The Design of a Multicast-based Distributed File System

Björn Grönvall, Assar Westerlund, and Stephen Pink
Swedish Institute of Computer Science and *Luleå University of Technology*
{bg, assar, steve}@sics.se

Abstract

JetFile is a distributed file system designed to support shared file access in a heterogenous environment such as the Internet. It uses multicast communication and optimistic strategies for synchronization and distribution.

JetFile relies on "peer-to-peer" communication over multicast channels. Most of the traditional file server responsibilities have been decentralized. In particular, the more heavyweight operations such as serving file data and attributes are, in our system, the responsibility of the clients. Some functions such as serializing file updates are still centralized in JetFile. Since serialization is a relatively lightweight operation in our system, serialization is expected to have only minor impact on scalability.

We have implemented parts of the JetFile design and have measured its performance over a local-area network and an emulated wide-area network. Our measurements indicate that, using a standard benchmark, JetFile performance is comparable to that of local-disk based file systems. This means it is considerably faster than commonly used distributed file systems such as NFS and AFS.

1 Introduction

JetFile [15] is a distributed file system designed for a heterogenous environment such as the Internet. A goal of the system is to provide ubiquitous distributed file access and centralized backup functions without incurring significant performance penalties.

JetFile should be viewed as an alternative to local file systems, one that also provides for distributed access. It is assumed that, if a user trusts the local disk to be sufficiently available to provide file access, then JetFile should be available enough too. Ideally a user should have no incentive to put her files on a local file system rather than on JetFile.

JetFile is targeting for the demands of personal computing. It is designed to efficiently handle the daily tasks of an "ordinary" user such as mail processing, document preparation, information retrieval, and programming.

Increasing read and write throughput beyond that of the local disk has not been a goal. We believe that most users find the performance of local-disk based file systems satisfactory. Our measurements will show that it is possible to build a distributed file system with performance characteristics similar to that of local file systems.

A paper by Wang and Anderson [37] identifies a number of challenges that a geographically dispersed file system faces. We reformulate these slightly differently as:

Availability: Communication failures can lead to a file not being available for reading or writing. As a system grows, the likelihood of communication failures between hosts increases.

Latency: Latencies are introduced by propagation delays, bandwidth limitations, and packet loss. The special case, network disconnection, can be regarded as either infinite propagation delay or guaranteed packet loss.

Bandwidth: To reduce cost, it is in general desirable to keep communication to a minimum. Replacing backbone traffic with local traffic is also desirable because of its lower pricing.

Scalability: Server processing and state should grow slowly with the total number of hosts, files, and bytes.

JetFile is designed to use four distinct but complementary mechanisms to address these problems.

Optimistic algorithms are used to increase update availability and to reduce update latency.

Hoarding and prefetching will be used to increase read availability and to reduce read latency.

Replication and multicast are used to increase read availability, reduce read latency, reduce backbone traffic, and to improve scalability.

Clients act as servers to decrease update latency, minimize traffic, and to improve scalability.

Latencies introduced by the network can often be large, especially over satellite or wireless networks. This is an artifact of the limited speed of light and/or that packets must be repeatedly retransmitted before they reach their destination.

To hide the effects of network latencies, JetFile takes an optimistic approach to file updates. JetFile promises to detect and report update conflicts, but only after the fact that they have occurred.

The optimistic approach assumes that write sharing will be rare and uses this fact to hide network latencies. Experience from Coda [23, 19] and Ficus [29] tell us that in their respective environments write sharing and update conflicts were rare and could usually be handled automatically and transparently.

JetFile is designed to support large caches (on the order of gigabytes) to improve availability and to avoid the effects of transmission delays.

JetFile takes a "best-effort" approach with worst-case guarantees to maintain cache coherency. Coherency is maintained through a *lease* [14] or *callback* [17]-style mechanism[1]. The *callback* mechanism is best-effort in the sense that there is a high probability, but no absolute guarantee, that a *callback* reaches all destinations. If there are nodes that did not receive a *callback* message, the amount of time they will continue to access stale data is limited by a worst-case bound. In the worst case, when packet loss is high, JetFile provides consistency at about the same level as NFS [30]. Under more normal network conditions, with low packet drop rates, cache consistency in JetFile is much stronger and closer to that of the Andrew File System [17].

Applications that require consistency stronger than what JetFile guarantee may experience problems. This is an effect of the optimistic and best-effort algorithms that JetFile uses. Only under "good" network conditions will JetFile provide consistency stronger than NFS. To exemplify this, if a distributed make fails when run over NFS, it will sometimes also fail when run over JetFile.

There is a scalability problem with conventional *callback* and *lease* schemes: servers are required to track the identity of caching hosts. If the number of clients is large, then considerable amounts of memory are consumed at the server. With our multicast approach, servers need not keep individual client state since callbacks find their way to the clients using multicast. Rather than centralizing/concentrating the *callback* state at one server, the state has been distributed over a number of multicast routers. As an additional benefit, only one packet has to be sent when issuing the *callback*, not one for each host.

Traditional distributed file systems are often based on a centralized server design and unicast communication. JetFile instead relies on peer-to-peer communication over multicast channels. Most of the traditional file server responsibilities have been decentralized. In particular, the more heavyweight operations such as serving file data and attributes are, in our system, the responsibility of the clients. Some functions such as serializing file updates are still centralized in JetFile. Since serialization is a relatively lightweight operation in our system, serialization is expected to only have a minor impact on scalability.

Scalability is achieved by turning every client into a server for the files accessed. As a result of clients taking over server responsibilities, there is no need to immediately *write-through*[2] data to some server after a file update.

Shared files such as system binaries, news, and web pages are automatically replicated where they are accessed. Replication is used as a means to localize traffic, distribute load, decrease network round-trip delays, and increase availability. Replicated files are retrieved from a nearby location when possible.

Files that are shared, will, after an update, be distributed directly to the replication or *via* other replication sites rather than being transferred *via* some other server.

For availability and backup reasons, updated files will also be replicated on a storage server a few hours after update or when the user "logs off." The storage server thus acts as an auxiliary site for the file.

JetFile is designed to reduce the amount of network traffic to what is necessary to service compulsory cache misses and maintain cache coherency. Avoiding network communication is often the most efficient way to hide the effects of propagation delays, bandwidth limitations, and transmission errors.

We have built a JetFile prototype that includes most of JetFile's key features. The prototype does not yet have a storage server nor does it include any security related features. However, the prototype is operational enough for preliminary measurements. We have made measurements that will show JetFile's performance to be similar to local-disk based file systems.

The rest of this paper is organized as follows:

The paper starts with a JetFile specific tutorial on IP multicast and reliable multicast. The tutorial is followed by a system overview. The sections to follow are: File Versioning, Current Table, JetFile Protocol, Implementation, Measurements, Related Work, Future Work, Open Issues and Limitations, and Conclusions.

[1] A *callback* is a notification sent to inform that a cached item is no longer valid.

[2] The file is still written to local disk with the *sync* policy of the local file system.

2 Multicast

Traditionally multicast communication has been used to transmit data, often stream-oriented such as video and audio, to a number of receivers. Multicast is however not restricted to these types of applications. The inherent location-transparency of multicast also makes its use attractive for peer to multi-peer communication and resource location and retrieval. With multicast communication it is possible to implement distributed systems without any explicit need to know the precise location of data. Instead, peers find each other by communicating over agreed upon communication channels. To find a particular data item, it is sufficient to make a request for the data on the agreed upon multicast channel and any node that holds a replica of the data item may respond to the request. This property makes multicast communication an excellent choice for building a system that replicates data.

Multicast communication can also be used to save bandwidth when several hosts are interested in the same data by "snooping" the data as it passes by. For instance after a shared file is updated and subsequently requested by some host, it is possible for other hosts to "snoop" the file data as it is transferred over the network.

2.1 IP Multicast

In IP multicast [9, 8] there are 2^{28} (2^{112} in IPv6) distinct multicast channels. Channels are named with IP addresses from a subset of the IP address space. In this paper, we will interchangeably use the terms multicast address, multicast channel, and multicast group.

To multicast a packet, the sender uses the name of the multicast channel as the IP destination address. The sender only sends the packet once, even when there are thousands of receivers. The sender needs no knowledge of receiver-group membership to be able to send a packet.

Multicast routers forward packets along distribution trees, replicating packets as trees branch. Like unicast packets, multicast packets are only delivered on a best-effort basis. I.e., packets will sometimes be delivered in a different order than they were sent, at other times; they may not be delivered at all.

Multicast routing protocols [11, 3, 10, 24] are used to establish distribution trees that only lead to networks with receivers. Hosts signal their interest in a particular multicast channel by sending an IGMP [12] *membership report* to their local router. This operation will graft the host's local network onto, or prevent the host's local network being pruned from, the multicast distribution tree.

The establishment of distribution trees is highly dependent on the multicast routing protocol in use. Multicast routing protocols can roughly be divided into two classes. Those that create source specific trees and those that create shared trees. Furthermore, shared trees can be either uni- or bi-directional.

We will briefly touch upon one multicast routing protocol: Core Based Trees (CBT) [3]. CBT builds shared bi-directional trees that are rooted at routers designated to act as the "core" for a particular multicast group. When a leaf network decides to join a multicast group, the router sends a message in the direction towards the core. As the message is received by the next hop router, the router takes notice of this and continues to forward the message towards the core. This process will continue until the message eventually reaches the distribution tree. At this point, the process changes direction back towards the initiating router. For each hop, a new (hop long) branch is added to the tree. Each router must for each active multicast group keep a list of those interfaces that have branches. Thus, CBT state scales $\mathcal{O}(G)$, where G is the number of active groups that have this router on the path towards this groups core (see [5] for a detailed analysis). Remember that different groups can have different cores.

To make IP multicast scalable, it is not required to maintain any knowledge of any individual members of the multicast group, nor of any senders. Group membership is aggregated on a subnetwork basis from the leaves towards the root of the distribution tree.

In a way, IP multicast routing can be regarded as a network level filter for unwanted traffic. At the level of the local subnetwork, network adaptors are configured to filter out local multicasts. Adaptor filters protect the operating system from being interrupted by unwanted multicast traffic and allow the host to spend its cycles on application processing rather than packet filtering.

2.2 Scalable Reliable Multicast

IP packets are only delivered with best-effort. For this reason, JetFile communication relies on the Scalable Reliable Multicast (SRM) [13] paradigm. SRM is designed to meet only the minimal definition of reliable multicast, i.e, eventual delivery of all data to all group members. As opposed to ISIS [6], SRM does not enforce any particular delivery order. Delivery order is to some extent orthogonal to reliable delivery and can instead be enforced *on top of* SRM.

SRM is logically layered above IP multicast and also relies on the same lightweight delivery model. To be scalable, it does not make use of any negative or positive packet acknowledgments, nor does it keep any knowledge of receiver-group membership.

SRM stems from the Application Level Framing (ALF) [7] principle. ALF is a design principle where protocols directly deal with application-defined aggregates suitable to fit into packets. These aggregates are commonly referred to as Application Data Units, or ADUs. An ADU is designed to be the smallest unit that an application can process out of order. Thus, it is the unit of error recovery and retransmission.

The contents of an ADU is arbitrary. It may contain active data such as an operation to be performed, or passive data such as file attributes. In the SRM context, ADUs always have persistent names. The name is assumed to always refer to the same data. To allow for changing data such as changing files, a version number is typically attached to the ADU's name.

The SRM communication paradigm builds on two fundamental types of messages, the *request* and the *repair*. The request is similar to the first half of a remote procedure call in that it requests for a particular ADU to be (re)transmitted. The repair message is quite different from its RPC counterpart. Any node that is capable of responding to the request prepares to do so but first initializes a randomized timer. When the timer expires, the repair is sent. However, if another node responds earlier (all nodes listen on the multicast address) the timer is canceled to avoid sending a duplicate repair. By initializing timers based on round-trip time estimates, repairs can be made from nodes that are as close as possible to the requesting node.

During times of network congestion, hosts behind the point of congestion will sometimes miss repair messages. In this case, it is sufficient if only one of the hosts make a request and then the corresponding repair will repair the state at all hosts.

If the reason to make a request is triggered by an external event such as the detection of a lost packet, one must be careful not to flood the network with request messages. In this case, multiple requests should be suppressed using a similar technique as with multiple repair suppression.

In SRM, reliable delivery is designed to be receiver-driven and is achieved by having each receiver responsible for detecting lost ADUs and initiating repairs. A lost ADU is detected through a "gap" in the version number sequence. Note that this approach only leads to eventual reliable delivery. There is no way for the receiver to know the "current" version number. The problem of maintaining current version numbers must be addressed outside of SRM in an application specific fashion. Also, applications will often be able to recover after a period of packet loss by only requesting the current data. Thus, it is not always necessary to catch up on every missed ADU. In essence, it is not reliable delivery of packets that matters; importance lies in reliable data delivery.

It is easy to build systems that replicate data with SRM. In JetFile we use SRM and replication to increase availability and scalability. This comes at a low cost since peers can easily locate replicas while at the same time the number of messages exchanged will be kept to a minimum.

3 System Overview

JetFile is built from a small number of components that interact by multicasting SRM messages[3]. These are:

File manager The traditional "file system client" that also acts as a file server to other file managers.

Versioning server The part of the system where file updates are serialized. Update conflicts are detected and resolved by file managers.

Storage server A server responsible for the long term storage of files and backup functions.

Key server A server that stores and distributes cryptographic keys used for signing and encrypting file contents.

We have not yet implemented the storage and key servers. This paper describes the file manager, the versioning server, and the protocol they use to interact. The key server is briefly discussed in the Future Work section.

In the JetFile instantiation of SRM, files are named using FileIDs. A FileID is similar to a conventional inode number. Using a hash function, FileIDs are mapped onto the multicast address space. It is assumed that the range of this mapping is large enough so that simultaneously used files will have a low probability of colliding on the same multicast address.

As mentioned in section 2.2, SRM only provides for eventual reliable delivery. Any stronger reliability must be implemented outside of SRM. In JetFile, this problem is addressed with a mechanism called the current table (section 7.2). The current table implements an upper bound to how long a file manager will be using version numbers that are no longer current.

When a file is actively used or replicated at a file manager, the file manager must join the corresponding multicast group. This way the manager will see the request for the file and will be able to send the corresponding repairs.

When reading a file, SRM is first used to locate and retrieve a segment of the file. When the file has been located, the rest of the file contents is fetched from this location using unicast.

Attached to the FileID is a version number. When a file changes, the versioning server is requested to generate a new version number. The request for the new

[3] Where multicast is not used it will be explicitly expressed.

version number and the corresponding repair are both multicast over the file channel. Because both the request and the repair are sent over the file channel these messages will also act as best-effort *callbacks*.

After a file has been updated, it is not immediately committed. Instead, the file is left in a tentative state. Only when/if the file is needed at some other host will it be committed. A file that has not yet been committed can not be seen by other machines.

JetFile directories are stored in regular JetFile versioned files. Filename related operations are always performed locally by manipulating the directory contents. Thus, both file creation and deletion are local operations.

4 File Versioning

Unlike traditional Unix file systems, JetFile adopts a file versioning model to handle file updates. A file is conceptually a suite of versions representing the file's contents at different times. To ensure that a file version is always consistent, JetFile prevents programs running at different nodes from simultaneously updating the file through the use of separate file versions. A new version of a file is writable at precisely one host. If some other host is to update the same file, the system guarantees that it will be writing to a different version. To update a file, it is necessary to request a new version number. Version numbers are assigned by the versioning server which acts as a serialization point for file updates. Once a file manager has acquired a new version number, it is allowed to update the file until the file is committed. A file is not committed until it is replicated at some other node. Thus, the new version number act as an update token for the file. The token is relinquished as a side effect of sending the file over the network.

Write-through techniques are not necessary in JetFile. After a file is updated, there is no need to write the file data through the file cache and over the network since the file manager is now, by definition, acting as a server for the file. The file will be put onto the network only when a file manager explicitly requests it and only at this point is the update token relinquished. This is an important property because it offloads both servers and networks in the common case when the file is not actively shared. Moreover, it is very likely that the file will soon be overwritten. Baker et. al [2] reports that between 65% and 80% of all written files are deleted (or truncated to zero length) within 30 seconds. Furthermore, it is shown that between 70% and 95% of the written bytes are overwritten or deleted within 2 hours.

Because file versions are immutable, JetFile insures that file contents will not change as a result of an update at some other node. The file contents is always consistent (assuming that applications write consistent output). This is particularly important for executable files. If the system allows changing the instructions that are being executed, havoc will surely arise. Most distributed and indeed even some local file systems do not keep the necessary state to prevent this from occurring.

JetFile groups sequences of writes into one atomic file update by bracketing file writes with pairs of `open` and `close` system calls. This approach implies that it is impossible to simultaneously "write share" a file at different hosts. The limited form of write sharing that JetFile supports is commonly referred to as "sequential write sharing" and means that one writer has to close the file before another can open the file for update. In our experience, sequential write sharing does not seriously limit the usefulness of a file system. Sequential write sharing is also the semantics chosen by both NFS and AFS.

Because JetFile only supports atomic file updates there need to be no special protocol elements corresponding to individual `writes`. Only `reads` have corresponding protocol elements.

Requests for new version numbers are addressed to the versioning server but sent with multicast. In this way file managers are informed that the file is about to change and can mark the corresponding cache item as changing. In response to the request, the versioning server sends an SRM repair message. This repair message will act as an unreliable *callback*.

To hide the effects of transmission delays and errors, JetFile takes an optimistic approach to file updates. When a file is opened for writing, the application is allowed to progress while the file manager, in parallel, requests a new version number. This approach is necessary to hide the effects of propagation delays in the network. For instance, the round-trip time between Stockholm and Sydney is about 0.5 seconds. If one was forced to update files in synchrony with the server and could not write the file in parallel with the request for a new file version, file update rates would be limited to two files per second, which is almost unusable.

It is arguable that these kind of optimistic approaches are dangerous and should be avoided because of the potential update conflicts that may arise. There is however empirical evidence indicating that sequential write sharing is rare, Kistler [19] and Spasojevic et. al [33] instrumented AFS to record and compare the identities of users updating files and directories. Kistler found that over 99% of all file and directory updates were by the previous writer. Spasojevic found that over 99% of all directory modifications were by the previous writer. Spasojevic were only able to report on directory write sharing due to a bug in the statistics collection tools. It should be noted that in Kistler's study few users would use more than one machine at a time and that thus cross-user sharing should be similar to cross-machine sharing. In the Spasojevic study it is unknown how users spread over the machines.

In the event that two applications unknowingly update a file simultaneously, conflicting updates are guaranteed to be assigned different version numbers. This fact is used to detect update conflicts: one of the file managers will receive an unexpected version number and will signal an update conflict[4]. We intend to resolve conflicts with *application specific resolvers* as is done in Coda [23] and Ficus [29]. Currently update conflicts are reported but the latest update "wins" and "shadows" the previous update. We have deferred conflict resolution as future work.

File manager caches are maintained in a least-recently-used fashion with the constraint that locally created files are not allowed to be removed from the cache unless they have been replicated at the storage server. It is the responsibility of the file manager that performs the update to make sure that modified files in some way get replicated at the storage server before they are removed. Updated files are allowed to be transferred to the storage server *via* other hosts and even using other protocols such as TCP. The important fact is that the file manager verifies that a replica exists at the storage server before the file is removed from the cache.

5 Current Table

Unlike the unicast *callbacks* used by AFS, the multicast *callbacks* in JetFile do not verify that they actually reach all of their destinations. For this reason, JetFile instead implements an upper bound on how long a file manager can unknowingly access stale data. The upper bound is implemented with the use of a *current table*. The current table conceptually contains a list of all files and their corresponding highest version numbers. The *lifetime* of the current table limits how long a client may access stale data in case callbacks did not get through. If a file manager for some reason does not notice that a file was assigned a new version number, this will at the latest be noticed at the reception of the next current table. The current table is produced by the versioning server and can always be consulted to give a version number that is "off" by at most *lifetime* seconds.

The file manager requests a new current table before the old one expires. Since the current table is distributed with SRM, the table is only transferred every *lifetime* seconds. This work does not impose much load on the versioning server. If some hosts did not receive the table when it was initially transmitted, it will be retransmitted with SRM. If a current table is retransmitted, it is important that the sender first decrement the *lifetime* by the amount of time the table was held locally. The use of *lifetimes* rather than absolute expiration dates has the advantage of not requiring synchronized clocks.

If the current table contained a list of all files, it would clearly be too large to be manageable. For this reason, the file name space has been divided into volumes [32] and the current table is produced on a per volume basis. *Lifetime* is currently fixed at 30 seconds but should probably adapt to whether the volume is changing.

The prototype current table consists of pairs of file and version numbers. The frequent special case when the version number is *one* is optimized to save space. Version *one* is assumed by default. We also expect that the current table can be compressed using delta encodings and using differences to prior versions of the current table. We see these optimizations as future work.

The combination of multicast best-effort *callbacks* and current tables is similar to leases [14]. One of the differences is that with current tables it is not necessary to renew individual leases as lease renewal is aggregated per-volume. Another important difference is that the versioning server is stateless with respect to what hosts were issued a lease. This allows for much larger and more aggressive caching. However, our scheme does not provide any absolute guarantees with its best-effort *callbacks*, as does traditional leases.

In the normal case, JetFile expects that either the request or response message for a new version number will act as a *callback* break. When this fails, consistency is not much worse than for NFS since we use the current table as a fallback mechanism. By avoiding state in the versioning server, it can serve a much larger number of file managers. This comes at the price of only slightly decreased consistency guarantees.

6 The JetFile Protocol

At the JetFile protocol level, files are identified by a file identifier (FileID) similar to a Unix inode number. The FileID is represented by a tuple (*organization*, *volume*, *file-number*). The *organization* field divides the FileID space so that different organizations can share files. This is similar to the cell concept in AFS and DFS. Grouping files into volumes [32] allows several versioning and storage servers to exist and share load within one organization. Each volume is served by precisely one versioning server.

Every FileID is mapped with a hash function into a IP multicast address (the *file address*). All communication related to a file is performed on that file's corre-

[4] Note that since update conflicts are not detected until after a file is closed, the process that caused the conflict may already be dead.

sponding address which thus acts as a shared communication channel. There is also a multicast address for each volume (the *volume address*): the current table is transferred over this channel.

The basic JetFile protocol is simple. Messages consist of a generic header and message type specific parameters, see figure 1. A simplified header consists of a FileID, file version number, and the message type. In requests, the version number zero is used to indicate that the latest version is requested.

org	vol	f-num	vers	msg-type	param...

Figure 1: JetFile message header

message type	parameters	...	
status-request			
status-repair	attributes		
data-request	file-offset	length	
data-repair	file-offset	length	data...
version-request	transaction id		
version-repair	transaction id		
wakeup			

Table 1: JetFile messages and parameters

To request the file attributes of a specific file version, the FileID and version number are put into a status-request. The attributes are returned in a status-repair. To request the current file attributes without knowing the current version number, zero is used as the version number.

Requests for new version numbers are handled slightly differently. Because of the non-idempotent nature of version number incrementation, the versioning server must insure that a repeated request does not increment the version number more than once. To prevent this from occurring, the request (and the repair) carry a transaction id that is used to match retransmitted requests.

File contents is retrieved using data-request and data-repair messages. The requested file segment is identified by offset and length. Retrieval of an entire file is however somewhat more involved than to retrieve only one segment. First, a segment of the file is requested using SRM, then, when we know one source of the file we can *unicast* SRM compatible data-requests to the source, and receive *unicast* data-repairs. A TCP like congestion window is used to avoid clogging intermediate links. A more precise description of this protocol is outside the scope of this paper.

The *wakeup message* is used to gain the attention of the versioning server and will be described in section 7.2. The list of messages types and their parameters is shown in table 1.

The JetFile file name space is hierarchical and makes FileIDs transparent to applications and users. As in many other Unix file systems, directories are stored in ordinary files. The file manager performs directory manipulations (such as the insertion of a new file name) and the translation from pathnames to FileIDs. Directory manipulations are implemented as ordinary file updates. Thus, JetFile does not require any protocol constructs to handle directory manipulations.

7 Implementation

We have implemented a prototype of the JetFile design for HP-UX 9.05. Implemented are the file manager (split between a kernel module and a user-level daemon) and the versioning server. The storage server, key server and security related features have not been implemented.

7.1 File Manager

The implementation of the file manager consists of a small module in the operating system kernel and a user-space daemon process. The daemon performs all the protocol processing and network communication as well as implements the semantics of the file system. The traditional approach would be to implement all of this code inside the kernel for performance reasons. That it was possible to split the file system implementation between a kernel module and a user-level daemon with reasonable performance was shown to work for AFS-like file systems in [34]. There, the authors describe the split implementation of the Coda MiniCache. We will confirm their results by showing that it is possible to implement an efficient distributed file system mostly in user-space. The advantages of a user-level implementation are improved debugging support, ease of implementation, and maintainability. Also, having kernel and user modules tends to isolate the operating system-specific and the file system-specific parts, making it easier to port to other Unix "dialects" and other operating systems.

The file manager caches JetFile files in a local file system (UFS). Only whole-file caching has been implemented. This is, however, an implementation limitation and not a protocol restriction.

7.1.1 Kernel Module

The kernel module implements a file system, a character pseudo-device driver, and a new system call. Communication with the user-space daemon is performed by sending events over the character device. Events are sent from user-space to update kernel data structures and to reschedule processes that have been waiting for data to arrive. The kernel, on the other hand, sends events with requests for data to be inserted into the cache or when about to update read-only cache items. When possible, events are sent asynchronously to support concurrency. For obvious reasons, this is not possible after experiencing a cache read miss. As an optimization, the device driver also supports the notion of "piggy backed" events, i.e. it is possible to send a number of events in sequence to minimize the number of kernel/user-space crossings. The contents of files are not sent over the character device, only references (*file handles*) are transferred.

A new system call is implemented to allow the user-space daemon to access files directly by file handle. This is simpler and more efficient than accessing them by name with `open`.

A typical example of the communication that can arise is the following:

1. the kernel module experiences a cache miss while trying to read an attribute.

2. the kernel module sends an event to the daemon and blocks waiting for the reply.

3. the daemon reads the event, does whatever is necessary to retrieve that data, installs the data in the kernel cache by sending one or more piggy backed events. The last event wakes up the blocked process.

The kernel module also implements a new file system in the Virtual File System (VFS) switch, called YFS (Yet another File System). The VFS-switch in HP-UX is quite similar to the Sun Vnode [21] interface.

The kernel cache contains specialized vnodes used by YFS (called *ynodes*), these have been augmented with fields to cache Unix attributes, access rights, and low-level file identifiers. There is also a reference to a local file vnode that contains the actual file data. The YFS redirects `reads` and `writes` to this local file with almost no overhead.

For simplicity of implementation, the user-space daemon handles `lookup` operations and the result is cached in the kernel directory-name lookup cache (DNLC [30]). When a lookup operation misses in the DNLC, an event is sent to user-space with a request to update the cache.

Pathname translation is performed by the VFS on a component by component basis through consulting the DNLC. For each pathname component, the cached access rights in the ynode are consulted to verify that the user is indeed allowed to traverse the directory. Symbolic links are cached in the same way as file data. After access rights verification, the `readlink` operation is performed by reading the value from the contents of the local file vnode.

To summarize, YFS consists of a cache of actively and recently used ynodes and a cache of translations from directory and filenames to ynodes. The ynodes cache Unix attributes, a reference to a local vnode, and access rights. The local vnode contains file data, directory data, or the value of a symbolic link.

7.1.2 User-space File Manager

The responsibilities of the user-space file manager is to keep track of locally created and cached files. The file manager listens for messages on the multicast addresses of all cached files and for events from the kernel on the character device. When a message arrives from the network, the target file is looked up in the table of all cached files. If the file is found, the message is processed otherwise the message is discarded. These spurious messages result from collisions in the hash from FileID to multicast address or limitations in the host's multicast filtering.

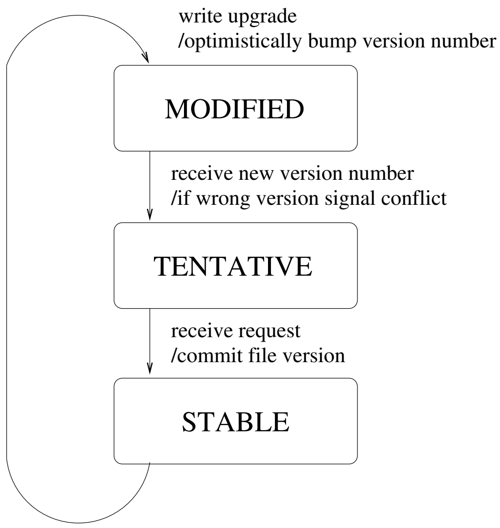

Figure 2: File states

For each cached file, the user-space file manager keeps track of the current version and of the file *state*. The most important states are shown in figure 2. Locally created files start in state *tentative*. The file will stay in this state and can be updated any number of times until some other file manager requests it, at this time the file is committed and the state changes to *stable*. Files received over the network always start in state *stable*.

If some application wants to modify a file that is *stable*, a request for a new version number is made. After sending the version-request, the file manager changes the state to *modified* and optimistically increments the version number by one. A file in the *modified* state only exists locally and cannot be sent to other file managers. When the new version number arrives, the file state will change to *tentative*. If the version number received from the versioning server is not the expected one, the file manager reports that there was an update conflict.

7.2 Versioning Server

The versioning server runs entirely as a user-space daemon. It keeps track of the current version of all files in a volume and replies to requests for new version numbers. Every version number request carries a random transaction id that will be copied to the reply. This is to handle messages that get lost and have to be retransmitted.

The versioning server does not always listen on all file addresses in the managed volumes. Initially, it only listens on volume addresses. When a file manager does not get a repair back from a version-request, the file manager sends a *wakeup message* on the volume address prompting the versioning server to join the multicast group named by the file address. When creating files, the file manager by definition has a token for creating the first version. This avoids needless requests for initial version numbers.

Allocation of file numbers in a volume is handled by the versioning server to guarantee their uniqueness.

The versioning server is also responsible for creating and distributing the current table. The table consists of all files in the volume and their latest version numbers.

8 Measurements

The goal of these measurements is to show that the JetFile design has performance characteristics that are similar to existing local file systems when running with a warm cache.

To make the evaluation we used a standard distributed file systems benchmark, the Andrew Benchmark [16]. This benchmark operates on a set of files constituting the source code of a simple Unix application. Its input is a subtree of 71 files totaling 370 kilobytes distributed over 5 directories. The output consists of 20 additional directories and 91 more files. Thus, in this benchmark the file manager joins 187 IP multicast groups plus one for the current table.

The benchmark consists of five distinct phases: *MakeDir*, which constructs 5 subtrees identical in structure to the source subtree; *CopyAll*, which populates one of the target subtrees with the benchmark files; *StatAll*, which recursively "stats" every file at least once; *ReadAll*, which reads every file byte twice, and finally *Compile* which compiles and links all files into an application.

Tests were conducted over a 10Mb/s Ethernet on HP 735/99 model workstations. In table 2 we present averages and standard deviations over seven runs. We use one machine as the benchmark machine; another is used as a JetFile file manager housing the files, thus acting as a server. Finally, a third machine is used as a versioning server.

To emulate a wide area network (WAN) with long transmission delays, we put a bridge between the benchmark machine and the two other machines. The bridge was configured to delay packets to yield a round trip time of 0.5 seconds.

The benchmark was run first in the local HP-UX Unix File System (UFS) with a hot cache, i.e. all files were already in the buffer cache and file attributes were cached in the kernel. We then ran JetFile with a hot cache, and finally, we ran the same benchmark but with a cold JetFile cache on the benchmark machine so that all files first had to be retrieved over the network.

Before we start to make comparisons, a few facts need to be pointed out:

For unknown reasons the HP-UX UFS initially creates directories of length 24 bytes. When the first name is added, the directory length is extended to 1024 bytes. When adding more names, the directory is normally not extended, only updated. All this results in an extra disk block being written when adding the first name to a directory. JetFile directories start with a length of 2048 bytes and stays that long until full. In both UFS and JetFile, changes to directory blocks are always written synchronously to disk.

Inode allocation in JetFile is asynchronous and logged. The log records are written to disk when either a log disk block becomes full or when the file system is idle. In HP-UX UFS, inode allocation is always synchronous.

We first compare UFS with JetFile, both running with hot caches:

In the *MakeDir* phase, JetFile performs slightly better because UFS suffer from extending directories when adding the first name. JetFile also benefit from its asynchronous inode allocation.

In the *CopyAll* phase, JetFile benefit the most from its asynchronous inode allocation. Note that JetFile stores file data in local UFS files. Both UFS and JetFile use precisely the same "flush changes to disk every 30 seconds" algorithm. Only inode allocation differ between UFS and JetFile.

In the *StatAll*, *ReadAll*, and *Compile* phases, performance of JetFile and UFS are very similar.

Next we compare JetFile with hot and cold caches:

The only difference in the *MakeDir* phase is that we need to acquire a new version number for the top level directory. Since we optimistically continue to write directories while waiting for the new version number to arrive, we should expect only marginal differences in

Phase	UFS hot	JetFile hot	E-WAN hot	JetFile cold	E-WAN cold
MakeDir	1.55 (0.01)	1.22 (0.06)	1.26 (0.03)	1.25 (0.03)	1.28 (0.02)
CopyAll	2.68 (0.06)	1.56 (0.02)	1.55 (0.04)	3.71 (0.13)	50.86 (0.21)
StatAll	2.60 (0.02)	2.59 (0.01)	2.58 (0.01)	2.60 (0.01)	2.58 (0.01)
ReadAll	4.99 (0.02)	5.01 (0.02)	5.01 (0.02)	5.01 (0.02)	5.02 (0.05)
Compile	11.16 (0.05)	11.05 (0.03)	11.05 (0.07)	11.08 (0.08)	11.04 (0.07)
Sum	22.98 (0.08)	21.43 (0.04)	21.45 (0.07)	23.65 (0.12)	70.79 (0.28)

Table 2: Phases of the Andrew benchmark. Means of seven trials with standard deviations. JetFile over an emulated WAN (E-WAN) had a round trip time of 0.5 seconds. All times in seconds, smaller numbers are better.

elapsed time, regardless of round trip time.

In the *CopyAll* phase with a cold cache, the benchmark makes a copy of every file byte, while at the same time JetFile transfers all the files over the network and writes them to local disk. In effect, every byte is written to disk twice, although asynchronously. The combination of waiting for the file bytes to arrive so that the copying can continue in combination with issuing twice as many disk writes explains why the elapsed time increases from 1.56 to 3.71 seconds.

Running with a cold cache over the emulated WAN results in times for the *CopyAll* phase to increase from 3.71 to 50.86 seconds. This is to expect since it takes 0.5 seconds to retrieve a one byte file over the emulated WAN. There is no way out of this problem other than to fetch files before they are referenced. Prefetching will be briefly discussed in the Future Work section 10.2.

File sys.	Warm cache	Cold cache
UFS	22.98 (0.08) 100%	N/A
JetFile	21.43 (0.04) 93%	23.65 (0.12) 103%
AFS	26.49 (0.15) 115%	28.40 (0.60) 124%
NFS	29.54 (0.05) 129%	30.20 (0.20) 131%

Table 3: Total elapsed time of the Andrew benchmark. Percent numbers are normalized to UFS with a hot cache. All times in seconds, smaller numbers are better.

It is interesting to compare JetFile to existing commercially available distributed file systems. We repeated the above experiments using the same machines and network. One machine was used as the benchmark machine and a second was used either as an AFS or NFS file server. Even with tuned commercial implementations such as AFS and NFS, users pay a performance penalty on the order of 20% to make their files available over the network.

It would be very interesting to measure the scalability of the system. Unfortunately, we currently do not have a JetFile port to a more popular kind of machine.

9 Related Work

The idea of building a storage system around the distribution of immutable object versions is not entirely new. These ideas can be traced back to the SWALLOW [28] distributed data storage system. SWALLOW is, however, mostly concerned with the mechanisms necessary to support commitment synchronization between objects, a relatively heavyweight mechanism that is usually not required in distributed file systems.

Another system based on optimistic versioned concurrency control is the Amoeba distributed file system [26]. Their concurrency algorithms allows for simultaneous read and write access by verifying read and write sets before a file is allowed to be committed. Since JetFile targets an environment where files are usually updated in their entirety and write sharing is rare, we decided to use a more lightweight approach.

Coda uses the technique "trickle integration" [27] to reduce delay and bandwidth usage when updating files. File updates are written to the server as a background task after they have matured and been subject to write annulation optimizations. This should be contrasted with the JetFile approach where clients are turned into servers. The JetFile approach make file updates available earlier. The Coda people argue that this is not always desirable, a strongly connected client can be forced to wait for data propagation before it can continue to read the file.

AFS [17] and DFS [18] both implement a limited form of read-only replication. This replication is static and has to be manually configured to meet the expected load and availability. In JetFile the replication is dynamic and happens where the files are actually used rather than where they are expected to be used.

In xFS [1] servers have been almost eliminated by making all writes to distributed striped logs. The design philosophy is "anything anywhere" which should make the system scalable. This is however a different form of scalability: the goal is to scale the number of nodes within a compute-cluster and to improve throughput. The goal of JetFile is to scale geographically over different network technologies. JetFile is also self configuring. There is no need to configure clients as servers or vice versa since JetFile clients per definition take server responsibilities for files accessed.

Frangipani [36] is also designed for compute-cluster scalability, but takes a different design approach. It is designed using a layered structure with files stored in Petal [25] (a distributed virtual disk) and complemented with a distributed lock service to synchronize access.

Both xFS and Frangipani suggest that protocols such as NFS, AFS, and DFS [18] be used to export the file system and provide for distributed access. As far as we know, nothing in our design prevents JetFile file managers from storing their files in xFS or Frangipani.

An earlier xFS paper [37] discusses a WAN protocol designed to connect several xFS clusters. Each cluster has a consistency server, and inter-cluster control traffic, flow through these. Like the JetFile protocol, this protocol also addresses typical WAN problems such as availability, latency, bandwidth, and scalability. This is done by a combination of moving file *ownership* (read or write) between clusters and on-demand-driven file transfers between clients. To reduce consistency server state, file *ownership* is aggregated on a directory subtree basis.

10 Future Work

10.1 Data Security and Privacy

The security architecture of JetFile is an important and central part of the entire design.

Validating the integrity of data is necessary in a global environment. It is also vital to be able to verify the originator of data to prevent impostors masquerading as originators. In JetFile, both these issues will be handled with the use of digital signatures [31].

To allow for cryptographic algorithms with different strengths, JetFile defines different types of keys. These key types in turn define signature types. For instance, there might be one key of type (MD5, RSA-512) which means that when this key is used to make signatures the algorithm MD5 should be used to generate 128 bit cryptographic checksums and RSA with 512 bit keys shall be used for encryption. A different and stronger key type is (SHA1, RSA-1024). Note that key types not only define the algorithms to use but also key lengths.

Since JetFile design is based on the Application Level Framing (ALF, section 2.2) principle it is most natural to protect application defined data units rather than the communication per se. The data units that make up a file are:

Data object: holds the file data and is only indirectly protected.

Status object: holds file attributes. It also contains a lists of cryptographic checksums, each sum protect a segment of the file. The status object is signed by the file author.

Protection object: holds a list of users that are allowed to write to the file and a reference to a privacy object. The protection object is signed by the system.

Privacy object: holds a list of users that are allowed to read the file. It also contains a key that is used to encrypt file contents when the file is transferred over the network.

To verify the integrity of a publicly readable file, first retrieve (with SRM) the key used to sign the status object, check that the author is allowed to write to the file by consulting the protection object and verify the signature. If everything is ok, verify the file segment checksums (checksum algorithm is derived from key type).

Verifying private files is only slightly different. Before the file segment checksums can be calculated the file data must be decrypted with the key that is stored in the privacy object.

The privacy object is not per file. Some files have no associated privacy object at all (publicly readable files). Private files can also be grouped together to share a common privacy object by all pointing out the same privacy object. Note that private files are always transferred encrypted over the network.

Privacy objects cannot be distributed with SRM and shall instead be transferred from the key server through encrypted channels (the channels that are used for bootstrapping, more about this below).

When a user "logs on" a workstation the necessary keys to make signatures are generated. The private key is held locally and is used to make signatures. The public key is registered with the key server using a bootstrap system such as Kerberos [35]. The key server will then sign this key and it will be distributed with SRM to all interested parties. The key used by the system for signing is given to the user during this initial bootstrap.

In JetFile signatures have limited lifetimes, i.e. an object that has been signed can only be sent over the network as long as the signature is still valid. There is no reason to discard an object from the local cache just because the signature expired. Expiration only means that if this object would be sent over the network, receivers will not accept it because the signature is no longer valid. The signature lifetime is derived from the key lifetime. When the key server publishes keys on behalf of users they are also marked with an issue and expiration date.

Signatures will eventually expire, there must be somewhere in the system to "upgrade" signatures. This task is assigned to the storage server. Files are initially signed by users, then stabilize and migrate to the storage server, and eventually their signature expires. At this point, the storage server creates a new signature using a system key and the file is ready for redistribution.

Encryption algorithms are often CPU intensive, we intend to use keys with short key lengths for short lived files such as the current table. Similarly when files are updated they are first signed with a lightweight signature. If the file stabilizes, its signature gets upgraded.

Our strategy of migrating files to the server every few hours is intended to avoid needless CPU-consuming encryptions. In systems such as AFS and NFS where file updates are immediately written through to the server, CPU usage can be costly. In JetFile we only have to invoke the encryption routines when the file is committed, and this happens only when the file is in fact shared, or when it must be replicated at the storage server.

If all files are both secret and shared then there will be extensive encryption. We expect, however, only a small percentage of files to fall into this category and accept that some encryption overhead is unavoidable. In normal operation, users do not share their secret files, so encryption only has to be performed on transfer to and from the storage server.

10.2 Hoarding and Prefetching

For JetFile to work well in a heterogenous environment it must be prepared to handle periods of massive packet loss and even to operate disconnected from the network. JetFile's optimistic approach to handle file updates is only one aspect of attacking these problems. There must also be mechanisms that try to avoid compulsory cache misses.

One way of avoiding compulsory cache misses is to identify subsets of regularly used files and then hoard them to local disk. A background task will be responsible for monitoring external file changes and to keep the local copies reasonably fresh. External file changes can be detected by snooping the current table. Hoarding has been investigated in Coda [20] and later refined in SEER [22]. We plan to integrate ideas from their systems into JetFile.

Hoarding can only be expected to work well with regularly used files. There must also be a mechanism to avoid compulsory cache misses on files that are not subject to hoarding. We suggest that files are prefetched as a background task controlled by network parameters delay and available bandwidth. If the delay is long and bandwidth is plentiful it makes sense to prefetch files to decrease the number of compulsory cache misses.

11 Open Issues and Limitations

JetFile requires file managers to join a multicast group for each file they actively use or serve. This implies that routers will be forced to manage a large multicast routing state.

The number of multicast addresses used may be decreased by hashing FileIDs onto a smaller range. This has the disadvantage of wasting network bandwidth, and to force file managers to filter out unwanted traffic. How to strike the balance between address space usage and probability of collision we do not know.

The concept of *wakeup messages* can be generalized to wakeup servers for files. A message can be sent to a file's corresponding volume address, the message will make passive servers join the file address, in effect activating servers. This idea can be further generalized. A message can be sent to an *organization address* to make passive servers join the volume address. A disadvantage of this approach is that *wakeup messages* will introduce a new type of delay.

Lastly, when writing this, we do not yet have any experience with large scale Inter-domain multicast routing. If the routers and multicast routing protocols of tomorrow will be able to cope with the load generated by JetFile remains to be seen.

12 Conclusions

The JetFile distributed file system combines new concepts from the networking world such as IP multicast routing and Scalable Reliable Multicast (SRM), as well as proven concepts from the distributed systems world such as caching and callbacks, to provide a scalable distributed file system for operation across the Internet or large intranets. JetFile is designed for ubiquitous distributed file access. To hide the effects of round-trip delays and transmissions errors, JetFile takes an optimistic approach to concurrency control. This is a key factor in JetFile's ability to work well over long high-speed networks as well as over high delay/high loss wireless networks.

Dynamic replication is used to localize traffic and distribute load. Replicas are synchronized, located, and retrieved using multicast techniques. JetFile assigns server duties to clients to avoid the often expensive effects of writing data through the local cache to a server. In the common case when files are not shared, this is the optimal means to avoid unwanted network effects such as long delays, packet loss, and bandwidth limitations. Multicast routing and SRM are used to keep communication to a minimum. In this way, files are easily updated at their replication sites.

Callback renewal is aggregated on a per-volume basis and shared between clients to reduce server load. Moreover, JetFile *callbacks* are stateless and best-effort. This implies that servers need not keep track of which hosts are caching particular files. Thus client caching can be much more aggressive. It should be pointed out, that because of the best-effort nature of our *callback* scheme, clients may, under some circumstances, not be aware of cache invalidity for up to 30 seconds.

We have implemented parts of the JetFile design and experimentally verified its performance over a local area network. Our measurements indicate that, using a standard benchmark, JetFile performance is close to that of a local disk-based file system.

13 Acknowledgments

We would like to thank our shepherd Tom Anderson and the anonymous reviewers for their constructive comments which lead to significant improvements of this paper.

We also thank Mikael Degermark for reading early versions of this paper and making many insightful comments.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, R. Y. Wang, *Serverless Network File Systems*, In Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995.

[2] M. G. Baker, J. Hartman, M. D. Kupfer, K. W. Shirriff, J. Ousterhout, *Measurements of a Distributed File System*, In Proceedings of the 13th ACM Symposium on Operating Systems Principles, 1991.

[3] T. Ballardie, P. Francis, J. Crowcroft, *Core Based Trees (CBT)*, In Proceedings of the ACM SIGCOMM 1993.

[4] D. Banks, C. Calamvokis, C. Dalton, A. Edwards, J. Lumley, G. Watson. *AAL5 at a Gigabit for a Kilobuck.* Journal of High Speed Networks, 3(2), pages 127-145, 1994.

[5] T. Billhartz, J. Cain, E. Farrey-Goudreau, D. Fieg, S. Batsell, *Performance and Resource Cost Comparisons for the CBT and PIM Multicast Routing Protocols*, IEEE Journal on Selected Areas in Communications, 15(3), Apr. 1997.

[6] K. Birman, A. Schiper, P. Stephenson, *Light–weight Causal and Atomic Group Multicast*, ACM Transactions on Computer Systems, 9(3), Aug. 1991.

[7] D. D., Clark, D. L. Tennenhouse, *Architectural considerations for a new generation of protocols*, In Proceedings of the ACM SIGCOMM 1990.

[8] S. Deering, *Host Extensions for IP Multicasting*, RFC 1112, Internet Engineering Task Force, 1989.

[9] S. Deering, *Multicast Routing in a Datagram Internetwork*, PhD thesis, Stanford University, Dec. 1991.

[10] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei, *The PIM Architecture for Wide-Area Multicast Routing*, IEEE/ACM Transactions on Networking, 4(2), Apr. 1996.

[11] S. Deering, C. Partridge, D. Waitzman, *Distance Vector Multicast Routing Protocol*, RFC 1075, Internet Engineering Task Force, 1988.

[12] W. Fenner, *Internet Group Management Protocol, Version 2*, RFC 2236, Internet Engineering Task Force, 1997.

[13] S. Floyd, V. Jacobson, , C. Liu, S. McCanne, L. Zhang, *A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing*, IEEE/ACM Transactions on Networking, 5(6), Dec. 1997.

[14] C. G. Gray, D. R. Cheriton, *Leases: an efficient fault-tolerant mechanism for distributed file cache consistency*, In Proceedings of the 12th ACM Symposium on Operating System Principles, 1989.

[15] B. Grönvall, I. Marsh, S. Pink, *A Multicastbased Distributed File System for the Internet*, In Proceedings of the 8th ACM European SIGOPS Workshop, 1996.

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

[17] M. L. Kazar, *Synchronization and Caching Issues in the Andrew File System* In Proceedings of the USENIX Winter Technical Conference, 1988.

[18] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Buttos, S. Chutani, C. F. Everhart, W. A. Mason, S. Tu, E. R. Zayas, *DEcorum file system architectural overview* In Proceedings of the Summer USENIX Technical Conference, 1990.

[19] J. J. Kistler, *Disconnected Operation in a Distributed File System*, PhD thesis, Carnegie Mellon University, May. 1993.

[20] J. J. Kistler, M. Satyanarayanan, *Disconnected Operation in the Coda File System* ACM Transactions on Computer Systems, 10(1), Feb. 1992.

[21] S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, In Proceedings of the USENIX Summer Technical Conference, 1986.

[22] G. H. Kuenning, G. J. Popek, *Automated Hoarding for Mobile Computers*, In Proceedings of the 16th ACM Symposium on Operating Systems Principles, 1997.

[23] P. Kumar, M. Satyanarayanan, *Flexible and Safe Resolution of File Conflicts*, In Proceedings of the USENIX Winter Technical Conference, 1995

[24] S. Kumar, P. Radoslavov, D. Thaler, C. Alaettinoglu, D. Estrin, M. Handley, *The MASC/BGMP Architecture for Inter-domain Multicast Routing*, In Proceedings of the ACM SIGCOMM 1998.

[25] E. K. Lee, C. A. Thekkath, *Petal: Distributed virtual disks*, In Proceedings of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 1996.

[26] S. J. Mullender, A. S. Tanenbaum, *A Distributed File Service Based on Optimistic Concurrency Control*, In Proceedings of the 10th ACM Symposium on Operating Systems Principles, 1985.

[27] L. Mummert, M. Ebling, M. Satyanarayanan, *Exploiting Weak Connectivity for Mobile File Access*, In Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995.

[28] D. P. Reed, L. Svobodova, *SWALLOW: A Distributed Data Storage System for a Local Network*, Local Networks for Computer Communications, North-Holland, Amsterdam 1981.

[29] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, G. Popek, *Resolving File Conflicts in the Ficus File System*, In Proceedings of the USENIX Summer Technical Conference, 1994.

[30] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, *Design and Implementation of the Sun Network Filesystem*, In Proceedings of the USENIX Summer Technical Conference, 1985.

[31] B. Schneier, *Applied Cryptography, Second Edition*, John Wiley & Sons, Inc, 1996.

[32] B. Sidebotham, *VOLUMES - The Andrew File System Data Structuring Primitive*, In Proceedings of the European Unix User Group Conference, Manchester, 1986.

[33] M. Spasojevic, M. Satyanarayanan, *An Empirical Study of a Wide-Area Distributed File System*, ACM Transactions on Computer Systems, 14(2), May. 1986.

[34] D. C. Steere, J. J. Kistler, M. Satyanarayanan, *Efficient User-Level File Cache Management on the Sun Vnode Interface*, In Proceedings of the USENIX Summer Technical Conference, 1990.

[35] J. S. Steiner, C. Neuman, J. I. Schiller, *Kerberos: An Authentication Service for Open Network Systems*, In Proceedings of the USENIX Winter Technical Conference, 1988.

[36] C. A. Thekkath, T. Mann, E. K. Lee, *Frangipani: A Scalable Distributed File System*, In Proceedings of the 16th ACM Symposium on Operating Systems Principles, 1997.

[37] R. Wang, T. Anderson, *xFS: A Wide Area Mass Storage File System* In Proceedings of the Fourth Workshop on Workstation Operating Systems, 1993.