# FTP-like Streams for the 9P Protocol

John Floren

Rochester Institute of Technology

December 10, 2010

Advisor: Dr. Muhammad Shaaban
Committee Members: Dr. Roy Melton, Dr. Ron Minnich

# Outline

- Background
- Motivation
- Implementation
    - User-level Implementation
    - Kernel-level Implementation
- Testing and Results
    - Test Programs
    - Test Results
- Conclusion

# Background: Plan 9

- Research operating system from Bell Labs
- By the late 80s, UNIX was showing its age
- The UNIX team decided to try again with Plan 9
- Includes features not found in UNIX:
    - Private namespaces
    - Unified filesystem protocol[1]
    - **Everything** is a file
    - Graphics and networking support

# 9P

- Unified filesystem protocol for Plan 9[5]
- Every file operation uses 9P in some fashion
- RPC transactions
  - Client sends T-message
  - Server responds with R-message
- Messages map closely to libc function calls
- Only one connection to the server–requests from multiple client programs are multiplexed

# Default 9P Messages

| Message Name | Function |
|---|---|
| Tversion/Rversion | Exchanges client/server 9P version numbers |
| Tauth/Rauth | Authenticates client with server |
| Rerror | Indicates an error (includes error string) |
| Tflush/Rflush | Aborts a previous request |
| Tattach/Rattach | Establishes a connection to a file tree |
| Twalk/Rwalk | Descends a directory hierarchy |
| Topen/Ropen | Opens a file or directory |
| Tcreate/Rcreate | Creates a file |
| Tread/Rread | Reads data from an open file |
| Twrite/Rwrite | Writes data to an open file |
| Tclunk/Rclunk | Closes an open file |
| Tremove/Rremove | Removes a file |
| Tstat/Rstat | Requests information about a file |
| Twstat/Rwstat | Changes information about a file |

Source: [1]

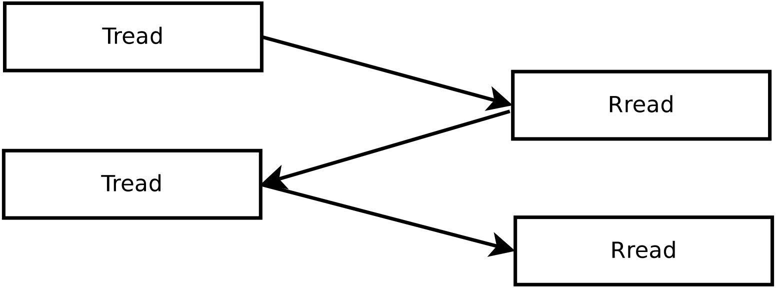
# File Servers in Plan 9

- "File server" is a generic term for a program that responds to 9P requests
- Files do not necessarily have to reside on a physical medium
    - `ftpfs`
    - `rio`
    - `gpsfs`
- File servers typically communicate using a TCP connection or a local pipe
- To the kernel, the connection to the filesystem appears as a generic `Chan` (channel)
- The kernel takes I/O requests from programs and converts them to 9P messages to send over the channel

# The Problem with 9P

- Copying files via 9P is slow.
- Why? 9P waits for a single response for every message sent.
- Over high-latency links, the waiting becomes problematic.
  - Client sends a `Tread` and starts waiting
  - 50 ms later, `Tread` arrives at server, which responds
  - Another 50 ms later, the `Rread` arrives at the client
- The client spends 100 ms waiting for each chunk of data read!

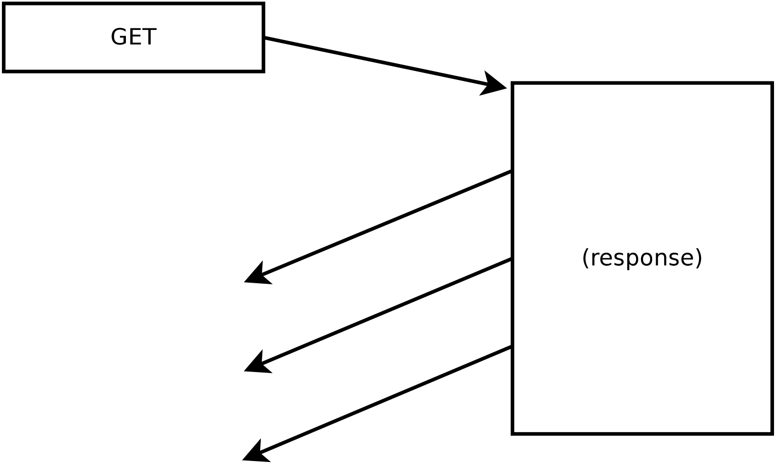
# The 9P Model



# Sequential Reading, FTP and HTTP

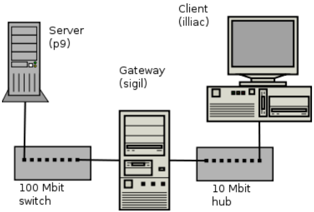
- Sequentially reading or writing a file is one of the most common cases
- 9P makes no concessions to sequential file I/O
- HTTP and FTP are focused on sequential I/O
  - Both are significantly limited compared to 9P
  - Both are quite fast over high-latency links
  - The trick: send all the data at once, don't wait for the client to ask
- Specifically, FTP in passive mode negotiates a separate TCP connection for data transfer.[6]

# The HTTP Model

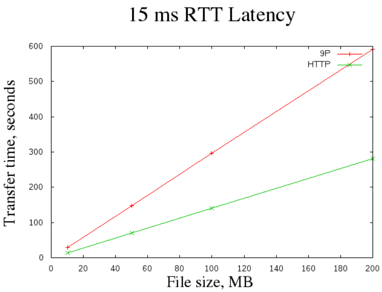


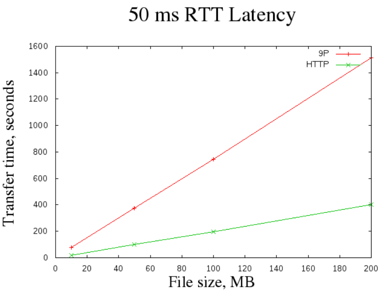
# Testing 9P

- HTTP and 9P were compared for transferring files over high-latency links.
- To control the experiment, a high latency connection was simulated over a LAN
- Latency induced using a Linux gateway running `netem`



# High-latency Results

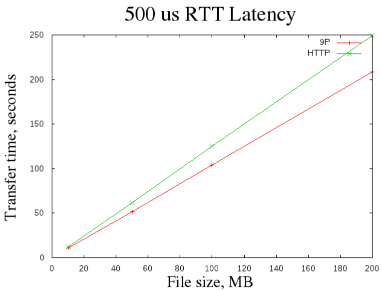



Copying a 200 MB file over 9P across a 50ms RTT link took more than 25 minutes. The same operation required less than 7 minutes with HTTP.

# Low-latency Results

Over low-latency links, 9P outperformed HTTP



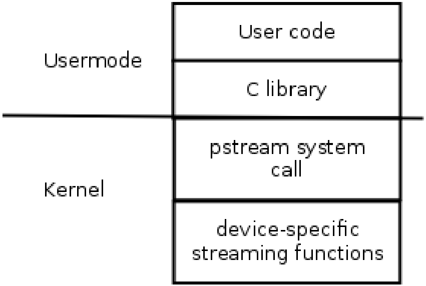
# Streams

- Give programmers the option of having FTP-like behavior
- Augment the `Tmessage/Rmessage/Tmessage/Rmessage` paradigm
- Negotiate an out-of-band channel for data
    - TCP already provides in-order, guaranteed delivery with flow control.
    - One TCP connection per file
- In the case of non-streaming servers, regular reads and writes can provide compatibility
- Streams negotiated using `Tstream/Rstream` messages
    - `size[4] Tstream tag[2] fid[4] isread[1] offset[8]`
    - `size[4] Rstream tag[2] count[4] data[count]`

# How Streams are Different

- Others have attempted to improve filesystem speeds transparently
  - Op[2] intercepts 9P messages and batched them
  - HTTPFS[3] uses GET, PUT, and local caching to work with HTTP-accessible files
  - LBFS[4] uses indexes of file blocks to avoid transmitting data which already exists at the other end
  - NFS[7] uses pre-fetching in an attempt to anticipate the next read
- Streams work explicitly, at the programmer's discretion
- Programmers know when files will be read sequentially
- Trying to be too clever may bite you with unexpected behavior

# Implementation Layers




- The implementation consists of several layers in both user-land and kernel code
- Each layer will be discussed in the following slides

# User-Level Functions

- User programs create and interact with streams using three new libc functions
  - `Stream* stream(int fd, vlong offset, char isread)`
  - `long sread(Stream* s, void* buf, long len)`
  - `long swrite(Stream* s, void* buf, long len)`
  - `int sclose(Stream* s)`
- The `Stream` structure contains the following elements:
  - `int ofd`: The file descriptor of the underlying file
  - `int conn`: A file descriptor for the TCP connection
  - `char *addr`: The IP and port used by the TCP stream
  - `vlong offset`: The current offset of the stream in the file
  - `char isread`: Flag indicating read/write status of stream; 1 indicates read
  - `char compatibility`: Flag indicating that compatibility mode should be used

# `stream` Function

- Sets up a new Stream structure
- Expects an open file descriptor and a desired offset into the file
- Calls the `pstream` system call
- If streaming is allowed, system call indicates such by returning 0
  - Syscall also sets the `addr` field of the Stream struct
  - `stream` function then calls `dial` with address to get a TCP connection
- If streaming is not allowed, system call returns -1
  - `stream` function sets `compatibility` flag in the Stream struct
  - Any reads or writes on the stream will actually be emulated

# `sread` and `swrite` Functions

- Read or write from a Stream
- If the compatibility flag is not set, reads and writes are done on the TCP connection in the Stream structure
- If the compatibility flag is set, reads and writes are done by calling the `pread` and `pwrite` functions
  - Reads and writes are done on the `ofd` file descriptor
  - The `offset` struct member is updated and used to specify offsets when performing emulated streaming

# System Call

- Was necessary to add a new system call
- `pstream` called by the `stream` libc function
- Provides an interface between user-level code and devices which actually set up streams
- Takes a file descriptor, pointer to a buffer, an offset, and a read/write flag
- Returns -1 or 0 based on streaming capability of device/server
- If streaming is supported, buffer is filled with a "dial" string upon return
    - Example: "tcp!129.21.0.123!23456"

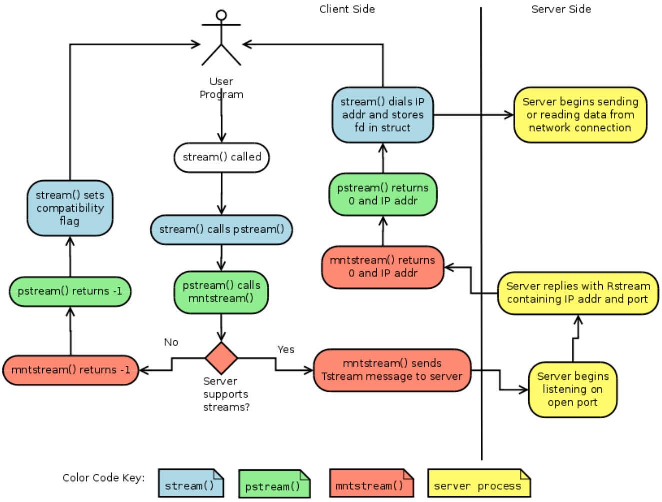
# Devices

- Individual devices are responsible for setting up streams
- Most devices are local; streams don't make sense, so compatibility mode is used
- The `devmnt` device is the most important
  - `devmnt` connects to remote filesystems
  - Attaches filesystems to the local namespace

# `devmnt` Modifications

- Was necessary to update 9P version
  - Originally spoke "9P2000"
  - Now reports as speaking "9P2000.s"
- When stream setup function in `devmnt` is called:
  - Version of remote server is checked
  - If version is not 9P2000.s, compatibility mode must be used, -1 is returned
  - Otherwise, a `Tstream` message is sent out
  - Message requests a new stream for the specified file, starting at a given offset
  - `isread` flag in the `Tstream` message identifies requested stream as read or write

# Stream Configuration Flowchart



# Adapting Programs for Streaming

- It is simple to modify a user program to use streams
  - Locate sequential file reading/writing situations
  - After the file is opened, call `stream` with the resulting file descriptor
  - Call `sread` or `swrite` instead of `read` or `write`
- 9P servers are slightly more difficult
  - Have server report its version as 9P2000.s
  - Add handler function for incoming `Tstream` messages
  - Begin listening on an open port
  - Send `Rstream` reply with IP and port
  - Wait for connection
  - Send/receive data to/from connection

# The `exportfs` file server

- The `exportfs` file server was modified to serve streams
- `exportfs` exports a server's namespace to a client
    - Simple design
    - Very commonly used
- 9P message handler modified to recognize `Tstream` messages
- If a stream is requested:
    - `exportfs` begins listening on an unused TCP port
    - Sends back the IP address and port number in the `Rstream` reply
    - Waits until client connects to the port
    - In the case of a read stream, continually send file data over the TCP connection until EOF.
    - In the case of a write stream, continually read data over the TCP connection until no data remains.

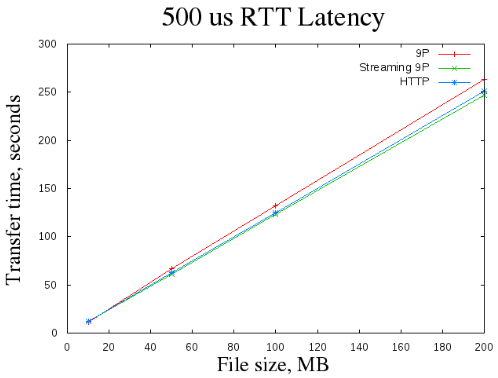
# Streaming `cp`

- Standard `cp` opens source and destination files, then copies 8 KB at a time using `read` and `write`
- In modified "streaming `cp`", or `scp`, both files are opened, then streams are created on the open files
- The source file is set up as a read stream
- Destination file is set up as a write stream
- `sread` and `swrite` are called to read 8 KB at a time from the read stream and write it back out to the write stream
- Compatibility mode allows use with non-streaming servers
- Modifications required the addition of 2 lines of code and slight changes to 2 other lines

# Testing Setup

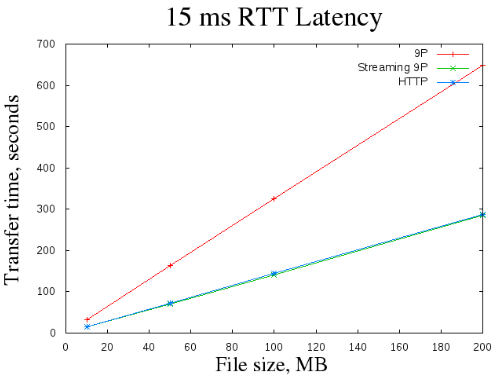
- Used same setup as preliminary HTTP vs. 9P measurements
- Tests consisted of copying files over HTTP, streaming 9P, and non-streaming 9P
- As before, randomly-generated test files of 10MB, 50MB, 100MB, and 200MB were transferred
- Tests were performed with latencies of 500 $\mu$s, 15 ms, and 50 ms RTT

# 500 $\mu s$ RTT



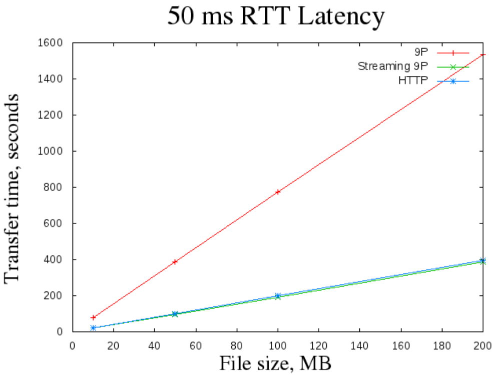

| File Size (MB) | 9P (sec.) | Streaming 9P (sec.) | HTTP (sec.) |
|---|---|---|---|
| 10 | 11.36 | 12.21 | 12.56 |
| 50 | 67.23 | 61.29 | 62.41 |
| 100 | 132.34 | 123.32 | 124.49 |
| 200 | 263.33 | 247.34 | 251.22 |

# 15 ms RTT



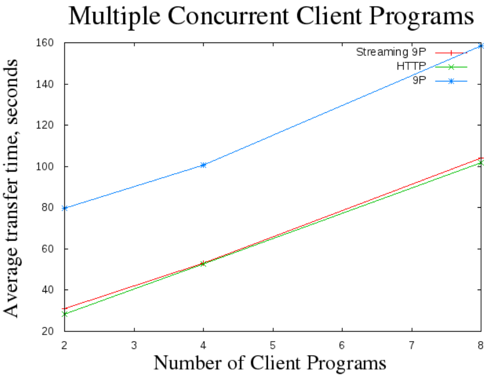

| File Size (MB) | 9P (sec.) | Streaming 9P (sec.) | HTTP (sec.) |
|---|---|---|---|
| 10 | 31.77 | 14.38 | 15.15 |
| 50 | 163.45 | 71.00 | 72.56 |
| 100 | 324.46 | 140.68 | 144.96 |
| 200 | 647.92 | 284.71 | 287.23 |

# 50 ms RTT




| File Size (MB) | 9P (sec.) | Streaming 9P (sec.) | HTTP (sec.) |
|---|---|---|---|
| 10 | 78.82 | 21.80 | 21.45 |
| 50 | 387.92 | 96.19 | 98.39 |
| 100 | 773.03 | 192.60 | 198.14 |
| 200 | 1535.81 | 385.69 | 395.53 |

# Concurrent Downloads, 50 ms RTT




| Number of Clients | Streaming 9P (sec.) | HTTP (sec.) | 9P (sec.) |
|---|---|---|---|
| 2 | 30.96 | 28.51 | 79.75 |
| 4 | 53.28 | 52.53 | 100.81 |
| 8 | 104.24 | 101.93 | 158.53 |

# Future Work

- Other 9P servers, such as Fossil and Venti, should be converted to use streams
- User programs which can benefit from streaming should be modified:
  - `page` document and image viewer
  - `mp3dec` music player
- Design may eventually be included in the Plan 9 distribution

# Conclusions

- Streaming 9P has demonstrated that it can match the performance of HTTP
- POSIX file semantics have been expanded to give attention to sequential file I/O
- Rather than try to improve performance transparently, explicit library functions let the programmer choose when streaming is important
- Requires very small programmer effort for large performance gains

# Bibliography I

[1] *Introduction to the Plan 9 file protocol, 9P*. Plan 9 online manual, section 5.

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

[5] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. 8(3):221–254, Summer 1995.

[6] J. Postel and J. Reynolds. RFC 959: File transfer protocol, October 1985.

[7] 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.