

Next up

- Client-server model
- RPC
- Mutual exclusion

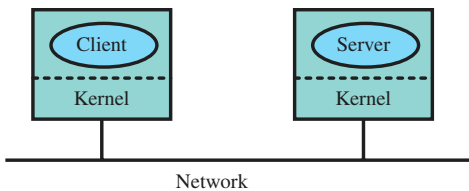


Client-server model

- Protocols address the problem of getting bits between machines
- How do we structure programs?
- The simplest design is client/server
 - *The OS is structured as a group of cooperating processes called **servers** that offer services to users, called **clients***



Client-server



Client-server protocol

- Usually connectionless (UDP)
- NFS
 - Client sends a request (read a block from a file)
 - Server returns a response (here is the block, no such block, etc)
- Unix system/libc calls
 - `int socketd = socket(domain, type, protocol)`
 - `bind(socketd, address)`
 - `sendmsg(socketd, &msg, flags)`
 - `recvmsg(socketd, &msg, flags)`

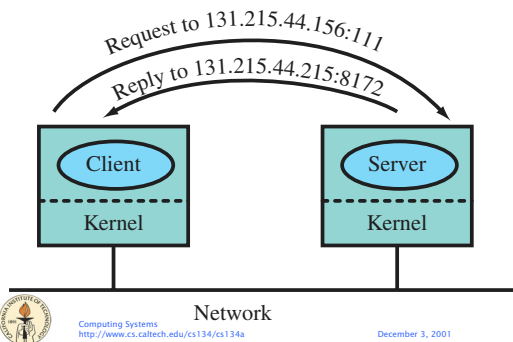


Addressing (naming)

- How do we find a server?
- Simplest scheme:
 - Address is IP/port pair (to distinguish between multiple servers running on the same machine)
 - Server address is coded into client either as a parameter, in a file, or hard-coded



Hard-coded addressing

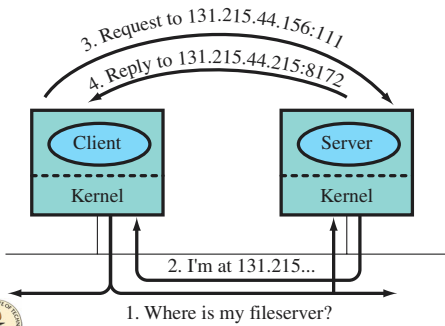


Broadcast lookup

- Used for ARP (determine Ethernet address from IP address)
- Servers just pick a name from a large, sparse space (say a 64-bit identifier)
- Steps
 - 1. Client broadcasts "where are you?"
 - 2. Server responds "here I am"
 - 3. Client sends request
 - 4. Server replies
- Caching can help avoid repeated broadcasts



Broadcast lookup

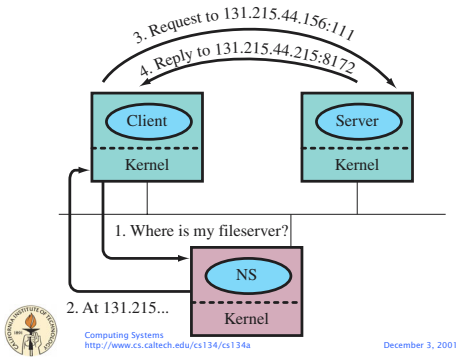


Using a nameserver

- DNS (maps IP names to IP numbers)
- The nameserver is a distributed database of location info
 - Servers register services



Nameserver



Problems

- Hard-coded: not transparent
- Broadcast: transparent, but extra system load
- Name server: may not be scalable
- What about faults?

Reliability

- Messages may get lost
- Handshaking
 - When a message is sent, the client blocks
 - When the message is received, the server kernel sends an ACK
 - Server sends result
 - Client kernel sends ACK
 - Client returns
- If a message is lost, it can be discovered by a timeout
- Idempotent?

Client-server protocols

- REQ: request: client wants service
- REP: reply: response from the server
- ACK: acknowledgement: the previous packet arrived
- AYA: are you alive?
- IAA: I am alive
- TA: try again (out of resources)
- AU: address unknown



Remote procedure calls

- Client-server model is awkward
- Based on sending and receiving message (I/O)
- Not a natural concept
- RPC: provide a function-call like interface
- Server *publishes* a function
 - `read(int fd, char *buf, int nbytes)`
- Client calls remote function
 - `read(int fd, char *buf, int nbytes)`



RPC design

- Usually based on function stubs
- Client has a function stub that formats a message
- Server has a stub to unformat the message



RPC sequence

- Client procedure calls client stub
- Client stub builds a message, traps to kernel
- Kernel sends message to remote kernel
- Remote kernel gives messages to server stub
- Server stub unpacks parameters and calls server
- Server does the work, and returns result to server stub
- Server stub packs the message and traps to kernel
- Remote kernel sends a message to client kernel
- Client kernel gives message to client stub
- Client stub unpacks result and returns to client

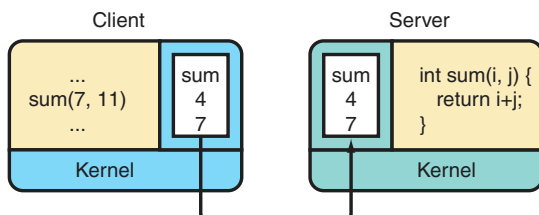


RPC parameter passing

- RPC should appear *transparent*: its not possible to tell between a local and remote procedure call (in C this is hard)
- Packing parameters into a message is called parameter *marshaling*
- Simple example: `sum(int i, int j)`



RPC: `sum`



RPC: heterogeneous systems

- Homogeneous distributed systems: every machine is the same
- Multiple kinds of machines (for example Intel/SPARC/Alpha)
- Character representations may differ (ASCII vs EBCDIC)
- Integers (1s complement vs 2s complement)
- Byte orders: little-endian (Intel) vs big-endian (SPARC)
- Word sizes: 32 bit ints vs 64 bit ints (Alpha)



Parameter marshaling

- Another problem: the bytes in an int must be reordered, but the bytes in a string must not
- The *types* can be used to determine how to marshal
- Next problem: what is the network format for message?
 - *Canonical*
 - ASCII for strings
 - 0/1 for Booleans
 - IEEE floats
 - *Client-based: client sends in native format, but indicates the format in the first byte*



RPC programming semantics

- Calling styles
 - *Call-by-name*
 - *Call-by-value*
 - *RPC: call-by-copy?*
- How do we marshal pointers?



Call-by-copy example

Client

```
void sum(int *result, int x, int y);
```

Server

```
void sum(int *result, int x, int y)
{
    *result = x + y;
}
```



Reading from a file

- `int read(int fd, char *buf, int size)`
- Marshal:
 - Send the file descriptor
 - Send the buffer (should be at least size bytes long)
 - Send the size
- Server
 - Get fd, size
 - Copy the buffer
 - Read data into buffer and send it back
- Client unmarshal
 - Receive data
 - Copy it into the buffer



Marshaling

- What to do about arbitrary data structures
- The compiler/run-time can marshal arbitrary objects by following pointers
 - Current compilers can't do this
 - Poor behavior in the presence of side-effects
 - May send too much data (all the reachable data)
- The client can send a *reference*
 - If server accesses data, it has to get it from the client



How to locate the server?

- Server specification

```
module FileServer = struct
  long read(in char name[MAX_PATH],
           out char buf[MAX_SIZE],
           in long bytes, int long position);
  long write(in char name[MAX_PATH],
            in char buf[MAX_SIZE],
            in long bytes, int long position);
  int create(in char name[MAX_PATH],
            in int mode);
  int unlink(in char name[MAX_PATH]);
end
```



Name lookup

- Server registers services with binder (portmapper)
 - Each procedure gets a unique ID
- During linking, client obtains server port from binder
- Advantages
 - Flexible
- Disadvantages
 - Overhead (each client starts over; cache the binding?)
 - Multiple binders (extra overhead to keep them consistent)



Failures

- Lots of opportunities for failure
 - Client may fail
 - Server may fail
 - Network may fail
 - Messages may be lost



Lost request messages

- Have the kernel start a timer
- If request is not ACKed within timeout, send it again
- What if ACK was lost?
 - Server may get duplicate requests
 - Add unique identifier (sequence number) to each request
- What if network is bad, or server is slow
 - Client may falsely conclude server is down



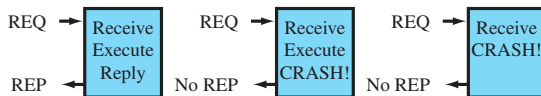
Lost reply messages

- Harder
- If client times out, send request again
 - Is server slow, or did the reply get lost?
 - Should the server buffer the reply?
 - What about requests that have side-effects?
- Requests that have no ill-effects on repeated use are called **idempotent**
 - e.g. transferring money into my bank account (not!)
- Can use a sequence number to identify repeated requests



Server crashes

- What to do for the two cases



Server crashes

- Three semantics
- At least once
 - Keep trying until the server responds
 - Works ok for idempotent requests
 - RPC will be executed once or many times
- At most once
 - Always report error on failure
 - RPC may be executed up to one time
- Exactly once
 - RPC is always carried out exactly once
 - Not computable



Client crashes

- Client sends a requests to a server, then crashes
- The executing process is called an orphan
- Ties up resources
- What if client reboots and immediately gets a reply (for what?)



Client crashes

- Extermination
 - Client keeps a log, kills orphans on reboot
- Reincarnation
 - Client broadcasts the beginning of a new epoch when it reboots
 - All remote processes are tagged with their epoch
- Gentle reincarnation
 - Servers kill process at the start of a new epoch
- Expiration
 - Give each RPC process a quantum T
 - When quantum expires, the client must be contacted