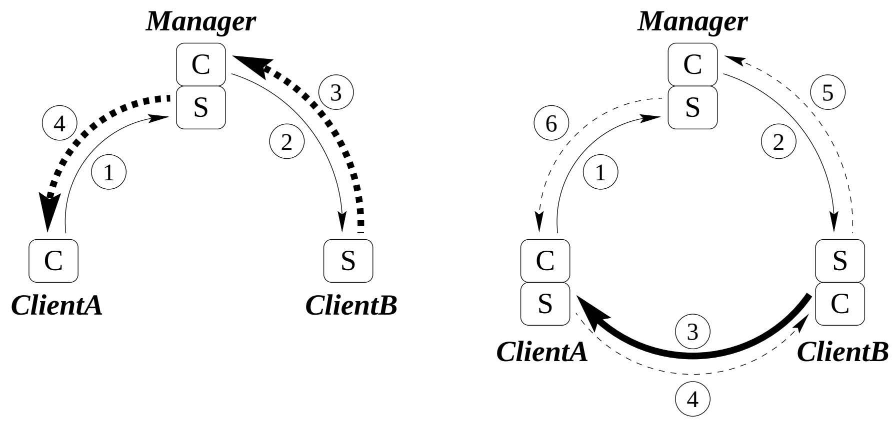


Figure 6: A multi-party communication example. The hosts labeled *ClientA* and *ClientB* are xFS clients and the host labeled *Manager* is an xFS metadata manager. This example illustrates a read miss on *ClientA*. Thin arrows indicate control messages and the thicker arrow represents a large data transfer.

Figure 6 shows a simple example of a multi-party communication in xFS. *ClientA* incurs a read



(a) (b)

Figure 7: Implementing multi-party communication using simple RPC. The component labeled "S" on a host is the RPC server and the component labeled "C" is the RPC client. The arrows denote the directions of communications. The numbers represent the logical times at which the communications occur. The solid lines are RPC requests and the dashed lines are RPC replies. The thicker arrow represents a large data transfer. Figure (a) shows the traditional client-server approach. The reply data retraces the steps of the RPC requests. Figure (b) shows the peer-to-peer approach. The reply data is sent from its source to the destination in one network hop.

miss and sends a request to the *Manager*. The *Manager* consults its bit vector of current cachers, and discovers that *ClientB* is caching the data. With a fast network and low overhead communication protocol, fetching the block from *ClientB*'s memory is faster than fetching it from disk, so the *Manager* sends a forwarding request to *ClientB*, and updates its bit vector to include *ClientA*. *ClientB* sends the data to *ClientA*, allowing *ClientA* to continue. In this example, the exchange of control messages between the clients and the manager must be efficient in order to realize the performance advantage of satisfying cache misses with remote memory. Also, each machine in this example must interact with *two* other machines in order to satisfy a single request. Such communication patterns are quite common in xFS. Another example occurs when a client writes in parallel to a number of storage servers in a stripe group.

Figure 7 shows two naive attempts to implement the example in Figure 6 using simple RPC. In Figure 7 (a), the reply data is forced to retrace the steps of the RPC requests. The obvious disadvantage is the extra cost of unnecessary data transfers between the clients and the manager. The approach shown in Figure 7 (b) attempts to address this inefficiency. In this case, because each host must initiate a request to and receive a request from two different hosts, each host is simultaneously an RPC client and an RPC server. *ClientA* first sends a request to the RPC server on the *Manager*. The *Manager* RPC server, acting as an RPC client, sends a new request to the RPC server on host B. The RPC server on *ClientB*, acting as an RPC client, supplies the requested data to the RPC server on *ClientA* via a third request. Only then does the RPC reply propagate back, retracing the steps taken by the three RPC requests.

When we used RPC layered on top of UDP for xFS communication in an early prototype, the first problem was performance. Even on fast switched networks, the overhead imposed by the layered protocols always reached milliseconds, defeating the benefit of using peer resources over the network. The contorted paths followed by some messages further hurt performance. The second problem was semantic inconvenience. The main benefit of RPC is that its semantics are close to those of a local procedure call. The semantics of a procedure call are that when the caller regains control, the desired operation has been performed exactly once and the caller has been given the call reply. In Figure 7 (b), however, the programmer is responsible for explicitly matching the data with the original request. Furthermore, this approach is also inefficient. The original requester has to wait for six network hops to receive its RPC reply! Depending on the characteristics of the network, this might take longer than the data transfer. To alleviate this inefficiency, one might be tempted to make the RPC servers reply sooner so that they can overlap the replies with other operations, thus further weakening the RPC semantics and complicating the job of the programmer.

5.2 Active Messages

To satisfy the performance and semantic requirements of the serverless architecture, we switched to Active Messages as the communication subsystem of xFS. We found that Active Messages provide excellent performance, reducing communication latency to tens of microseconds. The requirements of xFS also led to numerous modifications to Active Messages so that it can better support cluster computing. However, the Active Message interface was designed to be an "assembly language" for communications library writers rather than an interface to be used directly by applications. Indeed, we found that Active Message exposed several low-level details that complicated the programming model.

Each Active Message specifies a handler on the remote node and a set of arguments to the handler. Messages are delivered reliably (barring persistent failure) but potentially out-of-order (to reduce buffer management overhead). When a message arrives, the handler is invoked directly, and it runs to completion. This direct invocation is the source of much of Active Message's efficiency, but it also places restrictions on the programming model. Because handlers run at the priority of the network and prevent delivery of other messages while they run, they must execute quickly and without blocking.

To simplify deadlock-free management of switched networks, Active Message implementations typically require request-reply models of communication in which request handlers must use the network to send exactly one reply message, and reply handlers may not use the network.

The original Active Message system was suitable only for parallel applications, not for distributed operating systems. The requirements of xFS and other similar softwares being written as part of the Berkeley NOW project [2] led to many improvements to the Active Message interface to provide better support for cluster computing. Since many of these new features may not be known to readers familiar with the original Active Message system, and since these features are likely to be of use for distributed operating systems applications that try to make use of supercomputer-class networks and high-performance protocols originally designed for supercomputer applications, we briefly discuss some of the key new Active Message features here. More details can be found in [33].

- Naming. The original Active Message interface assumes a single program image (SPMD) model, where the communicating parties are simply referenced by contiguous node numbers, and the message handlers are referenced by absolute program counter addresses. In xFS, the different services, each of which can start and stop dynamically, no longer share a single program image, and nodes can be added or deleted dynamically. The new Active Message interface removes all the SPMD restrictions.

- Multiprogramming. The original Active Message interface assumes that the SPMD instances of the parallel applications are gang-scheduled on different nodes. To wait for an incoming messages, the program simply polls the network interface in a busy loop. Also, the interface provides no thread support. In a distributed environment such as xFS, there can be multiple processes (some of whom are not necessarily part of xFS) time-sharing a node and the network. The new interface accommodates this difference by providing *virtual networks* to communicating parties. When there are other processes running on the host where an xFS server executes, polling the network for incoming messages to the xFS server becomes inefficient. To address this difference, the new interface provides an event model that wakes up a server upon receipt of certain incoming messages. Furthermore, the new interface is also thread-safe to accommodate multithreaded xFS modules.

- Security. The original interface assumes that the communicating parties have exclusive use of the network. Each instance of the program can communicate with any other instances; therefore there are no protection issues involved. This is not acceptable in the multiprogramming environment where xFS operates. The new interface uses capabilities to authenticate the communicating parties.

- Fault Tolerance. The original Active Messages provide reliable message delivery. Communication errors can only be attributed to catastrophic failures in the supercomputer network fabric and this results in the termination of all instances of the SPMD program. A feature of xFS is high availability — the ability to survive individual node and network link failures. The new interface provides synchronous and asynchronous error detection for persistent network failures and the user of the interface can react accordingly to the error events.

- Medium-sized Messages. The original interface only supports two flavors of message: a "small message" interface that is designed to pass procedure call arguments in registers, and a "bulk message" interface for large transfers. These message sizes were devised to support the typical procedure calls and large data transfers found in parallel programs. Communication in distributed operating systems also consists of control messages and data transfers, but the control messages tend to be larger than the "small messages" provided by the original Active Messages while still being too small to efficiently take advantage of the bulk message interface. In xFS, for instance, most control messages are between 32 and 128 bytes. As a result, the new interface provides a "medium message" interface that provides automatic storage management without the additional network round trip that would be needed for a "bulk transfer."

- UDP Reference Implementation. The original Active Message implementations were custom built for each supercomputer architecture, which limited portability. A reference implementation of the new interface has been implemented on UDP. This reference implementation makes it palatable to use Active Messages as xFS's only message substrate because it allows our system to run on any network that supports UDP. Of course, to achieve good performance we still require a customized native implementation of Active Messages for each high speed network.

Even with the new features, Active Messages are restrictive, but we found that it was sufficient for our needs and, in fact, a better substrate on which to program than traditional RPC. There were four reasons for this.

- Some aspects of Active Message mapped easily onto some aspects of our system. For example, the asynchronous messaging model was a good fit with Teapot's protocol specification tool.

- Although neither RPC nor the Active Message interface was a particularly good match with multi-party communications patterns, at least Active Message's asynchronous message arrival abstraction was less misleading than the synchronous procedure call abstraction in RPC.

- Active Messages' speed advantage over RPC gives designers the freedom to focus on building a simple, correct protocol. If sending an extra message simplifies the protocol or makes it more robust, that might be a reasonable trade-off under Active Messages, while it would seldom be acceptable under RPC. Similarly, as the multi-party RPC example above illustrated, when using RPC we found ourselves introducing asynchrony to improve performance at the cost of complicating the protocol.

Even with these enhancements, Active Messages remain, by design, a low-level programming model. Some of its limitations are:

- Active Messages only provide an asynchronous point-to-point request-reply communication model.

- Active Messages place severe restrictions on the types of operations that are allowed in message handlers.

- Active Messages place restrictions on the size and contiguity of the data to be transferred.

- Active Messages may arrive out-of-order.

In the next subsection, we will discuss some of the desirable characteristics of a higher level abstraction.

5.3 Communications Support for New Operating Systems Services

Although we were successful in using Active Messages as our communications interface, we are not advocating using low-level message passing to build distributed systems. Work is needed on a new abstraction that supports this new generation of applications. We believe that this protocol should have the following characteristics:

- The protocol should retain the RPC client's "procedure call" abstraction. The original motivation for using RPC to structure distributed systems remains: procedure calls are a well-understood and useful programming abstraction.

- To support multi-party communication, the protocol must support a mechanism similar to continuation-passing that allows an RPC service routine to delegate the duty to respond to an RPC request to another RPC service routine on another node. Another useful extension is the support for scatter to and gather from multiple machines.

- Asynchronous messages between pairs of hosts should be delivered in the order they are sent.

- For performance, the protocol should be built directly on a high-performance low-level message abstraction like Active Messages. Empirical evidence suggests that it is difficult to make layered protocols fast. On the other hand, Active Messages have proven to be an excellent "assembly language" for constructing high performance network abstractions including Split-C [12], Thinking Machines message passing library [49], MPI [16], and Fast Sockets [41]. The Fast Sockets example is particularly relevant since it demonstrated a high-performance implementation of the Unix socket interface that, like RPC, has been widely used by distributed operating systems.

In the process of synthesizing this higher level communication abstraction using Active Messages, one must pay close attention to performance optimizations that traditional networking protocols do not address. For example, when gathering data from different senders, it is important to coordinate the communicating parties to prevent overrunning the bottleneck machine [8]. In xFS, we have seen the importance of this "rate-matching" technique when a client is assembling multiple data fragments from different storage servers. Another example is the choice of the different interfaces for different message sizes. The current Active Message interface provides three flavors of message interfaces. Again, the communication abstraction implementor's task here is analogous to that of a compiler writer who has a choice of different (combinations of) assembly language instructions. She must choose the appropriate interface to maximize performance.

In summary, we have seen that the communication needs of the new peer-to-peer distributed systems are significantly different from both the traditional client-server applications and the traditional super-computing applications. One must combine the best of these two worlds to arrive at a new interface to simplify the construction of these kinds of systems in the future.

6 Kernel Support

As described previously, one component of an xFS client is a kernel module that implements the kernel vnode interface for a new file system. We chose the vnode interface because this was the most straightforward way to make a new file system available to unmodified applications. A much earlier version of xFS was developed on Ultrix 4.2. The current version of xFS runs on Solaris 2.3-2.5. Although the Solaris interface has evolved considerably from its predecessors and is far more complex, both Ultrix and Solaris (and most other Unix operating systems) share an ancestral design that is not suited for today's high-performance distributed file systems. Some of the main issues are coherence, efficiency, and portability. Some of these problems have been noted in previous efforts to extend the vnode interface [40, 42, 46, 28, 21]. Unfortunately, few improvements have been made to the commercial operating systems; and the performance and interface problems are exacerbated by new serverless distributed systems.

6.1 Cache Coherence

The vnode interface was originally designed to integrate NFS into Unix based operating systems that only supported local file systems [29]. The local file systems obviously need not consider coherence, and NFS [43] only provides an ad hoc style of weak consistency. Some later research file systems, such as Andrew [23], had more sophisticated consistency semantics but consistency actions were only triggered on file open events. Once a file is open, its content and attributes cannot be affected by remote machines. This simplifies vnode coherence management. Other systems such as Sprite [35] and Spritely NFS [47] disable caching and fall back to a central server when a file is opened for concurrent writing sharing. This also simplifies coherence management. Under these restrictions, it is still relatively easy to support these file systems within the vnode framework.

xFS must maintain coherence of file attributes, directory contents, and file data at block granularity, without sacrificing performance, and without resorting to a central server. The vnode interface starts to show strain under these requirements. The biggest problem is maintaining consistency of the attributes. In the vnode layers that we have studied, attributes are cached in the vnodes themselves and are freely accessed by the kernel code; as a result, they cannot be easily invalidated. In the past, systems such as Spritely NFS have approached this problem with an ad hoc combination of performing attribute modifications on the server and periodic polling on the client for updates. xFS has no centralized attribute storage and it demands a more strict attribute coherence semantics; maintaining attribute coherence required extensive kernel modifications in Solaris. Another difficulty is maintaining file data coherence. In the past, file servers, such as those under Andrew, need not be informed when client memory cache evictions occur. This is because even when the data is no longer cached by the kernel memory, the copy cached on the local disk still needs to be invalidated. Also, none of these systems requires injection of data that is not demanded by a client into its cache. The xFS cooperative caching algorithm, on the other hand, requires the manager to have precise knowledge of the client cache contents so it can intelligently use the global cluster memory cache to satisfy file system requests. An xFS client also needs to have the ability of injecting an evicted data block into a peer client's cache as its backing store. Therefore a vnode interface that informs the managers of any changes in cache contents and allows easy manipulations by other hosts would be useful. Although it is possible to synthesize such an interface using the kernel cache interface in the operating systems we have studied, explicit support of these operations in the vnode interface would provide a cleaner abstraction.

6.2 Implementation Overhead

The vnode interface was designed at a time when file system performance was relatively poor. A kernel cache miss in those days meant disk accesses, slow network accesses, or both. Consequently, not much effort was devoted to bringing down the implementation overhead on cache misses. For example, although Andrew has a client architecture that is very similar to that of xFS, the interface overhead of the kernel is less of an issue because a kernel cache miss results in at least one access to the local disk cache. In xFS, on the other hand, a cache miss can usually be handled by a much faster transfer from another client's memory.

With the advent of high-performance network file systems such as xFS, file system performance can approach network speeds, and as network performance continues to improve, the implementation overhead of the vnode layer is becoming one of the limiting factors in improving file system performance. For example, the vnode layer under Solaris 2.5 on an UltraSPARC 170 imposes a minimum of 130 μs of overhead for a small read operation on a local node, while a user-to-user remote memory operation can complete a round trip with only 20 μs. A closer examination revealed that the bulk of the overhead can be attributed to the inefficiency in the virtual memory system. Similar problems were noted in the OSF/1 kernel, whose high kernel miss overhead impacted the design of GMS [15].

6.3 Portability

The vnode layers of different operating systems are considerably different [52]. One needs arcane knowledge of the kernel internals in order to port a file system to a different vnode layer. This is one of the major obstacles to deploying xFS on a variety of different platforms. For a cluster file system such as xFS, a portable operating system interface not only would allow us to more easily take full advantage of the resources in a heterogeneous cluster, but it would also reduce the system complexity.

To this end, the vnode interfaces have failed miserably. Despite many years of research in extensible file systems [39, 5, 40, 38, 42, 22, 1, 28, 52], the construction of portable file systems has remained a difficult undertaking, especially for the commercial operating systems. We are currently investigating portable and efficient interposition agents as a means of distributing our code [17].

In this section, we have seen that the vnode layer has achieved the goal of making xFS available to unmodified applications, but at a considerable cost. Some of the disadvantages are the complexity due to the lack of coherence support, interface overhead on cache misses, and poor portability. As high-performance distributed file systems become more popular, the kernel interface must evolve to cater to their needs.

7 Conclusion

Emerging high performance networks and workstations can not only deliver performance competitive with traditional supercomputers on highly parallel scientific applications, it has also enabled the development of a new generation of distributed systems that offer superior performance, availability, and scalability. We have reported some of the lessons that we have learned during the implementation of xFS, an example of a new peer-to-peer paradigm in distributed systems. We have seen a mismatch between this new paradigm and the traditional tools and interfaces available to distributed system builders. To address some of these inadequacies, we have successfully adopted a number of technologies including formal methods, event loops, fast messaging layers, and new kernel interfaces. The use of these technologies, however, has not been a straightforward transplantation. We believe our experience is not only applicable to xFS, but also provides insights for future system builders and encourage the development of new tools and interfaces that can make the construction of such systems easy.

References

[1] Abrosimov, V., Armand, F., and Ortega, M. I. A Distributed Consistency Server for the CHORUS System. In *Proc. of the 3rd USENIX Symposium on Experiences with Distributed and Multiprocessor Systems* (March 1992).

[2] Anderson, T., Culler, D., Patterson, D., and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro* (Feb. 1995), 54–64.

[3] Anderson, T., Dahlin, M., Neefe, J., Patterson, D., Roselli, D., and Wang, R. Serverless Network File Systems. *ACM Transactions on Computer Systems 14*, 1 (Feb. 1996), 41–79.

[4] Anderson, T., Owicki, S., Saxe, J., and Thacker, C. High-Speed Switch Scheduling for Local-Area Networks. *ACM Transactions on Computer Systems 11*, 4 (Nov. 1993), 319–52.

[5] Bershad, B. N., and Pinkerton, C. B. Watchdogs: Extending the UNIX File System. In *Proc. of the 1988 Winter USENIX* (Februrary 1988).

[6] Birrell, A. D., and Nelson, B. J. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39–59.

[7] Boden, N., Cohen, D., Felderman, R., Kulawik, A., Seitz, C., Seizovic, J., and Su, W. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE MICRO* (Feb. 1995), 29–36.

[8] Brewer, E., and Kuszmaul, B. How to Get Good Performance from the CM5 Data Network. In *Proc. of the 1994 International Parallel Processing Symposium* (April 1994).

[9] Chandra, S., Richards, B., and Larus, J. R. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proc. of SIGPLAN Conference on Programming Language Design and Implementation* (May 1996).

[10] Corbett, P., Baylor, S., and Feitelson, D. Overview of the Vesta Parallel File System. *Computer Architecture News 21*, 5 (Dec. 1993), 7–14.

[11] Culler, D., Keeton, K., Liu, L., Mainwaring, A., Martin, R., Rodrigues, S., Wright, K., and Yoshikawa, C. The Generic Active Message Interface Specification. http://now.cs.berkeley.edu/Papers/Papers/gam_spec.ps, 1994.

[12] Culler, D. E., Dusseau, A., Goldstein, S. C., Krishnamurthy, A., Lumetta, S., von Eicken, T., and Yelick, K. Parallel Programming in Split-C. In *Proc. of Supercomputing '93* (November 1993).

[13] Dill, D. L., Drexler, A. J., Hu, A. J., and Yang, C. H. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors* (1992), pp. 522–525.

[14] Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proc. of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 122–136.

[15] Feeley, M. J., Morgan, W. E., Pighin, F. P., Karlin, A. R., Levy, H. M., and Thekkath, C. A. Implementing Global Memory Management in a Workstation Cluster. In *Proc. of the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 201–212.

[16] Forum, M. P. I. MPI: A Message-Passing Interface Standard. Tech. Rep. CS-94-320, University of Tennessee Computer Science Department, May 1994.

[17] Ghormley, D. P., Petrou, D., and Anderson, T. E. SLIC: Secure Loadable Interposition Code. Tech. Rep. UCB/CSD 96/920, University of California at Berkeley, November 1996.

[18] Gosling, J., and McGilton, H. The Java(tm) Language Environment: A White Paper. http://-java.dimensionx.com/whitePaper/java-whitepaper-1.html, 1995.

[19] Greenwald, M., and Cheriton, D. The Synergy Between Non-blocking Synchronization and Operating System Structure. In *Proc. of the Second Symposium on Operating Systems Design and Implementation* (October 1996), pp. 123–136.

[20] Hartman, J., and Ousterhout, J. The Zebra Striped Network File System. *ACM Transactions on Computer Systems* (Aug. 1995).

[21] Heidemann, J., and Popek, G. File-system Development with Stackable Layers. *ACM Transactions on Computer Systems 12*, 1 (Feb. 1994), 58–89.

[22] Heidemann, J. S., and Popek, G. J. A Layered Approached to File System Development. Tech. Rep. CSD-91007, UCLA Computer Science Department, March 1991.

[23] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

[24] Hutchinson, N. C., and Peterson, L. L. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering 17*, 1 (January 1991).

[25] Johnson, K. L., Kaashoek, M. F., and Wallach, D. A. CRL: High Performance All-Software Distributed Shared Memory. In *Proc. of the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 213–228.

[26] Keeton, K. K., Anderson, T. E., and Patterson, D. A. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Proc. of 1995 Hot Interconnects III* (August 1995).

[27] Keleher, P., Cox, A. L., Dwarkadas, S., and Zwaenepoel, W. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the 1994 Winter Usenix Conference* (January 1994), pp. 115–132.

[28] Khalidi, Y. A., and Nelson, M. N. Extensible File Systems in Spring. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (December 1993), pp. 1–14.

[29] Kleiman, S. R. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proc. of the 1986 Summer Usenix Conference* (June 1986), pp. 238–247.

[30] Kubiatowicz, J., and Agarwal, A. Anatomy of a Message in the Alewife Multiprocessor. In *Proc. of the 7th International Conf. on Supercomputing* (July 1993).

[31] Lee, E. K., and Thekkath, C. E. Petal: Distributed Virtual Disks. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996), pp. 84–92.

[32] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th International Symposium on Computer Architecture* (May 1990), pp. 148–159.

[33] Mainwaring, A., and Culler, D. Active Message Application Programming Interface and Communication Subsystem Organization. Tech. Rep. UCB/CSD 96/918, University of California, Berkeley, October 1996.

[34] Nelson, G., Leino, K., Saxe, J., and Stata, R. Extended Static Checker Home Page. http://www.research.digital.com/SRC/esc/Esc.html, 1996.

[35] Nelson, M., Welch, B., and Ousterhout, J. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988).

[36] Ousterhout, J. K. The Role of Distributed State. In *CMU Computer Science: a 25th Anniversary Commemorative* (1991), R. F. Rashid, Ed., Addison-Wesley, pp. 199–217.

[37] Ousterhout, J. K. Why Threads Are a Bad Idea. http://www.sunlabs.com/~ouster/, 1995.

[38] Peterson, L., Hutchinson, N., O'Malley, S., and Rao, H. The x-kernel: A Platform for Accessing Internet Resources. *IEEE Computer 23*, 5 (September 1990), 23–33.

[39] Rees, J., Levine, P. H., Mishkin, N., and Leach, P. J. An Extensible I/O System. In *Proc. of the 1986 Summer USENIX* (June 1986).

[40] Richard G. Guy, e. a. Implementation of the Ficus Replicated File System. In *Proc. of the 1990 Summer USENIX* (June 1990).

[41] Rodrigues, S. H., Anderson, T. E., and Culler, D. E. High-Performance Local-Area Communication Using Fast Sockets. In *Proc. of the Winter 1997 USENIX* (Jan. 1997).

[42] Rosenthal, D. S. H. Evolving the Vnode Interface. In *Proc. of the 1990 Summer USENIX* (June 1990).

[43] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and Implementation of the Sun Network Filesystem. In *Proc. of the Summer 1985 USENIX* (June 1985), pp. 119–130.

[44] Savage, S., Anderson, T. E., Burrows, M., and Nelson, G. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *Proceedings of the ACM Sixteenth Symposium on Operating Systems Principles* (Oct. 1997).

[45] Scales, D. J., and Lam, M. S. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proc. of the First Symposium on Operating Systems Design and Implementation* (November 1994), pp. 101–114.

[46] Skinner, G. C., and Wong, T. K. Stacking Vnodes: A Progress Report. In *Proc. of the 1993 Summer USENIX* (June 1993).

[47] Srinivasan, V., and Mogul, J. Spritely NFS: Experiments with Cache Consistency Protocols. In *Proc. of the 12th Symposium on Operating Systems Principles* (Dec. 1989), pp. 45–57.

[48] Sun Microsystems. *JavaBeans API Specification*, 1996. http://www.javasoft.com/beans/.

[49] Tucker, L., and Mainwaring, A. CMMD: Active Messages on the CM-5. *Parallel Computing 20*, 4 (August 1994), 481–496.

[50] von Eicken, T., Basu, A., Buch, V., and Vogels, W. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 40–53.

[51] von Eicken, T., Culler, D., Goldstein, S., and Schauser, K. E. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)* (May 1992), pp. 256–266.

[52] Webber, N. Operating System Support for Portable Filesystem Extensions. In *Proceedings of the 1993 USENIX Winter Conference* (January 1993), pp. 219–228.

[53] Woodruff, A., Aoki, P. M., Brewer, E., Gauthier, P., and Rowe, L. A. An Investigation of Documents from the World Wide Web. http://www.bmrc.berkeley.edu/papers/inktomi/, 1996.