CS 162: Operating Systems and Systems Programming

Lecture 9: Inter-Process Communication (IPC) & Remote Procedure Calls (RPC)

Sept 26, 2019
Instructor: David Culler
https://cs162.eecs.berkeley.edu
Recall: Lock Solution – 3rd cut

mutex buf_lock = <initially unlocked>
CondVar buf_signal = <initially nobody>

Producer(item) {
    lock buffer
    while (buffer full) {cond_wait(buf_signal, buf_lock) }
    Enqueue(item);
    unlock buffer
    broadcast(buf_signal)
}

Consumer() {
    lock buffer
    while (buffer empty) {cond_wait(buf_signal, buf_lock) }
    item = queue();
    unlock buffer
    broadcast(buf_signal)
    return item
}

Release lock; signal others to run; reacquire on resume
n.b. OS must do the reacquire
Why User must recheck?
Monitors

- **Lock**: protects access to shared data
- Always acquire lock when accessing
- Queue of threads waiting to enter the monitor
Monitors in practice

• Locks for mutual exclusion
• Condition variables for waiting
• A monitor is a lock and zero or more condition variables with some associated data and operations
  • Java provides this natively
  • POSIX threads: Provides locks and condvars, have to build your own
Monitors

- **Condition Variables**: queue of threads waiting for something to become true inside critical sect.
- Atomically release lock and start waiting
  - Another thread using the monitor will signal them
- The condition: Some function of monitor's data
Why the while Loop?

• Can we "hand off" the lock directly to the signaled thread so no other thread "sneaks in?"
  • Yes. Called **Hoare-Style Monitors**
  • Many textbooks describe this scheme

• Most OSs implement **Mesa-Style Monitors**
  • Allows other threads to sneak in
  • Much easier to implement
  • Even easier if you allow "spurious wakeups"
  • ```wait()``` can return when no signal occurred, in rare cases
  • POSIX allows spurious wakeups
Interlude: Concurrency Is Hard

• Even for practicing engineers trying to write mission-critical, bulletproof code!

• Therac-25: Radiation Therapy Machine with Unintended Overdoses (reading on course site)

• Mars Pathfinder Priority Inversion (JPL Account)

• Toyota Uncontrolled Acceleration (CMU Talk)
  • 256.6K Lines of C Code, ~9-11K global variables
  • Inconsistent mutual exclusion on reads/writes
Comparing Synchronization

• Semaphores can implement locks
  • Acquire() { semaphore.P(); }
  • Release() { semaphore.V(); }

and Condition Variables

• Monitors combine locks and CVs in a structured fashion

• Modern view: concurrent objects (e.g., Java)

• Are there other important common patterns?
IPC/RPC Background

• Collections of threads interact through shared objects and signals in a common address space
  • Multiple threads in a user process
  • Multiple threads forming the kernel

• Processes are isolated from each other – distinct address spaces – so they interact through external means
  • Files, Pipes, Sockets
  • Function as communication channels
  • Narrow interfaces, limited interactions
  • On the same machine or not

• These are forms of message passing
  • Can be utilized between threads in a process too
  • GO channels, MPI, CSP, …
  • THE paradigm for large parallel machines and clusters
  • AND across any network (of course)
Recall: Communication between processes

- Can we view files as communication channels?

\[ \text{write}(wfd, wbuf, wlen); \]
\[ \text{n} = \text{read}(rfd, rbuf, rmax); \]

- We have seen one example – pipes
- Routinely used with the shell
  \[ \text{>>> grep list src/*//*.c | more} \]
Recall: Communication Across the world looks like file IO

- Connected queues over the Internet
  - But what’s the analog of open?
  - What is the namespace?
  - How are they connected in time?

```c
write(wfd, wbuf, wlen);
```

```c
n = read(rfd, rbuf, rmax);
```
Socket API

- Base level Network programming interface
Recall: Sockets w/ Protection & Parallelism

Client

Create Client Socket

Connect it to server (host:port)

Connection Socket

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept syscall()

Connection Socket

Child

Close Listen Socket

read request

write response

Close Connection Socket

Close Connection Socket

Parent

Close Server Socket

Aside: why does the server need to close the connection socket?
Recall: What Is A Protocol?

• A protocol is an agreement on how to communicate

• Includes
  • Syntax: how a communication is specified & structured
    • Format, order messages are sent and received
  • Semantics: what a communication means
    • Actions taken when transmitting, receiving, or when a timer expires

• Described formally by a state machine
  • Often represented as a message transaction diagram
IPC Issue: Representation

• You have mostly experienced writing and reading strings
  • i.e., sequential stream of characters (formerly bytes)

• What about an int? struct? array? list?

• An object in memory has a machine-specific binary representation.
  • Threads in a common process address space are all in the same machine and have the same view of what’s in memory.
  • Offsets into fields, follow pointers

• When a object is externalized, it must become a sequential sequence of bytes
  • And it must be possible to read it back and create an equivalent object
Endian-ness

• For a byte-address machine, which end of a machine-recognized object (e.g., int) does its byte-address refer to?

• BigEndian: address is the most-significant bits

• LittleEndian: address is the least-significant bits

```
int main(int argc, char *argv[]) {
    int val = 0x12345678;
    int i;
    printf("val = \%x\n", val);
    for (i = 0; i < sizeof(val); i++) {
        printf("val[%d] = \%x\n", i, ((uint8_t *) &val)[i]);
    }
}
```

(base) CullerMac19:code09 culler$ ./ endian
val = 12345678
val[0] = 78
val[1] = 56
val[2] = 34
val[3] = 12
What endian is the Internet?

NAME

arpa/inet.h - definitions for internet operations

SYNOPSIS

#include <arpa/inet.h>

DESCRIPTION

The in_port_t and in_addr_t types shall be defined as described in <netinet/in.h>.

The in_addr structure shall be defined as described in <netinet/in.h>.

The INET_ADDRSTRLEN [IP6] and INET6_ADDRSTRLEN [6] macros shall be defined as described in <netinet/in.h>.

The following shall either be declared as functions, defined as macros, or both. If functions are declared, function prototypes

```c
uint32_t htonl(uint32_t);
uint16_t htons(uint16_t);
uint32_t ntohl(uint32_t);
uint16_t ntohs(uint16_t);
```

The uint32_t and uint16_t types shall be defined as described in <inttypes.h>.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```c
in_addr_t     inet_addr(const char *);
char          *inet_ntoa(struct in_addr);
const char   *inet_ntop(int, const void *restrict, char *restrict,
                        socklen_t);
int           inet_pton(int, const char *restrict, void *restrict);
```

Inclusion of the <arpa/inet.h> header may also make visible all symbols from <netinet/in.h> and <inttypes.h>.
Abstracting away representation

NAME

netinet/in.h - Internet address family

SYNOPSIS

#include <netinet/in.h>

DESCRIPTION

The <netinet/in.h> header shall define the following types:

in_port_t
   Equivalent to the type uint16_t as defined in <inttypes.h>.

in_addr_t
   Equivalent to the type uint32_t as defined in <inttypes.h>.

The sa_family_t type shall be defined as described in <sys/socket.h>.

The uint8_t and uint32_t type shall be defined as described in <inttypes.h>. Inclusion of the <netinet/in.h> header

The <netinet/in.h> header shall define the in_addr structure that includes at least the following member:

in_addr_t s_addr

The <netinet/in.h> header shall define the sockaddr_in structure that includes at least the following members:

sa_family_t       sin_family       AF_INET.
in_port_t         sin_port         Port number.
struct in_addr    sin_addr         IP address.
Aside: write_words / read_words

- All the issues of data representation arise in non-text files as well.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int i, numbers[100];
    FILE *wfile = fopen("intdata.bin", "w");
    for (i = 0; i < 100; i++) {
        numbers[i] = i*i;
    }
    size_t wlen = fwrite(numbers, sizeof(int), 100, wfile);
    printf("Wrote %zu ints\n", wlen);
    fclose(wfile);
}
```
What about richer objects?

• Consider the list of `word_count_t` of HW0/1 ...

• Each element contains:
  • An `int`
  • A `pointer` to a string (of some length)
  • A `pointer` to the next element

• `fprintf_words` writes these as a sequence of lines (character strings with `\n`) to a file stream

• What if you wanted to write the whole list as a binary object (and read it back as one)?
  • how do you represent the string?
  • Does it make any sense to write the pointer?

```c
typedef struct word_count
{
    char *word;
    int count;
    struct word_count *next;
} word_count_t;
```
Serialization

• Converting data structures into a canonical linear format so that they can be stored/retrieved or transmitted/received.
• Values, structs, lists & trees are easy
  • graphs are hard
• Many choices with different pros/cons
  • JSON & XML common in web
  • Sun External Data Representation (XDR) std since 80’s
  • Built in to languages like Java, Lisp, …
  • Google Protocol Buffers provide simple description language
• Use a tool that fits your needs
Data Serialization Formats

- JSON and XML are commonly used in web applications
- Lots of ad hoc formats
# Data Serialization Formats

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Avro</td>
<td>Apache Software</td>
<td>NA</td>
<td>No</td>
<td>Apache Avro® 1.1 specification(s)</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>Yes (built-in)</td>
<td>N/A</td>
<td>NA</td>
</tr>
<tr>
<td>Apache Parquet</td>
<td>Apache Software</td>
<td>NA</td>
<td>No</td>
<td>Apache Parquet® [1]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>Java, Python</td>
<td>N/A</td>
</tr>
<tr>
<td>ASKJ</td>
<td>W3C, BR, IFU, ITU</td>
<td>NA</td>
<td>Yes</td>
<td>ISO 8859, X.430 or X.431</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Bencode</td>
<td>IETF</td>
<td>NA</td>
<td>No</td>
<td>Bencode Protocol [2] [3]</td>
<td>Partially (numbers and names are ASCII)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Bini</td>
<td>Bernard Reuter</td>
<td>NA</td>
<td>No</td>
<td>Bini Specification [4]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>BSON</td>
<td>MongoDB</td>
<td>JSON</td>
<td>No</td>
<td>BSON Specification(s)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CXML</td>
<td>Cameron Structs, P. McInnes</td>
<td>CXML (based on)</td>
<td>Yes</td>
<td>RFC 2596</td>
<td>Yes</td>
<td>No</td>
<td>Yes through tagging</td>
<td>No (CDDL)</td>
<td>N/A</td>
<td>Yes (OSI)</td>
</tr>
<tr>
<td>Comma-separated values (CSV)</td>
<td>RFC, author unknown, Stenback</td>
<td>CSV (based on)</td>
<td>Yes</td>
<td>RFC 4180</td>
<td>Partial (values are CSV)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No (XLS)</td>
<td>N/A</td>
</tr>
<tr>
<td>Common Data Representation (CDR)</td>
<td>Object Management Group</td>
<td>CDR (based on)</td>
<td>Yes</td>
<td>CDR, Formal 1.1-83</td>
<td>Partial (values are CDR)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No (CDR)</td>
<td>N/A</td>
</tr>
<tr>
<td>D-Bus Message Protocol (DBP)</td>
<td>freeseldp.org</td>
<td>D-Bus</td>
<td>Yes</td>
<td>D-Bus Specification(s)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>(see D-Bus)</td>
<td>N/A</td>
</tr>
<tr>
<td>Efficient XML Interchange (EXI)</td>
<td>W3C</td>
<td>XML, Efficient XML</td>
<td>Yes</td>
<td>Efficient XML, Interchange (EXI)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Partial (signature is EXI)</td>
<td>N/A</td>
<td>Yes (see D-Bus)</td>
</tr>
<tr>
<td>FlatBuffers</td>
<td>Google</td>
<td>NA</td>
<td>No</td>
<td>FlatBuffers specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Partial (internal to the buffer)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Fast IRNet</td>
<td>ISO, BR, ITU, ITU</td>
<td>XML</td>
<td>Yes</td>
<td>ITUT X.881 and ISO/IEC 24601-1-2007</td>
<td>Yes</td>
<td>No</td>
<td>Yes through tag</td>
<td>No (WIN)</td>
<td>N/A</td>
<td>Yes (tag)</td>
</tr>
<tr>
<td>FHIR</td>
<td>Health Level, IHE</td>
<td>FHIR</td>
<td>No</td>
<td>FHIR specification</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>(XML schema)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Ion</td>
<td>Amazon</td>
<td>JSON</td>
<td>No</td>
<td>Ion specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No (XLS)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Jaxen serialization</td>
<td>Oracle Corporation</td>
<td>XML</td>
<td>Yes</td>
<td>Java Object serialization</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No (XML)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>JSON</td>
<td>Douglas Crockford</td>
<td>JavaScript</td>
<td>Yes</td>
<td>JSON Schema in JavaScript</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No (XML)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MessagePack</td>
<td>Scalable Fwhouse</td>
<td>JSON (based on)</td>
<td>No</td>
<td>MessagePack format specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Netbios</td>
<td>Dan Bernstein</td>
<td>NA</td>
<td>No</td>
<td>Netbios specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>OSGI</td>
<td>Rolf Veen</td>
<td>?</td>
<td>Yes</td>
<td>OSGI specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>OPC-UA (library)</td>
<td>OPC Foundation</td>
<td>NA</td>
<td>Yes</td>
<td>OPC-UA specification(s)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>OpenEXI</td>
<td>Eric Langille</td>
<td>C, PHP</td>
<td>No</td>
<td>OpenEXI specification(s)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No (PHP)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RESTful (Python)</td>
<td>Guido van Rossum</td>
<td>Python</td>
<td>Yes</td>
<td>RESTful specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No (XML)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Property list</td>
<td>NIST (draft)</td>
<td>ASP/</td>
<td>Partial</td>
<td>Property list for XML formatting</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Partial (XML, WSDL)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Protocol Buffers (protobuf)</td>
<td>Google</td>
<td>NA</td>
<td>No</td>
<td>Google Protocol Buffers specification</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes (built-in)</td>
<td>C++, Java, Python, JavaScript, Go</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Inter-Process Communication (IPC)

• Mechanism to create a communication channel between distinct processes
  • User to User, Kernel to User, Same machine or different ones, Different programming languages, …

• Serialization format understand by both

• Can have authentication and authorization mechanism associated with it

• Failure in one process isolated from the other
  • But may have to take exceptional measures

• Many uses and interaction patterns
  • Logging process, Window management, …
  • Moving some system functions out of kernel to user space
IPC to simplify / extend OS

- What if the file system is not local to the machine, but on the network?
- Is there a general mechanism for providing services to other processes?
Recall: Request/Response Protocol

Client (issues requests)

write(rqfd, rqbuf, buflen);

Server (performs operations)

n = read(rfd, rbuf, rmax);

n = read(resfd, resbuf, resmax);

write(wfd, respbuf, len);

Serialized Objects

requests

wait

responses

service request
Domain Name System (DNS)

• Another proto-RPC distributed system

• **Purpose:** Convert a human readable name (www.google.com) to an IP Address (169.229.15.7)

• Why?
  • Humans don't want to remember IP addresses
  • But IP routes traffic based on IP addresses

• Other benefits
  • Service can change hosts (IP Address) but keep name
  • Fancy things like sending Google users to different hosts for load balancing
Example: DNS

Hostname Resolution Request

Hostname Resolution Response (IP)
DNS

• A hierarchical system for naming
• Names are divided into labels, right to left:
  • www.eecs.berkeley.edu

• Each domain owned by a particular organization
  • Top level handled by ICANN
  • Subsequent levels owned by organizations

• Resolution by repeated queries
  • Name server for each domain: <root>, edu, berkeley.edu, eecs.berkeley.edu
DNS – Root Server

• How do we know where to start?

• Hardcoded list of root servers and backups (updated rarely)

• Or use your ISP's server
  • Makes repeated queries on your behalf
  • Called a recursive resolver
Don’t libraries provide Services?

main( ... ) {
    “...”;
    res = libr_fun(p1, p2,...);
    “do something with results”
}

result_t libr_fun(arg_t argl, ...) {
    “access the arguments”;
    “do something useful for caller”
    “return results”
}
And aren’t system calls doing this?

- User program call library function
- Library function formats args for syscall
  - Issues syscall exception
- Syscall handler unpacks the args and calls (dispatches to) the subsystem function that handles the call
- Subsystem performs the operations
- Syscall rtn handlers puts result in reg and resumes user thread
- Library function gets syscall result and returns to the user program
Remote Procedure Call (RPC)

• Idea: Make communication look like an ordinary function call

• Wrapper library like for system calls
  • Called *stubs*

• Also wrappers at the receiving end
  • Read messages from socket, dispatch to actual function

• Look like local function calls
**RPC Concept**

**Client (caller)**

\[ r = f(v1, v2); \]

**Server (callee)**

\[ res_t f(a1, a2) \]

---

**Diagram:**

- **Client Stub:**
  - bundle args
  - send
  - return
  - receive
  - unbundle ret vals

- **Server Stub:**
  - bundle ret vals
  - send
  - return
  - call
  - receive
  - unbundle args
RPC Information Flow

Client (caller)
\[ r = f(v1, v2); \]

Server (callee)
\[ res_t = f(a1, a2) \]

Machine A

Machine B
Six steps

1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.

2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshaling.

3. The client's local operating system sends the message from the client machine to the server machine.

4. The local operating system on the server machine passes the incoming packets to the server stub.

5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshaling.

6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.
Stubs

• Client and server use “stubs” to glue pieces together
  • Client stub is responsible for “marshaling” arguments and “unmarshaling” the return values
  • Server-side stub is responsible for “unmarshaling” arguments and “marshaling” the return values
• Regular function call (x86 calling convention etc…)
• RPC function “name” is “resolved” to remote handler function at server
  • Dispatch similar to syscall

```c
FILE *ropen( name, mode) {
  ‘send <#, name, mode>’
  ‘rcv result’
}
```

typ open(name,mode) {
  … open the file
}

9/26/19

37
Stub Generation

• We need to “discover” available methods
• Interface definition language (IDL)
  • Contains, among other things, types of arguments/return
  • Sent from server to client for stub generation
  • IDL “compiler” generates stub functions
Marshaling

• Marshaling involves (depending on system) converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.
  • Needs to account for cross-language and cross-platform issues
    • Eg. Big endian vs Little endian

• Overhead.

• Many many different formats
RPC Binding

• How does client know which machine to send RPC?
  • **Binding**: the process of converting a user-visible name into a network endpoint
    • Static: fixed at compile time
    • Dynamic: performed at runtime

• **Dynamic Binding**
  • Most RPC systems use dynamic binding via name service
  • Why dynamic binding?
    • Access control: check who is permitted to access service
    • Fail-over: If server fails, use a different one

• Registry at server binds to RPC server stubs
Break
Do I need to implement RPC to use it?

• No! (Usually)
• Lot of existing RPC libraries
  • JSON RPC
  • XML RPC
  • Java RMI
  • Apache Thrift
  • REST
  • gRPC
Interface Definition Language

Pseudocode
protocol myprotocol {
    1: int32 mkdir(string name);
    2: int32 rmdir(string name);
}
Marshalling Example: mkdir("/directory/name")
returns 0
Client Sends: \001/directory/name\0
Server Sends: \0\0\0\0
Our Toy Example

Local Process

```c
char *l_mkdir(char *s) {
    send(svr, #1, s)
    res = rcv(srv)
    return res
}

int main( . . .) {
    st = l_mkdir(“cs162”);
}
```

Remote Server Process

```c
#include <sys/stat.h>

RPC_server() {
    while (req = rcv) {
        p = funcode(req)
        args = getargs(rq)
        res = RPC_Funs[p](args)
        reply(req, res)
    }
}

Char *r_rmdir(char * s) {
}

Char *r_mkdir(char * s) {
    stat = mkdir(s, mode);
    return stat
}
```
Interface Definition Language

• Compiled by a tool (e.g., gRPC) to generate stubs in various source languages (C, C++, Java, Python, …)

• Any client/server stubs can interact if they both adhere to the same protocol
  • Must be able to unmarshall what the other side marshalled

• Implementation language doesn't matter if we send right bits "over the wire"
  • And this is not specific to RPC
Network File System (NFS)

- Three Layers for NFS system
  - **UNIX file-system interface**: open, read, write, close calls + file descriptors
  - **VFS layer**: distinguishes local from remote files
    - Calls the NFS protocol procedures for remote requests
  - **NFS service layer**: bottom layer of the architecture
    - Implements the NFS protocol

- **NFS Protocol**: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
RPC: Really like a function call?

• What if something fails?
  
  \[
  \text{result} = \text{myprotocol\_mkdir}(\text{ctx}, \text{"/directory/name"});
  \]

• What should result be?

• Do we really know if the server made the directory on its side?
  
  • Maybe error occurred with server's file system?
  
  • Unrelated problem caused server to crash?

• If client doesn't hear back from server: did server crash or is it just taking a long time?
RPC: Really like a function call?

• What if we're doing remote file IO?
  
  remoteFile *rf = remoteio_open(ctx, "oski.txt");
  remoteio_puts(ctx, rf, "Bear\n");
  remoteio_close(ctx, rf);

• What if the client fails before it closes?
• Will the file be left open forever?

• Remember: local case is easy, OS cleans up after terminated processes
RPC: Really like a function call?

• Performance
• Cost of Procedure call << same-machine RPC << network RPC
• Means programmers must be aware they are using RPC (limits to transparency!)
  • Caching can help, but may make failure handling even more complex
Welcome to Distributed Systems

• Things get complicated when we have multiple machines in the picture!
  • Each can fail independently
  • Each has its own view of the world
    • Server: Client hasn't closed oski.txt, may still want to write
    • Client after crash: I need to open oski.txt again

• We'll study these and many other problems later!
Interlude: HTTP

• Application protocol for The Web
  • Retrieve a specific object, upload data, etc.

• Runs on top of TCP (sockets)

• Like any protocol, stipulates:
  • **Syntax**: Content sent over socket connection
  • **Semantics**: Meaning of a message
  • Valid replies and actions taken upon message receipt

• Arguably a primitive form of RPC
  • Parsing text, Hardcoded operations
  • No registry of available functions
  • No formal marshal/unmarshal
  • REST calls get a lot closer …
HTTP "RPC"

GET /search.html

200 OK
< HTML Content>
HTTP "RPC"

POST /users/oksi/photos
<image content>

201 Created
HTTP Messages

• Text-based: We just send character strings over our TCP socket connection
• To make a request, browser might write something like the following on a socket:

GET /hello.html HTTP/1.0\r\nHost: 128.32.4.8:8000\r\nAccept: text/html\r\nUser-Agent: Chrome/45.0.2454.93\r\nAccept-Language: en-US,en;q=0.8\r\n\r\n
HTTP Messages

• Text-based: We just send strings over our TCP socket connection
• We then read the following response from the web server:

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 128

<html>
<body>
  <h1>Hello World</h1>
  <p>Hello, World!</p>
</body>
</html>
```
HTTP and State

• Remember this situation?
  ```c
  remoteFile *rf = remoteio_open(ctx, "oski.txt");
  remoteio_puts(ctx, rf, "Bear\n");
  remoteio_close(ctx, rf);
  ```

• Client fails: does file stay open forever?
• Server *maintains* state between requests from client
HTTP and State

• HTTP avoids this issue – *stateless protocol*
• Each request is self-contained
  • Treated independently of all other requests
  • Even previous requests from same client!

• So how do we get a *session*?
  • Client stores a unique ID locally – a *cookie*
  • Client adds this to each request so server can customize its response
REST calls over http?

HTTP GET
http://www.appdomain.com/dirs/mkdir?name=cs162&mode=tmp
RPC Locally

- Doesn't need to be between different machines
- Could just be different address spaces (processes)

- Gives **location transparency**
  - Move service implementation to wherever is convenient
  - Client runs same old RPC code

- Much faster implementations available locally
  - (Local) Inter-process communication
  - We'll see several techniques later on
Microkernels

• Split OS into separate processes
  • Example: File System, Network Driver are external processes

• Pass messages among these components (e.g., via RPC) instead of system calls
Microkernels

- Microkernel itself provides only essential services
  - Communication
  - Address space management
  - Thread scheduling
  - Almost-direct access to hardware devices (for driver processes)
Why Microkernels?

Pros
• Failure Isolation
• Easier to update/replace parts
• Easier to distribute – build one OS that encompasses multiple machines

Cons
• More communication overhead and context switching
• Harder to implement?
Flashback: What is an OS?

- Always:
  - Memory Management
  - I/O Management
  - CPU Scheduling
  - Communications
  - Multitasking/multiprogramming

- Maybe:
  - File System?
  - Multimedia Support?
  - User Interface?
  - Web Browser?

*Not provided in a strict microkernel*
Influence of Microkernels

• Many operating systems provide some services externally, similar to a microkernel
  • OS X and Linux: Windowing (graphics and UI)

• Some currently monolithic OSs started as microkernels
  • Windows family originally had microkernel design
  • OS X: Hybrid of Mach microkernel and FreeBSD monolithic kernel
Summary

• Remote Procedure Call: Invoke a procedure on a remote machine
  • Provides same interface as normal function call
  • Automatic packing and unpacking of user arguments

• Microkernels: Run system services outside of kernel

• HTTP: Application Layer Protocol
  • Like RPC, but stateless

• Domain Name Service: Map names to IP addresses
  • Hierarchical organization among many servers