Managing Concurrency in Distributed Systems

Pierre Jansen
Sape Mullender

Slides will be on `plan9.bell-labs.com/who/sape`

Who am I

Sape Mullender

- Studied at the Vrije Universiteit, PhD 1985
- Project Leader at CWI, the Centre for Mathematics and Computer Science, until 1991
- Full Professor, Universiteit Twente, until 1998, part-time since then
  'A professor is one who talks in someone else's sleep'
- Researcher at Lucent Technologies Bell Laboratories (Alcatel-Lucent since December 1st)
- Worked on four distributed systems: Amoeba (VU), Nemesis (Twente), Inferno and Plan 9 (Bell Labs)
- Now working on distributed wireless base stations

The UNIX Room

In this room, UNIX was first built in the late sixties by Ken Thompson and Dennis Ritchie. Ken wrote most of the operating system, Dennis designed the C language and built the compiler.

In the late eighties and early nineties, Plan 9 from Bell Labs was developed here as the natural successor of Bell-Labs Unix.

By that time, Unix had been adopted as the default operating system by computer science departments and research institutions world wide. It had many warts implanted into it: sockets for networking, the X window system, pseudo TTYs, *ioctl* and *select.* There was BSD Unix, System V Unix, Mach, Ultrix, HPUX, Solaris, ï

Computing Sciences in the Plan 9 Days

Principal workers on Plan 9 were Sean Dorward (compilers, audio), Bob Flandrena (mail), Eric Grosse (security), Jim McKie (drivers, kernel), Rob Pike (graphics, editors), Dave Presotto (mail, networking), Sean Quinlan (file systems), Dennis Ritchie, Ken Thompson (file systems, C compiler), Howard Trickey (Posix compatibility, graphics), Phil Winterbottom (kernel, debugging).

What Makes Distributed Applications and Services Hard to Build?

- Unbridled *ambition*
  (See Saltzer's slides on the Web site)
- *Indeterminism*
- *Fault Tolerance*

So what causes indeterminism?

Indeterminism

Basically caused by having interaction between concurrent activities.

Timing of these activities is dictated by the environment:

- when requests arrive
- where data sits on a disk
- how reliable the network is (especially wireless)
- local schedulers, activities in other applications

Indeterminism exists in local applications too: most programs with a GUI are indeterministic to some degree: next event may be mouse click, key stroke, file operation, or computation completing.

Servers are usually indeterministic: requests arrive, I/O and computations complete.

This Course

How to build such applications is the subject of this course. We won't bother you much with theory theoreticians never got a big system to work.

Programming isn't science, it's engineering. It's a *craft*.

What Makes a Craftsman?

How did Stradivarius make such great violins? Or how did Rembrandt make such great paintings?

Three ingredients:

1. **Apprenticeship**
   You need to learn from a master
2. **Knowledge**
   You need to know everything about the craft
3. **Practice**
   You need to learn to apply that knowledge by trial and error

In this class you'll see a lot of 2., a little of 1., and none of 3. Practise you'll have to do by yourself.

So How will You Become a Master?

Simply by addressing all three components.

1. **Apprentice**
   Hard when there are no masters about. But studying excellent code written by masters is a good substitute. How to recognize good code? You'll know it when you see it (but you won't if you never see it; despite all the hype, Linux contains few examples of good code).
2. **Practise**:
   Design and build serious applications, learn from your mistakes.
   Prepare to throw one away; you will anyhow *[Fred Brooks, 1986]*

Read the Lions' book (see web site), study Plan 9 source code.

Example Application

We'll focus on one example to illustrate the issues in design and implementation of a concurrent and distributed application.

The application has a real-time component, a graphical user interface, a database and a large collection of files under its control.

The GUI can be used to control the application from multiple locations.

What we do not address here is fault tolerance: neither database nor system state are replicated. Nevertheless, the application presents plenty of interesting issues.

Overall Design

The hardest problem. A good architect can take enough steps back to see the *greater picture*.

An operating-system architect, for example, must see

- how machine dependency, file systems, security, memory protection, graphics, system-call interface, etc. interact
- what modules to define so that each module has a well-defined task, yet module interfaces allow the system to operate efficiently,
- structure modules and module interfaces so that the system can be tested piecemeal and bugs can be tracked down and resolved.

This is hard; it takes experience and it is often achieved by trial and error. Obviously, the more systems of similar ilk you've designed, the better you get at it.

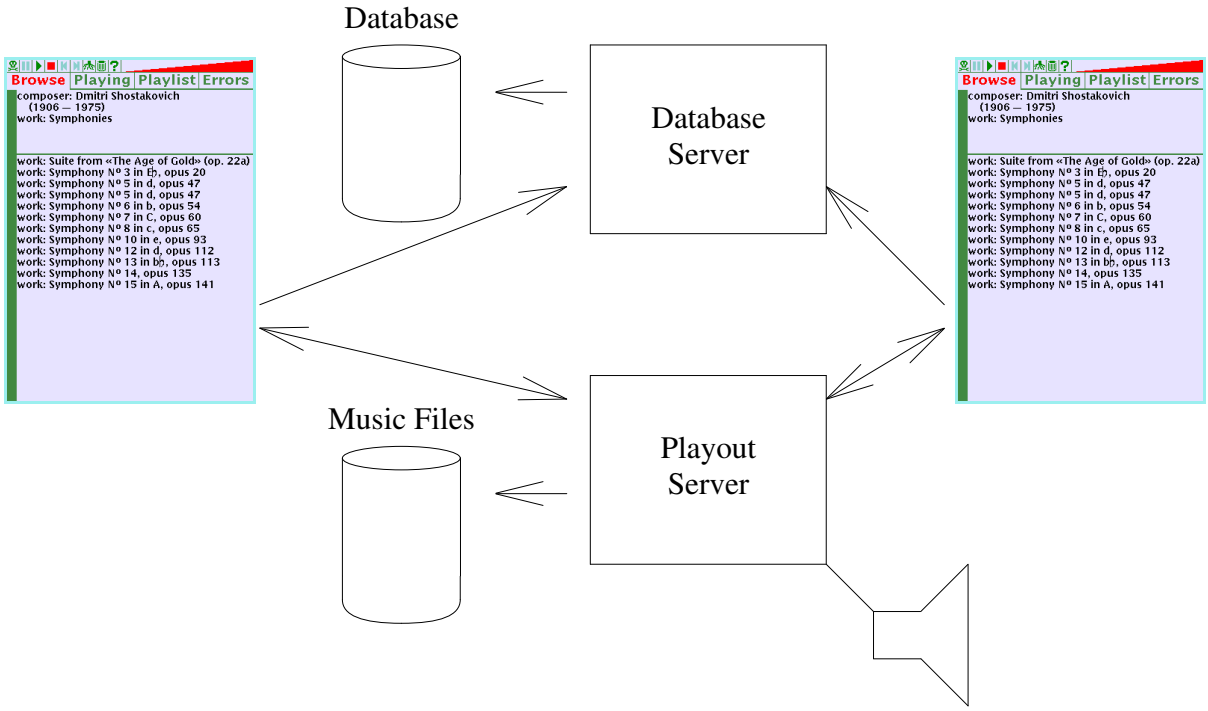
Jukebox Design

Before tackling the design, we need to know some things.

- We'd like to be able to have multiple GUIs controlling the jukebox: I like to have one on my PC, one on my laptop and one on my iPaq (which I keep next to my HiFi).
- The Bell Labs music database contains well over 300 GB of music data (3 TB uncompressed): more than 200 GB of 20th century popular music (it pretty much covers the top-100 for every week in the 20th century); 60 GB of classical music; and a lot of other stuff: famous speeches, humor, language courses, etc.
  Almost all of the music files are compressed using Bell-Labs' own PAC (Perceptual Audio Coding) standard, some are compressed with MPEG.
  This data is not easily and certainly not legally replicated (although I do have a small subset here, for demonstration purposes).
- Besides the music files themselves, there are two descriptive databases: one for pop music (25MB), one for classical music (2 MB). The GUI searches these databases, but they cannot always be stored in/near the GUI (the iPaq, for example, doesn't have enough storage).
- Different people want to use different databases with different search features.
- The actual playback of the music, to state the obvious, must be done near an audio device.

So what components would you design?

Design Summary



Design Summary

" Where is the data?

If the amount of data that needs to be searched is large, but the outcome of the query is small, then resolve the query at/near the database and ship the result across the network.

" What pieces must be independently changeable?

We may want to change the GUI (e.g., from mouse to keyboard based), we may want to be able to plug in a different music database (e.g., change between pop and classical).

" Where are the interfaces?

In our example, the interfaces are to the data (covered above), to the graphical user interfaces (one or more) and to the audio system.

" Where is performance critical?

Can network communication be avoided across performance-critical interfaces? In our example application, communication between decoder and audio interface is much more critical than between file system and decoder: there is ten times more data on the former.

Interfaces

Consider communication between GUI and music database. The GUI posts queries, the database answers them. Could be a classical RPC interface.

So, what is Remote Procedure Call?

It uses a standard procedural interface to cause the execution of operations on remote machines. You call a local procedure which packs up the arguments and sends them to another machine. When the answer comes back, the procedure returns the reply as the result of the procedure call.

RPC

An RPC *stub* is linked into the caller program. The caller calls the RPC procedure using the regular procedure-call mechanism. The call goes to the stub, which

1. Takes the parameters and puts them in a message (in some machine independent format).
2. Sends the message to the server and waits for a response message
3. When it arrives, takes the return value and any return parameters from the message and returns them to the caller.

To the caller, the whole thing looks like regular procedure call, except for this:

- Not all parameters types can be passed (pointers!);
- If local functions fail, they bring down the application (so why worry about it?); if remote ones fail, the local one has the option of continuing (fault tolerance!) so you have to deal with errors;
- When caller and callee are on different hardware or use different languages, there may not be a one-to-one mapping of parameter types (e.g., floating-point format, integer size, character set).

What RPC Doesn't Do …

- The procedure-call mechanism by itself cannot describe where to send procedure calls. There needs to be a *binding* step before the first call can be made.
- There is no obvious protection mechanism. Who's allowed to make what calls?
- And there's no security mechanism: How can you tell you're really talking to the server? How does the server know who the client is?

This requires additional non-RPC mechanisms.

Nevertheless, RPC is a compelling mechanism for client/server communication, so it got standardized ï

RPC Standards

NCP (Network Computing Protocol Apollo), XDR (External Data Representation SUN), DCS (Distributed Computing Standard The Open Software Foundation).

And the biggest, nastiest, meanest, ugliest of them all: Corba

What makes Corba evil? It is what makes so many standards evil: standardization bodies do not come up with the minimum amount of mechanism that will do the job; instead they come up with an awful union of all possible ways to do the job.

There is a rich history of such standards in a variety of fields: programming languages: PL1, Ada; directory services: X.400; can you imagine an email address that goes:

```
S=Mullender; G=Sape; OU=INF; P=utwente; A=d400; C=nl;
```

All these standards are embraced with great enthusiasm by industry and government alike and then telephony standards excepted they die a quiet death.

Network File Access

Another way to do remote operations is to model the target objects for the operations as *files*, and to do the operations by reading and writing the files.

This gives you an obvious interface for protection and security it's built into the file system interface.

But it doesn't deal with data representation differences between different machines; for this, textual representation of data is your friend.

Plan 9 uses this approach. The commands (things you write) and responses (things you read) are usually presented as text, allowing the user to do operatings simply by typing them in:

```
cdfs                          Start the cd/dvd server
echo ingest >/mnt/cd/ctl      Accept a disk
cp file /mnt/cd/wd            Write file as a data track
rm /mnt/cd/wd                 Fixate the disk
echo eject >/mnt/cd/ctl       Eject the disk
```

On any other system, a second application would have been required to interpret keyboard commands and pass them to the server.

An extra advantage of this approach is that access control and authentication come almost for free. Access control mechanisms already form part of a file system interface. In an RPC system, these issues must be faced from scratch.

But what I like best about these file-base interfaces is that you can use them without writing code: You just

```
echo request > file
```

Plan 9

Plan 9 models many interfaces as files. In many of these, access control is an issue which makes the file-system model with its built-in mechanisms for access control so much more useful.

In almost all cases, the file system interfaces are text based; the client translates commands and numbers into strings, sends them through a file request to the server and the server parses the input. It's not as efficient as parameter marshalling in RPC systems, but it's efficient enough and it is so much more debuggable!

A good example of a Plan 9 file system interface is the window system, *rio*(1).

Why does this work?

This works really well in Plan 9 and the reason is that Plan 9 has a mount table for groups of processes. Unix, in contrast, has one mount table per machine. In Plan 9, different processes can have different views of the name space.

Rio, the window server, for instance, has the hardware mouse (`/dev/mouse`) and the hardware keyboard (`/dev/cons`) in its name space. Processes running in a window inherit a name space that has a file system implemented by *rio* in their name space with a synthesized `/dev/mouse` and `/dev/cons`.

We use this mechanism a lot in Plan 9.

" The *dump* file system gives access to the file system at the time of one of the daily snapshots of the past ten years or so. By binding a file or directory from the dump file system on top of its current version, one can selectively go *'back in time'*, for example, to see if a bug was recently introduced or not.

" The *ftpfs* file system presents a file tree accessed via *ftp* as a regular file system. Thus:

```
ftpfs -a sape@plan9.bell-labs.com ftp.cs.utwente.nl
cp /n/ftp/pub/file $home/file
```

" The *plumb* file system allows application to communicate with each other through the *plumber*, a file system that allows opening files in appropriate applications by simply clicking the mouse on them. It resembles double clicking in Windows but is far more powerful.

Back to the Jukebox

The jukebox consists of three applications:

1. The Playlist File System, `games/playlistfs`,
2. The Music Database Server, `games/jukefs`, and
3. The Graphical Interface, `games/jukebox`.

There can be one instance each of the playlist and database file systems, and one or more or , if you like to work the system through the command line, zero graphical interfaces. We'll concentrate, at least for now, on the Playlist File System.

The Playlist File System

So what should the interface to our file system be like? Our first choice, inherent in the name: *Playlist **File** System*, was to implement it as a file system. This allows using it and testing it without needing a separate application to talk to the interface.

Our next choice was to implement the interface as three files:

```
/mnt/playctl
/mnt/playlist
/mnt/playvol
```

Reading `/mnt/playlist` gives the list of files being played or to be played. Each line consists of a file name and an identifying string could be text, but we use an *object ID*.

Reading `/mnt/playctl` shows the current state of the playlist file system; for instance, `play 24` means that the 24th entry in the play list is currently being played.

Reading `/mnt/playvol` shows the current playback volume as a percentage of the maximum volume, e.g., `volume 75`.

Writing these files changes the state of the playlist server. If, say, the state expressed in the `playctl` file is `play 13`, and we write the string `play 25` into `/mnt/playctl`, playout of the 13th file in the playlist will stop and the 25th file will start playing. Or, if we write the string `pause` and, some time later, the string `resume` into `/mnt/playctl`, I'm sure you can guess what happens.

`/mnt/playvol` can be similarly written.

/mnt/playlist

The playlist file itself has been implemented as an *append-only* file. Writing a string like

```
/sys/lib/music/classic/mozart/opera/408.pac      20043
```

to `/mnt/playlist` will append the file to the end of the playlist.

'Why do it this way?', you will ask (well, I *hope* you will).

The answer is twofold. First, append-only works for me. If I want to play the playlist out of order, I can do so by choosing which item to play.

But the important reason is to provide an interface that will allow a client (e.g., a graphical user interface) to find out when `/mnt/playlist`, or, for that matter, the other two files changes.

One, inelegant, solution is to have the clients periodically reread the files, e.g., once every few seconds. But this is esthetically unsatisfactory and it also causes ugly delays in the interface.

We allow clients to read until they reach end-of-file. End of file is signalled by a *read* that returns zero bytes. If, after a read that returns zero bytes, *another* read is issued, at the offset of the previous read, then we block that read operation until the state of the file changes.

Here's a fragment of the `games/jukebox` code.

Playlist tracker in `games/jukebox`

The code is slightly abbreviated for readability.

```
for(;;){
        m = read(fd, buf, sizeof buf);
        if(m == 0){
                /* End of file, next read will block until item is added */
                continue;
        }
        if(m < 0){
                /* File was truncated, reread from scratch */
                rerrstr(buf, sizeof(buf));
                assert(strcmp(buf, "reading past eof") == 0);
                emptyplaylist();
                continue;
        }
        /* Regular read */
        addplaylist(buf, m);
}
```

Organizing the Playlist Server

The playlist server has two main activities;

1.Play music files

2.Serve the file system (which may have multiple clients).

Task 1 is a real-time task (albeit with very loose real-time constraints: the audio driver buffers up to a second worth of audio data).

Some of the file service *read* requests must be blocked until there is a state change, so we must organize the code to deal with this.

The state managed by the playlist server consists of

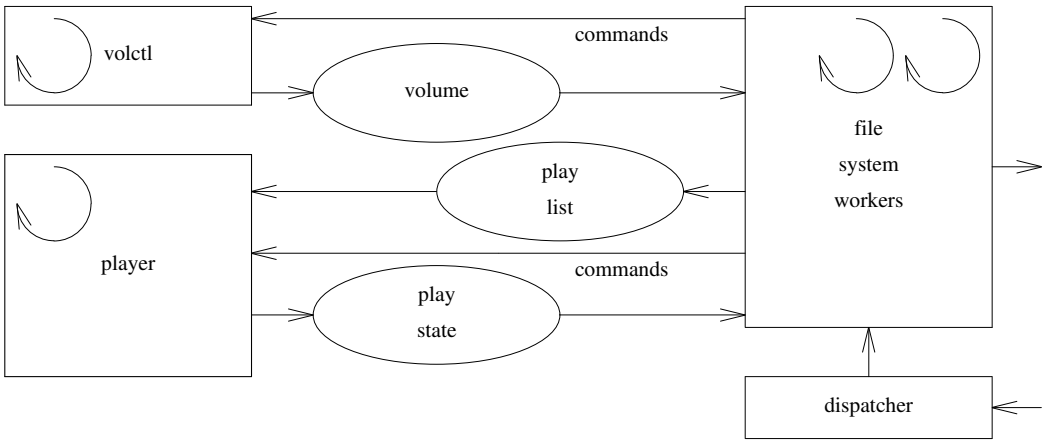
" The playlist (read by Task 1, read and written by Tasks 2 (plural!));

" The play state (play/stop/pause + number in playlist) which is modified in different ways by Tasks 1 and 2; and

" The volume setting, read and written by Tasks 2.

We have devised the following structure for the application:

Playlist Server Structure



Playlist Server Structure

- The *play list* is maintained by a set of workers, and read by the *Player Process*.
- The *play state* is maintained by the *Player Process*. Workers send state change requests to this process and then observe the state changes.
- The *volume state* is maintained by the *Volume Process* in analogy to the *play state*.

Why do it this way?

We'll show a thread- and process-management and message-passing strategy that allows concurrency in a **lock-free manner**.

To appreciate this, let's (re)examine some of the pitfalls of unfettered concurrency.

Classical Race Condition

```
typedef struct Queue Queue;
struct Queue {
        void *inp, *outp;
        void *data[Size];
}

void*
consumer(Queue *q)
{
        void *datum;

        while(q->inp == q->outp) sleep(100);
        datum = *q->outp++;
        if(q->outp == &q->data[Size])
                q->outp = &q->data[0];
        return datum;
}

void
producer(Queue *q, void *datum)
{
        while(q->inp == &q->outp[1]
          || (q->inp == &q->data[Size] && q->outp == &q->data[0]))
                sleep(100);
        *q->inp++ = datum;
        if(q->inp == &q->data[Size])
                q->inp = &q->data[0];
}
```

Classical Race Condition

```
typedef struct Queue Queue;
struct Queue {
        long inp, outp;
        void *data[Size];       /* Size must be power of two */
}

void*
consumer(Queue *q)
{
        void *datum;

        while(q->inp == q->outp) sleep(100);
        datum = *q->data[q->outp++ & (Size-1)];
        return datum;
}

void
producer(Queue *q, void *datum)
{
        while(q->inp - q->outp == Size)
                sleep(100);
        q->data[q->inp++ & (Size-1)] = datum;
}
```

Is this really race free?

Lock-Free Queue Single Consumer, Single Producer, Correct Version

```
typedef struct Queue Queue;
struct Queue {
        long inp, outp;
        void *data[Size];       /* Size must be power of two */
}

void*
consumer(Queue *q)
{
        void *datum;

        while(q->inp == q->outp) sleep(100);
        datum = *q->data[q->outp & (Size-1)];
        q->outp++;
        return datum;
}

void
producer(Queue *q, void *datum)
{
        while(q->inp - q->outp == Size) sleep(100);
        q->data[q->inp & (Size-1)] = datum;
        q->inp++;
}
```

Challenge: Devise a lock-free multi-producer, multi-consumer queue. Can it be done? If you can construct one, mail me the code before next class (sape@plan9.bell-labs.com)