Multiservice servers

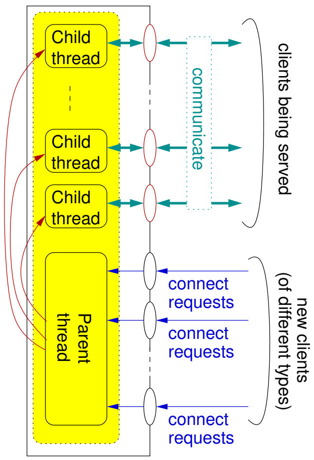
Stefan D. Bruda

CS 464/564, Fall 2023

Multiservice servers

- Why?
  - Because it sounds like fun
  - Because we may need it
    - E.g., a database server might receive requests from clients, but also from other database servers which want to keep information in sync
- How?

```
loop
    listen for clients on ports p_1 p_2 ... p_n
    if a client (of type x) requests connection on port p_x then
        fork/pthread_create
    in child process/new thread do
        handle clients of type x
        terminate
forever
```



Super servers

- More whys
  - It is also the case that there are a whole bunch of small TCP services out there
    - Some of them may be used once a month or something
    - Keeping one server running for each and every such a service is an utter waste of resources
  - It makes sense to run a "super server" which will listen to many sockets and launch the appropriate server only when needed
    - These servers are separate executables that do not run unless the super server launches them
- How?

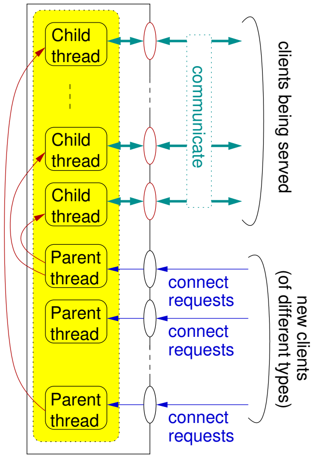
```
loop
    listen for clients on ports p_1 p_2 ... p_n
    if a client (of type x) requests connection on port p_x then
        fork/pthread_create
    in child process/new thread do
        handle clients of type x  --> Lanuch a particular server that
                                      handles clients of type x
                                      (using execve)
        terminate
forever
```


Implementation of multiservice or super servers

- As far as server design is concerned, a multiservice server is not that different
  - In particular, you can build such a server that
    - is iterative (does not make much sense though),
    - simulates concurrency in one thread of execution,
    - uses multiple processes, or
    - uses one process with multiple threads of execution
- Sometimes it make sense to launch a different program when a connection request arrives
  - More flexible: small changes in various application protocols being handled do not need the recompilation of the whole thing
- Sometimes it make sense to implement everything in one program
  - E.g., when the different protocols are closely related to each other and make no sense when considered in isolation

Implementation of multiservice or super servers (cont'd)

- Multiservice servers listen to several master sockets with no way to anticipate which of them will receive the next connection request
  - You can have an individual thread (or process) listening on each master socket
  - Alternatively, you can use `poll` or `select` in the master thread
- When launching different programs that handle the actual communication, it makes much more sense to use processes
  - Indeed, you just do fork immediately followed by execve in the child process
- When the multiple application protocols are handled by one program, it makes more sense to use threads
  - Those protocols share a big deal of data, else you would have handled them using separate programs…



Super server configuration

- A multiservice server (with all the code in one program) does not need a lot of configuration
- It is reasonable though to expect the ability to configure a super server
  - We start with a super server skeleton
  - An administrator may then add or delete services to our skeleton as needed
- Static configuration: The configuration information is written in a configuration file, read by the server each time it starts
  - If an administrator wants to add (or delete) a service, she will change this file, stop the server, and launch it again.
- Dynamic configuration: We have the same configuration file, but
  - The server does not need to be stopped and restarted
  - Instead, the administrator changes this file, and tells the server that the file has been modified by sending a signal
  - Once the server receives the signal, it re-reads the configuration file and applies the changes
  - Civilized servers react this way when they receive `SIGHUP` (1)
  - What if there is no signal mechanism?

Dynamic configuration through control sockets

If there is no signal mechanism, we can use for reconfiguration and many other thing… (drum roll) sockets!

- Recall that on any machine running TCP/IP the IP address 127.0.0.1 always denotes the machine itself and only the machine itself
- So we can have an extra master socket, the control socket as follows:
  - A control socket listens only to the address 127.0.0.1 (`INADDR_LOOPBACK`) and receives control messages
  - One such control message could be a request to re-read the configuration file, and we then implement dynamic configuration
- The control socket is actually more general than the `SIGHUP` signal, and thus useful for other tasks as well
  - Indeed, we can use it to send any imaginable commands to the server!
  - For instance, in a server with a monitor thread we can ask the monitor thread to print information on demand rather than periodically

Control socket: example

```
void* monitor (void* ignored) {
  const int cport = 8000; // control port
  int csock, ssock, connections, n;
  char com[256];
  struct sockaddr_in client_addr; // the address of the client...
  unsigned int client_addr_len = sizeof(client_addr); // ... and its length
  csock = controlsocket(cport,0);
  while (1) {
    ssock = accept(csock, (struct sockaddr*)&client_addr, &client_addr_len);
    while (1) { // we keep reading commands from the control client...
      int done = 0;
      if ( (n = readline(ssock,com,256)) < 0 ) {
        perror("readline (control)"); done = 1; }
      else if ( n == 0 ) done = 1;
      else if ( strncmp("QUIT",com,strlen("QUIT")) == 0 ) done = 1;
      else if ( strncmp("DUMP",com,strlen("DUMP")) != 0 ) continue;
      if (done) {
        shutdown(ssock,1); close (ssock); break;  // from the inner while loop.
      }
      // we have received a DUMP command so we get busy:
      pthread_mutex_lock(&mon.mutex);
      ...
      pthread_mutex_unlock(&mon.mutex);
    } // inner while
  } // outer while
}
```

INETD: THE SUPER SERVER

Many Unix systems do not run a server for each and every service they offer; instead, they run inetd (the "internet daemon")

- Motivation: offer many services without using excessive system resources
- More motivation: ECHO is a useful service for network debugging, but does not make much sense in a production system; it should be easy to enable and disable it
- Inetd is dynamically configurable (it understands `SIGHUP`)
- The configuration is stored in `/etc/inetd.conf`, with lines like this:

service name	socket type	protocol	wait?	userid	server program	argu-ments
ftp	stream	tcp	nowait	root	/usr/sbin/proftpd	

- Problem: how does the called program know what socket to communicate on with the client?
  - Inetd moves the connection (i.e., opened slave socket) to index zero in the child's descriptor table
  - So a "subserver" (such as `/usr/sbin/proftpd`) will just read from and write to socket descriptor 0

MOVING THE SOCKET DESCRIPTOR

- The super server (e.g., inetd) will do something like this:

```
if (fork() == 0) {
  close(msock);   // child does not listen to master socket...
  close(0);
  dup2(ssock,0);  // copy ssock to index 0 in the descriptor table
  char** slave_server_args = {0};
  execve(slave_server, slave_server_args,envp);
}
else
  ... (parent code)
```

- Then the slave server will do:

```
while ((n = readline(0,req,ALEN-1)) != 0) {
  if (strcmp(req,"quit") == 0) { break; }
  send(0,ack,strlen(ack),0);
  send(0,req,strlen(req),0);
  send(0,"\n",1,0);
}
```


XINETD

- In fact, newer systems use xinetd (the "extended internet daemon")
- Behaviour is similar to inetd except that the place of a configuration file is taken by a directory (`/etc/xinetd.d`)
- For any service you can use, you drop into this directory a small text file

```
< hoare:~ > cat /etc/xinetd.d/cups-lpd
service printer
{
        socket_type = stream
        protocol = tcp
        wait = no
        user = lp
        group = lp
        passenv =
        server = /usr/libexec/cups/daemon/cups-lpd
        server_args = -o document-format=application/octet-stream
        disable = yes
}
```