DCS216 Operating Systems

Lecture 09
Inter-process Communication (3)

Mar 25th, 2024

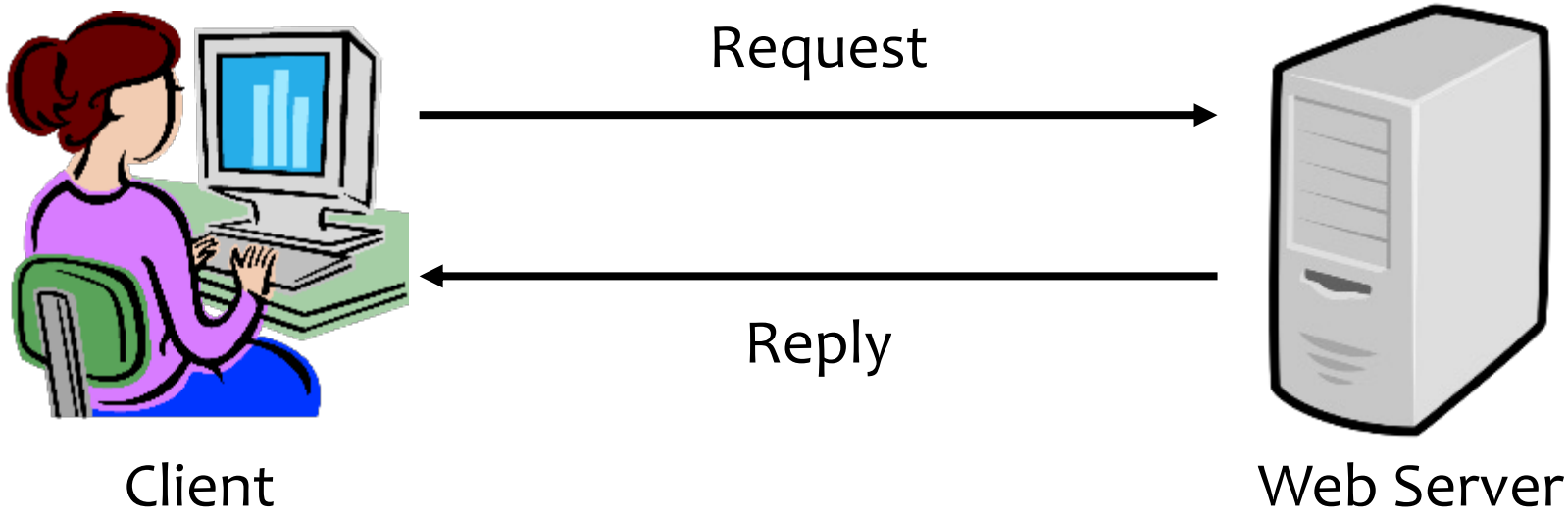
Instructor: Xiaoxi Zhang

Sun Yat-sen University

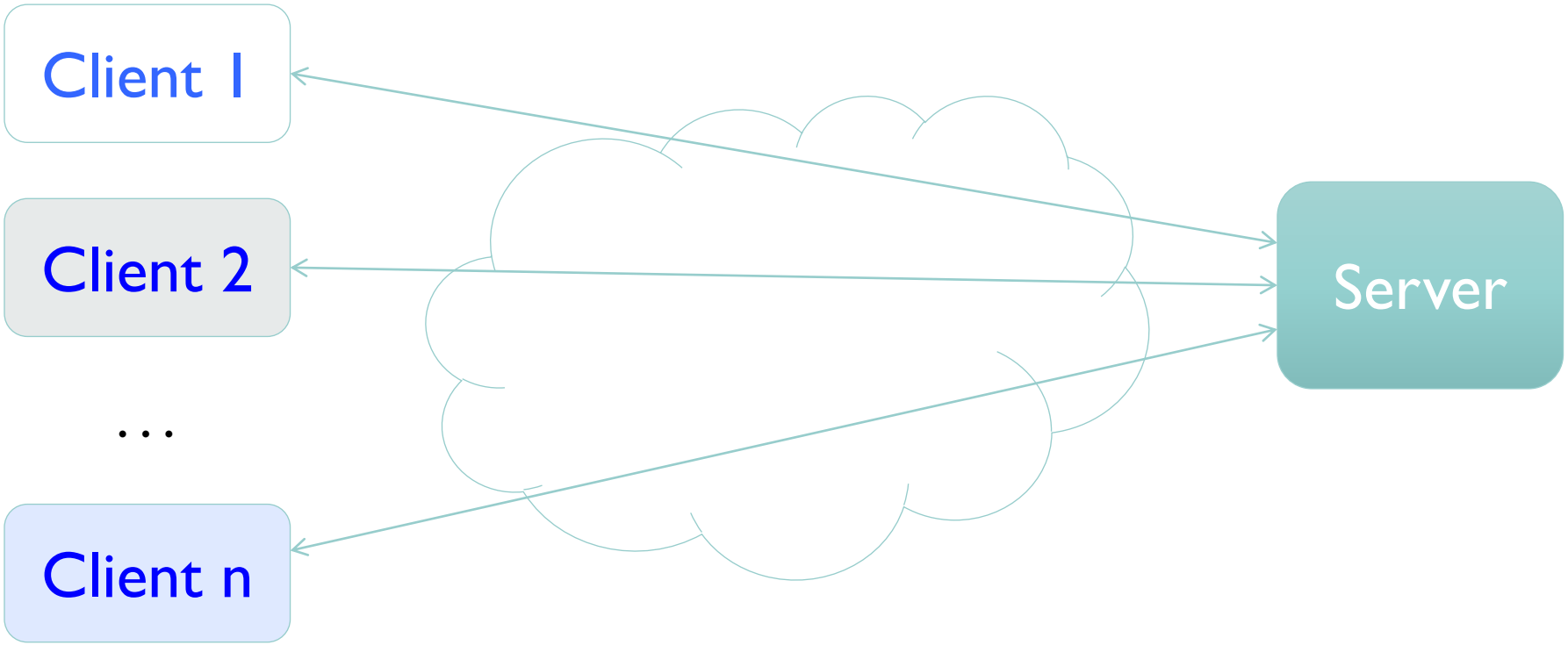
Content

- **Content**
  - Overview
  - Shared-Memory Systems
  - Message-Passing Systems
  - Pipes
  - Communication in Client-Server Systems
    - Sockets
    - Remote Procedure Calls (RPCs)

- **Web Server**

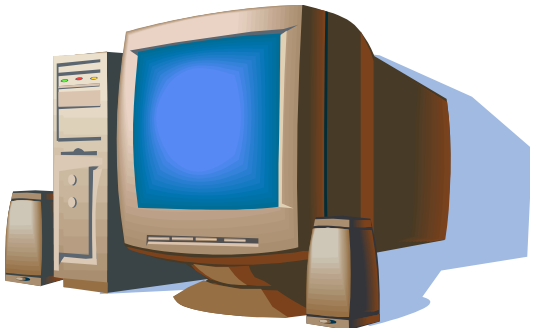


- **Client-Server Protocols: Cross-Network IPC**
  - Many clients accessing a common server
  - File servers, www, FTP, databases, ...



Communication in Client-Server Systems

- **Client is "sometimes on"**
  - Sends the server requests for services when interested
  - E.g., Web **browser** on laptop/phone
  - Doesn't communicate directly with other clients
  - Needs to know server's address
- **Server is "always on"**
  - Services requests from many clients
  - E.g., Web **server** for www.google.com
  - Doesn't initiate contact with clients
  - Needs a fixed, well-known address

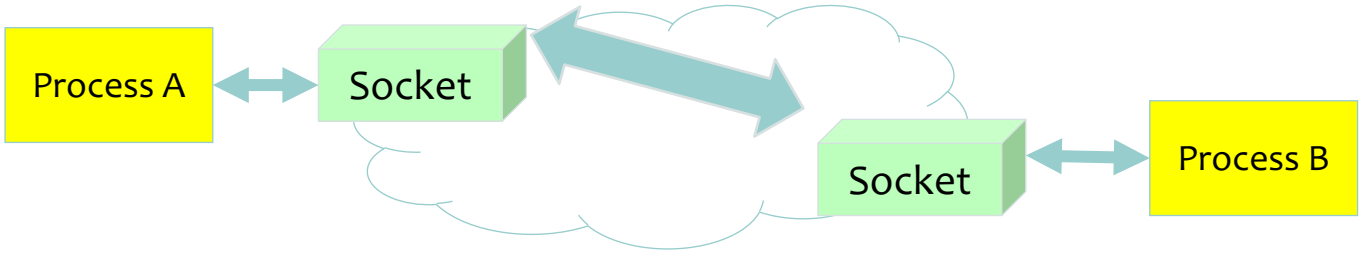


- **What is a Network Connection?**
  - Bidirectional stream of bytes between two processes on possibly different machines
    - For now, we are discussing "TCP Connections"
  - Abstractly, a connection between two endpoints A and B consists of:
    - A **queue** (bounded buffer) for data sent from A to B
    - A **queue** (bounded buffer) for data sent from B to A

Sockets (套接字)

- The Socket Abstraction: Endpoint for Communication
  - Queues to temporarily hold results
- Key idea: Communication across the network looks like File I/O
  - Remember: **Everything in UNIX is a FILE!**

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




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

- Connection:
  - Two Sockets Connected Over the network → IPC over network
  - How to `open()`?
  - What is the namespace?
  - How are they connected in time?

Sockets

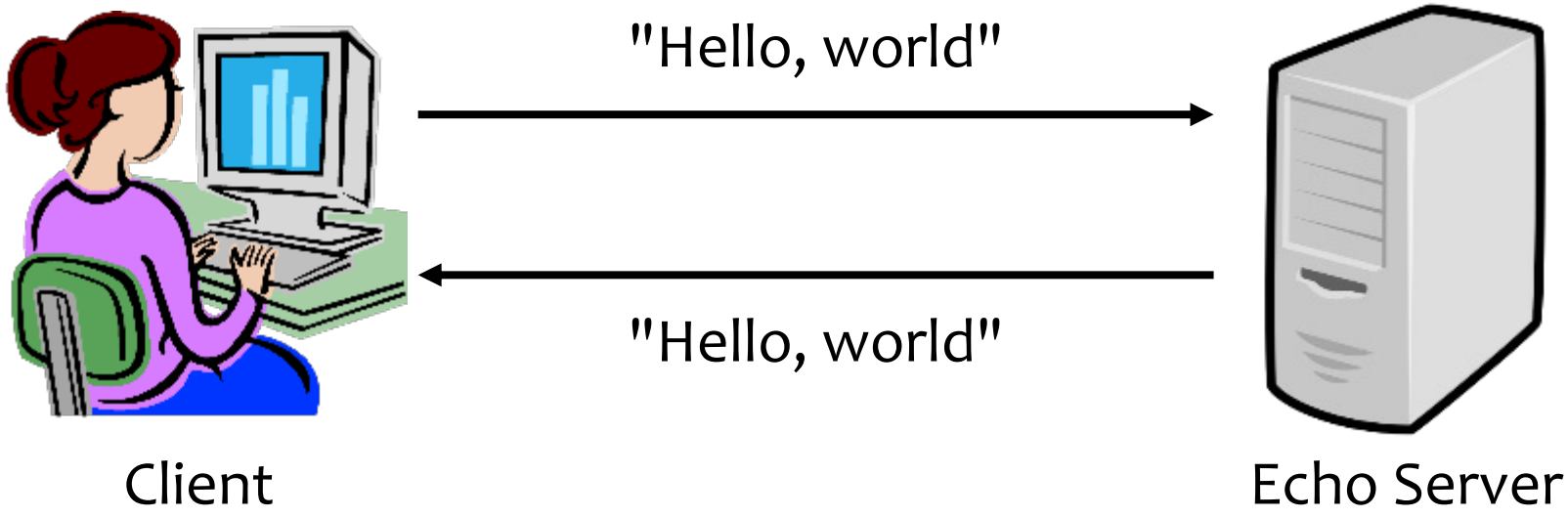
- **Socket: An abstraction for one endpoint of a network connection**
  - Another mechanism for Inter-Process Communication (IPC)
  - Most OSes provide socket API, even if they don't comply with the rest of UNIX I/O
  - Standardized by POSIX
  - Same abstraction for any kind of network
    - Local (within the same machine, e.g., UNIX Domain Socket)
    - The Internet (TCP/IP, UDP/IP)
    - Things that no one uses anymore (OSI, Appletalk, IPX, ...)

Sockets: More Details

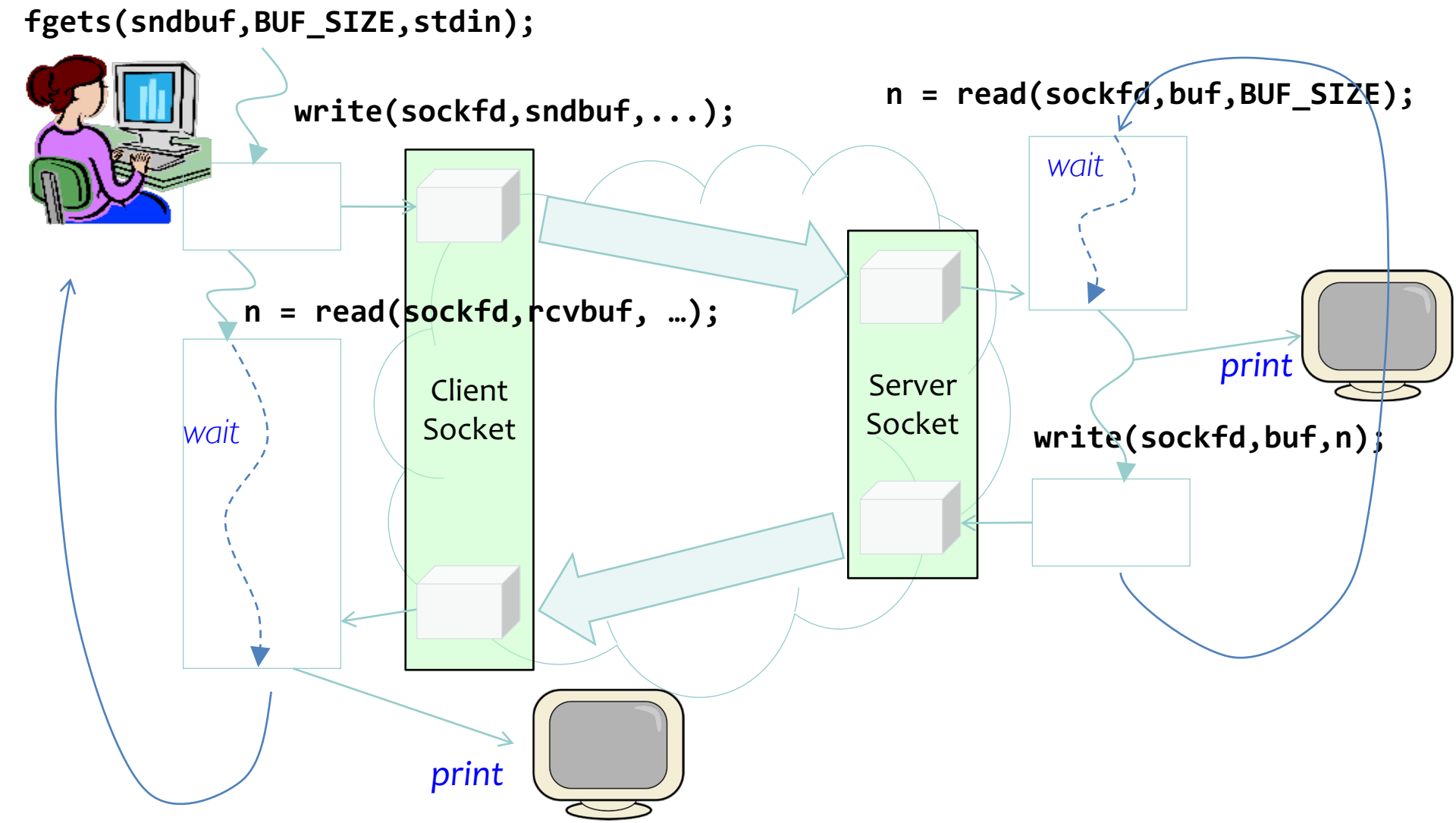
- Looks just like a file with a **file descriptor**
  - Corresponds to a network connection (two queues)
  - `write()` adds to output queue (queue of data destined for other side)
  - `read()` removes from input queue (queue of data destined for this side)
  - Some operations do not work, e.g., `leek()`.

- How can we use sockets to support real applications?
  - A bidirectional byte stream isn't useful on its own
  - May need messaging facility to partition stream into chunks
  - May need **RPC** facility to translate one environment to another and provide the abstraction of a function call over the network

Simple Socket Example: Echo Server



Simple Socket Example: Echo Server



Simple Socket Example: Echo Server

```
void client(int sockfd) {
    char sndbuf[BUF_SIZE];
    char rcvbuf[BUF_SIZE];

    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);          // Prompt
        write(sockfd, sndbuf, strlen(sndbuf)+1); // Send

        int n = read(sockfd, rcvbuf, BUF_SIZE);  // Receive
        write(STDOUT_FILENO, rcvbuf, n);         // Echo
    }
}
```

```
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);      // Echo
    }
}
```

Sockets

- **What Assumptions are we making?**
  - Reliable
    - Write to a file → Read it back. Nothing is lost.
    - Write to a (TCP) socket → Read from the other side, same.
    - Like pipes
  - In Order (sequential stream)
    - Write X then write Y → Read gets X then read gets Y

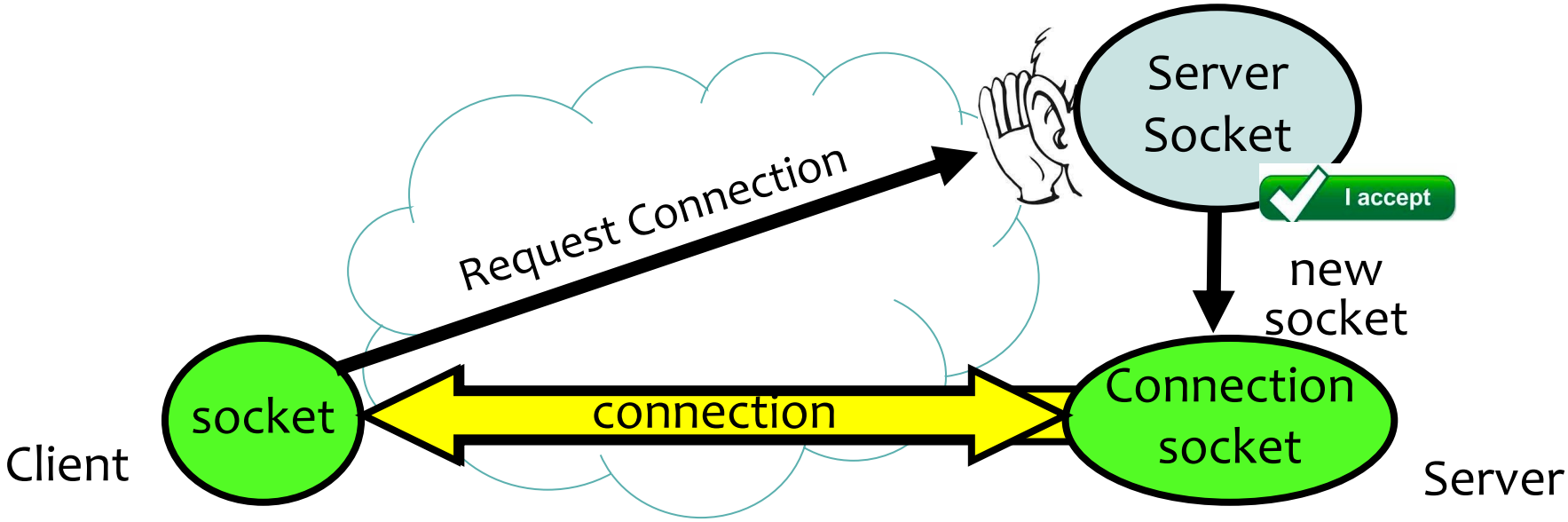
Socket Creation

- **File systems** provide a collection of permanent objects
  - Processes perform `open()`, `read()`, `write()`, `close()` on them
  - Files exist independently of processes
  - Easy to name what file to `open()`
- **Pipes:** one-way communication between processes on the same host
  - Single queue
  - Created transiently by a call to `pipe()`
  - Passed from parent to children (file descriptors inherited from parent)
- **Sockets**: two-way communication between processes on the same host, or on different hosts
  - Two queues (one in each direction)
  - Processes can be on separate hosts: no common ancestor
  - How do we name the objects we are opening?
  - How do these completely independent processes know that the other wants to "talk" to them?

Namespaces for Communication over IP

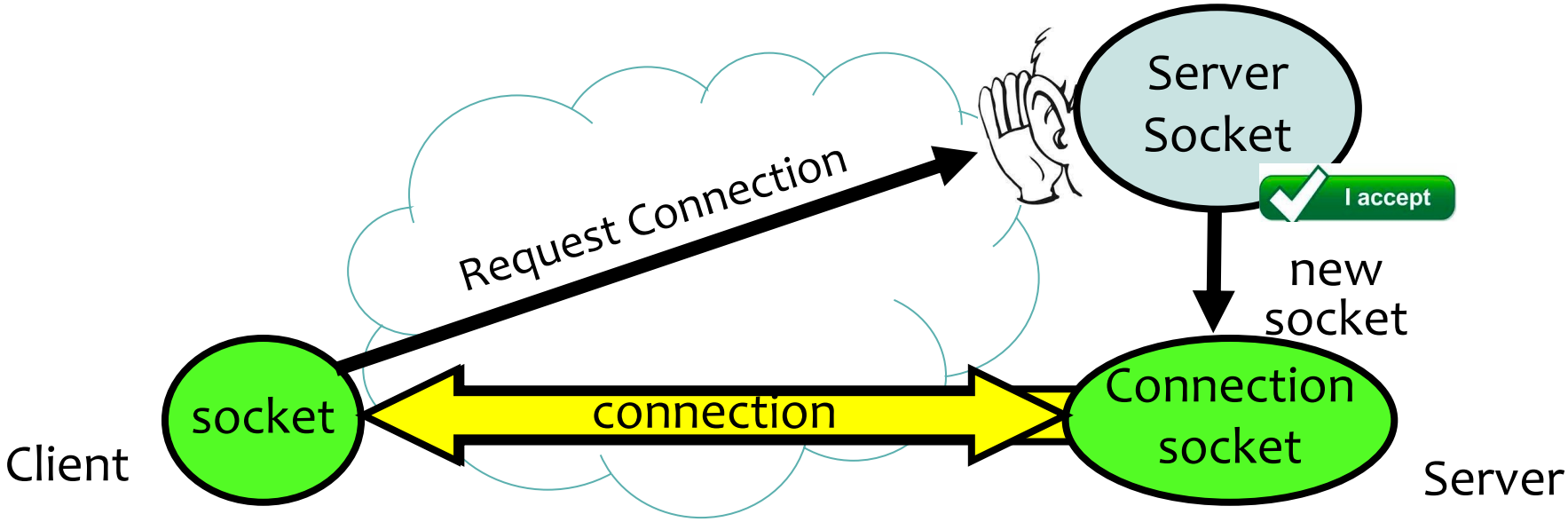
- Hostname
  - E.g., www.sysu.edu.cn
- IP Address
  - 202.116.64.8 (IPv4, 32-bit Integer)
  - 2001:250:3002:10::8 (IPv6, 128-bit Integer)
- Port Number
  - 0 ~ 1023 are "well known" or "system" ports
    - Superuser privileges to bind to one of these ports
  - 1024 ~ 49151 are "registered" ports
    - Assigned by IANA for specific services
  - 49152 ~ 65535 ($2^{15}+2^{14}$ to $2^{16}-1$) are "dynamic" or "private" ports
    - Automatically allocated as "ephemeral ports"

- **Socket Connection Setup over TCP/IP**
  - Special kind of socket: **server socket**
    - Has file descriptor
    - Can't read or write
  - Two Operations:
    - `` `listen()` ``
    - `` `accept()` ``

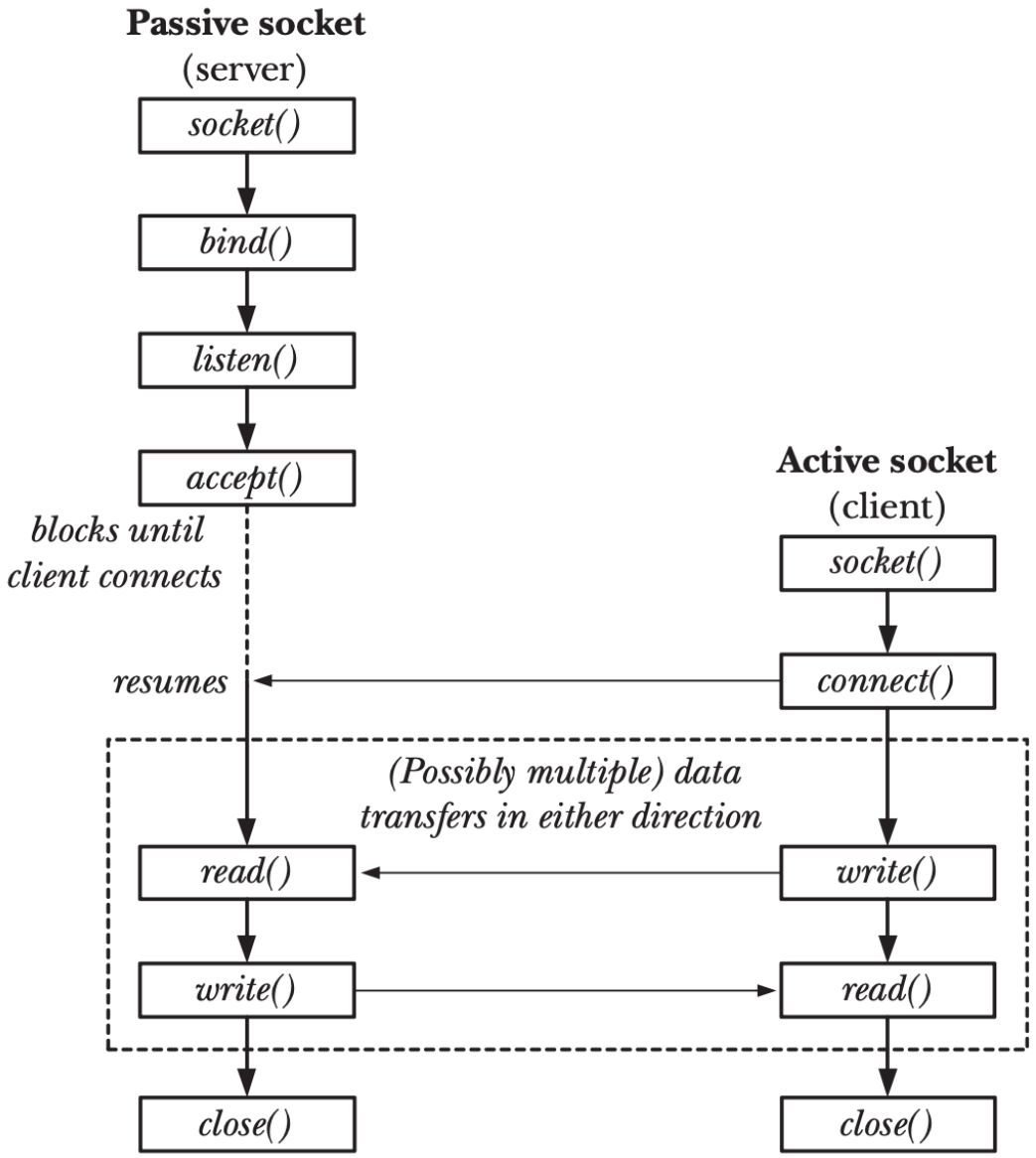


Socket Connection Setup over TCP/IP

- 5-Tuple identifies each connection
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol



Socket Connection Setup over TCP/IP



Socket Connection Setup over TCP/IP

```c
/* server.c */
void server(int sockfd) {
    char buf[BUF_SIZE];
    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
int main() {
    int servfd, sockfd;
    struct sockaddr_in addr;
    // Create socket fd
    servfd = socket(AF_INET, SOCK_STREAM, 0);
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(PORT);
    // Bind sockfd to port
    bind(servfd, (struct sockaddr *)&addr,
         sizeof(addr));
    // Listen
    listen(servfd, 10);
    while (1) {
        // Accept a connection
        sockfd = accept(servfd, NULL, NULL);
        server(sockfd);
        close(sockfd);
    }
    close(servfd);
    return 0;
```

```c
/* client.c */
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);
        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
int main() {
    int sockfd;
    struct sockaddr_in addr;
    // Create socket fd
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    addr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1",
              &addr.sin_addr);
    addr.sin_port = htons(PORT);
    // Connect to server
    connect(sockfd, (struct sockaddr *)&addr,
            sizeof(addr));

    client(sockfd);

    close(sockfd);

    return 0;
}
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

```
$ ./server
```

```
$ ./client
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

```
$ ./server
```

```
$ ./client
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

Server blocks at `read()`, waiting for client write

```
$ ./server
```

```
$ ./client
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

Server blocks at `read()`, waiting for client write

```
$ ./server
```

```
$ ./client
Hello          # <-- Client input at stdin
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

Server blocks at `read()`, waiting for client write

```
$ ./server
```

```
$ ./client
Hello          # <-- Client input at stdin
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

Server unblocks

```
$ ./server
Hello
```

```
$ ./client
Hello          # <-- Client input at stdin
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

Server unblocks

Client blocks at `read()`, waiting for server write

```
$ ./server
Hello
```

```
$ ./client
Hello          # <-- Client input at stdin
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

Client blocks at `read()`, waiting for server write

```
$ ./server
Hello
```

```
$ ./client
Hello          # <-- Client input at stdin
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);      // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

Client blocks at `read()`, waiting for server write

```
$ ./server
Hello
```

```
$ ./client
Hello          # <-- Client input at stdin
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);      // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

Client successfully `read()` from sockfd

```
$ ./server
Hello
```

```
$ ./client
Hello          # <-- Client input at stdin
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);      // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

Client `write()` to stdout

```
$ ./server
Hello
```

```
$ ./client
Hello          # <-- Client input at stdin
Hello          # --> Client output at stdout
```

Socket Connection

```c
// server.c
void server(int sockfd) {
    char buf[BUF_SIZE];

    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
```

```c
// client.c
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);

        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
```

Client and server interact via **sockfd**, just like a regular file descriptor.

```
$ ./server
Hello
Goodbye
```

```
$ ./client
Hello          # <-- Client input at stdin
Hello          # --> Client output at stdout
Goodbye        # <-- Client input at stdin
Goodbye        # --> Client output at stdout
```

Sockets

- A socket is created using the `socket()` system call, which returns a file descriptor used to refer to the socket in subsequent calls:
  - `fd = int socket(int domain, int type, int protocol);`

Domains

- Sockets exist in a communication domain, which determines the method of identifying a socket and the range of communication
- In UNIX systems, 3 types of domains are commonly supported:
  - `AF_UNIX`: The **UNIX** domain, also called the LOCAL domain, allows communication between processes on the same host
  - `AF_INET`: The **IPv4** domain allows communication between processes between hosts connected via IPv4 network
  - `AF_INET6`: The **IPv6** domain allows communication between processes between hosts connected via IPv6 network

- **Sockets**
  - A socket is created using the `socket()` system call, which returns a file descriptor used to refer to the socket in subsequent calls:
    - `fd = int socket(int domain, int type, int protocol);`
- **Domains**
  - Sockets exist in a communication domain, which determines the method of identifying a socket and the range of communication
  - In UNIX systems, 3 types of domains are commonly supported:
    - `AF_UNIX`
    - `AF_INET`
    - `AF_INET6`

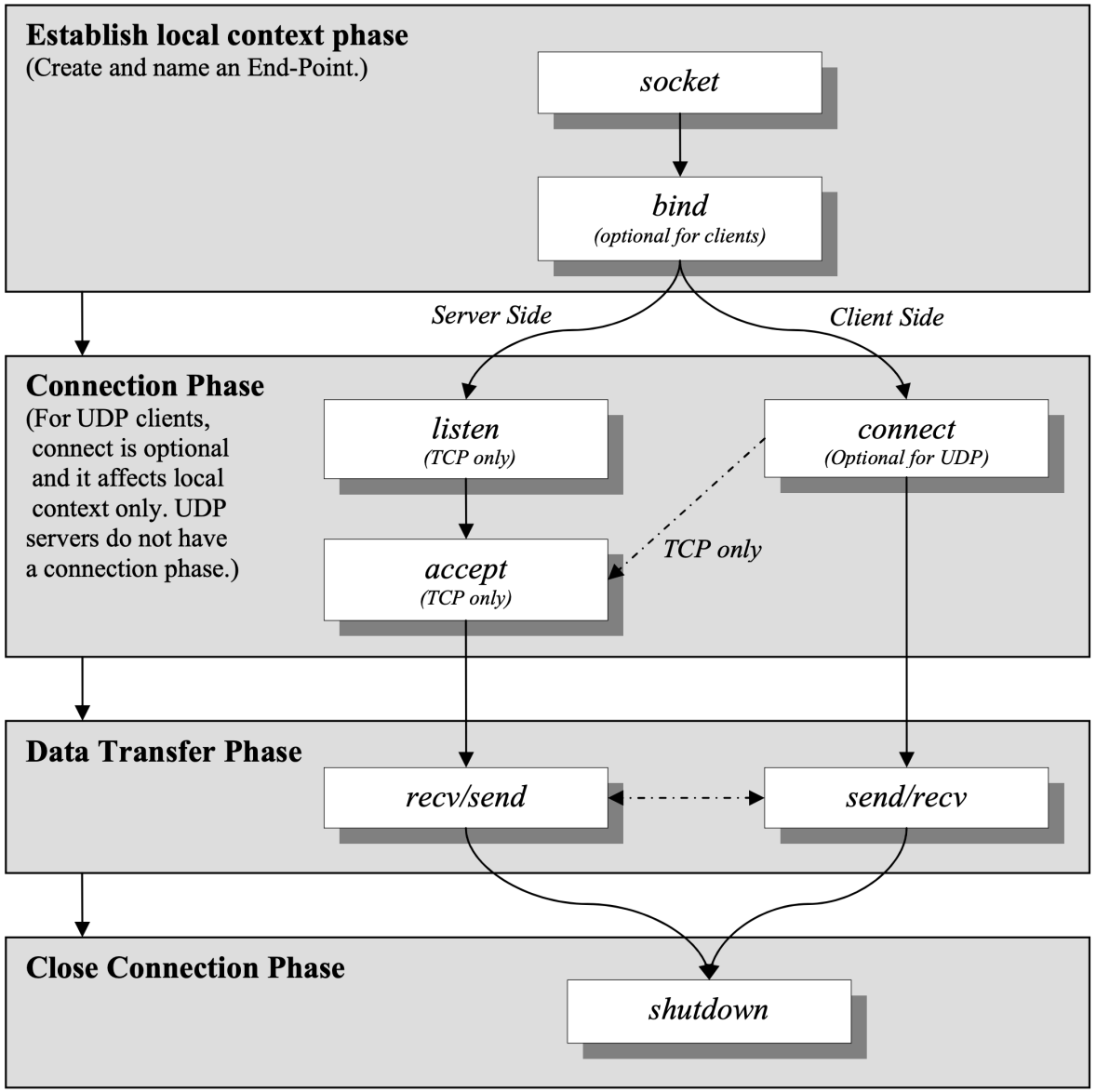
Domain	Communication performed	Communication between applications	Address format	Address structure
AF_UNIX	within kernel	on same host	pathname	*sockaddr_un*
AF_INET	via IPv4	on hosts connected via an IPv4 network	32-bit IPv4 address + 16-bit port number	*sockaddr_in*
AF_INET6	via IPv6	on hosts connected via an IPv6 network	128-bit IPv6 address + 16-bit port number	*sockaddr_in6*

- **Sockets**
  - A socket is created using the `socket()` system call, which returns a file descriptor used to refer to the socket in subsequent calls:
    - `fd = int socket(int domain, int type, int protocol);`
- **Socket Types**
  - The `type` argument determines the type of the socket, which further determines the communication characteristics. There are two commonly used socket types: stream and datagram
    - `SOCK_STREAM`: provides a reliable, bidirectional, byte-stream communication channel. E.g., TCP.
    - `SOCK_DGRAM`: provides an unreliable, connection-less, message-oriented communication channel. E.g., UDP.

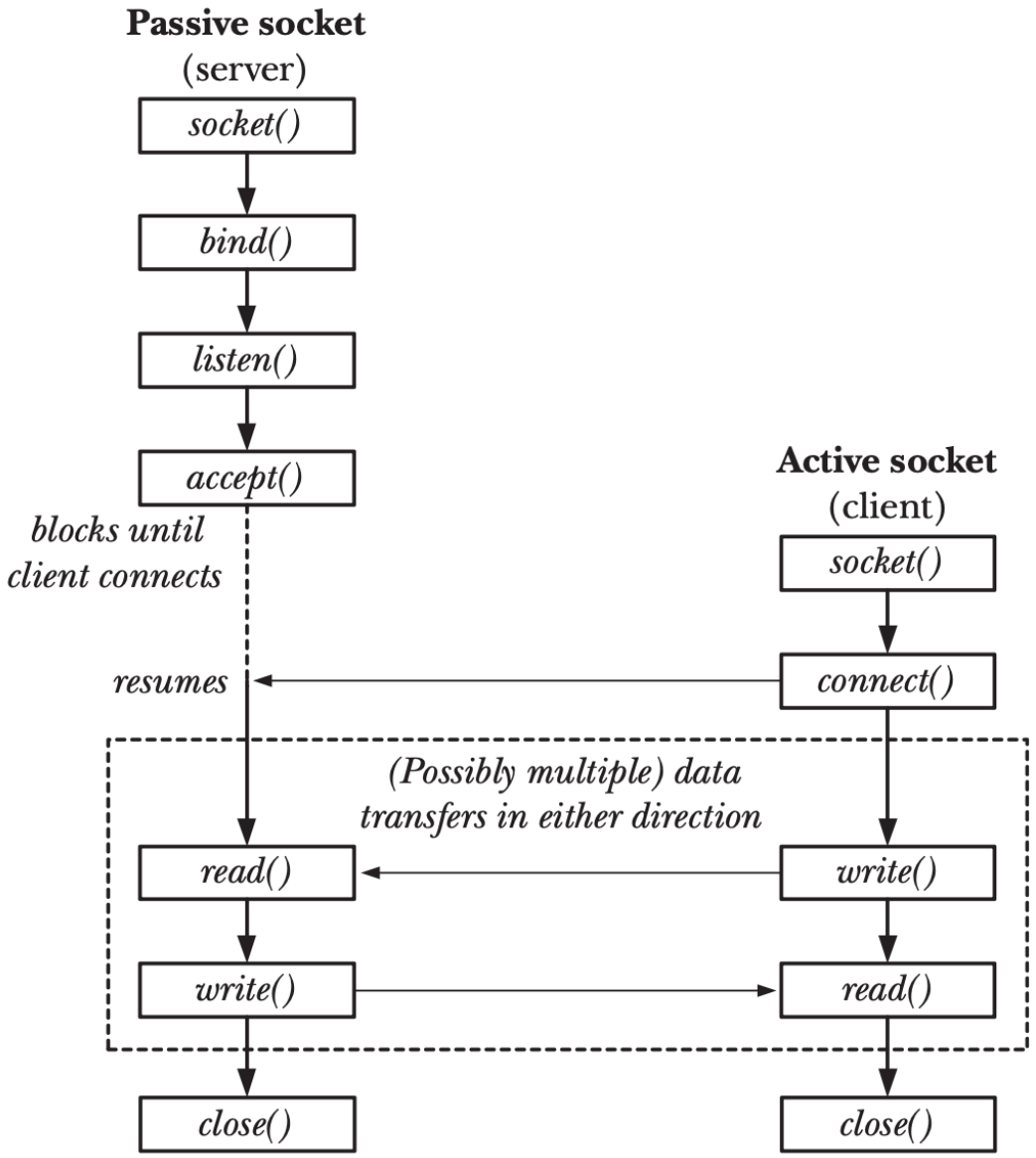
Socket system calls

- The key socket related system calls are:
  - `socket()`: creates a new socket
  - `bind()`: binds a socket to an address
  - `listen()`: allows a stream socket to accept incoming connections from other sockets
  - `accept()`: accepts a connection from a peer application on a listening stream socket
  - `connect()`: establishes a connection with another socket
  - I/O operations:
    - Conventional I/O: `read()`, `write()`
    - Socket-specific I/O: `send()`, `recv()`
  - `close()`: terminates the stream socket connection

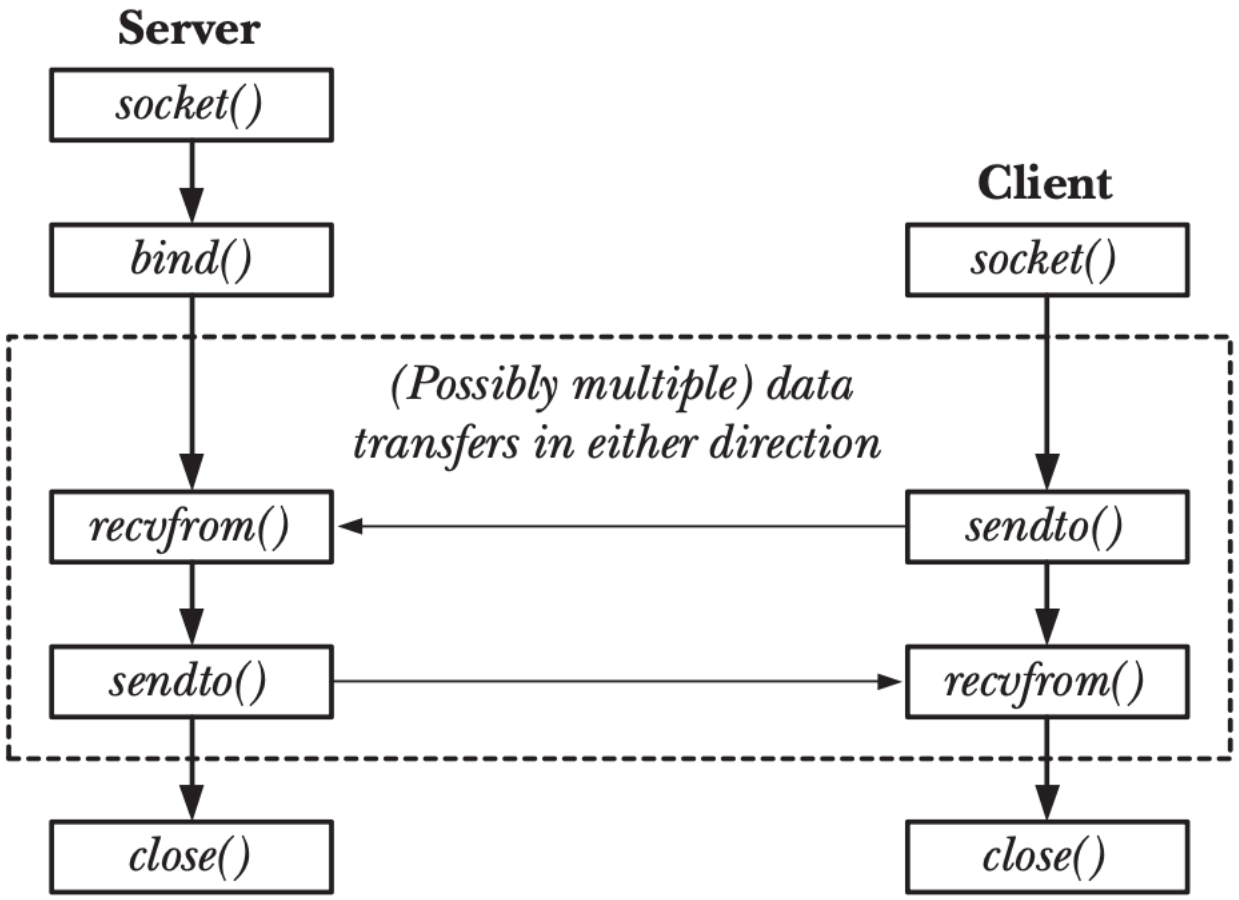
Socket Programming



Stream Sockets



- Datagram Sockets



UNIX Domain Socket

- Sockets can be used as IPC within the same host, too

Domain	Communication performed	Communication between applications	Address format	Address structure
AF_UNIX	within kernel	on same host	pathname	*sockaddr_un*
AF_INET	via IPv4	on hosts connected via an IPv4 network	32-bit IPv4 address + 16-bit port number	*sockaddr_in*
AF_INET6	via IPv6	on hosts connected via an IPv6 network	128-bit IPv6 address + 16-bit port number	*sockaddr_in6*

Socket Connection Setup over TCP/IP

```c
/* server.c */
void server(int sockfd) {
    char buf[BUF_SIZE];
    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
int main() {
    int servfd, sockfd;
    struct sockaddr_in addr;
    // Create socket fd
    servfd = socket(AF_INET, SOCK_STREAM, 0);
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(PORT);
    // Bind sockfd to port
    bind(servfd, (struct sockaddr *)&addr,
         sizeof(addr));
    // Listen
    listen(servfd, 10);
    while (1) {
        // Accept a connection
        sockfd = accept(servfd, NULL, NULL);
        server(sockfd);
        close(sockfd);
    }
    close(servfd);
    return 0;
```

```c
/* client.c */
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);
        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
int main() {
    int sockfd;
    struct sockaddr_in addr;
    // Create socket fd
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    addr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1",
              &addr.sin_addr);
    addr.sin_port = htons(PORT);
    // Connect to server
    connect(sockfd, (struct sockaddr *)&addr,
            sizeof(addr));

    client(sockfd);

    close(sockfd);

    return 0;
}
```

Socket Connection Setup over UNIX Domain (Local)

```c
/* server.c */
void server(int sockfd) {
    char buf[BUF_SIZE];
    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
int main() {
    int servfd, sockfd;
    struct sockaddr_in addr;
    // Create socket fd
    servfd = socket(AF_INET, SOCK_STREAM, 0);
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(PORT);
    // Bind sockfd to port
    bind(servfd, (struct sockaddr *)&addr,
         sizeof(addr));
    // Listen
    listen(servfd, 10);
    while (1) {
        // Accept a connection
        sockfd = accept(servfd, NULL, NULL);
        server(sockfd);
        close(sockfd);
    }
    close(servfd);
    return 0;
```

```c
/* local_server.c */
void server(int sockfd) {
    char buf[BUF_SIZE];
    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
int main() {
    int servfd, sockfd;
    struct sockaddr_un addr;
    // Create socket fd
    servfd = socket(AF_UNIX, SOCK_STREAM, 0);
    addr.sun_family = AF_UNIX;
    sprintf(addr.sun_path, "/tmp/server_%d.sock",
            getuid());
    // Bind sockfd to port
    bind(servfd, (struct sockaddr *)&addr,
         sizeof(addr));
    // Listen
    listen(servfd, 10);
    while (1) {
        // Accept a connection
        sockfd = accept(servfd, NULL, NULL);
        server(sockfd);
        close(sockfd);
    }
    close(servfd);
    return 0;
```

Socket Connection Setup over UNIX Domain (Local)

```
$ colordiff server.c local_server.c
1c1
< /* server.c */
---
> /* local_server.c */
9c9
< #include <netinet/in.h>
---
> #include <sys/un.h>
30c30
<     struct sockaddr_in addr;
---
>     struct sockaddr_un addr;
33c33,36
<     servfd = socket(AF_INET, SOCK_STREAM, 0);
---
>     servfd = socket(AF_UNIX, SOCK_STREAM, 0);
>
>     addr.sun_family = AF_UNIX;
>     sprintf(addr.sun_path, "/tmp/server_%d.sock", getuid());
35,37d37
<     addr.sin_family = AF_INET;
<     addr.sin_addr.s_addr = INADDR_ANY;
<     addr.sin_port = htons(PORT);
```

Socket Connection Setup over UNIX Domain (Local)

```c
/* client.c */
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);
        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
int main() {
    int sockfd;
    struct sockaddr_in addr;
    // Create socket fd
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    addr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1",
              &addr.sin_addr);
    addr.sin_port = htons(PORT);
    // Connect to server
    connect(sockfd, (struct sockaddr *)&addr,
            sizeof(addr));

    client(sockfd);

    close(sockfd);

    return 0;
}
```

```c
/* local_client.c */
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);
        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
int main() {
    int sockfd;
    struct sockaddr_un addr;
    // Create socket fd
    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    addr.sun_family = AF_UNIX;
    sprintf(addr.sun_path, "/tmp/server_%d.sock",
            getuid());

    // Connect to server
    connect(sockfd, (struct sockaddr *)&addr,
sizeof(addr));

    client(sockfd);

    close(sockfd);

    return 0;
}
```

Socket Connection Setup over UNIX Domain (Local)

```
$ colordiff client.c local_client.c
1c1
< /* client.c */
---
> /* local_client.c */
9c9
< #include <netinet/in.h>
---
> #include <sys/un.h>
31c31
<     struct sockaddr_in addr;
---
>     struct sockaddr_un addr;
34c34
<     sockfd = socket(AF_INET, SOCK_STREAM, 0);
---
>     sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
36,38c36,37
<     addr.sin_family = AF_INET;
<     inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr);
<     addr.sin_port = htons(PORT);
---
>     addr.sun_family = AF_UNIX;
>     sprintf(addr.sun_path, "/tmp/server_%d.sock", getuid());
```

Socket Connection Setup over UNIX Domain (Local)

```c
/* local_server.c */
void server(int sockfd) {
    char buf[BUF_SIZE];
    while (1) {
        int n = read(sockfd, buf, BUF_SIZE);
        if (n <= 0) return;
        write(STDOUT_FILENO, buf, n);
        write(sockfd, buf, n);     // Echo
    }
}
int main() {
    int servfd, sockfd;
    struct sockaddr_un addr;
    // Create socket fd
    servfd = socket(AF_UNIX, SOCK_STREAM, 0);
    addr.sun_family = AF_UNIX;
    sprintf(addr.sun_path, "/tmp/server_%d.sock",
            getuid());
    // Bind sockfd to port
    bind(servfd, (struct sockaddr *)&addr,
         sizeof(addr));
    // Listen
    listen(servfd, 10);
    while (1) {
        // Accept a connection
        sockfd = accept(servfd, NULL, NULL);
        server(sockfd);
        close(sockfd);
    }
    close(servfd);
    return 0;
```

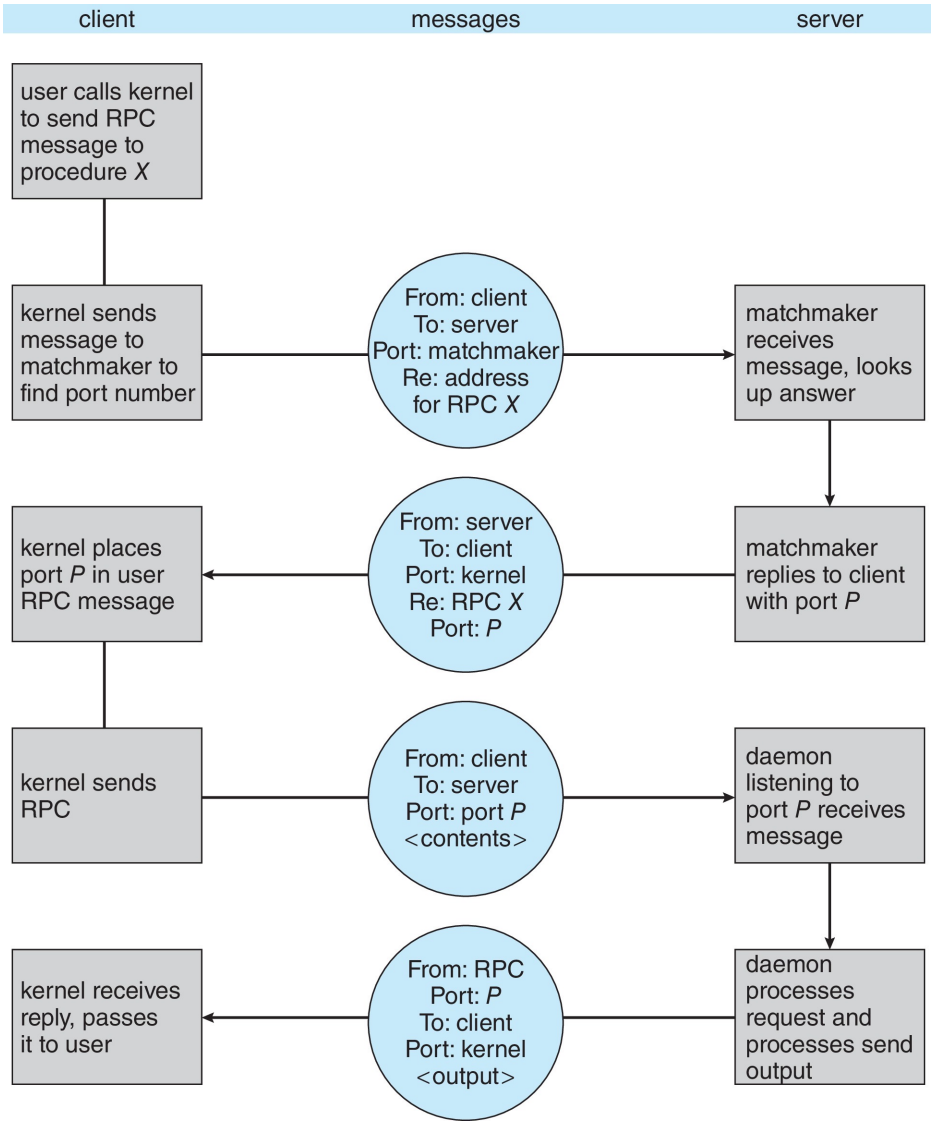
```c
/* local_client.c */
void client(int sockfd) {
    char sndbuf[BUF_SIZE], rcvbuf[BUF_SIZE];
    while (1) {
        fgets(sndbuf, BUF_SIZE, stdin);
        write(sockfd, sndbuf, strlen(sndbuf)+1);
        int n = read(sockfd, rcvbuf, BUF_SIZE);
        write(STDOUT_FILENO, rcvbuf, n);
    }
}
int main() {
    int sockfd;
    struct sockaddr_un addr;
    // Create socket fd
    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
    addr.sun_family = AF_UNIX;
    sprintf(addr.sun_path, "/tmp/server_%d.sock",
            getuid());

    // Connect to server
    connect(sockfd, (struct sockaddr *)&addr,
sizeof(addr));

    client(sockfd);

    close(sockfd);

    return 0;
}
```

Remote Procedure Calls

- Remote Procedure Call (RPC) abstracts procedure calls between processes on networked systems
  - RPC is a protocol that allows a program on one computer to execute a procedure on another computer without having to worry about the network communications between them.
  - Again uses ports for service differentiation
- Stubs(桩函数) – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and marshals the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compiled from specification written in Microsoft Interface Definition Language (MIDL)

- **Remote Procedure Calls**
  - Data representation handled via External Data Representation (XDL) format to account for different architectures
    - Big-endian and little-endian
  - Remote communication has more failure scenarios than local
    - Messages can be delivered exactly once rather than at most once
  - OS typically provides a rendezvous (or matchmaker) service to connect client and server

Remote Procedure Call Example



Remote Procedure Call Example

- 1. Create a file named `adder.x` with the following content:

```
struct add_operands {
    int a;
    int b;
};


program ADDER_PROGRAM {
    version ADDER_VERSION {
        int ADD(add_operands) = 1;
    } = 1;
} = 0x20000001;
```

Remote Procedure Call Example

- 1. Create a file named `adder.x` with the following content:

```
struct add_operands {
    int a;
    int b;
};


program ADDER_PROGRAM {
    version ADDER_VERSION {
        int ADD(add_operands) = 1;
    } = 1;
} = 0x20000001;
```

- 2. Generate the C code using `` `rpcgen` ``:

```
$ rpcgen -a -C adder.x
$ ls
adder_client.c  adder_clnt.c  adder.h  adder_server.c  adder_svc.c
adder.x  adder_xdr.c  Makefile.adder
```

Remote Procedure Call Example

- 2. Generate the C code using `rpcgen`:

```
$ rpcgen -a -C adder.x
$ ls
adder_client.c  adder_clnt.c  adder.h  adder_server.c  adder_svc.c
adder.x  adder_xdr.c  Makefile.adder
```

  - `adder.h`: Header file for the RPC program.
  - `adder_svc.c`: Server-side stubs.
  - `adder_clnt.c`: Client-side stubs.
  - `adder_xdr.c`: XDR routines file.
  - `adder_server.c`: Template for the server.
  - `adder_client.c`: Template for the client.

Remote Procedure Call Example

- 3. Edit `adder_server.c` to implement the `ADD` function:

```c
#include "adder.h"

int *add_1_svc(add_operands *argp, struct svc_req *rqstp) {
    static int result;

    result = argp->a + argp->b;

    return &result;
}
```

Remote Procedure Call Example

- 4. Edit `adder_client.c` to call the server function:

```c
#include "adder.h"

void adder_program_1(char *host, int a, int b) {
    CLIENT *clnt;
    int *result_1;
    add_operands add_1_arg;

#ifndef DEBUG
    clnt = clnt_create(host, ADDER_PROGRAM,
ADDER_VERSION, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror(host);
        exit(1);
    }
#endif /* DEBUG */

    add_1_arg.a = a;
    add_1_arg.b = b;
    result_1 = add_1(&add_1_arg, clnt);
    if (result_1 == (int *)NULL) {
        clnt_perror(clnt, "call failed");
    } else {
        printf("Result: %d\n", *result_1);
    }
#ifndef DEBUG
    clnt_destroy(clnt);
#endif /* DEBUG */
}
```

```c
int main(int argc, char *argv[]) {
    char *host;

    if (argc < 4) {
        printf("usage: %s server_host num1
num2\n", argv[0]);
        exit(1);
    }
    host = argv[1];
    int a = atoi(argv[2]);
    int b = atoi(argv[3]);

    adder_program_1(host, a, b);
    exit(0);
}
```

Remote Procedure Call Example

- 4. Edit `adder_client.c` to call the server function:

```c
#include "adder.h"

void adder_program_1(char *host, int a, int b) {
    CLIENT *clnt;
    int *result_1;
    add_operands add_1_arg;

#ifndef DEBUG
    clnt = clnt_create(host, ADDER_PROGRAM,
ADDER_VERSION, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror(host);
        exit(1);
    }
#endif /* DEBUG */

    add_1_arg.a = a;
    add_1_arg.b = b;
    result_1 = add_1(&add_1_arg, clnt);
    if (result_1 == (int *)NULL) {
        clnt_perror(clnt, "call failed");
    } else {
        printf("Result: %d\n", *result_1);
    }
#ifndef DEBUG
    clnt_destroy(clnt);
#endif /* DEBUG */
}
```

```c
int main(int argc, char *argv[]) {
    char *host;

    if (argc < 4) {
        printf("usage: %s server_host num1
num2\n", argv[0]);
        exit(1);
    }
    host = argv[1];
    int a = atoi(argv[2]);
    int b = atoi(argv[3]);

    adder_program_1(host, a, b);
    exit(0);
}
```

Remote Procedure Call Example

- 5. Compile and generate `adder_server` and `adder_client`:

```
$ gcc -o adder_server adder_server.c adder_svc.c adder_xdr.c -ltirpc
$ gcc -o adder_client adder_client.c adder_clnt.c adder_xdr.c -ltirpc
```

- 6. Start the server:

```
$ ./adder_server
```

- 7. In another terminal, run the client to add two numbers:

```
$ ./adder_client localhost 3 4
Result: 7
```

Thank you!