A Cache to Bash for 9P

Geoff Collyer

Bell Laboratories
Murray Hill, New Jersey 07974
geoff@plan9.bell-labs.com

Charles Forsyth

Vita Nuova
www.vitanuova.com
forsyth@vitanuova.com

ABSTRACT

We needed to reduce the load on the file server shared by thousands of nodes in a Blue Gene Plan 9 cluster. Plan 9 had two existing caching mechanisms for 9P file servers: a volatile cache controlled by `devmnt`, and a persistent disk-based cache managed by the user-level server *cfs*. Both sent the server all 9P requests except reads satisfied by the cache. *Cfs* was quickly converted to a *ramcfs* that used the large memory of a Blue Gene I/O node instead of a disk, but most 9P traffic still passed through. A redesign produced *fscfs*, which breaks the direct link between its client's 9P transactions and those it makes to the server, producing a dramatic reduction in traffic seen by the server.

Introduction

As part of a project[1,2] exploring alternatives in distributed systems infrastructure on ultrascale platforms, we ported Plan 9 from Bell Labs[3] to several models of IBM's Blue Gene system. Blue Gene[4] comprises up to 65,536 compute nodes and 1,024 I/O nodes, which are partitioned into processing sets (or *psets*). Each pset has an I/O node providing system services to up to 64 compute nodes. Only the I/O nodes are connected to an external Ethernet. Each node on the Blue Gene/P model has 4 PowerPC processors. Nodes have no permanent storage except a tiny NVRAM. All storage for programs and files is provided by external file servers, accessed by the I/O nodes via Ethernet, and accessible to the CPU nodes only via a specialised network connecting them to the I/O nodes. In our experimental environment, we run Plan 9 throughout, with different configuration and initialisation for CPU nodes and I/O nodes.

Lacking a native Plan 9 file server at the Blue Gene site, we serve files from a Linux server running hosted Inferno.[5] All nodes boot with a built-in file system *paqfs*(4), which has a limited set of files for bootstrap. Each I/O node imported a full file system from the Inferno service on Linux, bound it into a more elaborate name space used by our model for distributed computation,[6] and made that composite name space available to its CPU nodes using *exportfs*(4). Consequently every non-local file access performed

at a CPU node was forwarded by the I/O node to the file server.

It was obvious that having all nodes ultimately send all file service requests directly or indirectly to a single Plan 9 file server would be slow, but we wanted to focus on developing our model of computation, before worrying about making it efficient. Unfortunately, as we ran experiments on even modest configurations, it became clear that the naive structure led to failures, and an unusable system. In particular, the Linux server ran out of file descriptors with only a few hundred nodes, but to be fair even a native file server would run short of some resource eventually. To compensate, we could the replicate the file system across many servers, but all the requests — and the data — would still traverse the same network.

A reasonable alternative structure that would scale with the number of nodes arranges the nodes into a tree. We can avoid transmitting redundant copies of the data by introducing caching into the tree.[7] We can further avoid needless duplication of file system operations across all nodes by caching the results of a given set of operations for later use if that set recurs.

Underlying assumptions and requirements

Plan 9 is running on many thousands of nodes, supporting a few scientific computing programs for a given run. The programs are typically a few megabytes, and increasingly require a collection of supporting system files, not just Plan 9 executables, but the libraries for scripting languages such as Perl or Python. Such shared files are read, not usually written; files that are written are unlikely to be shared concurrently. Write requests are uncommon. Big data files consumed or produced by the application are managed by another service, for instance through channels provided by the clustering software.[6] During both initialisation and subsequent phase changes of an application, system call traces show that similar sets of commands and file accesses are executed on all CPU nodes, sometimes at nearly the same time.

Although initially we would build the caching structure explicitly, the design should allow extension to support a truly hierarchical cache, perhaps using some ideas from Envoy.[7]

9P

First, a quick review of relevant aspects of the 9P protocol. A *file server* in Plan 9 is any program that implements the server side of the file service protocol, 9P.[8] A 9P client sends a request to a 9P server and receives a reply. A 9P server is passive: it generates no message except in response to an explicit request from the client. Each request has a *tag* that numbers the request, an integer *type* that denotes the desired operation, and a set of parameter values for the operation. Parameters can be integers, strings, byte arrays, and structured values such as *Qid* and *Dir*. The position and type of each parameter is completely determined by the operation type. There are no variants. Replies have a similar structure. The reply to a request has a tag equal to the tag of the request, a type derived from the request type, and a set of results that depends on that type. Alternatively, a server can respond to any request with an error message. (Tags allow the server to satisfy I/O requests out of order, although that does not happen here.)

Authentication data is carried as opaque data exchanged using `Tread` and `Rread` requests through a special *fid* established by a `Tauth` request. Thus, only the endpoints need to know the formats and content of the authentication data.

Because the message formats are simple and completely defined, and authentication data is handled cleanly, one can easily write a variety of services that act as intermediaries to 9P conversations. In particular, a caching service can simply interpose itself on a 9P connection. The design of any 9P caching service is driven by considering the desired response to the requests in the protocol, in much the way that a compiler design is driven by the abstract syntax of its language. They are listed in Table 1.

`Tversion tag msize version`	start a new session
`Tauth tag afid uname aname`	optionally authenticate subsequent attaches
`Tattach tag fid afid uname aname`	attach to the root of a file tree
`Twalk tag fid newfid nwname nwname*wname`	walk up or down in the file tree
`Topen tag fid mode`	open a file (directory) checking permissions
`Tcreate tag fid name perm mode`	create a new file
`Tread tag fid offset count`	read data from an open file
`Twrite tag fid offset count data`	write data to an open file
`Tclunk tag fid`	discard a file tree reference (ie, close)
`Tremove tag fid`	remove a file
`Tstat tag fid`	retrieve a file's attributes
`Twstat tag fid stat`	set a file's attributes
`Tflush tag oldtag`	flush pending requests (eg, on interrupt)

Table 1 9P requests

Existing caching support

Plan 9 has long included the cache file system *cfs*, a user-level file server interposed on the connection between a Plan 9 client and a remote file server. It caches file data on a local disk, to reduce latency. It intercepts all 9P exchanges on the connection. Most messages are sent on unchanged, but it adds any file data it sees to the cache, and satisfies file reads from the cache if possible. The cache is write-through, so the cached data is never more recent than the server's. Files and file data are both kept on a least-recently-used basis, up to the size of the disk partition allocated to the cache. *Cfs* uses the Qid value returned by each `open` to detect and discard out-of-date cached data. The cache persists on a disk partition between boots, but because the cache is write-through it can simply be reformatted if invalid.

In the Fourth Edition, Plan 9 acquired an optional kernel-level cache. It is an optional kernel component, and must be explicitly enabled by the MCACHE option of the `mount` system call. If present and enabled, it will cache data from files served by that mount. Unlike *cfs*, the cache resides in physical memory pages, and the cache is lost on reboot. Otherwise it is similar: client reads are satisfied from the cache if possible; data exchanged with the server through read and write will be added to the cache; files and data are managed least-recently-used; and out-of-date cache entries are detected using the Qid value returned by `open`. The cache pages are only briefly mapped into the kernel address space when accessed, reducing the pressure on the kernel address space, and only a limited amount of data per file can be cached, to help constrain the physical memory dedicated to the cache.

Ramcfs

Ramcfs was the first attempt at mitigating the problem. It was created by changing a copy of *cfs* to use a region of memory as its cache rather than a region of disk. Unlike *cfs*, *ramcfs* initialises the cache each time it starts, since there is no persistent storage. We added a `-r` option that declares that all the files are unchanging (readonly) and thus may be cached more aggressively. On boot, each IO node would first insert *ramcfs* on its connection to the central file server. Thus, the subsequent access by the I/O node's CPU nodes would pass through the cache on the I/O node.

Ramcfs had its own disadvantages. It always reserved a big chunk of memory, to make a virtual block device. Since the cache runs on I/O nodes not CPU nodes, that does not limit the memory available to a computation, but it was still a bit of a hack. More seriously, because *ramcfs* was based on *cfs* it also forwarded all requests involving metadata to the remote server.

Fscfs

Existing components were inadequate, so we implemented a new one, *fscfs* that not only caches data, but reduces the number of operations seen by the file server. Like *cfs* and *ramcfs*, it acts as a write-through cache for data, but when many processes are making identical file system requests over time, it effectively aggregates them into single requests at the server. This has a dramatic effect. Table 2 shows statistics measured at a single IO node. (The values on the other IO nodes in the run were similar.) The interval is from the initial connection to the file server up to the point where the CPU nodes had started all their network services and were ready for duty. The table gives the number of various file system operations seen by the IO node, as produced by its 64 CPU node clients, and the number of operations it had to send on to the server. The final row shows the number of bytes read by all clients, and the number of bytes read from the server and cached at the IO node to satisfy the client read requests. The difference is dramatic. The reduction in load seen by the server will also be magnified by the addition of each new IO node and its cluster.

Op	*IO node*	*Server*
`Tversion`	1	1
`Tattach`	1	1
`Twalk`	7,855	56
`Topen`	1,486	77
`Tread`	6,823	133
`Tclunk`	4,749	0
`Tstat`	4,224	4,224
bytes read	19,913,992	462,722

Table 2 Statistics from *fscfs* on an IO node with 64 CPU nodes.

Design and implementation

To provide data caching, *fscfs* uses the same strategy as the existing caches: it makes a copy of data as it passes through in either direction, and stores it in a local cache. Subsequent `Tread` requests for the same data will be satisfied from the cache. `Twrite` requests are passed through to the server, replacing data in the cache if successful. Cached data is associated with each active file, but the memory it occupies is managed

on a least-recently-used basis across the whole set of files. When a specified threshold for the cache has been reached, the oldest cached data is discarded.

Unlike the existing caches, however, *fscfs* handles more 9P requests itself without delegating them to the server. Those requests include `Twalk`, `Topen`, `Tstat` and `Tclunk`.

To do that required two significant changes from previous schemes. We explain them by reference to several internal data types, shown concisely below:

```
Fid    ::   fid: u32int  qid: Qid  path: Path opened: SFid  mode: (R | W | RW)
SFid   ::   fid: u32int

Path   ::   name: string  qid: Qid  parent: Path  kids: set of Path  (Valid | Invalid)
Valid  ::   sfid: SFid  file: optional File
Invalid ::  reason: string

File   ::   open: array of SFid  dir: Dir  clength: u64int  cached: sparse array of Data
```

First, *fscfs* separates its client from the server, by managing two sets of fids. One set is allocated by its client, as before; those are seen only by *fscfs*, which remembers them in values of the Fid type. The other set of fids is allocated and controlled by *fscfs*; only those fids are seen by the server. It records them in SFid values. Second, using the distinction between fid sets, *fscfs* caches the results of walks and opens. The distinction allows a cached fid (known to the server) to outlive a fid allocated by the client. It also allows several client fids to share a single server fid. That alone reduces the resource requirements at the server as the number of client references grows.

From each `Tattach` referring to a file server's tree, *fscfs* grows a *Path* tree representing all the paths walked in that tree, successfully or unsuccessfully. A successful walk results in an end-point that records the SFid referring to that point in the server's hierarchy. (Note that intermediate names need not have server fids.) If a walk from a given point failed at some stage, that is noted by a special Path value at that point in the tree, which gives the error string explaining why the walk failed. If a subsequent `Twalk` from the client retraces an existing Path, *fscfs* can send the appropriate response itself, including failures and walks that were only partially successful. If names remain in the walk request after traversing the existing Path, *fscfs* allocates a new SFid for the new Path end-point, sends the whole request to the server, and updates the Path appropriately from the server's response. Remembering failures is a great help when, for instance, many processes on many nodes are enumerating possible names for different possible versions of (say) Python library files or shared libraries, most of which do not exist. (It would be more rational to change the software not to do pointless searches, but it is not always possible to change components from standard distributions.)

When a file or directory has been opened, the corresponding Path will have a *File* structure that has the SFid(s) for each mode (read, write, read/write) with which a client has opened the file, the currently known file length, and any data cached for that file. The File's SFid is distinct from that in the Path because the latter might later be walked elsewhere, and it is illegal to walk an open fid. Furthermore, files can anyway be opened simultaneously with different modes, each needing a distinct fid on the server to carry the correct mode. Files in Plan 9 can also be opened 'remove-on-close' (ORCLOSE). As a special case, the client Fid for such a file will be given its own unique SFid on open, to ensure that the timing of the remove remains the same from the client's point of view. The file will be removed when that particular client closes it.

Some requests update the file system: `Twrite`, `Tcreate`, `Topen` (with OTRUNC), `Twstat`, and `Tremove`. Those are delegated to the server, and on success the local state is updated to match. In other words, the Path and File caches are write-through. For instance, a successful `Twrite` request will add the data written to the cache. A successful `Tcreate` will extend the current Path tree, possibly replacing a previous Invalid entry for the newly-created name, and then open a new SFid for the file. A more subtle case is `Tremove`, which always frees any existing *sfid* for the file, as required by *remove*(5), but only on success does it mark the file as *removed* in the Path tree and discard any cached data.

Fscfs always delegates operations on certain types or classes of files to the server, specifically authentication files, append-only, and exclusive-use files. Currently it also delegates `Topen` and `Tread` for directories, because *read*(5) does not allow a seek in a directory except to the start. We are currently changing the software to cache the whole directory on the first read, so that directory reading does not provoke excessive load on the file server. The reader might have noticed that `Tstat` was also not handled locally when the measurements in Table 2 were made.

Discussion and further work

There is no great mystery about implementing a cache for a file system, whether directly in a kernel, as with the UNIX buffer cache, or by intercepting a file service protocol, as with NFS or 9P. Even so, there are typically some subtle points.

Fscfs is just over 2,000 lines of C code, including some intended for future use. It has more than satisfied our initial requirements, although much more can be done. It aggregates common operations in a general but straightforward way. Its Path cache is similar to Matuszek's file handle cache in his NFS cache,[9] and closer to 9P home, some of the subtle cases in fid handling in *fscfs* seem to turn up in the implementation of 9P in the Linux kernel, where Linux lacks anything corresponding exactly to a fid.[10]

Currently, as is true in our environment, *fscfs* assumes that the client requests always represent the same user. Removing that assumption requires growing separate Path trees from distinct `Tauth` and `Tattach` instances, to ensure that errors (eg, 'permission denied') are reported in the correct context. Instead of having a Path refer directly to a File, the Path would record only the Qid, and the File would be found in a Qid to File map, to ensure cached data is correctly shared and updated.

More interesting additions would be to make our file system access fault-tolerant and more efficient. Given the structured nature of our target networks and systems, one attractive approach is to use a variant of random trees, which could also provide load spreading.[11]

References

1. Ronald G Minnich, Matthew J Sottile, Sung-Eun Choi, Erik Hendriks, Jim McKie, ''Right-weight kernels: an off-the-shelf alternative to custom light-weight kernels,'' *ACM SIGOPS Operating Systems Review* **40**(2), pp. 22-28 (April 2006).

2. Ron Minnich, Jim McKie, Charles Forsyth, Latchesar Ionkov, Andrey Mirtchovski, Eric Van Hensbergen, Volker Strumper, *Rightweight Kernels*, http://www.cs.unm.edu/~fastos/07meeting/usenix-june2007.pdf.

3. Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, Phil Winterbottom, ''Plan 9 from Bell Labs,'' *Computing Systems* **8**(3), pp. 221-

254 (Summer 1995).

4. A Gara, M A Blumrich, D Chen, G L-T Chiu, P Coteus, M E Giampapa, R A Haring, P Heidelberger, D Hoenicke, G V Kopcsay, T A Liebsch, M Ohmacht, B D Steinmacher-Burow, T Takken, P Vranas, ''Overview of the Blue Gene/L system architecture,'' *IBM Journal of Research and Development* **49**(2-3), pp. 195-212 (2005).

5. *The Inferno Programmer's Manual,* Vita Nuova Holdings Limited (2000).

6. Eric Van Hensbergen, Noah Paul Evans, Phillip Stanley-Marbell, ''A unified execution model for cloud computing,'' *ACM SIGOPS Operating Systems Review* **44**(2), pp. 12-17 (April 2010).

7. R G Ross, ''Cluster storage for commodity computation,'' UCAM-CL-TR-690, University of Cambridge Computer Laboratory (June 2007).

8. C H Forsyth, ''The Ubiquitous file Server in Plan 9,'' *Proceedings of the Libre Software Meeting*, Dijon, France (2005).

9. Stephen Matuszek, *Effectiveness of an NFS cache*, http://matuszek.net/development/distributed/nfscache/index.html.

10. Eric Van Hensbergen, Ron Minnich, ''Grave Robbers from Outer Space: Using 9P2000 Under Linux,'' *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, pp. 83-94, USENIX (2005).

11. David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, Daniel Lewin, ''Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web,'' *Proceedings of the twenty-ninth annual ACM symposium on Theory of Computing*, El Paso, Texas, pp. 654-663 (1997).