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)
}
```

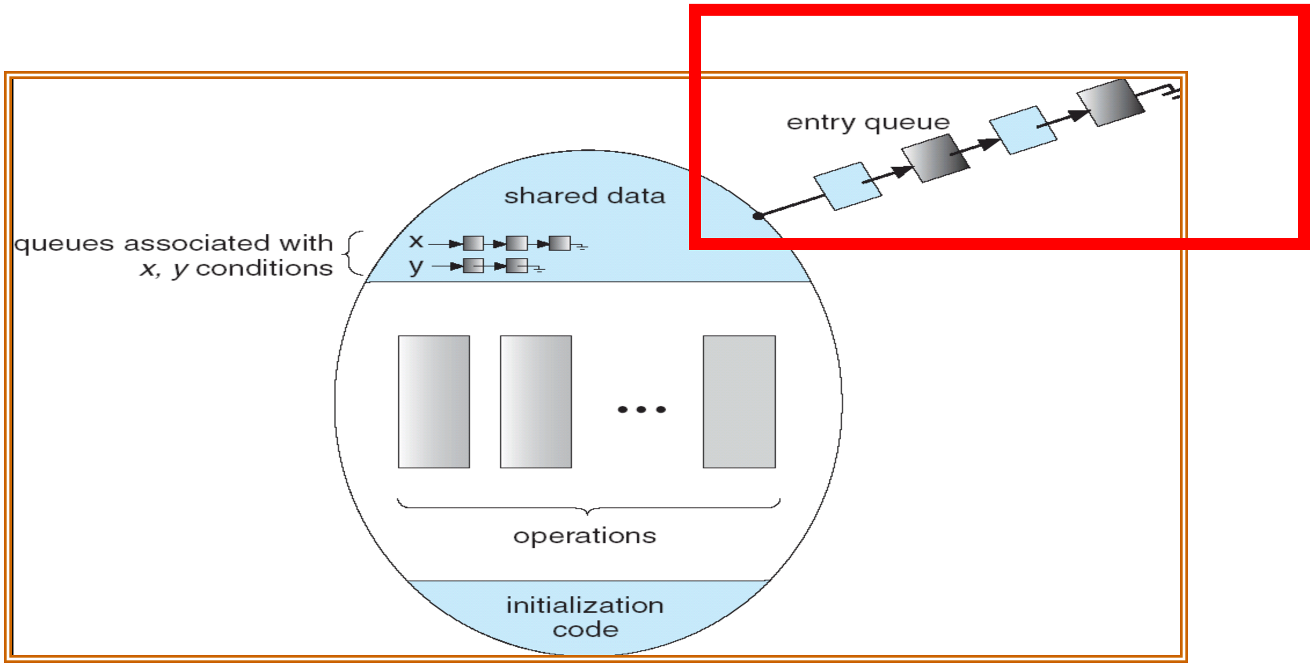
Release lock; signal others to run; reacquire on resume

n.b. OS must do the reacquire

Why User must recheck?

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

Monitors

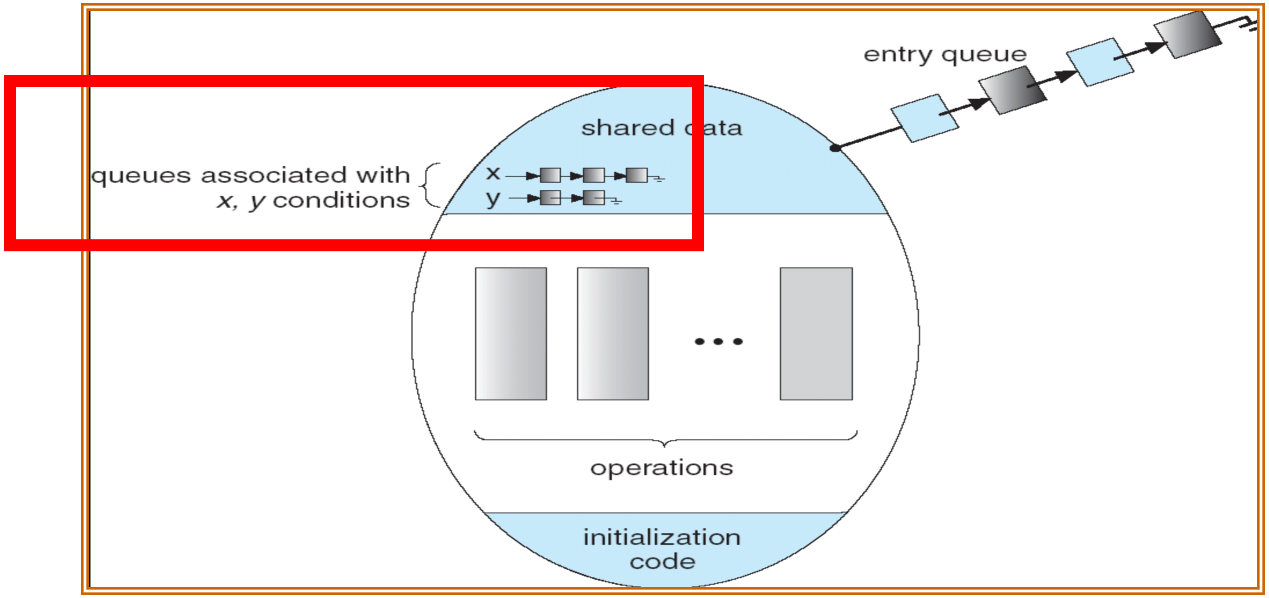


- **Lock:** protects access to shared data
- Always acquire lock when accessing
- Queue of threads waiting to enter the monitor

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?

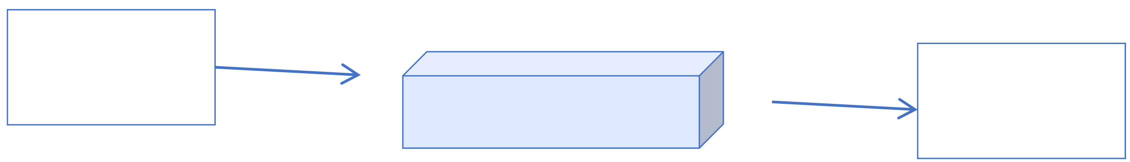
IPC/RPC Background

- 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?

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



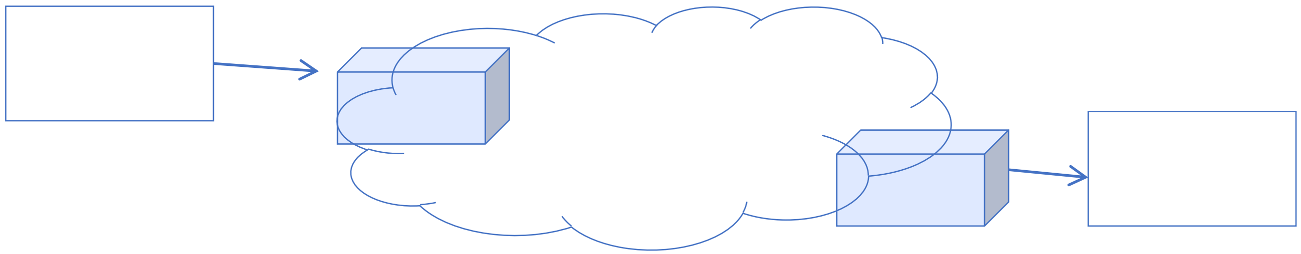
```
n = read(rfd,rbuf,rmax);
```

- We have seen one example – pipes
- Routinely used with the shell

```
>>>  grep list src/*/*.c | more
```

Recall: Communication Across the world looks like file IO

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

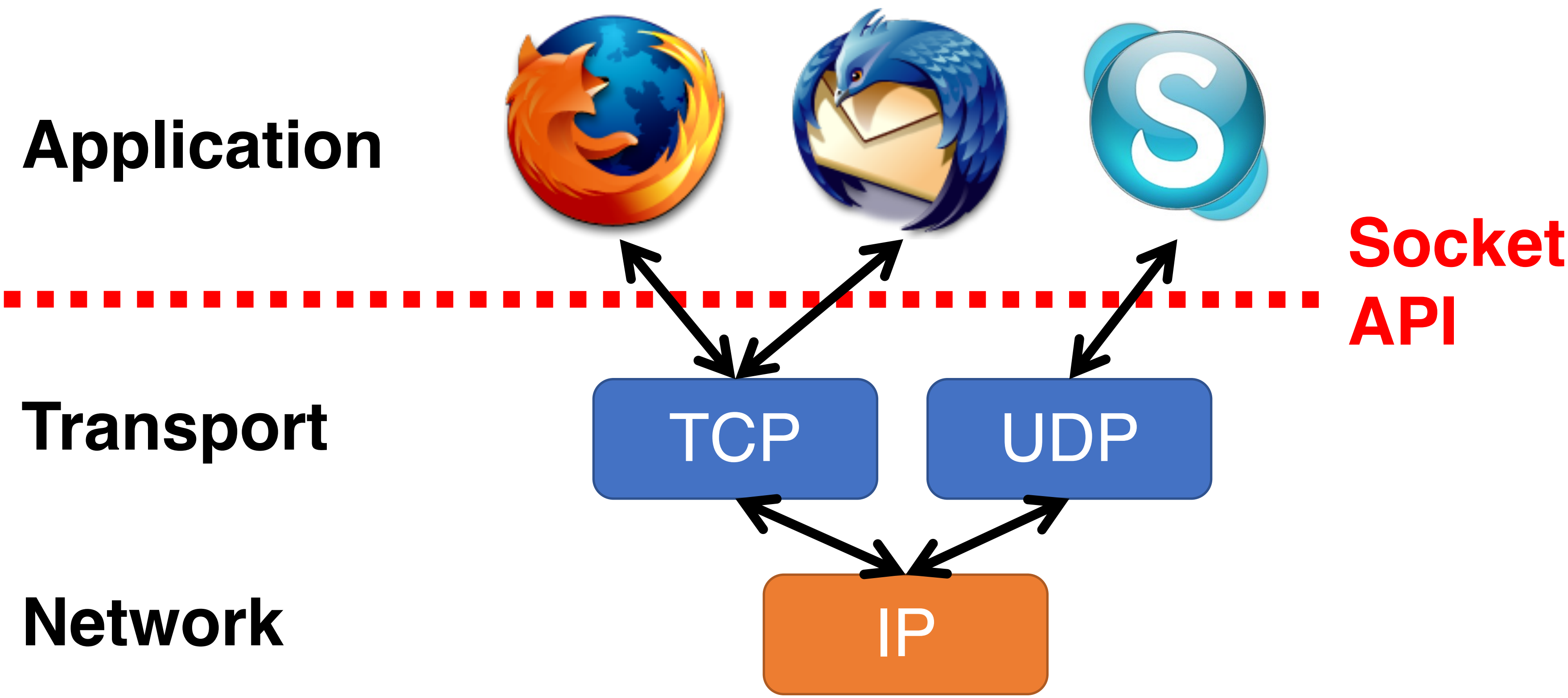


```
n = read(rfd,rbuf,rmax);
```

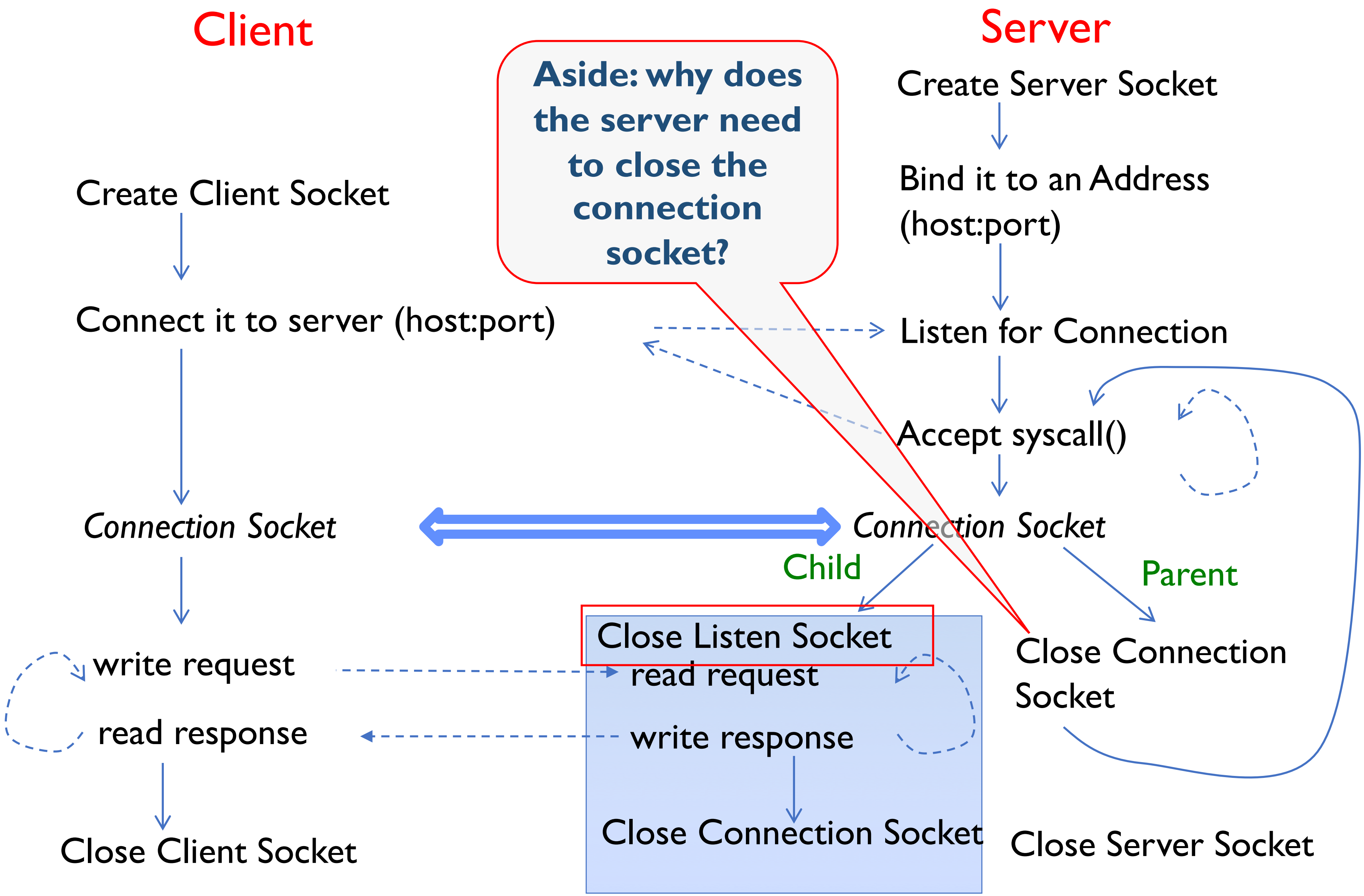
- Connected queues over the Internet
  - But what's the analog of open?
  - What is the namespace?
  - How are they connected in time?

Socket API

• Base level Network programming interface



Recall: Sockets w/ Protection & Parallelism



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

Processor	Endianness
Motorola 68000	Big Endian
PowerPC (PPC)	Big Endian
Sun Sparc	Big Endian
IBM S/390	Big Endian
Intel x86 (32 bit)	Little Endian
Intel x86_64 (64 bit)	Little Endian
Dec VAX	Little Endian
Alpha	Bi (Big/Little) Endian
ARM	Bi (Big/Little) Endian
IA-64 (64 bit)	Bi (Big/Little) Endian
MIPS	Bi (Big/Little) Endian

```
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?

- Big Endian
- Network byte order
- vs "host byte order"

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 ⌫ 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

```
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.

```
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>* heade

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

```
(base) CullerMac19:code09 culler$ ./io
Wrote 100 ints
(base) CullerMac19:code09 culler$ cat intdata.bin
        $1@Qdy????!Di???@q??I??A??Y??@??9???    a       ?
?
?
d
 ?
  @
   ?
?????$?@?d??)?a?D????@??Y !?!?"A#$?$?%I&(base) CullerMac19
(base) CullerMac19:code09 culler$ hexdump intdata.bin
0000000 00 00 00 00 01 00 00 00 04 00 00 00 09 00 00 00
0000010 10 00 00 00 19 00 00 00 24 00 00 00 31 00 00 00
0000020 40 00 00 00 51 00 00 00 64 00 00 00 79 00 00 00
0000030 90 00 00 00 a9 00 00 00 c4 00 00 00 e1 00 00 00
0000040 00 01 00 00 21 01 00 00 4
0000050 90 01 00 00 b9 01 00 00 e
0000060 40 02 00 00 71 02 00 00 a
0000070 10 03 00 00 49 03 00 00 8
0000080 00 04 00 00 41 04 00 00 8
0000090 10 05 00 00 59 05 00 00 a
00000a0 40 06 00 00 91 06 00 00 e
```

```c
#include <stdlib.h>
#include <stdio.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);
}
```

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

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

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

- `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?

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


```
{
    "glossary": {
        "title": "example glossary",
                "GlossDiv": {
            "title": "S",
                        "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                                        "SortAs": "SGML",
                                        "GlossTerm": "Standard Generalized Markup Language",
                                        "Acronym": "SGML",
                                        "Abbrev": "ISO 8879:1986",
                                        "GlossDef": {
                        "para": "A meta-markup language, used to create markup languages such as DocBook.",
                                                "GlossSeeAlso": ["GML", "XML"]
                    },
                                        "GlossSee": "markup"
                }
            }
        }
    }
}
```

```
<!DOCTYPE glossary PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
 <glossary><title>example glossary</title>
  <GlossDiv><title>S</title>
   <GlossList>
    <GlossEntry ID="SGML" SortAs="SGML">
     <GlossTerm>Standard Generalized Markup Language</GlossTerm>
     <Acronym>SGML</Acronym>
     <Abbrev>ISO 8879:1986</Abbrev>
     <GlossDef>
      <para>A meta-markup language, used to create markup
languages such as DocBook.</para>
      <GlossSeeAlso OtherTerm="GML">
      <GlossSeeAlso OtherTerm="XML">
     </GlossDef>
     <GlossSee OtherTerm="markup">
    </GlossEntry>
   </GlossList>
  </GlossDiv>
 </glossary>
```

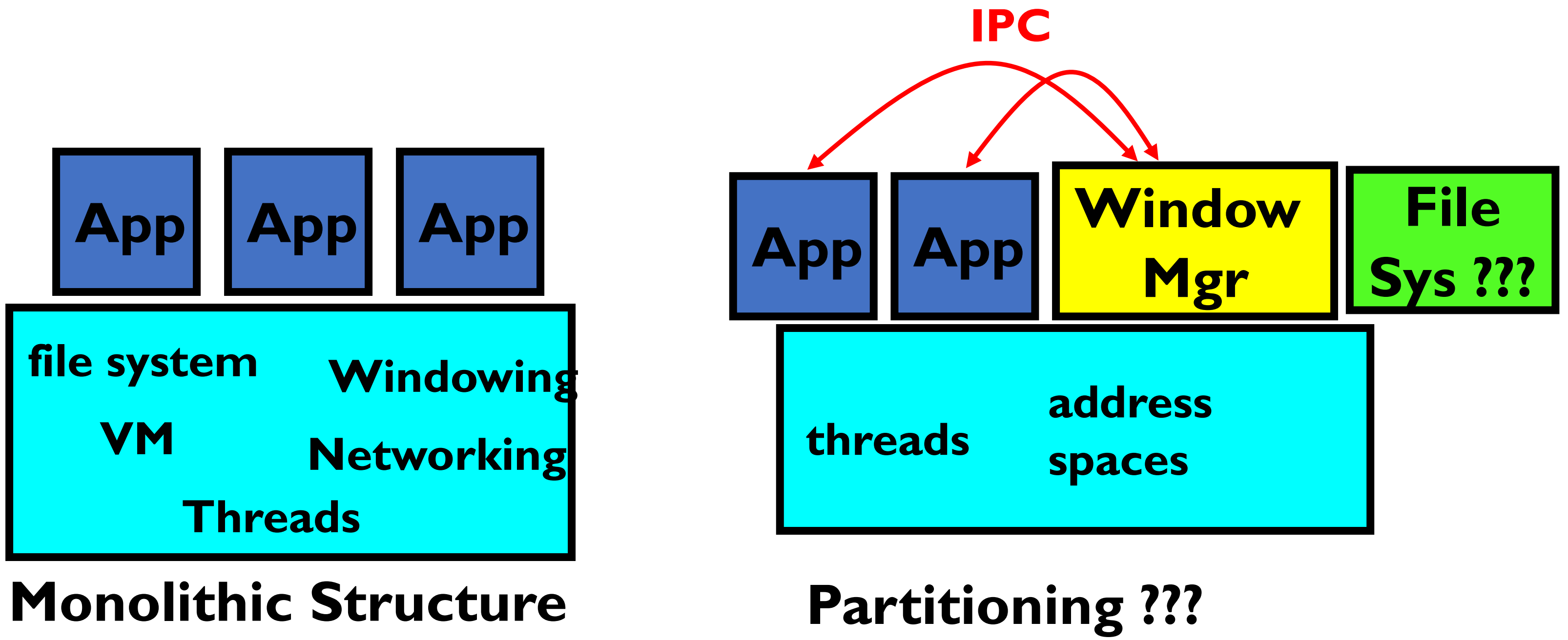
Data Serialization Formats

Name	Creator-maintainer	Based on	Standardized?	Specification	Binary?	Human-readable?	Supports references?[e]	Schema-IDL?	Standard APIs	Supports Zero-copy operations [hide]
Apache Avro	Apache Software Foundation	N/A	No	Apache Avro™ 1.8.1 Specification	Yes	No	N/A	Yes (built-in)	N/A	N/A
Apache Parquet	Apache Software Foundation	N/A	No	Apache Parquet[1]	Yes	No	No	N/A	Java, Python	No
ASN.1	ISO, IEC, ITU-T	N/A	Yes	ISO/IEC 8824; X.680 series of ITU-T Recommendations	Yes (BER, DER, PER, OER, or custom via ECN)	Yes (XER, JER, GSER, or custom via ECN)	Partial[f]	Yes (built-in)	N/A	Yes (OER)
Bencode	Bram Cohen (creator) BitTorrent, Inc. (maintainer)	N/A	*De facto* standard via BitTorrent Enhancement Proposal (BEP)	Part of BitTorrent protocol specification	Partially (numbers and delimiters are ASCII)	No	No	No	No	N/A
Binn	Bernardo Ramos	N/A	No	Binn Specification	Yes	No	No	No	No	Yes
BSON	MongoDB	JSON	No	BSON Specification	Yes	No	No	No	No	N/A
CBOR	Carsten Bormann, P. Hoffman	JSON (loosely)	Yes	RFC 7049	Yes	No	Yes through tagging	Yes (CDDL)	No	Yes
Comma-separated values (CSV)	RFC author: Yakov Shafranovich	N/A	Partial (myriad informal variants used)	RFC 4180 (among others)	No	Yes	No	No	No	No
Common Data Representation (CDR)	Object Management Group	N/A	Yes	General Inter-ORB Protocol	Yes	No	Yes	Yes	ADA, C, C++, Java, Cobol, Lisp, Python, Ruby, Smalltalk	N/A
D-Bus Message Protocol	freedesktop.org	N/A	Yes	D-Bus Specification	Yes	No	No	Partial (Signature strings)	Yes (see D-Bus)	N/A
Efficient XML Interchange (EXI)	W3C	XML, Efficient XML	Yes	Efficient XML Interchange (EXI) Format 1.0	Yes	Yes (XML)	Yes (XPointer, XPath)	Yes (XML Schema)	Yes (DOM, SAX, StAX, XQuery, XPath)	N/A
FlatBuffers	Google	N/A	No	flatbuffers github page Specification	Yes	Yes (Apache Arrow)	Partial (internal to the buffer)	Yes [2]	C++, Java, C#, Go, Python, Rust, JavaScript, PHP, C, Dart, Lua, TypeScript	Yes
Fast Infoset	ISO, IEC, ITU-T	XML	Yes	ITU-T X.891 and ISO/IEC 24824-1:2007	Yes	No	Yes (XPointer, XPath)	Yes (XML schema)	Yes (DOM, SAX, XQuery, XPath)	N/A
FHIR	Health_Level_7	REST basics	Yes	Fast Healthcare Interoperability Resources	Yes	Yes	Yes	Yes	Hapi for FHIR[1] JSON, XML, Turtle	No
Ion	Amazon	JSON	No	The Amazon Ion Specification	Yes	Yes	No	No	No	N/A
Java serialization	Oracle Corporation	N/A	Yes	Java Object Serialization	Yes	No	Yes	No	Yes	N/A
JSON	Douglas Crockford	JavaScript syntax	Yes	STD 90/RFC 8259 (ancillary: RFC 6901, RFC 6902), ECMA-404, ISO/IEC 21778:2017	No, but see BSON, Smile, UBJSON	Yes	Yes (JSON Pointer (RFC 6901); alternately: JSONPath, JPath, JSPON, json:select()), JSON-LD	Partial (JSON Schema Proposal, ASN.1 with JER, Kwalify, Rx, Itemscript Schema), JSON-LD	Partial (Clarinet, JSONQuery, JSONPath), JSON-LD	No
MessagePack	Sadayuki Furuhashi	JSON (loosely)	No	MessagePack format specification	Yes	No	No	No	No	Yes
Netstrings	Dan Bernstein	N/A	No	netstrings.txt	Yes	Yes	No	No	No	Yes
OGDL	Rolf Veen	?	No	Specification	Yes (Binary Specification)	Yes	Yes (Path Specification)	Yes (Schema WD)		N/A
OPC-UA Binary	OPC Foundation	N/A	No	opcfoundation.org	Yes	No	Yes	No	No	N/A
OpenDDL	Eric Lengyel	C, PHP	No	OpenDDL.org	No	Yes	Yes	No	Yes (OpenDDL Library)	N/A
Pickle (Python)	Guido van Rossum	Python	*De facto* standard via Python Enhancement Proposals (PEPs)	[3] PEP 3154 -- Pickle protocol version 4	Yes	No	No	No	Yes ([4])	No
Property list	NeXT (creator) Apple (maintainer)	?	Partial	Public DTD for XML format	Yes[a]	Yes[b]	No	?	Cocoa, CoreFoundation, OpenStep, GnuStep	No
Protocol Buffers (protobuf)	Google	N/A	No	Developer Guide: Encoding	Yes	Partial[d]	No	Yes (built-in)	C++, C#, Java, Python, Javascript, Go	No
	John McCarthy					Yes				

Inter-Process Communication (IPC)

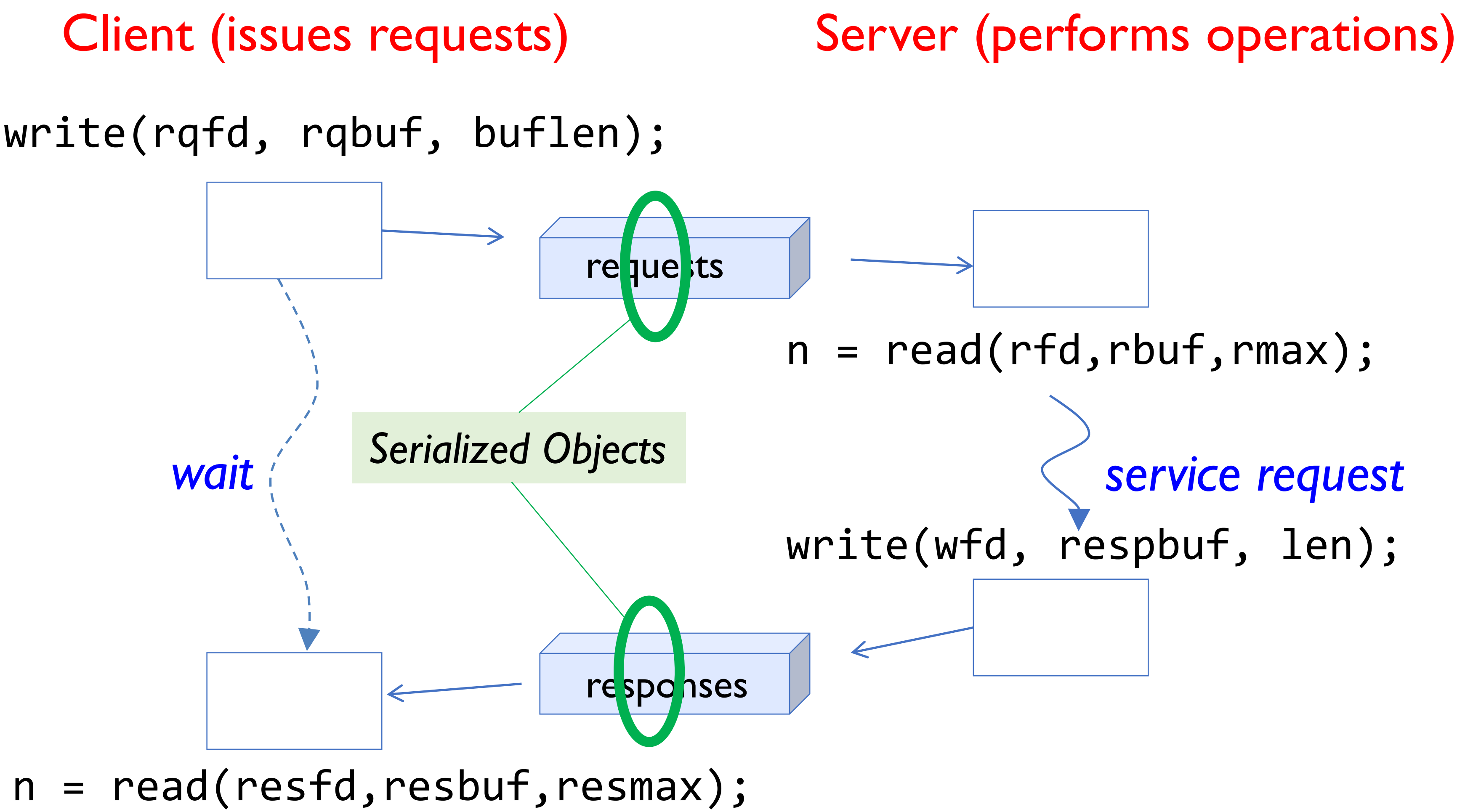
- Mechanism to create a communication channel between distinct processes
  - User to User, Kernel to User, Same machine or different ones, Different programming languages, …
- Serialization format understand by both
- Can have authentication and authorization mechanism associated with it
- Failure in 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?

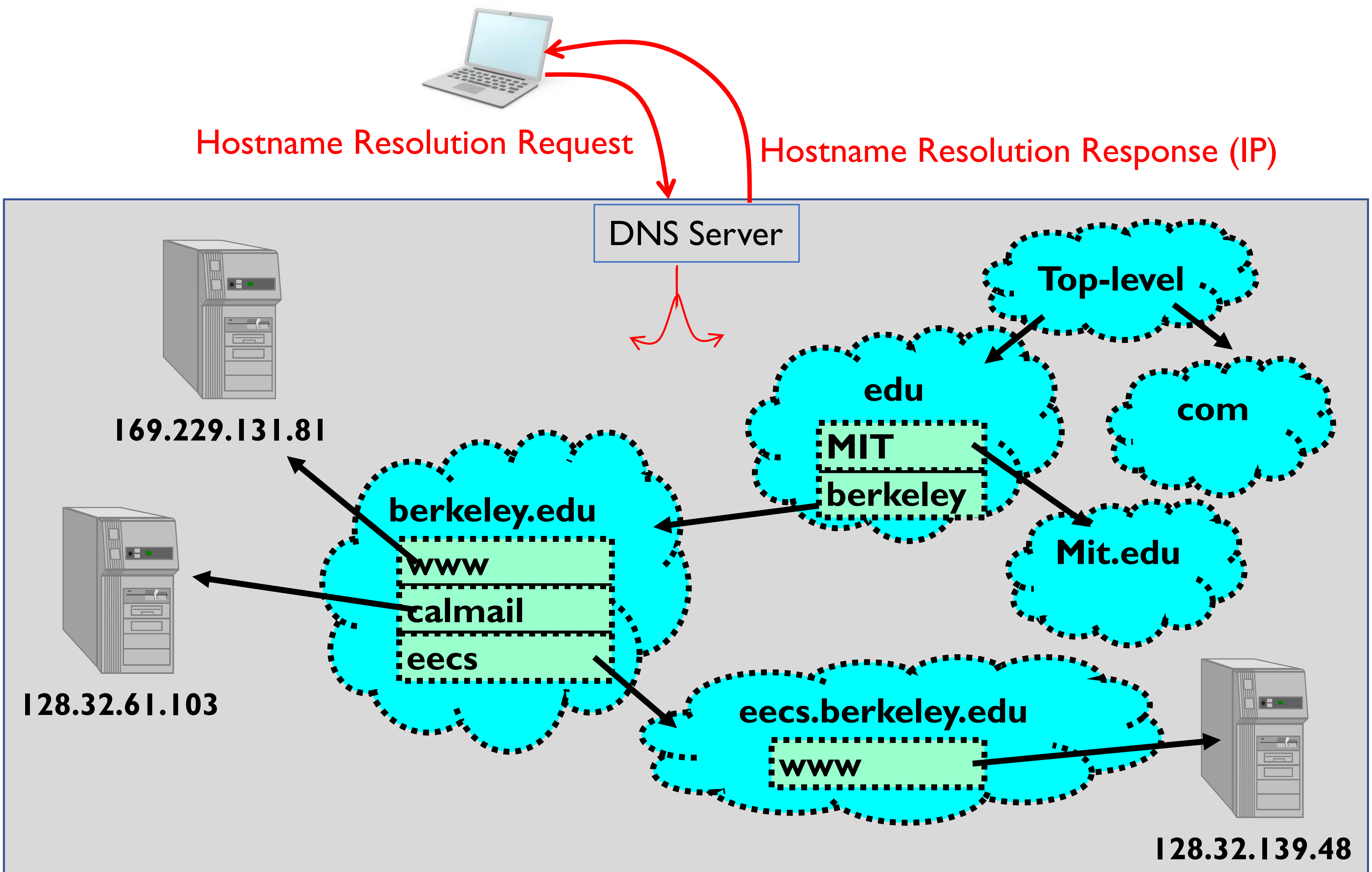
Recall: Request/Response Protocol



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 Server
Top-level
edu
MIT
berkeley
com
Mit.edu
berkeley.edu
www
calmail
eecs
eecs.berkeley.edu
www
169.229.131.81
128.32.61.103
128.32.139.48

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?

p2
p1

Stack call frame

```
main( … ) {

  “…”;

  res = libr_fun(p1, p2,…);


  “do something with results”
}
```

```
result_t libr_fun(arg_t arg1, … ) {

  “access the arguments”;

  “do something useful for caller”

  “return results”
}
```

res

Return Results

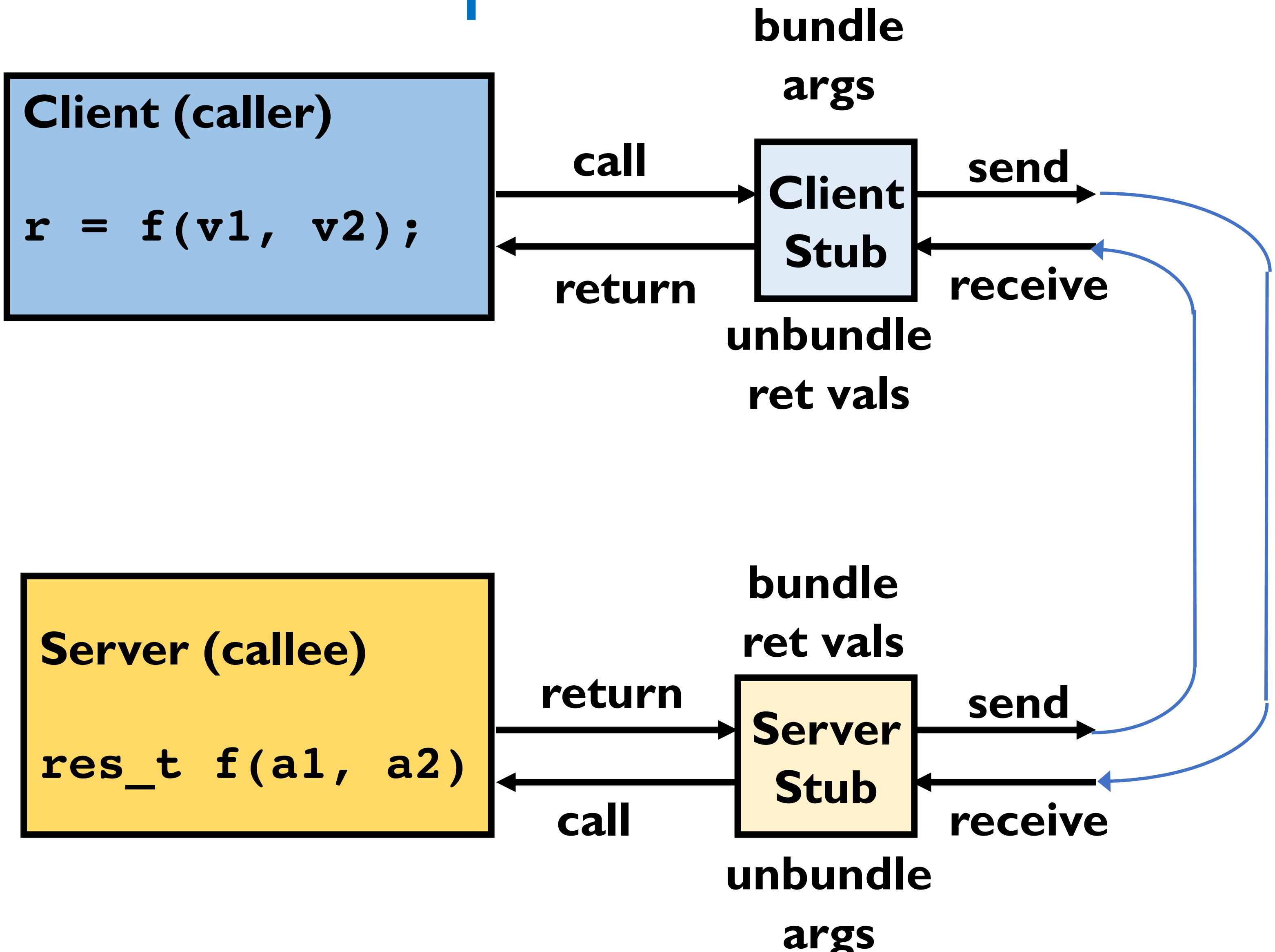
And aren't system calls doing this ?

- 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

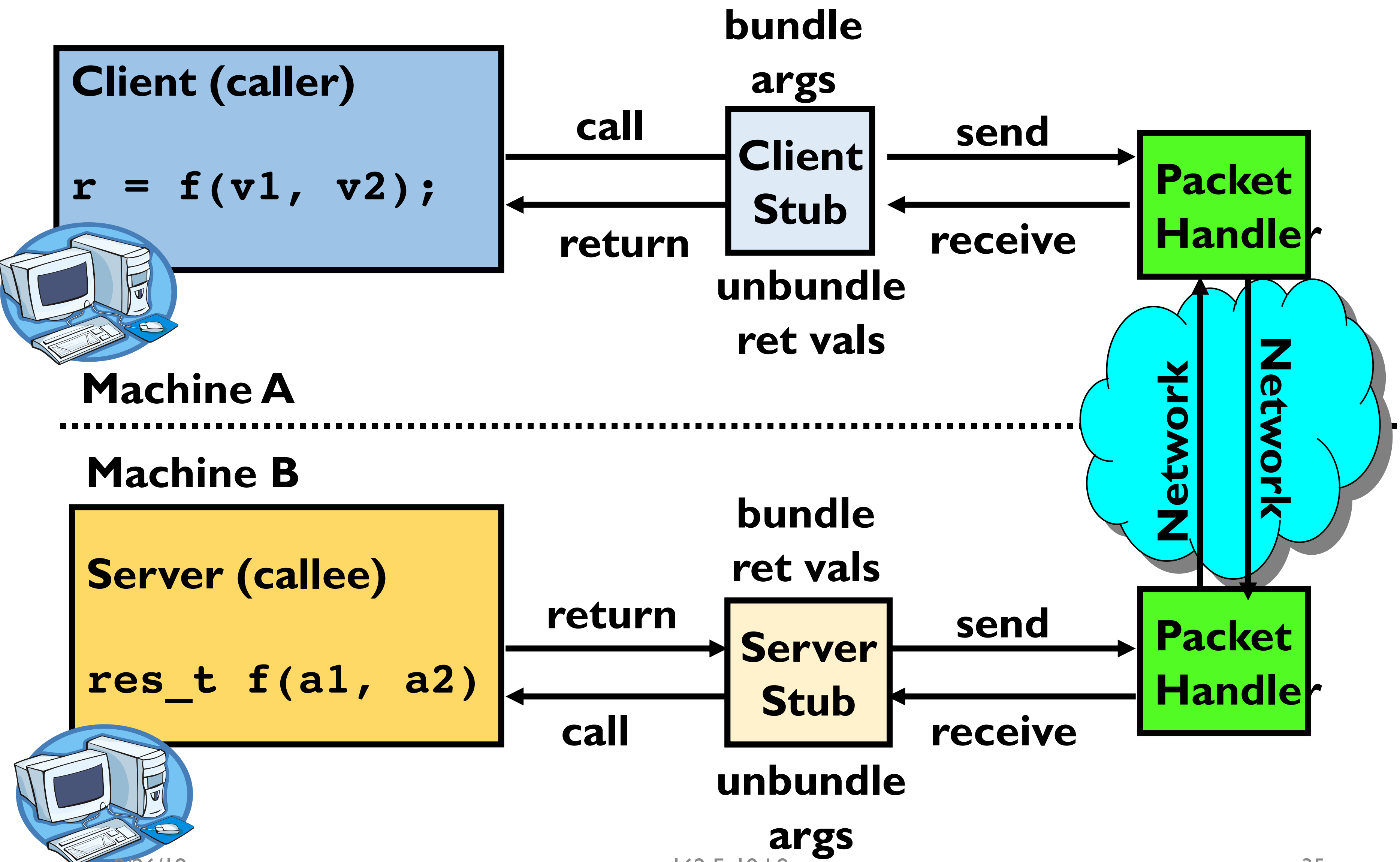
Remote Procedure Call (RPC)

- 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


bundle
args
Client (caller)
r = f(v1, v2);
call
return
Client
Stub
send
receive
unbundle
ret vals
bundle
ret vals
Server (callee)
res_t f(a1, a2)
return
call
Server
Stub
send
receive
unbundle
args

RPC Information Flow



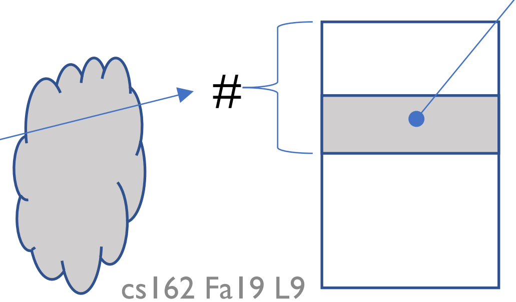
Six steps

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

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




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

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

Remote Server Process

```
#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
}
```

```
char *l_mkdir(char *s) {

   send(svr, #1, s)
   res = rcv(srv)

   return res
}

int main( . . .) {

   st = l_mkdir("cs162");

}
```

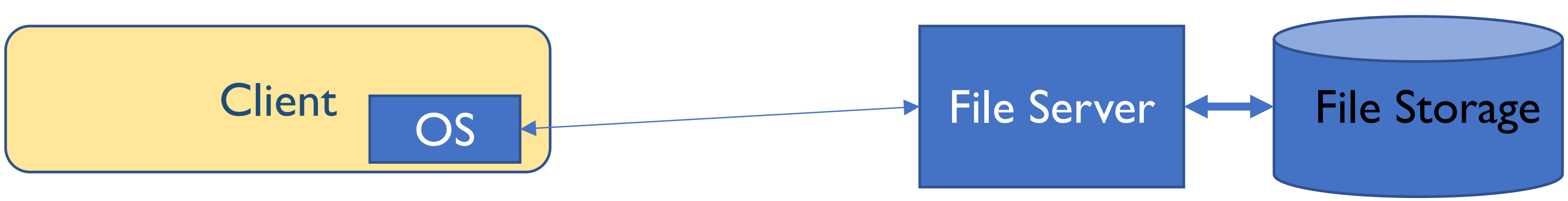
Local Process

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?

```
result = myprotocol_mkdir(ctx, "/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

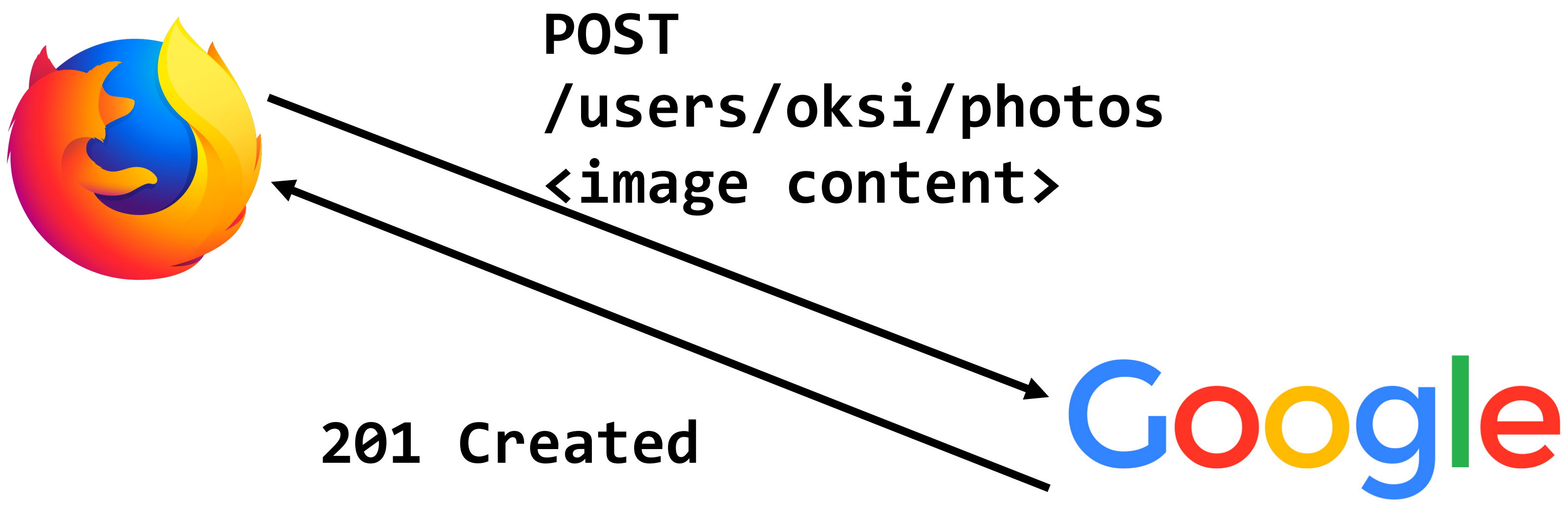
- Application protocol for The Web
  - Retrieve a specific object, upload data, etc.
- Runs on top of TCP (sockets)
- Like any protocol, stipulates:
  - **Syntax**: Content sent over socket connection
  - **Semantics**: Meaning of a message
  - Valid replies and actions taken upon message receipt
- Arguably 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"



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\n
Host: 128.32.4.8:8000\r\n
Accept: text/html\r\n
User-Agent: Chrome/45.0.2454.93\r\n
Accept-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\r\n
Content-Type: text/html\r\n
Content-Length: 128\r\n
\r\n
<html>\n
<body>\n
  <h1>Hello World</h1>\n
  <p>\n
    Hello, World!\n
  </p>\n
</body>\n
</html>\n
```

HTTP and State

- Remember this situation?

```
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
```

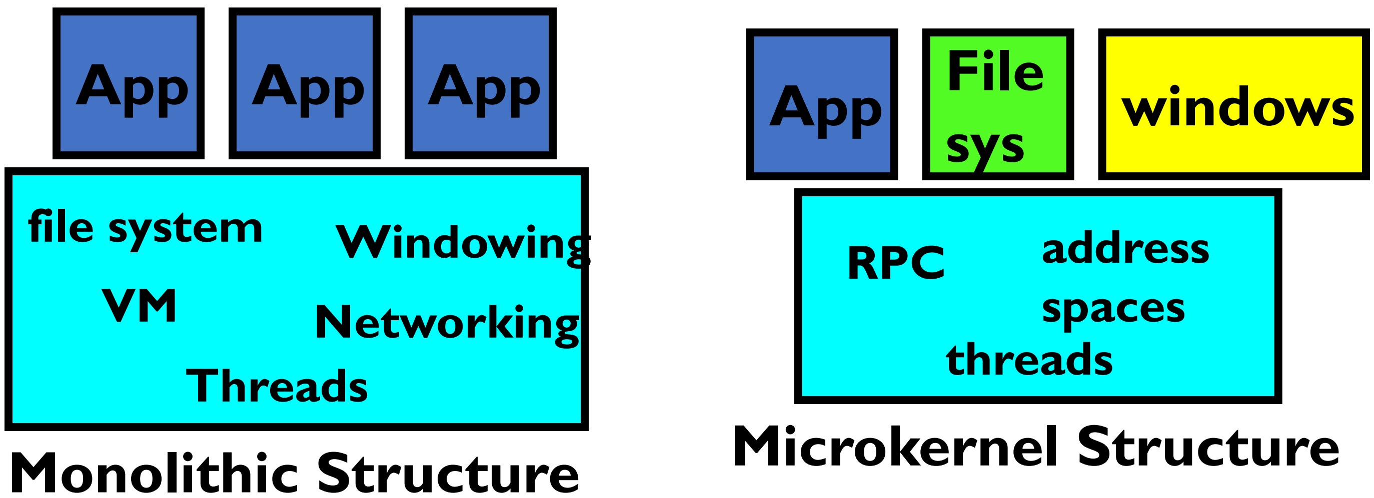
RPC Locally

- 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

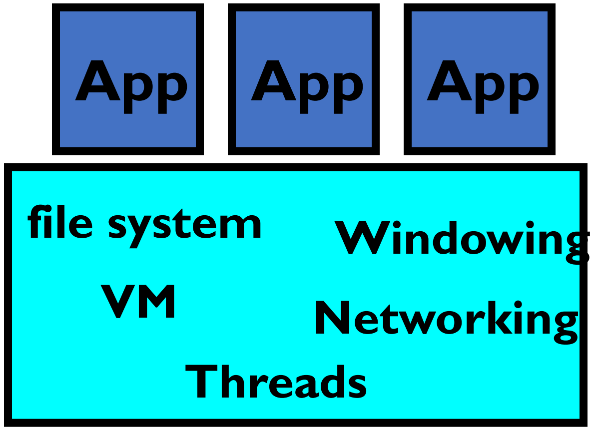
Microkernels

- 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

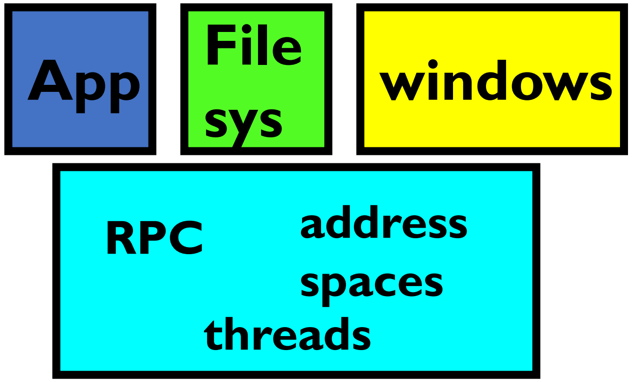


Microkernels

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



Monolithic Structure



Microkernel Structure

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