



IIT KHARAGPUR



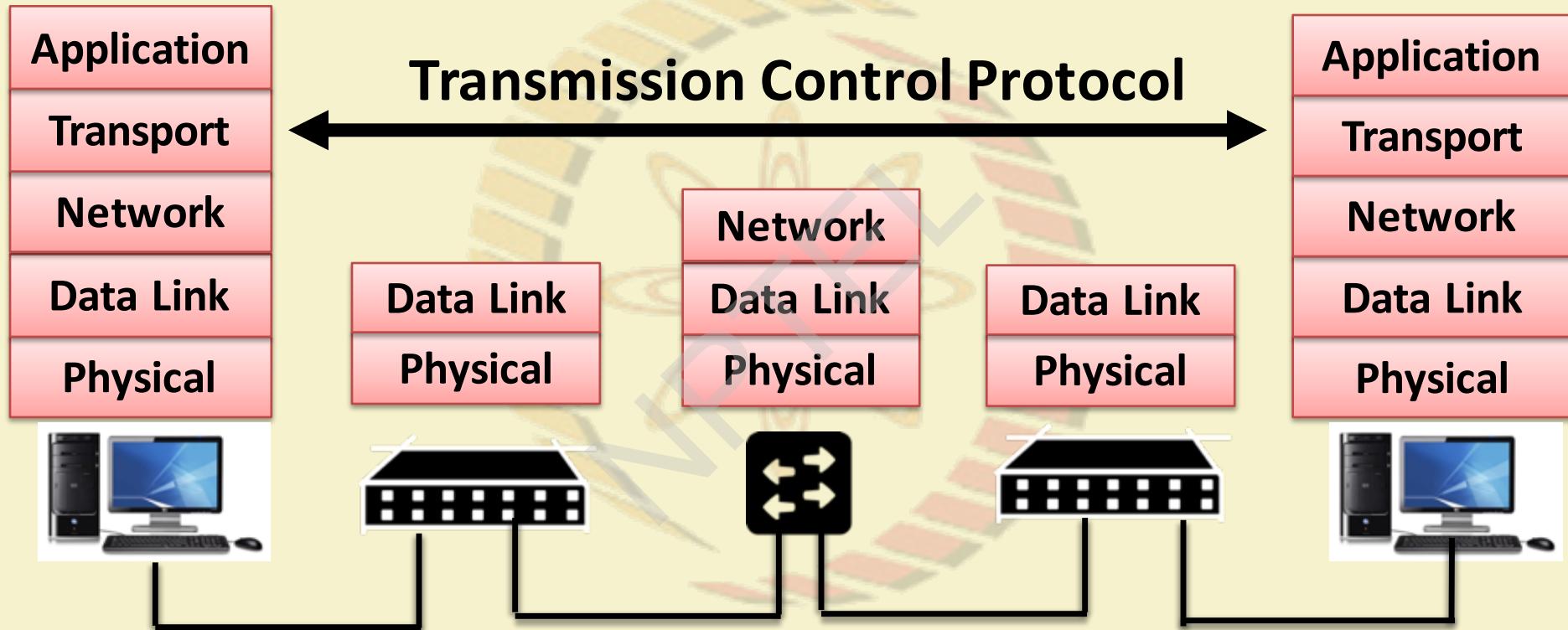
NPTEL ONLINE
CERTIFICATION COURSES

COMPUTER NETWORKS AND INTERNET PROTOCOLS

SOUMYA K GHOSH
COMPUTER SCIENCE AND ENGINEERING,
IIT KHARAGPUR

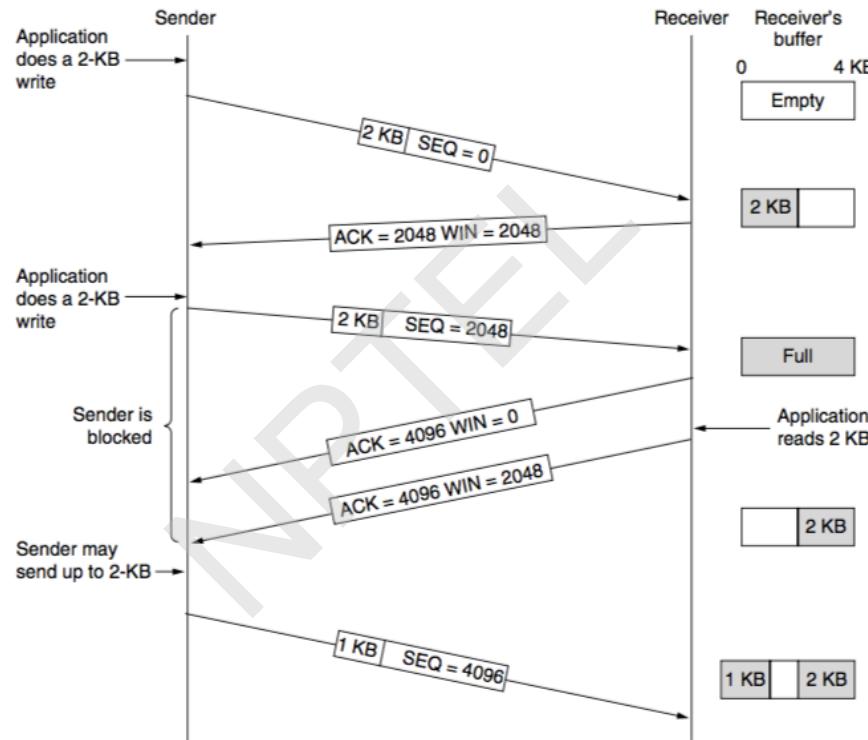
SANDIP CHAKRABORTY
COMPUTER SCIENCE AND ENGINEERING,
IIT KHARAGPUR

Transmission Control Protocol III (Flow Control)



TCP Sliding Window

Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES

Delayed Acknowledgements

- Consider a telnet connection, that reacts on every keystroke.
- In the worst case, whenever a character arrives at the sending TCP entity, TCP creates a 21 byte TCP segment, 20 bytes of header and 1 byte of data. For this segment, another ACK and window update is sent when the application reads that 1 byte. This results in a huge wastage of bandwidth.

Delayed Acknowledgements

- **Delayed acknowledgements:** Delay acknowledgement and window updates for up to 500 msec in the hope of receiving few more data packets within that interval.
- **However, the sender can still send multiple short data segments.**

Nagle's Algorithm

- When data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged.
- Then send all buffered data in one TCP segment and start buffering again until the next segment is acknowledged.
 - Only one short packet can be outstanding at any time.

Nagle's Algorithm

- **Do we want Nagle's Algorithm all the time?**
- **Nagle's Algorithm and Delayed Acknowledgement**
 - Receiver waits for data and sender waits for acknowledgement – results in starvation



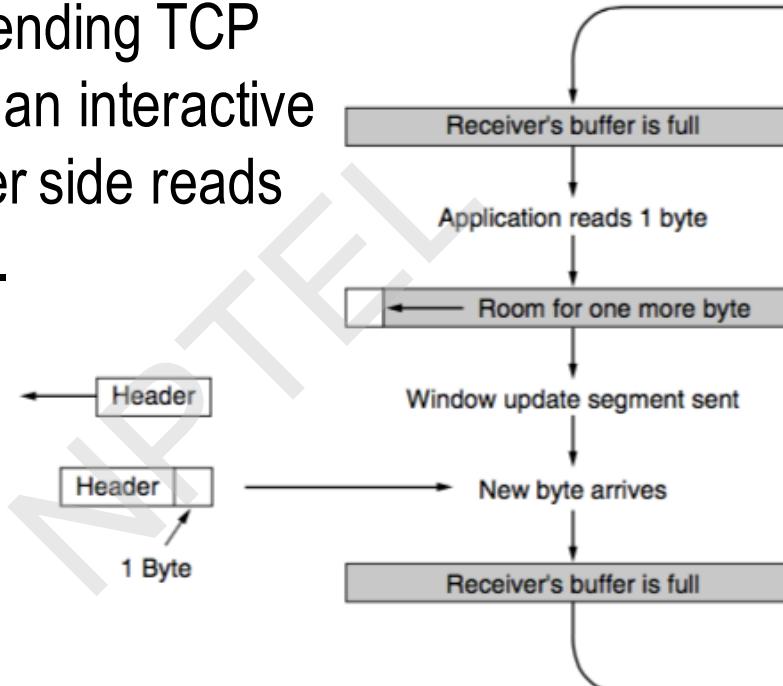
IIT KHARAGPUR



NPTEL
NPTEL ONLINE
CERTIFICATION COURSES

Silly Window Syndrome

- Data are passed to the sending TCP entity in large blocks, but an interactive application on the receiver side reads data only 1 byte at a time.

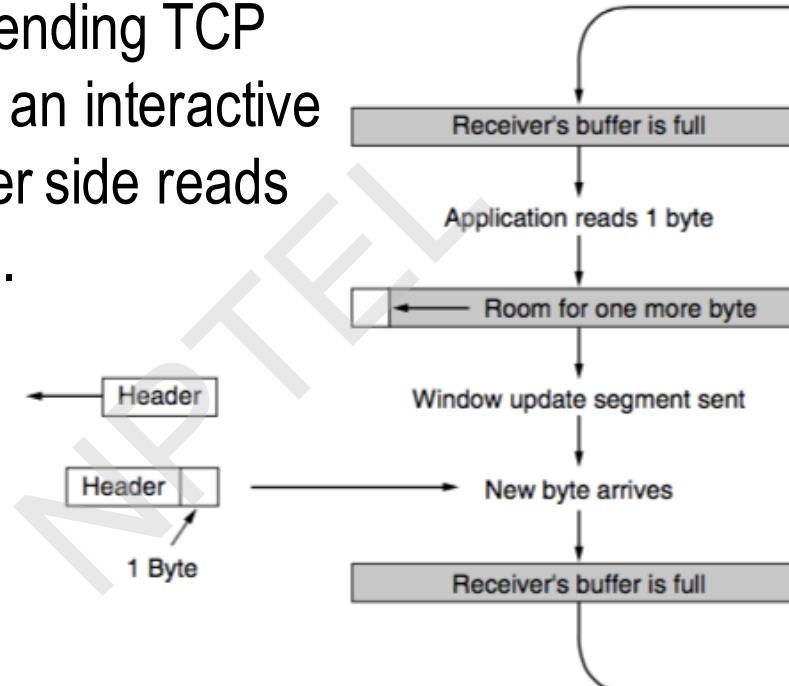


Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell

Silly Window Syndrome

- Data are passed to the sending TCP entity in large blocks, but an interactive application on the receiver side reads data only 1 byte at a time.

Clark's solution: Do not send window update for 1 byte.
Wait until sufficient space is available at the receiver buffer.



Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell

Handling Short Segments – Sender and Receiver Together

- Nagle's algorithm and Clark's solution to silly window syndrome are complementary
- **Nagle's algorithm:** Solve the problem caused by the sending application delivering data to TCP a byte at a time
- **Clark's solution:** Receiving application fetching the data up from TCP a byte at a time
- Exception: The PSH flag is used to inform the sender to create a segment immediately without waiting for more data

Handling Out of Order in TCP

- TCP buffers out of order segments and forward a duplicate acknowledgement to the sender.
- **Acknowledgement in TCP – Cumulative acknowledgement**

Handling Out of Order in TCP

- Receiver has received bytes 0, 1, 2, _, 4, 5, 6, 7
 - TCP sends a cumulative acknowledgement with ACK number 2, acknowledging everything up to byte 2
 - Once 4 is received, a duplicate ACK with ACK number 3 (next expected byte) is forwarded – **triggers congestion control**
 - After timeout, sender retransmits byte 3
 - Once byte 3 is received, it can send another cumulative ACK with ACK number 8 (next expected byte)



IIT KHARAGPUR



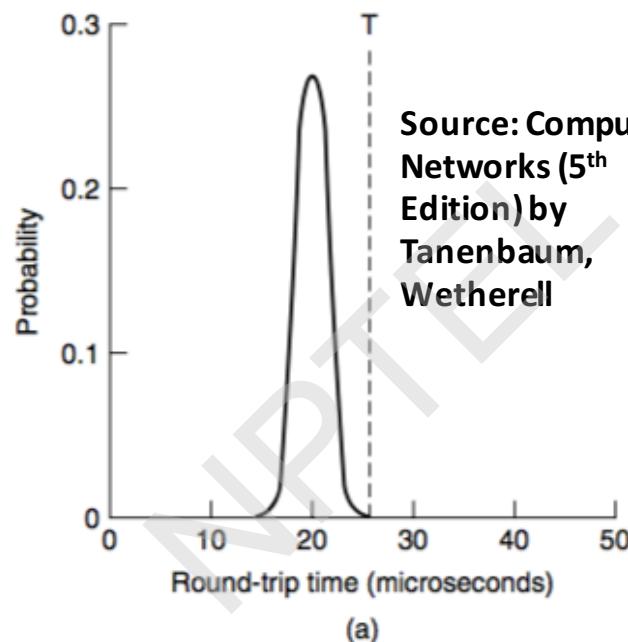
NPTEL
NPTEL ONLINE
CERTIFICATION COURSES

TCP Timer Management

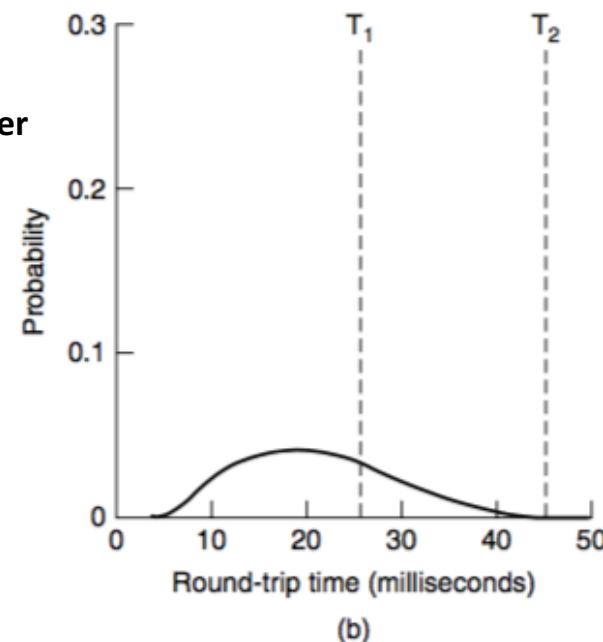
- **TCP Retransmission Timeout (RTO):** When a segment is sent, a retransmission timer is started
 - If the segment is acknowledged before the timer expires, the timer is stopped
 - If the timer expires before the acknowledgement comes, the segment is retransmitted
- **What can be an ideal value of RTO ?**
- **Possible solution:** Estimate RTT, and RTO is some positive multiples of RTT
- **RTT estimation is difficult for transport layer – why?**

RTT at Data Link Layer vs RTT at Transport Layer

Use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance.



Data Link Layer



Transport Layer



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES

RTT Estimation at the Transport Layer

Jacobson's algorithm (1988) - used in TCP

- For each connection, TCP maintains a variable, **SRTT (smoothed Round Trip Time)** – best current estimate of the round trip time to the destination
- When a segment is sent, a timer is started (both to see how long the acknowledgement takes and also to trigger a retransmission if it takes too long)
- If the ACK gets back – measure the time (say, R)
- Update SRTT as follows
$$SRTT = \alpha SRTT + (1 - \alpha)R \text{ (Exponentially Weighted Moving Average)}$$
- α is a smoothing factor that determines how quickly the old values are forgotten.
Typically $\alpha = 7/8$

Problem with EWMA

- Even given a good value of SRTT, choosing a suitable RTO is nontrivial.
- Initial implementation of TCP used $RTO = 2SRTT$
- Experience showed that a constant value was too inflexible, because it failed to respond when the **variance went up (RTT fluctuation is high)**
– happens normally at high load
- Consider variance of RTT during RTO estimation.

RTO Estimation

- Update RTT variation (*RTTVar*) as follows.

$$RTTVar = \beta RTTVar + (1 - \beta)|SRTT - R|$$

- Typically $\beta = \frac{3}{4}$

- RTO is estimated as follows,

$$RTO = SRTT + 4 \times RTTVar$$

- Why 4 ?

- Somehow arbitrary
 - Jacobson's paper is full of clever tricks – use integer addition, subtraction and shift – computation is lightweight



IIT KHARAGPUR



NPTEL
NPTEL ONLINE
CERTIFICATION COURSES

Karn's Algorithm

- How will you get the RTT estimation, when a segment is lost and retransmitted again?
- **Karn's algorithm:**
 - Do not update estimates on any segments that has been retransmitted
 - The timeout is doubled each successive retransmission until the segments gets through the first time



IIT KHARAGPUR



NPTEL
NPTEL ONLINE
CERTIFICATION COURSES

Other TCP Timers

- **Persistent TCP Timer:** Avoid deadlock when receiver buffer is announced as zero
 - After the timer goes off, sender forwards a probe packet to the receiver to get the updated window size
- **Keepalive Timer:** Close the connection when a connection has been idle for a long duration
- **TCP TIME_WAIT:** Wait before closing a connection – twice the packet lifetime



thank you!



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



IIT KHARAGPUR



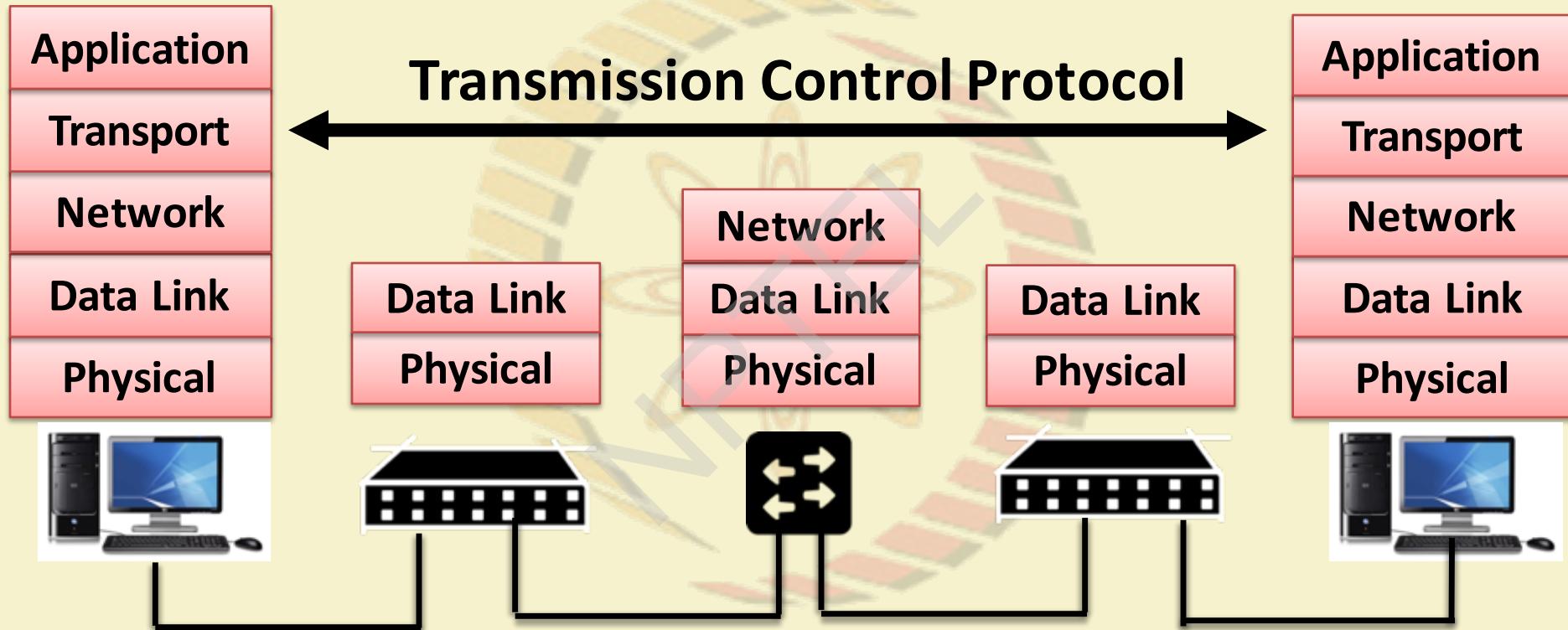
NPTEL ONLINE
CERTIFICATION COURSES

COMPUTER NETWORKS AND INTERNET PROTOCOLS

SOUMYA K GHOSH
COMPUTER SCIENCE AND ENGINEERING,
IIT KHARAGPUR

SANDIP CHAKRABORTY
COMPUTER SCIENCE AND ENGINEERING,
IIT KHARAGPUR

Transmission Control Protocol IV (Congestion Control)



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES

TCP Congestion Control

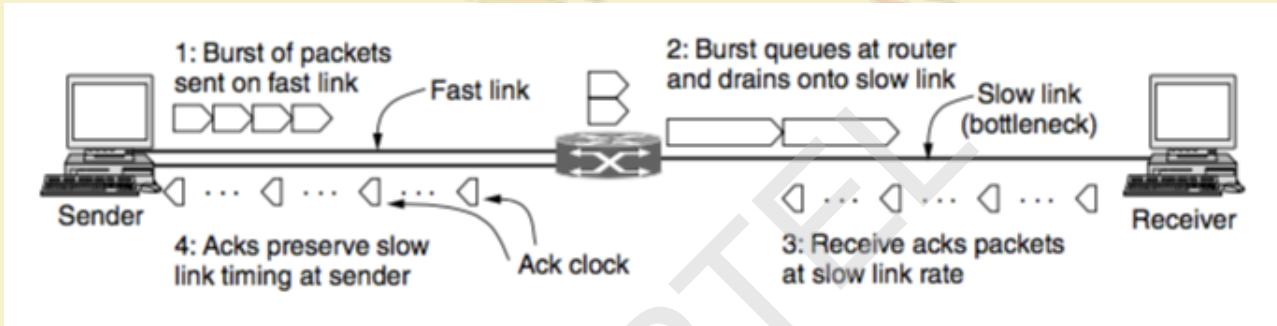
- Based on implementation of AIMD using a window and with packet loss as the binary signal
- TCP maintains a **Congestion Window (CWnd)** – number of bytes the sender may have in the network at any time
- **Sending Rate = Congestion Window / RTT**
- **Sender Window (SWnd) = Min (CWnd, RWnd)**
- RWnd – Receiver advertised window size

1986 Congestion Collapse

- In 1986, the growing popularity of Internet led to the first occurrence of congestion collapse – a prolonged period during which goodput dropped precipitously (more than a factor of 100)
- Early TCP Congestion Control algorithm – Effort by Van Jacobson (1988)
- **Challenge for Jacobson** – Implement congestion control without making much change in the protocol (made it instantly deployable)
- **Packet loss is a suitable signal for congestion – use timeout to detect packet loss. Tune CWnd based on the observation from packet loss**

Adjust CWnd based on AIMD

- One of the most interesting ideas – use ACK for clocking



Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell

- ACK returns to the sender at about the rate that packets can be sent over the slowest link in the path.
- Trigger CWnd adjustment based on the rate at which ACK are received.

Increase Rate Exponentially at the Beginning – The Slow Start

- AIMD rule will take a very long time to reach a good operating point on fast networks if the CWnd is started from a small size.
- A 10 Mbps link with 100 ms RTT
 - Appropriate CWnd = BDP = 1 Mbit
 - 1250 byte packets -> 100 packets to reach BDP
 - CWnd starts at 1 packet, and increased 1 packet at every RTT
 - 100 RTTs are required 10 sec before the connection reaches to a moderate rate

Increase Rate Exponentially at the Beginning – The Slow Start

- **Slow Start - Exponential increase of rate to avoid slow convergence**
 - Rate is not slow at all ! ☺
 - CWnd is doubled at every RTT



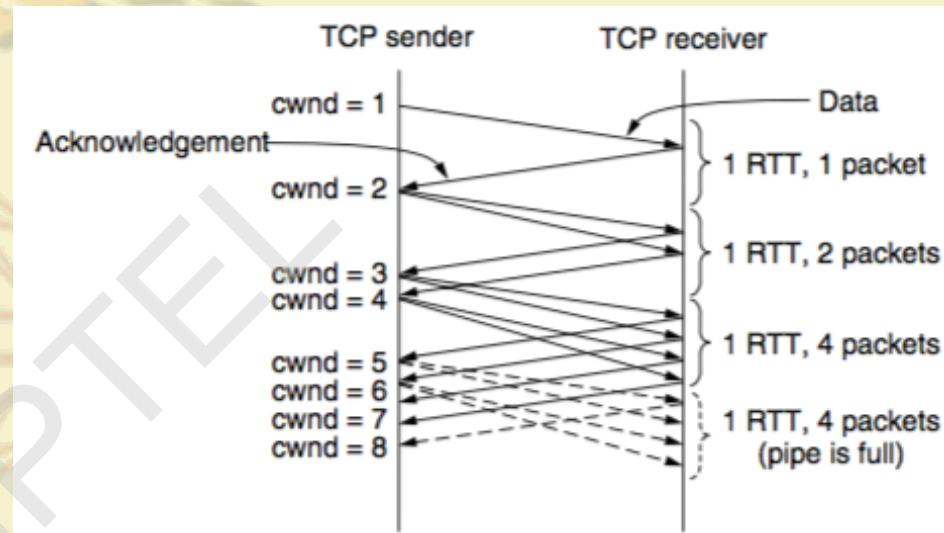
IIT KHARAGPUR



NPTEL
NPTEL ONLINE
CERTIFICATION COURSES

TCP Slow Start

- Every ACK segment allows two more segments to be sent
- For each segment that is acknowledged before the retransmission timer goes off, the sender adds one segment's worth of bytes to the congestion window.



Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell

Slow Start Threshold

- Slow start causes exponential growth, eventually it will send too many packets into the network too quickly.
- To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold (ssthresh)**.
- Initially ssthresh is set to BDP (or arbitrarily high), the maximum that a flow can push to the network.
- Whenever a packet loss is detected by a RTO, the ssthresh is set to be half of the congestion window

Additive Increase (Congestion Avoidance)

- Whenever ssthresh is crossed, TCP switches from slow start to additive increase.
- Usually implemented with an partial increase for every segment that is acknowledged, rather than an increase of one segment per RTT.
- A common approximation is to increase Cwnd for additive increase as follows:

$$CWnd = Cwnd + \frac{MSS \times MSS}{CWnd}$$

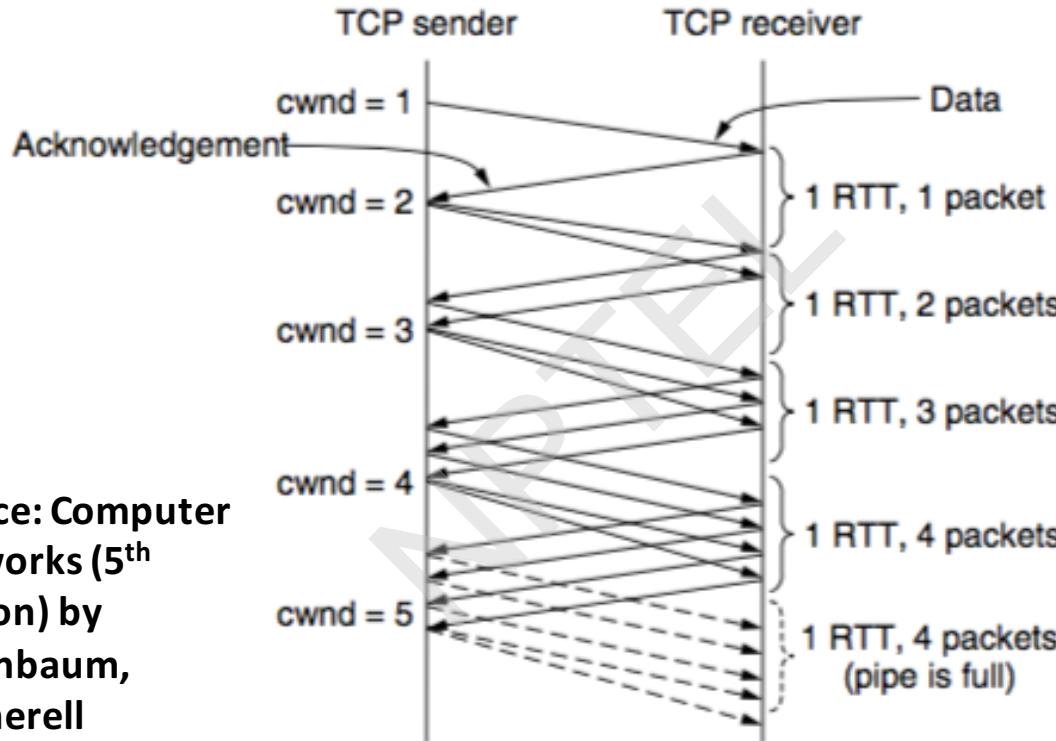


IIT KHARAGPUR



NPTEL
NPTEL ONLINE
CERTIFICATION COURSES

Additive Increase – Packet Wise Approximation



Source: Computer
Networks (5th
Edition) by
Tanenbaum,
Wetherell



IIT KHARAGPUR



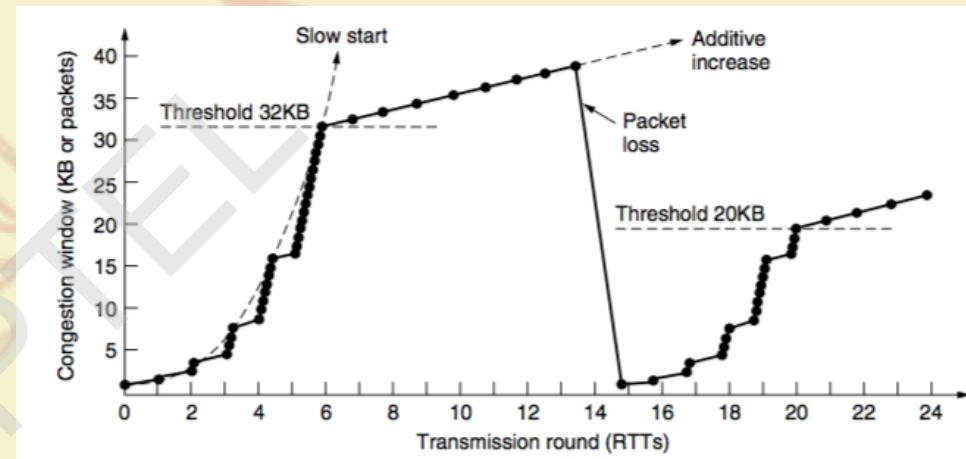
NPTEL ONLINE
CERTIFICATION COURSES

Triggering an Congestion

- Two ways to trigger a congestion notification in TCP – (1) RTO, (2) Duplicate ACK
- **RTO:** A sure indication of congestion, however time consuming
- **Duplicate ACK:** Receiver sends a duplicate ACK when it receives out of order segment
 - A loose way of indicating congestion
 - TCP arbitrarily assumes that THREE duplicate ACKs (DUPACKs) imply that a packet has been lost – triggers congestion control mechanism
 - The identity of the lost packet can be inferred – the very next packet in sequence
 - Retransmit the lost packet and trigger congestion control

Fast Retransmission – TCP Toho

- Use THREE DUPACK as the sign of congestion
- Once 3 DUPACKs have been received,
 - Retransmit the lost packet (**fast retransmission**) – takes one RTT
 - Set ssthresh as half of the current CWnd
 - Set CWnd to 1 MSS



Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell

Fast Recovery – TCP Reno

- Once a congestion is detected through 3 DUPACKs, do TCP really need to set CWnd = 1 MSS ?
- DUPACK means that **some segments are still flowing in the network** – a signal for temporary congestion, but not a prolonged one
- Immediately transmit the lost segment (**fast retransmit**), then transmit additional segments based on the DUPACKs received (**fast recovery**)

Fast Recovery – TCP Reno

Fast recovery:

1. set ssthresh to one-half of the current congestion window. Retransmit the missing segment.
2. set cwnd = ssthresh + 3.
3. Each time another duplicate ACK arrives, set cwnd = cwnd + 1. Then, send a new data segment if allowed by the value of cwnd.
4. Once receive a new ACK (an ACK which acknowledges all intermediate segments sent between the lost packet and the receipt of the first duplicate ACK), exit fast recovery.
 - This causes setting cwnd to ssthresh (the ssthresh in step 1). Then, continue with linear increasing due to congestion avoidance algorithm.

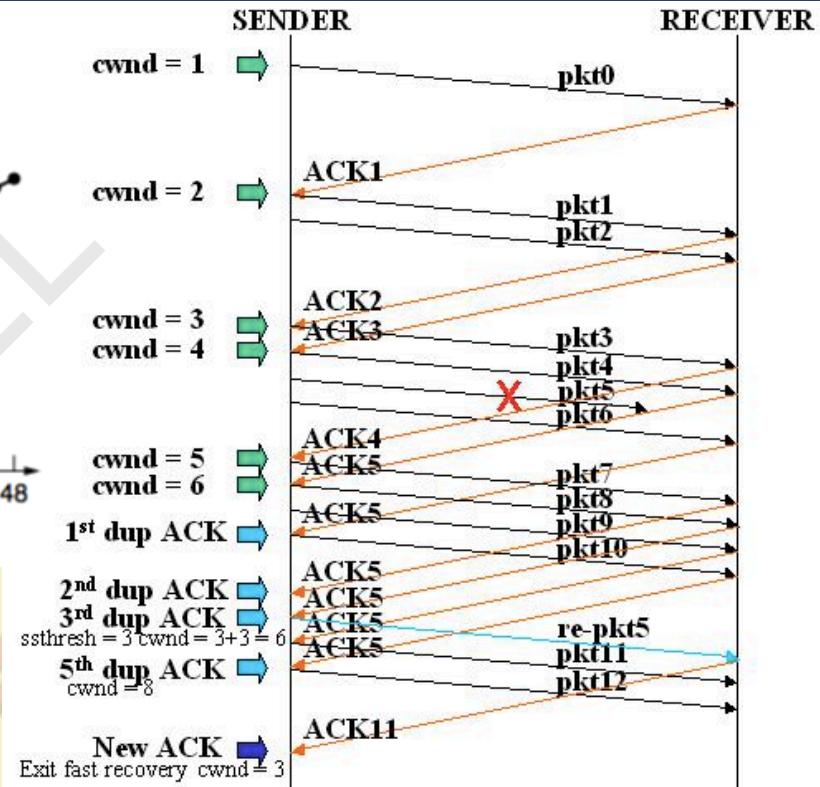
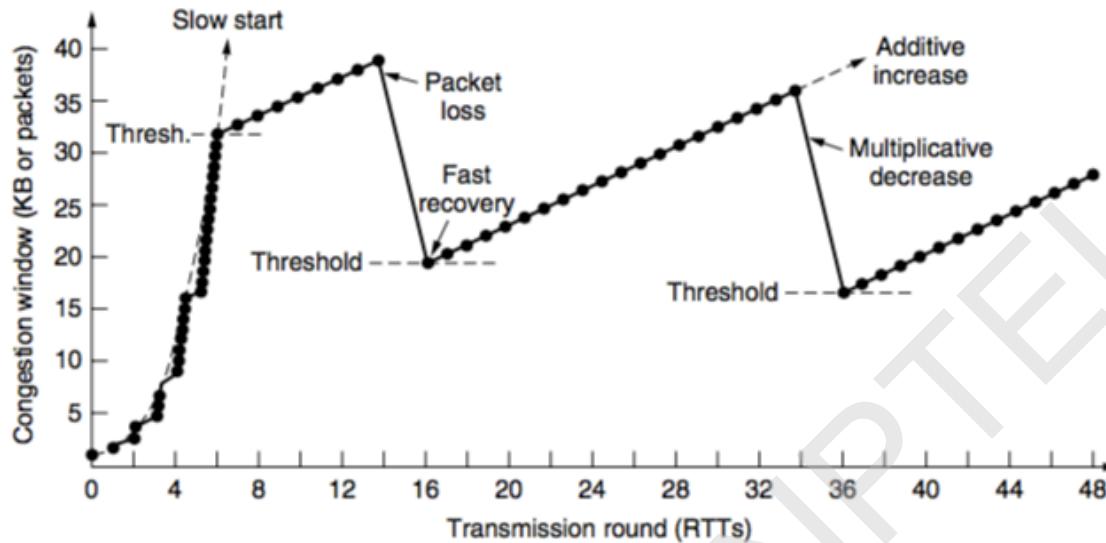


IIT KHARAGPUR



NPTEL
NPTEL ONLINE
CERTIFICATION COURSES

Fast Recovery – TCP Reno





thank you!



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES



IIT KHARAGPUR



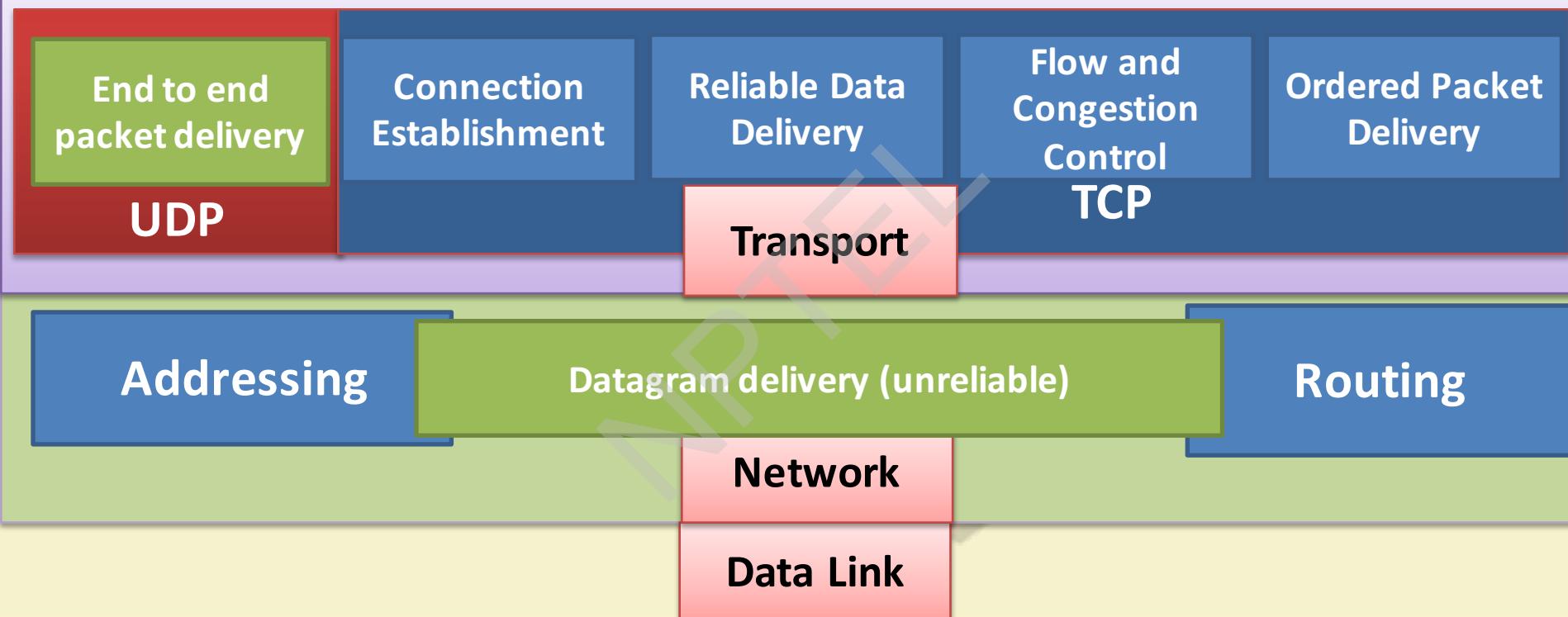
NPTEL ONLINE
CERTIFICATION COURSES

COMPUTER NETWORKS AND INTERNET PROTOCOLS

SOUMYA K GHOSH
COMPUTER SCIENCE AND ENGINEERING,
IIT KHARAGPUR

SANDIP CHAKRABORTY
COMPUTER SCIENCE AND ENGINEERING,
IIT KHARAGPUR

User Datagram Protocol (UDP)



IIT KHARAGPUR



NPTEL
NPTEL ONLINE
CERTIFICATION COURSES

UDP

Features

- **Simple Protocol**
 - A wrapper on top of IP layer
- **Fast**
 - No flow control, no congestion control

Uses

- **Provide performance**
 - No data holding in buffer like TCP
- **Short and sweet**
 - Have no overhead
 - Suitable for short messages

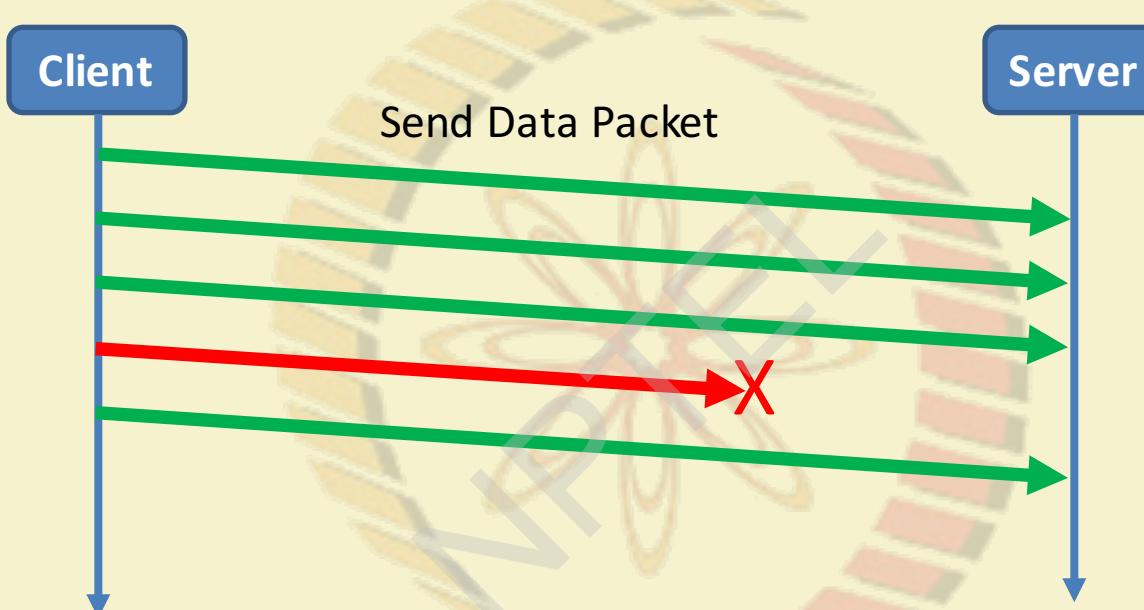


IIT KHARAGPUR



NPTEL
NPTEL ONLINE
CERTIFICATION COURSES

UDP is connection less and Unreliable



- **No connection establishment, No reliability, No acknowledgement**

UDP Header



IIT KHARAGPUR



NPTEL
NPTEL ONLINE
CERTIFICATION COURSES

Applications

Protocol	Keyword	Comment
DNS	domain	Simple request response messaging system is faster than TCP
BOOTP/DHCP	Network configuration	Short messaging helps faster configuration
TFTP	File transfer	Lightweight file transfer protocol to transfer small files
SNMP	Network management	Simple UDP protocol easily cut through congestion than TCP
QUIC	Advance transport protocol	UDP provide direct access to IP while TCP can't

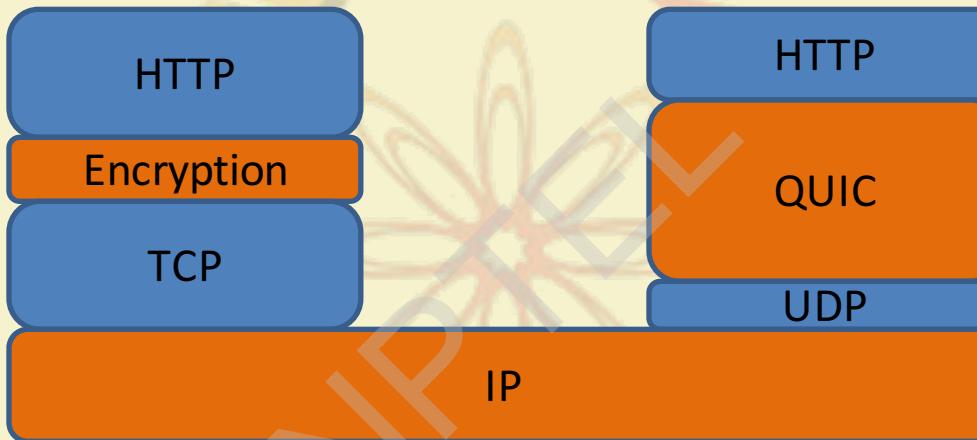


IIT KHARAGPUR

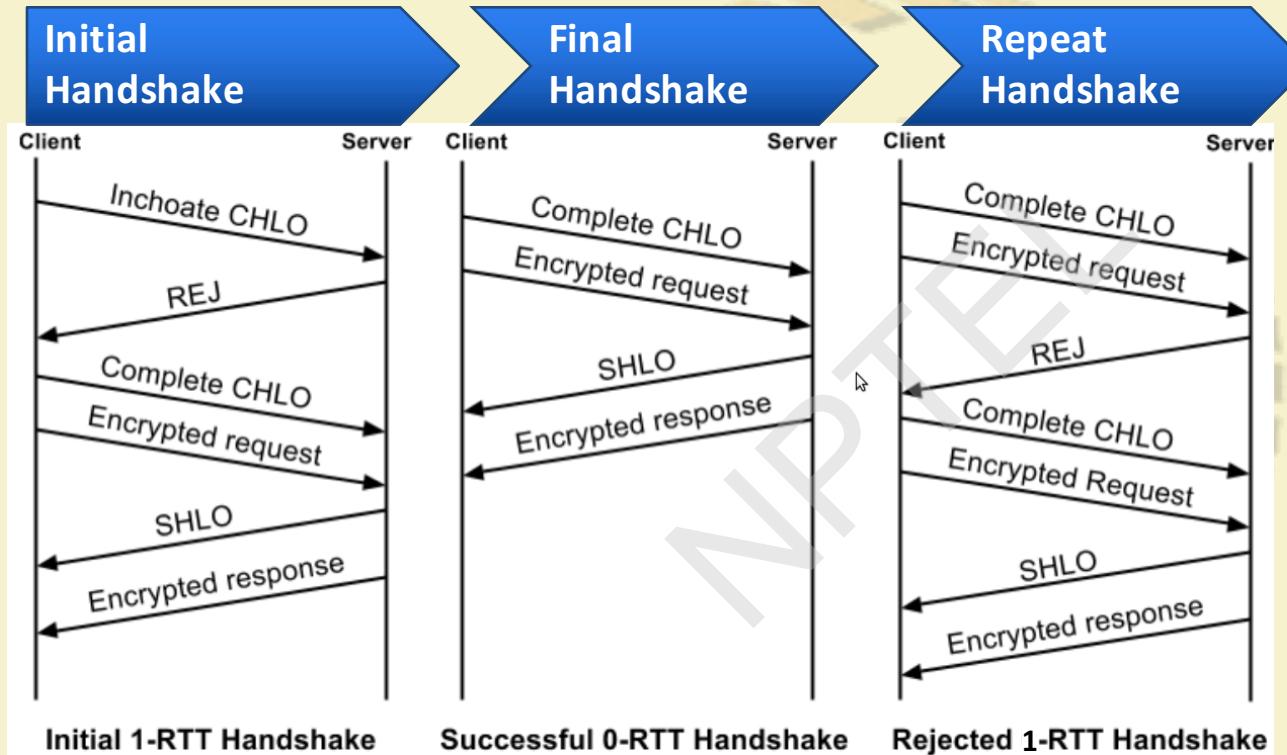


NPTEL ONLINE
CERTIFICATION COURSES

Quick UDP Internet Connections (QUIC)

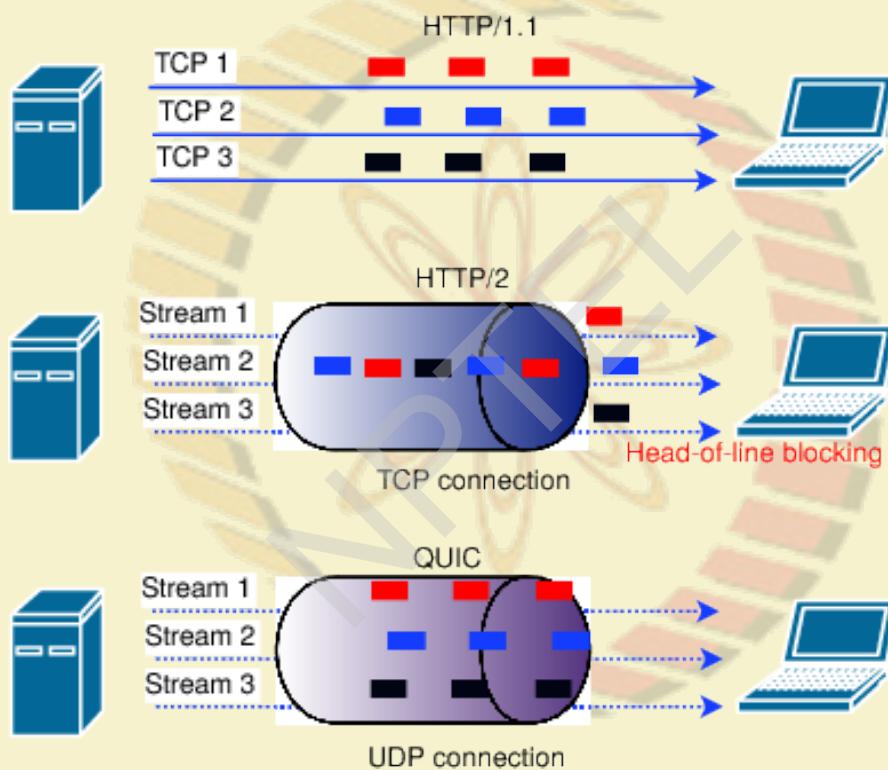


QUIC: 0-RTT Connection



Source: ACM
SIGCOMM, 2017

QUIC: Multi Stream HoL Blocking Free Protocol





thank you!



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES

Introduction to Socket Programming

Sandip Chakraborty

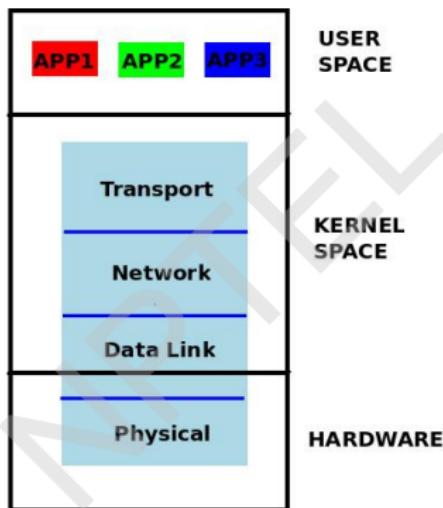
Department of Computer Science and Engineering,

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

August 8, 2018

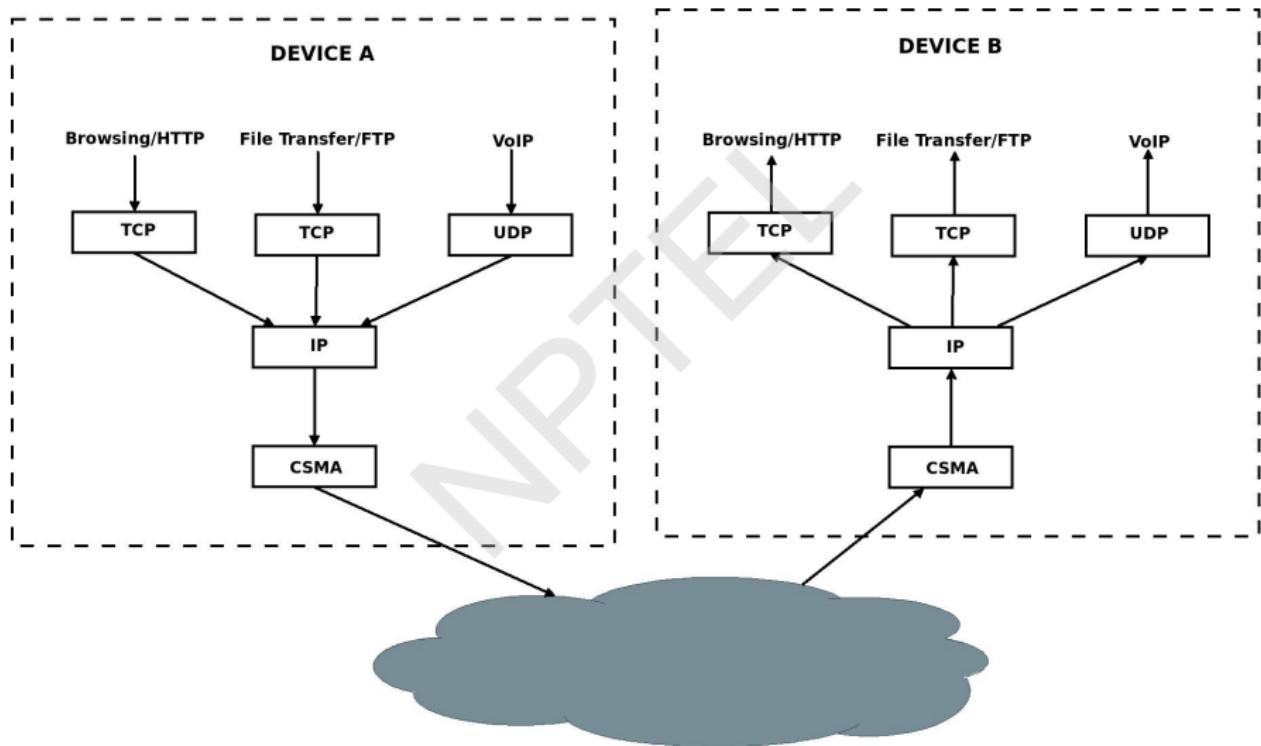


Connecting Network with Operating System

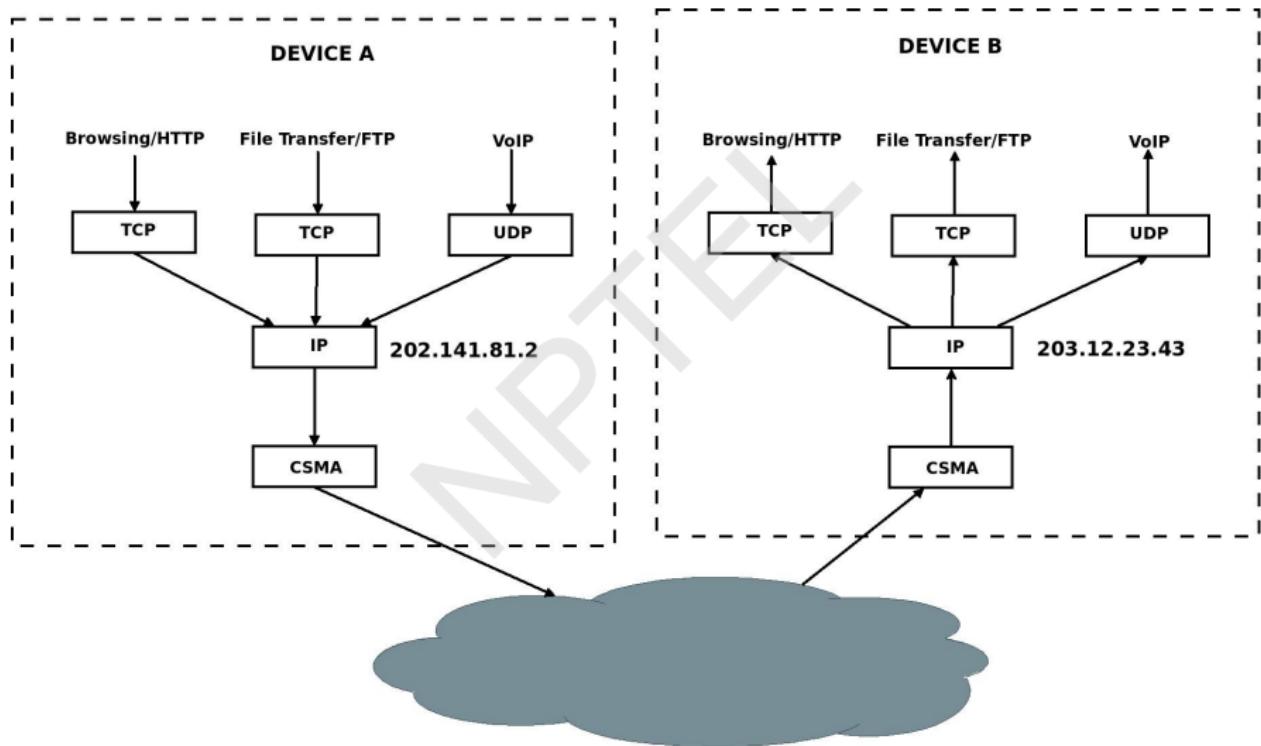


Check the `net` module (download Kernel source and check `/usr/src/linux/net`)!

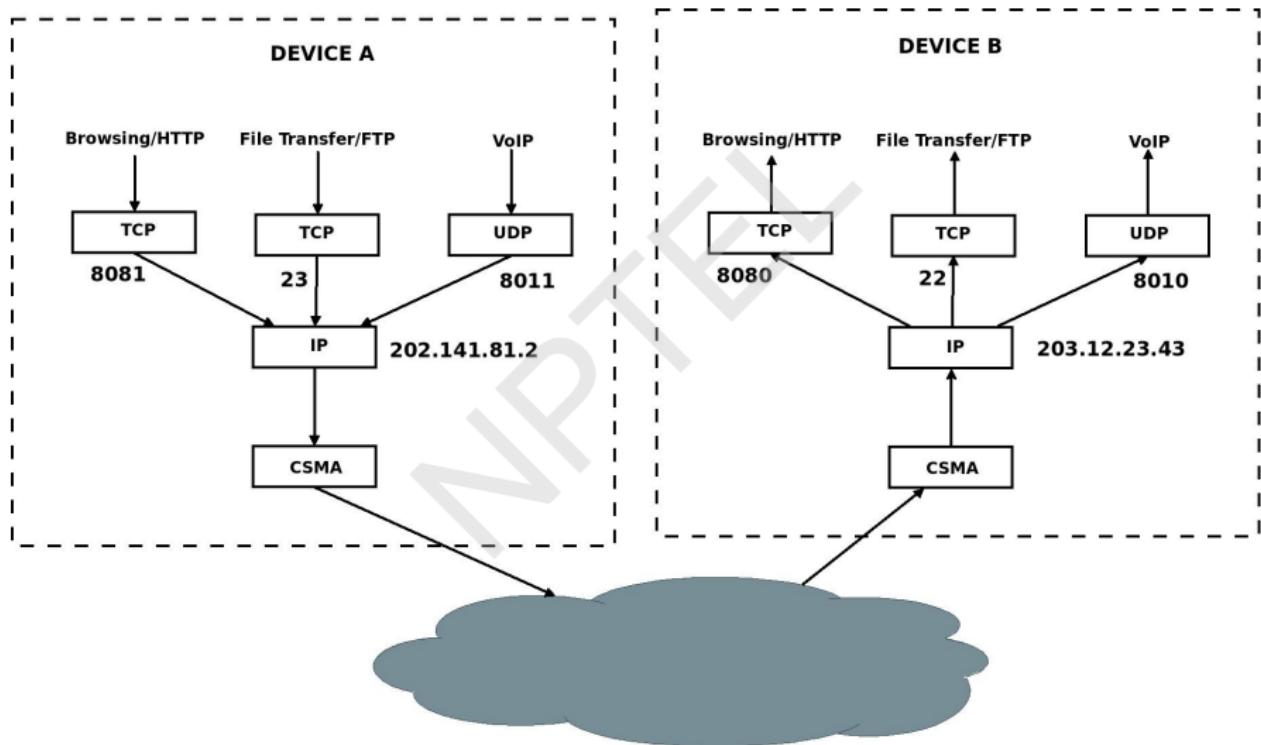
Application Multiplexing in TCP/IP



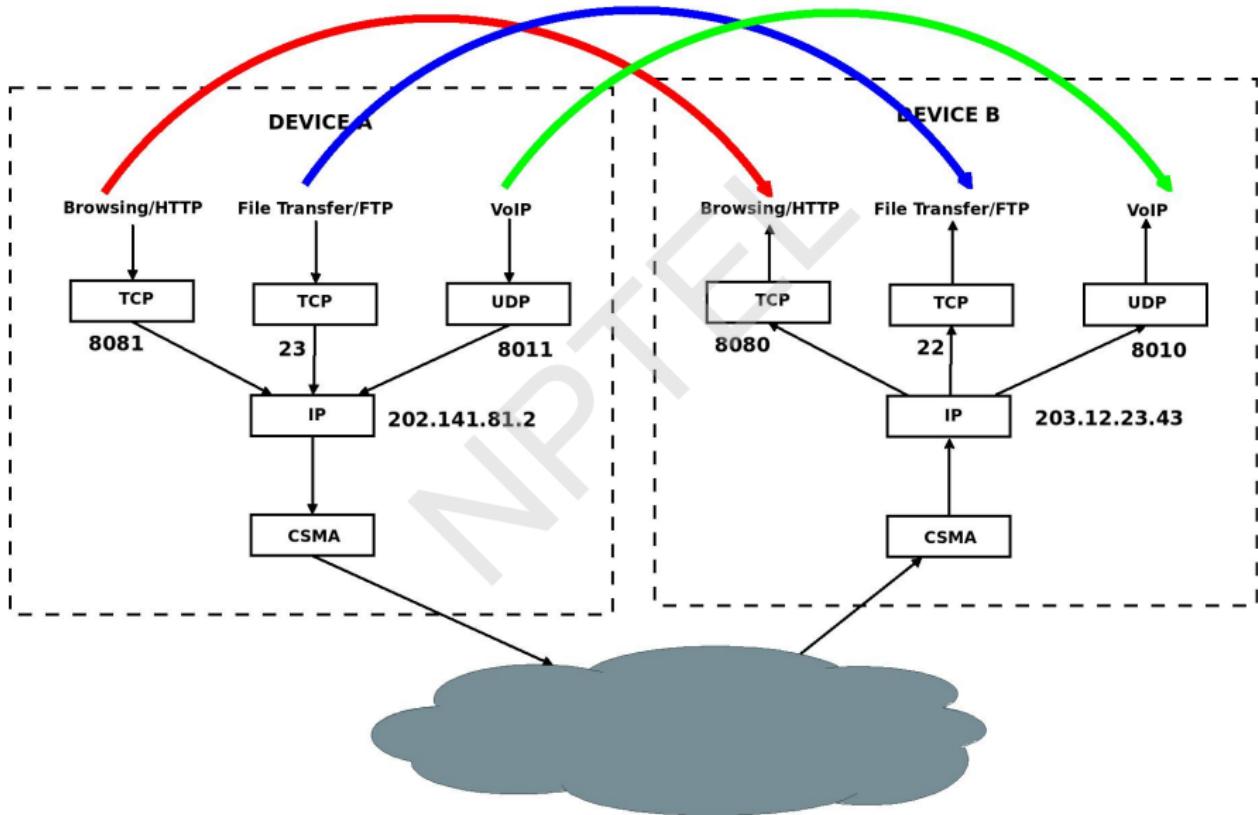
Application Multiplexing in TCP/IP



Application Multiplexing in TCP/IP

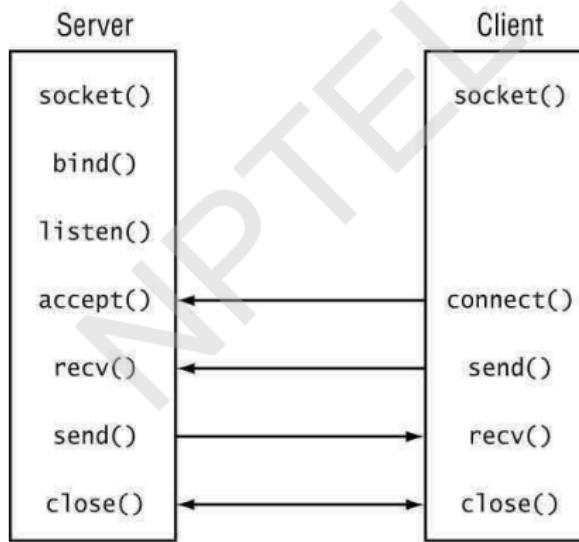


What are Sockets?



Socket Programming Framework/API

A set of **system calls** to get the service from TCP/IP protocol stack (net module in the OS kernel).



Socket Types

- The Internet is a trade-off between performance and reliability - **Can you say why?**
- Some application requires fine grained performance (example - multimedia applications), while others require reliability (example - file transfer)
- Transport layer supports two services - Reliable (TCP), and Unreliable (UDP)
- Two types of sockets:
 - ① **Stream Socket (SOCK_STREAM)**: Reliable, connection oriented (TCP based)
 - ② **Datagram Socket (SOCK_DGRAM)**: Unreliable, connection less (UDP based)

Socket API

- `int s = socket(domain, type, protocol);` - Create a socket
 - domain: Communication domain, typically used AF_INET (IPv4 Protocol)
 - type: Type of the socket - SOCK_STREAM or SOCK_DGRAM
 - protocol: Specifies protocols - usually set to 0 – **Explore!**
- `int status = bind(sockid, &addrport, size);` - Reserves a port for the socket.
 - sockid: Socket identifier
 - addrport: struct sockaddr_in - the (IP) address and port of the machine (address usually set to INADDR_ANY chooses a local address)
 - size: Size of the sockaddr structure

struct sockaddr_in

- `sin_family` : Address family, `AF_INET` for IPv4 Protocol
- `sin_addr.s_addr`: Source address, `INADDR_ANY` to choose the local address
- `sin_port`: The port number
- We need to use `htons()` function to convert the port number from **host byte order** to **network byte order**.

```
struct sockaddr_in serveraddr;
int port = 3028;
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = INADDR_ANY;
serveraddr.sin_port = htons(port);
```

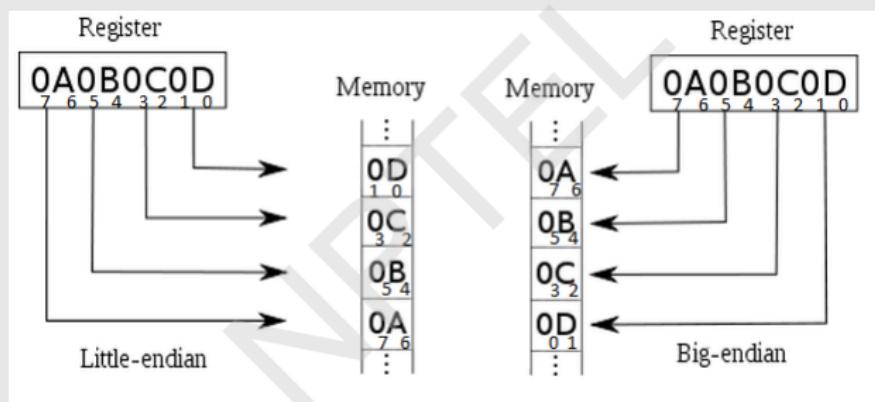
Host Byte Order to Network Byte Order - Why?

- Little Endian and Big Endian System

NPTEL

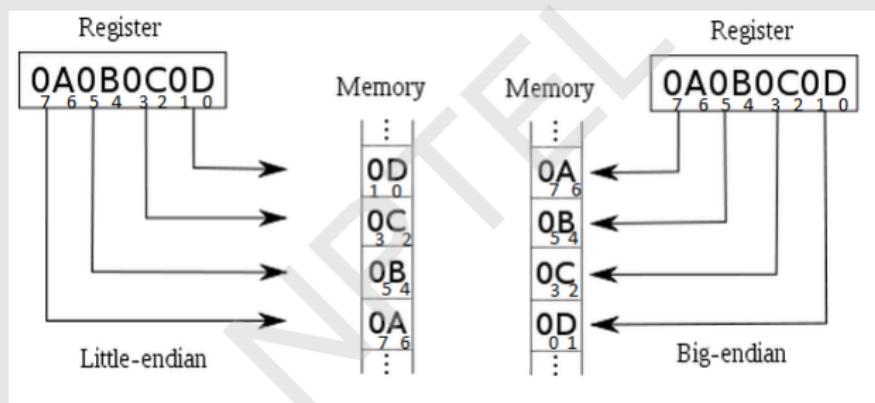
Host Byte Order to Network Byte Order - Why?

- Little Endian and Big Endian System



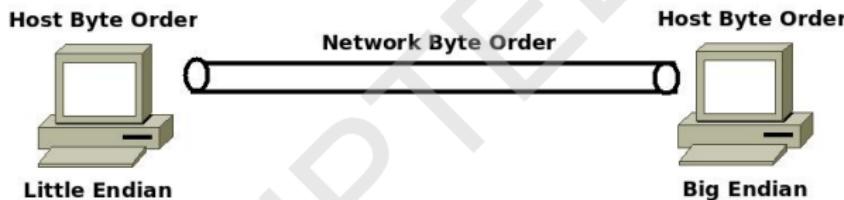
Host Byte Order to Network Byte Order - Why?

- Little Endian and Big Endian System



- Assume a communication from a Little Endian to a Big Endian System or vice-versa!

Host Byte Order to Network Byte Order - Why?



Listen and Accept a Socket Connection

```
struct sockaddr_in cli_addr;  
listen(sockfd,5);  
clilen = sizeof(cli_addr);  
newsockfd = accept(sockfd,(struct sockaddr *) &cli_addr,  
&clilen);
```

- **Active Open and Passive Open**

- The server needs to announce its address, remains in the open state and waits for any incoming connections - Passive Open
- The client only opens a connection when there is a need for data transfer - Active Open
- Connection is initiated by the client

Data Transfer through Sockets

① For SOCK_STREAM:

- `read(newsockfd,buffer,255);`
- `write(newsockfd, "I got your message",18);`

② For SOCK_DGRAM:

- `recvfrom(sock,buf,1024,0,(struct sockaddr *)&from,&fromlen);`
- `sendto(sock, "Got your message",17,0,(struct sockaddr *)&from,fromlen);`

Putting it All Together

Check the details and sample codes at

http://www.linuxhowtos.org/C_C++/socket.htm.

Socket Programming Tutorials

- Beej's Guide to Network Programming -
<http://beej.us/guide/bgnet/>
- <http://cs.baylor.edu/~donahoo/practical/CSockets/textcode.html>
- <http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html>

Thank You

Socket Programming - Concurrent Servers

Sandip Chakraborty

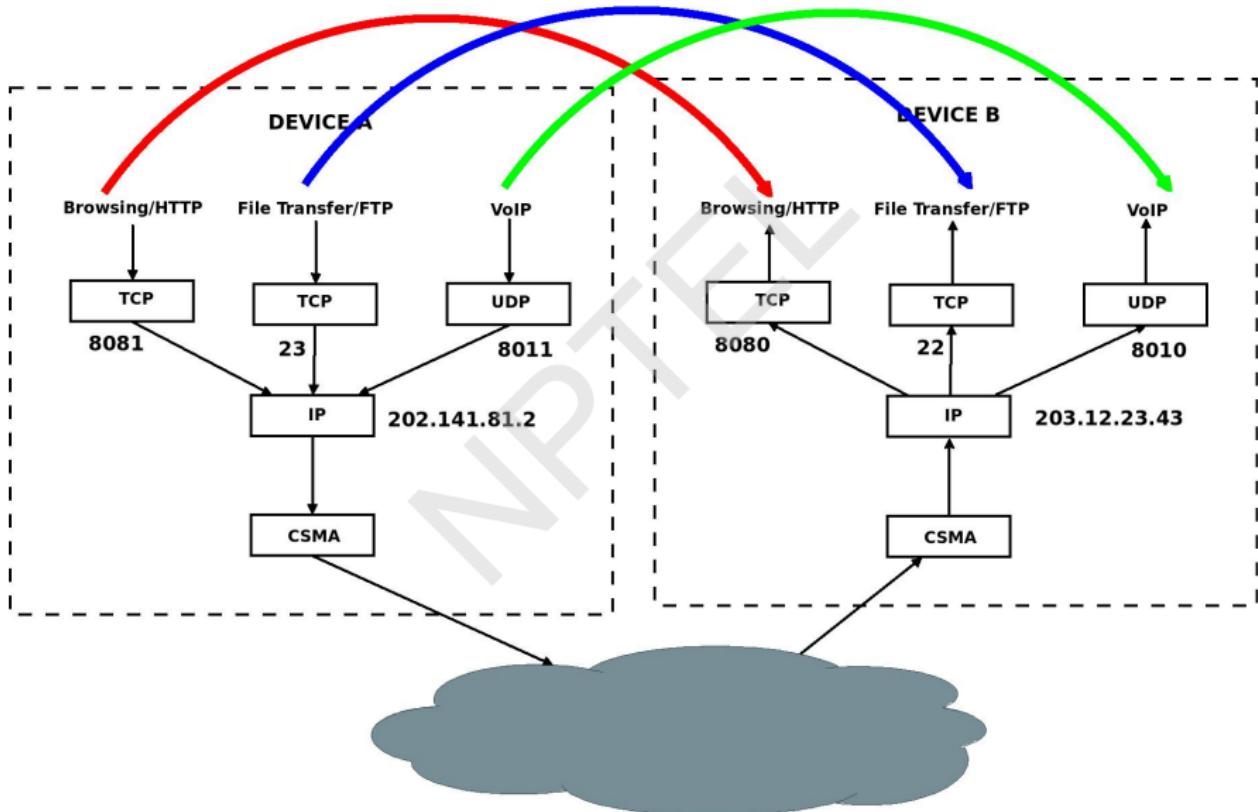
Department of Computer Science and Engineering,

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

August 8, 2018

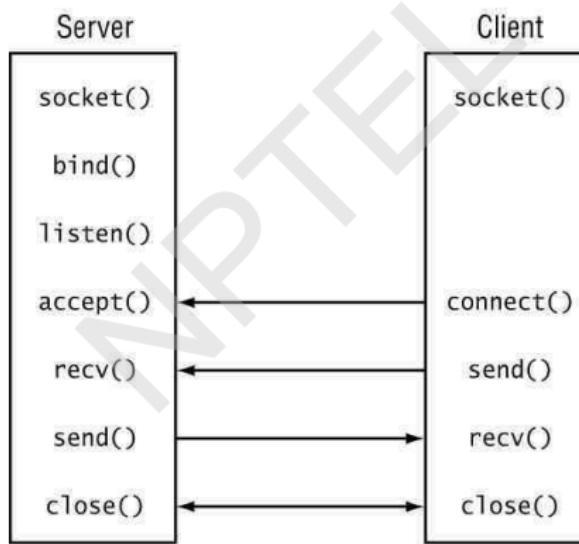


What are Sockets?

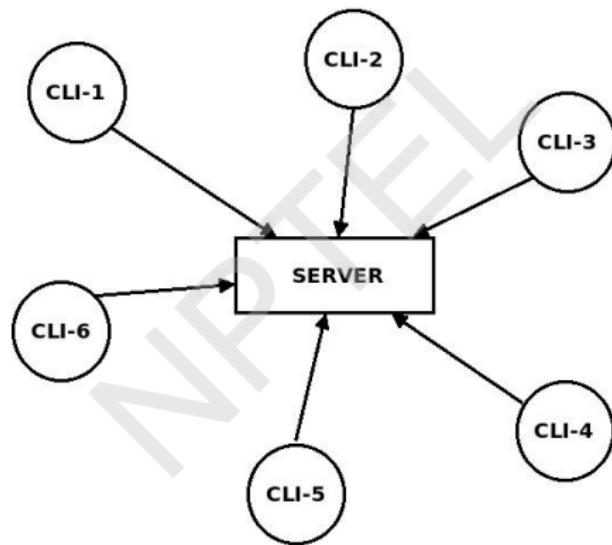


Socket Programming Framework/API

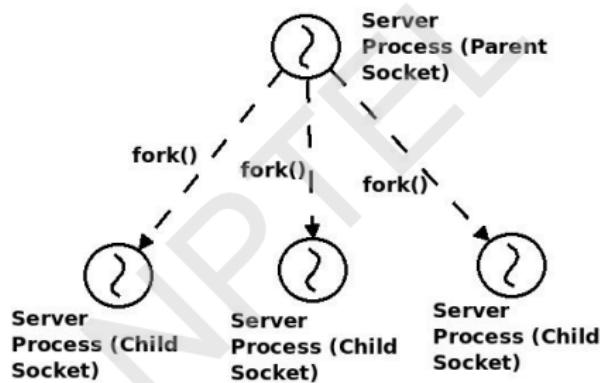
A set of **system calls** to get the service from TCP/IP protocol stack (net module in the OS kernel).



Concurrent Servers



Extending the Server Socket for Multiple Connections



Iterative Server

```
/*
 * listen: make this socket ready to accept connection requests
 */
if (listen(parentfd, 5) < 0) /* allow 5 requests to queue up */
    error("ERROR on listen");

/*
 * main loop: wait for a connection request, echo input line,
 * then close connection.
 */
clientlen = sizeof(clientaddr);
while (1) {

    /*
     * accept: wait for a connection request
     */
    childfd = accept(parentfd, (struct sockaddr *) &clientaddr, &clientlen);
    if (childfd < 0)
        error("ERROR on accept");
```

How Iterative Server Works

- The `listen()` call sets a flag that the socket is in listening state and set the maximum number of backlog connections.
- The `accept()` call blocks a listening socket until a new connection comes in the connection queue and it is accepted.
- Once the new connection is accepted, a new socket file descriptor (say `connfd`) is returned, which is used to read and write data to the connected socket.
- All other connections, which come in this duration, are backlogged in the connection queue.
- Once the handling of the current connected socket is done, the next `accept()` call accepts the next incoming connection from the connection queue (if any), or blocks the listening socket until the next connection comes.

Extending Iterative Server to Concurrent Server

- Parallel processing of each incoming sockets, so that the accept() call is executed more frequently.

```
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                   &clilen) ;

if (newsockfd < 0) {
    printf("Accept error\n");
    exit(0);
}

/* Having successfully accepted a client connection, the
server now forks. The parent closes the new socket
descriptor and loops back to accept the next connection.
*/
if (fork() == 0) {

    /* This child process will now communicate with the
       client through the send() and recv() system calls.
    */
    close(sockfd); /* Close the old socket since all
                     communications will be through
                     the new socket.
    */

    /* We initialize the buffer, copy the message to it,
       and send the message to the client.
    */

    strcpy(buf, "Message from server");
    send(newsockfd, buf, strlen(buf) + 1, 0);

    /* We again initialize the buffer, and receive a
       message from the client.
    */
}
```

Thank You

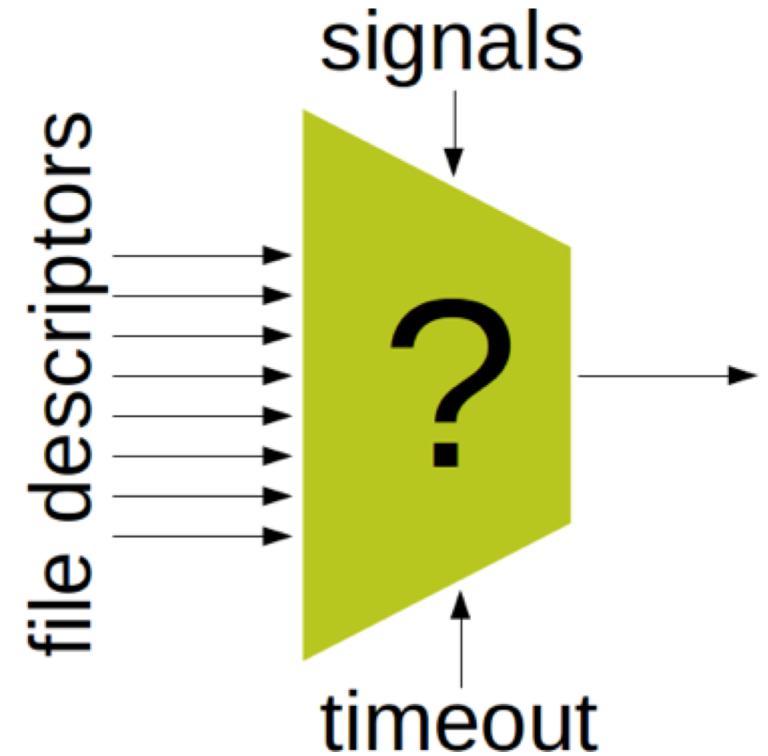
Iterative Server Implementation - Select() System Call

Department of Computer Science
and Engineering

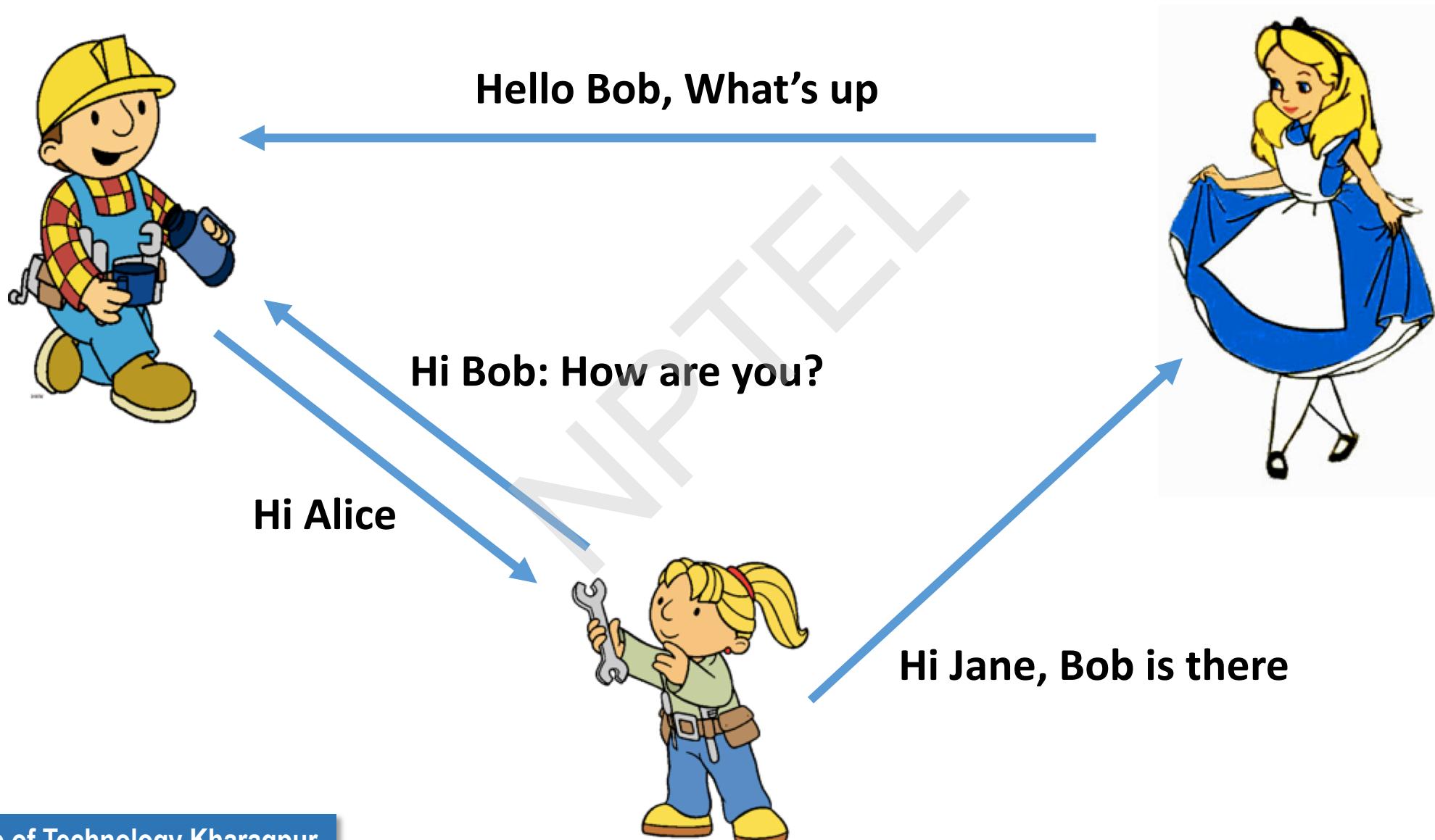


INDIAN INSTITUTE OF TECHNOLOGY
KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ernet.in



An Example – Peer to Peer Chat Application



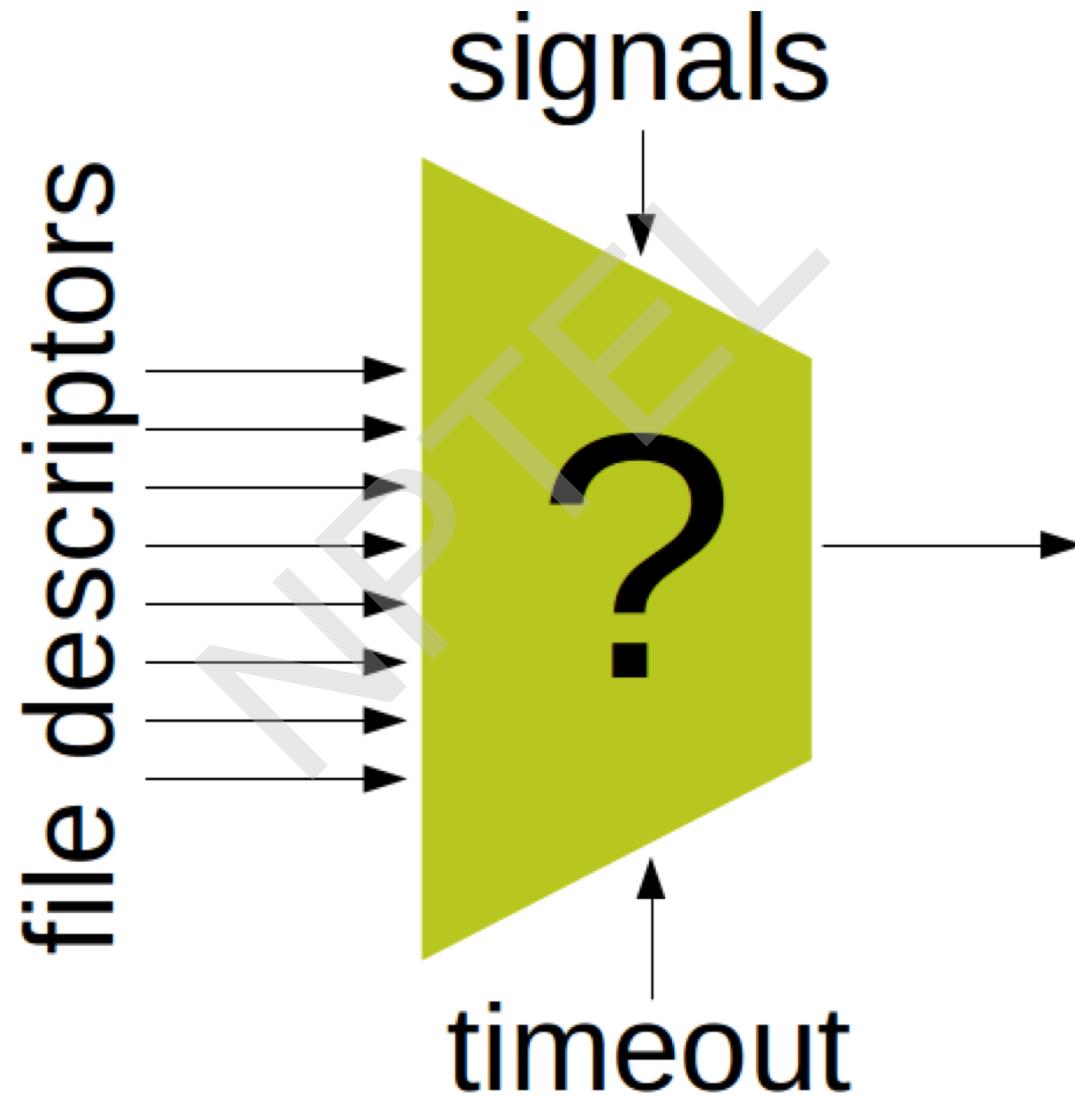
Peer to Peer Chat Application

- No central server to control chat message delivery
- Every user runs its own chat server (a TCP server for incoming connections and messages)
- **Note, UNIX maintains every connection (socket) as a file descriptor**
- You are also reading data from the STDIN (the standard input file descriptor)
 - here the keyboard
- Message communication is **asynchronous** – you can receive a message while typing

The select() system call

- Selects from multiple file descriptors – a concurrent way to handle multiple file descriptors simultaneously, even from a single process or thread
- What happens in an iterative server implementation that you have done earlier?
 - accept() call is blocked until you have completed the read() and write() calls
 - What if you do multiples read() and write() activities after accepting an incoming connection? – **all other connections are blocked and waiting in the connection queue (may get starved !!!)**
 - select() is the way to break this blocking
- **Advantage:** You do not need to create multiple child processes now. No need to worry about zombies !!!

Select is a Multiplexer ...



How the select() call works

highest-numbered file descriptor in any of the three sets, plus 1

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

Read file descriptors

exception file descriptors

Write file descriptors

A timeout value

Synchronous I/O multiplexing over asynchronous inputs

The Three File Descriptors

- *readfds* will be watched to see if characters become available for reading
- *writefd*s will be watched to see if space is available for write
- *exceptfds* will be watched for exceptional conditions

The Timeout in Select

Timeout: the interval that `select()` should block waiting for a file descriptor to become ready. The `select()` call will block until either

- A file descriptor becomes ready
- The call is interrupted by a signal handler; or
- The timeout expires.

```
struct timeval
{
    long tv_sec;      /* seconds */
    long tv_usec;     /* microseconds */
};
```

How Do You Pass the File Descriptors to Select

```
int socket_fd, result;  
fd_set readset;  
...  
/* Socket has been created and connected to the other party */  
...  
  
/* Call select() */  
do {  
    FD_ZERO(&readset);  
    FD_SET(socket_fd, &readset);  
    result = select(socket_fd + 1, &readset, NULL, NULL, NULL);  
} while (result == -1 && errno == EINTR);
```

- **FD_ZERO**: Initializes the file descriptor set `fd_set` – a bitmap of fixed size
- **FD_SET**: Set a file descriptor as a part of a `fd_set`

How Select Works

- Loops over the file descriptors
- For every file descriptor (FD), it calls that FD's **poll()** method, which will add the caller to that FD's wait queue, and return which events (readable, writeable, exception) currently apply to that FD.
- If any file descriptor matches the condition that the user was looking for, **select()** will simply return immediately, after updating the appropriate **FD_SETs** that the user passed.
- If not, however, **select()** will go to sleep, for up to the maximum timeout the user specified.

How Select Works

- If, during that interval, an interesting event happens to any file descriptor that **select()** is waiting on, that FD will notify its wait queue.
 - This will cause the thread sleeping inside **select()** to wake up,
 - It will repeat the above loop and see which of the FD's are now ready to be returned to the user.
- Return values of **select()**
 - **-1:** Means an error was encountered, you should do something about it. I just print the error
 - **0:** Means the call timed out without any event ready for the sockets monitored
 - **>0:** Is the number of sockets that have events pending (read, write, exception)

After Select Returns

```
if (result > 0) {  
    if (FD_ISSET(socket_fd, &readset)) {  
        /* The socket_fd has data available to be read */  
        result = recv(socket_fd, some_buffer, some_length, 0);  
        if (result == 0) {  
            /* This means the other side closed the socket */  
            close(socket_fd);  
        }  
        else {  
            /* I leave this part to your own implementation */  
        }  
    }  
}
```

- **FD_ISSET:** tests to see if a file descriptor is part of the set