Adding HTTP-like Streams to the 9P Filesystem Protocol

by

John Floren

A Thesis Proposal Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Muhammad Shaaban
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
March 2010

Approved By:

Muhammad Shaaban
Associate Professor, RIT, Department of Computer Engineering
Primary Adviser

Secondary Adviser One
Secondary Adviser Title and Department

Secondary Adviser Two
Secondary Adviser Title and Department

Abstract

The Plan 9 operating system from Bell Labs has broken a great deal of new ground in the realm of operating systems, providing a lightweight, network-centric OS with private namespaces and other important concepts. In Plan 9, all file operations, whether local or over the network, take place through the 9P file protocol. Although 9P is both simple and powerful, developments in computer and network hardware have over time outstripped the performance of 9P. Today, file operations, especially copying files over a network, are much slower with 9P than with protocols such as HTTP or FTP.

9P operates in terms of reads and writes with no buffering or caching; it is essentially a translation of the Unix file operations (open, read, write, close, etc) into network messages. Given that the original Unix systems only dealt with files on local disks, it seems that it may be time to extend 9P (and the file I/O programming libraries) to take into consideration the fact that many files now exist at the other end of network links. Other researchers have attempted to rectify the problem of network file performance through caching and other programmer-transparent fixes, but there is a second option. *Streams* allow programmers to read and write data sequentially (an extremely common case) while reducing the number of protocol messages by half and avoiding some of the problems with round-trip latency. By adding streaming messages to the 9P protocol and extending the regular POSIX I/O functions, it should be possible to provide an interface that programmers can use to read files sequentially at nearly the speed of HTTP.

1. Thesis Objectives

The Plan 9 operating system serves all file operations using the 9P protocol. The contents of the local disk, the contents of removable media, remote file servers, and even device drivers such as the VGA and hard drive devices are accessed via 9P. It is essentially a network-transparent set of file operations.

In Plan 9, client programs access files using normal procedure calls, such as open, close, read, write, create, and stat. Calling these functions generate system calls; those system calls which operate on files are translated into 9P and sent over the network to the appropriate file server. Messages come in pairs: a client sends a T-message (such as `Tread`) and receives an R-message (such as `Rread`) in response. The only exception is Rerror, which may be sent in response to any T-message.

When a user opens a file, the client program makes an `open()` system call, which is translated into a 9P message, `Topen`, which is sent to the file server. The file server responds with a `Ropen` message when the file has been opened. The client then sends `Tread` and `Twrite` messages to read and write the file, with the server responding with `Rread` and `Rwrite`. When the transaction is completed, the client sends `Tclunk` and receives `Rclunk` in return, at which point the file is closed.

This mode of operation is conceptually very simple. Data is sent only when requested by the client; there is no read ahead, buffering, or inherent caching. In 1995, when files were small and networks tended to be local within an organization, this was not a problem. Today, files may regularly stand in the range of gigabytes. Large files are very frequently transferred over high-latency networks, such as a trans-continental Internet link.

Anecdotal evidence indicates that downloading a file from a Plan 9 server to a Plan 9 client using 9P is several times slower than downloading the same file via HTTP. One example is the simple case of playing an MP3 file from a remote machine. If the file is

accessed via 9P, playback stutters every few seconds, because the player application issues read calls for the next chunk of data and runs out of audio to play while it is still waiting for the new data to arrive. If, however, the file is fetched via HTTP and piped to the music player (`"hget http://server/file.mp3 | games/mp3dec"`), playback continues smoothly.

The structure of 9P itself may now be ready for extension. As it stands, when a client sends a read request, it must suffer the effects of latency in waiting for a response. For example, consider a 300 ms latency between client and server. The client sends a `Tread` message asking for 2048 bytes of data, which after 300 ms arrives at the server. The server processes the request and replies with a `Rread`, which then takes 300 ms to return to the client with the data. The client must wait at least 600 ms total between the asking for the information and receiving it; given blocking reads, this means 600 ms have been wasted. If, on the other hand, the server was to send as much data as possible to the client kernel for buffering, the client may find that the data it has requested is already local. Instead of sending a `Topen`, a client could send `Tsopen`, ”stream open”. The server would then begin sending data from the beginning of the file to the client. Read calls from the client would then return data that was already local.

This concept of ”streaming” the entire file at once differs from traditional caching in several important points. In general, caching applies to every application working in the namespace served by the cache. It ”just happens”, without requiring any modifications to the client programs. Streaming requires the application writer to explicitly request a stream in places where it makes sense to have a stream–such as writing out a file from a text editor or playing an MP3–while still allowing the use of traditional, non-cached, non-sequential reads and writes elsewhere.

Plan 9 is now also being explored in supercomputing applications; in this as much as in any other application, it is essential to be able to transfer data quickly. Supercomputing operations in particular must distribute large amounts of information to a large number

of nodes as quickly as possible. While the internal networks of supercomputers have extremely low latency, the use of `Tread/Rread` pairs adds to the congestion of the network, injuring performance. Using streams to transfer data sets and executable programs between supercomputer nodes could have a huge impact on the time a computation takes, in a field where small delays add up fast.

Performance of Plan 9's file server speed will be measured by transferring files between computers. It will be compared to HTTP, which as of today is certainly the most common method for transferring files, especially over long distances.

1.1 Preliminary Measurements

A number of tests were performed to compare the transfer speeds of HTTP and 9P. Two Plan 9 computers, one running as a CPU/auth/file server ("gnot") and one running as a standalone terminal ("illiac") were connected via a network consisting of a 100 Mbit switch and a 10 Mbit hub, with a Linux computer acting as a gateway between the switch and the hub. The Linux computer was configured using the `netem` kernel extensions to induce delay between the two halves of the network. The use of a 10 Mbit hub served to reduce the available bandwidth, which in combination with the artificially-induced latency was intended to resemble an Internet connection.

File Size (MB)	9P (sec.)	HTTP (sec.)
10	10.60	11.91
50	51.96	62.04
100	103.64	124.80
200	208.81	249.50

Table 1.1: HTTP vs. 9P, no induced latency, average RTT 500 s

As Table 1.1 shows, there is very little difference between 9P and HTTP when latency is low–in fact, 9P slightly outperforms HTTP.

File Size (MB)	9P (sec.)	HTTP (sec.)
10	29.57	14.08
50	147.44	70.81
100	296.06	140.38
200	590.94	281.63

Table 1.2: HTTP vs. 9P, induced latency of 15 ms RTT

However, when 7.5 ms of latency is introduced in each direction (total RTT of 15 ms), 9P falls badly behind (see Table 1.2. While HTTP takes only 1.13 times as long to copy a 200 MB file with 15 ms RTT versus 500 s RTT, 9P takes 2.83 times as long!

File Size (MB)	9P (sec.)	HTTP (sec.)
10	76.16	19.43
50	374.07	98.88
100	747.13	197.90
200	1512.31	400.00

Table 1.3: HTTP vs. 9P, induced latency of 50 ms RTT

Table 1.3 shows the results of the final test, with a RTT of 50 ms. Note that it took HTTP about 250 seconds to copy a 200 MB file with no induced latency, and required less than twice that time (400 seconds) to copy the same file with 25 ms of latency in each direction. 9P, on the other hand, went from a speedy 208 seconds to about 1,512 seconds–over 25 minutes!

Given that latencies on the Internet can easily reach surpass the times tested here–at the time of writing, the RTT from Rochester, NY to Livermore, CA was around 90 ms–it is clear that 9P is not suitable for transferring data across the Internet.

1.2 Supporting Work

In 2007, Francisco Ballesteros et. al.[2] presented their work on a high-latency system. Their ongoing project is called Octopus, a distributed system built on top of Inferno, which in turn is derived from Plan 9. A replacement for the 9P protocol was written which combines several file operations into two operations, `Tput/Rput` and `Tget/Rget`. Thus, a client may send a single `Tget` request to open a file, get the metadata, and read some of the data.

The Op system was implemented using a user-mode file server, Oxport, and a user-mode client, Ofs. Oxport presents the contents of a traditional 9p-served file tree via Op. Ofs then communicates with Oxport using Op; Ofs provides the exported file tree to clients on the local machine. When a user program, such as cat, attempts to read a file that actually resides on a remote machine, the 9P commands are sent to Ofs. Ofs then attempts to batch several 9P messages into a single Op message; a call to `open()` normally sends `Twalk` and `Topen` messages, but Ofs will batch them both into a single `Tget` message to walk to the file, open it, and fetch the first `MAXDATA` bytes of data. Metadata from files and directories is also cached locally for the duration of a "coherency window" in the interest of reducing network read/writes while not losing coherency with the server file system.

The Op protocol showed significant reductions in program run-time latency. An `lc` (equivalent to `ls`) that originally took 2.3 seconds to complete using original 9P took only 0.142 seconds with a coherency window of 2 seconds. The bandwidth was improved by orders of magnitude when building software using `mk`.

While Op gave good results, its design is not optimal. The Op protocol is only spoken by Oxport and Ofs. As user-level programs, they incur more overhead than in-kernel optimizations. Also, when the functionality is implemented as a transparent operation, it does not allow the programmer to choose between traditional 9P-style operations and Op; a separate streaming system would give that additional flexibility. Op also saw poorer performance than regular 9P on low-latency links, which are the norm in supercomputers; as Plan 9 expands into supercomputing and across the global Internet, it needs an improved 9P

protocol that can work well over both high- and low-latency connections. Finally, Op was optimized for small files, having a relatively small `MAXDATA` (the amount of data that can be transferred in one Rget message). Op would still need to execute many Tget requests to transfer a large file, which is one of the cases of interest in this thesis.

The NFS protocol version 4 utilizes a similar strategy with its COMPOUND RPCs, which allow clients to batch up several RPCs into one message. Thus, a client could read from a file in one RPC by sending a LOOKUP, OPEN, and READ all in a single COMPOUND RPC.[8]

Oleg Kiselyov in 1999 presented a paper[3] describing his work on a file system based on HTTP. This file system, HTTPFS, is capable of reading, writing, creating, appending, and deleting files using only the standard `GET, PUT, HEAD,` and `DELETE` requests.

The HTTPFS consists of two components, a client and a server. The client can in fact be any program which accesses files; such a program is converted to an HTTPFS client by linking with a library providing HTTPFS-specific replacements for the regular file system calls, such as open, read, close, etc. These replacement functions simply call the standard system calls, *unless* the filename given begins with `http://`. If a URI is given, the function instead creates an appropriate HTTP request and sends it to the server. For example, calling `open("http://hostname/cgi-bin/admin/MCHFS-server.pl/README.html", O_RDONLY)`[3] causes the client to send out a `GET` request for that file. When the file is received, the client caches it locally; reads and writes then take place on the locally cached copy.

Kiselyov wrote an example HTTPFS server, called MCHFS. MCHFS acts much like a regular web server, allowing any browser to get listings of files. However, it also allows the user to access the entire server filesystem under the path component `DeepestRoot`, as in `open("http://hostname/cgi-bin/admin/MCHFS-server.pl /DeepestRoot/etc/passwd", O_RDONLY)`[3].

An interesting factor of the HTTPFS design is its approach toward caching and concurrency. When a file is opened, the entire file is fetched via HTTP and cached locally. All

reads and writes to that file then take place on the local copy. Finally, when `close()` is called, the local copy is written back to the server using `PUT`.

In some ways, HTTPFS is quite similar to the goal of this thesis. It reads in the entire remote file at once, then redirects reads and writes to the local copy. However, as with Op, it does not give the user any choice: all HTTP-served files are read all at once and cached locally, while all other files are accessed traditionally. The goal of 9P streams is to allow the programmer a clear choice in accessing files, either by the traditional open/read/write methods, or using streams.

The Low-Bandwidth File System (LBFS) adapted typical caching behavior for low-bandwidth operations. The most salient change was the use of hash-indexed data chunks. LBFS clients maintain a large local file cache; files are divided into chunks and indexed by a hash. These hashes are then used to identify which sections of a file have been changed and avoid retransmitting unchanged chunks. When reading a file, the client issues a GETHASH request to get the hashes of all chunks in the file, then issues READ RPCs for those chunks which are not already stored locally. This technique provided excellent results but the entire premise is rapidly becoming far less important; bandwidth is rarely a problem today. LBFS made no provisions to account for latency, which remains a problem over long links due to the limitations of switching technology and the speed of light.

Patterson, Gibson, and Satyanarayanan in 1993 experimented with the use of Transparent Informed Prefetching (TIP) to alleviate the problems of network latency and low bandwidth. With TIP, a process informs the file system of its future file accesses. For instance, the `make` program prepares a directed acyclic graph of dependencies, including files; this list of files would be sent to the filesystem for pre-fetching. In testing, separate processes were used to perform pre-fetching from local disks and Coda file servers. Results showed significant (up to 30

Other researchers have applied parallelism to the task.[4][1] Filesystems such as PVFS stripe the data across multiple storage nodes; a client computer then fetches chunks of files from several different computers simultaneously, reducing the impact of latency and the

bottleneck of bandwidth to some extent. Others[5] spread data across the disks of the client nodes and indexed it using a metadata server. A simple parallel file transfer program already exists in Plan 9: the fcp program uses several threads, each copying their own portion of the file–but it is unreasonable to expect every program to implement multi-threaded file reading to cover the holes in 9P. Parallel file systems are frequently not an option; often, only one computer is available to serve files. The complexity of coordinating multiple servers and having the client deal with all of these servers makes a simpler solution desirable.

1.3 Project Deliverables

The following things will be produced by the conclusion of the thesis:

- The thesis document itself.
- A white paper that summarizes this work, to be submitted for publication.
- A corpus of C code to implement the improvements.

2. Schedule

Tentative timeline; dates are approximate:

- TODO

3. Required Resources

To properly develop and test this work, four computers are required:

- A server running Plan 9 to provide file and authorization services.
- A computer running Plan 9 to act as a workstation and test server.
- A computer running Plan 9 to act as a test client.
- A computer running Linux, to do network bridging and induce latency between the test server and client.

As of July, all four computers have been acquired and set up.

Bibliography

[1] Phillip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *ALS'00: Proceedings of the 4th annual Linux Showcanse & Conference*, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[2] Francisco Ballesteros et. al. Building a Network File System Protocol for Device Access over High Latency Links. In *IWP9 Proceedings*, December 2007.

[3] Oleg Kiselyov. A Network File System over HTTP: Remote Access and Modification of Files and *files*. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, June 1999.

[4] Jason Lee, Dan Gunter, Brian Tierney, Bill Allcock, Joe Bester, John Bresnahan, and Steve Tuecke. Applied techniques for high bandwidth data transfers across wide area networks. In *In Proceedings of International Conference on Computing in High Energy and Nuclear Physics*, 2001.

[5] Pierre Lombard and Yves Denneulin. NFSP: A distributed NFS server for Clusters of Workstations. *Parallel and Distributed Processing Symposium, International*, 1:0035, 2002.

[6] A. Muthitacharoen, B. Chen, and D. Mazires. A low-bandwidth network file system. In *Proc. 18th ACM Symp. Op. Sys. principles*, 2001.

[7] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27:23–34, 1993.

[8] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. RFC 3530: Network file system (nfs) version 4 protocol, April 2003. Status: PROPOSED STANDARD.