A Multicast-based Distributed File System for the Internet

Björn Grönvall, Ian Marsh, and Stephen Pink
Swedish Institute of Computer Science
{bg, ianm, steve}@sics.se

Abstract

JetFile is a file system designed with multicast as its distribution mechanism. The goal is to support a large number of clients in an environment such as the Internet where hosts are attached to both high and low speed networks, sometimes over long distances. JetFile is designed for reduced reliance on servers by allowing client-to-client updates using scalable reliable multicast. Clients on high speed networks prefetch large numbers of files. On low speed networks such as wireless, special caching policies are used to decrease file access latency. The prototype implementation of JetFile is on the JetStream gigabit local area network which provides hardware support for many multicast addresses. The multicast Internet backbone (Mbone) is the wide area testbed for JetFile.

1 Introduction

To achieve scalability in a wide area network environment, the next generation of distributed file systems need a new paradigm of communication. The prevailing mode of communication for current distributed file systems is *unicast*, where a client finds a server, opens a connection and transfers the file or parts of the file to the client's cache. We propose to use *multicast* as the main paradigm of communication to achieve low load on networks and servers. In a multicast-based distributed file system, files migrate to requesters from possessors without the requesters necessarily knowing the possessors' identity and location. Thus, files are replicated on clients who need them without overloading servers and, because modern networks often have hardware support for multicast, files are replicated without undue load on networks and hosts.

In addition, most distributed file systems have been designed for hosts connected to medium-speed local area networks such as Ethernets. Tomorrow's networks, however, will have a much larger dynamic range, from gigabit fiber optic to slow speed cellular networks. On long fiber optic networks, the dominant communication latency is the propagation time of the network. On low speed wireless networks, the dominant latency is the transmission time on the wireless link. In both cases, adaptive policy can avoid network round trip times and hence file access delay.

The JetFile distributed file system is designed to cope with this delay through prefetching policies based on the bandwidth and delay of a communication path. For long high speed networks, JetFile prefetches large numbers of files to prevent multiple round trips across the network. On low speed networks, prefetching policies first bring over only those parts of a file that a user needs to begin work. Later, in the background, the rest of the file may be transferred and other related files hoarded to allow users to operate on their files even when disconnected from the network, as often happens on wireless links. Thus, JetFile minimizes latency in communication between clients and servers on both high and low speed networks and provides support for disconnected operation.

The purpose of this paper is to describe the multicast model of communication in JetFile as well as the prefetching policies needed to adapt to a wide range of networks. The paper begins with a rationale for multicast in a distributed file system. We then describe how JetFile uses scalable reliable multicast for distribution. Consistency and scalability problems are then addressed for wide area distributed file systems. Included are also sections on data security and privacy as well as a discussion of network issues followed by a conclusion.

2 Why Multicast?

Multicast is a scalable method of replicating data among many nodes in a wide area distributed system. The inherent location transparency in multicast communication enables data to be retrieved regardless of its location. This eliminates the need to contact a central server to either retrieve the data, or to find the actual location of data. This approach can be used to eliminate unnecessary backbone-traffic and lengthy round trip delays since it is not always necessary to retrieve data from the source. Multicast communication can also be used to save bandwidth when there is sharing by "snooping" the data as it passes by.

Traditionally, distributed file systems have been designed on a unicast model of communication. The client/server approach is for a client to first locate the server, open up a connection, and retrieve the file. There are, however, scaling problems with centralized systems. These problems include overloading of the server by clients, network congestion, and reliability. xFS [1] solves these problems with a "serverless" implementation. We think that there are advantages to having a server and that the major scaling problems can be overcome by using a multicast model of communication.

Our solution to this problem is for clients to act as servers for file and meta-data requests but retain some functionality such as security, reliability and consistency in traditional servers. Retrieving a copy from your neighbor's cache will save both network bandwidth and server resources. The multicast paradigm provides location transparency by allowing requesters to directly request data without first having to locate the server that has the data. By assigning a multicast address to each file using a simple mapping between file identifiers and multicast addresses, a user need only translate the name of a file to its multicast address and the underlying multicast routing protocols locate the file wherever it resides. Several files may map (hash) to the same multicast address, so the requests carry file identifiers to make it possible to distinguish requests for different files. There are however 2^{112} multicast addresses in IPv6 so this should not happen frequently.

Many new local area networks provide increased hardware support for multicast communication. For example, the network on which JetFile is being implemented, JetStream [3] has hardware support for more than 16,000 multicast addresses. This makes it feasible to do hardware multicast filtering at file granularity. The hardware multicast filters prevent the host and its system software from being interrupted by packets relevant only to files that are not cached on that host.

On the Internet level, multicast routing protocols [4, 5] are designed to protect local networks from packets destined to group addresses for which there are no subscribers. IP multicast [7] is however not a reliable transfer protocol. To provide reliability of data transfer we use a version of the "Scalable Reliable Multicast" (SRM) algorithm described in [9].

3 Reliable Multicast Distribution

By substituting SRM for the classical client/server remote procedure call paradigm, we make communication reliable and move many of the classic server roles to the clients, henceforth referred to as *managers*. To avoid acknowledgment implosion, we use a framework that does not expect acknowledgments from each receiver in a group. When a message is missing, the manager explicitly requests retransmission by issuing a *repair request*. Anyone receiving the initial message is able to respond with a *repair message*.

For example, in the case of a cache miss, when a file is opened for reading, the manager joins the group associated with the file. A repair request for the file is sent to that group address and a member of the group responds with a repair message. Traffic associated with transferring this file will consequently *only* be seen by those parties that are interested in this particular file.

Versioning is used to ensure message ordering and communication reliability. Packet loss is detected through "gaps" in the version number sequence. If packet loss is detected, repair requests are multicasted, where one repair message can repair files on many hosts. To suppress multiple repair messages, randomized timers are used. This scheme only guarantees the eventual delivery of all packets. We require another mechanism to make guarantees about present version numbers. This is discussed in the next section.

4 Consistency

In JetFile, files have version numbers which are incremented when a file is updated. Once a new file version has been created it can be freely replicated on other managers. Applications that open files for reading are ensured that the file contents will not change as they read the file, i.e they read the version they initially opened. This is particularly important with executable files, if the system allows changing the instructions that are being executed havoc will surely occur. Unfortunately many commercial distributed file systems allow this. Updates are bracketed by pairs of open and close system calls, this implies that changes won't propagate to other hosts until the file is closed. This form of write sharing is commonly referred to as "sequential write sharing" and means that one writer has to close the file before another opens it for update. In our experience, sequential write sharing does not seriously limit the usefulness of a file system, it is also the semantic chosen by both NFS and AFS.

JetFiles's design prevents concurrent write sharing from occurring by defining consistency on a per file-version basis. When a file is opened for writing, a new version number is assigned to that file. This is done by the manager sending a message to the *versioning server* which responds with a new version number. The user can continue to write the file *optimistically* while waiting for the version number to arrive, so communication with the versioning server does not impose any overall delay. It is guaranteed that the same version number will never be assigned to two different files since the versioning server acts as a serialization point for version number generation. Should two applications unknowingly update a file almost simultaneously the conflicting files will be assigned different version numbers, this makes it possible to later resolve the conflict with an *application specific resolver* as is done in Coda [13] and Ficus [15]. Version number request and repair messages are sent to the multicast group associated to the particular file. Thus, all interested managers know when a new version number has been assigned to a file.

Managers may not always be aware of the current version of a file. This may be due to a lost message or a manager has failed to "snoop" on the relevant multicast group. To make sure that managers are aware of the latest versions of files, we employ a scheme to retrieve the current version number of any file in the system. Since the versioning server acts as a serialization point it knows all version numbers and can upon request produce a table of all current version numbers. This table is periodically requested by managers to prevent them from using stale versions. Information about current version numbers is distributed with SRM. Thus, it is not necessary for every manager to request this information from the versioning server. A *time-to-live* (TTL) field in the table prevents managers from distributing old information and also insures that *current tables* do not circulate indefinitely.

This scheme differs from a more traditional use of callbacks or leases. Since the requests and responses for new version numbers are done using multicast they also act as unreliable *callback breaks*. The current table guarantees that if packets are dropped, stale version numbers will not be in use for any longer than the TTL. For example, a TTL of 30 seconds could be chosen to give a worst case guarantee similar to NFS. In the normal case it is expected that either the request or response message for a new version number act as a callback break.

Should the current table be a list of all files, it would clearly be unmanageable. The file system name space, therefore, is divided into volumes and the current table is produced on a per volume basis. Furthermore, in many situations it is possible to send *diffs* relative to previous tables in order to keep size down. To ensure the uniqueness of *file numbers* (inode numbers), within a volume these are also generated at the versioning server.

5 Scalability

There is a trend in distributed file system design towards decentralization. The goal of xFS, for example, is to eliminate the traditional server by allowing a client to be configured as a server. Some services are however not easily distributed, we retain the server concept for versioning and security, which are easier implemented at a centralized point.

In our system, any manager is able to satisfy a file read request. The system automatically replicates frequently used files, eliminating the need for managers to consult the server on every access. For reliability, a master copy of every file is kept on a *storage server*. This acts as a central file repository in a known location where managers can deposit files for long term storage. If a manager has updated a file it must write it back to the storage server prior to leaving the multicast group or there will be no server for the updated file.

Updates of a file are not necessarily written through the manager's cache. Immediate write through is unnecessary because the manager will act as a server for that file and is very likely to be overwritten in any case. Eventually, when the file is "stable", it will be replicated on the storage server. Current trends in research indicate delayed writes to be beneficial. [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.

For scalability reasons directory name lookup operations are performed by the file managers. Directories are implemented as simple tables stored as ordinary files and lookup is performed in a component by component fashion. The file system name space is divided into organizations and further into volumes. Each volume is assigned one storage server and one versioning server[1] allowing several machines to absorb the generated load. As an organization grows, administrators can add more servers and relocate volumes as necessary.

6 Network Adaptation

The evolution of network technology in recent years show two trends. Local and wide area wireline networks are increasing in bandwidth. It is not uncommon for networks users to have bandwidths of more than 100 Mb/s, and networks of gigabits/sec are around the corner. On the other hand, there has also been a wireless revolution in communications encouraging the growth of mobile computing. Although wireless bandwidths are likely to increase, wireless networks are also likely to be one or two orders of magnitude slower than wireline networks. Thus, the dynamic range of networks is increasing and JetFile is designed to cope with this large dynamic range with its adaptable write-back and prefetch policies.

6.1 High speed networks

On a high speed wide-area network, prefetching large numbers of files can be used to hide network propagation delay. This is important because propogation delay on a high bandwidth network is the same as for a low bandwidth network. Put in another way, the delay for the first bit of a message on a high speed network is the same as for a low bandwidth network of the same length, other things equal. The last bit of the message, of course, arrives sooner on the high speed network than on the low speed [14].

For a distributed file system such as JetFile this means that fetching large portions of a user's working file set will eliminate round-trip delays to other managers or servers that are snooping on the multicast addresses for those files. [11]. We intend to perform aggressive prefetching on networks with high bandwidth/delay products such as parts of the Internet that connect high speed local area networks such as JetStream via high bandwidth backbones.

By recording previous file access patterns it is possible to predict future file references [12] and to retrieve those files in advance so that when they are actually used they will already be available locally. After an initial *open*, related file opens are recorded for the file that was initially accessed. This profiling information is stored together with the file meta-data and is used in our system to guide the prefetching algorithms. Prefetching is implemented as a background task, whenever a new file is referenced related files are scheduled for prefetch. The most passive form of prefetching is to join on the relevant multicast addresses in the hope that data can be "snooped", when this fails repair requests are sent at a rate to not flood the network.

[1] Servers need not be co-located.

6.2 Low speed networks

In contrast, prefetching on a low speed network could be costly and time consuming. Cost and available bandwidth are two determining factors when deciding upon write-back and prefetch policy. A user on a relatively expensive wide-area cellular network may wish to do only moderate amounts of file prefetching and limit the number of times a file is transferred to safe storage. Typically, files will only be written-back in a bulk transfer at the end of an editing session, since this strategy does not require the user to be connected during the session and uses available bandwidth more effectively.

Perhaps the best "prefetch" policy on a low-speed network is to fetch only those parts of the file that are necessary for a user to begin work. This policy wagers that the rest of the file may never be used and so does not have to be transferred from server to client and back; kind of "anti-prefetch" policy. Prefetching, in the form of file hoarding, however, has been shown to be useful for distributed file system clients that are mobile [10]. Hoarding allows the user to operate more effectively while disconnected from the server. JetFile will attempt to combine these two policies. On low speed networks, only those parts of a file shown by previous access patterns to be likely to be used will be fetched initially on reference to the file. Then, as a lower priority background process, the rest of the file and perhaps other files will be prefetched and cached to provide support for disconnected operation.

7 Data Security and Privacy

Validating the integrity of data is necessary in a global environment. In particular it is vital to be able to verify the originator of data to prevent impostors masquerading as originators. IP multicast routing protocols permit users to receive packets by simply listening on an address. Thus, encryption and authentication are needed to enforce file system permissions.

When a file is created, we lazily[2] make a cryptographic signature with public key cryptography to *seal* the file. The seal confirms the creator of the file, the public half of the key is registered with the *key server*. The private half is used to calculate the seal and is held locally. Upon reception of a file, the receiver uses the public half of the key to verify the originator's identity.

To protect file data, we perform a cryptographic checksum such as MD5 on the data to ensure that it has not been changed during transmission. The key server stores the public half of public key pairs. It can also be used to hold and distribute keys for symmetric encryption. Since we use delayed writes, we only have to perform these calculations in the event that some other manager wants to read the file.

In JetFile, protection is allotted on a per file basis. The file creator is responsible for generating and registering keys with the key server for symmetric encryption. This key is used to encrypt and decrypt before and after transmission.

If all files needed to be protected *and* shared, there would have to be extensive encryption. We expect, however, only a small percentage of files to fall into this category and accept that some encryption overhead is necessary. 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. When the file is safely deposited on the storage server, it must be removed from the local cache.

8 Wide Area Network Issues

When designing a distributed file system for WANs such as the Internet, issues such as flow and congestion control become important. Since JetFile uses IP multicast and SRM instead of TCP connections to reliably distribute data, one cannot count on the latter to provide end-to-end flow and congestion control. We have developed some mechanisms in JetFile that use features of IP multicast to provide flow control and prevent congestion. We also use a new feature in IPv6 to provide local scope to multicast transmissions.

[2] This is only necessary when a file is actually shared..

8.1 Prefetch Policies and Multicast Addresses

JetFile exploits the large multicast IPv6 address space, 2^{112} available multicast addresses out of a possible 2^{128} total addresses, to aid in providing flow and congestion control in hosts and networks. When a host capable of sending very fast has receivers that are both fast and slow, there is a dilemma of how fast to send. If data is sent at the high speed, the low speed receivers cannot keep up; their receive buffers overflow causing losses and retransmissions. And even when the receiver is capable of "sinking" data from a fast sender, there may be bottleneck links on the path between sender and receiver that cause network congestion, also resulting in loss and retransmissions. On the other hand, if the sender only sends as fast as its slowest receiver can receive, those hosts that can receive at high speed will not be used to their full capacity.

To deal with this dilemma, JetFile assigns more than one multicast address per file. One address is reserved for the comparatively low bandwidth file meta-data transfer and the others are used for file data transfers. Depending on the speed and buffering capabilities of receivers and networks paths between senders and receivers, file managers choose to communicate on one or more of the data addresses. For example, there may be three hosts interested in reading the same file. One host resides on a locally attached high speed network such as JetStream, another host may be connected to an Ethernet, and a third have access only to a slow speed cellular link. In our scheme, a sender can multicast file data in response to a request or repair message on three different multicast addresses corresponding to the bandwidth/delay conditions on the path to the receiver or the capability of the receiver itself to sink data. A receiver will request on and listen to only those multicast addresses for the file that will not cause buffers to be overrun, either in the receiver itself (flow control) or in the network (congestion control). Different prefetch policies are defined to reflect properties of the underlying networks. Should a network become congested, managers will leave the corresponding multicast group(s) and thereby limit the scope of too bandwidth consuming communication.

Sending on multiple addresses at different rates could be a burden on the sender if the sending rates were badly chosen. Thus, the spread between the rates must be large as we have illustrated in the hypothetical example. Sending at 800 Mb/s, 10 Mb/s and 9600 bits/s is not much more costly to the sender than sending only at 800 Mb/s.

8.2 Multicast Scoping

A potential problem with using multicast for distribution is limiting the multicast to those recipients that are interested in them. Thus, JetFile is currently being implemented on a local area network that provides hardware filtering support for tens of thousands of multicast group addresses, sparing the operating system software from having to service a software interrupt for every multicast packet on the network. Also, JetFile makes use of IP multicast routing protocols such as DVMRP (Distance Vector Multicast Routing Protocol) and sparse-mode PIM (Protocol Independent Multicast). DVMRP [8] prunes away portions of an internetwork from a multicast, protecting segments of the network with no group member hosts from unnecessary transmissions. PIM establishes rendezvous points in the internetwork for senders and sparsely positioned receivers to meet.

We will use IPv6 addressing[6] and can thus limit the scope of our transmissions effectively as this new protocol allows multicasts to be scoped to the local node, link, site, or organization. Since multicast scoping is encoded in the IPv6 multicast address itself, multicast addresses can be reused across different scopes. This virtually increases the already large IPv6 multicast address space (2^{112}) and insures that JetFile can share its address space with other multicast-based distributed applications without fear of global address depletion.

9 Implementation

Currently a prototype implementation of JetFile is being done on HP-UX. The user space file manager maintains a cache of file data and meta data inside the kernel. File data is stored in native UFS files making it easy to redirect reads and writes to the UFS implementation, it is also easy for the file manager to directly access these files when file data is transferred to/from the network. When an application accesses something not in the cache it sends a request to the file manager to update the cache and then sleeps. When the file manager has updated the cache it sends a message to wakeup the sleeping process. Subsequently the process wakes up, reexamines the cache and resume. If an application opens a file for update a message is sent asynchronously to the file manager so that a new version number can be assigned to that file.

A pseudo device driver has been implemented to support asynchronous message passing between kernel space and the file manager. Kernel to user space messages are queued inside the kernel and are retrieved with the read system call. User to kernel space messages are similarly transfered using the write system call, once the write is executing in kernel space the messages are immediately processed by the writing (message sending) thread.

Our implementation of the file manager as a combination of a file system cache in kernel space that is maintained by a user space process is similar to the implementation of the Coda cache managar that is separated into Venus (the user space process) and the MiniCache [16].

10 Related Work

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. Unfortunately this also implies that all components of the system mutually trust each other. JetFile still keeps a few trusted servers but moves classical servers functions to the managers to offload its servers. Trust is implemented on a protocol level so that it is not necessary to trust managers.

xFS is designed for a switched local area network such as ATM. xFS shows that a unicast model of communication along with decentralized server functionality can use the underlying bandwidth of a switched network efficiently. The aggregate bandwidth of a switch can only be used if there are many connections through the switch concurrently. So a system with many clients and one server would not be able to exploit switch bandwidth.

For a non-switched, i.e, broadcast network, such as Ethernet and FDDI among local area networks and the Internet as a wide area network, multicast distribution is best way to use bandwidth efficiently. On a local area broadcast network, multicast is trivial as all hosts can listen to messages sent to a multicast group. On the Internet (or intranets), multicast routing protocols have been developed to provide efficient distribution only to networks that have member hosts, tending to prune networks off the distribution tree that have no member hosts. Unfortunately, on a switched network, multicast is not trivial to implement and can result in poor bandwidth utilization as packets must be copied multiple times to all destination ports. Thus, JetFile employs the right distribution mechanism for its target networks.

AFS and DFS implement a limited form of read only replication. This replication is static and have 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.

Security and privacy in JetFile is based on a model quite similar to PGP. Both systems use public keys to verify the integrity and origin of data. In PGP there is no central key server; instead, keys are manually passed around between users. JetFile keeps a central key server where keys are registered. Keys also have a limited lifetime and eventually expire.

11 Conclusion

JetFile is a distributed file system designed for use in the wide area on heterogeneous networks such as the Internet. To achieve less reliance on central servers, JetFile uses scalable reliable IP multicast for file distribution. JetFile adapts its prefetch policy to accommodate the underlying network and so provides low latency file access for users on both high and low bandwidth networks. Write-back policies are designed to minimize both network and server resources. To protect data in a wide area network, JetFile uses encryption and authentication.

References

[1] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A Patterson, Drew S. Roselli, Randolph, Y. Wang. *Serverless Network File Systems.* 15th Symposium on Operating Systems Principles, ACM Transactions on Computer Systems, 1995.

[2] Mary G. Baker, John Hartman, Michael D. Kupfer, Ken W. Shirriff, John Ousterhout. *Measurements of a Distributed File System.* Proceedings of the Thirteenth ACM Symposium on Operating System Principles, Pacific Grove, CA, October 1991.

[3] David Banks, Costas Calamvokis, Chris Dalton, Aled Edwards, John Lumley, Greg Watson. *AAL5 at a Gigabit for a Kilobuck.* Journal of High Speed Networks, 3(2), pages 127-145, 1994.

[4] D. Cheriton and Steve Deering. *Multicast Routing in Datagram Internetworks and Extended LANs.* ACM Transactions on Computer Systems, pages 85-111, May 1990.

[5] S. Deering, D. Estrin, D. Farinacci, Van Jacobson, C.G. Liu, L. Wei . *An Architecture for Wide-Area Multicasting Routing.* ACM SIGCOMM 1994, London, September 1994.

[6] Steve Deering and R. Hinden. *IP Version 6 Addressing Architecture.* RFC 1883, Xerox PARC, Ipsilon Networks, December 1995.

[7] Steve Deering. *Host Extensions for IP Multicasting.* Request For Comments 1112, Stanford, CA: Computer Science Department, August 1989.

[8] Steve Deering, Craig Partridge, D. Waitzman. *Distance Vector Multicast Routing Protocol*, RFC 1075, BBN STC, Stanford University, November 1988.

[9] Sally Floyd, Van Jacobson, Steve McCanne, Ching-Gung Liu, Lixia Zhang. *A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing.* ACM SIGCOMM, Pages 342-356, Boston, MA, October 1995.

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

[11] Anders Klemets and Stephen Pink. *Low Latency File Access in a High Bandwidth Environment.* Proceedings of the Sixth ACM European SIGOPS Workshop, September 1994.

[12] Thomas M. Kroeger, Darrell D.E. Long. *Predicting File Actions from Prior Events.* Proceedings of the 1996 Usenix Winter Technical Conference, San Diego.

[13] *Flexible and Safe Resolution of File Conflicts.* Kumar, P., Satyanarayanan, M. Proceedings of the USENIX Winter 1995 Technical Conference Jan. 1995, New Orleans, LA

[14] Craig Partridge.*Gigabit Networking.* Addison-Wesley. 1994.

[15] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, Gerald Popek *Resolving File Conflicts in the Ficus File System* Proceedings of the 1994 Usenix Summer Technical Conference, Boston MA.

[16] David C. Steere, James J. Kistler, M. Satyanarayanan *Efficient User-Level File Cache Management on the Sun Vnode Interface* Proceedings of the 1990 Summer USENIX Conference June 1990, Anaheim, CA.