Persistent 9P Sessions for Plan 9

Gorka Guardiola, paurea@gmail.com
Russ Cox, rsc@swtch.com
Eric Van Hensbergen, ericvh@gmail.com

ABSTRACT

Traditionally, Plan 9 [5] runs mainly on local networks, where lost connections are rare. As a result, most programs, including the kernel, do not bother to plan for their file server connections to fail. These programs must be restarted when a connection does fail. If the kernel's connection to the root file server fails, the machine must be rebooted. This approach suffices only because lost connections are rare. Across long distance networks, where connection failures are more common, it becomes woefully inadequate. To address this problem, we wrote a program called `recover`, which proxies a 9P session on behalf of a client and takes care of redialing the remote server and reestablishing connection state as necessary, hiding network failures from the client. This paper presents the design and implementation of `recover`, along with performance benchmarks on Plan 9 and on Linux.

1. Introduction

Plan 9 is a distributed system developed at Bell Labs [5]. Resources in Plan 9 are presented as synthetic file systems served to clients via 9P, a simple file protocol. Unlike file protocols such as NFS, 9P is *stateful*: per-connection state such as which files are opened by which clients is maintained by servers. Maintaining per-connection state allows 9P to be used for resources with sophisticated access control policies, such as exclusive-use lock files and chat session multiplexers. It also makes servers easier to implement, since they can forget about file ids once a connection is lost.

The benefits of having a stateful protocol come with one important drawback: when the network connection is lost, reestablishing that state is not a completely trivial operation. Most 9P clients, including the Plan 9 kernel, do not plan for the loss of a file server connection. If a program loses a connection to its file server, the connection can be remounted and the program restarted. If the kernel loses the connection to its root file server, the machine can be rebooted. These heavy-handed solutions are only appropriate when connections fail infrequently. In a large system with many connections, or in a system with wide-area network connections, it becomes necessary to handle connection failures in a more graceful manner than restarting the server, especially since restarting the server might cause other connections to break.

One approach would be to modify individual programs to handle the loss of their file servers. In cases where the resources have special semantics, such as exclusive-use lock files, this may be necessary to ensure that application-specific semantics and invariants are maintained. In general, however, most remote file servers serve traditional on-disk file systems. For these connections, it makes more sense to delegate the handling of connection failure to a single program, rather than need to change every client (including `cat` and `ls`).

We wrote a 9P proxy called `recover` to handle network connection failures and to hide them from clients, so that the many programs written assuming connections never fail can continue to be used without modification. Keeping the recovery logic in a single program makes it easier to debug, modify, and even to extend. For example, in some cases it might make sense to try dialing a different file system when one fails.

This paper presents the design and implementation of `recover`, along with performance measurements.

2. Design

Recover proxies 9P messages between a local client and a remote server. It posts a 9P service pipe in /srv or mounts itself directly into a name space, and then connects to the remote server on demand, either by dialing a particular network address or by running a command (as in *sshsrv*).

Recover keeps track of every active request and fid in the local 9P session. When the connection to the remote server is lost, the remote tags and fids are lost, but the local ones are simply marked ‘‘not ready.’’ When recover later receives a new request over the local connection, it first redials the remote connection, if necessary, and then reestablishes any fids that are not ready. To do this, recover must record the path and open mode associated with each fid, so that the fid can be rewalked and reopened after reconnecting. Reestablishment of remote state is demand-driven: recover will not waste network bandwidth or server resources establishing connections or fids that are not needed.

The details of what state needs to be reestablished vary depending on the 9P message. The rest of this section discusses the handling of each message and then some special cases. We assume knowledge of the 9P protocol; for details, see section 5 of the Plan 9 manual [9].

Version. Not directly proxied. Recover assumes the 9P2000 version of the protocol. It is considered an error if the remote server lowers its chosen maximum message size from one connection to the next.

Auth. Not directly proxied. Recover requires no authentication from its local client, and uses the local factotum [2] to authenticate to the remote server. 9P2000 has no mechanism to reinitiate authentication in the middle of a 9P conversation, so using the local factotum is really the only choice. A recover running on a shared server could use the host owner’s factotum to authenticate on behalf of other users, allowing multiple users to share the connection. This connection sharing behavior, which would need to trust the uname in the Tattach message, is not implemented.

Attach. Not directly proxied. Recover keeps a single root fid per named remote attachment and treats Tattach as a clone (zero-element walk) of that root fid.

Walk. Recover reestablishes the *fid* if necessary and then issues the walk request. On success, recover records the path associated with *newfid* in order to recreate *newfid* if needed in the future.

Open. Recover reestablishes *fid* if necessary and then forwards the request. On success, recover records the open mode associated with *fid* so that it can be reopened in the future.

Create. Handled similarly to open, except that recover must also update *fid*’s path on success.

Read, **Write**, **Stat**, **Wstat**. Recover reestablishes *fid* if necessary, and then forwards the request.

Clunk. If *fid* is ready, recover forwards the request. If *fid* is not ready, recover simply deletes the state it is keeping for *fid* and sends a successful Rclunk. There is one special case: if *fid* was opened remove-on-close, the *fid* must be reestablished if it is not ready, and recover rewrites the Tclunk into a Tremove when forwarding the request.

Flush. If the request named by *oldtag* has been sent to the remote server on the current connection, the Tflush request must be forwarded. Otherwise, if the connection has been lost since the corresponding request was sent, the request is no longer active: recover frees its internal state and responds with a Rflush.

Special cases

Remove on close. Before forwarding a Topen or Tcreate request, recover removes the remove-on-close (ORCLOSE) bit from the open mode, so that the file will not be removed on connection failure. When forwarding a Tclunk of such a fid, recover rewrites the message into a Tremove to implement the remove-on-close.

Exclusive-use files. Recover does not allow opening of exclusive-use files (those with the QTEXCL qid type bit), to avoid breaking the exclusive-use semantics on such files. This prohibition could be relaxed to simply disallowing reestablishment of exclusive-use fids, but disallowing all access seemed safer, especially in cases such as the mail system where the exclusive-use files are used as locks that protect other, non-exclusive-use files.

Directory reads. 9P imposes restrictions on the offset used in a directory read: it must be zero, to start reading at the beginning of the directory, or it must be the sum of the offset of the last read and the number of bytes returned by that read. Internally, most 9P servers keep an internal representation of where the last read on a fid left off and treat a zero offset as meaning ''start over'' and non-zero offsets as meaning ''continue from last read.'' It is not possible to start a directory read in the middle if the connection is lost and the read must be restarted. `Recover` handles a failed mid-directory read by causing it to start over at the beginning of the directory, duplicating entries that were returned on the previous connection. This behavior is not ideal, but is not worse than the other stateless approaches. Perhaps the best approach would be to read the entire directory to implement a zero-offset read and then serve subsequent reads from a saved copy, but this issue does not arise frequently enough to bother us.

3. Implementation

`Recover` is implemented as two shared-memory procs, one reading local 9P requests (`listensrv`) and one reading remote 9P responses (`listennet`). Both procs can manipulate the per-fid and per-request state and can issue new messages to the remote server; they use a single lock to avoid both running at the same time. Errors writing to the remote server connection are ignored: connection failures are noticed as failed `reads` in `listennet`.

The difficult part of writing a program like `recover` is testing it. One of the authors wrote `recover` in 1999 for the second edition of Plan 9. It worked well for day-to-day use mounting file servers across wide-area internet connections, and was used until 9P2000, the current version of 9P, was introduced in 2001. Even so, it occasionally failed in mysterious ways and was never fully debugged.

To test `recover` after convering it for use in 9P2000, we added a testing flag giving a sequence of deterministic read failures to introduce (for example, `3,5` means that the third read should fail, and then the fifth read after that should fail). These read failures are interpreted by `listennet` as connection failures, causing it to hang up and redial. We tested recover using a small workload that exercised each message, running the workload with all possible combinations of one and two simulated failures. This testing turned up some bugs, but the deterministic framework made them easy to isolate and correct.

4. Performance

`Recover` runs on Plan 9 and also on Linux using the Plan 9 from User Space [3] libraries. We measured `recover`'s performance using the Postmark benchmark. All measurements are shown as the average of 16,384 transactions.

Figure 1 compares the performance of direct file server connections against connections proxied by `recover`, in three different configurations running on Plan 9: a local user-level file server running on the same machine, a file server connected via a 1Gbit/s ethernet, and a file server connected via a 100Mbit/s ethernet. The local connection is bottlenecked mainly by context switching overhead, so introducing a second user level process essentially halves the throughput. The overhead of `recover` is far less noticeable on the ethernets, where local processing is no longer the bottleneck.

Figure 2 shows a similar comparison for Linux, using a local connection and a 1Gbit/s network connection. In this case there are three configurations in each comparison. The first is the Linux kernel mounting a user-level 9P file server directly. The second is the Linux kernel mounting a user-level 9P file server via the Plan 9 from User Space program `srv`, which multiplexes multiple clients onto a single service connection. The third is the Linux kernel mounting the user-level 9P file server via `recover`. On Linux, `recover` posts its 9P service using the `srv` program, so the comparison between the last two configurations isolates the slowdown due to `recover`. The relatively large difference between `srv` and `recover` on the 1Gbit/s connection, especially in light of the relatively small difference on the local connection, is unexplained. Profiling using `oprofile` [8] indicated that most of the time was spent in locking routines, suggesting a performance problem in either the Plan 9 from User Space thread library, but we did not investigate deeply. We did not measure Linux on a 100Mbit/s ethernet connection, nor did we measure `recover` posting a service without `srv`.

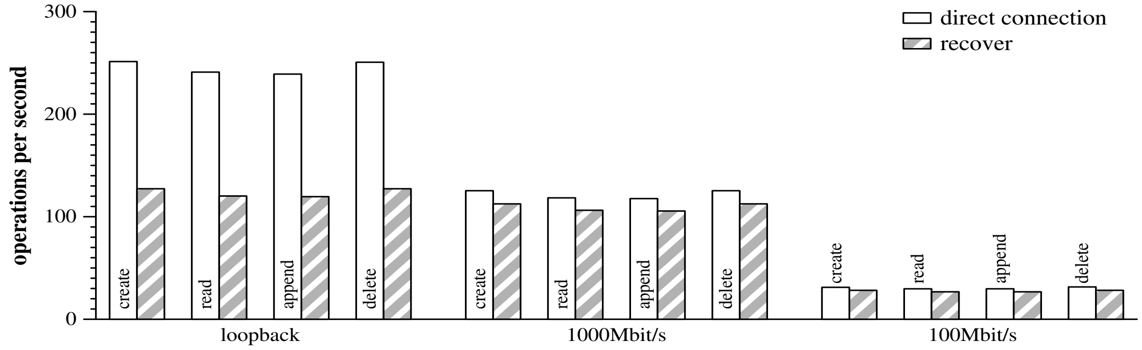

Figure 1: Performance of `recover` compared to direct connection on Plan 9.

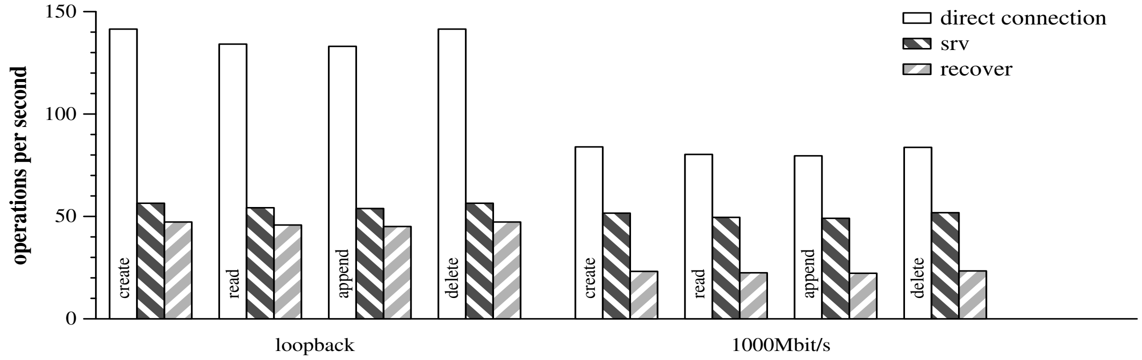

Figure 2: Performance of `recover` compared to `srv` and direct connection on Linux.

Overall, the performance of `recover` seems entirely reasonable in the configurations where it is expected to be used. On Plan 9, using `recover` on a fast network connection incurs approximately a 10% performance penalty. On Linux the penalty is higher, but we suspect performance bugs in other software. Over slower long-distance connections, the performance penalty is hardly noticeable.

5. Discussion

`Recover` is not the first attempt to bring persistent 9P connections to Plan 9.

In the mid to late 1990s, Phil Winterbottom added this capability to a development version of the Plan 9 kernel, but it was never robust enough to be relied upon. Writing `recover` in user space made testing and debugging considerably simpler. Use in high-performance situations might require moving `recover` back into the kernel, but doing so would require finding some way to conduct exhaustive failure testing in order to be robust.

Peter Bosch's `aan` [9] protocol is also used to provide persistent 9P connections. It serves as a lower-level transport protocol just above TCP, using a custom protocol to provide the view of a persistent network connection across multiple TCP sessions. `Aan` is protocol-agnostic but relies on having custom software and persistent state at both sides of the connection. `Recover` depends only on custom software and persistent state on the client side. As a result, `recover` is slightly easier to deploy and is robust even against server failures or restarts. Vic Zandy's TCP racks and rocks [7] also provide persistent network connections, and although they are implemented differently, for the purposes of this discussion they are logically equivalent to `aan`.

Finally, note that `recover` is targeted mainly at traditional disk-based file systems and is not appropriate in all contexts. Synthetic file systems such as `/net` [6] often assign meanings to individual file operations and destroy state when files are closed. On such systems, simply reconnecting and replaying a sequence of opens and walks does not recreate the state at the time of the connection failure. Handling these situations requires failure-aware applications or infrastructure such as Plan B's `/net` [1].

References

[1] Francisco J. Ballesteros, Eva M. Castro, Gorka Guardiola Muzquiz, Katia Leal Algara, and Pedro de las Heras Quiros. ``/net: A Network Abstraction for Mobile and Ubiquitous Computing Environments in the Plan B Operating System. 6th IEEE Workshop on Mobile Computing Systems and Applications, 2004.

[2] Russ Cox, Eric Grosse, Rob Pike, David L. Presotto, and Sean Quinlan. ``Security in Plan 9.'' USENIX Security Symposium, 2002.

[3] Russ Cox. Plan 9 from User Space. `http://swtch.com/plan9port`

[4] Jeffrey Katcher. ``Postmark: a New File System Benchmark.'' Network Appliance Technical Report TR3022, October 1997. `http://www.netapp.com/tech_library/3022.html`

[5] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. ``Plan 9 from Bell Labs.'' Computing Systems, **8**, 3, Summer 1995, pp. 221-254.

[6] Dave Presotto and Phil Winterbottom. ``The Organization of Networks in Plan 9.'' Proceedings of the 1993 USENIX Winter Conference, pp. 43-50.

[7] Victor C. Zandy and Barton P. Miller. ``Reliable network connections.'' Proceedings of the 8th annual International Conference on Mobile Computing and Networking, 2002, pp. 95-106.

[8] OProfile — a System Profiler for Linux. `http://oprofile.sourceforge.net/`

[9] Plan 9 manual. `http://plan9.bell-labs.com/sys/man/5/INDEX.html`