Networking via TCP

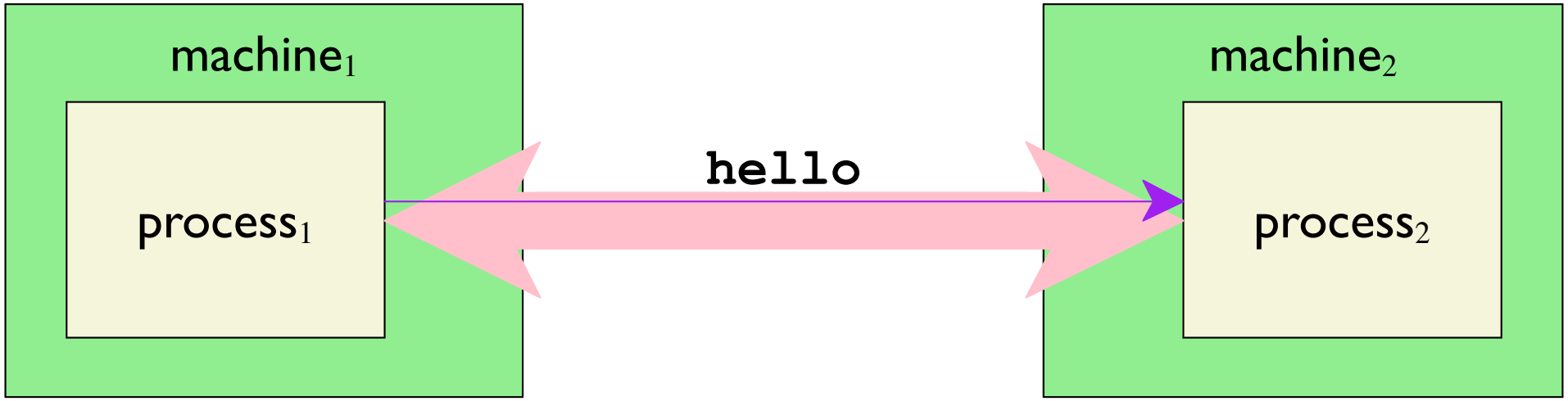
Networking via TCP

Networking via TCP

 = `tcp_open(....);`

 = `tcp_open(....);`

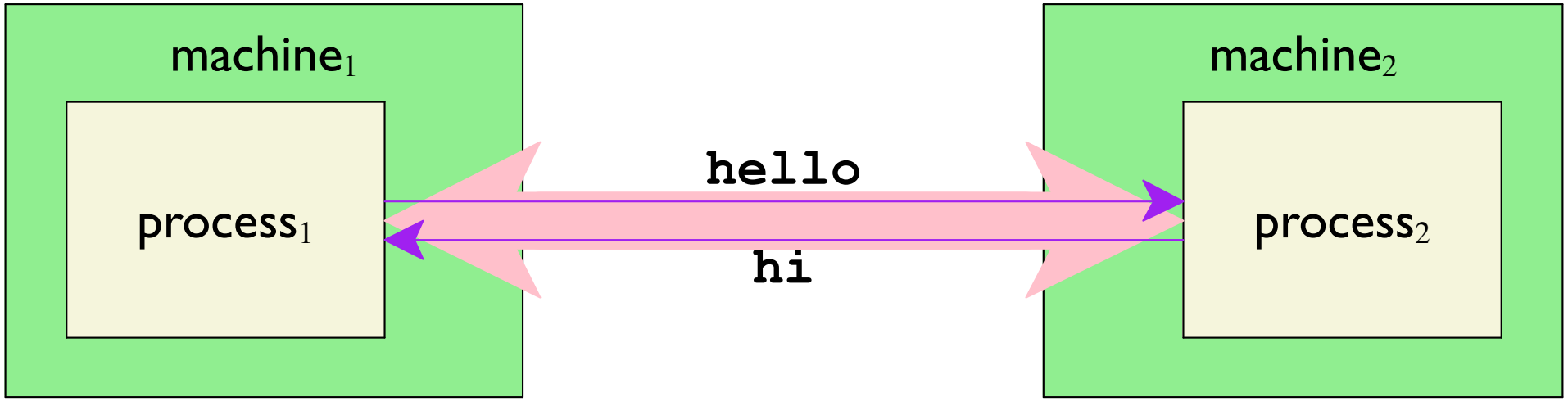
Networking via TCP



↔ = `tcp_open(....);` ↔ = `tcp_open(....);`

`Write(↔, "hello", 5);` `Read(↔, buffer, 5);`

Networking via TCP

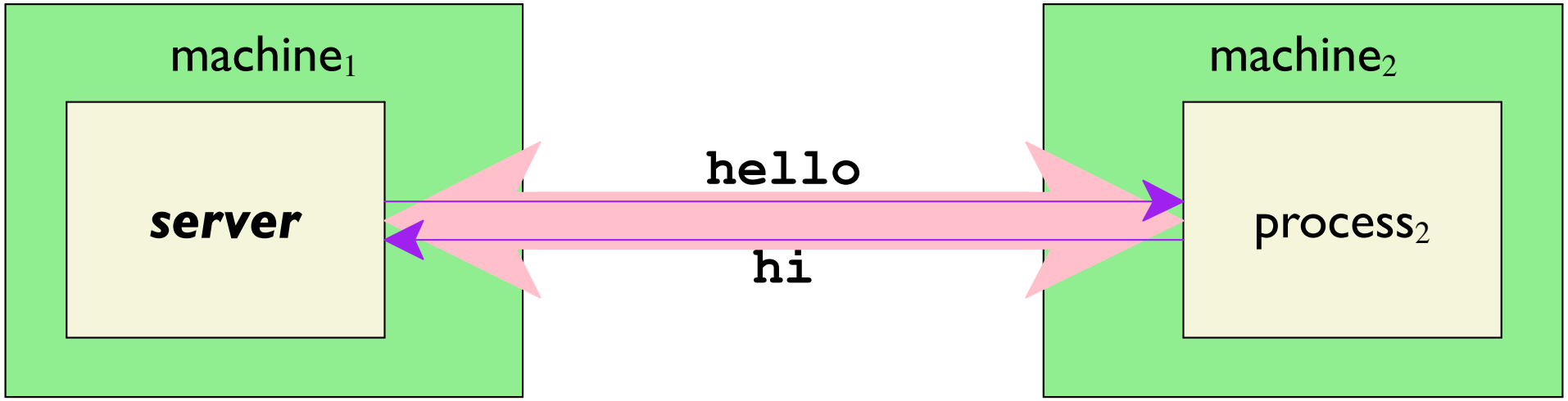


↔ = `tcp_open(....);` ↔ = `tcp_open(....);`

```
Write(↔, "hello", 5);
Read(↔, buffer, 2);
```

```
Read(↔, buffer, 5);
Write(↔, "hi", 2);
```

Networking via TCP

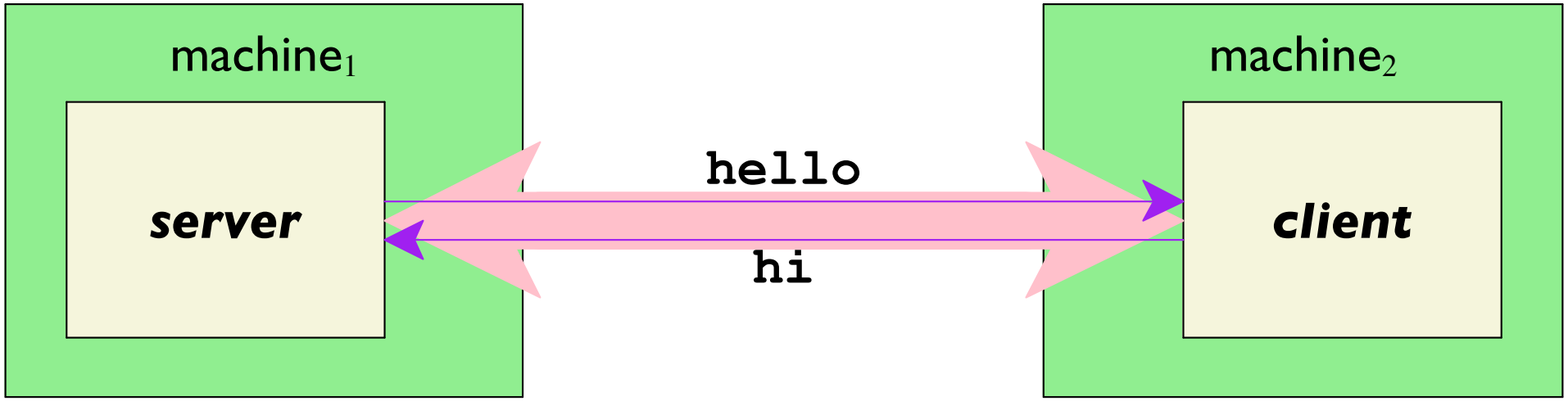



```
lnr = Open_listenfd(....);
    = Accept(lnr, ....);
```

```
Write(    , "hello", 5);
Read(     , buffer, 2);
```

```
Read(     , buffer, 5);
Write(    , "hi", 2);
```

Networking via TCP




```
lnr = Open_listenfd(....);
      = Accept(lnr, ....);
```

```
                                  = Open_clientfd(....
```

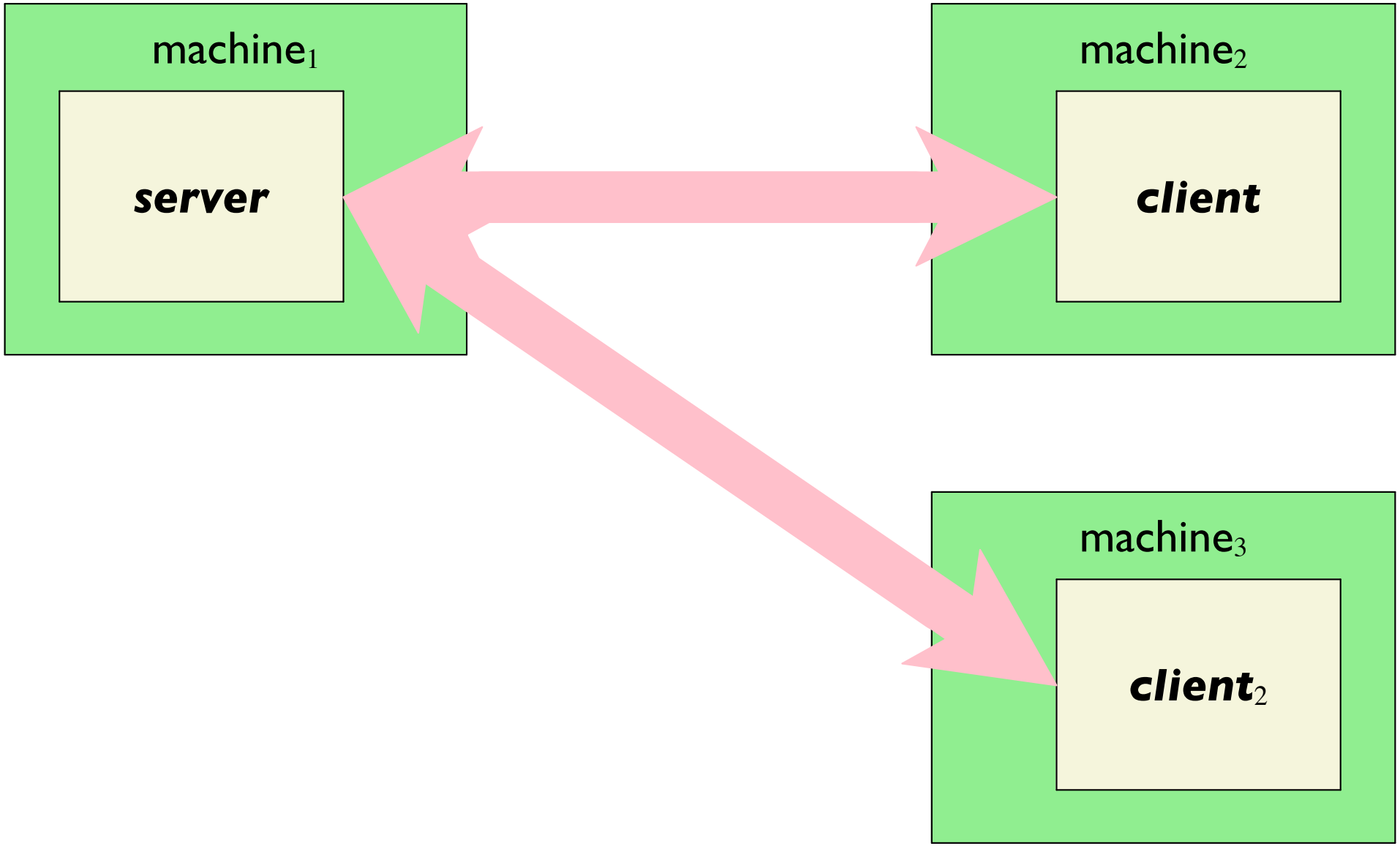
```
Write(      , "hello", 5);
```

```
Read(      , buffer, 2);
```

```
Read(      , buffer, 5);
```

```
Write(      , "hi", 2);
```

Networking via TCP



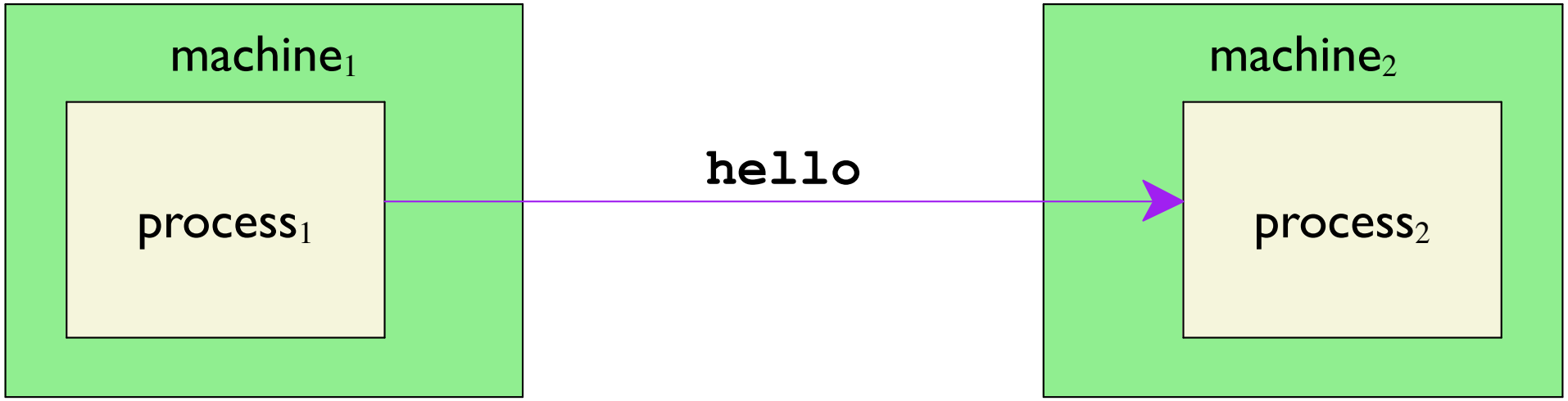
Networking via UDP

Networking via UDP


```
➤ = Socket(....);
Bind(➤, ....);
```

```
◀ = Socket(....);
Bind(◀, ....);
```

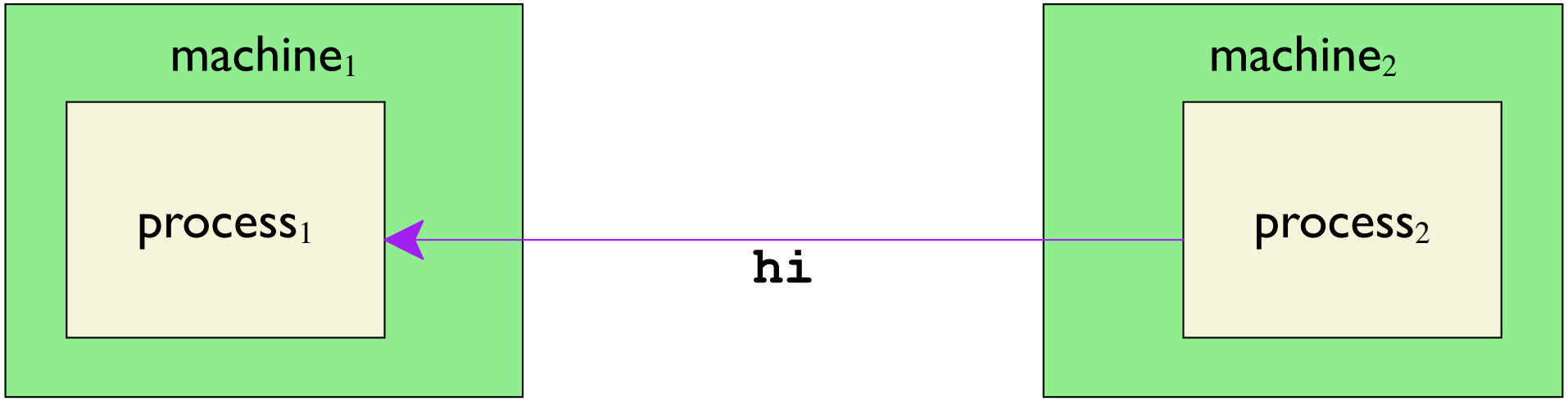
Networking via UDP




```
➤ = Socket(....);
Bind(➤, ....);

Sendto(➤, "hello", 5, ...);
```

```
➤ = Socket(....);
Bind(➤, ....);

Recv(➤, buffer, 5, 0);
```

Networking via UDP




```
➤ = Socket(....);
Bind(➤, ....);

Recv(➤, buffer, 2);
```

```
➤ = Socket(....);
Bind(➤, ....);

Sendto(➤, "hi", 2, ...);
```

TCP vs. UDP

TCP

Connection- and stream-oriented

Reliable

The most widely used networking protocol

UDP

Connectionless and packet-oriented

Best-effort

Minimal structure over next primitive layer

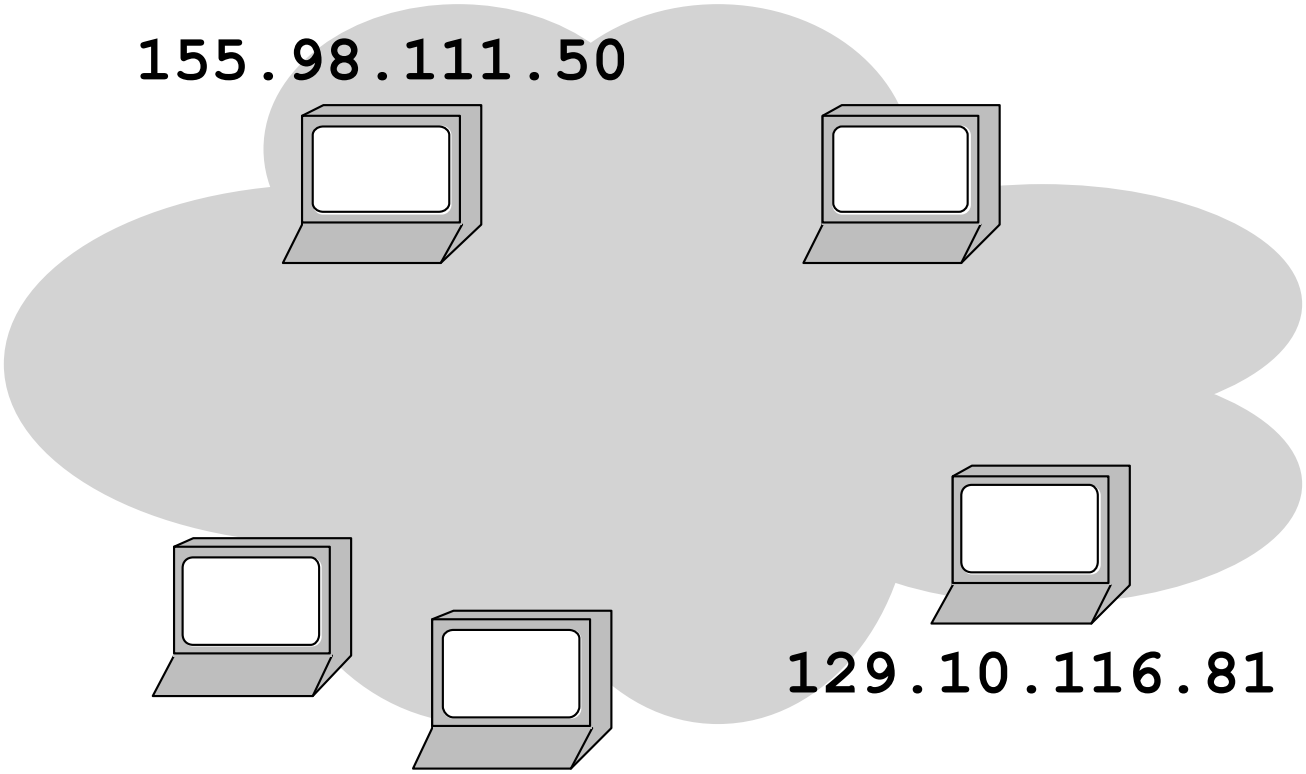
Both built on ***IP***

Finding Hosts on a Network

Using ***IP***, a ***host*** is named by a 32-bit value

More precisely, this is IPv4

Written as dot-separated, unsigned 8-bit values

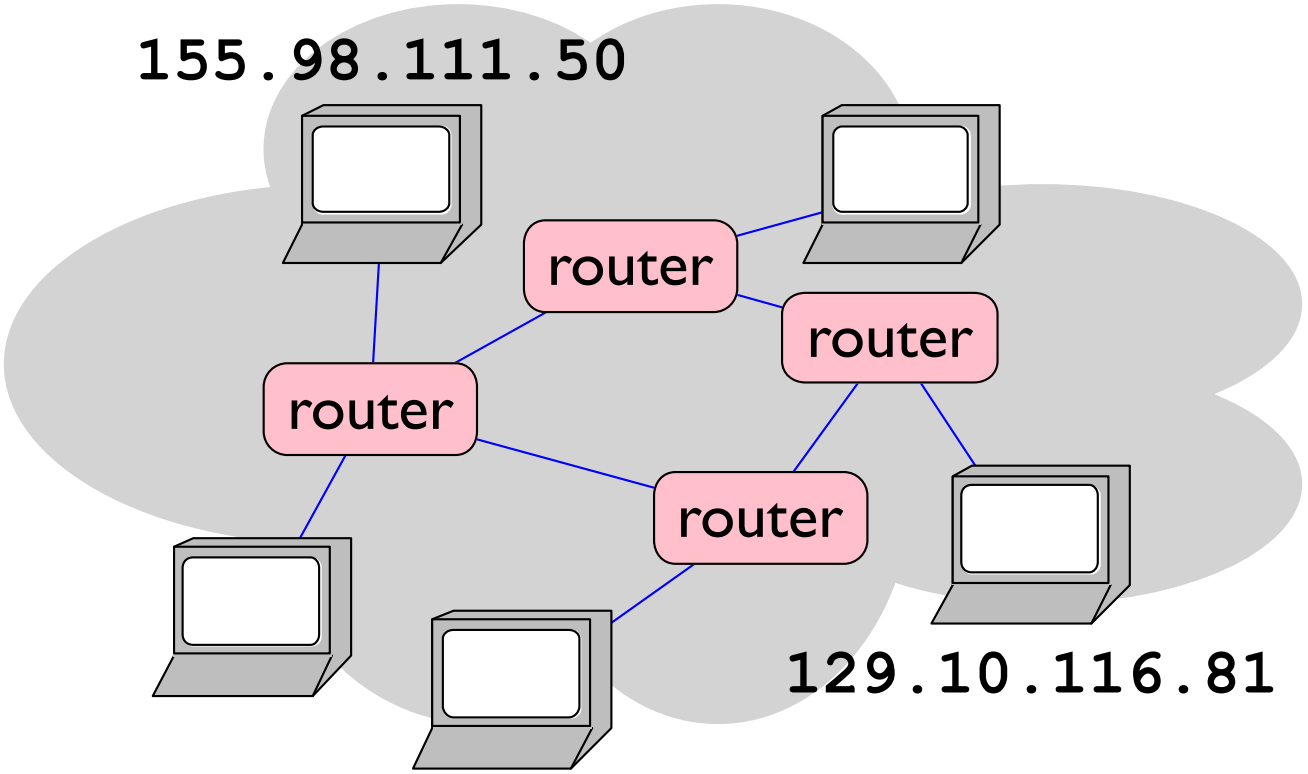


Finding Hosts on a Network

Using ***IP***, a ***host*** is named by a 32-bit value

More precisely, this is IPv4

Written as dot-separated, unsigned 8-bit values

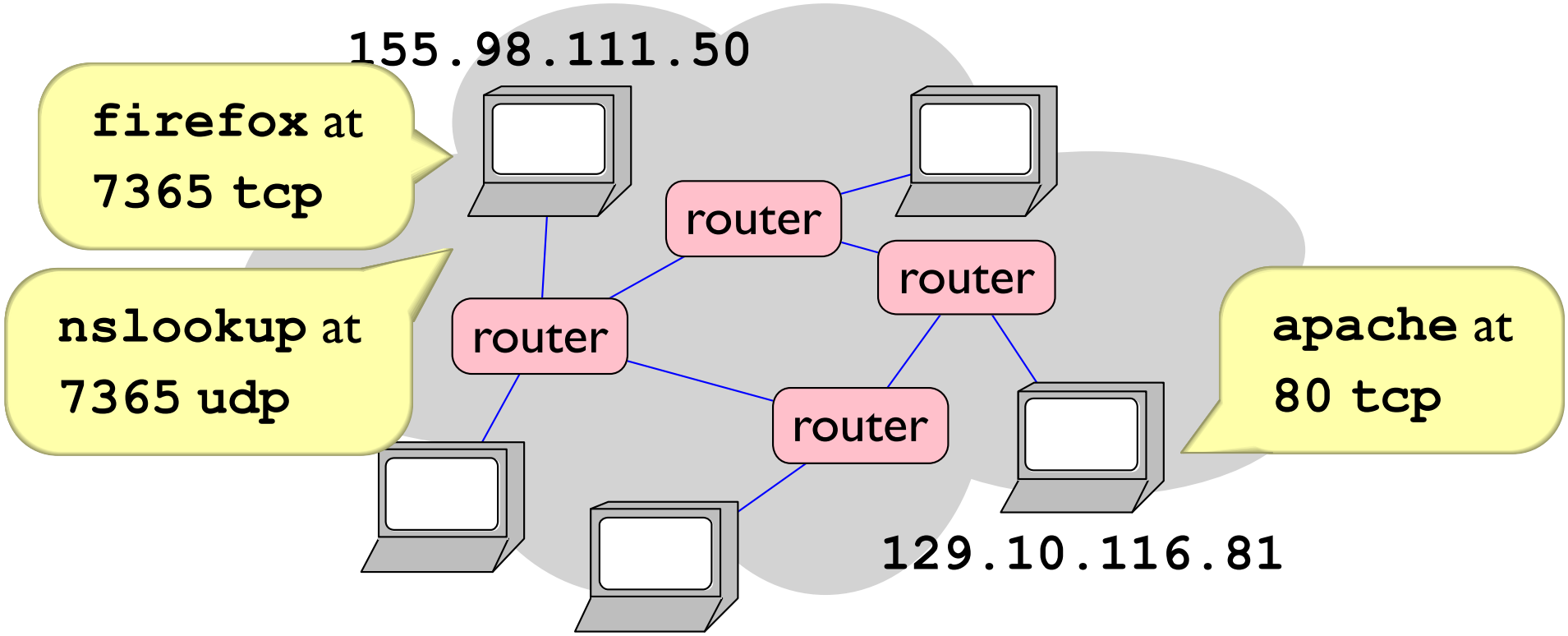


Finding Hosts on a Network

Using ***IP***, a ***host*** is named by a 32-bit value

More precisely, this is IPv4

A ***port*** plus ***protocol*** identifies an endpoint within a host

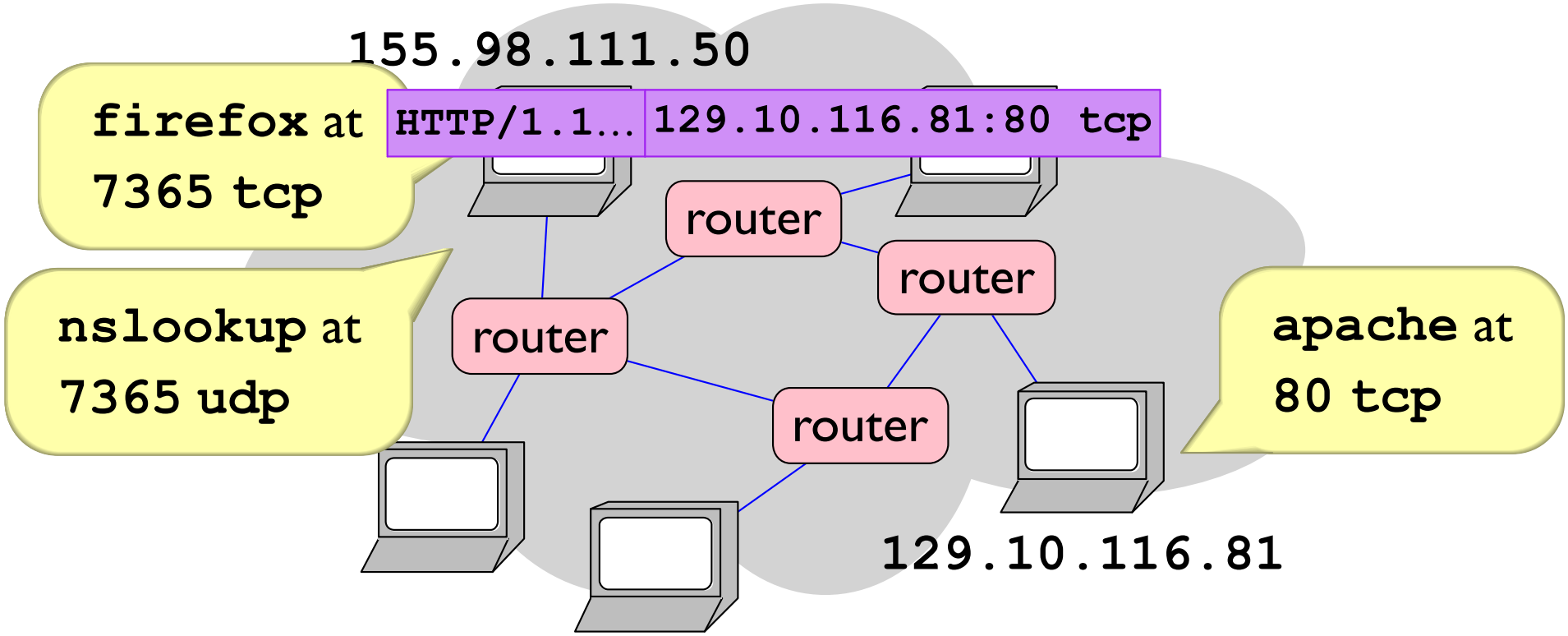


Finding Hosts on a Network

Using ***IP***, a ***host*** is named by a 32-bit value

More precisely, this is IPv4

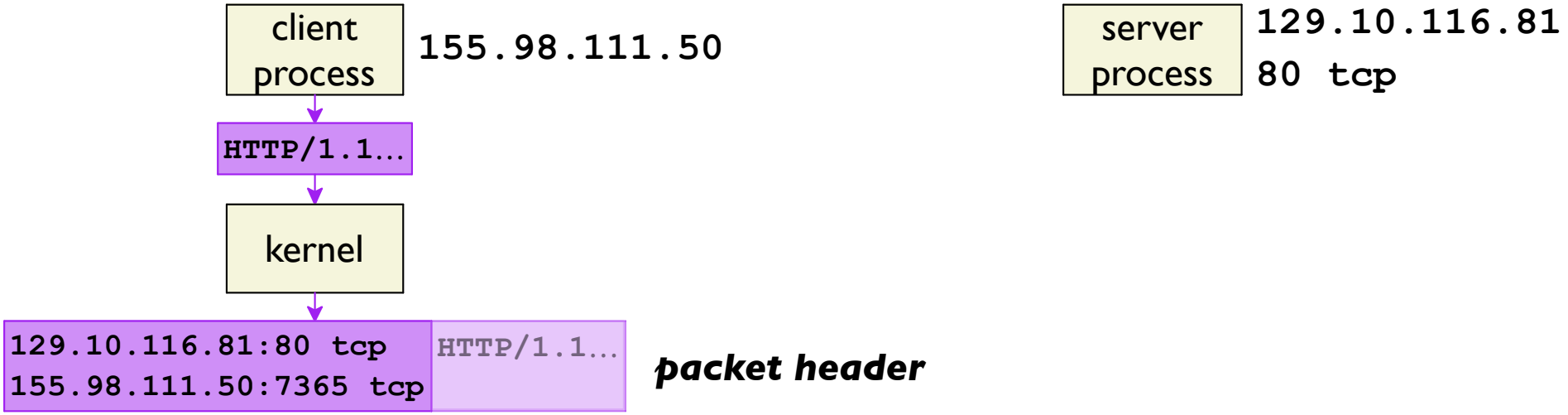
A ***port*** plus ***protocol*** identifies an endpoint within a host



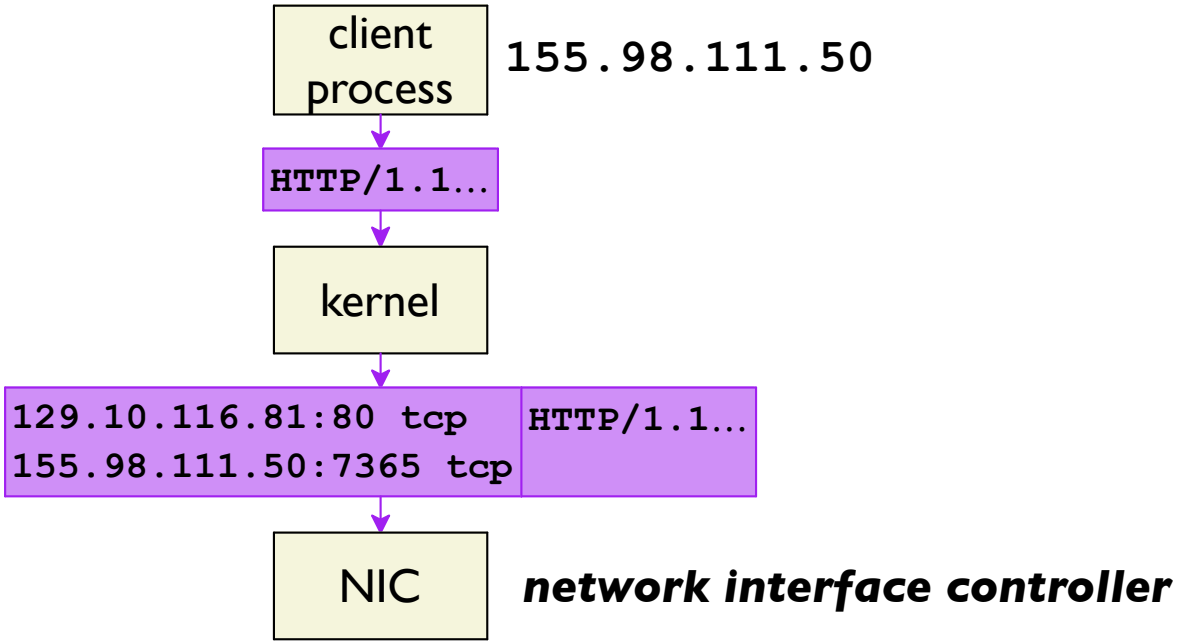
Message Transport

Message Transport

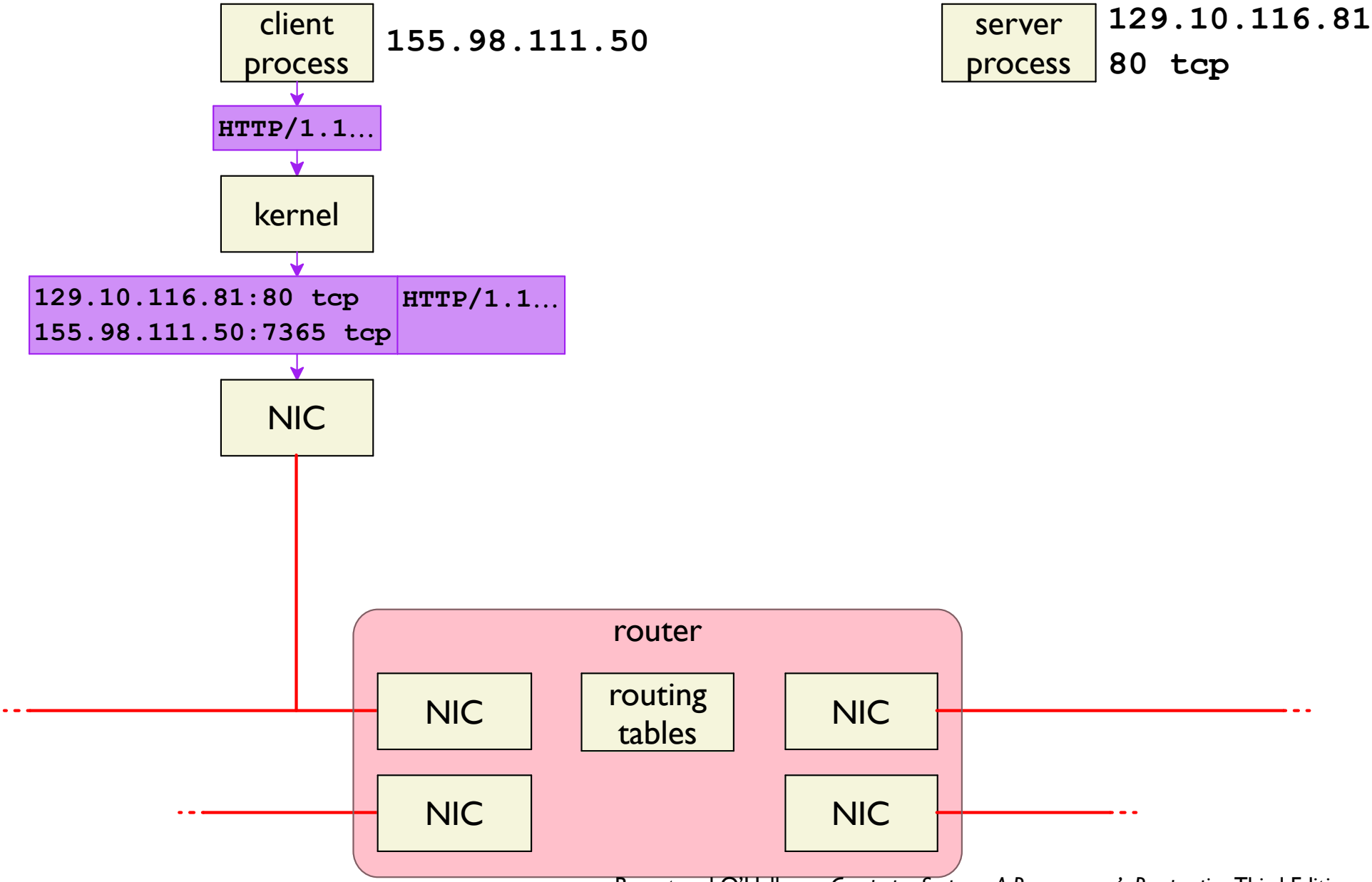
Message Transport



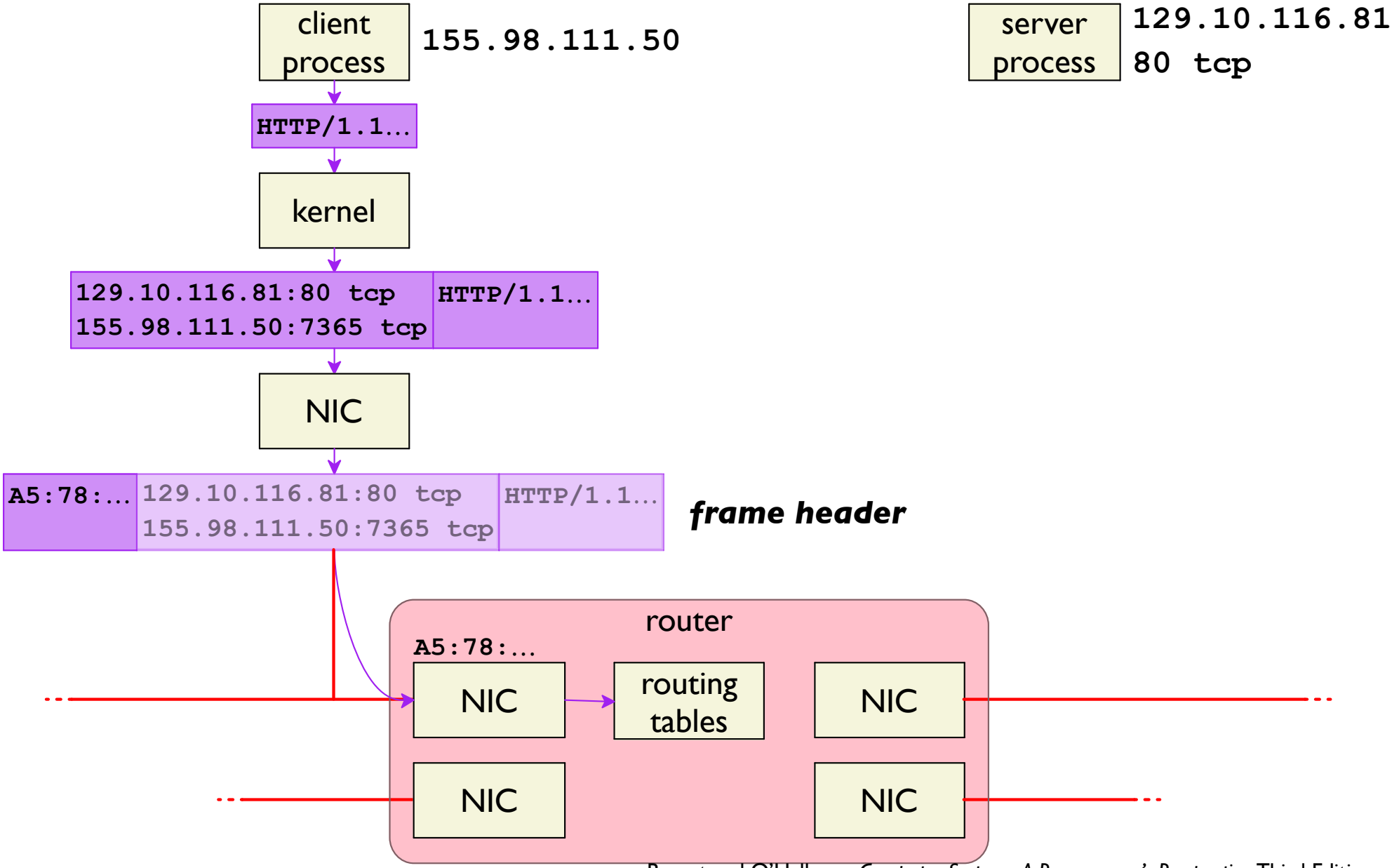
Message Transport



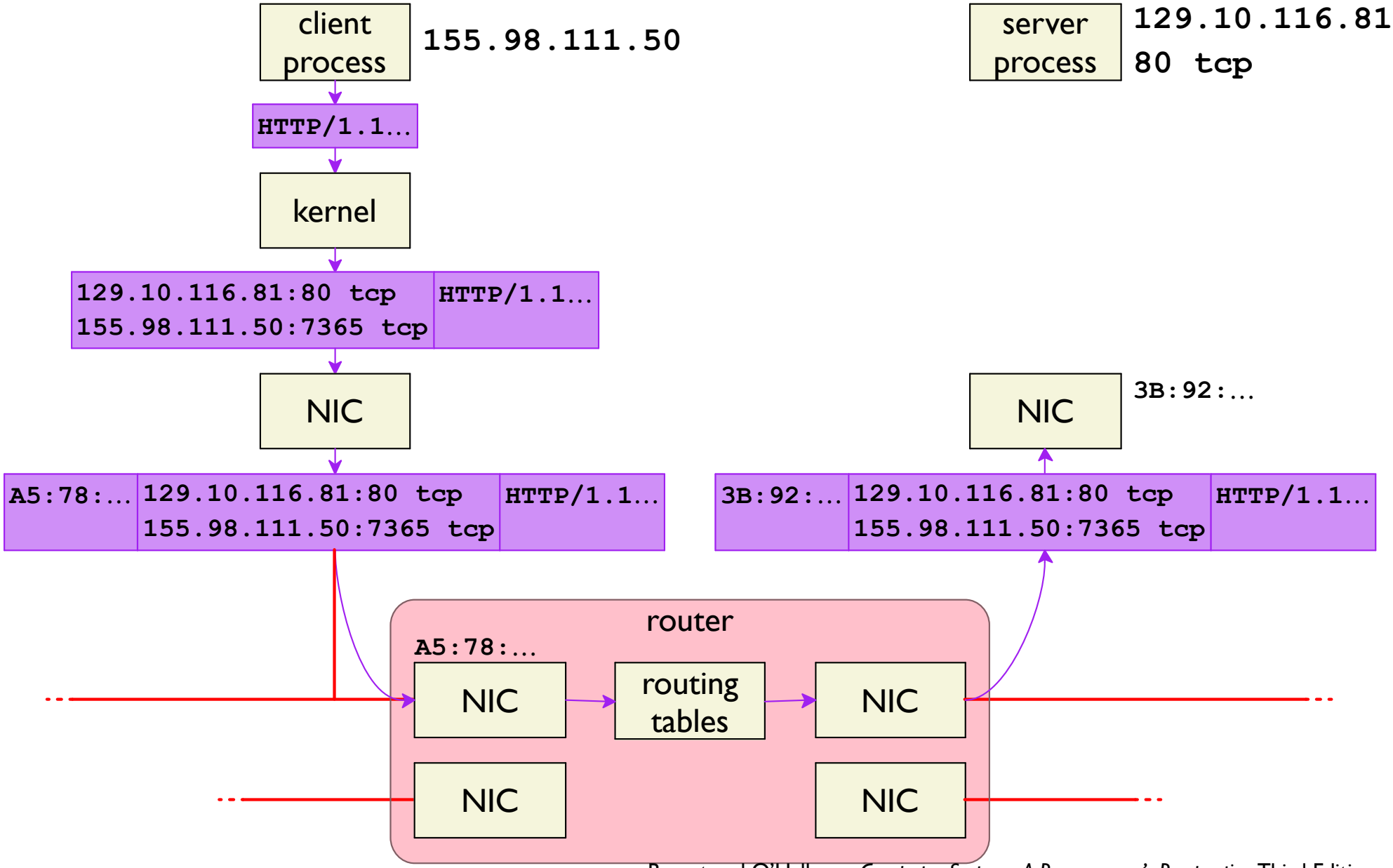
Message Transport



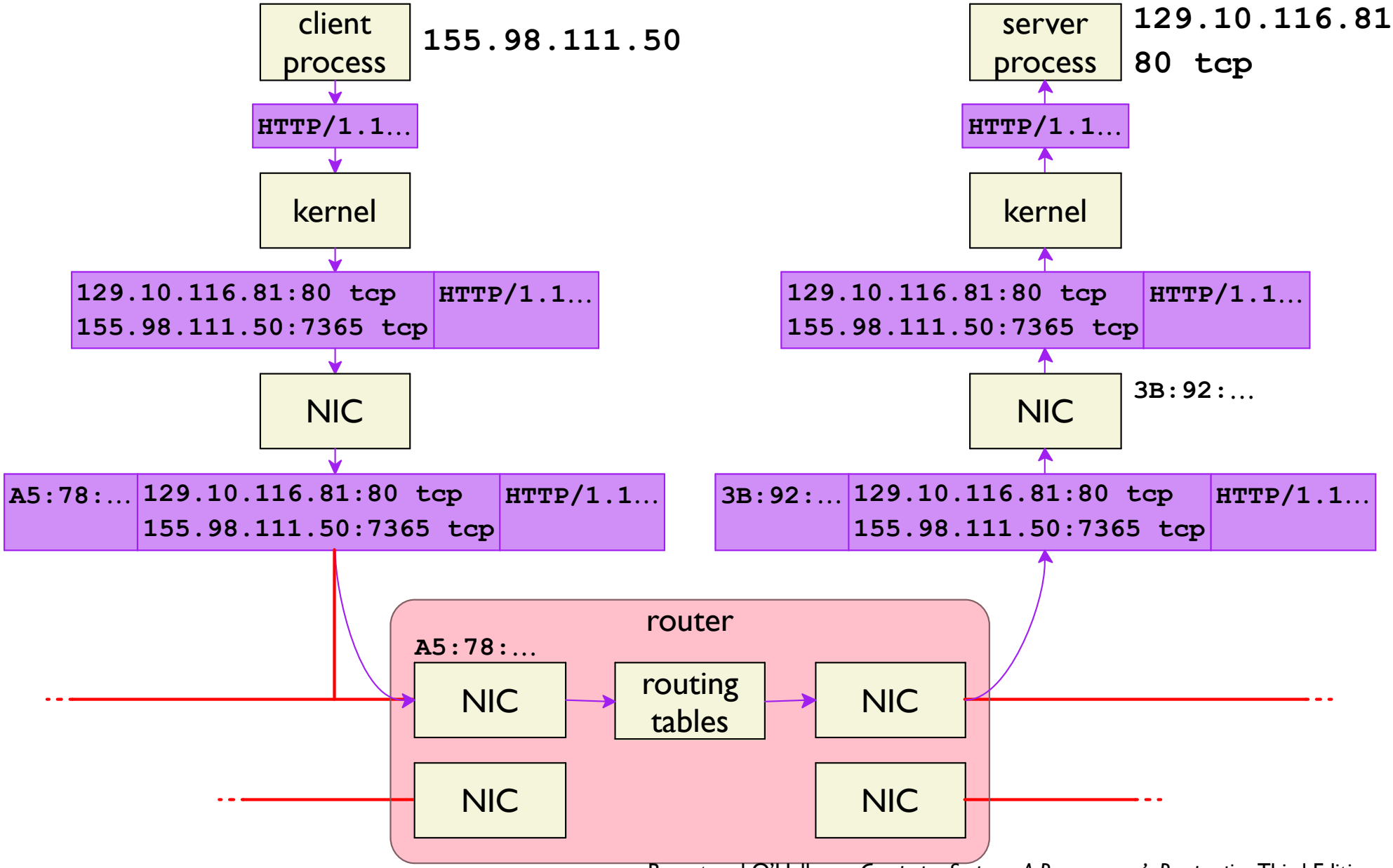
Message Transport



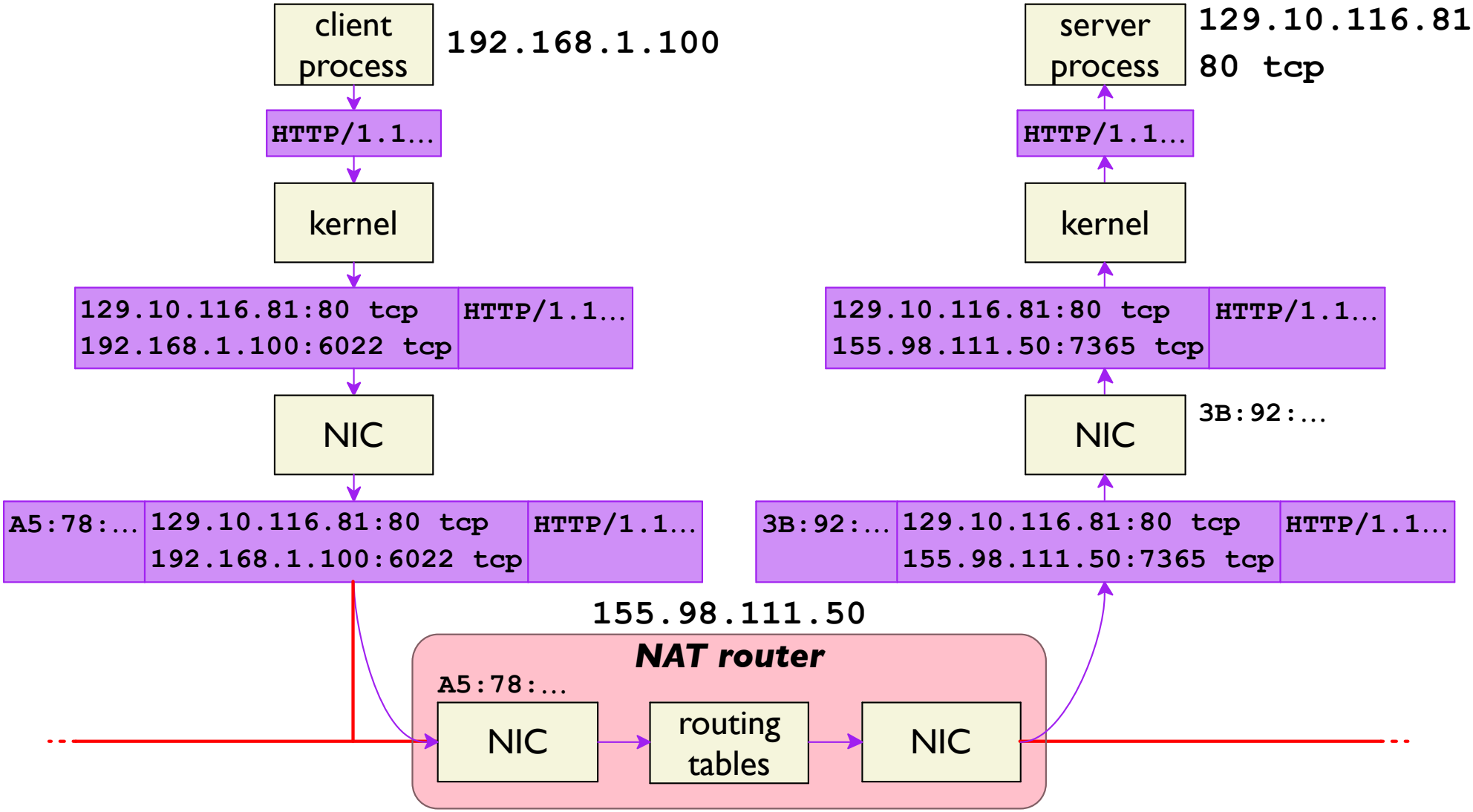
Message Transport



Message Transport



Message Transport



Name Resolution

IP locates hosts by number

e.g., `155.98.111.50`

```
$ ssh 155.98.111.50
```

For many purposes, names are obviously better

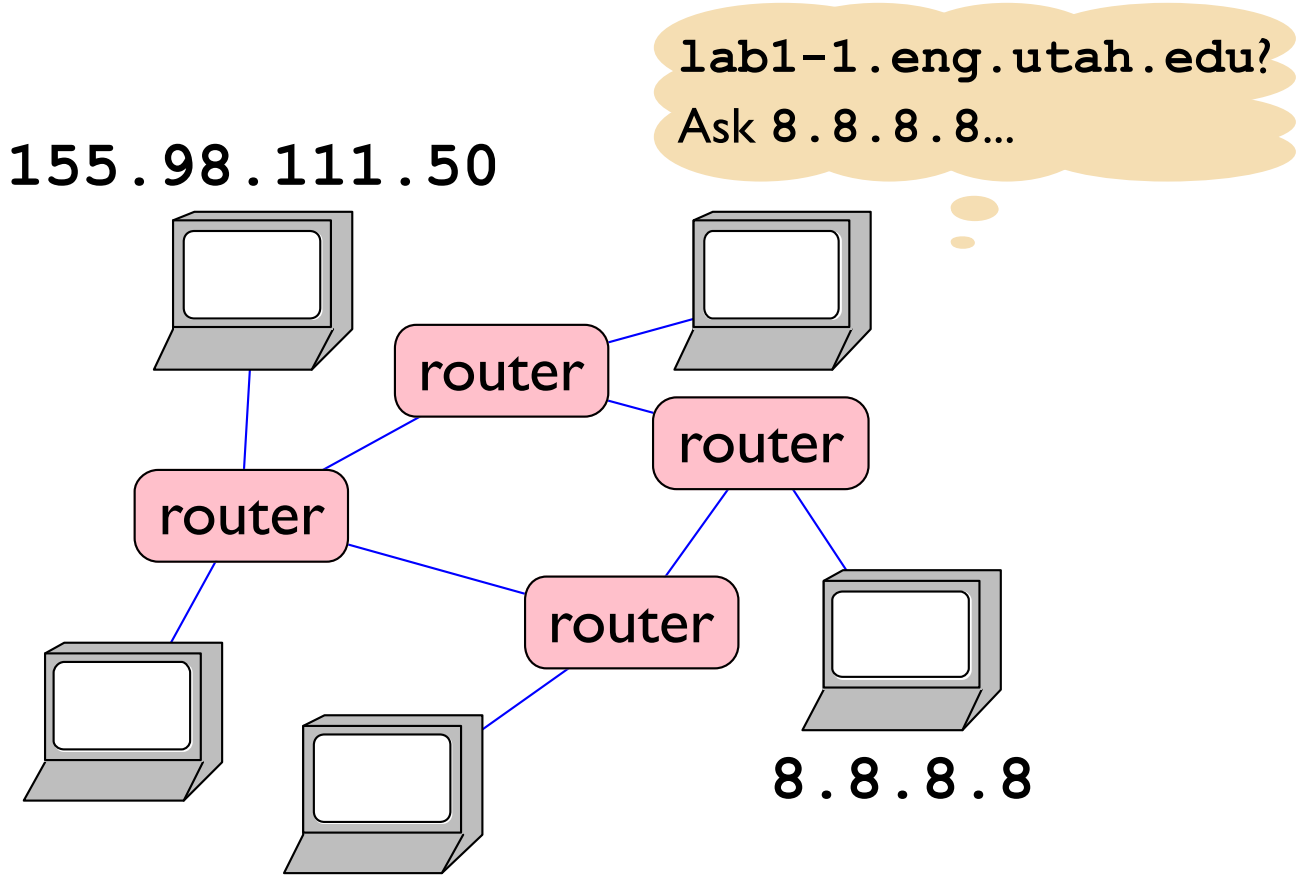
e.g., `lab1-1.eng.utah.edu`

```
$ ssh lab1-1.eng.utah.edu
```

Name Resolution

DNS (Domain Name System) maps names to remote addresses

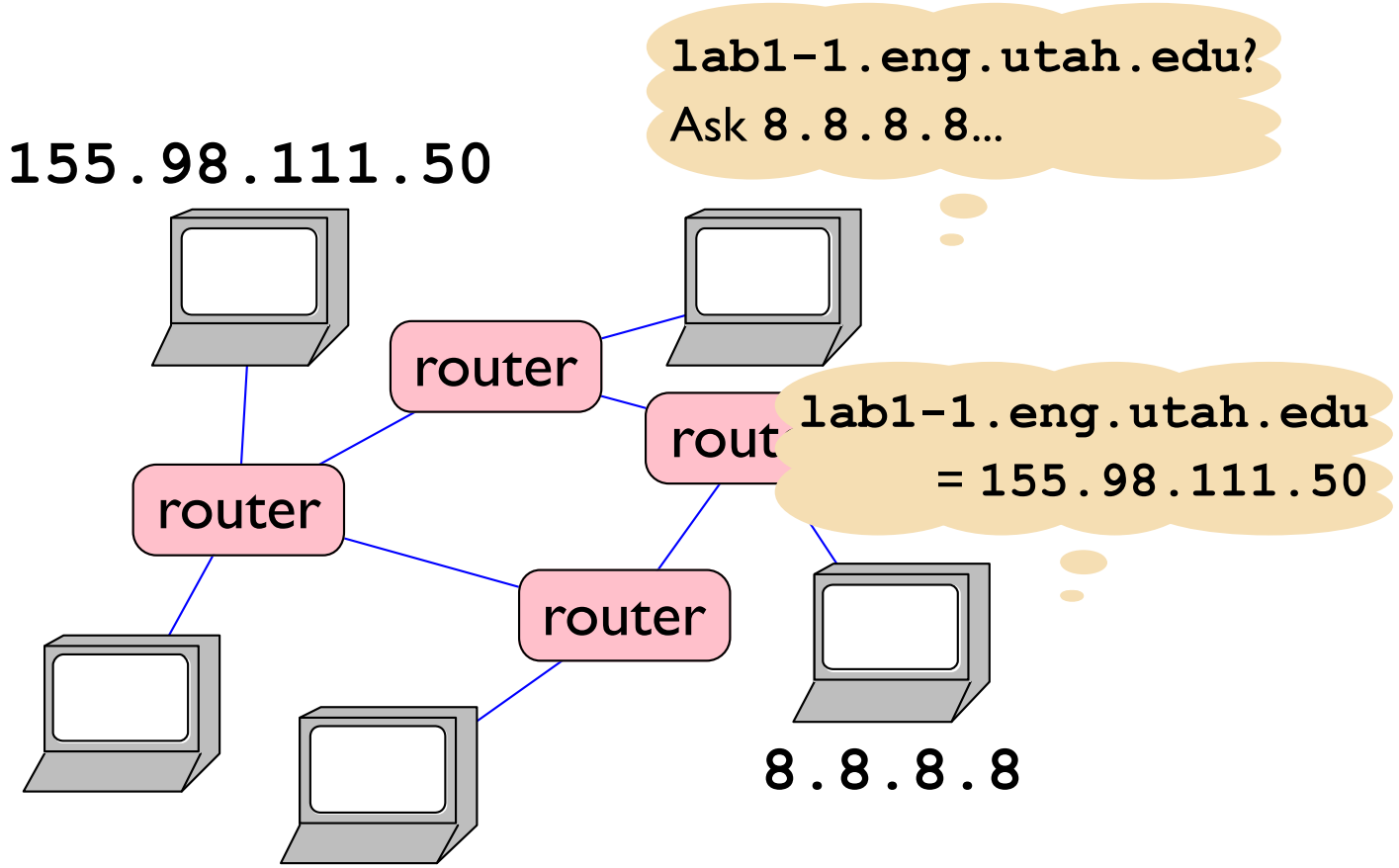
- identify DNS server by address
- DNS server maps names to addresses and vice versa



Name Resolution

DNS (Domain Name System) maps names to remote addresses

- identify DNS server by address
- DNS server maps names to addresses and vice versa



Name Resolution

Multiple names can map to the same address

```
$ ./hostinfo www.eng.utah.edu
155.98.110.30
$ ./hostinfo www.cade.utah.edu
155.98.110.30
```

We'll implement **`hostinfo`**...

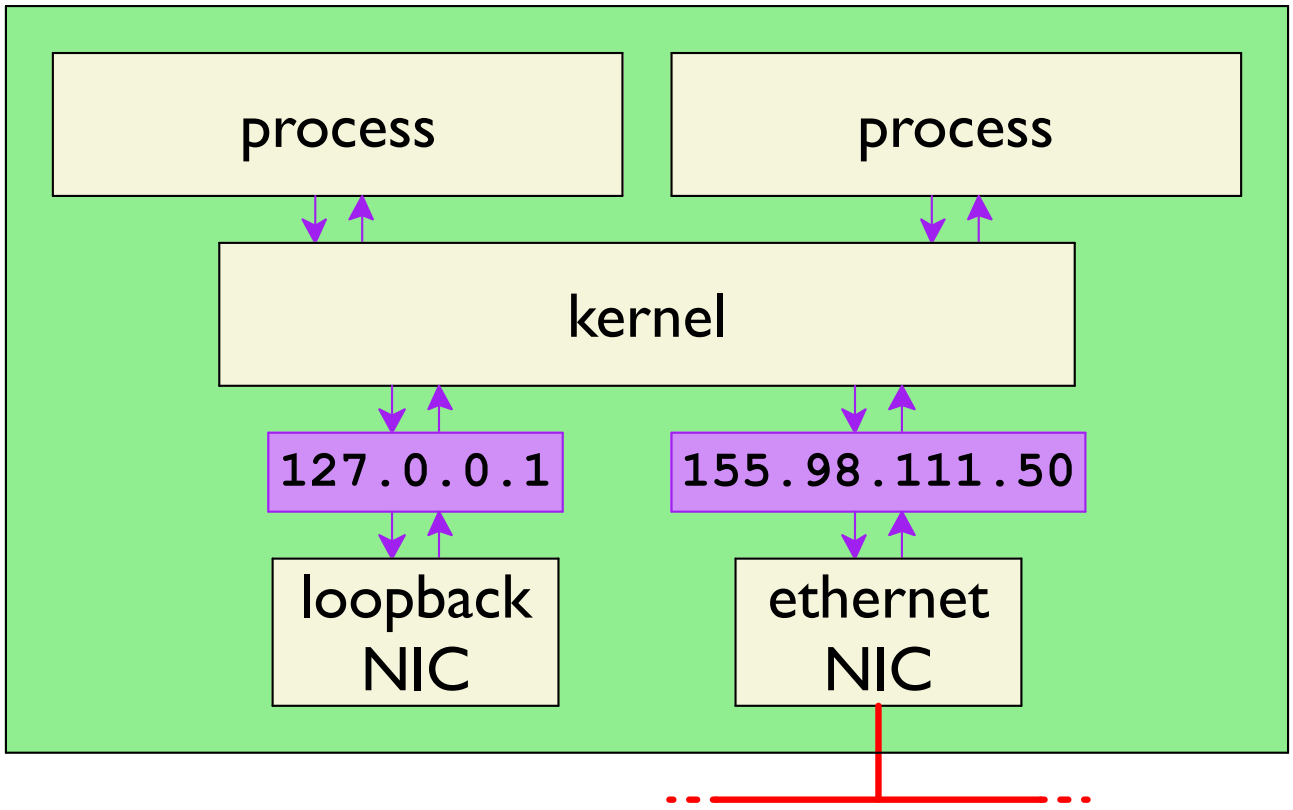
Name Resolution

A single name can map to multiple addresses

```
$ ./hostinfo twitter.com
104.244.42.129
104.244.42.65
104.244.42.193
104.244.42.1
```

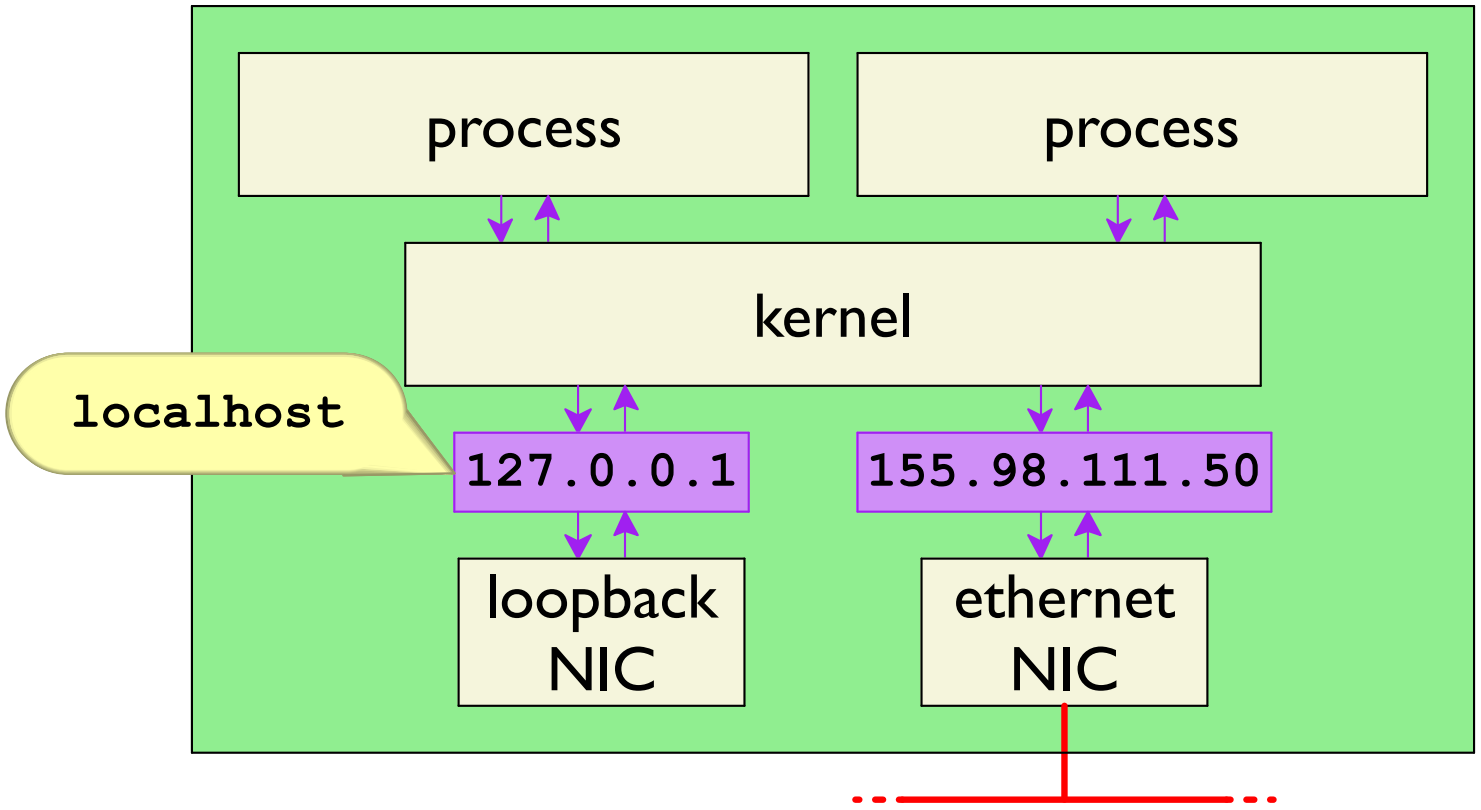
Name Resolution

Naming is not just about finding remote hosts



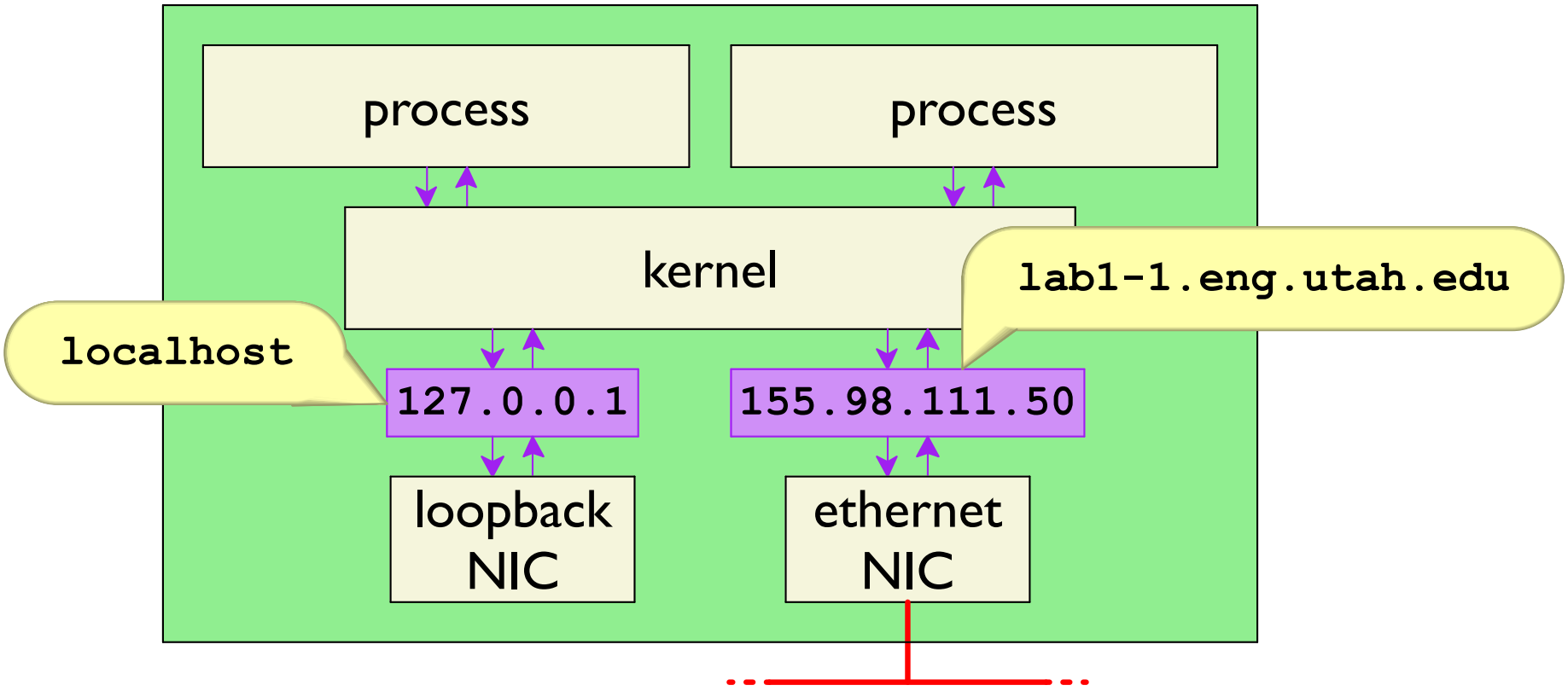
Name Resolution

Naming is not just about finding remote hosts



Name Resolution

Naming is not just about finding remote hosts



Name Resolution

Naming is not just about IPv4

- **`localhost`** via IPv4 = **`127.0.0.1`**
- **`localhost`** via IPv6 = **`::1`**

System calls need to support many protocols

C Library for Name Resolution

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *hostname, const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

Converts a combination of name and port to one or more protocol-specific addresses

- **`hostname`** can be **`NULL`** for "any here"
- **`servname`** is a port number or alias, **`NULL`** for "any"
- **`hints`** can request IPv4, TCP, etc.
- **`res`** is the result: set to a linked list of addresses

C Library for Name Resolution

/usr/include/netdb.h

```
struct addrinfo {
  int ai_flags;
  int ai_family;              /* protocol family */
  int ai_socktype;            /* socket type */
  int ai_protocol;            /* protocol */
  socklen_t ai_addrlen;       /* length of ai_addr */
  struct sockaddr *ai_addr; /* address */
  char *ai_canonname;
  struct addrinfo *ai_next; /* next in list */
};
```

Represents an IPv4 address When **`ai_family`** = **`AF_INET`**

C Library for Name Resolution

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen,
                int flags);
```

The reverse of **`getaddrinfo`**

Set **`flags`** to

`NI_NUMERICHOST | NI_NUMERICSERV`

for numeric address and port

C Library for Name Resolution

hostinfo.c

```
#include "csapp.h"

int main(int argc, char **argv, char **envp) {
  struct addrinfo hints, *addrs, *addr;
  char host[256];

  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET;       /* Request IPv4 */
  hints.ai_socktype = SOCK_STREAM; /* TCP connection */
  Getaddrinfo(argv[1], NULL, &hints, &addrs);

  for (addr = addrs; addr != NULL; addr = addr->ai_next) {
    Getnameinfo(addr->ai_addr, addr->ai_addrlen,
                host, sizeof(host),
                NULL, 0,
                NI_NUMERICHOST);
    printf("%s\n", host);
  }
  ....
}
```

C Library for Name Resolution

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo(struct addrinfo *ai);
```

Frees options allocated by **`getaddrinfo`**

UDP/IP and TCP/IP

IP is the addressing and packet-transfer layer

`127.0.0.1`

- Packets can get lost
- Packets can get reordered

UDP is a thin layer on IP

- Packets can get lost
- Packets can get reordered

TCP is a substantial layer on IP

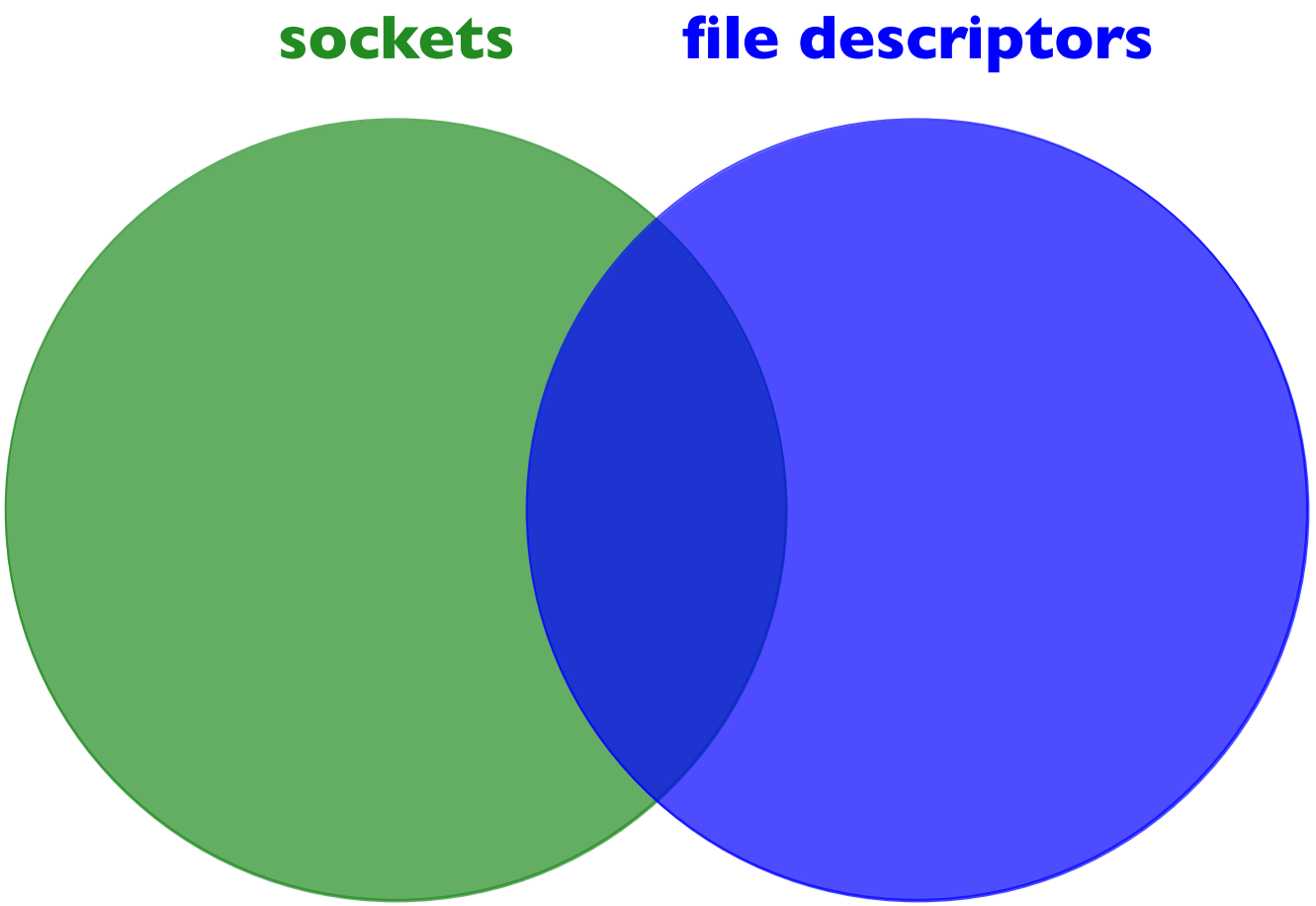
- Mostly hides packet nature behind a stream interface
- Retries as needed to get data sent
- Tags packets with sequence numbers for ordering

Sockets

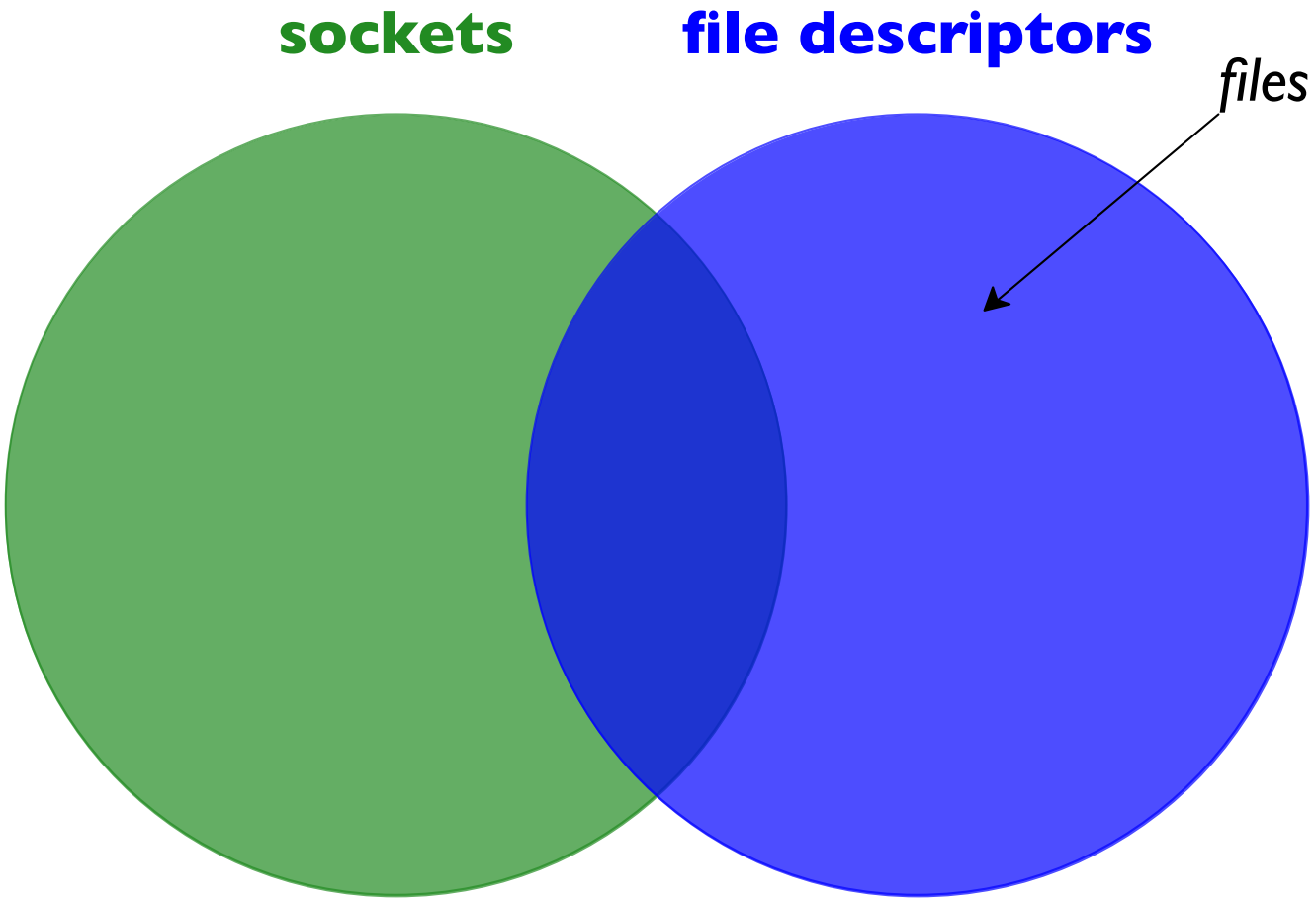
A generic communication is a ***socket***

A socket is represented as an **`int`**

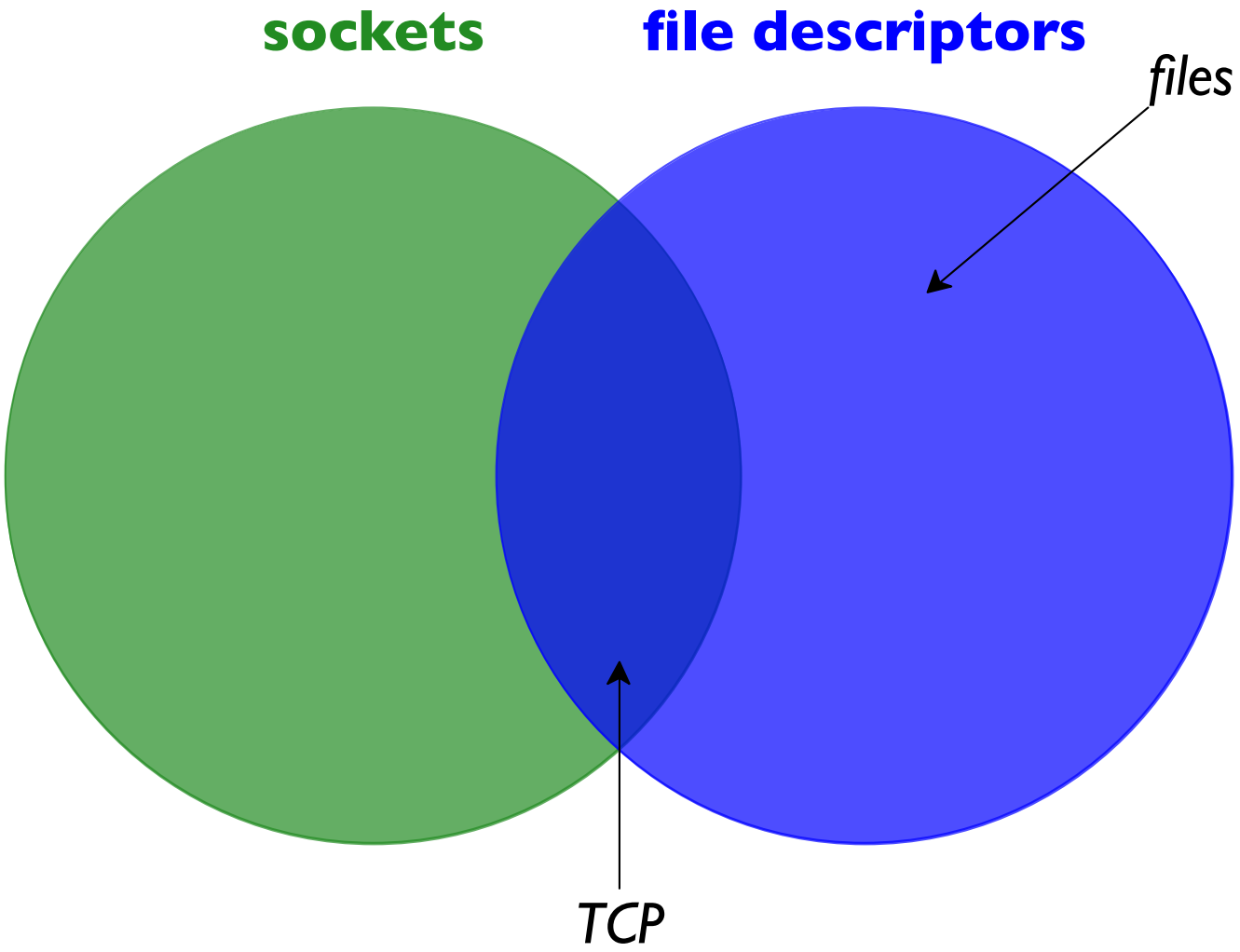
Sockets



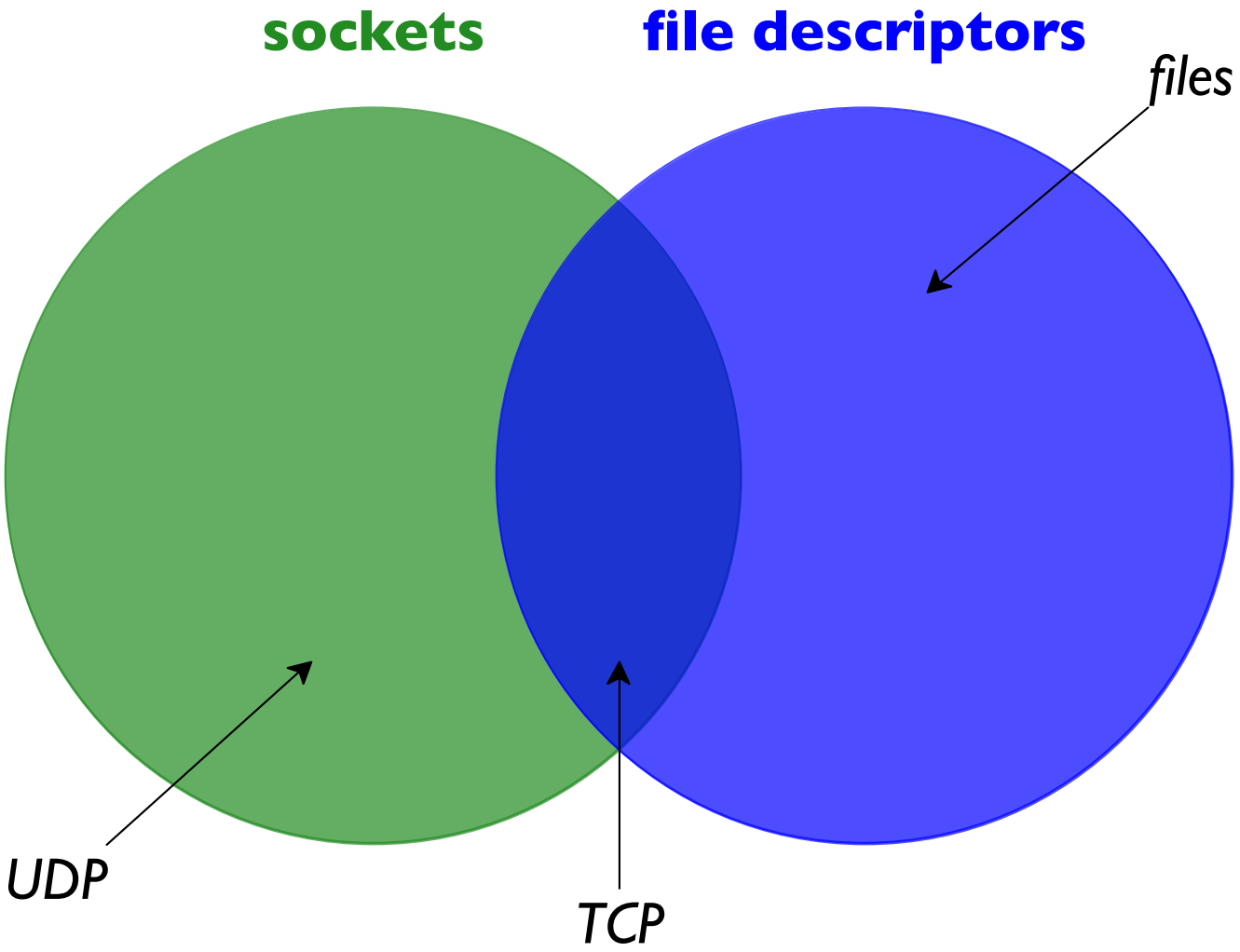
Sockets



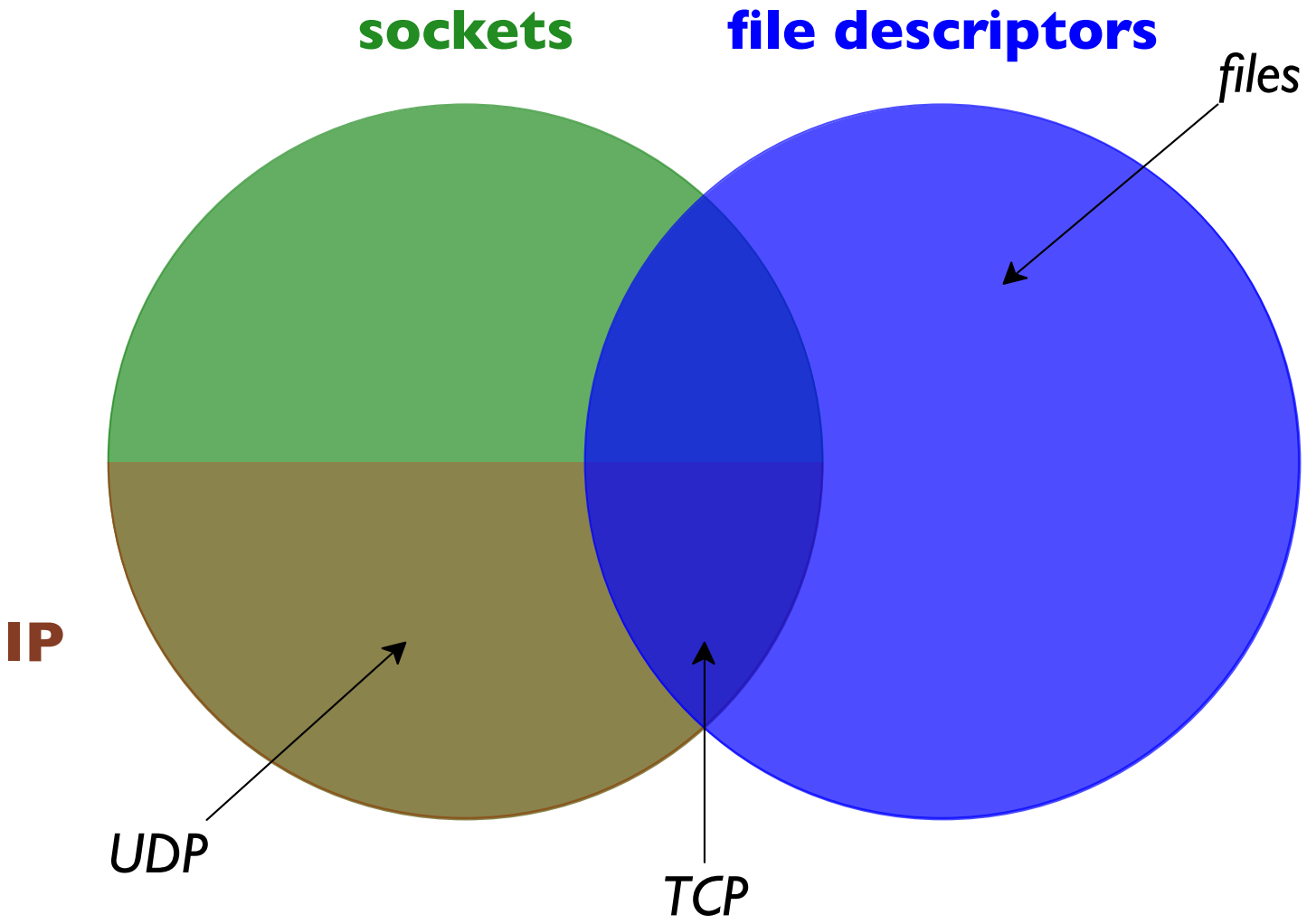
Sockets



Sockets



Sockets



Sockets

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Creates a new socket

- **`domain`** is a protocol family; **`PF_INET`** means IPv4
- **`type`** is
  - **`SOCK_DGRAM`** for UDP
  - **`SOCK_STREAM`** for TCP
- **`protocol`** is a kind of subtype

For portable code, get arguments from the result of **`getaddrinfo`**

Binding Sockets

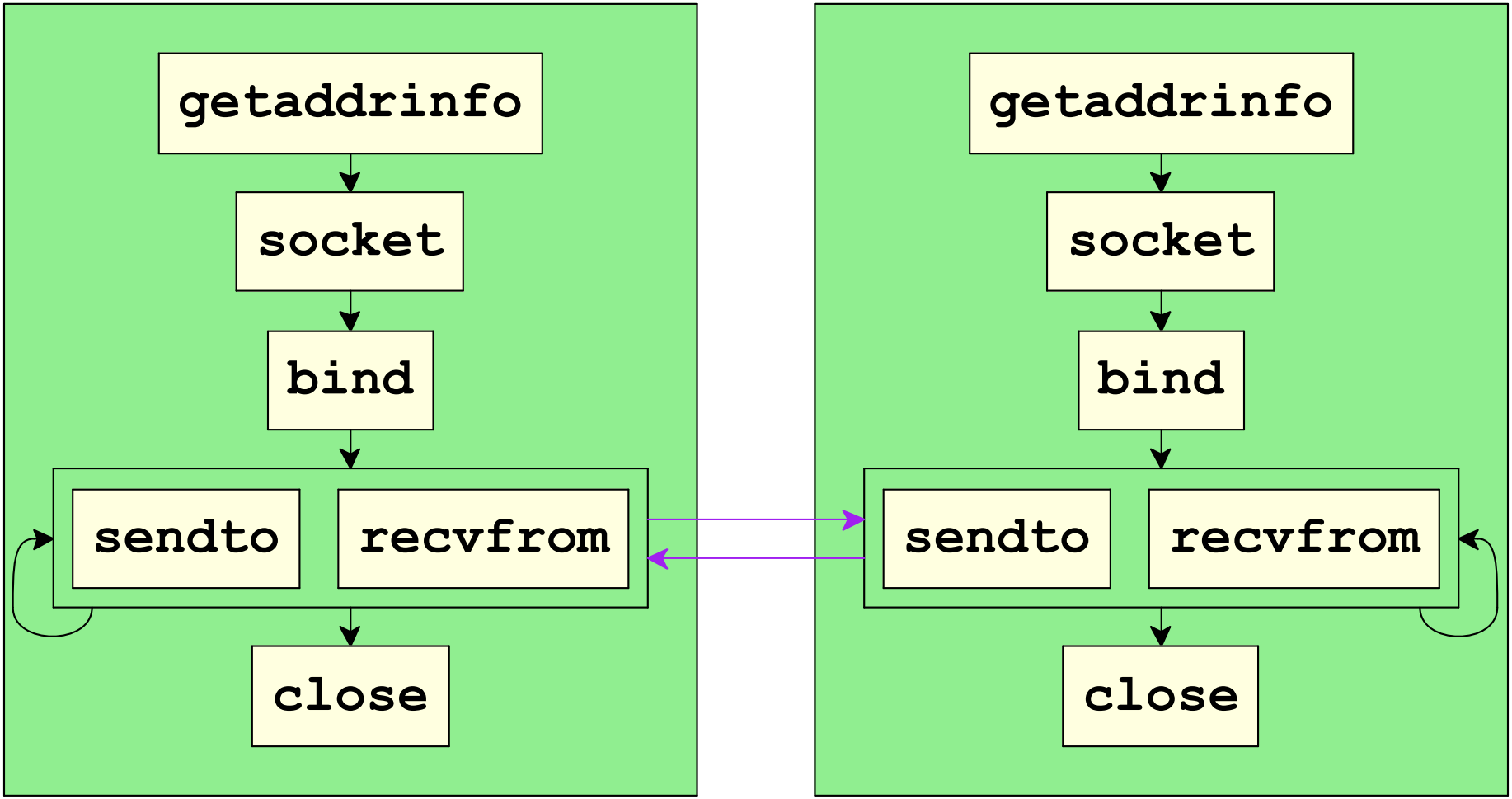
```
#include <sys/socket.h>

int bind(int socket,
         struct sockaddr *addr, socklen_t addr_len);
```

Attaches a socket to a specific address

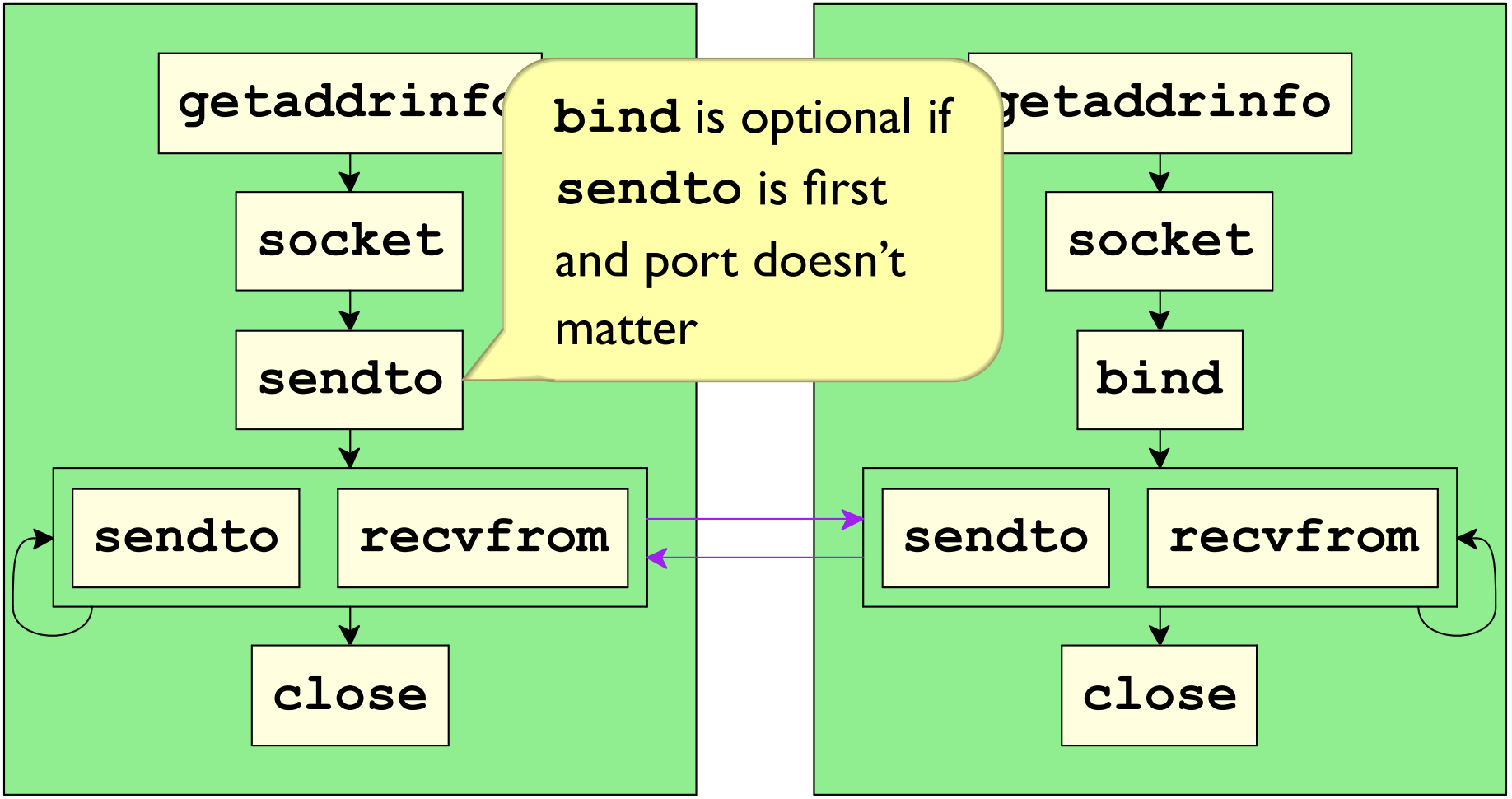
If other processes know the address, they can send a message to the socket

The **`addr`** and **`addr_len`** arguments come from **`getaddrinfo`**

Using UDP

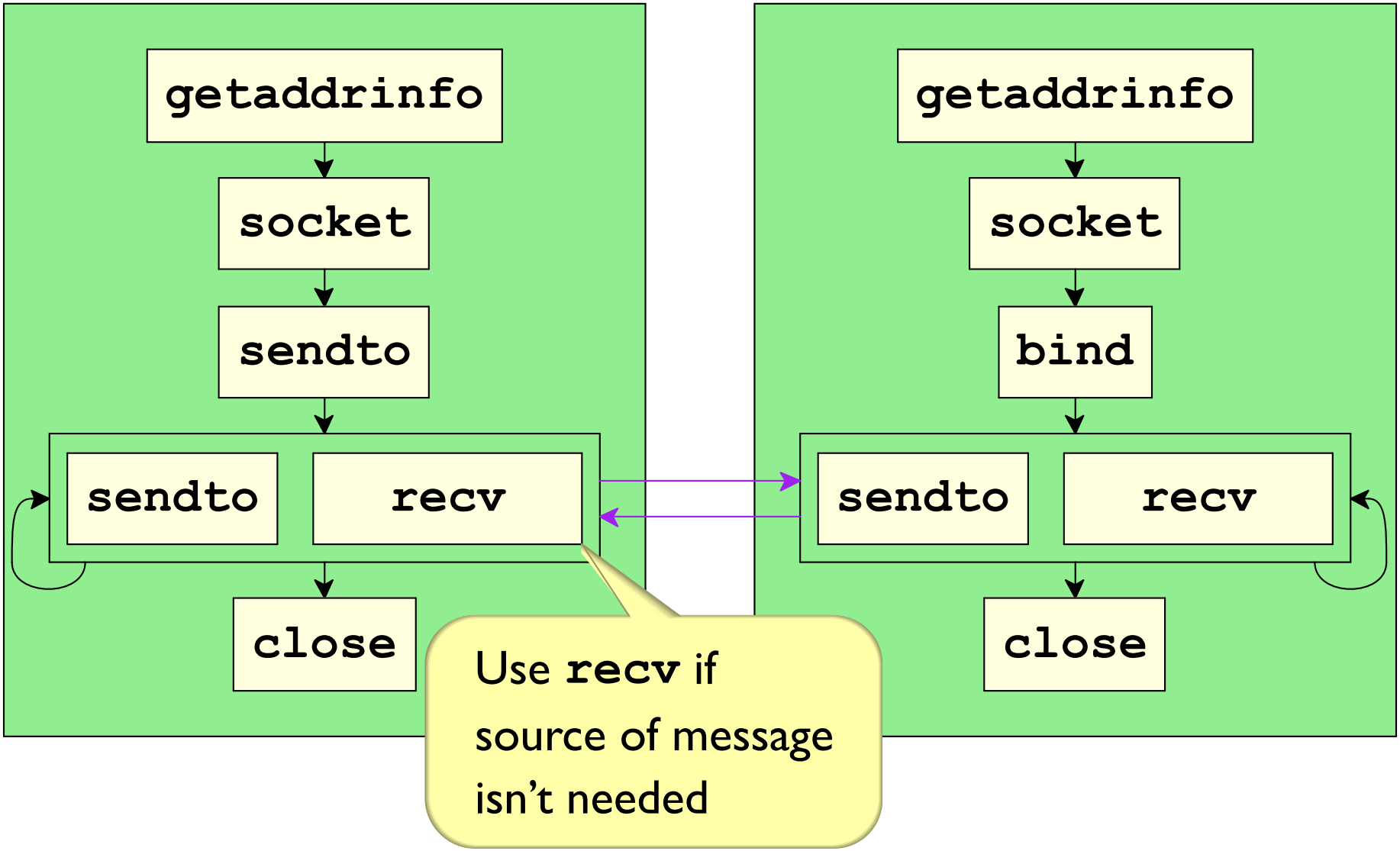

getaddrinfo
socket
bind
sendto
recvfrom
close
getaddrinfo
socket
bind
sendto
recvfrom
close

Using UDP



Automatically selected ports are in the ***ephemeral port*** range

Using UDP


getaddrinfo
socket
sendto
sendto
recv
close
getaddrinfo
socket
bind
sendto
recv
close
Use recv if source of message isn't needed

UDP Server

udp_recv.c

```
#include "csapp.h"

int main(int argc, char **argv) { /* argv[0] == portno */
  struct addrinfo hints, *addrs;
  int s;

  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET;        /* Request IPv4 */
  hints.ai_socktype = SOCK_DGRAM;   /* Accept UDP connections */
  hints.ai_flags = AI_PASSIVE;      /* ... on any IP address */
  Getaddrinfo(NULL, argv[0], &hints, &addrs);

  s = Socket(addrs->ai_family, addrs->ai_socktype, addrs->ai_protocol);
  Bind(s, addrs->ai_addr, addrs->ai_addrlen);
  Freeaddrinfo(addrs);

  while (1) {
    char buffer[MAXBUF];
    size_t amt;
    amt = Recv(s, buffer, MAXBUF, 0);
    Write(1, buffer, amt);
    Write(1, "\n", 1);
  }

  return 0;
}
```

UDP Client

udp_send.c

```
#include "csapp.h"

int main(int argc, char **argv, char **envp) {
  char *hostname = argv[1], *portno = argv[2];
  struct addrinfo hints, *addrs;
  char host[256], serv[32];
  int s;
  size_t amt;

  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET;       /* Request IPv4 */
  hints.ai_socktype = SOCK_DGRAM; /* UDP connection */
  Getaddrinfo(hostname, portno, &hints, &addrs);

  Getnameinfo(addrs->ai_addr, addrs->ai_addrlen,
              host, sizeof(host), serv, sizeof(serv),
              NI_NUMERICHOST | NI_NUMERICSERV);
  printf("sending to %s:%s\n", host, serv);

  s = Socket(addrs->ai_family, addrs->ai_socktype, addrs->ai_protocol);
  amt = Sendto(s, argv[3], strlen(argv[3]), 0,
               addrs->ai_addr, addrs->ai_addrlen);
  Freeaddrinfo(addrs);

  return (amt != strlen(argv[3]));
}
```

UDP Server

udp_recv.c

```
#include "csapp.h"

int main(int argc, char **argv) { /* argv[0] == portno */
  struct addrinfo hints, *addrs;
  int s;

  memset(&hints, 0
  hints.ai_family
  hints.ai_socktyp
  hints.ai_flags
  Getaddrinfo(NULL, argv[0], &hints, &addrs);

  s = Socket(addrs->ai_family, addrs->ai_socktype, addrs->ai_protocol);
  Bind(s, addrs->ai_addr, addrs->ai_addrlen);
  Freeaddrinfo(addrs);

  while (1) {
    char buffer[MAXBUF];
    size_t amt;
    amt = Recv(s, buffer, MAXBUF, 0);
    Write(1, buffer, amt);
    Write(1, "\n", 1);
  }

  return 0;
}
```

Using "**localhost**" constrains to a loopback connection

Revised UDP Server

udp_recvfrom.c

```
....
int counter = 0;
....

while (1) {
  char buffer[MAXBUF];
  size_t amt;
  struct sockaddr_in from_addr;
  unsigned int from_len = sizeof(from_addr);

  amt = Recvfrom(s, buffer, MAXBUF, 0,
                 (struct sockaddr *)&from_addr, &from_len);
  Write(1, buffer, amt);
  Write(1, "\n", 1);

  Getnameinfo((struct sockaddr *)&from_addr, from_len,
              host, sizeof(host),
              serv, sizeof(serv),
              NI_NUMERICHOST | NI_NUMERICSERV);
  printf(" from %s:%s [%d]\n", host, serv, ++counter);
}

....
```

Revised UDP Client

udp_from_send.c

```
char *myportno = argv[1];
char *hostname = argv[2];
char *portno = argv[3];
....
Getaddrinfo(NULL, myportno, &hints, &my_addrs);

....

if (argc == 6)
  copies = atoi(argv[5]);
else
  copies = 1;

while (copies--)
  amt = Sendto(s, argv[4], strlen(argv[4]), 0,
               addrs->ai_addr, addrs->ai_addrlen);

....
```

TCP and Connections

TCP provides reliability by tracking and retrying lost packets

⇒ needs an explicit ***connection*** between processes

⇒ needs explicit notions of client and server

Server side has two sockets:

A ***listener*** to receive clients

A per-client socket to communicate with the client

Each client has a single socket

Listening for Connections

```
#include <sys/socket.h>

int listen(int socket, int backlog);
```

In the background, allow TCP connections via **`socket`**

`socket` must be bound to an address already

Clients connect to that address

`backlog` indicates how many not-yet-accepted connections to allow in waiting

Accepting Connections

```
#include <sys/socket.h>

int accept(int socket,
           struct sockaddr *addr, socklen_t *addr_len);
```

Accepts a connection from **`socket`** and returns it as a new socket

`socket` must be previously passed to **`listen`**

`addr` and **`addr_len`** are filled with the client's address

... but a server doesn't usually care

Making Connections

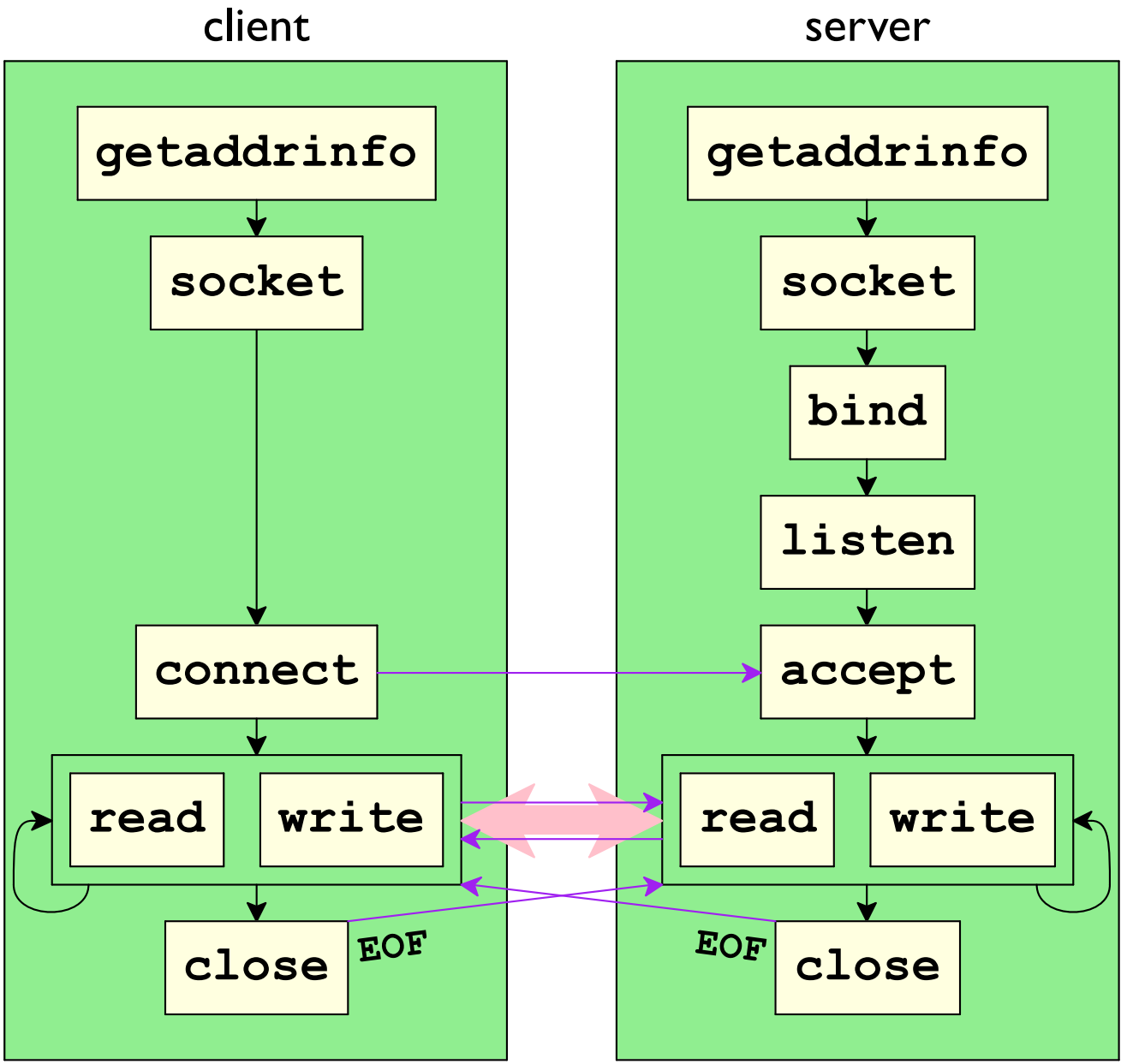
```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int socket,
            struct sockaddr *addr, socklen_t addr_len);
```

Binds **`socket`** to a TCP connection as a client

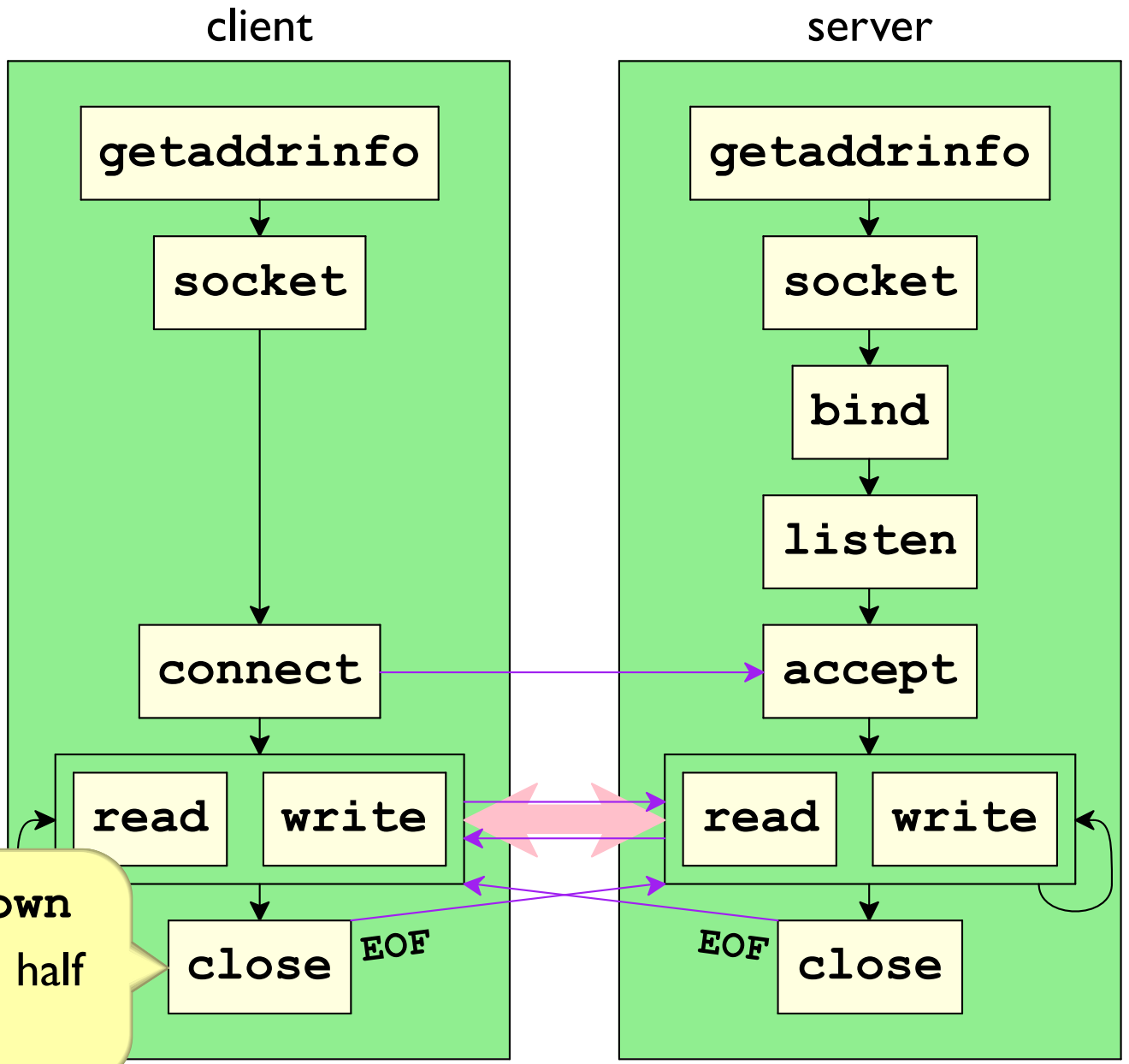
`addr` and **`addr_len`** are the server's address

The server sees the connection when it calls **`accept`**

Using TCP

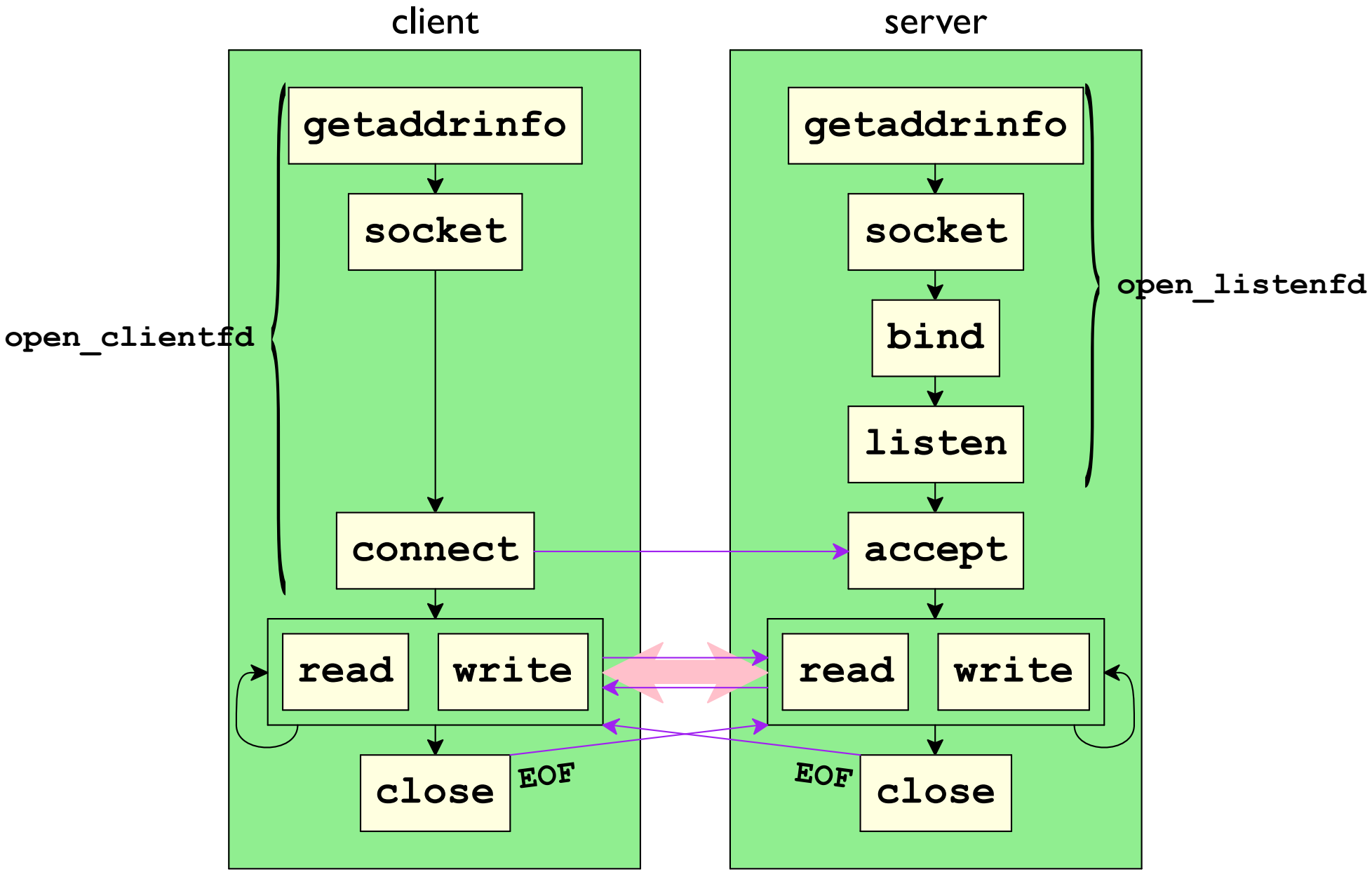

client
server
getaddrinfo
socket
connect
read
write
close
EOF
getaddrinfo
socket
bind
listen
accept
read
write
EOF
close

Using TCP

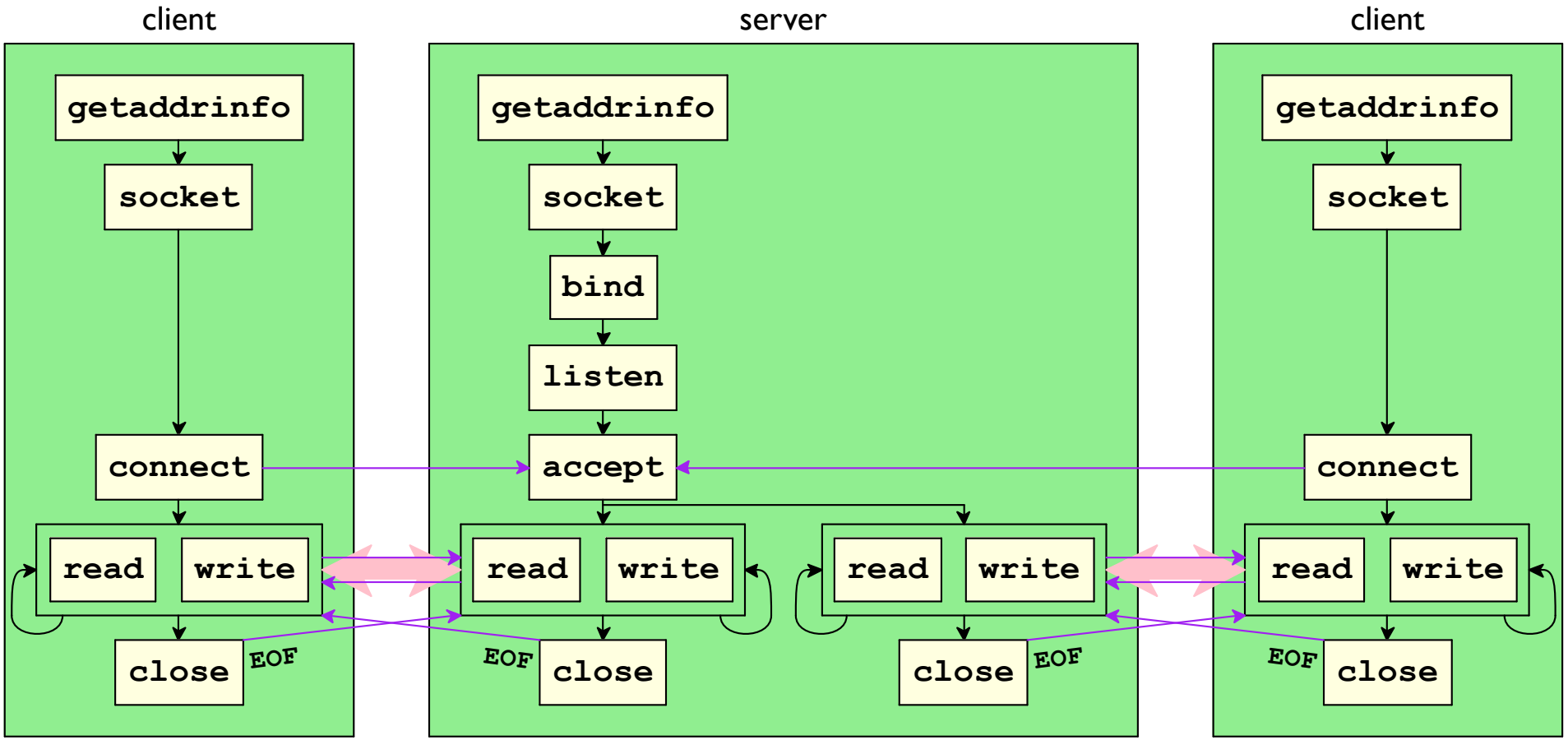


Use **shutdown** to leave read half open

Using TCP


client
server
getaddrinfo
socket
open_clientfd
connect
read
write
close
EOF
getaddrinfo
socket
bind
listen
open_listenfd
accept
read
write
EOF
close

Using TCP


client
server
client
getaddrinfo
socket
bind
listen
connect
accept
read
write
close
EOF

TCP Server

tcp_server.c

```
....
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_INET;         /* Request IPv4 */
hints.ai_socktype = SOCK_STREAM;   /* Accept TCP connections */
hints.ai_flags = AI_PASSIVE;       /* ... on any IP address */
Getaddrinfo(NULL, portno, &hints, &addrs);

ls = Socket(addrs->ai_family, addrs->ai_socktype, addrs->ai_protocol);
Bind(ls, addrs->ai_addr, addrs->ai_addrlen);
Freeaddrinfo(addrs);
Listen(ls, 5);

while (1) {
  ....
  s = Accept(ls, (struct sockaddr *)&addr, &len);
  amt = Read(s, buffer, MAXBUF);
  write(1, buffer, amt);
  write(1, "\n", 1);
  write(s, &amt, sizeof(amt));
  Close(s);
}
....
```

TCP Client

tcp_client.c

```
....
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_INET;         /* Request IPv4 */
hints.ai_socktype = SOCK_STREAM;  /* TCP connection */
Getaddrinfo(hostname, portno, &hints, &addrs);

s = Socket(addrs->ai_family, addrs->ai_socktype, addrs->ai_protocol);
Connect(s, addrs->ai_addr, addrs->ai_addrlen);
Freeaddrinfo(addrs);

copies = ((argc == 5) ? atoi(argv[4]) : 1);
len = strlen(argv[3]);
while (copies--) {
  amt = Write(s, argv[3], len);
  if (amt != len) app_error("incomplete write");
}

got = 0;
amt = Read(s, &got, sizeof(got));
printf("server got %ld\n", got);
Close(s);
....
```

Robust I/O

Provided by **`csapp.c`**:

```
ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

Like **`read`** and **`write`**, but loops as needed for short counts and signal interruptions

Revised TCP Server

Robust I/O, wait for an EOF from clients:

tcp_server2.c

```
....
s = Accept(ls, (struct sockaddr *)&addr, &len);

while (1) {
  char buffer[MAXBUF];
  size_t amt = Rio_readn(s, buffer, MAXBUF);
  if (amt == 0) {
    printf("client is done\n");
    Rio_writen(s, &total_amt, sizeof(total_amt));
    break;
  } else {
    Rio_writen(1, buffer, amt);
    Rio_writen(1, "\n", 1);
    total_amt += amt;
  }
}

Close(s);
....
```

Revised TCP Client

tcp_client2.c

```
....

while (copies--)
  Rio_writen(s, argv[3], len);

Shutdown(s, SHUT_WR);

got = 0;
amt = Rio_readn(s, &got, sizeof(got));
if (amt != sizeof(got))
  app_error("response truncated");
printf("server got %ld\n", got);

Close(s);

....
```

Re-Revised TCP Server

Client can declare amount it will send

tcp_server3.c

```
....
s = Accept(ls, (struct sockaddr *)&addr, &len);

amt = Rio_readn(s, &total_amt, sizeof(total_amt));
if (amt != sizeof(total_amt))
  app_error("amount truncated");

buffer = malloc(total_amt);
amt = Rio_readn(s, buffer, total_amt);

Rio_writen(1, buffer, amt);
Rio_writen(1, "\n", 1);
free(buffer);

write(s, &amt, sizeof(amt));
Close(s);
....
```

Re-Revised TCP Client

tcp_client3.c

```
....

len = strlen(argv[3]);

amt = copies * len;
Rio_writen(s, &amt, sizeof(amt));

while (copies--)
  Rio_writen(s, argv[3], len);

got = 0;
amt = Rio_readn(s, &got, sizeof(got));
if (amt != sizeof(got))
  app_error("response truncated");

printf("server got %ld\n", got);

Close(s);

....
```

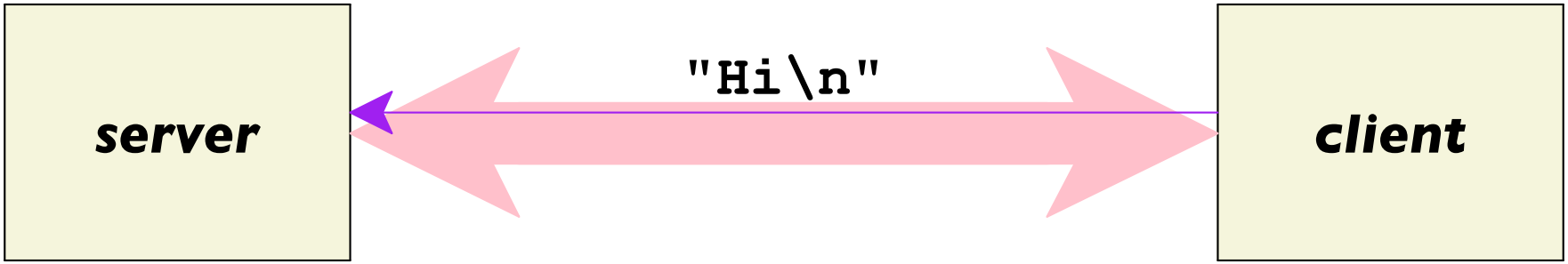
Echo Server

Another example: a server that echoes back every line


```
lnr = Open_listenfd(....);
fd = Accept(lnr, ....);          fd = Open_clientfd(....);
```

Echo Server

Another example: a server that echoes back every line

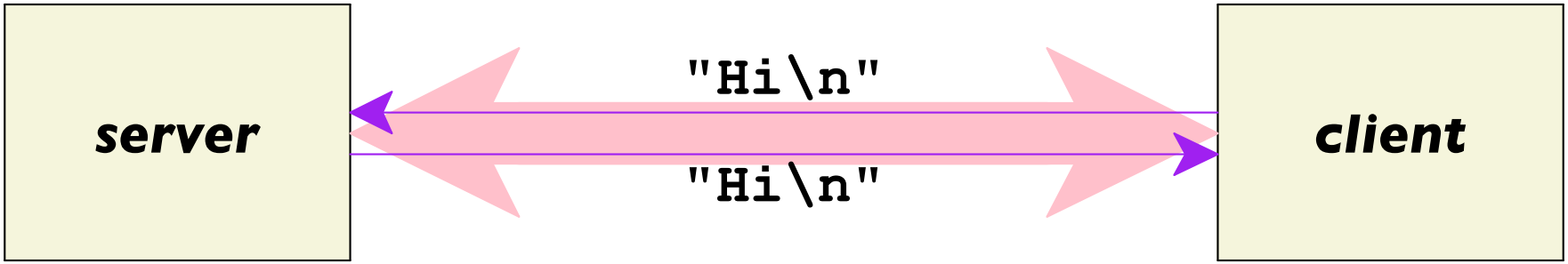



```
lnr = Open_listenfd(....);
fd = Accept(lnr, ....);
```

```
fd = Open_clientfd(....);

Rio_writen(fd, "Hi\n", 3);
```

Echo Server

Another example: a server that echoes back every line




```
lnr = Open_listenfd(....);
fd = Accept(lnr, ....);
```

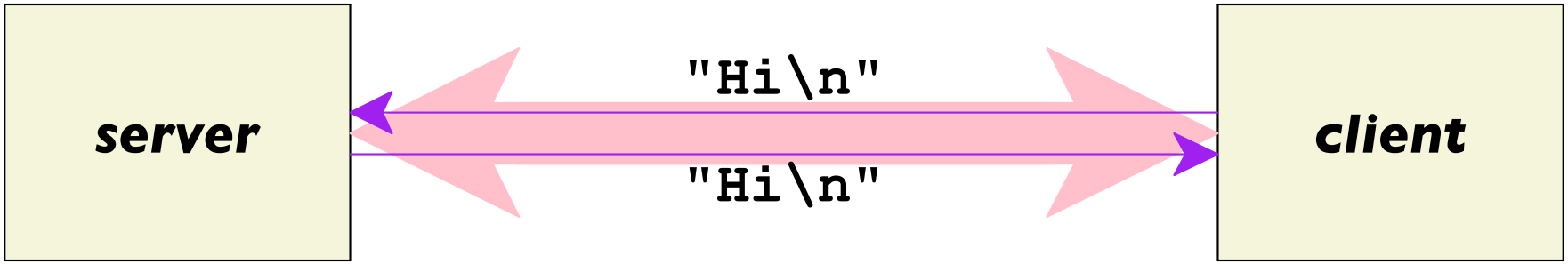
```
fd = Open_clientfd(....);
```

```
Rio_writen(fd, "Hi\n", 3);
```

```
Rio_writen(fd, ....);
```

Echo Server

Another example: a server that echoes back every line

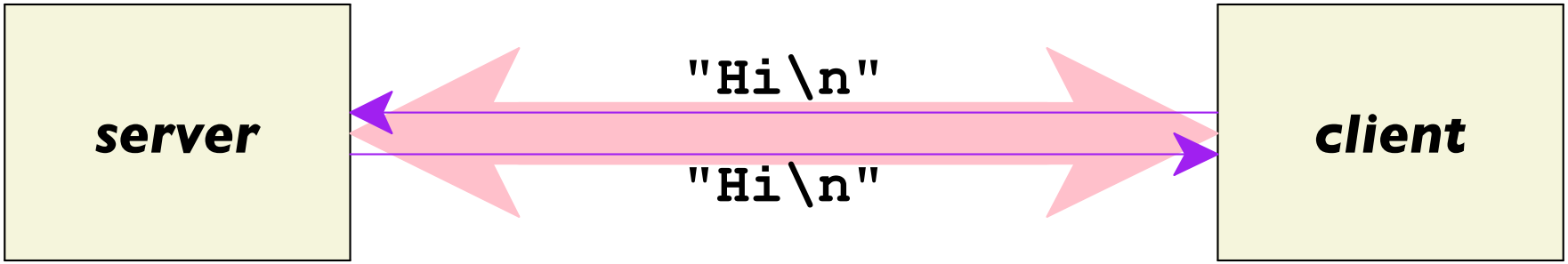



```
lnr = Open_listenfd(....);
fd = Accept(lnr, ....);

while (readline(fd, ....))
  Rio_writen(fd, ....);
```

```
fd = Open_clientfd(....);

Rio_writen(fd, "Hi\n", 3);

...
```

Echo Server

Another example: a server that echoes back every line




```
ln                                ....);
fd                                .);          fd = Open_clientfd(....);
```

Reading one byte at a time would be slow

```
while (readline(fd, ....))        Rio_writen(fd, "Hi\n", 3);
  Rio_writen(fd, ....);
                                  ...
```

Robust Buffered Reading

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);
ssize_t rio_readlineb(rio_t *rp, void *buf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *buf, size_t n);
```

`rio_initb` initializes a buffer to hold bytes that are read but not yet consumed

`rio_readlineb` fills the buffer as needed and consumes bytes that make a line

`rio_readnb` is like **`rio_readn`**, but keeps using the buffer

Echo Server

echo.c

```
int main(int argc, char **argv) {
  int listenfd, connfd;
  char client_hostname[MAXLINE], client_port[MAXLINE];
  struct sockaddr_storage clientaddr; /* Enough room for any addr */

  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    Getnameinfo((SA *) &clientaddr, clientlen,
                client_hostname, MAXLINE, client_port, MAXLINE, 0);
    printf("Connected to (%s, %s)\n", client_hostname, client_port);

    echo(connfd);

    Close(connfd);
  }
}
.....
```

Echo Server

echo.c

```
....

void echo(int connfd) {
  size_t n;
  char buf[MAXLINE];
  rio_t rio;

  Rio_readinitb(&rio, connfd);

  while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
    printf("server received %ld bytes\n", n);
    Rio_writen(connfd, buf, n);
  }
}
```

Telnet

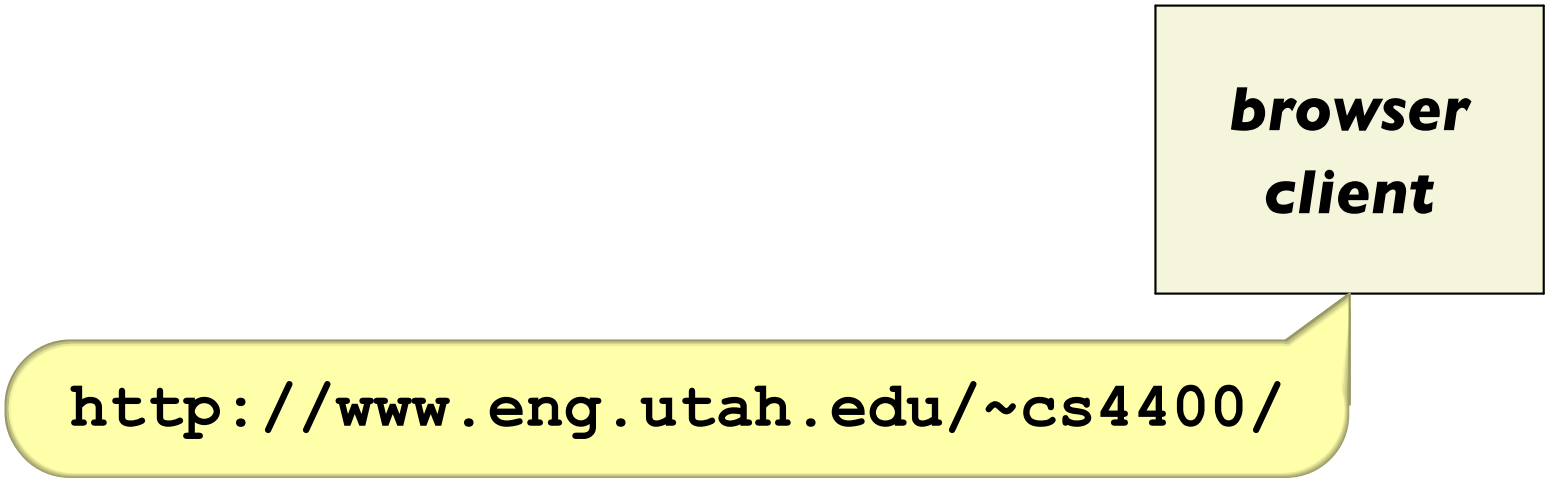
Instead of implementing an echo client, we can just use **`telnet`**

If the echo serever is at **`locahost`** on **`4567`**:

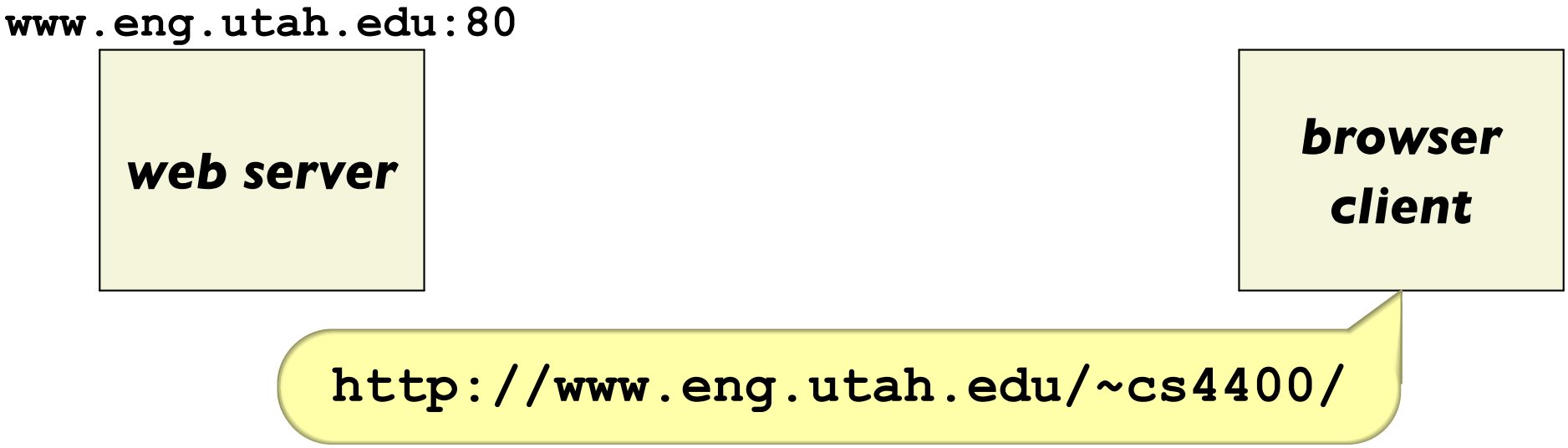
```
$ telnet localhost 4567
```

Use Ctl-] and then **`quit`** to stop

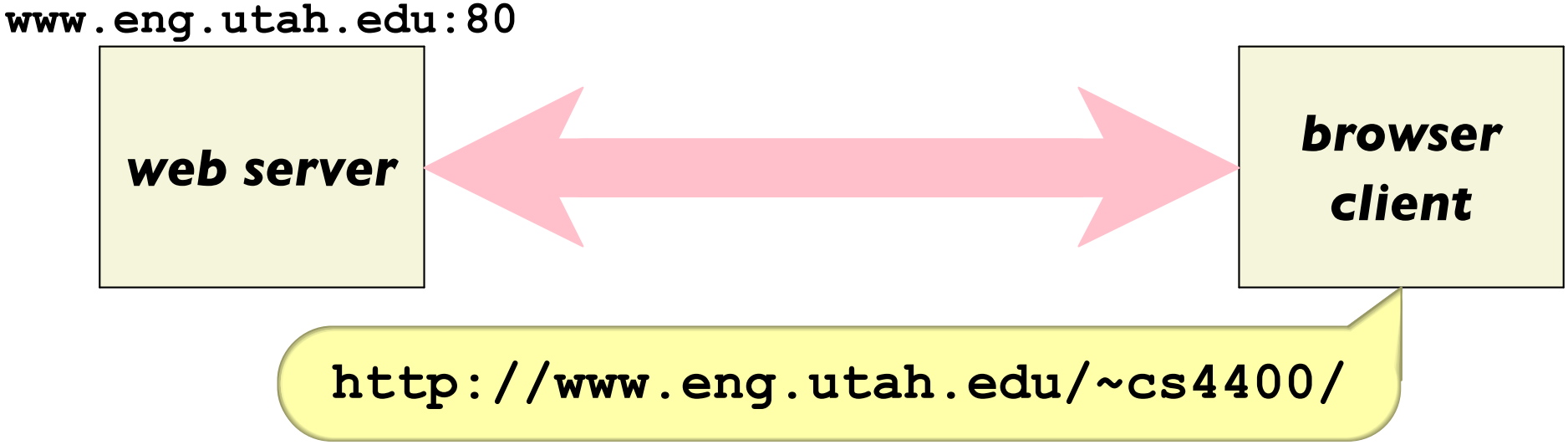
HTTP



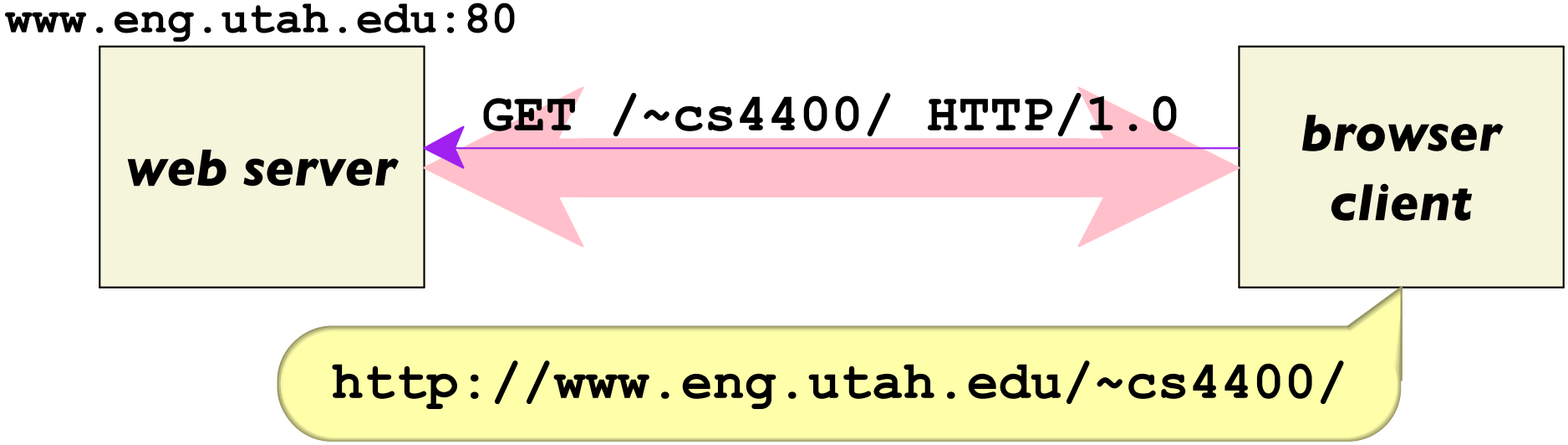
HTTP


www.eng.utah.edu:80
web server
browser client
http://www.eng.utah.edu/~cs4400/

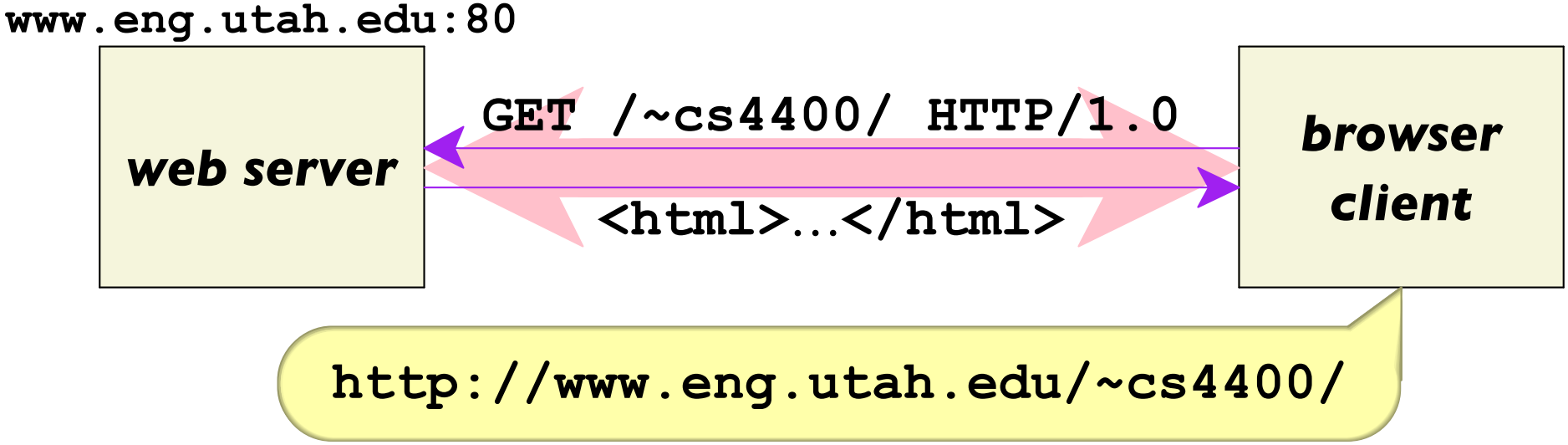
HTTP


www.eng.utah.edu:80
web server
browser client
http://www.eng.utah.edu/~cs4400/

HTTP


www.eng.utah.edu:80
web server
GET /~cs4400/ HTTP/1.0
browser client
http://www.eng.utah.edu/~cs4400/

HTTP

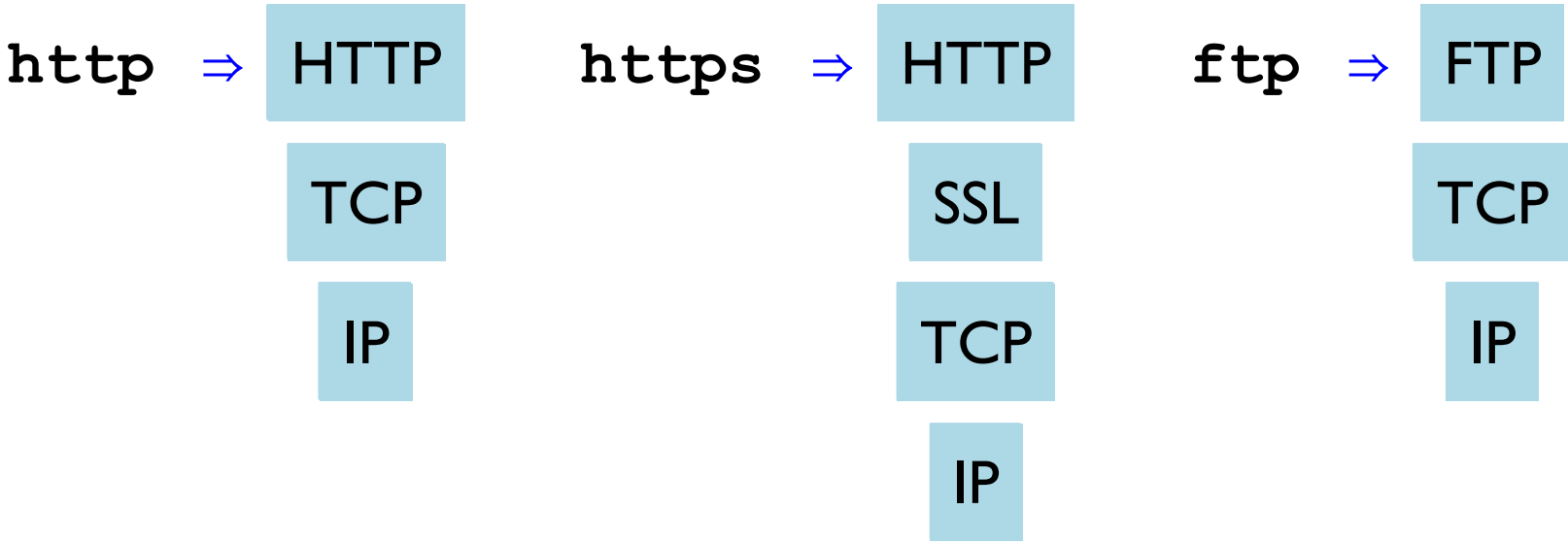

www.eng.utah.edu:80
web server
GET /~cs4400/ HTTP/1.0
<html>...</html>
browser client
http://www.eng.utah.edu/~cs4400/

URLs

scheme : / / *host* : *port* / *path* ? *query* # *fragment*

each color is optional

scheme — protocol to use



Assume **`http`** for the rest

URLs

scheme : / / *host* : *port* / *path* ? *query* # *fragment*

each color is optional

scheme — protocol to use

host — the server's host

`www.eng.utah.edu`

`google.com`

`127.0.0.1`

URLs

scheme : / / *host* : *port* / *path* ? *query* # *fragment*

each color is optional

scheme — protocol to use

host — the server's host

port — the server's port

defaults to 80

`http://localhost/` ⇒ port 80 on `localhost`

`http://localhost:8090/` ⇒ port 8090 on `localhost`

URLs

scheme : / / *host* : *port* / *path* ? *query* # *fragment*

each color is optional

scheme — protocol to use

host — the server's host

port — the server's port

path — item to get from the server, defaults to empty

Meaning of *path* is up to the server:

- could be a file
- could be a request to compute
- could be a request to change

URLs

scheme://*host*:*port*/*path*?*query*#*fragment*

each color is optional

scheme — protocol to use

host — the server's host

port — the server's port

path — item to get from the server, defaults to empty

`http://www.eng.utah.edu/~cs4400/network.pdf`

⇒ gets a file from the CADE filesystem

`https://goo.gl/om5FJq`

⇒ looks up redirection URL

URLs

scheme : / / *host* : *port* / *path* ? *query* # *fragment*

each color is optional

scheme — protocol to use

host — the server's host

port — the server's port

path — item to get from the server, defaults to empty

query — options related to *path*

Sequence of *key*=*value* separated by `&` or `;`

`http://www.youtube.com/watch?v=KFyuGneWhJ4`

`http://twitter.com/search?q=utah&src=typd`

Meaning of *query* is also up to the server

URLs

scheme : / / *host* : *port* / *path* ? *query* # *fragment*

each color is optional

scheme — protocol to use

host — the server's host

port — the server's port

path — item to get from the server, defaults to empty

query — options related to *path*

fragment — more options related to *path*

Use of *fragment* is up to the **client**

not sent to the server

`http://www.eng.utah.edu/~cs4400/#(part._staff)`

Encoding

scheme : / / *host* : *port* / *path* ? *query* # *fragment*

What if a *path* includes **?**?

What if a *query* includes **#**?

What if a *value* in a *query* includes **&**?

Percent encoding:

- Replace a character with **%***XX*
  - Each *X* is a hex digit
  - **0x***XX* is the character's value
- Replace a space with **+**

Access **lost+found** with **user** as **me&you**:

http://server.com/lost%2bfound?user=me%26you

HTTP

`http://`*host*`:`*port*`/`*path*`?`*query*`#`*fragment*

Client connects to *host*`:`*port* with TCP and sends...

Still more choices!

- **`GET`** mode — the default
- **`POST`** mode — better for sending data
- ...

HTTP

http://*host***:***port***/***path***?***query***#***fragment*

Client connects to *host*:*port* with TCP and sends...

GET /*path*?*query* **HTTP/1.0**CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

where CRLF is a two-byte sequence:

- carriage return (CR) character, which is ASCII 13
- a linefeed (LF) character, which is ASCII 10

HTTP

http://*host*:*port*/*path*?*query*#*fragment*

Client connects to *host*:*p*... Or **HTTP/1.1**

GET /*path*?*query* **HTTP/1.0**CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

where CRLF is a two-byte sequence:

- carriage return (CR) character, which is ASCII 13
- a linefeed (LF) character, which is ASCII 10

HTTP

http://*host*:*port*/*path*?*query*#*fragment*

Client connects to *host*:*port* with TCP and sends...

metadata for request
- wanted format
- ...

GET /*path*?*query* **HTTP/1.0**CRLF
field: *value*CRLF ← zero or more as *header*
CRLF

where CRLF is a two-byte sequence:

- carriage return (CR) character, which is ASCII 13
- a linefeed (LF) character, which is ASCII 10

HTTP

http://*host***:***port***/***path***?***query***#***fragment*

Client connects to *host*:*port* with TCP and sends...

GET /*path*?*query* **HTTP/1.0**CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

Adjust **echo.c** to print received lines

and point a web browser at it

HTTP

http://*host***:***port***/***path***?***query***#***fragment*

Client connects to *host*:*port* with TCP and sends...

GET /*path*?*query* **HTTP/1.0**CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

Server replies with

HTTP/1.0 *status* *status-message*CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

data

HTTP

http://*host*:*port*/*path*?*query*#*fragment*

Client connects to *host*:*port* with TCP and sends...

GET /*path*?*query* **HTTP/1.0**CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

Server replies with

HTTP/1.0 *status* *status-message*CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

data

metadata for request
- data size
- data format
- ...

HTTP

http://*host***:***port***/***path***?***query***#***fragment*

Client connects to *host*:*port* with TCP and sends...

GET /*path*?*query* **HTTP/1.0**CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

Server replies with

HTTP/1.0 *status* *status-message*CRLF

field: *value*CRLF

CRLF

data

status	*status-description*
200	OK
301	Moved permanently
400	Bad request
501	Not implemented

HTTP

http://*host***:***port***/***path***?***query***#***fragment*

Client connects to *host*:*port* with TCP and sends...

GET /*path*?*query* **HTTP/1.0**CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

Server replies with

HTTP/1.0 *status* *status-message*CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

data

telnet to a web server

HTTP

http://*host***:***port***/***path***?***query***#***fragment*

Client connects to *host*:*port* with TCP and sends...

POST /*path*?*query* **HTTP/1.0**CRLF

field: *value*CRLF ← zero or more as *header*

CRLF

data

Server reply is the same as for **GET**

Common Header Fields

`Content-Length` — length of response data or **`POST`** data

`Content-Type` — type response data or **`POST`** data

- **`text/html`**: HTML
- **`text/html; charset=utf-8`**: HTML with UTF-8 content
- **`application/x-www-form-urlencoded`**: POST data that uses the *query* format

`Connection` — **`close`** to get a single response

Field names are case-insensitive

TINY Web Server

The ***TINY Web Server*** is a useful web server in 250 lines of code

For each */path*`?`*query* request:

If */path* does not start **`/cgi-bin`**:

- Assume that *path* refers to a file
- Report **`Content-Length`** as file size
- Infer **`Content-Type`** from file extension
- Send file content as reply data

If */path* starts **`/cgi-bin`**:

- Assume that *path* refers to an executable
- Set **`QUERY_STRING`** environment variable to *query*
- Run executable to generate result

TINY Web Server — Main

tiny.c

```
int main(int argc, char **argv) {
  int listenfd, connfd;
  char hostname[MAXLINE], port[MAXLINE];
  struct sockaddr_storage clientaddr;

  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(clientaddr);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    Getnameinfo((SA *) &clientaddr, clientlen, hostname, MAXLINE,
                port, MAXLINE, 0);
    printf("Accepted connection from (%s, %s)\n", hostname, port);

    doit(connfd);

    Close(connfd);
  }
}
....
```

TINY Web Server — Handling a Connection

tiny.c

```
void doit(int fd) {
  int is_static;
  char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
  char filename[MAXLINE], cgiargs[MAXLINE];
  rio_t rio;

  Rio_readinitb(&rio, fd);

  if (!Rio_readlineb(&rio, buf, MAXLINE))
    return;
  sscanf(buf, "%s %s %s", method, uri, version);

  read_requesthdrs(&rio);

  is_static = parse_uri(uri, filename, cgiargs);

  ....
}
```

TINY Web Server — Reading Headers

tiny.c

```
void read_requesthdrs(rio_t *rp) {
  char buf[MAXLINE];

  Rio_readlineb(rp, buf, MAXLINE);
  while (strcmp(buf, "\r\n"))
    Rio_readlineb(rp, buf, MAXLINE);
}
```

TINY Web Server — Parsing URLs

tiny.c

```
int parse_uri(char *uri, char *filename, char *cgiargs) {
  ....

  if (!strstr(uri, "cgi-bin")) {
    ....
    if (uri[strlen(uri)-1] == '/')
      strcat(filename, "home.html");
    ....
    return 1; /* => static content */
  } else {
    ....
    return 0; /* => dynamic content */
  }
}
```

TINY Web Server — Handling a Connection

tiny.c

```
void doit(int fd) {
  struct stat sbuf;

  ....
  if (stat(filename, &sbuf) < 0) {
    clienterror(fd, filename, "404", "Not found",
                ....);
    return;
  }

  if (is_static) {
    ....
    serve_static(fd, filename, sbuf.st_size);
  } else {
    ....
    serve_dynamic(fd, filename, cgiargs);
  }
}
```

TINY Web Server — Serving Files

tiny.c

```
void serve_static(int fd, char *filename, int filesize) {
  int srcfd;
  char *srcp, filetype[MAXLINE], buf[MAXBUF];

  get_filetype(filename, filetype);
  sprintf(buf, "HTTP/1.0 200 OK\r\n");
  sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
  sprintf(buf, "%sConnection: close\r\n", buf);
  sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
  sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
  Rio_writen(fd, buf, strlen(buf));

  srcfd = Open(filename, O_RDONLY, 0);
  srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
  Close(srcfd);
  Rio_writen(fd, srcp, filesize);
  Munmap(srcp, filesize);
}
```

TINY Web Server — Inferring a File Type

tiny.c

```
void get_filetype(char *filename, char *filetype) {
  if (strstr(filename, ".html"))
    strcpy(filetype, "text/html");
  else if (strstr(filename, ".gif"))
    strcpy(filetype, "image/gif");
  else if (strstr(filename, ".png"))
    strcpy(filetype, "image/png");
  else if (strstr(filename, ".jpg"))
    strcpy(filetype, "image/jpeg");
  else
    strcpy(filetype, "text/plain");
}
```

TINY Web Server — Serving Generated Content

tiny.c

```
void serve_dynamic(int fd, char *filename, char *cgiargs) {
 char buf[MAXLINE], *emptylist[] = { NULL };

 /* Return first part of HTTP response */
 sprintf(buf, "HTTP/1.0 200 OK\r\n");
 Rio_writen(fd, buf, strlen(buf));
 sprintf(buf, "Server: Tiny Web Server\r\n");
 Rio_writen(fd, buf, strlen(buf));

 if (Fork() == 0) {
   setenv("QUERY_STRING", cgiargs, 1);
   Dup2(fd, STDOUT_FILENO); /* Redirect stdout to client */
   Execve(filename, emptylist, environ);
 }
 Wait(NULL);
}
```

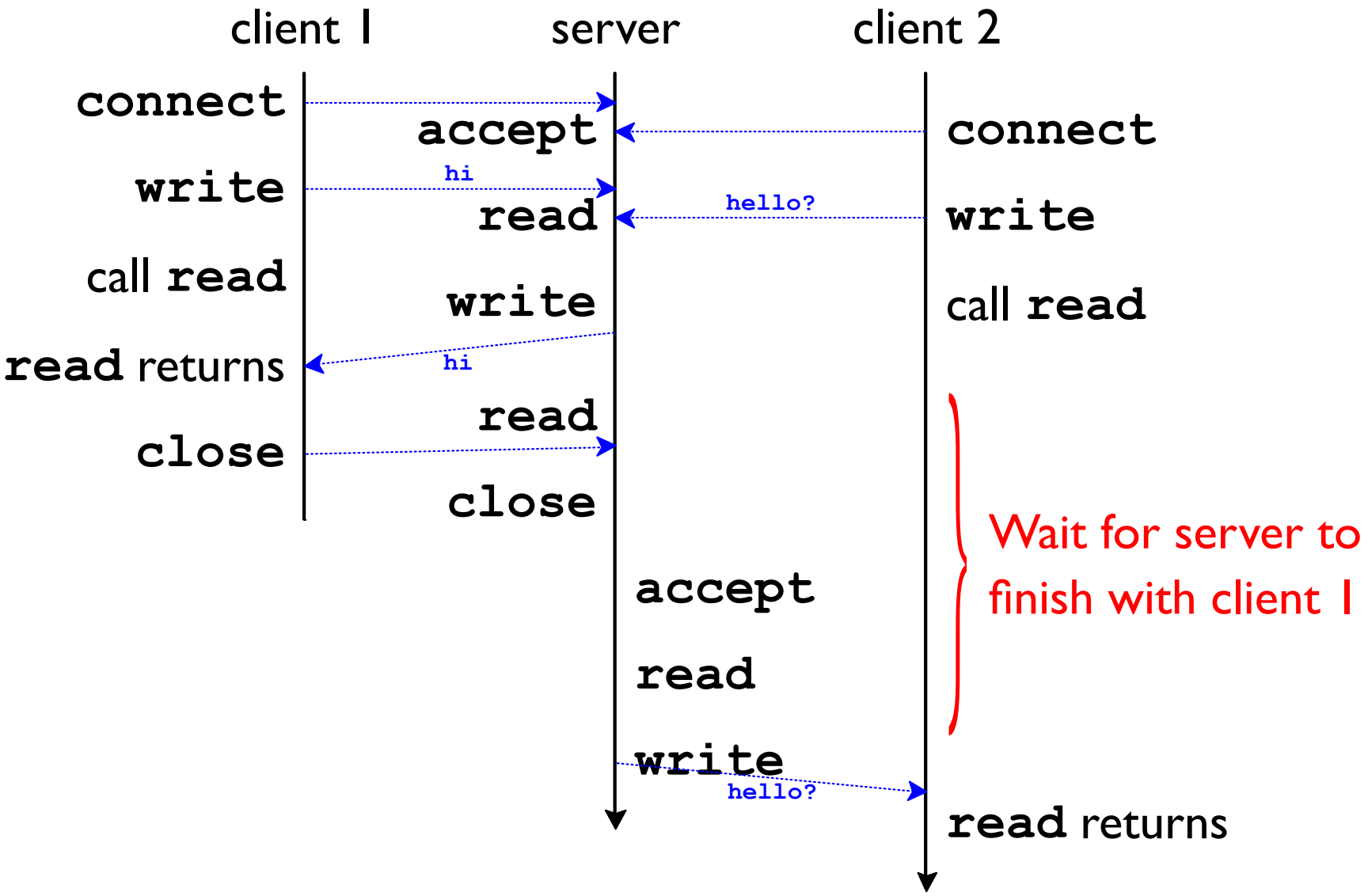
Echo Server and Multiple Clients

The current echo server operates ***sequentially***:

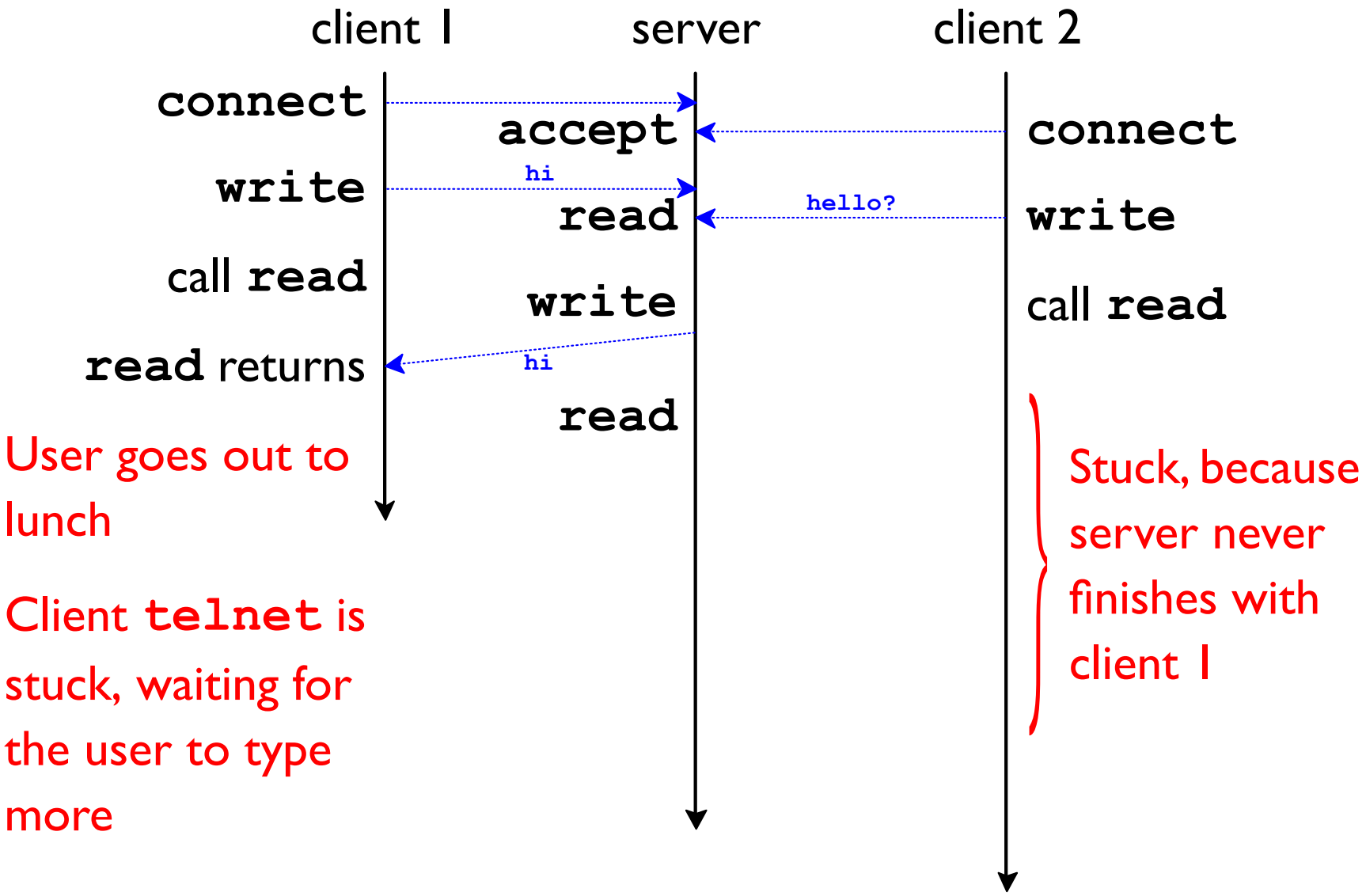
- Only one client is served at a time
- After a client sends EOF, next client can be served

TCP listener queue allows multiple clients to connect,
but only one of them receives echoes at a time

Echo Server and Multiple Clients



Echo Server and Multiple Clients

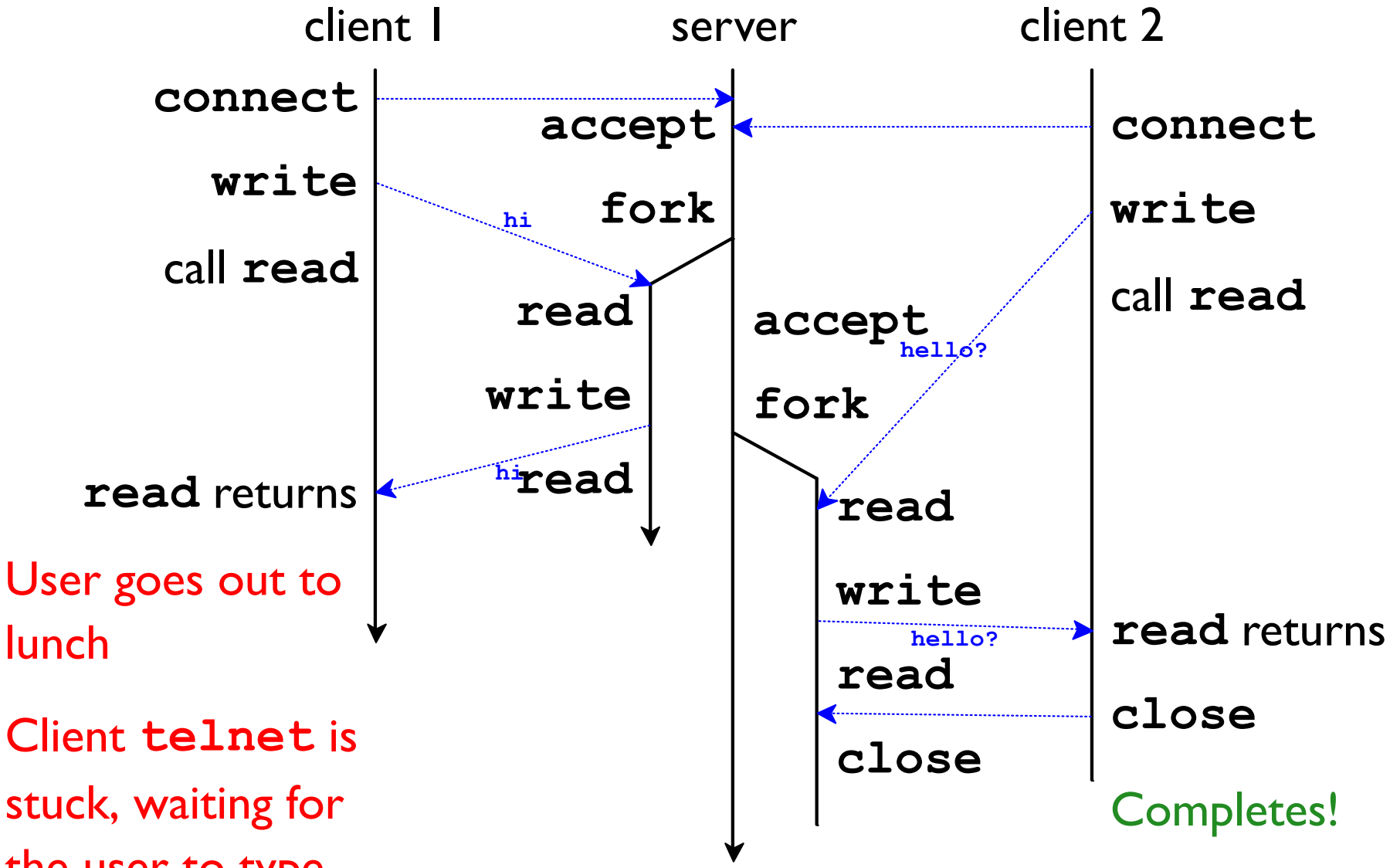


Echo Concurrency

Options for serving clients ***concurrently***:

- Per-connection processes **`fork`**
- Per-connection threads **`pthread_create`**
- Event-driven with multiplexed I/O **`select`**

Echo Concurrency by Processes



Echo Concurrency by Processes

p_echo.c

```
int main(int argc, char **argv) {
  ....
  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    if (Fork() == 0) {
       Close(listenfd);
       Getnameinfo((SA *) &clientaddr, clientlen,
                   client_hostname, MAXLINE, client_port, MAXLINE, 0);
       printf("Connected to (%s, %s)\n", client_hostname, client_port);

       echo(connfd);

       Close(connfd);
       exit(0);
    }
    Close(connfd);
  }
}
.....
```

Echo Concurrency by Processes

p_echo.c

```
int main(int argc, char **argv) {
  ....
  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    if (Fork() == 0) {
       Close(listenfd);
       Getnameinfo((SA *) &clientaddr, clientlen,
                   client_hostname, MAXLINE, client_port, MAXLINE, 0);
       printf("Connected to (%s, %s)\n", client_hostname, client_port);

       echo(connfd);

       Close(connfd);
       exit(0);
    }
    Close(connfd);
  }
}
.....
```

an important **Close** to avoid a leak

Echo Concurrency by Processes

p_echo.c

```
int main(int argc, char **argv) {
  ....
  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    if (Fork() == 0) {
       Close(listenfd);
       Getnameinfo((SA *) &clientaddr, clientlen,
                   client_hostname, MAXLINE, client_port, MAXLINE, 0);
       printf("Connected to (%s, %s)\n", client_hostname, client_port);

       echo(connfd);

       Close(connfd);
       exit(0);
    }
    Close(connfd);
  }
}
.....
```

still leaking PIDs — but where to **waitpid**?

Echo Concurrency by Processes

p_echo.c

```
void sigchld_handler(int sig)  {
  while (waitpid(-1, 0, WNOHANG) > 0)
    ;
}

int main(int argc, char **argv) {
  ....
  Signal(SIGCHLD, sigchld_handler);
  ....
}

....
```

Echo Concurrency by Processes

✓ Processes are great when each connection is independent

✗ Processes are not so great if connections interact

Try making `p_echo.c` track total bytes sent

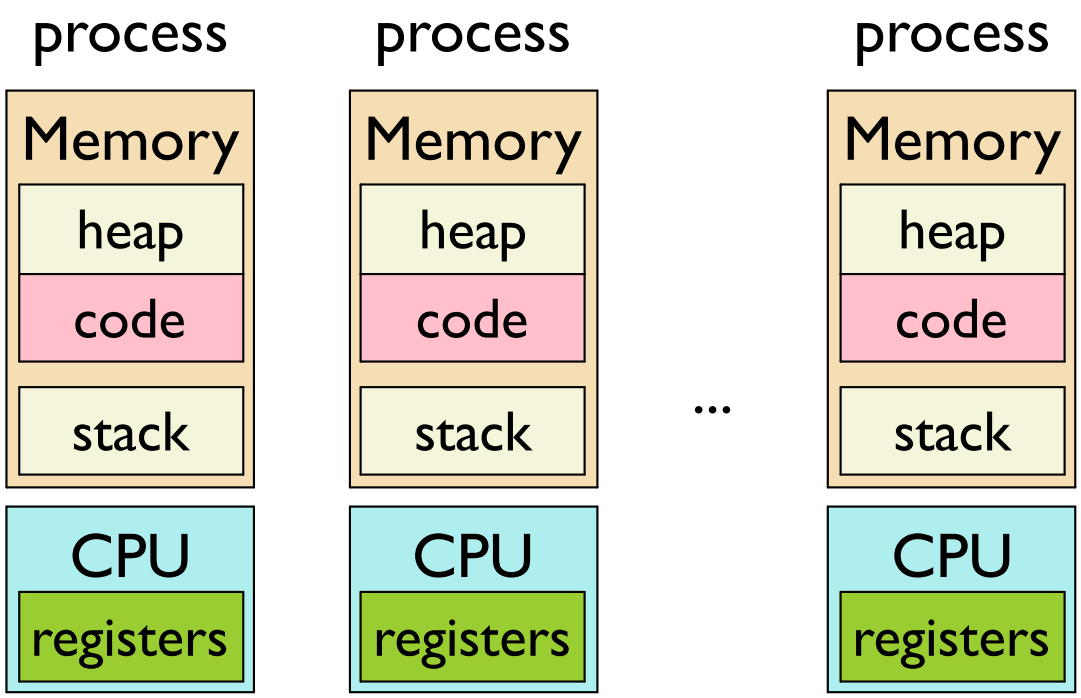
Threads

A ***thread*** is like a process in that it has its own stack, registers, and control flow

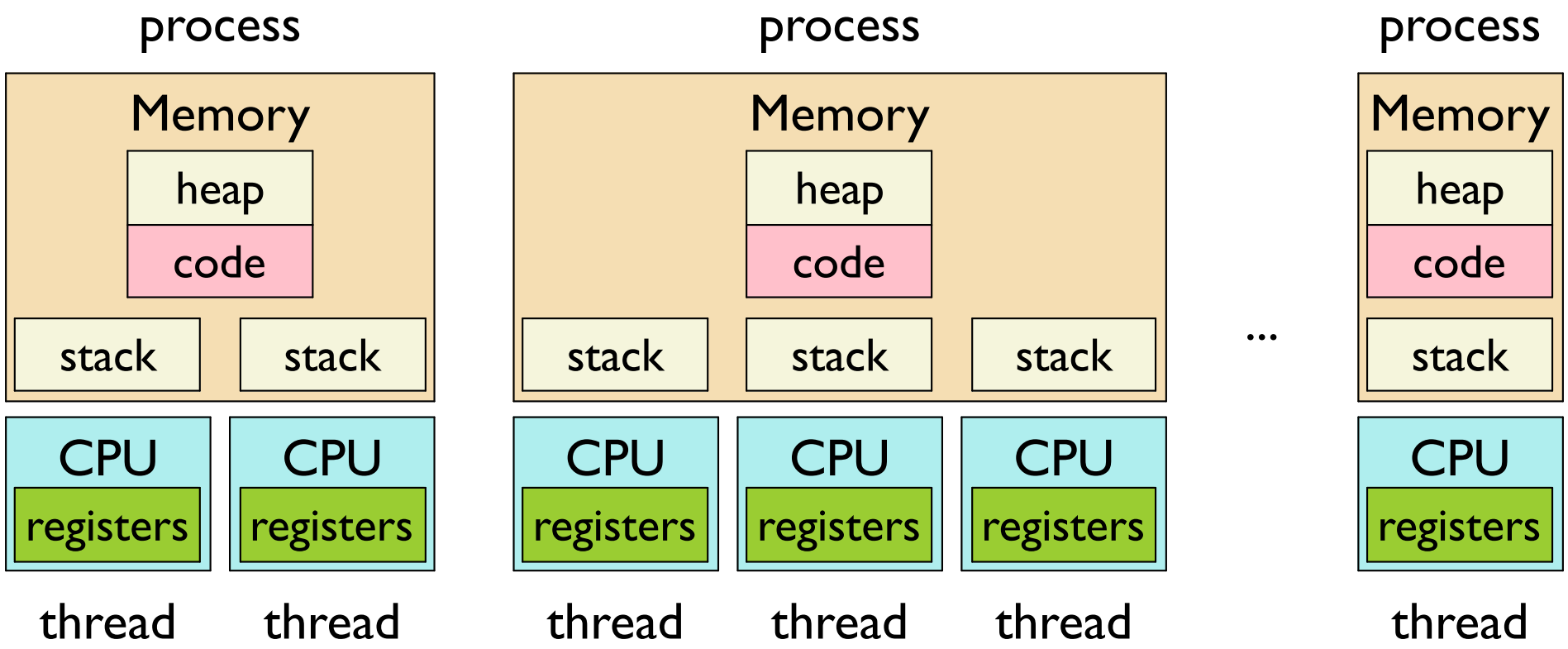
Threads within a process share a virtual address space, file descriptors, etc.

⇒ easier communication among threads

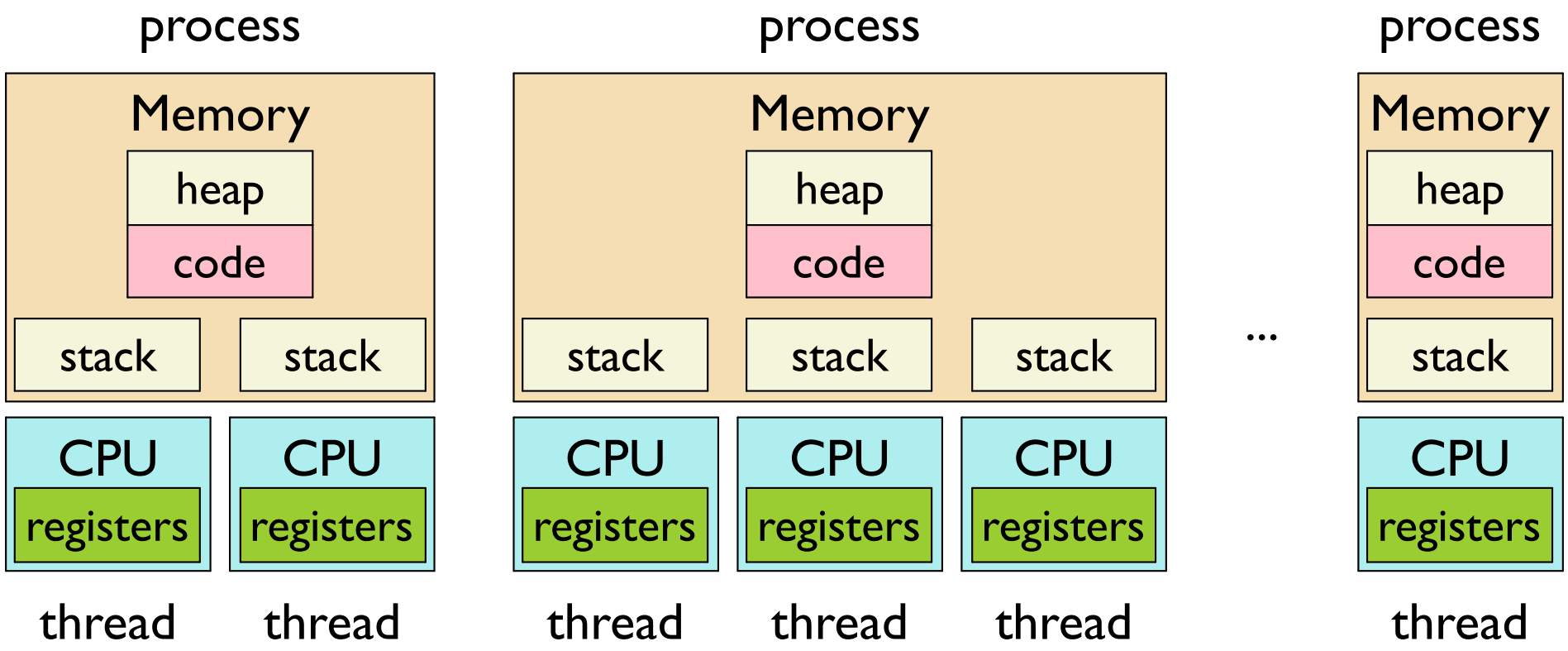
Processes: The Illusion



Threads: The Illusion



Threads: The Illusion



⇒ **`pthread_create`** cannot return twice like **`fork`**

Creating Threads

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_function)(void *),
                   void *start_arg);
```

Creates a new thread that calls

```
start_function(start_arg);
```

Handle to the new thread written to **`*thread`**

analogous to a PID

Options specified in **`attr`**, which can be **`NULL`**

Creating Threads

```
#include "csapp.h"

int count = 0;

void *show_var(void *name) {
  printf("%s %p\n", name, &name);
  count++;
  return NULL;
}

int main() {
  pthread_t th;
  show_var("orig");
  Pthread_create(&th, NULL, show_var, "new");
  Sleep(1);
  printf("%d\n", count);
  return 0;
}
```

Copy

gcc -pthread ....

Processes versus Threads

`pid_t`	**`pthread_t`**
`fork`	**`pthread_create`**
`waitpid`	**`pthread_join`**
`getpid`	**`pthread_self`**
`exit` or **`return`**	**`thread_exit`** or **`return`**
`kill`	**`pthread_cancel`**

Processes versus Threads

`pid_t`	`pthread_t`
`fork`	`pthread_create`
`waitpid`	`pthread_join`
`getp`	`pthrea`
`exit` or `return`	`thread_exit` or `return`
`kill`	`pthread_cancel`

from `main`

from `start_function`

Processes versus Threads

`pid_t`	**`pthread_t`**
`fork`	**`pthread_create`**
`waitpid`	**`pthread_join`**
`getpid`	**`pthread_self`**
`exit` or **`return`**	**`thread_exit`** or **`return`**
`kill`	**`pthread_cancel`**

sortof — there's also **`pthread_kill`**

Waiting for a Thread to Complete

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **result_p);
```

Waits for **`thread`** to finish

Puts the thread's return value in **`*result_p`**

Reaps the thread's identity (so **`thread`** must not be used anymore)

Waiting for a Thread to Complete

```
#include "csapp.h"

int count = 0;

void *show_var(void *name) {
  printf("%s %p\n", name, &name);
  count++;
  return name;
}

int main() {
  pthread_t th;
  void *result;
  show_var("orig");
  Pthread_create(&th, NULL, show_var, "new");
  Pthread_join(th, &result);
  printf("%s\n", result);
  return 0;
}
```

Copy

Threads are Not Hierarchical

Unlike processes, a thread doesn't have a parent or children

Any thread can **`pthread_join`** any other thread

Threads are ***peers***

The ***main thread*** is only a little special:

- It can return from **`main`** to exit the process
- Signals sent to the process go to the main thread

Threads are Not Hierarchical

```
#include "csapp.h"

void *go1(void *ignored) {
  Sleep(1);
  printf("1\n");
  return NULL;
}

void *go2(void *th) {
  Pthread_join(*(pthread_t *)th, NULL);
  printf("2\n");
  return NULL;
}

int main() {
  pthread_t one, two;
  Pthread_create(&one, NULL, go1, NULL);
  Pthread_create(&two, NULL, go2, &one);
  Pthread_join(two, NULL);
  return 0;
}
```

Copy

Threads are Not Hierarchical

```
#include "csapp.h"

void *go1(void *ignored) {
  Sleep(1);
  printf("1\n");
  return NULL;
}

void *go2(void *th) {
  Pthread_join(*(pthread_t *)th, NULL);
  printf("2\n");
  return NULL;
}

int main() {
  pthread_t one, two;
  Pthread_create(&one, NULL, go1, NULL);
  Pthread_create(&two, NULL, go2, &one);
  Pthread_join(two, NULL);
  return 0;
}
```

Copy

Prints 1 then 2

Comment out **`Pthread_join`** in **`main`**

⇒ exits without printing

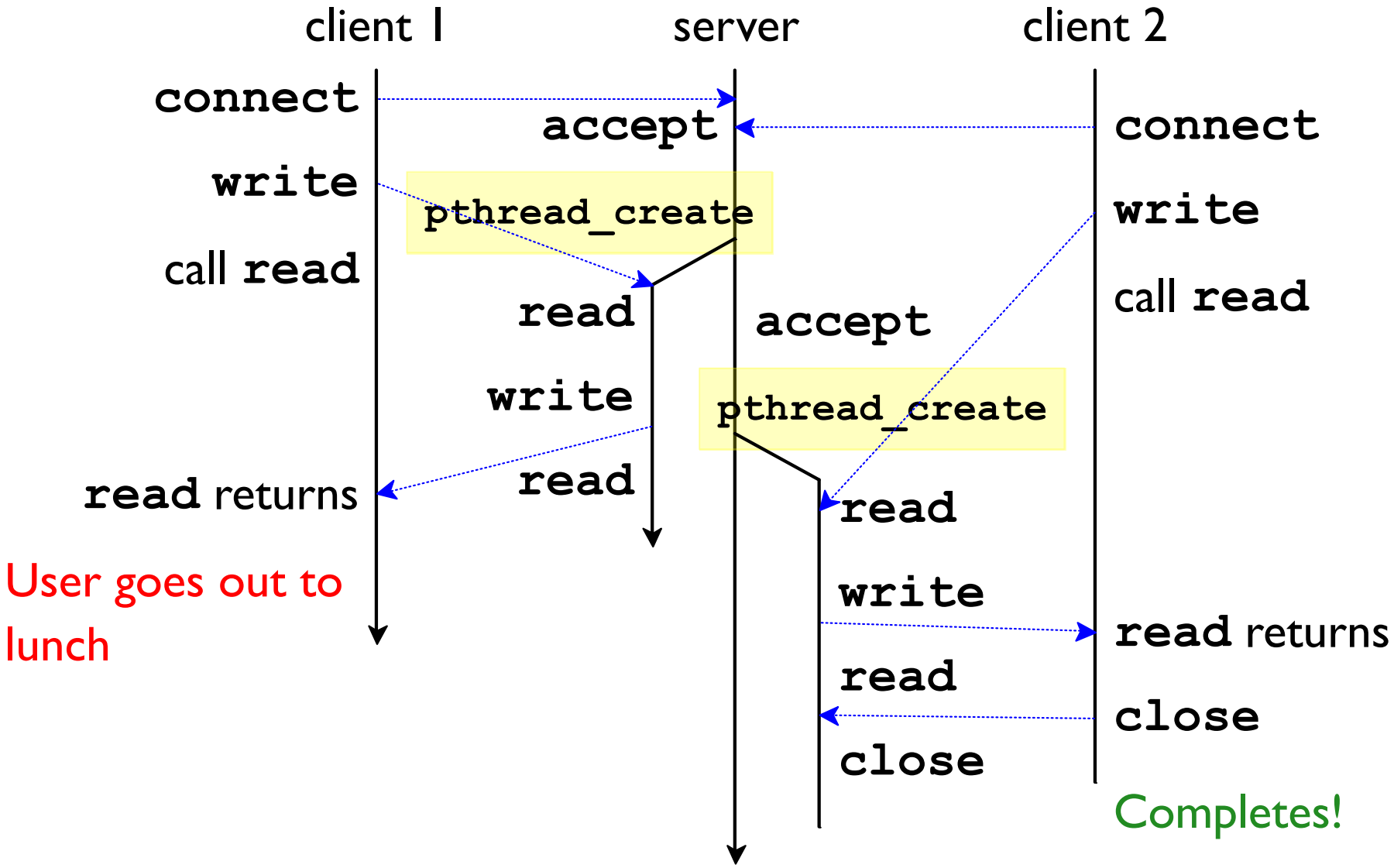
Reaping Thread Identity without Waiting

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Reaps the thread's identity (so **`thread`** must not be used anymore), even if **`thread`** is still running

replaces using **`waitpid`** in a signal handler

Echo Concurrency by Threads



Echo Concurrency by Threads

t_echo.c

```
int main(int argc, char **argv) {
  ....

  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    Getnameinfo((SA *) &clientaddr, clientlen,
          client_hostname, MAXLINE, client_port, MAXLINE, 0);
    printf("Connected to (%s, %s)\n", client_hostname, client_port);

    connfd_p = malloc(sizeof(int));
    *connfd_p = connfd;
    Pthread_create(&th, NULL, echo, connfd_p);
    Pthread_detach(th);
  }

  return 0;
}
....
```

Echo Concurrency by Threads

t_echo.c

```
int main(int argc, char **argv) {
  ....

  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    Getnameinfo((SA *) &clientaddr, clientlen,
          client_hostname, MAXLINE, client_port, MAXLINE, 0);
    printf("Connected to (%s, %s)\n", client_hostname, client_port);

    connfd_p = malloc(sizeof(int));
    *connfd_p = connfd;
    Pthread_create(&th, NULL, echo, connfd_p);
    Pthread_detach(th);
  }

  return 0;
}
....
```

Don't need to wait

Don't need thread result

Echo Concurrency by Threads

t_echo.c

```
int main(int argc, char **argv) {
  ....

  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    Getnameinfo((SA *) &clientaddr, clientlen,
          client_hostname, MAXLINE, client_port, MAXLINE, 0);
    printf("Connected to (%s, %s)\n", client_hostname, client_port);

    connfd_p = malloc(sizeof(int));
    *connfd_p = connfd;
    Pthread_create(&th, NULL, echo, connfd_p);
    Pthread_detach(th);
  }

  return 0;
}
....
```

&connfd doesn't work

Echo Concurrency by Threads

t_echo.c

```
....
void *echo(void *connfd_p) {
  int connfd = *(int *)connfd_p;
  size_t n;
  char buf[MAXLINE];
  rio_t rio;

  free(connfd_p);

  Rio_readinitb(&rio, connfd);
  while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
    printf("server received %ld bytes\n", n);
    Rio_writen(connfd, buf, n);
  }

  Close(connfd);

  return NULL;
}
```

Towards Multiplexed I/O

```
for (j = 0; j < NUM_CLIENTS; j++)
  fds[j] = Open_clientfd(hostname, port);

for (j = 0; j < NUM_CLIENTS; j++) {
  snprintf(buf, MAXBUF, "client%d\n", j);
  Rio_writen(fds[j], buf, strlen(buf));
}

for (j = 0; j < NUM_CLIENTS; j++) {
  snprintf(buf, MAXBUF, "client%d\n", j);
  amt = Rio_readn(fds[j], rbuf, strlen(buf));
  rbuf[amt] = 0;
  if (strcmp(buf, rbuf))
    app_error("didn't get expected echo");
}

for (j = 0; j < NUM_CLIENTS; j++)
  Close(fds[j]);
```

Towards Multiplexed I/O

```
for (j = 0; j < NUM_CLIENTS; j++)
  fds[j] = Open_clientfd(hostname, port);

for (j = 0; j < NUM_CLIENTS; j++) {
  snprintf(buf, MAXBUF, "client%d\n", j);
  Rio_writen(fds[j], buf, strlen(buf));
}

for (j = 0; j < NUM_CLIENTS; j++) {
  snprintf(buf, MAXBUF, "client%d\n", j);
  amt = Rio_readn(fds[j], rbuf, strlen(buf));
  rbuf[amt] = 0;
  if (strcmp(buf, rbuf))
    app_error("didn't get expected echo");
}

for (j = 0; j < NUM_CLIENTS; j++)
  Close(fds[j]);
```

Makes **`NUM_CLIENTS`** "concurrent" connections by explicitly spending a little time on each one

Towards Multiplexed I/O

The same idea can work for the echo server, but...

Need a way to check whether any data is available

Towards Multiplexed I/O

The same idea can work for the echo server, but...

- ~~Need a way to check whether any data is available~~
- Need a way to wait until *some* connection has data

select

```
#include <sys/select.h>

int select(int nfds,
           fd_set *readfds, fd_set *writefds, fd_set *errorfds,
           struct timeval *timeout);

void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
int  FD_ISSET(int fd, fd_set *fdset);
```

Blocks until

- a file descriptor in **`readfds`** has input;
- a file descriptor in **`writeds`** can hold output;
- a file descriptor in **`errorfds`** has an error; or
- **`timeout`** time passes

and clears non-ready in **`readfds`**, **`writefds`**, and **`errorfds`**

Echo Concurrency by Events

e_echo.c

```
typedef struct {
  int maxfd;        /* Largest descriptor in read_set */
  fd_set read_set;  /* All active descriptors */
  fd_set ready_set; /* Subset ready for reading  */
  int nready;       /* Number of ready descriptors */
  int maxi;         /* Highwater index into client array */
  int clientfd[FD_SETSIZE];    /* Active descriptors */
  rio_t clientrio[FD_SETSIZE]; /* Active read buffers */
} pool;
```

Echo Concurrency by Events

e_echo.c

```
int main(int argc, char **argv) {
  ....
  static pool pool;

  listenfd = Open_listenfd(argv[1]);
  init_pool(listenfd, &pool);

  while (1) {
    pool.ready_set = pool.read_set;
    pool.nready = Select(pool.maxfd+1,
                         &pool.ready_set, NULL, NULL,
                         NULL);

    if (FD_ISSET(listenfd, &pool.ready_set)) {
      clientlen = sizeof(struct sockaddr_storage);
      connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
      ....
      add_client(connfd, &pool);
    }
    check_clients(&pool);
  }
}
```

Echo Concurrency by Events

e_echo.c

```
void init_pool(int listenfd, pool *p) {
  /* Initially, there are no connected descriptors */
  int i;
  p->maxi = -1;
  for (i=0; i< FD_SETSIZE; i++)
    p->clientfd[i] = -1;

  /* Initially, listenfd is only member of select read set */
  p->maxfd = listenfd;
  FD_ZERO(&p->read_set);
  FD_SET(listenfd, &p->read_set);
}
```

Echo Concurrency by Events

e_echo.c

```
void add_client(int connfd, pool *p) {
  int i;
  p->nready--;
  for (i = 0; i < FD_SETSIZE; i++)
    if (p->clientfd[i] < 0) {
      p->clientfd[i] = connfd;
      Rio_readinitb(&p->clientrio[i], connfd);

      FD_SET(connfd, &p->read_set);

      if (connfd > p->maxfd)
        p->maxfd = connfd;
      if (i > p->maxi)
        p->maxi = i;
      break;
    }
  if (i == FD_SETSIZE)
    app_error("add_client error: Too many clients");
}
```

Echo Concurrency by Events

e_echo.c

```
void check_clients(pool *p) {
  int i, connfd, n;
  char buf[MAXLINE];

  for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
    connfd = p->clientfd[i];
    if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
      rio_t rio = p->clientrio[i];
      p->nready--;
      if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        byte_cnt += n;
        printf("Server received %d (%d total) bytes on fd %d\n",
               n, byte_cnt, connfd);
        Rio_writen(connfd, buf, n);
      } else {
        Close(connfd);
        FD_CLR(connfd, &p->read_set);
        p->clientfd[i] = -1;
      }
    }
  }
}
```

Echo Concurrency

Per-connection processes **`fork`**

- ✓ Easy for independent
- ✗ Difficult for cooperating

Per-connection threads **`pthread_create`**

- ✓ Easy for independent or cooperating
- ✗ Maybe *too* easy for cooperating...

Event-driven with multiplexed I/O **`select`**

- ✓ Complete control of scheduling
- ✗ Manual task switching

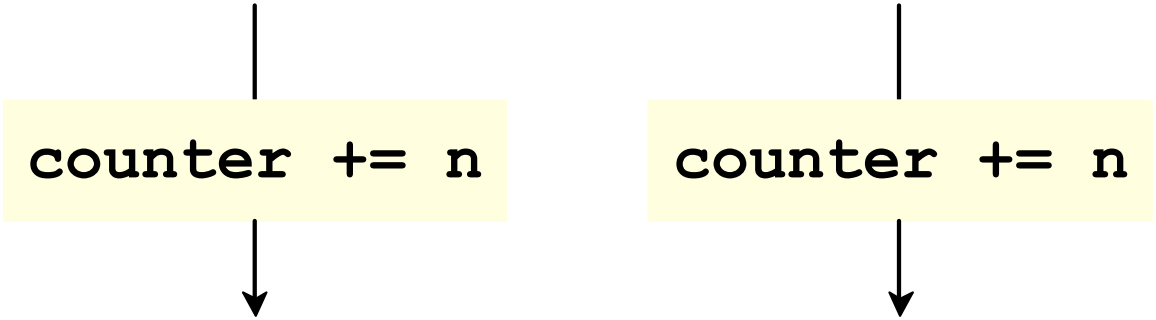
Sharing with Theads

Try changing **`t_echo.c`** to count total bytes:

`t_echo.c`

```
....
static size_t counter = 0;

int main() {
  ....
  Pthread_create(&th, NULL, echo, connfd_p);
  ....
}

void *echo(void *connfd_p) {
  ....
  while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
    // printf("server received %ld bytes\n", n);
    counter += n;
    Rio_writen(connfd, buf, n);
  }
  printf("total bytes so far: %ld\n", counter);
  ....
}
```

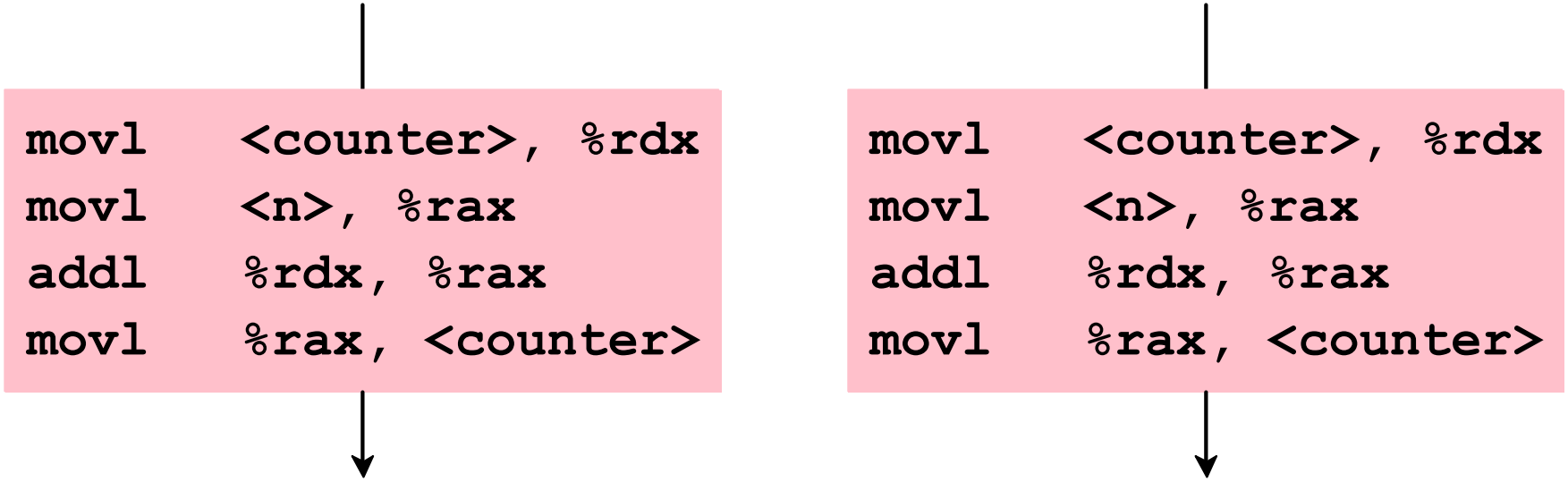
Concurrent Variable Updates



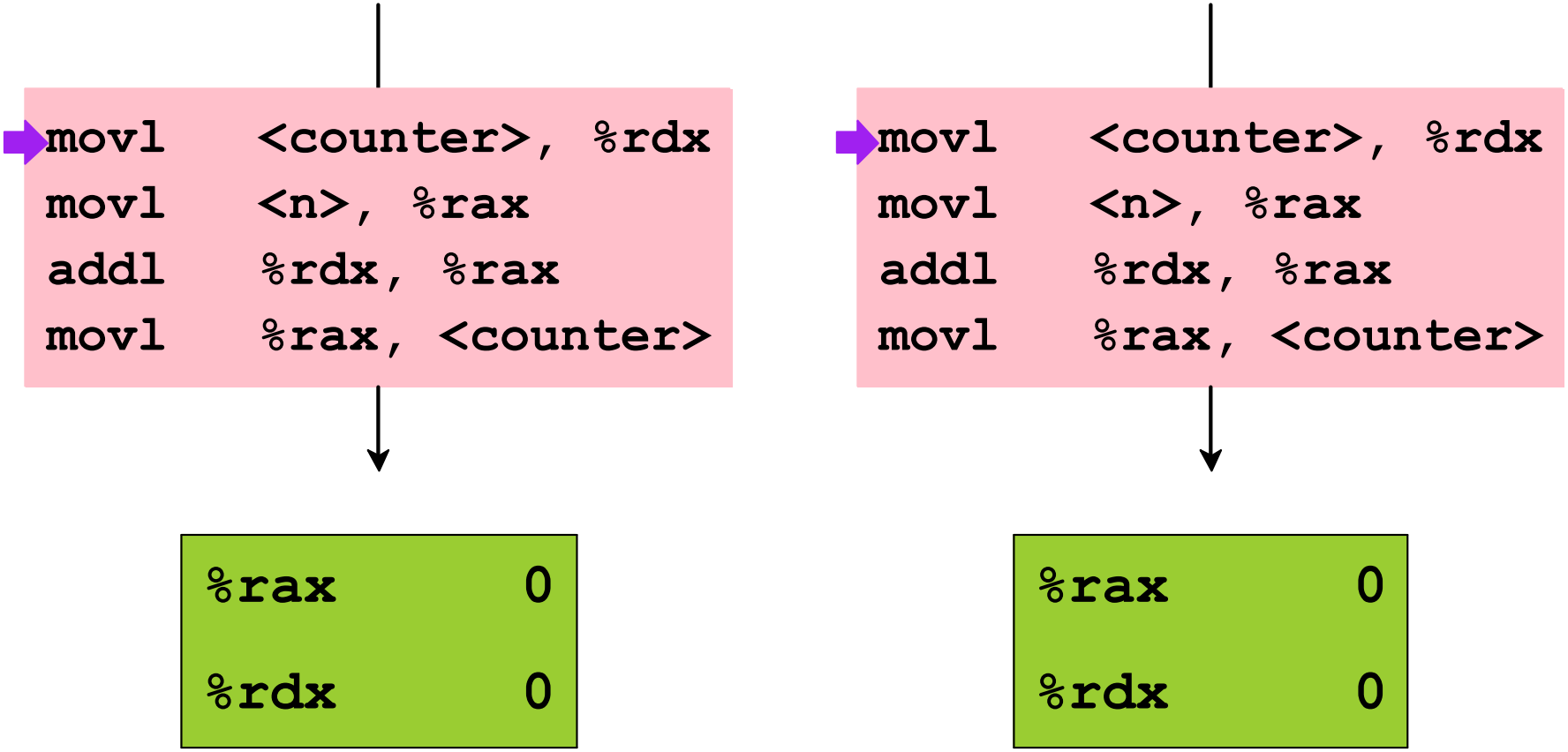
Problem: the program has a ***race condition***

Two threads race to update `counter`

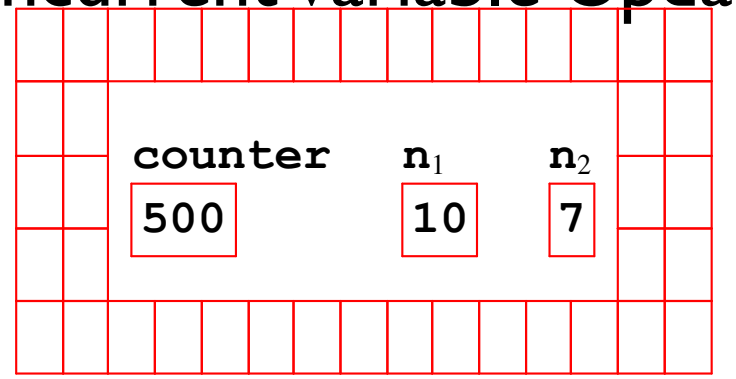
Concurrent Variable Updates



Concurrent Variable Updates



Concurrent Variable Updates



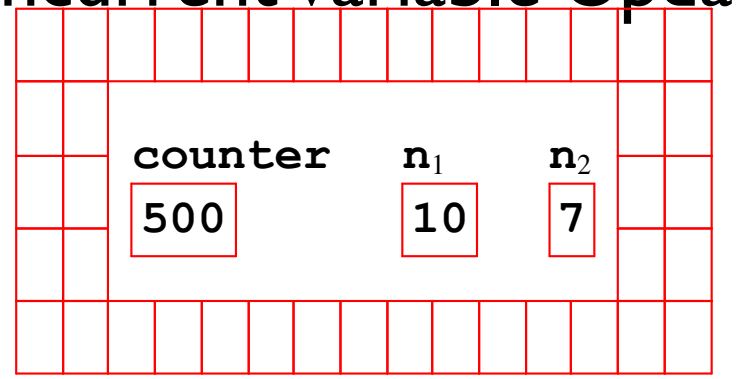

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	0
%rdx	0

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	0
%rdx	0

Concurrent Variable Updates



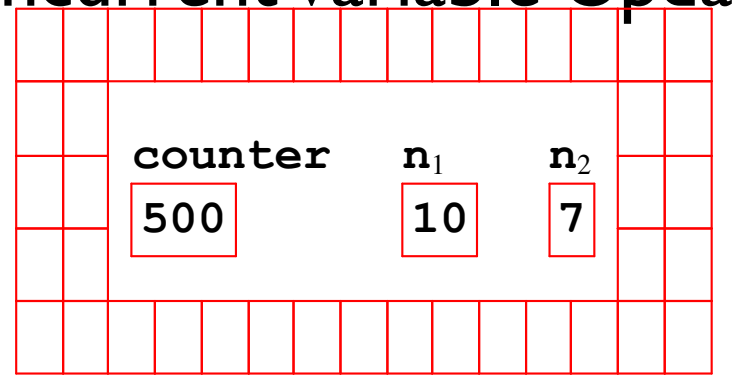

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	0
%rdx	500

%rax	0
%rdx	0

Concurrent Variable Updates



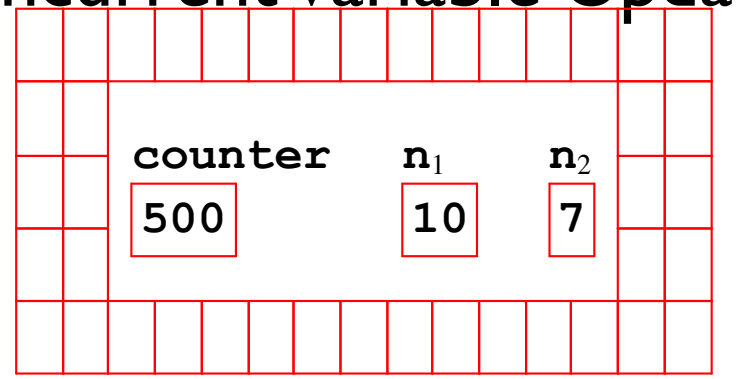

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	0
%rdx	500

%rax	7
%rdx	500

Concurrent Variable Updates



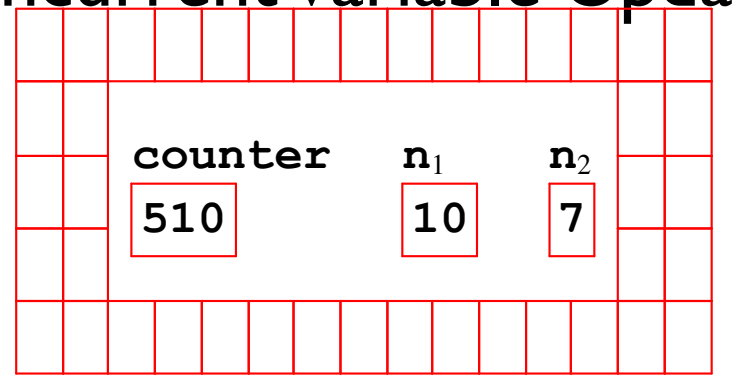

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	510
%rdx	500

%rax	7
%rdx	500

Concurrent Variable Updates



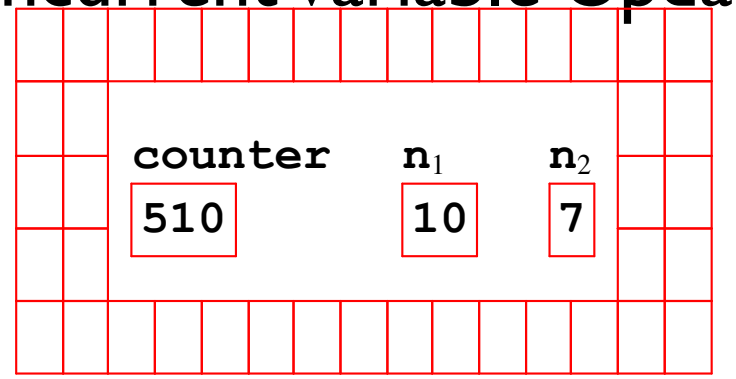

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	510
%rdx	500

%rax	7
%rdx	500

Concurrent Variable Updates



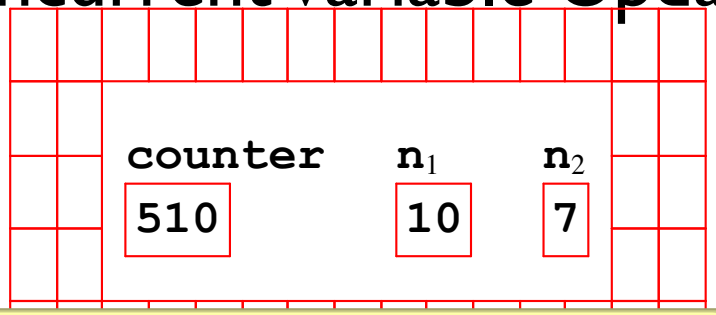

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	510
%rdx	500

%rax	507
%rdx	500

Concurrent Variable Updates



read–add–write sequence is not ***atomic***

```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

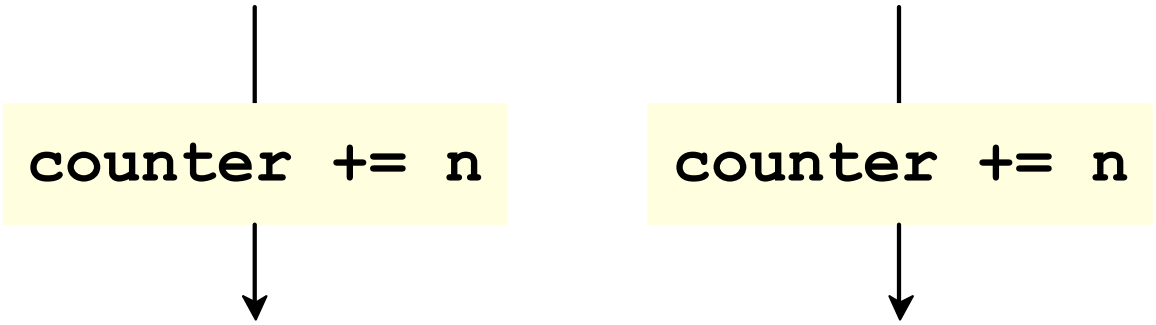
```
movl    <counter>, %rdx
movl    <n>, %rax
addl    %rdx, %rax
movl    %rax, <counter>
```

%rax	510
%rdx	500

%rax	507
%rdx	500

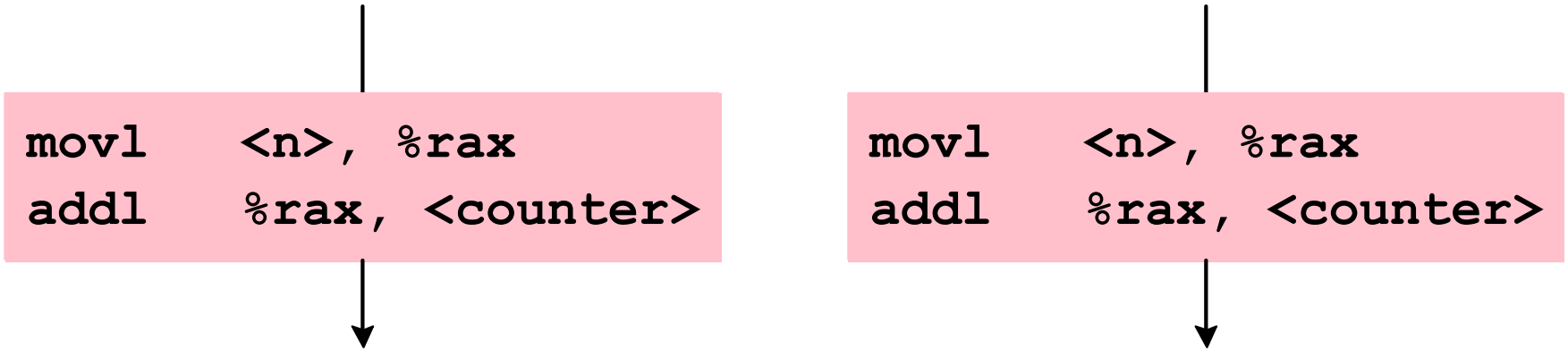
Concurrent Variable Updates

Try compiling with `-O2`



Concurrent Variable Updates

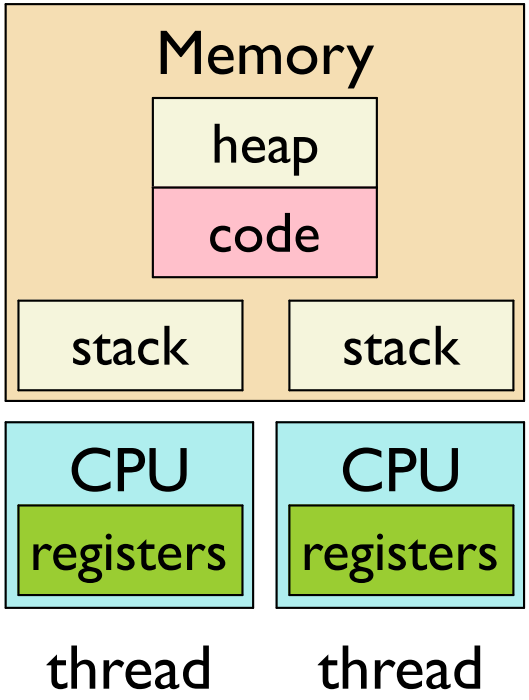
Try compiling with `-O2`



Doesn't work with a multiprocessor

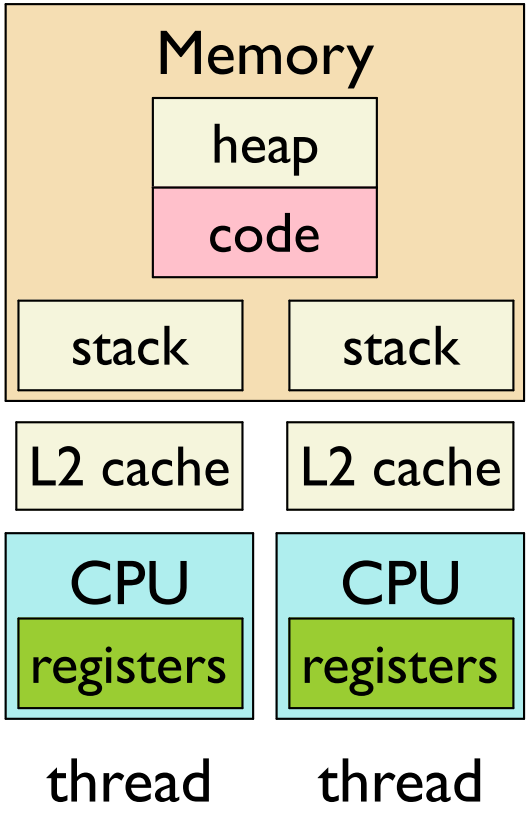
Threads and Processors

Intended illusion:



Threads and Processors

Observable behavior:



Cache coherence is expensive, so the machine just doesn't do it!

... unless you insist

Global Variables and Optimization

Remember that C compilers can make assumptions:

```
long counter = 1;

void count_to(long n) {
  while (counter < n)
   counter++;
}

void wait_for_it() {
  while (counter < 100000)
   ;
}

....
```

Global Variables and Optimization

Remember that C compilers can make assumptions:

```
long counter = 1;

void count_to(long n) {
  while (counter < n)
   counter++;
}

void wait_for_it() {
  while (counter < 100000)
   ;
}

....
```

```
long counter = 0;

void count(long n) {
  long v = counter;
  while (v < n)
    v++;
  counter = v;
}

void wait_for_it() {
  if (counter < 100000)
   while (1)
     ;
}

....
```

Threads and Sharing

Successful sharing among threads requires explicit synchronization

- ✓ Side-steps question of machine-code atomicity
- ✓ Declares need for cache coherence
- ✓ Exposes constraints to compiler

A program with a race condition is practically always a buggy program

Synchronization for Sharing

Several general approaches to sharing:

No sharing — pass messages, instead

✓ No one changes your data while you look at it

✗ Communication must be explicitly scheduled

Transactions — system finds a good ordering

✓ Programmer declares needed atomicity

✗ Requires substantial extra infrastructure

Locks — constrain allowed orders

✓ Almost like declaring atomicity

✗ Declare and using locks correctly is still difficult

Synchronization for Sharing

Several general approaches to sharing:

No sharing — pass messages, instead

✓ No one changes your data while you look at it

✗ Communication must be explicitly scheduled

Transactions — system finds a good ordering

✓ Programmer declares needed atomicity

✗ F

Most common, especially for systems programming

Locks — constrain allowed orders

✓ Almost like declaring atomicity

✗ Declare and using locks correctly is still difficult

Machine-Level Synchronization

`lock cmpxchg`*x source*`,` *dest*

Atomically checks whether **`%rax`** matches *dest* and

- if equal, copies *source* to *dest*, sets **`ZF`**
- if not equal, clears **`ZF`**

Atomicity means that if **`dest`** is a memory address, caches are forced to agree during the instruction

A.K.A. ***compare and swap*** (CAS)

Accessible in **`gcc`** via

`__sync_bool_compare_and_swap(`*addr*`,` *old_val*`,` *new_val*`)`

Machine-Level Synchronization

```
#include "csapp.h"

volatile int counter;

void *count(void *_n) {
  int i, n = *(int *)_n;

  for (i = 0; i < n; i++)
    counter++;

  return NULL;
}

int main(int argc, char **argv) {
  pthread_t a, b;
  int n = 30000;
  Pthread_create(&a, NULL, count, &n);
  Pthread_create(&b, NULL, count, &n);
  Pthread_join(a, NULL);
  Pthread_join(b, NULL);
  printf("result: %d\n", counter);
}
```

Copy

Machine-Level Synchronization

```
#include "csapp.h"

volatile int counter;

void *count(void *_n) {
  int i, n = *(int *)_n;

  for (i = 0; i < n; i++)
    counter++;

  return NULL;
}

int main(int argc, char **argv) {
  pthread_t a, b;
  int n = 30000;
  Pthread_create(&a, NULL, count, &n);
  Pthread_create(&b, NULL, count, &n);
  Pthread_join(a, NULL);
  Pthread_join(b, NULL);
  printf("result: %d\n", counter);
}
```

Copy

`volatile` forces separate load and store on **`counter`**

Result is unspecified

Machine-Level Synchronization

CAS ensures a consistent result:

```
....

    int old_counter;
    do {
      old_counter = counter;
    } while (!__sync_bool_compare_and_swap(&counter,
                                           old_counter,
                                           old_counter+1));

....
```

Copy

CAS is too low-level for most purposes

✗ Failure is a form of busy waiting

✗ Sometimes, multiple values need to change together

Locking for a Critical Region

A ***critical region*** is a section of code that should be running in only one thread at a time

```
for (i = 0; i < n; i++) {
  counter++;
}
```

Locking for a Critical Region

A ***critical region*** is a section of code that should be running in only one t

Only one thread should increment at a time

```
for (i
  counter++;
}
```

Locking for a Critical Region

A ***critical region*** is a section of code that should be running in only one thread at a time

```
for (i = 0; i < n; i++) {
  lock();
  counter++;
  unlock();
}
```

`lock()` returns if currently unlocked, otherwise waits

`unlock()` only if previously `lock()`ed

`lock` and `unlock` are not actual function names...

Locking for a Critical Region

A ***critical region*** is a section of code that should be running in only one thread at a time

```
for (i = 0; i < n; i++) {
  lock();
  count = lookup(name);
  if (count < 10)
    update(name, count + 1);
  unlock();
}
```

`lock()` returns if currently unlocked, otherwise waits

`unlock()` only if previously `lock()`ed

Locking for Specific Data

Locks can be more ***fine-grained***, such as locking specific object instead of a section of code

```
for (i = 0; i < n; i++) {
  lock(locks[i]);
  count = lookup(orders[i], name);
  if (count < 10)
    update(orders[i], name, count + 1);
  unlock(locks[i]);
}
```

Locking as a Signaling Mechanism

Since **`lock()`** waits for another thread's **`unlock()`**, locks can effectively send a "signal" from one thread to another

```
int value = 0;
lock_t ready_lock;

int main() {
  ....
  lock(ready_lock);
  Pthread_create(&th, NULL, go, NULL);
  ....
  value = 1;
  unlock(ready_lock);
  ....
}

void *go(void *ignored) {
  lock(ready_lock);
  .... value ....
}
```

Locking as a Signaling Mechanism

Since **`lock()`** waits for another thread's **`unlock()`**, locks can effectively send a "signal" from one thread to another

```
int value = 0;
lock_t ready_lock;

int main() {
  ....
  lock(ready_lock);
  Pthread_create(&th, NULL, go, NULL);
  ....
  value = 1;
  unlock(ready_lock);
  ....
}

void *go(void *ignored) {
  lock(ready_lock);
  .. value ....
}
```

Cannot proceed until main thread gets to **`unlock`**

Locking as a Signaling Mechanism

If **`unlock()`** doesn't have to be in the **`lock()`** thread, signaling can work the other way, too

```
int value = 0;
lock_t ready_lock;

int main() {
  ....
  lock(ready_lock);
  Pthread_create(&th, NULL, go, NULL);
  lock(ready_lock);
  .... value ....
}

void *go(void *ignored) {
  value = 1;
  unlock(ready_lock);
  ....
}
```

Locking as a Signaling Mechanism

If **`unlock()`** doesn't have to be in the **`lock()`** thread, signaling can work the other way, too

```
int value = 0;
lock_t ready_lock;

int main() {
  ....
  lock(ready_lock);
  Pthread_create(&th, NULL, go, NULL);
  lock(ready_lock);
  .. value ....
}

void *go(void *ignored) {
  value = 1;
  unlock(ready_lock);
  ....
}
```

Cannot proceed until new thread gets to **`unlock`**

Kinds of Locks

Mutex

- **`pthread_mutex_t`**
- **`pthread_mutex_init(`***mutex*`,  `*attr*`)`**
- **`pthread_mutex_lock(`***mutex*`)`**
- **`pthread_mutex_unlock(`***mutex*`)`**

`...lock()` and balancing **`...unlock()`** must be same thread

Semaphore

- **`sem_t`**
- **`Sem_init(`***sem*`,  `*ps_share*`,  `*value*`)`**
- **`P(`***sem*`)`** = **`lock()`**, but with a counter
- **`V(`***sem*`)`** = **`unlock()`**, with the counter

`P()` and balancing **`V()`** threads can be different

Kinds of Locks

Mutex

- **`pthread_mutex_t`**
- **`pthread_mutex_init`**(*mutex*, *attr*)
- **`pthread_mutex_lock`**(*mutex*)
- **`pthr`**

`...lock`

Sometimes, we create a semaphore and name it **`mutex`**, because it's used that way

Semaphore

- **`sem_t`**
- **`Sem_init`**(*sem*, *ps_share*, *value*)
- **`P`**(*sem*) = **`lock()`**, but with a counter
- **`V`**(*sem*) = **`unlock()`**, with the counter

`P()` and balancing **`V()`** threads can be different

Semaphores

```
#include "csapp.h"

void Sem_init(sem_t *sem, int ps_share, unsigned int value);
void P(sem_t *sem);
void V(sem_t *sem);
void Sem_destroy(sem_t *sem);
```

`Sem_init` creates **`sem`** with initial count **`value`**

`1` as **`value`** for a mutex

0 as **`ps_share`**

`P` waits until **`sem`** has a non-0 count, then decrements

corresponds to **`lock`**, also called "wait"

`V` increments **`sem`**'s count

corresponds to **`unlock`**, also called "post"

`Sem_destroy` destroys **`sem`**

Semaphore Example

```
....
sem_t count_sem;

void *count(void *_n) {
  int i, n = *(int *)_n;

  for (i = 0; i < n; i++) {
    P(&count_sem);
    counter++;
    V(&count_sem);
  }

  return NULL;
}

int main(int argc, char **argv) {
  ....
  Sem_init(&count_sem, 0, 1);
  Pthread_create(&a, NULL, count, &n);
  Pthread_create(&b, NULL, count, &n);
  .....
}
```

Copy

Semaphores for Echo

t_echo.c

```
....
sem_t ready_sem, count_sem;

int main(int argc, char **argv) {
  ....
  Sem_init(&count_sem, 0, 1);
  Sem_init(&ready_sem, 0, 0);
  ....
     Pthread_create(&th, NULL, echo, &connfd);
     P(&ready_sem);
  ....
}

void *echo(void *connfd_p) {
  ....
  V(&ready_sem);
  ....
    P(&count_sem);
    counter += n;
    V(&count_sem);
  ....
}
```

Semaphores as Per-Object Locks

counter.c

```
typedef struct {
  int val;
  sem_t sem;
} counter;

counter *make_counter() {
  counter *c = malloc(sizeof(counter));
  c->val = 0;
  Sem_init(&c->sem, 0, 1);
  return c;
}

void counter_add(counter *c, int amt) {
  P(&c->sem);
  c->val += amt;
  V(&c->sem);
}
....
void destroy_counter(counter *c) {
  Sem_destroy(&c->sem);
  free(c);
}
```

Limiting Echo Threads

Our echo server runs N threads for N concurrent clients

Using a fixed number of threads, instead:

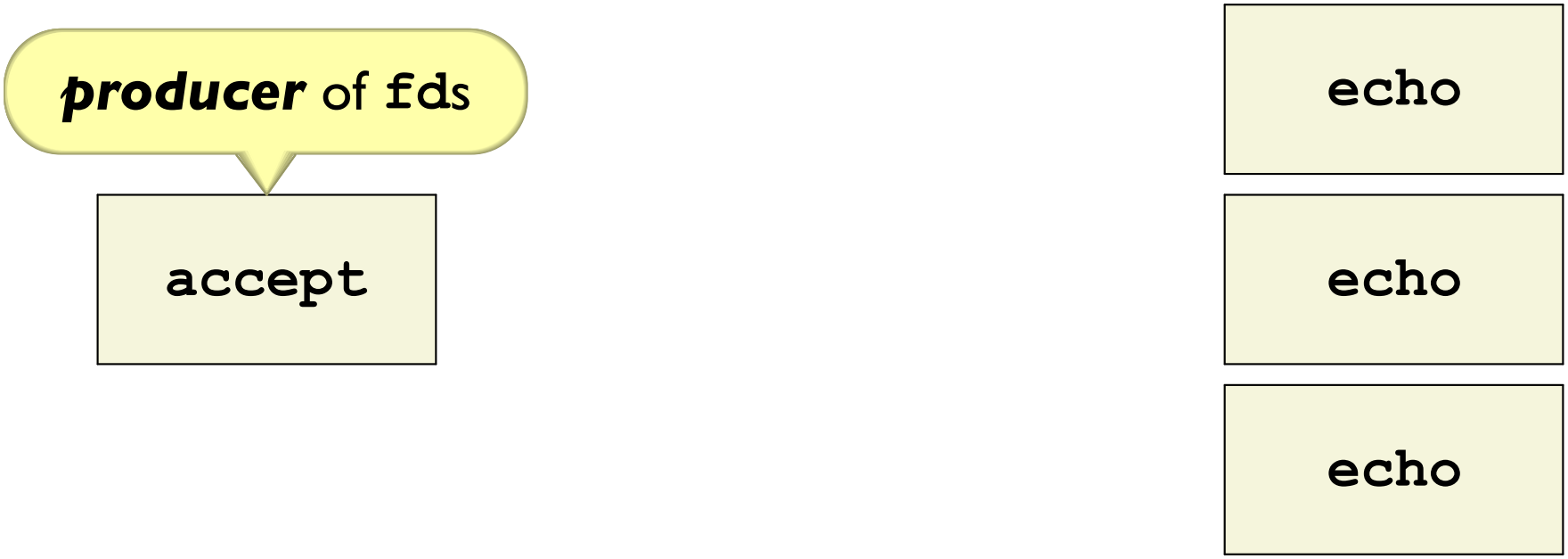
- ✓ limits the server's resource consumption
- ✓ lowers cost of handling each connection

Limiting Echo Threads

Our echo server runs N threads for N concurrent clients

Using a fixed number of threads, instead:

- ✓ limits the server's resource consumption
- ✓ lowers cost of handling each connection

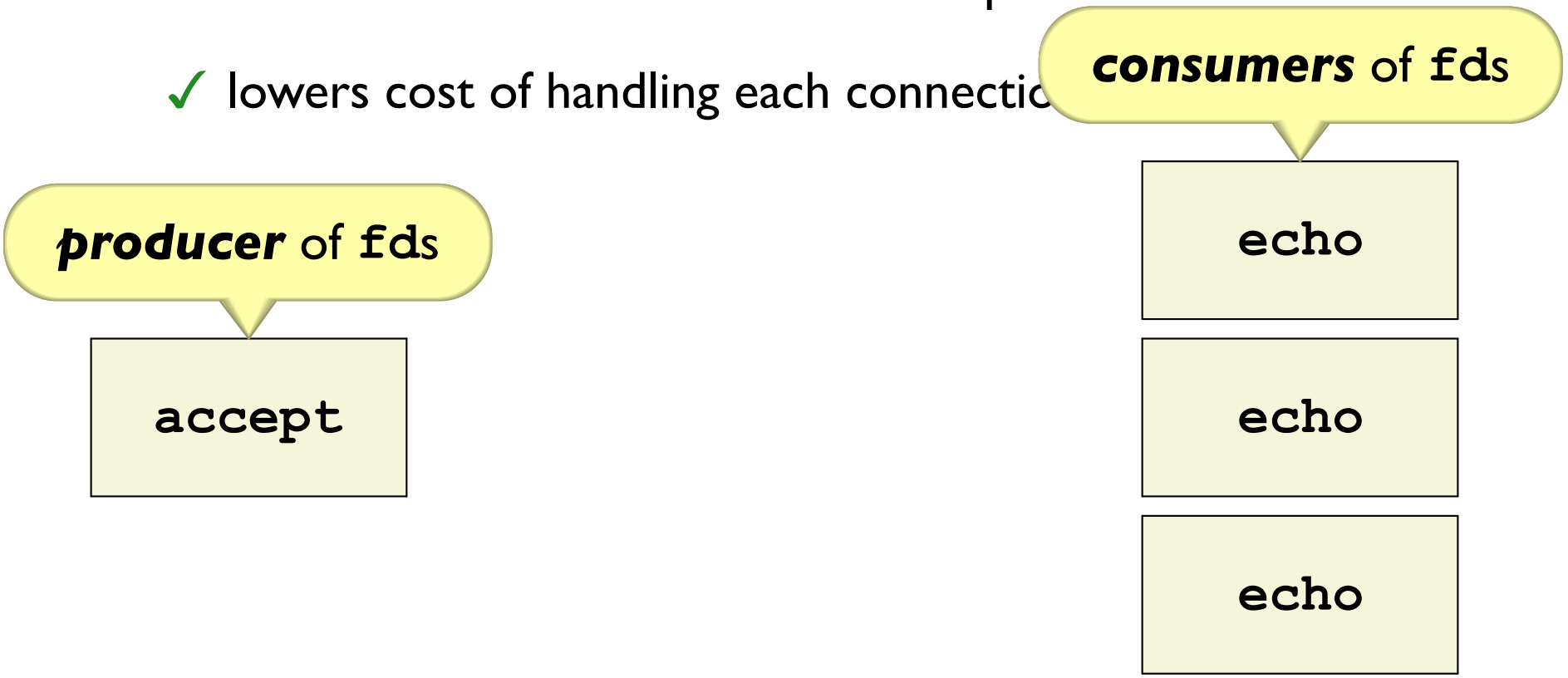


Limiting Echo Threads

Our echo server runs N threads for N concurrent clients

Using a fixed number of threads, instead:

- ✓ limits the server's resource consumption
- ✓ lowers cost of handling each connectio

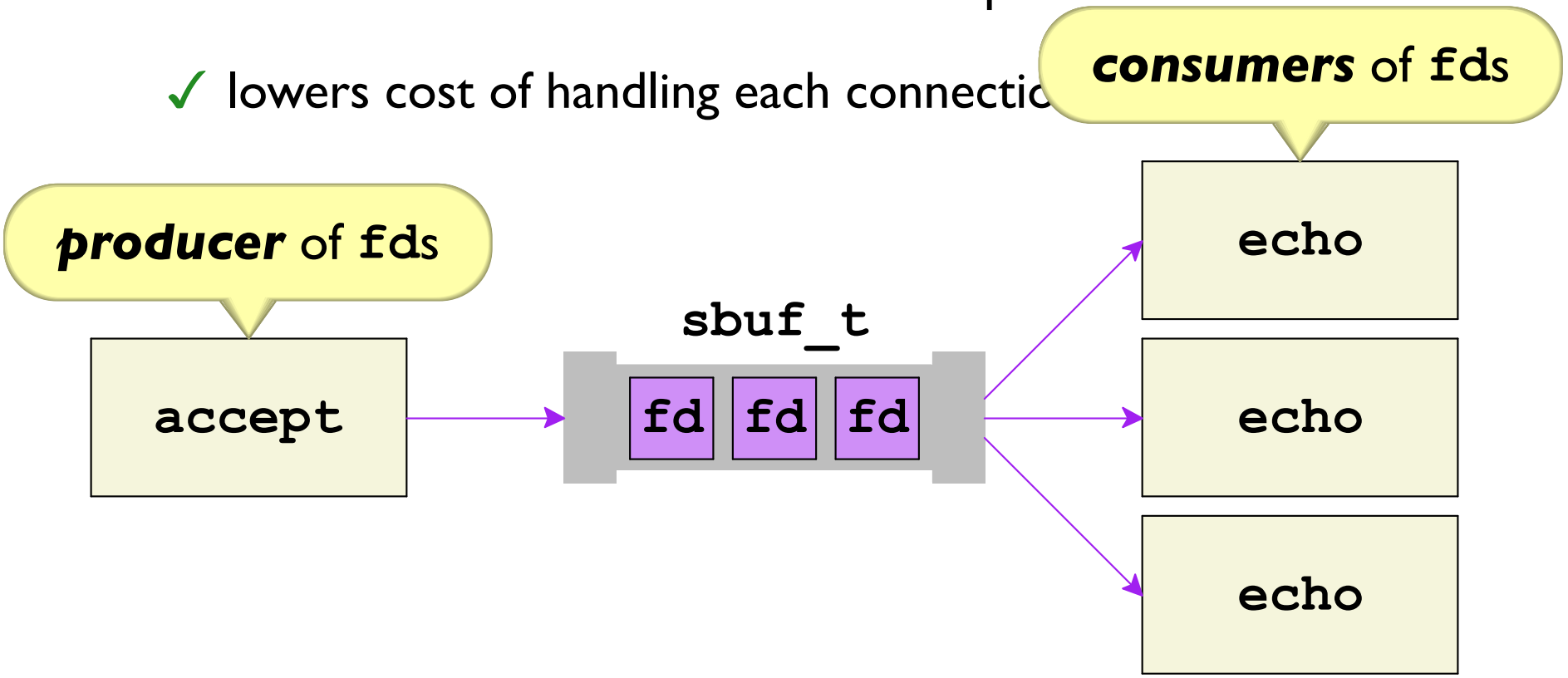


Limiting Echo Threads

Our echo server runs N threads for N concurrent clients

Using a fixed number of threads, instead:

- ✓ limits the server's resource consumption
- ✓ lowers cost of handling each connectio

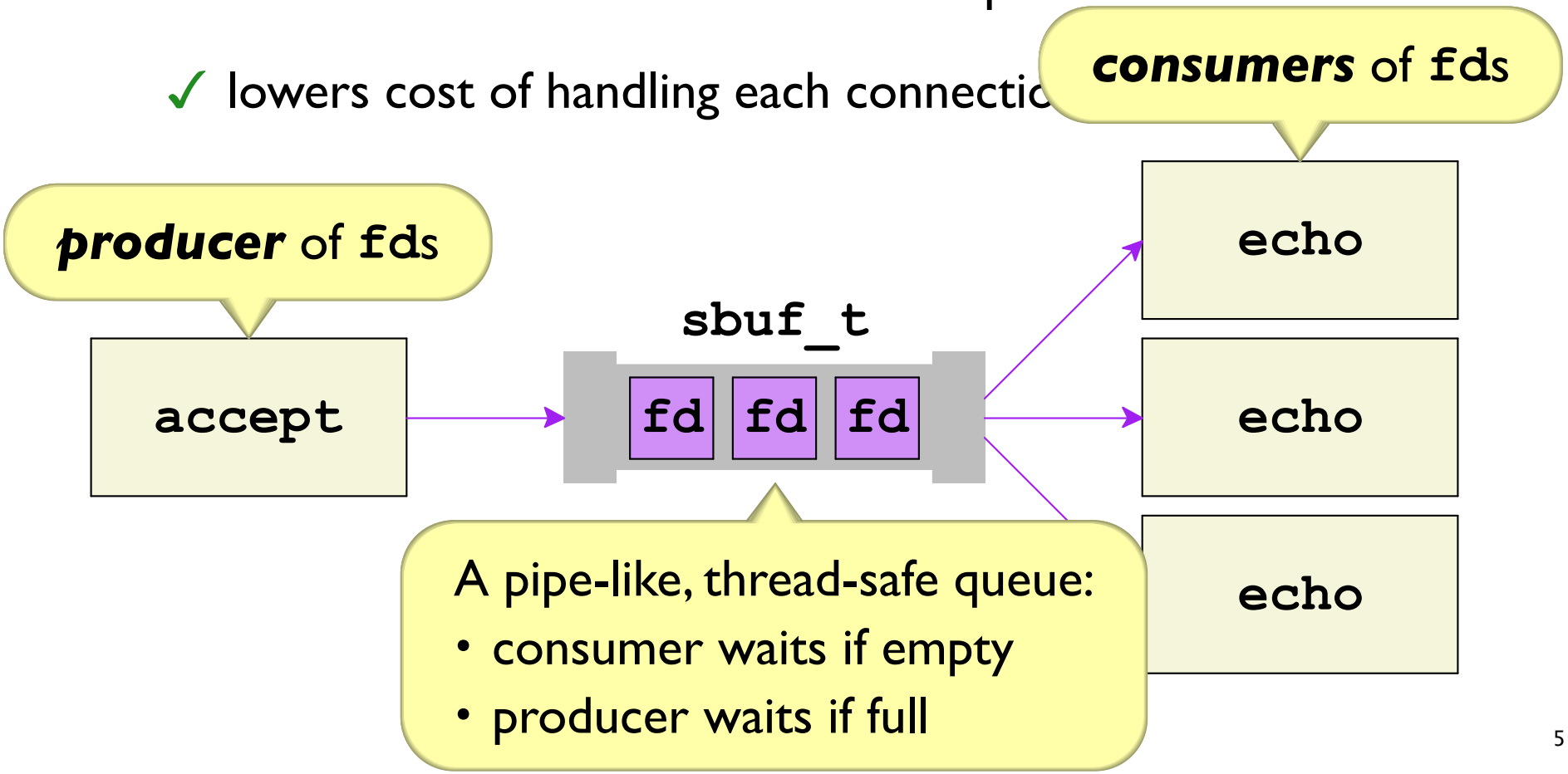


Limiting Echo Threads

Our echo server runs N threads for N concurrent clients

Using a fixed number of threads, instead:

- ✓ limits the server's resource consumption
- ✓ lowers cost of handling each connectio



Implementing a Limited Queue with Semaphores

Strategy: use semaphore count to reflect availability

- **`sbuf_insert`** (for producer) — count is available slots
- **`sbuf_remove`** (for consumer) — count is available values

⇒ two counter semaphores, plus one as a mutex

Implementing a Limited Queue with Semaphores

sbuf.h

```
typedef struct {
    int *buf;       /* Buffer array */
    int n;          /* Maximum number of slots */
    int front;      /* buf[(front+1)%n] is first item */
    int rear;       /* buf[rear%n] is last item */
    sem_t mutex;    /* Protects accesses to buf */
    sem_t slots;    /* Counts available slots */
    sem_t items;    /* Counts available items */
} sbuf_t;
```

Implementing a Limited Queue with Semaphores

sbuf.c

```
....

void sbuf_init(sbuf_t *sp, int n) {
  sp->buf = Calloc(n, sizeof(int));
  sp->n = n;                      /* max of n items */
  sp->front = sp->rear = 0;       /* empty iff front == rear */
  Sem_init(&sp->mutex, 0, 1); /* for locking */
  Sem_init(&sp->slots, 0, n); /* initially n empty slots */
  Sem_init(&sp->items, 0, 0); /* initially zero data items */
}

....
```

Implementing a Limited Queue with Semaphores

sbuf.c

```
....

void sbuf_insert(sbuf_t *sp, int item) {
  P(&sp->slots);   /* wait for available slot */
  P(&sp->mutex);   /* lock */
  sp->buf[(++sp->rear)%(sp->n)] = item;
  V(&sp->mutex);   /* unlock */
  V(&sp->items);   /* announce available item */
}

....
```

Implementing a Limited Queue with Semaphores

sbuf.c

```
....

int sbuf_remove(sbuf_t *sp) {
  int item;
  P(&sp->items);  /* wait for available item */
  P(&sp->mutex);  /* lock */
  item = sp->buf[(++sp->front)%(sp->n)];
  V(&sp->mutex);  /* unlock */
  V(&sp->slots);  /* announce available slot */
  return item;
}

....
```

Producer–Consumer Echo Server

pc_echo.c

```
 ....
 sbuf_t connfds;

 int main(int argc, char **argv) {
   ....
   sbuf_init(&connfds, SBUF_SIZE);

   for (i = 0; i < NUM_THREADS; i++) {
     Pthread_create(&th, NULL, echo, NULL);
     Pthread_detach(th);
   }
   ....
     connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
     sbuf_insert(&connfds, connfd);
   ....
}
....
```

Producer–Consumer Echo Server

pc_echo.c

```
....

void *echo(void *ignored) {
  ....
  while (1) {
    connfd = sbuf_remove(&connfds);

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
      printf("server received %ld bytes\n", n);
      Rio_writen(connfd, buf, n);
    }

    Close(connfd);
  }
}
```

Threads and `errno`

Suppose one thread is running

```
fd = open(...);
if (fd < 0)
  fprintf(stderr, "%d", errno);
```

and another is running

```
fd = connect(...);
if (fd < 0)
  fprintf(stderr, "%d", errno);
```

Can the **`open`** thread get the **`errno`** value for **`connect`**?

No, `errno` is ***thread-local*** *Whew!*

Thread-Safe Functions

Standard library functions are generally ***thread-safe***

OK in multiple threads:

- **`malloc`** and **`free`**
- **`read`** on the same file descriptor
- **`fread`** on the same file handle
- **`getaddrinfo`** to fill different records

Not OK in multiple threads:

- **`getenv`** when **`setenv`** might be called
- **`rio_readnb`** on a specific buffer

Concurrency vs. Parallelism

Concurrency = multiple control flows overlapping in time

possibly on a uniprocessor

reduces ***latency***

Parallelism = multiple control flows at the same time

requires a multiprocessor

can improve ***throughput***

parallelism $\Rightarrow$ concurrency concurrency $\not\Rightarrow$ parallelism