Managing Concurrency in Distributed Systems

Pierre Jansen
Sape Mullender

Slides and program sources are on `plan9.bell-labs.com/who/sape`
If you want to try out the Plan 9 distribution, look on `plan9.bell-labs.com`

The Playlist File System

Today, we'll talk about the *file-server* part of the Playlist File System. After all, that's what makes it a file system.

Then, we'll review the overall design of the system and see what the alternatives could be.

What is 9P?

A protocol for accessing an object hierarchy.

Like Unix file systems

" rwxrwxrwx protections

" directories read like files

Unlike Unix

" no symbolic links

104 bit unique *qid* identifies objects

" 8 bit type

" 64 bit unique path

" 32 bit version

The version is changed whenever the object is written. By convention, it is incremented.

Protocol Fundamentals

Not stateless

- server and client each maintain state
- recovery after hangup is possible though not easy

The state is indexed by a *fid* (file id) chosen by the client.

- reference to an object
- open state
- user/access rights

Most messages contain a *fid*.

Protocol Fundamentals

Request/reply semantics

" server never sends anything unrequested

" replies do not have to be immediate

" replies do not have to be in order

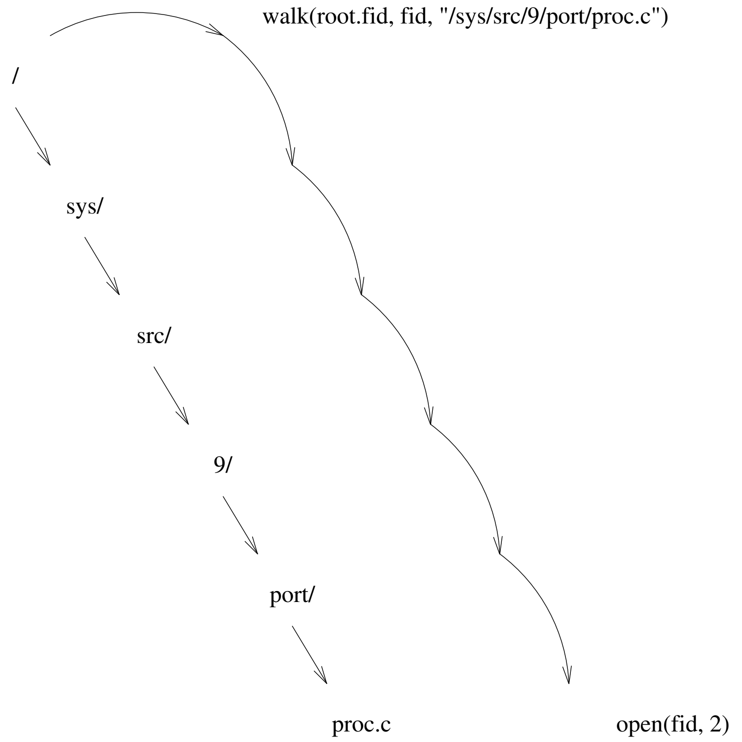
A *tag* matches requests to replies. All messages contain a *type* and a *tag*.

The Requests

Tversion	exchange version (9P2000) and maximum message size
Tsession	exchange session state
Tattach	attach to a root
Twalk	walk (and clone) a fid
Topen	
Tcreate	
Tread	
Twrite	
Tclunk	discard fid state
Tremove	
Tstat	read a directory entry
Twstat	write a directory entry
Tnop	do nothing

The Dance

open("/sys/src/9/port/proc.c", ORDWR);



Client side

Little to do; use one of several commands to get a 9P link to a resource and mount it into your namespace.

srv dial a connection and mount with incremental auth

import dial a connection, authenticate, and then mount with no authentication

mount mount an established connection

Authentication

On each mount user authentication occurs via the auth or attach message:

size[4] **Tauth** *tag*[2] *afid*[4] *uname*[s] *aname*[s]
size[4] **Rauth** *tag*[2] *aqid*[13]

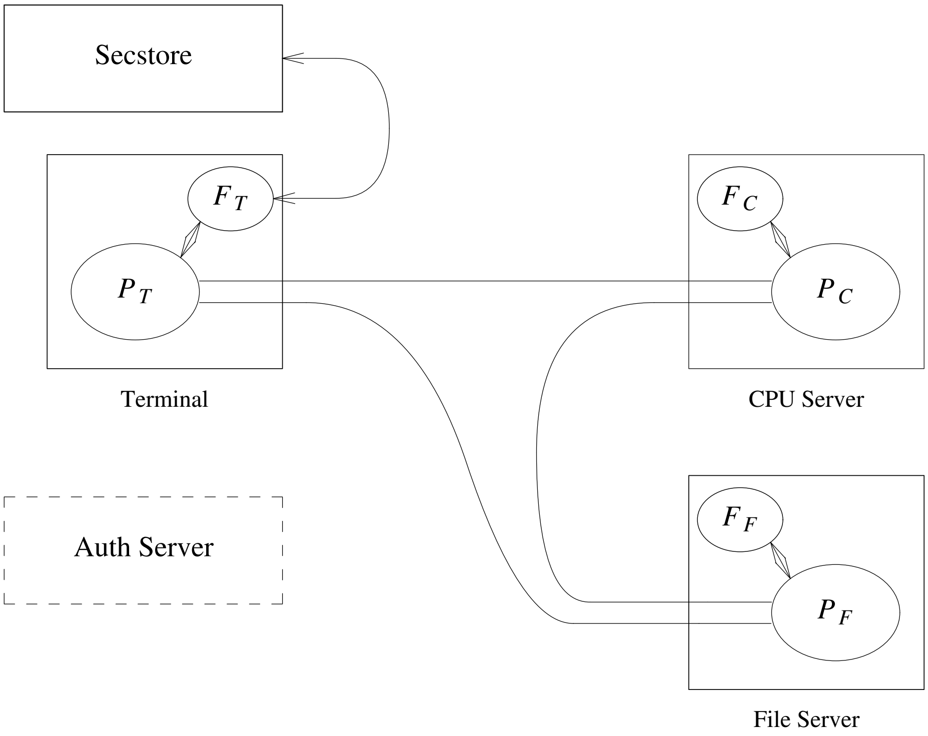
size[4] **Tattach** *tag*[2] *fid*[4] *afid*[4] *uname*[s] *aname*[s]
size[4] **Rattach** *tag*[2] *qid*[13]

The authentication protocol itself is outside the 9P protocol. This allows different authentication algorithms to be used and it allows update of the authentication mechanism without changing 9P.

Afid refers to an unnamed *authentication file* that is used to exchange the challenge/response messages in the authentication protocol.

At both ends it is *plumbed* into a per-machine authentication server called *Factotum.*

Factotum


Secstore
F_T
P_T
Terminal
F_C
P_C
CPU Server
Auth Server
F_F
P_F
File Server

Marshaling

9P messages:

```
typedef struct Fcall Fcall;

int     convM2S(char*, Fcall*, int);
int     convS2M(Fcall*, char*);
```

Directory entries for *read*, *stat*, and *wstat*:

```
typedef struct Dir Dir;

int     convM2D(char*, Dir*);
int     convD2M(Dir*, char*);
```

How to start a server

Three ways:

" started by *aux/listen*(8) with stdin/stdout for I/O (e.g. *cpu*(1))

```
#!/bin/cpu -R
```

" started from a shell and self mounting (e.g. *ramfs*(4) or *usbaudio*(4))

" started from a shell and posted in `/srv` (e.g. *playlistfs*(7))

Posting into /srv

```
void
post(char *name, int sfd)
{
	int fd;
	char buf[32];

	fd = create(name, OWRITE, 0666);
	if(fd < 0)
		return;
	sprint(buf, "%d", sfd);
	if(write(fd, buf, strlen(buf)) != strlen(buf))
		sysfatal("srv write: %r");
	close(fd);
}

if(pipe(srvfd) < 0)
	sysfatal("pipe failed: %r");
procrfork(srv, nil, 8192, RFFDG);
close(srvfd[0]);	/* don't deadlock if child fails */
playinit();
srvfile = smprint("/srv/playlist.%s", srvpost);
post(srvfile, srvfd[1]);
free(srvfile);
if(mount(srvfd[1], -1, mntpt, MBEFORE, "") < 0)
	sysfatal("mount failed: %r");
```

Main Loop, react to requests

```
fmtinstall('F', fcallconv);
r = reqalloc();
for(;;){
	n = read9pmsg(srvfd[0], r->indata, messagesize);
	if(n == 0)
		continue;
	if(n < 0){
		rerrstr(e, sizeof e);
		if (strcmp(e, "interrupted") == 0)
			continue;
		sysfatal("srvio: %s", e);
	}
	if(convM2S(r->indata, n, &r->ifcall) == 0)
		continue;

	if(debug & DbgFs)
		fprint(2, "io:<-%F0, &r->ifcall);
	sendp(dispatchc, r);
	r = reqalloc();
}
```

Worker, execute requests

```
void
work(Worker *w)
{
	Req *r;
	char *err;
	int n;

	r = w->r;
	r->ofcall.data = (char*)r->indata;
	if(!fcalls[r->ifcall.type])
		err = "bad fcall type";
	else {
		r->fid = newfid(r->ifcall.fid);
		err = (*fcalls[r->ifcall.type])(w);
	}
	...
}
```

Main Loop definitions

```
struct Req
{
        uchar   indata[Messagesize];
        uchar   outdata[Messagesize];
        Fcall   ifcall;
        Fcall   ofcall;
        Fid*    fid;
};

char  *(*fcalls[])(Worker*) = {
        [Tflush]        rflush,
        [Tversion]      rversion,
        [Tauth]         rauth,
        [Tattach]       rattach,
        [Twalk]         rwalk,
        [Topen]         ropen,
        [Tcreate]       rcreate,
        [Tread]         rread,
        [Twrite]        rwrite,
        [Tclunk]        rclunk,
        [Tremove]       rremove,
        [Tstat]         rstat,
        [Twstat]        rwstat,
};
```

Fulfilling a request

```
char*
rwalk(Worker *w)
{
	Fid *f, *nf;
	char *rv;
	int i;
	File *savefile;
	Req *r;

	r = w->r;
	f = r->fid;
	if(f->flags & Open)
		return Eisopen;

	r->ofcall.nwqid = 0;
	nf = nil;
	savefile = f->file;

	/* clone if requested */
	if(r->ifcall.newfid != r->ifcall.fid){
		nf = doclone(f, r->ifcall.newfid);
		if(nf == nil)
			return "new fid in use";
		f = nf;
	}

	/* if it's just a clone, return */
	if(r->ifcall.nwname == 0 && nf != nil)
		return nil;

	/* walk each element */
```

Fulfilling a request

```
        rv = nil;
        for(i = 0; i < r->ifcall.nwname; i++){
                rv = dowalk(f, r->ifcall.wname[i]);
                if(rv != nil){
                        if(nf != nil)
                                nf->flags &= ~(Open|Busy);
                        else
                                f->file = savefile;
                        break;
                }
                r->ofcall.wqid[i] = f->file->dir.qid;
        }
        r->ofcall.nwqid = i;

        /* we only error out if no walk  */
        if(i > 0)
                rv = nil;
        return rv;
}
```

Replying

```
void
work(Worker *w)
{
	Req *r;
	char *err;
	int n;

	r = w->r;
	r->ofcall.data = (char*)r->indata;
	if(!fcalls[r->ifcall.type])
		err = "bad fcall type";
	else {
		r->fid = newfid(r->ifcall.fid);
		err = (*fcalls[r->ifcall.type])(w);
	}
	if(err){
		r->ofcall.type = Rerror;
		r->ofcall.ename = err;
	}else{
		r->ofcall.type = r->ifcall.type + 1;
		r->ofcall.fid = r->ifcall.fid;
	}
	r->ofcall.tag = r->ifcall.tag;
	if(debug & DbgFs)
		fprint(2, "io:->%F0, &r->ofcall);/**/
	n = convS2M(&r->ofcall, r->outdata, messagesize);
	if(write(srvfd[0], r->outdata, n) != n)
		sysfatal("mount write");
	reqfree(r);
	w->r = nil;
}
```

Waiting for change ...

The *version* of the Qid changes when the file changes

```
char*
rwrite(Worker *w)
{
	...
	if(f->file == &files[Qplayctl]){
		nf = tokenize(r->ifcall.data, fields, 4);
		if (nf == 0){
			r->ofcall.count = r->ifcall.count;
			return nil;
		}
		if (nf == 2)
			i = strtol(fields[1], nil, 0);
		else
			i = playstate.off;
		newstate = playstate;
		switch(lookup(fields[0], statetxt)){
		case Play:
			...
		}
		if (newstate.m != playstate.m)
			sendul(playc, newstate.m);
		f->file->dir.qid.vers++;
	} else
	...
	return nil;
}
```

Checking for change ...

- Associate a subscriber queue with each object whose change may be waited for.
- Waiting workers append themselves to that queue, then block on reception of an event
- When object changes, each worker on queue receives a change notification
- Race conditions? Not here, all threads involved share a proc and are therefore serially scheduled (no two threads run concurrrently)
- In an OS Kernel where you have more than one processor and/or interrupts?

  Here it’s much harder: you need to

  1. Check for wakeup condition
  2. Put yourself on queue
  3. Put yourself to sleep awaiting wakeup

  All in one, *atomic* operation to prevent losing a wakeup call. (We’ll get back to this).

Checking for Change in the Playlist Server

```
char*
rread(Worker *w)
{
	...
	m.cmd = 0;
	while(f->vers == f->file->dir.qid.vers && (f->flags & Eof)){
		/* Wait until file state changes (f->file->dir.qid.vers is incremented) */
		sendp(f->file->workers, w);
		recv(w->eventc, &m);
	}
	if(f->file == files + Qplayctl){
		f->flags &= ~Eof;
		if(m.cmd == Error){
			snprint(r->ofcall.data, sizeof r->indata, "%s %ud %q",
				statetxt[m.cmd], m.off, m.arg);
			free(m.arg);
		}else if(f->vers == f->file->dir.qid.vers){
			r->ofcall.count = 0;
			/* Arrange for this fid to wait next time: */
			f->flags |= Eof;
			return nil;
		}else{
			snprint(r->ofcall.data, sizeof r->indata, "%s %ud",
				statetxt[playstate.cmd], playstate.off);
			f->vers = f->file->dir.qid.vers;
		}
		r->ofcall.count = strlen(r->ofcall.data);
		if(r->ofcall.count > r->ifcall.count)
			r->ofcall.count = r->ifcall.count;
	}else ...
	return nil;
}
```

Asynchrony and flush...

Because life just isn’t fair.

Some requests may take a LONG time.

Client may go away before server has replied. Client’s kernel will generate a *flush* request.

If the client’s kernel dies, by the way, the *pipe* will break;

Flushing a request

```
char*
rflush(Worker *w)
{
	Wmsg m;
	int i;
	Req *r;

	r = w->r;
	m.cmd = Flush;
	m.off = r->ifcall.oldtag;
	m.arg = nil;
	for(i = 1; i < nelem(files); i++)
		bcastmsg(files[i].workers, &m);
	return 0;
}
```

Broadcast to all waiters on a queue

```
void
bcastmsg(Channel *q, Wmsg *m)
{
	Worker *w, marker;
	void *a;

	a = m->arg;
	/*
	 * Use a marker to mark the end of the queue.
	 * This prevents workers from getting the
	 * broadcast and putting themselves back on the
	 * queue before the broadcast has finished
	 */
	marker.eventc = nil;	/* Only markers have eventc == nil */
	sendp(q, &marker);
	while((w = recvp(q)) != &marker){
		if(w->eventc == nil){
			/* somebody else's marker, put it back */
			sendp(q, w);
		}else{
			if(a) m->arg = strdup(a);
			send(w->eventc, m);
		}
	}
	free(a);
	m->arg = nil;
}
```

Reacting to Flush

```
char*
rread(Worker *w)
{
	...
	m.cmd = 0;
	while(f->vers == f->file->dir.qid.vers && (f->flags & Eof)){
		/* Wait until file state changes (f->file->dir.qid.vers is incremented) */
		sendp(f->file->workers, w);
		recv(w->eventc, &m);
		if(m.cmd == Flush && m.off == r->ifcall.tag)
			return (char*)~0;	/* no answer needed */
		assert(m.cmd != Work);
	}
	...
}
```

Reacting to Flush

```
void
work(Worker *w)
{
	...
	err = (*fcalls[r->ifcall.type])(w);
	if(err != (char*)~0){	/* ~0 indicates Flush received */
		if(err){
			r->ofcall.type = Rerror;
			r->ofcall.ename = err;
		}else{
			r->ofcall.type = r->ifcall.type + 1;
			r->ofcall.fid = r->ifcall.fid;
		}
		r->ofcall.tag = r->ifcall.tag;
		n = convS2M(&r->ofcall, r->outdata, messagesize);
		if(write(srvfd[0], r->outdata, n) != n)
			sysfatal("mount write");
	}
	reqfree(r);
	w->r = nil;
}
```

Kernel sleep and wakeup

```
/*
 *  sleep if a condition is not true.  Another process will
 *  awaken us after it sets the condition.  When we awaken
 *  the condition may no longer be true.
 */
void
sleep(Rendez *r, int (*f)(void*), void *arg)
{
        ...
}

/*
 *  Expects that only one process can call wakeup for any given Rendez.
 *  We hold both locks to ensure that r->p and p->r remain consistent.
 *  Richard Miller has a better solution that doesn't require both to
 *  be held simultaneously, but I'm a paranoid - presotto.
 */
Proc*
wakeup(Rendez *r)
{
        ...
}
```

Sleep

```
void
sleep(Rendez *r, int (*f)(void*), void *arg)
{
	int s;

	s = splhi();
	lock(r);
	lock(&up->rlock);
	/*
	 *  Wakeup only knows there may be something to do by testing
	 *  r->p in order to get something to lock on.
	 */
	r->p = up;
	if((*f)(arg) || up->notepending){
		/* never mind */
		r->p = nil;
		unlock(&up->rlock);
		unlock(r);
	}else{
		/* committed to change state and call scheduler */
		up->state = Wakeme;
		up->r = r;
		procsave(up);
		if(setlabel(&up->sched)) {
			/* here when the process is awakened */
			procrestore(up);
			spllo();
		}else{
			/* here to go to sleep (i.e. stop Running) */
			unlock(&up->rlock);
			unlock(r);
			gotolabel(&m->sched);
		}
	}
	splx(s);
}
```

Wakeup

```
Proc*
wakeup(Rendez *r)
{
	Proc *p;
	int s;

	s = splhi();

	lock(r);
	p = r->p;

	if(p != nil){
		lock(&p->rlock);
		if(p->state != Wakeme || p->r != r)
			panic("wakeup: state");
		r->p = nil;
		p->r = nil;
		ready(p);
		unlock(&p->rlock);
	}
	unlock(r);

	splx(s);
	return p;
}
```

Kernel

- Tricky to get right. It took many iterations.
- You need two locks. Imagine:

  P1:`lock(a); lock(b)`

  P2:`lock(b); lock(a)` We always `lock(&rendez)` first!
- In user land, this is where a threads package like ours pays off. You just don't have to worry about races and deadlocks like this. Even so, enough complexity remains.

So, How Well Did We Do?

- We have a lockless playlist file system
- Code looks clean and understandable (or as understandable as something this complicated can be)
- `Channels` serve a plethora of functions:
  - simple queues
  - producer/consumer arrangements (using two channels, `full` and `empty`)
  - help in dispatching arrangements (using a channel of `Worker` structures and an event channel inside each)
  - each of these mechanisms requires only one or two lines of code to use
- Threads and Procs are used to keep the code organized in well-defined pieces (but deciding on the division of labor in threads and procs requires careful thought it's pretty much the only hard part of the design).

Final Remarks

- Programming is a *craft* much more than it is a *science*
- You learn it by reading and understanding well-written code written by others
- You master it by doing it yourself
- Read and understand the playlist server code
- Or, install Plan 9 on a partition on your disk (or run it directly from CD-ROM) and peruse the Plan 9 sources. Even though it is not as application rich as Linux, it is the best example of the craft of code engineering in systems today

Complex problems never have simple solutions

There is a plethora of *tools* that claim to deal with complexity.

" They don't work

If a problem is complicated, then you cannot merely apply some tool to make that complexity go away.

Tools such as Rose RT, Corba/Tina, Object Orientation, etc. *never* make a complicated problem simpler.

Object Orientation

- O yeah, what about `C++` and object-orientation?
  - Encapsulating functionality and exposing only interfaces is a fine idea one we practise all the time. And a language that makes this easier is a fine idea too.
  - But *object orientation* has become something of a religion and that is no good
  - `C++`, especially, has some really nasty features (templates, overloading!) that, when improperly used, make code utterly unmaintainable
  - ‘When your hammer is `C++`, everything begins to look like a thumb’ [Steve Hoflich]
  - And what about code reuse? Well, I haven’t seen any evidence at all of increased code reuse in object-oriented language environments. I’ve seen some to the contrary ...
  - And, as you can see in our code, conventional programming languages are just as good at exposing only interfaces and hiding implementation and we can choose to do it only where appropriate

Speaking of Code Reuse

- Something of a holy grail in software engineering
- What code gets reused in practice?
  - Simple functions: `malloc`, `free`, `sin`, `cos`, `strcmp`, `memmove`
  - Complete programs:

```
amstel% cat /rc/bin/kill
#!/bin/rc
rfork e
U=`{cat /dev/user}
for(i){
	ps | sed -n '/^'$U' .* '$i'$/s%[^ ]*  *%~>/proc/%
	s%  *.*%/note%
	s%~%echo kill%p'
}
```

  - But nothing in between. Why?

Because it's too hard. Complex modules have complex sometimes hidden interfaces: memory management & garbage collection, error reporting, data representation.

Three Models for Code Development

- **Traditional: Hierarchical**
  Most non-systems software gets written this way. Teams can be very large: hundreds of programmers. Result tends to be bloated, inefficient, but it often works (in the end). Read *'The Mythical Man-Month'* by Frederick Brooks!

  An elephant is a mouse built to a specification.
  Bosses never realize that the bottle neck is always at the top.

- **Open Source: Nobody in Charge**
  (disclaimer: not all open source software is built this way). If there are enough contributors, this works surprisingly well although at what cost cannot be measured. But there are no pleasant surprises:

  The only thing original about Linux is the Penguin.
  '***Gcc*** is not the the *Holy Grail* of computing it is the *Holy Cow*' [forsyth].
  'The X server has to be the biggest program I've ever seen that doesn't do anything for you' [ken].
  'Software has gotten too fat and unreliable, so we started with Linux' [Nicholas Negroponte]

- **Extreme Programming: Small Teams**
  All original systems were built this way. Allows creative ideas to come to fruition; requires a willingness to 'throw one away' and start over (but that's much less work than most think). Basic method is to code 'the next thing you need', deal with unusual cases when they occur.

Another Perspective

- **Top Down Design**
  Hierarchical. Specification gets written and this is then implemented
- **Bottom Up Design**
  Open Sources. There is no specification at all. Contributers contribute random pieces
- Where does that leave 'Extreme Programming'?
  It's really Top Down with Feedback: You learn from writing pieces of code that need to be written anyway and, as you learn, you learn what the overall design should be. And, you are always *prepared to throw one away, because you will anyhow* [Fred Brooks].

The Good News and the Bad News

- Programming is hard and there is no software engineering method that makes a complex problem simple.
- But that’s what makes it fun and that’s what gives people like you and me jobs.

Slides

I'll put them on

```
plan9.bell-labs.com/who/sape
```

Follow the appropriate link.

And remember, learn by trial and error: you'll never get it right on the first attempt. Use the experience to redesign where necessary.

It's still quicker than trying to get a specification correct before you start coding.