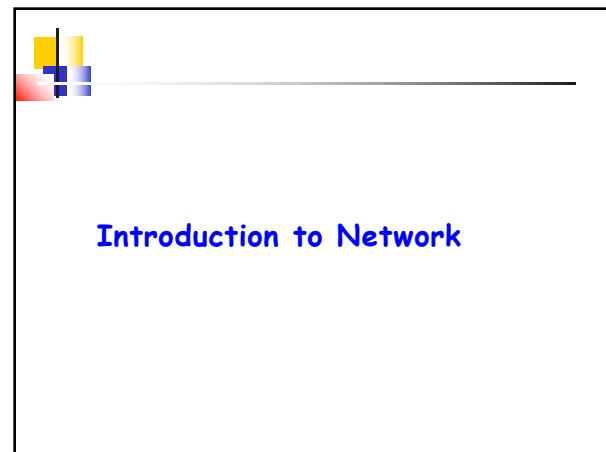
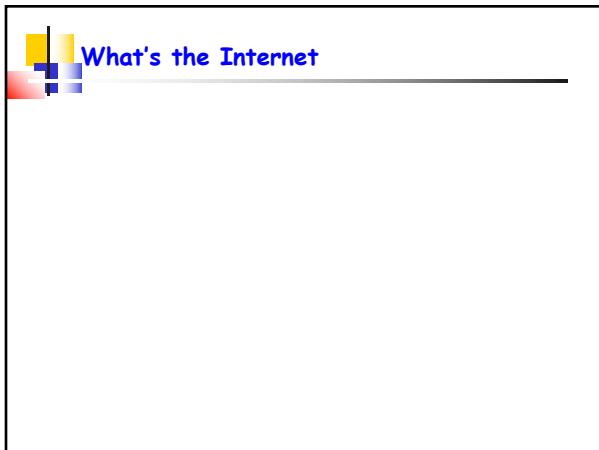


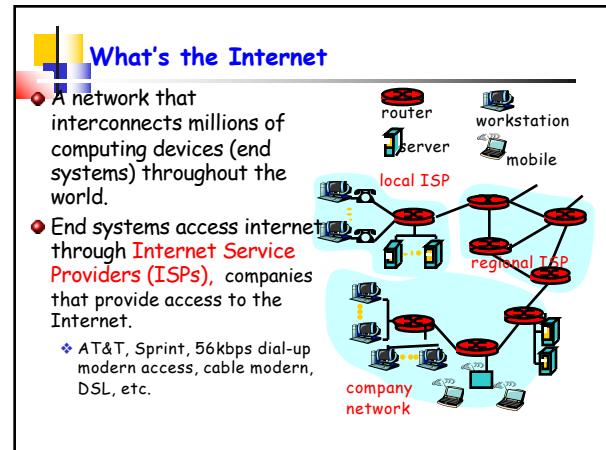
**CS458/CS558**  
**Introduction to Computer Security**



**Introduction to Network**

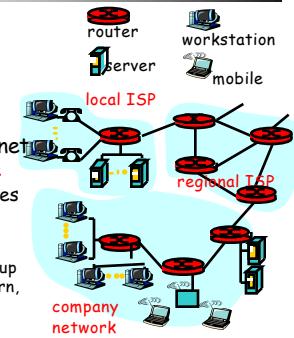
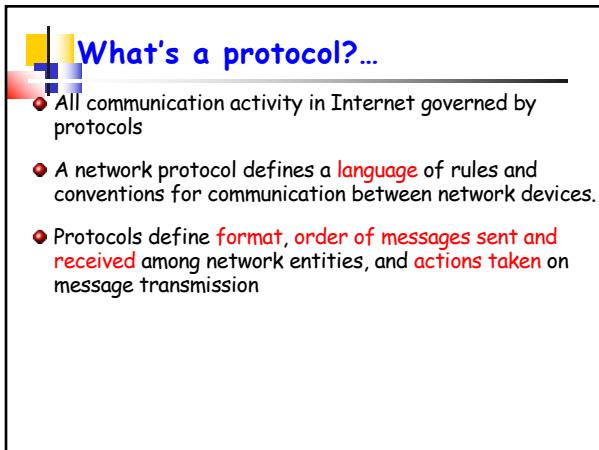


**What's the Internet**



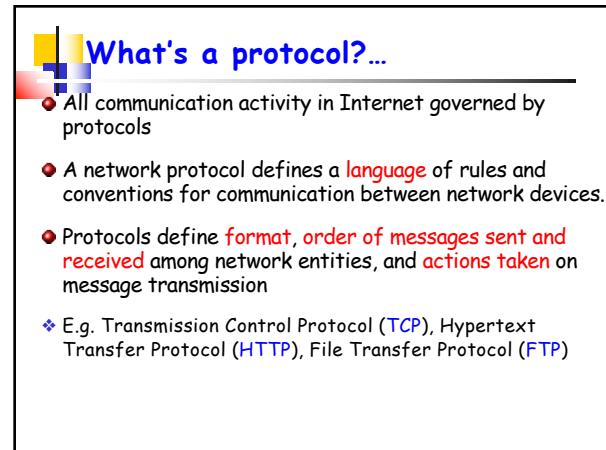
**What's the Internet**

- A network that interconnects millions of computing devices (end systems) throughout the world.
- End systems access internet through **Internet Service Providers (ISPs)**, companies that provide access to the Internet.
  - ❖ AT&T, Sprint, 56kbps dial-up modern access, cable modem, DSL, etc.

**What's a protocol?...**

- All communication activity in Internet governed by protocols
- A network protocol defines a **language** of rules and conventions for communication between network devices.
- Protocols define **format, order of messages sent and received** among network entities, and **actions taken on message transmission**

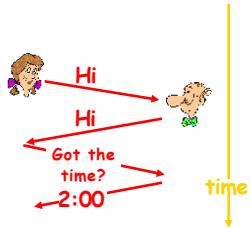


**What's a protocol?...**

- All communication activity in Internet governed by protocols
- A network protocol defines a **language** of rules and conventions for communication between network devices.
- Protocols define **format, order of messages sent and received** among network entities, and **actions taken on message transmission**
  - ❖ E.g. Transmission Control Protocol (**TCP**), Hypertext Transfer Protocol (**HTTP**), File Transfer Protocol (**FTP**)

## What's a protocol?

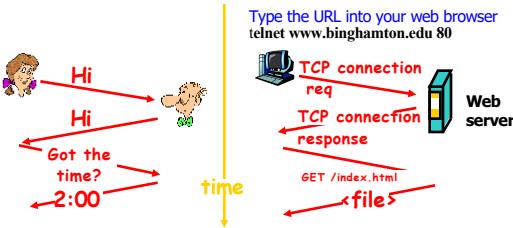
A human protocol and a computer network protocol



A network protocol is similar to a human protocol except that the entities sending and receiving msgs are hardware/software components of some device.

## What's a protocol?

A human protocol and a computer network protocol:



A network protocol is similar to a human protocol except that the entities sending and receiving msgs are hardware/software components of some device.

## Protocol Layers

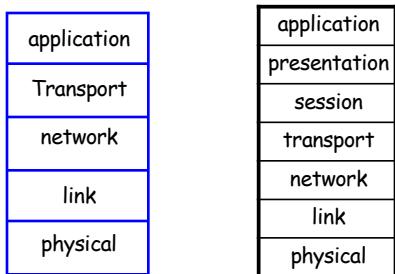
## Protocol Layers

- Dealing with complex systems:

- Provide a structural way to discuss system components.
- Modularization eases maintenance, updating of system
  - Change of implementation of layer's service transparent to rest of system

## Protocol Layers (Cont.)

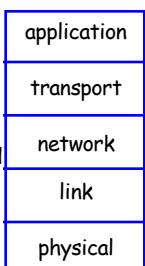
- TCP/IP model: 5 layers
- OSI reference model: 7 layers



## Internet protocol stack (TCP/IP Model)

- Application

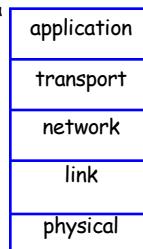
- Provides a means for the user to access information on the network through an application.
- Supports network applications and application-layer protocols such as **FTP, HTTP, SMTP**.
- Data sent over the network is passed into the application layer where it is encapsulated into the application layer protocol. The data is passed down into the transport layer.



## Internet protocol stack (TCP/IP Model)

### Transport

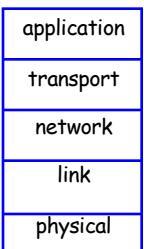
- Provides transparent transfer of data between end users
- Controls the reliability of a given link through flow control, segmentation/desegmentation, and error control



## Internet protocol stack (TCP/IP Model)

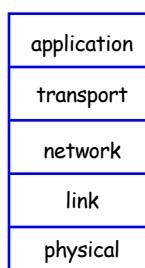
### Transport

- Provides transparent transfer of data between end users
- Controls the reliability of a given link through flow control, segmentation/desegmentation, and error control
- Converts messages into TCP segments or User Datagram Protocol (UDP), etc.
  - > TCP: a reliable connection-oriented protocol
  - > UDP: an unreliable, connectionless protocol, e.g. streaming media (audio, video, etc).



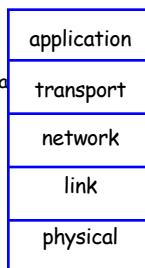
## Internet protocol stack (TCP/IP Model)

- Network:** routes datagrams from source to destination
  - Routers operate at this layer
  - IP, routing protocols
- Link:** provides the functional and procedural means to transfer data between network entities
  - Bridges and link-layer switches operate.

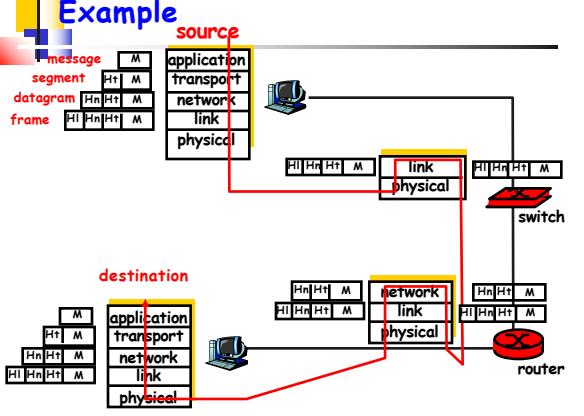


## Internet protocol stack (TCP/IP Model)

- Physical:** encodes and transmits raw data over network communications media (e.g. optical fiber).
  - Make sure that when one side sends a 1 bit, it is received by the other side as 1 bit.



## Example



## Reference

- TCP/IP model:** [http://en.wikipedia.org/wiki/TCP/IP\\_model](http://en.wikipedia.org/wiki/TCP/IP_model)



## Socket Programming

### Client-Server Model

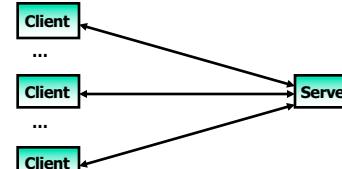
- Most network applications use the **client-server model**.



- Client:** requests, receives service from an always-on **server**
  - > Needs to know of the existence of and the address of the **server**.
- Server** does not need to know the address of the **client** prior to the connection being established.
- Once a connection is established, both sides can send and receive information.
- A good analogy is a person who makes a phone call to another person.
- e.g. **Web browser/server; email client/server**

### Client-Server Model

- Most network applications use the **client-server model**.



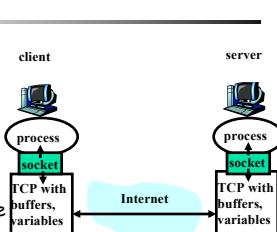
- Clients usually communicate with one server a time
- It is not unusual for a server to be communicating with multiple clients

### Socket

- The system calls for establishing a connection are different for the client and the server
- But both involve the basic construct of a **socket**.

### Sockets

- Process sends/receives messages to/from its **socket**
- Socket analogous to door
  - & Sending process shoves message out door
  - & Transport infrastructure brings message to the door at receiving process



### Addressing Processes

- For a process to receive messages, it must have an **identifier**.

## Addressing Processes

- For a process to receive messages, it must have an identifier.
- Identifier includes both the IP address and port number associated with the process on the host.
  - ❖ A host has an IP address
  - ❖ Does the IP address of the host on which the process runs suffice for identifying the process?
    - > Answer: no, many processes can be running on same host
  - ❖ Port: A 16-bit number to identify the application process that is a network endpoint.

## IP Address (IPv4)

- An identifier for each machine connected to an IP network.
  - ❖ 32 bit binary number
  - ❖ Represented as dotted decimal notation:
    - > 4 decimal values, each representing 8 bits (octet), in the range 0 to 255.
- Example:
  - ❖ Dotted Decimal: 140.179.220.200
  - ❖ Binary: 10001100.10110011.11011100.11001000

## Ports

- A 16-bit number to identify the application process that is a network endpoint.
- Reserved ports or well-known ports (0 to 1023)
- Standard TCP ports for well-known applications: Telnet (23), ftp(21), http (80).
- Ephemeral ports (1024-65535) : for ordinary user-developed programs.

## Establish A TCP Socket on the Client Side

- Create a socket with the socket() system call
- Specify server's IP address and port
- Establish connection with server using the connect() system call
- Send and receive data, e.g., use the read() and write() system calls.

## Socket()

- Create a socket with the socket() system call
    - //Contains data definitions and socket structures.
  - #include <sys/socket.h>
  - int socket(int family, int type, int protocol)
- Returns: non-negative descriptor if OK, -1 on error
- ❖ Integer descriptor: identify the socket in all future function calls
  - ❖ Protocol family constants
    - > e.g. AF\_INET: IPv4, AF\_INET1: IPv6.
  - ❖ Type of socket
    - > SOCK\_STREAM: stream socket, SOCK\_DGRAM: datagram socket
  - ❖ Protocol: normally 0 except for raw socket

## Specify Server's IP Address and Port

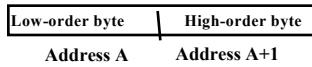
- Specify server's IP address and port
- E.g. for TCP connection:
 

```
struct sockaddr_in servaddr;
//set the socket address structure 0
bzero(&servaddr, sizeof(servaddr));
//set the address family to AF_INET
servaddr.sin_family = AF_INET;
//set the port number.
servaddr.sin_port = htons(<port number>);
//set the ip address.
if (inet_pton(AF_INET, <ip addr>, &servaddr.sin_addr) <= 0)
```

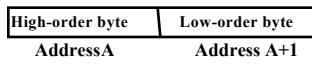
## Network-Byte Ordering

Two ways to store 16-bit/32-bit integers

- Little-endian byte order (e.g. Intel)



- Big-endian byte order (E.g. Sparc)



## Network-Byte Ordering (cont.)

- How do two machines with different byte-orders communicate?
  - Using network byte-order
  - Network byte-order = big-endian order

- Converting between the host byte order and the network byte order (<netinet/in.h>)

- h: host: s: short, l: long
  - > uint16\_t htons(uint16\_t n)
  - > uint32\_t htonl(uint32\_t n)
  - > uint16\_t ntohs(uint16\_t n)
  - > uint32\_t ntohl(uint32\_t n)

## Specify Server's IP Address and Port

- Specify server's IP address and port

- E.g. for TCP connection:

```
struct sockaddr_in servaddr;
//set the socket address structure
bzero(&servaddr, sizeof(servaddr));
//set the address family to AF_INET
servaddr.sin_family = AF_INET;
//set the port number.
servaddr.sin_port = htons(<port number>);
//set the ip address.
if (inet_pton(AF_INET, <ip addr>, &servaddr.sin_addr) <= 0)
```

## Inet\_pton, inet\_ntop

<arpa/inet.h>

//Returns 1 if OK, 0 if input is not a valid format, -1 on error  
int inet\_pton(int family, const char \*strptr, void \*addrptr);

//Returns the pointer to result if OK, NULL on errors  
const char \*inet\_ntop(int family, const void \*addrptr, size\_t len);

- p: presentation

- Usually an ASCII string

- n: network

- Binary value that goes into a socket address structure

## Connect()

- Establish a connection with the TCP server using the `connect()` system call

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr,
socklen_t addrlen);
Return 0 if OK, -1 on error
```



## read(), write()

- Send and receive data, e.g., use the `write()` and `read()` system calls.

//Read up to count bytes from the socket into the buffer

// Return the number of bytes read

int read(int sockfd, void \*buf, int count);

// Write data to a TCP connection

int write(int sockfd, void \*buf, int count)

### Establish A Socket on the Server Side

- 1. Create a socket with the `socket()` system call
- 2. Bind the socket to an address using the `bind()` system call.
- 3. Listen for connections with the `listen()` system call
- 4. Accept a connection with the `accept()` system call.
- 5. Send and receive data

### bind(), listen()

- The server specifies the IP address and port number associated with a socket using `bind()`.

```
int bind(int sockfd, const struct sockaddr *myaddr,
socklen_t addrlen)
```

- Listen for connections with the `listen()` system call.

```
int listen(int sockfd, int backlog)
```

**backlog:** the number of maximum pending clients



### accept()

- Accept a connection with the `accept()` system call.

```
int accept(int sockfd, struct sockaddr *client_addr,
socklen_t *addrlen)
```

- `accept()` returns a new descriptor that is automatically created by the kernel. This descriptor refers to the TCP connection with the client.



### Example of Client-Server Operation

A Simple Daytime Client and Server

### Daytime client

- Connects to a daytime server
- Retrieves the current date and time

% cli 128.226.6.39

Wed Feb 05 18:10:00 2014

### Daytime client

```
int main(int argc, char **argv) {
    int sockfd, n;
    char recvline[MAX + 1];
    struct sockaddr_in servaddr;
    if( argc != 2 ) {
        printf("Usage: cli <IP address>");
        exit(1);
    }
    /* Create a TCP socket */
    if((sockfd=socket(AF_INET,SOCK_STREAM, 0))<0){
        perror("socket");
        exit(2);
    }
    /* Specify server's IP address and port */
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(10000); /* daytime server port */
    if(inet_pton(AF_INET, argv[1], &servaddr.sin_addr)<=0) {
        perror("inet_nton");
        exit(3);
    }
}
```

```


/* Connect to the server */
if(connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
    perror("connect"); exit(4);
}

/* Read from socket */
while ((n = read(sockfd, recvline, MAX)) > 0) {
    recvline[n] = '\0'; /* null terminate */
    printf("%s", recvline);
}

if (n < 0) { perror("read"); exit(5); }
close(sockfd);
}

```

## Daytime Server

1. Waits for requests from Client
2. Accepts client connections
3. Sends the current time
4. Terminates connection and goes back waiting for more connections.

```


int main(int argc, char **argv) {
    int listenfd, connfd;
    struct sockaddr_in servaddr, cliaddr;
    char buff[MAX];
    time_t ticks;

    /* Create a TCP socket */
    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    /* Initialize server's address and well-known port */
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    /* allowed your program to work without knowing the IP address of the machine it was running on */
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(10000); /* daytime server */

    /* Bind server's address and port to the socket */
    bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
}

```

```


/* Convert socket to a listening socket - max 100 pending clients*/
listen(listenfd, 100);

for (;;) {
    /* Wait for client connections and accept them */
    clilen = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);

    /* Retrieve system time */
    ticks = time(NULL);
    sprintf(buff, sizeof(buff), "%s\r\n", ctime(&ticks));

    /* Write to socket */
    write(connfd, buff, strlen(buff));

    /* Close the connection */
    close(connfd);
}

```

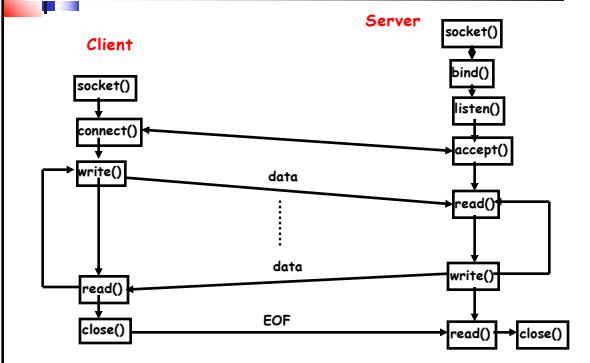
## Run Daytime Client-Server

- Compiling the code on bingsuns
 

```
gcc cli.c -o cli -lresolv -lsocket -lsl
gcc ser.c -o ser -lresolv -lsocket -lsl
```
- Executing the code on bingsuns
 

```
./ser
./cli 128.226.6.39
```

## TCP Connection Sequence



## Summary: Socket API

- **int socket(int family, int type, int protocol):** Creates a socket
- **int connect(int sockfd, const struct sockaddr \*servaddr, socklen\_t addrlen)**
  - ❖ Enables a client to connect to a server.
- **int bind(int sockfd, const struct sockaddr \*myaddr, socklen\_t addrlen)**
  - ❖ Allows a server to specify the IP address/port\_number associated with a socket
- **int listen(int sockfd, int backlog)**
  - ❖ Allows the server to specify a socket that can be used to accept connections.
- **int accept(int sockfd, struct sockaddr \*client\_addr, socklen\_t \*addrlen)**
  - ❖ Allows a server to wait till a new connection request arrives.
- **int close(int sockfd):** Terminates any connection associated with a socket and releases the socket descriptor.

## Concurrent Servers

- Daytime client-server: iterative servers
- Concurrent Servers: handle multiple clients simultaneously
  - ❖ Fork
  - ❖ Threads

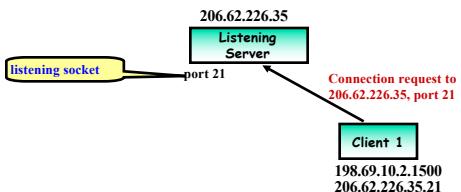
## Concurrent Servers

- Daytime client-server: iterative servers
- Concurrent Servers: handle multiple clients simultaneously
  - ❖ Fork
  - ❖ Threads

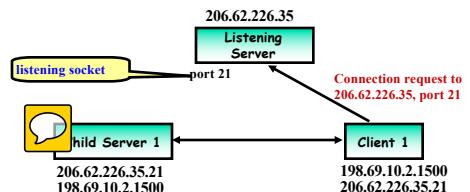
## Forking Concurrent Servers

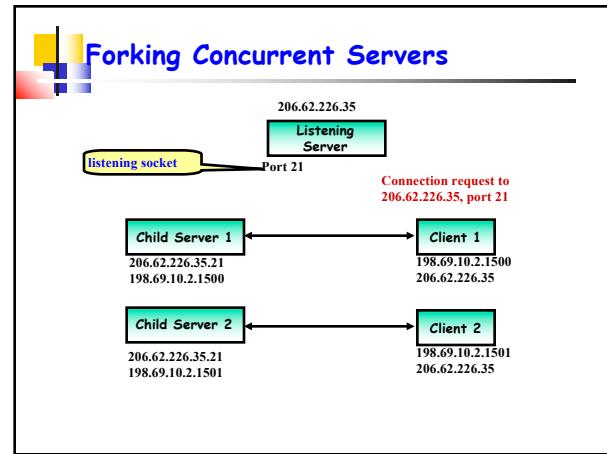
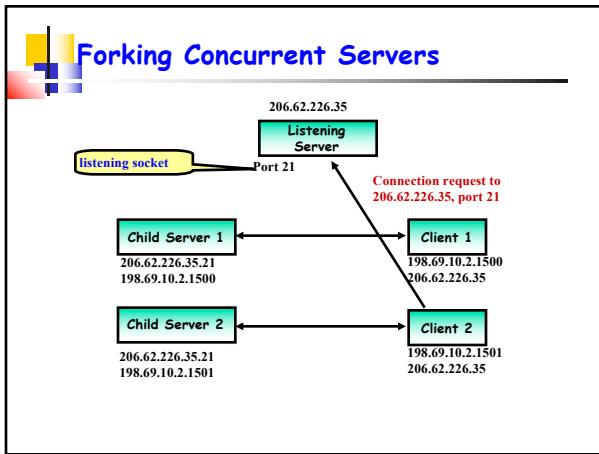
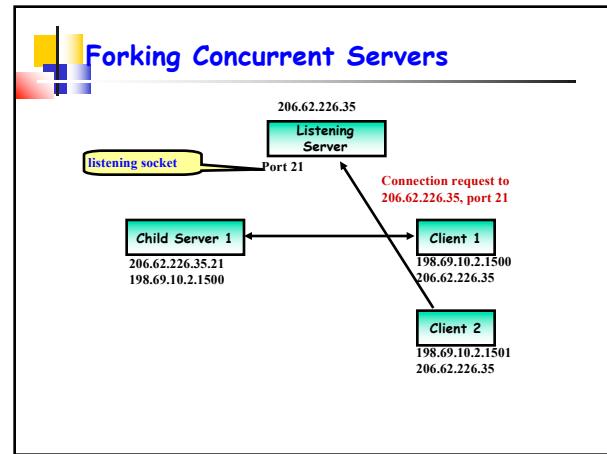
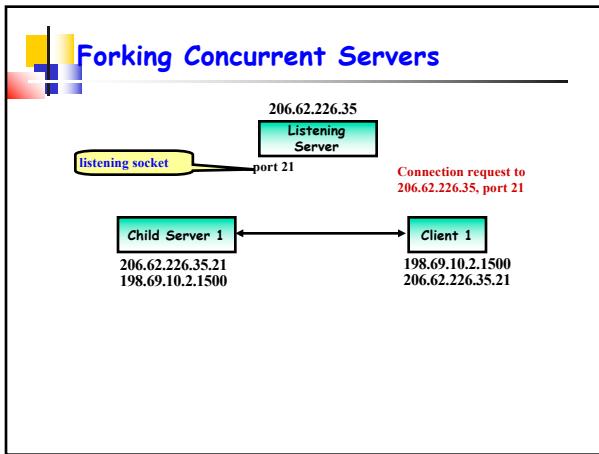


## Forking Concurrent Servers



## Forking Concurrent Servers





### Forking Server Example

```
listenfd = socket( ... )
bind(listenfd, ... )
listen(listenfd,...);
for (;;) {
    /* wait for client connection */
    connfd = accept(listenfd,...);
    if( (pid = fork()) == 0) {
        /* Child Server */
        close(listenfd);           //child closes listening socket
        service_client(connfd);   //process the request
        close(connfd);            //done with this client
        exit(0);                  //child terminates
    }
    /* Parent */
    close(connfd);             //parent closes connected socket
}
```

### Java Socket Programming: An Example

## Client

```
import java.io.*;
import java.net.*;
class TCPClient {
    public static void main(String argv[]) throws Exception {
        String modifiedSentence;
        Socket sock = new Socket("bingsuns.binghamton.edu",
6789);
        /*Open an input and output stream to the socket.*/
        PrintWriter out =
            new PrintWriter(sock.getOutputStream(),true);
        BufferedReader in =
            new BufferedReader(
            new InputStreamReader(sock.getInputStream()));
    }
}
```

## Client

```
/*Writes out the string to the underlying output stream.*/
out.println("hello");
/*Read a line of text*/
modifiedSentence = in.readLine();
System.out.println("FROM SERVER: " +
modifiedSentence);
sock.close();
}}
```

## Server

```
import java.io.*;
import java.net.*;
class TCPServer {
    public static void main(String argv[]) throws Exception{
        String clientSentence, capitalizedSentence;
        ServerSocket listen = new ServerSocket(6789);
        while(true) {
            Socket conn = listen.accept();
            BufferedReader in = new BufferedReader(
                new InputStreamReader(conn.getInputStream()));
            PrintWriter out =
                new PrintWriter(conn.getOutputStream(),true);
            clientSentence = in.readLine();
            System.out.println("FROM CLIENT:" + clientSentence);
            capitalizedSentence = clientSentence.toUpperCase();
            out.println(capitalizedSentence);
            conn.close();
        }
    }
}
```

## Compilation & Execution

- Compiling the code on bingsuns
 

```
javac TCPServer.java
javac TCPClient.java
```

- Executing the code on bingsuns
 

```
java TCPServer
java TCPClient
```

## References

- Package java.io
  - <http://java.sun.com/j2se/1.4.2/docs/api/java/io/package-summary.html>
- Tutorials and examples
  - <http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html>
  - <http://java.sun.com/docs/books/tutorial/networking/sockets/>
  - <http://www.prasannatech.net/2008/07/socket-programming-tutorial.html>
  - <http://zerioh.tripod.com/ressources/sockets.html>
  - <http://java.sun.com/docs/books/tutorial/essential/io/>