

---

# Network Applications and Network Programming

9/21/2009

1

## Outline

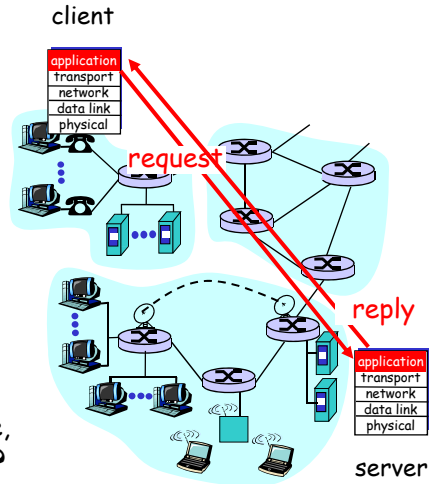
---

- Recap
- Email
- DNS
- Network application programming
- FTP

2

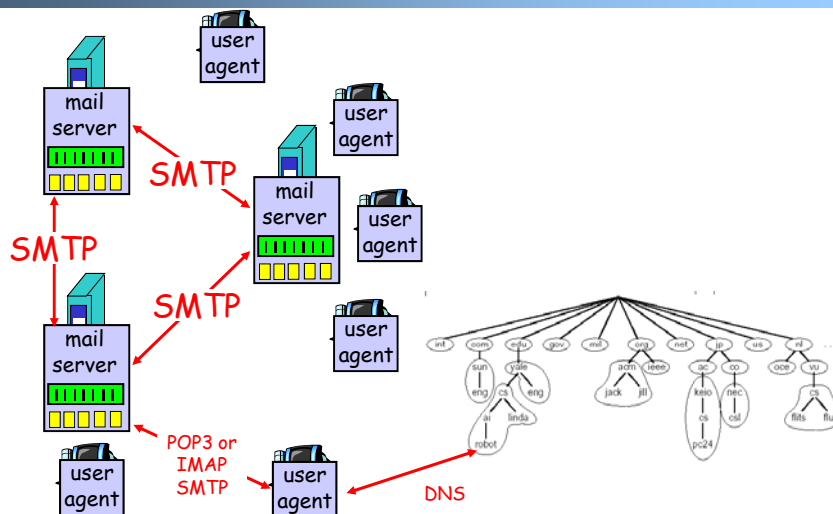
## Recap

- The basic paradigm of network applications is the client-server (C-S) paradigm
  - a client/server is a process at a port number of a host
- Key general questions of a C-S application
  - Is the application extensible, scalable, robust, and secure?



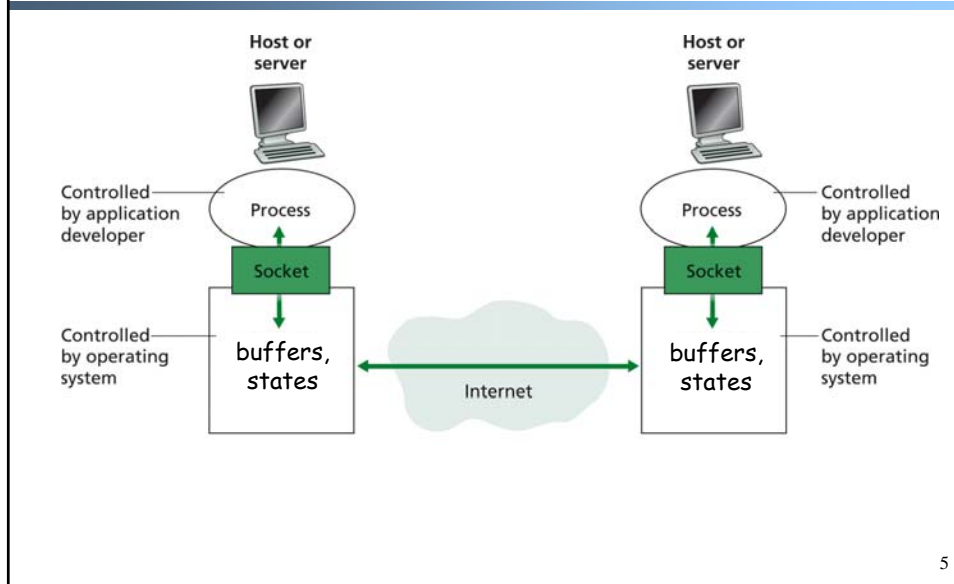
3

## Recap: Email Architecture



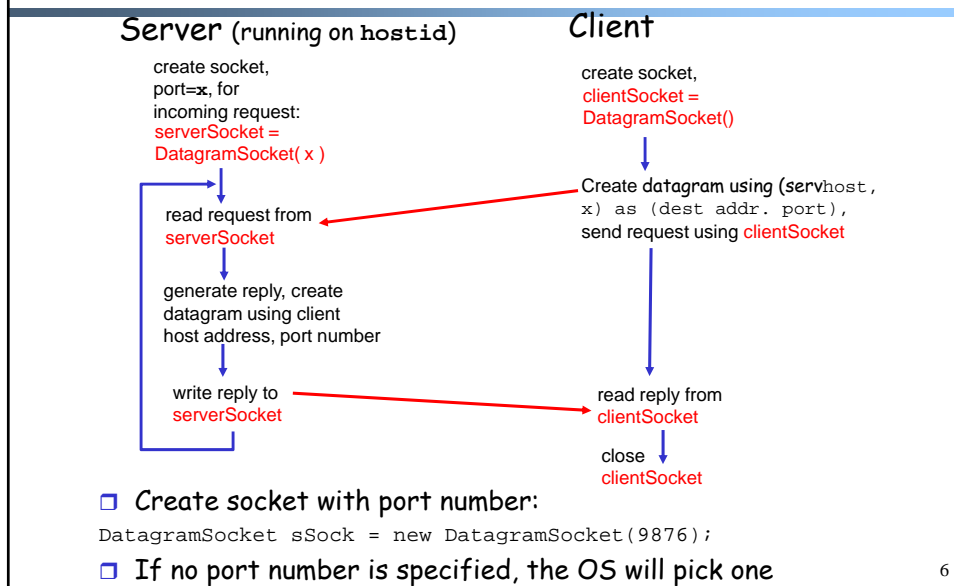
4

## Recap: Big Picture



5

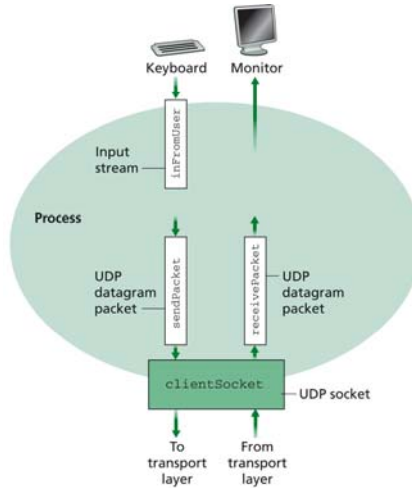
## Recap: UDP



6

## Example: UDPClient.java

- A simple UDP client which reads input from keyboard, sends the input to server, and reads the reply back from the server.



<http://zoo.cs.yale.edu/classes/cs433/programming/examples-java-socket/UDP/>

7

## Example: Java client (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception
    {
        Create
        input stream → BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
            String sentence = inFromUser.readLine();
            byte[] sendData = new byte[1024];
            sendData = sentence.getBytes();

        Create
        client socket → DatagramSocket clientSocket = new DatagramSocket();

        Translate
        hostname to IP
        address using DNS → InetAddress siPAddress = InetAddress.getByName("servname");
    }
}
```

8

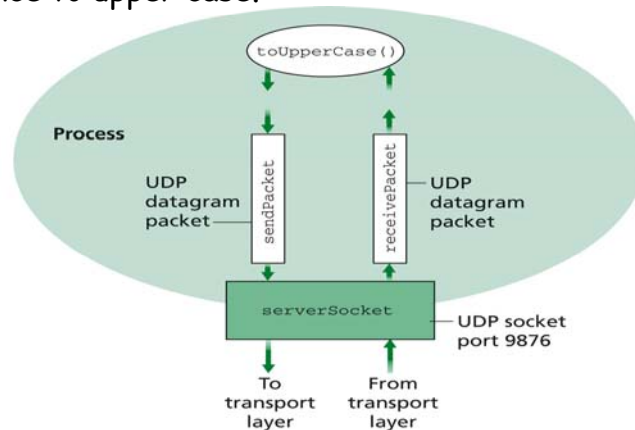
## Example: Java client (UDP), cont.

```
    Create datagram with data-to-send, length, IP addr, port } DatagramPacket sendPacket =
                                                                    new DatagramPacket(sendData, sendData.length, sIPAddress, 9876);
    Send datagram to server } clientSocket.send(sendPacket);
                                                                    byte[] receiveData = new byte[1024];
                                                                    DatagramPacket receivePacket =
                                                                    new DatagramPacket(receiveData, receiveData.length);
    Read datagram from server } clientSocket.receive(receivePacket);
                                                                    String modifiedSentence =
                                                                    new String(receivePacket.getData());
                                                                    System.out.println("FROM SERVER:" + modifiedSentence);
                                                                    clientSocket.close();
                                                                    }
                                                                    }
```

9

## Example: UDPServer.java

- A simple UDP server which changes any received sentence to upper case.



10

## Example: Java server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception
    {
        Create datagram socket at port 9876 → DatagramSocket serverSocket = new DatagramSocket(9876);

        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];

        while(true)
        {
            Create space for received datagram → DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);
            Receive datagram → serverSocket.receive(receivePacket);
        }
    }
}
```

11

## Example: Java server (UDP), cont

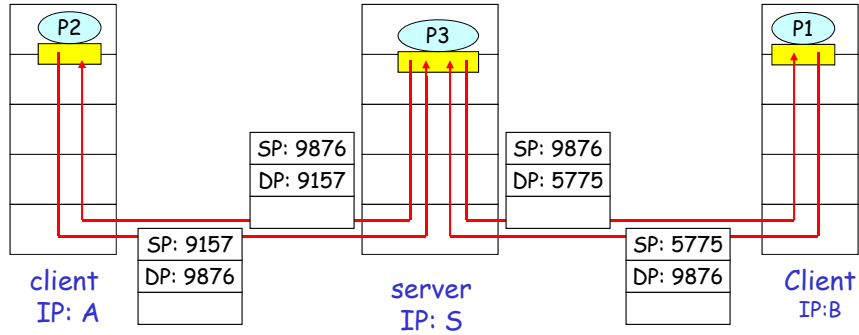
```
String sentence = new String(receivePacket.getData());
Get IP addr port #, of sender → InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();
sendData = capitalizedSentence.getBytes();
Create datagram to send to client → DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length,
        IPAddress, port);
Write out datagram to socket → serverSocket.send(sendPacket);
}
}
End of while loop, loop back and wait for another datagram
```

12

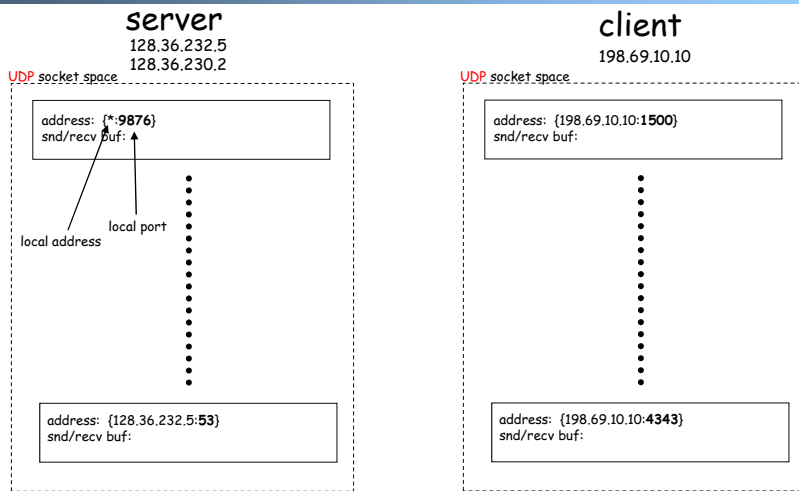
# UDP Connectionless Demux

```
DatagramSocket serverSocket = new DatagramSocket(9876);
```



Source Port (SP) provides "return address"

# UDP Provides Multiplexing/Demultiplexing



Packet demultiplexing is based on (dst address, dst port) at dst  
`%netstat -u -n -a`

## Discussion

- How robust is the sample program?

15

## Discussion

- What are challenges in implementing DNS using UDP?

Identification	Flags	
Number of questions	Number of answer RRs	-12 bytes
Number of authority RRs	Number of additional RRs	
Questions (variable number of questions)		-Name, type fields for a query
Answers (variable number of resource records)		-RRs in response to query
Authority (variable number of resource records)		-Records for authoritative servers
Additional information (variable number of resource records)		-Additional "helpful" info that may be used

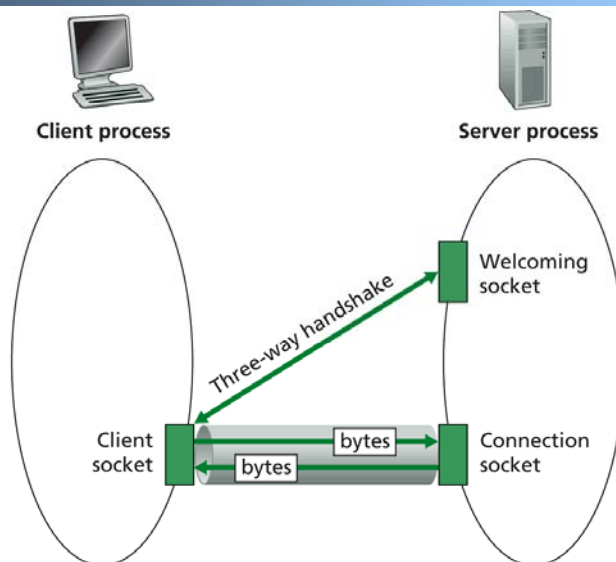
16

## Outline

- Recap
- Email
- DNS
- Network application programming
  - UDP
  - TCP

17

## Big Picture: Connection-Oriented TCP



18

## Socket Programming with TCP

### Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

### Client contacts server by:

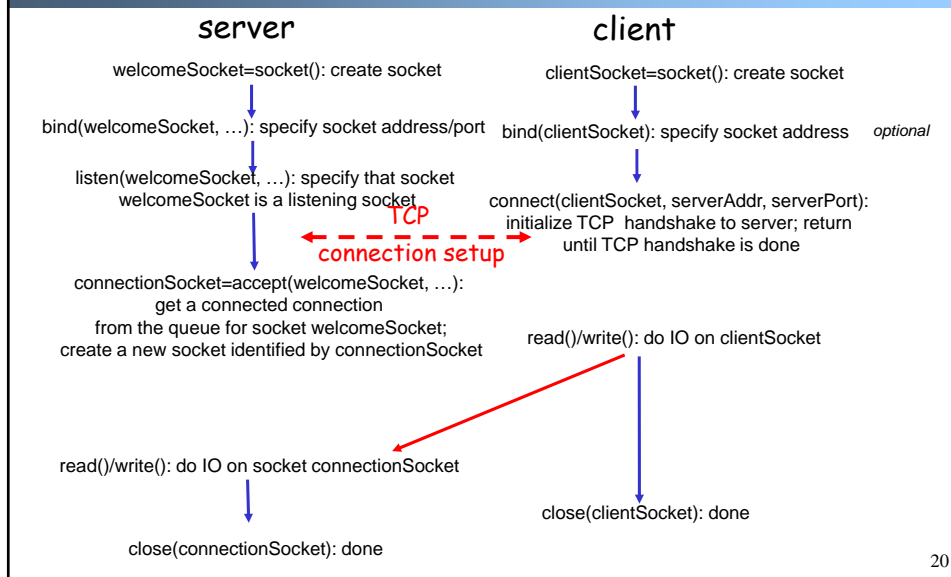
- creating client-local TCP socket
- specifying IP address, port number of server process
- When **client creates socket**: client TCP establishes connection to server TCP

- When contacted by client, **server TCP creates new socket** for server process to communicate with client

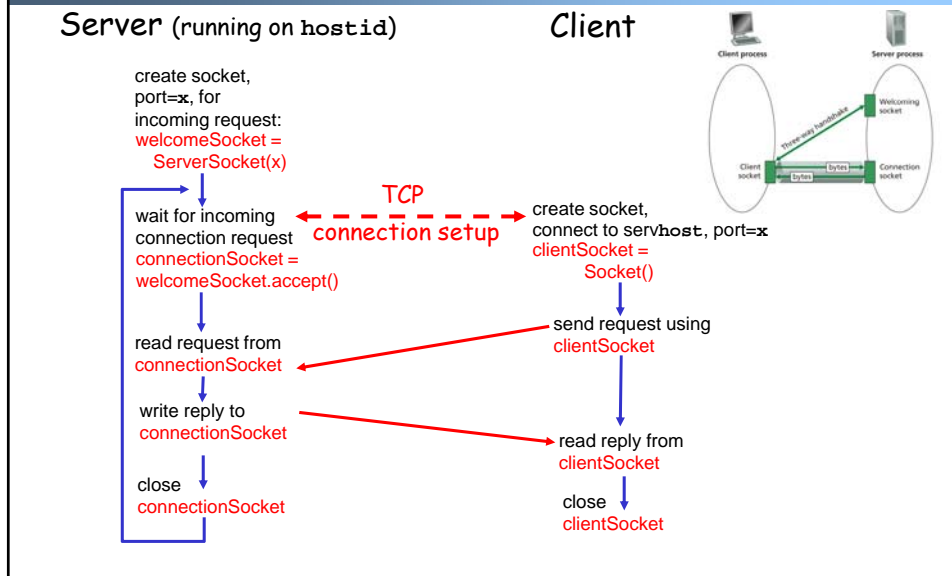
### application viewpoint

*TCP provides reliable, in-order transfer of bytes ("pipe") between client and server*

## Connection-oriented TCP: Big Picture (C version)



## Client/server socket interaction: TCP



## ServerSocket

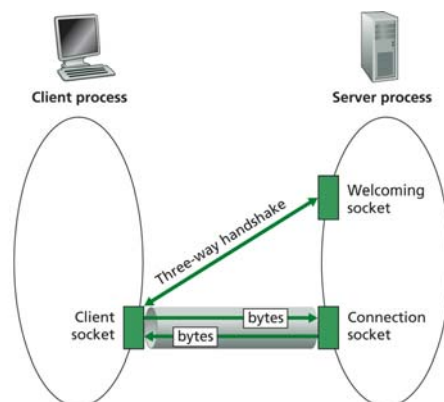
- ❑ **ServerSocket()**
  - creates an unbound server socket.
- ❑ **ServerSocket(int port)**
  - creates a server socket, bound to the specified port.
- ❑ **ServerSocket(int port, int backlog)**
  - creates a server socket and binds it to the specified local port number, with the specified backlog.
- ❑ **ServerSocket(int port, int backlog, InetAddress bindAddr)**
  - create a server with the specified port, listen backlog, and local IP address to bind to.
- ❑ **bind(SocketAddress endpoint)**
  - binds the ServerSocket to a specific address (IP address and port number).
- ❑ **bind(SocketAddress endpoint, int backlog)**
  - binds the ServerSocket to a specific address (IP address and port number).
- ❑ **Socket accept()**
  - listens for a connection to be made to this socket and accepts it.
- ❑ **close()**
  - closes this socket.

## (Client) Socket

- ❑ **Socket(InetAddress address, int port)**  
creates a stream socket and connects it to the specified port number at the specified IP address.
- ❑ **Socket(InetAddress address, int port, InetAddress localAddr, int localPort)**  
creates a socket and connects it to the specified remote address on the specified remote port.
- ❑ **Socket(String host, int port)**  
creates a stream socket and connects it to the specified port number on the named host.
- ❑ **bind(SocketAddress bindpoint)**  
binds the socket to a local address.
- ❑ **connect(SocketAddress endpoint)**  
connects this socket to the server.
- ❑ **connect(SocketAddress endpoint, int timeout)**  
connects this socket to the server with a specified timeout value.
- ❑ **InputStream getInputStream()**  
returns an input stream for this socket.
- ❑ **OutputStream getOutputStream()**  
returns an output stream for this socket.
- ❑ **close()**  
closes this socket.

23

## Multiplexing/Demultiplexing Issue

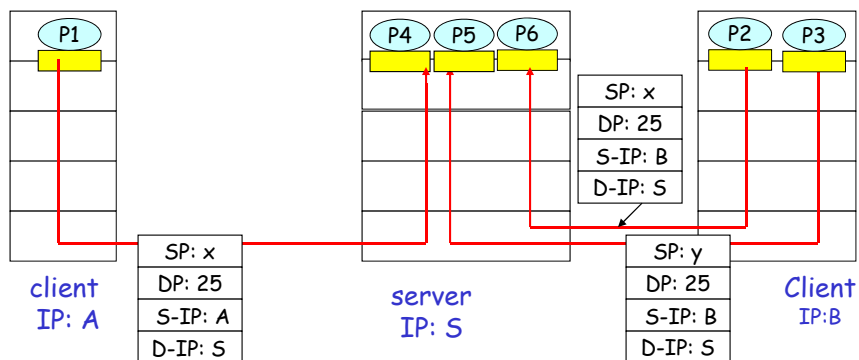


24

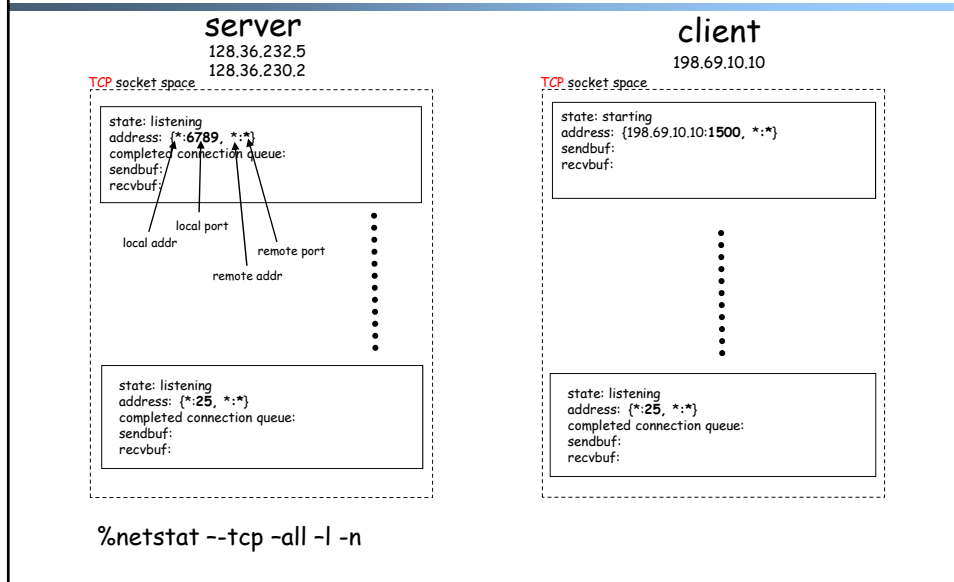
## TCP Connection-Oriented Demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
  
- recv host uses all four values to direct segment to appropriate socket
  - server can easily support many simultaneous TCP sockets: different connections/sessions are automatically separated into different sockets

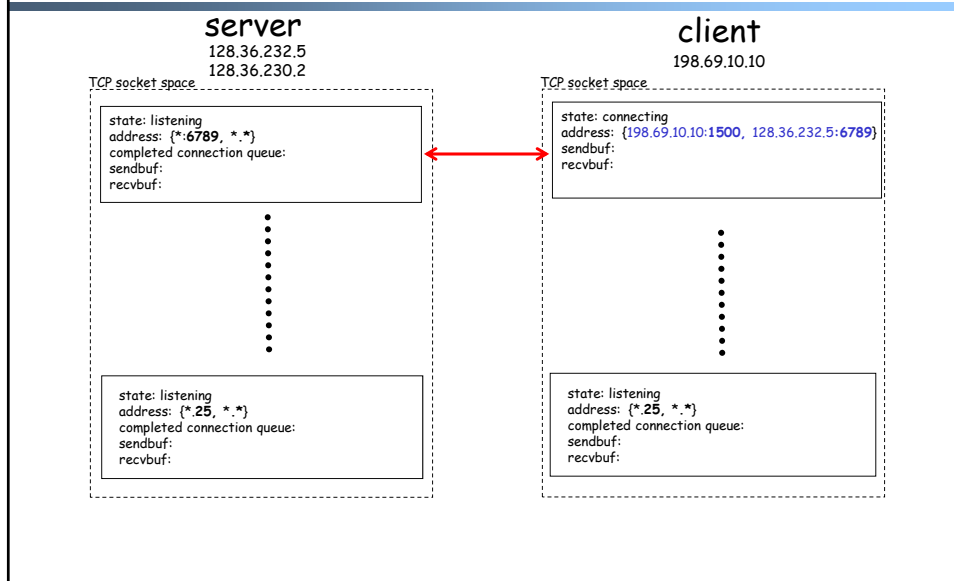
## Connection-Oriented Demux



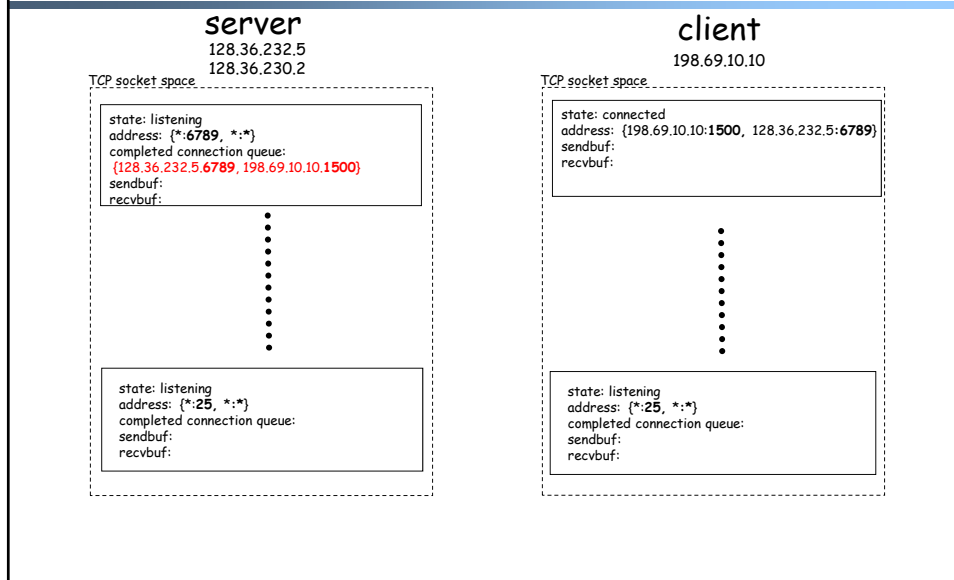
# Under the Hood: TCP Multiplexing



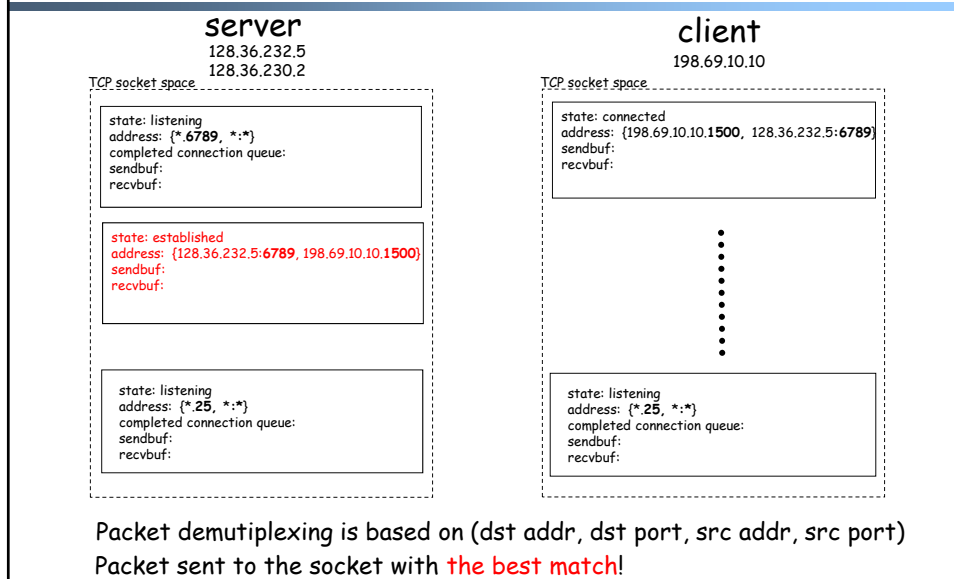
# Example: Client Initiates Connection



## Example: TCP Handshake Done



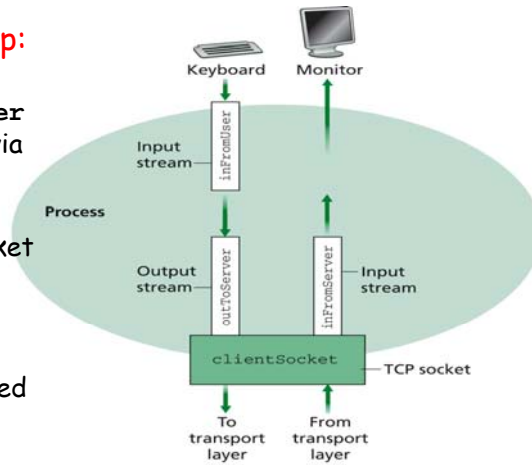
## Example: Server accept()



## TCP Example

### Example client-server app:

- 1) client reads line from standard input (`inFromUser` stream), sends to server via socket (`outToServer` stream)
- 2) server reads line from socket
- 3) server converts line to uppercase, sends back to client
- 4) client reads, prints modified line from socket (`inFromServer` stream)



<http://zoo.cs.yale.edu/classes/cs433/programming/examples-java-socket/TCP/>

31

## Example: Java client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {
```

```
    public static void main(String argv[]) throws Exception
    {
```

```
        String sentence;
        String modifiedSentence;
```

```
        Create input stream → BufferedReader inFromUser =
                                new BufferedReader(new InputStreamReader(System.in));
                                sentence = inFromUser.readLine();

        Create client socket, connect to server → Socket clientSocket = new Socket("server.name", 6789);

        Create output stream attached to socket → DataOutputStream outToServer =
                                                new DataOutputStream(clientSocket.getOutputStream());
```

## Example: Java client (TCP), cont.

```
    outToServer.writeBytes(sentence + '\n');

    BufferedReader inFromServer =
        new BufferedReader(new
            InputStreamReader(clientSocket.getInputStream()));

    modifiedSentence = inFromServer.readLine();

    System.out.println("FROM SERVER: " + modifiedSentence);

    clientSocket.close();
}
}
```

Send line to server

Create input stream attached to socket

Read line from server

## Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

    public static void main(String argv[]) throws Exception
    {
        String clientSentence;
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(6789);

        while(true) {
            Socket connectionSocket = welcomeSocket.accept();

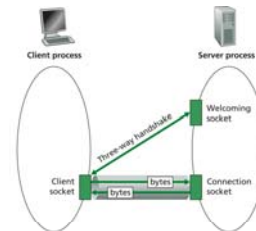
            BufferedReader inFromClient =
                new BufferedReader(new
                    InputStreamReader(connectionSocket.getInputStream()));

```

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket



## Example: Java server (TCP), cont

```
Read in line from socket → clientSentence = inFromClient.readLine();
Create output stream, attached to socket → capitalizedSentence = clientSentence.toUpperCase() + '\n';
DataOutputStream outToClient =
new DataOutputStream(connectionSocket.getOutputStream());
Write out line to socket → outToClient.writeBytes(capitalizedSentence);
}
}
}
End of while loop,
loop back and wait for
another client connection
```

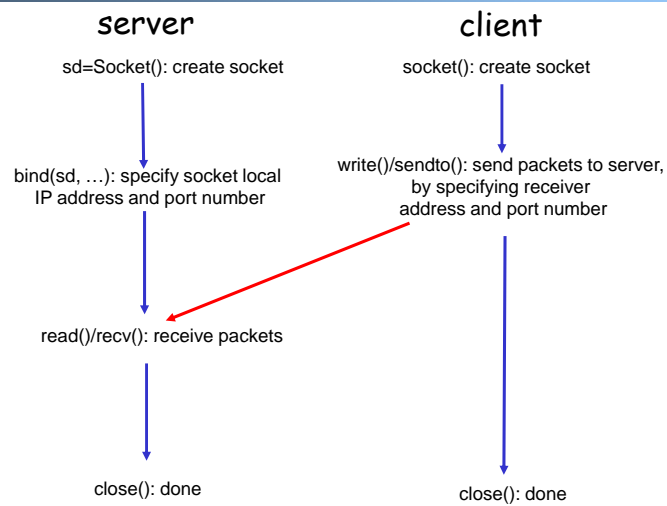
## Discussion

- How robust is the sample program?

## Backup: C/C++ Version

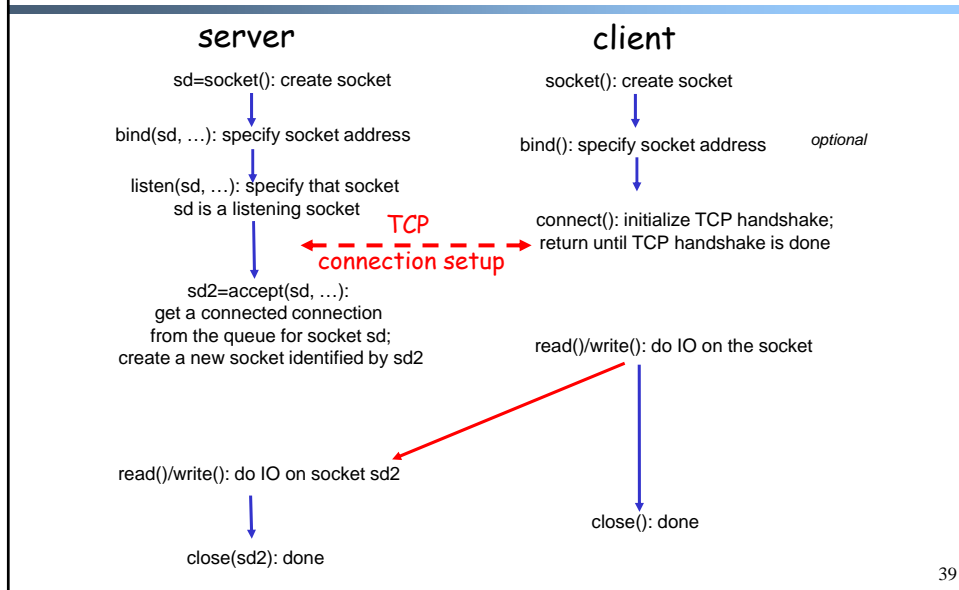
37

## Connectionless UDP: Big Picture



38

## Connection-oriented: Big Picture (C version)



## Unix Programming: Mechanism

- UNIX system calls and library routines (functions called from C/C++ programs)
- `%man 2 <function name>`
- A word on style: **check all return codes**

```
if ((code = syscall()) < 0) {  
    perror("syscall");  
}
```

## Creating Sockets

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- creates an endpoint for communication
- return value: -1 if an error occurs; otherwise the return value is a *descriptor* referencing the socket
- what are the possible outcomes of this system call?

41

## Creating Sockets: Parameters

- **domain**: address family (protocol family)
  - determine address structure
  - e.g. AF\_UNIX, AF\_INET, AF\_INET6
  - we will use AF\_INET only
- **type**: service of a socket
  - e.g. SOCK\_DGRAM provides unreliable, connectionless service
  - e.g. SOCK\_STREAM provides connection-oriented reliable byte-stream service

42

## Creating Sockets: Parameters (cont.)

- *protocol*: specifies particular protocol
  - Usually already defined by domain and type (e.g. TCP for AF\_INET and SOCK\_STREAM; UDP for AF\_INET and SOCK\_DGRAM)
  - we will use 0 (default protocol)

- Example

```
if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) {  
    perror("socket");  
    exit(1);  
}
```

43

## Binding the Address for a Socket

```
#include <sys/types.h>  
#include <sys/socket.h>  
int bind(int sd, struct sockaddr *my_addr,  
        socklen_t addrlen);
```

- assigns the local address of a socket
- return value: -1 if an error occurs; otherwise 0
- what are the possible outcomes of this system call?

44

## Socket Address

- Several types of addresses
- We will use `sockaddr_in` (`<netinet/in.h>`)

```
struct sockaddr_in {
    sa_family_t    sin_family; /*AF_INET*/
    uint16_t       sin_port; /* network order*/
    struct in_addr sin_addr;
};
struct in_addr {
    uint32_t       s_addr; /* network order*/
};
```

Two types of byte ordering: little endian, and big endian

45

## Internet Addresses and Ports

- `sin_port`
  - 16 bits
  - 0-1024 reserved for system
  - well-known ports are important
  - If you specify 0, the OS picks a port
- `s_addr`
  - 32 bits
  - `INADDR_ANY` for any local interface address

46

## Internet Addresses and Ports: Example

```
struct sockaddr_in myaddr;

bzero( (char*)myaddr, sizeof(myaddr) );
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(80); /* bind to HTTP port*/
myaddr.sin_addr.s_addr = htos(INADDR_ANY); /* any address*/

if ( (bind(sockfd, (struct sockaddr*)&myaddr, sizeof(myaddr)) < 0) ) {
    perror("bind");
    exit(1);
}
```

47

## Set a Socket in the Listening State (Server)

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sd, int backlog);
```

- Specify the willingness to accept new connection
- *backlog*: specify the number of pending connections
- return value: -1 if an error occurs; otherwise 0
- what are the possible outcomes of this system call?

48

## Initialize Connection Setup (Client)

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

- For `SOCK_STREAM`, initialize connection to the server; for `SOCK_DGRAM`, just set the destination address and set the socket in connected state
- return value: -1 if an error occurs; otherwise 0
- what are the possible outcomes of this system call?

49

## Accept a Connection (Server)

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sd, struct sockaddr *peer_addr,
           socklen_t addrlen);
```

- remove the first connection from the pending connection queue, create a new socket in connected state, the original `sd` is not changed and still in listening state
- return value: -1 if an error occurs; otherwise **the descriptor of the newly connected socket**
- what are the possible outcomes of this system call?

50

## Read/Write to a Socket

- ❑ read()/write() of the file interface for connected-oriented
- ❑ Socket specific system call
  - send()/sendto()/sendmsg()
  - recv()/recvfrom()/recvmsg()

51

## Read from a socket by using read()

```
#include <unistd.h>
```

```
ssize_t read(int sockfd, void *buf, size_t  
count);
```

- read **up to** count from the socket
- return value: -1 if an error occurs; 0 if end of file; otherwise number of bytes read
- what are the possible outcomes of this system call?

52

## Write to a socket by using write()

```
#include <unistd.h>
ssize_t write(int sockfd, const void *buf,
              size_t count);
```

- write **up to** count to the socket
- return value: -1 if an error occurs; otherwise number of bytes write
- what are the possible outcomes of this system call?

53

## Send to a Socket

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int sd, const void *msg, size_t len, int flags);
int sendto(int sd, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);
int sendmsg(int sd, const struct msghdr *msg, int flags)
```

- return value: -1 if an error occurs; otherwise the number of bytes sent

54

## sendmsg(): scatter and collect

```
struct msghdr {
    void          *msg_name;    // peer address
    socklen_t     msg_namelen; // address length
    struct iovec *msg_iov;     // io vector
    size_t        msg_iovlen;  // io vector length
    void          *msg_control;
    socklen_t     msg_controllen;
    int           msg_flags;
};
struct iovec {
    void          *iov_base;
    size_t        iov_len;
};
```

55

## Receive from a Socket

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int sd, void *buf, size_t len, int flags);
int recvfrom(int sd, void *buf, size_t len, int flags,
             struct sockaddr *from, socklen_t fromlen);
int recvmsg(int sd, struct msghdr *msg, int flags);
```

- return value: -1 if an error occurs; otherwise the number of bytes received

56

## Close a Socket

```
#include <unistd.h>
int close(int sd);
- return value: -1 if an error occurs; otherwise 0

#include <sys/socket.h>
int shutdown(int sd, int how);
- how: if 0, no further receives; if 1, no further sends;
  if 2, no further sends or receives
- return value: -1 if an error occurs; otherwise 0
```

57

## Support Routines: Address from and to String Formats

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

int inet_pton(int af, const char *src, void *dst);
  return value: positive if successful

const char *inet_ntop(int af, const void *src, char
  *dst, size_t cnt);
  return value: NULL is error
```

Note: inet\_addr() deprecated!

58

## Support Routines: Network/Host Order

```
#include <netinet/in.h>
```

```
unsigned long int htonl(unsigned long int hostlong);  
unsigned short int htons(unsigned short int hostshort);
```

```
unsigned long int ntohl(unsigned long int networklong);  
unsigned short int ntohs(unsigned short int  
networkshort);
```

59

## DNS Service

```
#include <netdb.h>  
extern int h_errno;
```

```
struct hostent *gethostbyname(const char *name);
```

```
Struct hostent {  
    char *h_name;        // official name  
    char **h_aliases;    // a list of aliases  
    int h_addrtype;  
    int h_length;  
    char **h_addr_list;  
}  
#define h_addr h_addr_list[0]  
- return value: NULL if fails
```

60

## Summary: Socket Programming

- ❑ %man 2 <name>
- ❑ System calls (functions)
  - int socket(int domain, int type, int protocol);
  - int bind(int sd, struct sockaddr \*my\_addr, socklen\_t addrlen);
  - int listen(int sd, int backlog);
  - int connect(int sd, const struct sockaddr \*serv\_addr, socklen\_t addrlen);
  - int accept(int sd, struct sockaddr \*peer\_addr, socklen\_t addrlen);
  
  - read(int sockfd, void \*buf, size\_t count);
  - write(int sockfd, const void \*buf, size\_t count)

61

## Java Stream

- ❑ Input/output stream
  - sock.getInputStream();
  - sock.getOutputStream();
- ❑ public abstract class OutputStream
- ❑ public abstract void write(int b) throws IOException  
public void write(byte[] data) throws IOException  
public void write(byte[] data, int offset, int length) throws IOException  
public void flush( ) throws IOException  
public void close( ) throws IOException
- ❑ Filter stream

62

## Output

- ❑ Streams can also be buffered in software, directly in the Java code as well as in the network hardware. Typically, this is accomplished by chaining a `BufferedOutputStream` or a `BufferedWriter` to the underlying stream
- ❑ Consequently, if you are done writing data, it's important to flush the output stream. For example, suppose you've written a 300-byte request to an HTTP 1.1 server that uses HTTP Keep-Alive. You generally want to wait for a response before sending any more data. However, if the output stream has a 1,024-byte buffer, the stream may be waiting for more data to arrive before it sends the data out of its buffer. No more data will be written onto the stream until the server response arrives, but the response is never going to arrive because the request hasn't been sent yet!

63

## Input

- ❑ `public abstract class InputStream`  
This class provides the fundamental methods needed to read data as raw bytes. These are:
- ❑ `public abstract int read( ) throws IOException`  
`public int read(byte[] input) throws IOException`  
`public int read(byte[] input, int offset, int length) throws IOException`  
`public long skip(long n) throws IOException`  
`public int available( ) throws IOException`  
`public void close( ) throws IOException`

64

## Input

- a read attempt won't completely fail but won't completely succeed, either. Some of the requested bytes may be read, but not all of them. For example, you may try to read 1,024 bytes from a network connection, when only 512 have actually arrived from the server; the rest are still in transit. They'll arrive eventually, but they aren't available at this moment. To account for this, the multibyte read methods return the number of bytes actually read. For example, consider this code fragment:
  - `byte[] input = new byte[1024]; int bytesRead = in.read(input);`

65

## Input

- All three `read( )` methods return -1 to signal the end of the stream. If the stream ends while there's still data that hasn't been read, the multibyte `read( )` methods return the data until the buffer has been emptied. The next call to any of the `read( )` methods will return -1. The -1 is never placed in the array.
- If you do not want to wait until all the bytes you need are immediately available, you can use the `available( )` method to determine how many bytes can be read without blocking. This returns the minimum number of bytes you can read. You may in fact be able to read more, but you will be able to read at least as many bytes as `available( )` suggests. For example:

66

## Print Stream

---

- **PrintStream is evil and network programmers should avoid it like the plague!**
  - **Println() plat-form dependent**
    - Using `println()` makes it easy to write a program that works on Windows but fails on Unix and the Mac.
  - The second problem is that `PrintStream` assumes the default encoding of the platform on which it's running
  - The third problem is that `PrintStream` eats all exceptions. This makes `PrintStream` suitable for textbook programs such as `HelloWorld`, since simple console output can be taught without burdening students with first learning about exception handling and all that implies. However, network connections are much less reliable than the console.

67

## PushBackInputStream

---

68

## Data Stream

---

- ❑ `DataInputStream.getLine()` is depreciated
- ❑ Know the format

69

- ❑ Compressing Stream
- ❑ Digest streams
- ❑ Encryption streams

70

## Reader/Writer

- The exception to the rule of similarity is `ready()`, which has the same general purpose as `available()` but not quite the same semantics, even modulo the byte-to-char conversion. Whereas `available()` returns an `int` specifying a minimum number of bytes that may be read without blocking, `ready()` only returns a `boolean` indicating whether the reader may be read without blocking. The problem is that some character encodings, such as UTF-8, use different numbers of bytes for different characters. Thus, it's hard to tell how many characters are waiting in the network or filesystem buffer without actually reading them out of the buffer.

71

- The `PrintWriter` class is a replacement for Java 1.0's `PrintStream` class that properly handles multibyte character sets and international text. Sun originally planned to deprecate `PrintStream` in favor of `PrintWriter` but backed off when it realized this step would invalidate too much existing code, especially code that depended on `System.out`. Nonetheless, new code should use `PrintWriter` instead of `PrintStream`.

72