IIT KHARAGPUR

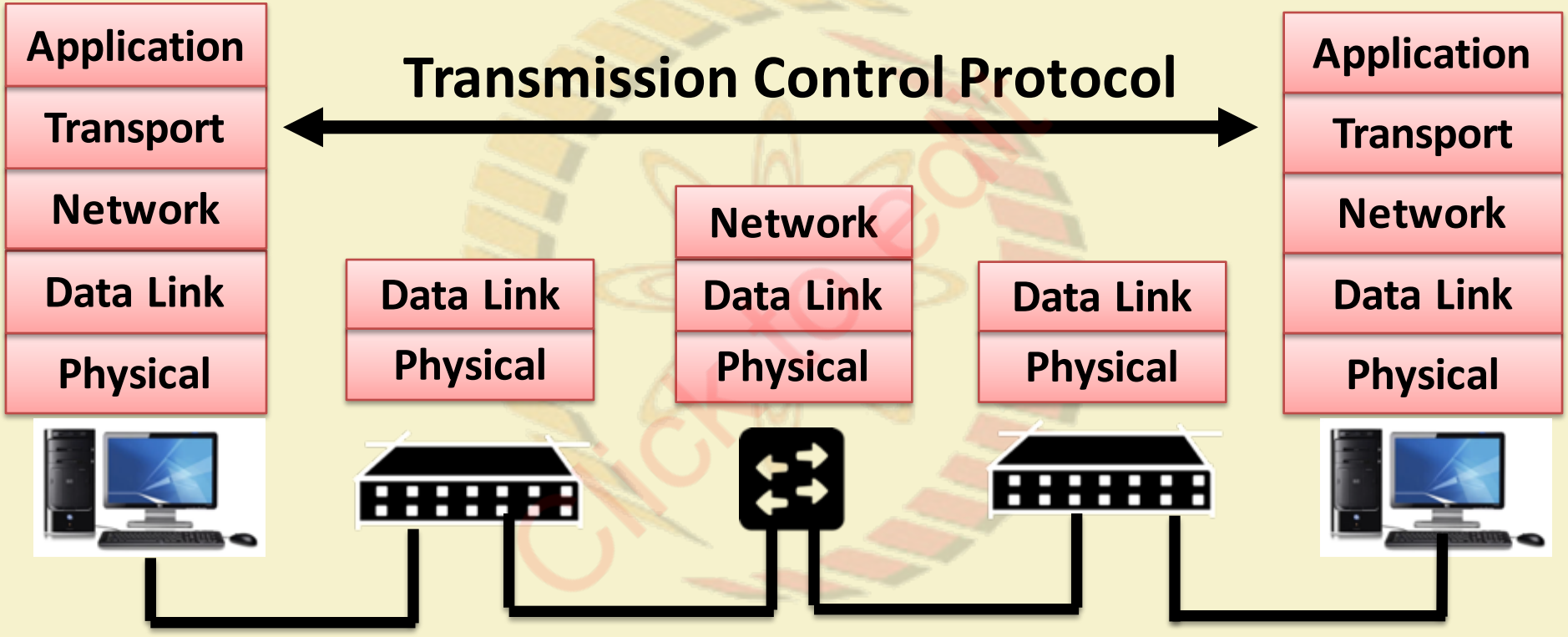
NPTEL ONLINE
CERTIFICATION COURSES

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)



TCP Congestion Control

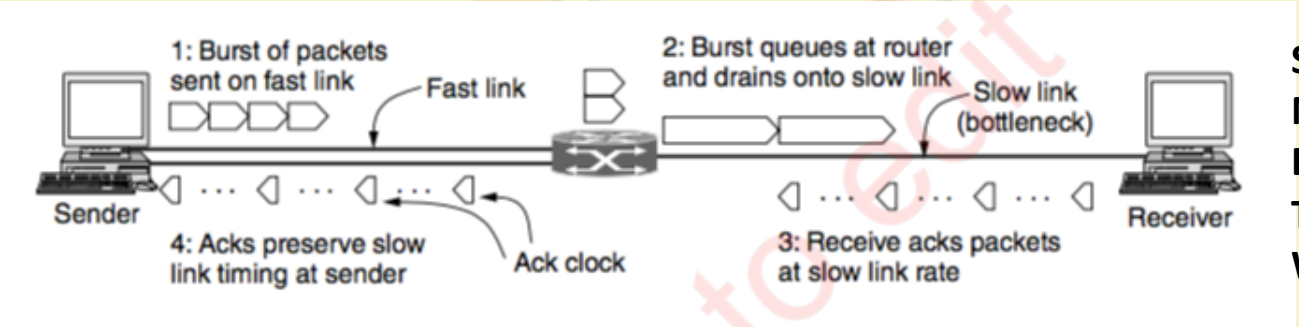
- Based on implementation of AIMD using a window and with packet loss as the binary signal
- TCP maintains a **Congestion Window (CWnd) –** number of bytes the sender may have in the network at any time
- **Sending Rate = Congestion Window / RTT**
- **Sender Window (SWnd) = Min (CWnd, RWnd)**
- RWnd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

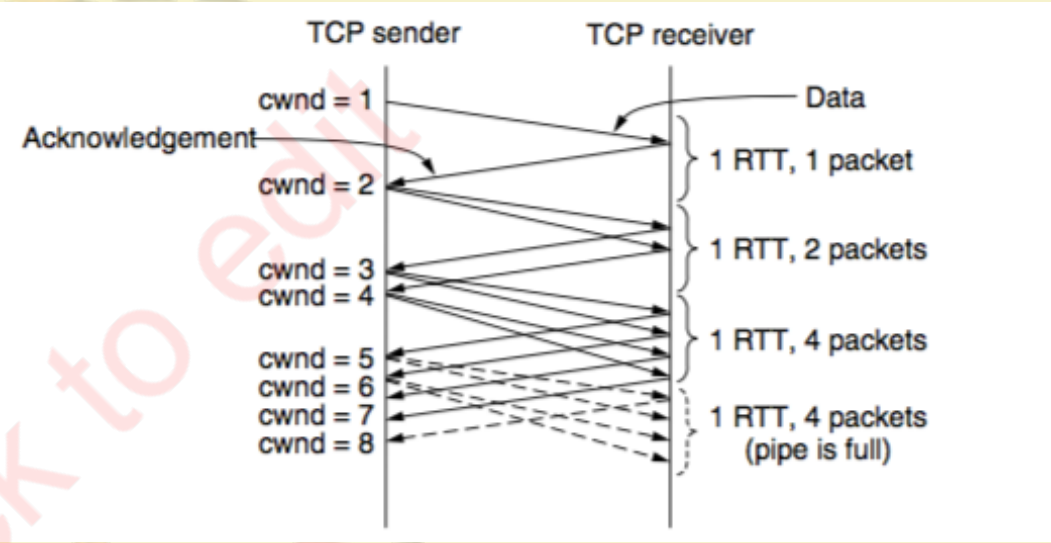
- A 10 Mbps link with 100 ms RTT
  - Appropriate CWnd = BDP = 1 Mbit
  - 1250 byte packets -> 100 packets to reach BDP
  - CWnd starts at 1 packet, and increased 1 packet at every RTT
  - 100 RTTs are required 10 sec before the connection reaches to a moderate rate

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**

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

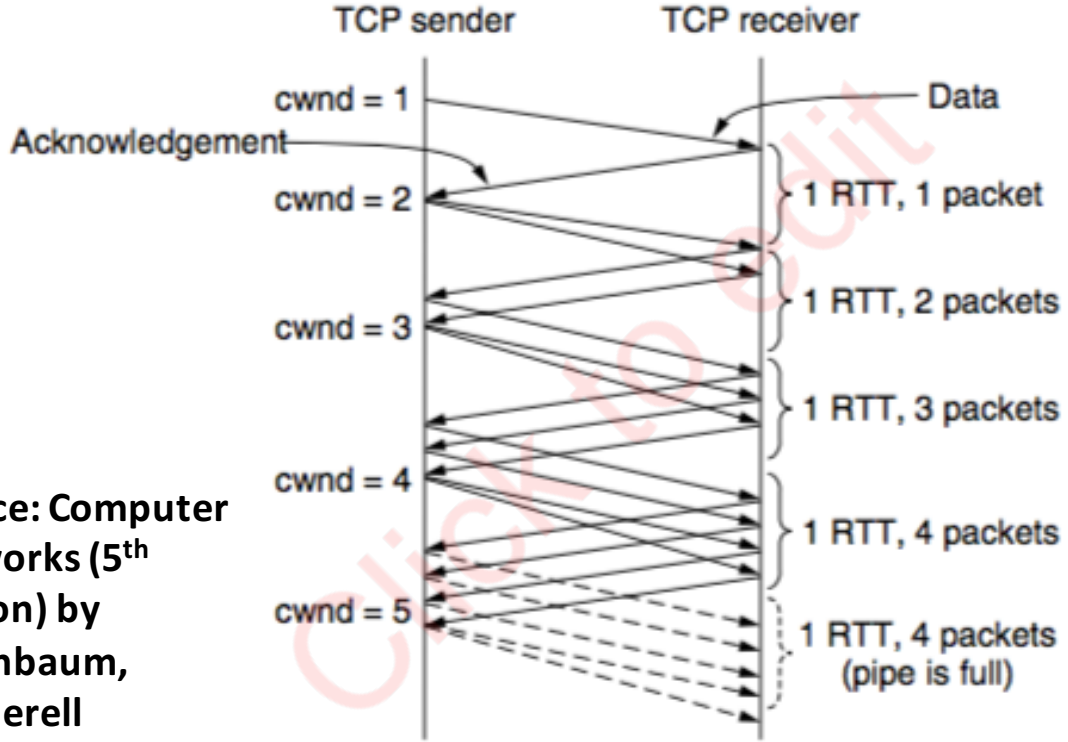
- Slow start causes exponential growth, eventually it will send too many packets into the network too quickly.
- To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold (ssthresh).**
- Initially ssthresh is set to BDP (or arbitrarily high), 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:

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

Additive Increase – Packet Wise Approximation



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

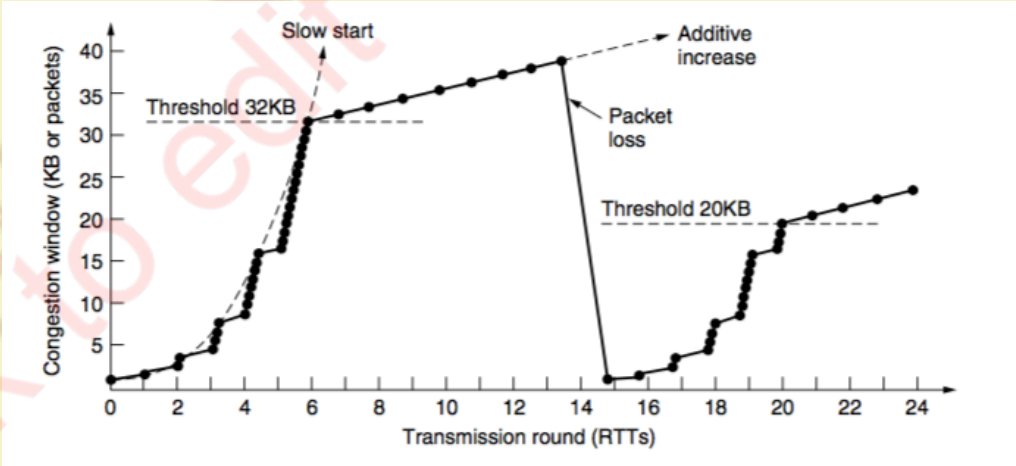
Triggering an Congestion

- 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 Tohoe

- 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

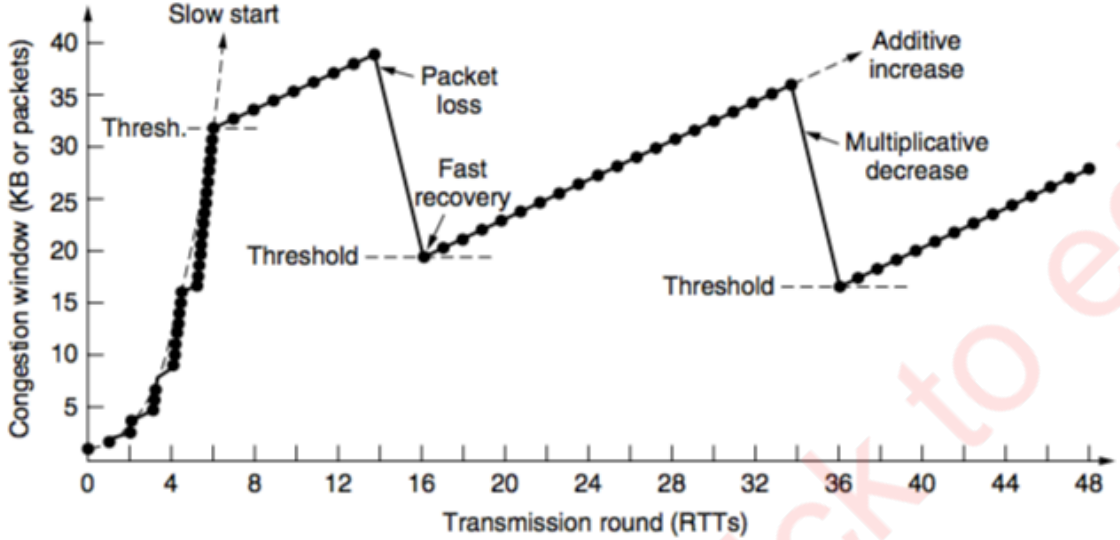
- Immediately transmit the lost segment (**fast retransmit)**, then transmit additional segments based on the DUPACKs received **(fast recovery)**

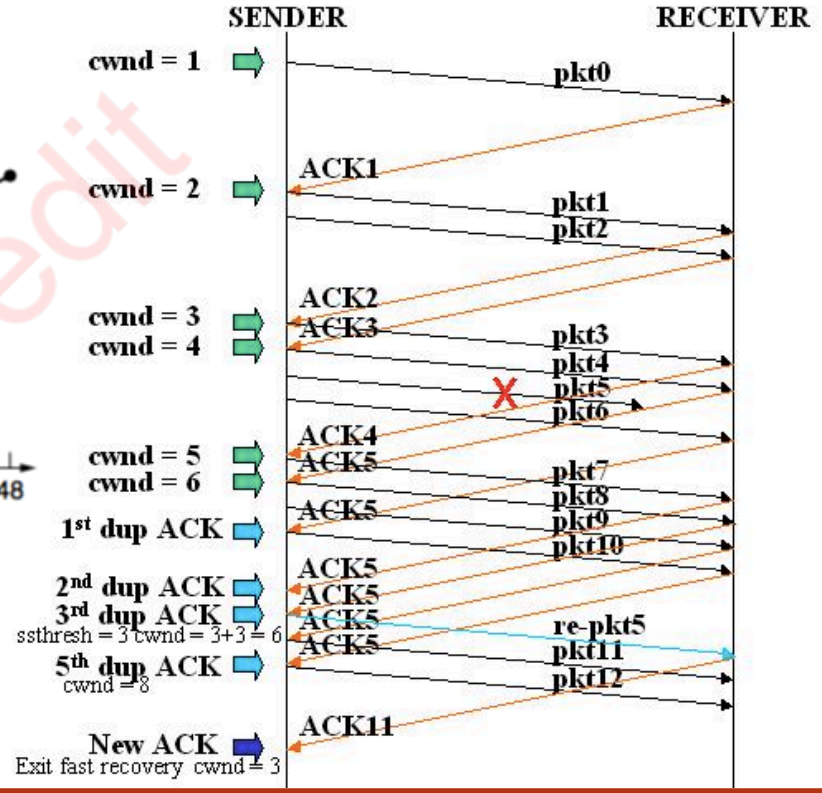
Fast Recovery – TCP Reno

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.

Fast Recovery – TCP Reno






thank you!

IIT KHARAGPUR

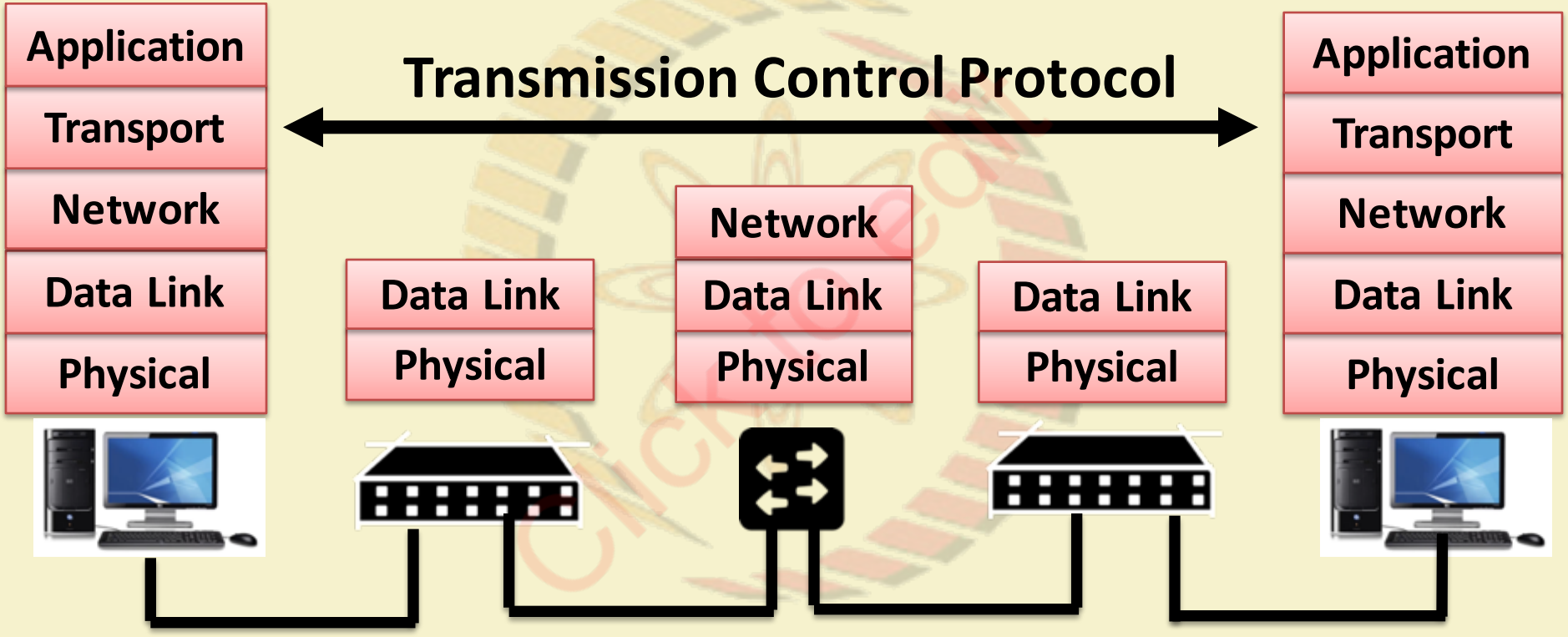
NPTEL ONLINE
CERTIFICATION COURSES

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)



TCP Congestion Control

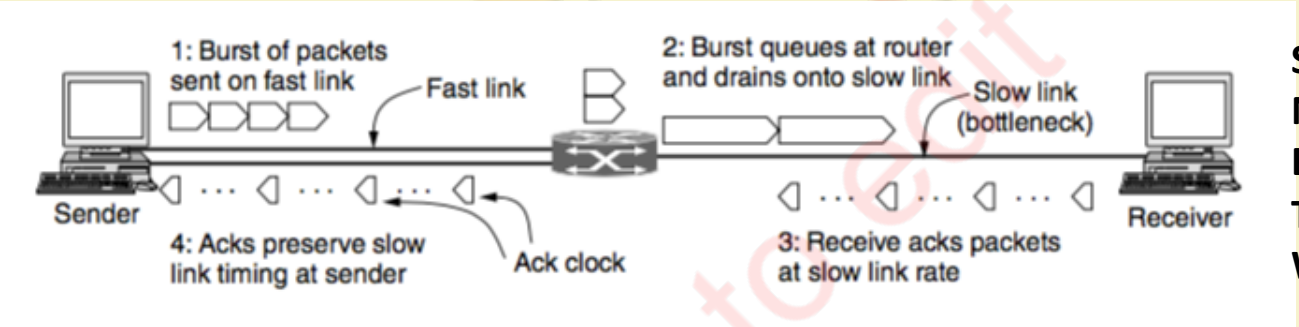
- Based on implementation of AIMD using a window and with packet loss as the binary signal
- TCP maintains a **Congestion Window (CWnd) –** number of bytes the sender may have in the network at any time
- **Sending Rate = Congestion Window / RTT**
- **Sender Window (SWnd) = Min (CWnd, RWnd)**
- RWnd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

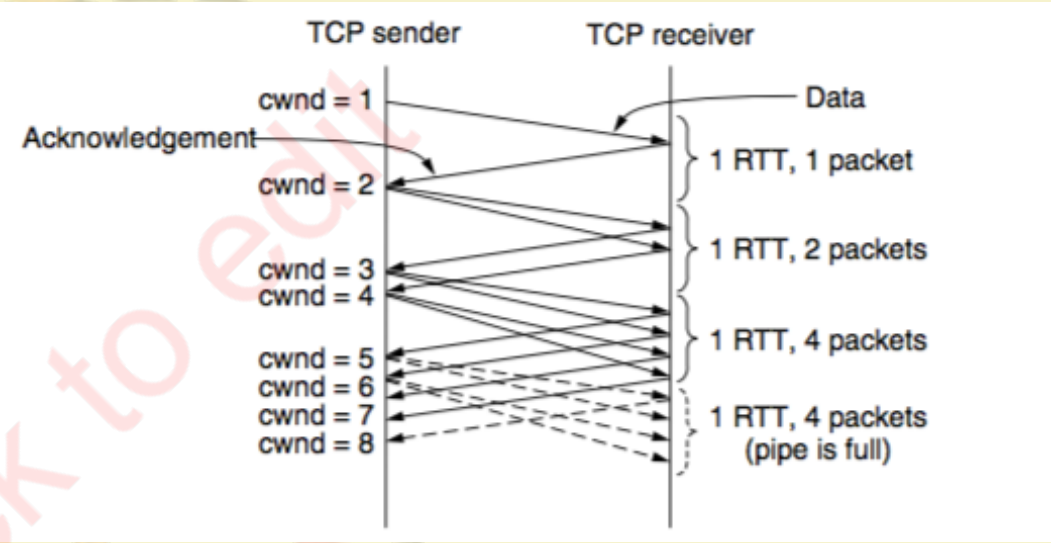
- A 10 Mbps link with 100 ms RTT
  - Appropriate CWnd = BDP = 1 Mbit
  - 1250 byte packets -> 100 packets to reach BDP
  - CWnd starts at 1 packet, and increased 1 packet at every RTT
  - 100 RTTs are required 10 sec before the connection reaches to a moderate rate

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**

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

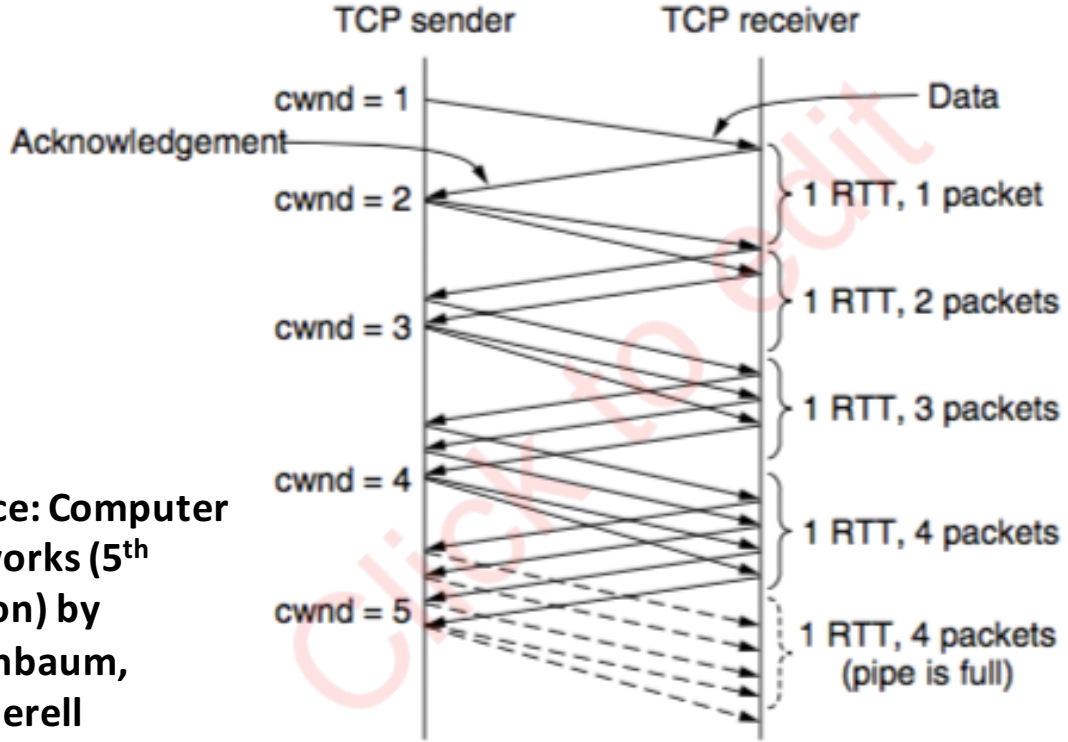
- Slow start causes exponential growth, eventually it will send too many packets into the network too quickly.
- To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold (ssthresh).**
- Initially ssthresh is set to BDP (or arbitrarily high), 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:

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

Additive Increase – Packet Wise Approximation



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

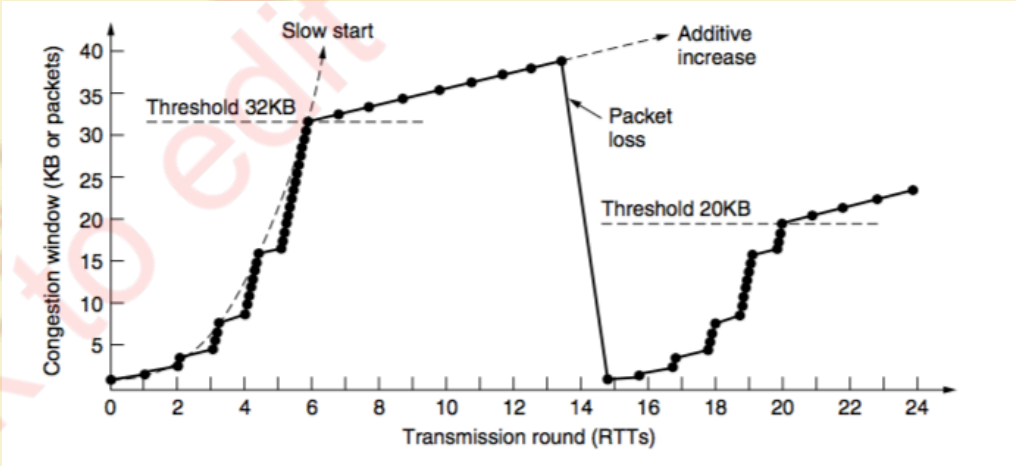
Triggering an Congestion

- 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 Tohoe

- 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

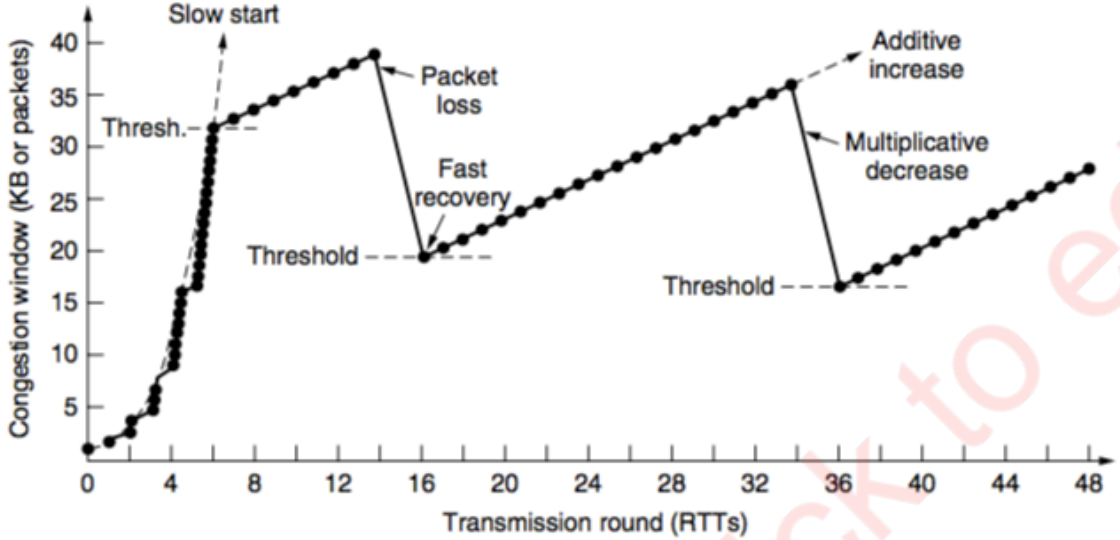
- Immediately transmit the lost segment (**fast retransmit)**, then transmit additional segments based on the DUPACKs received **(fast recovery)**

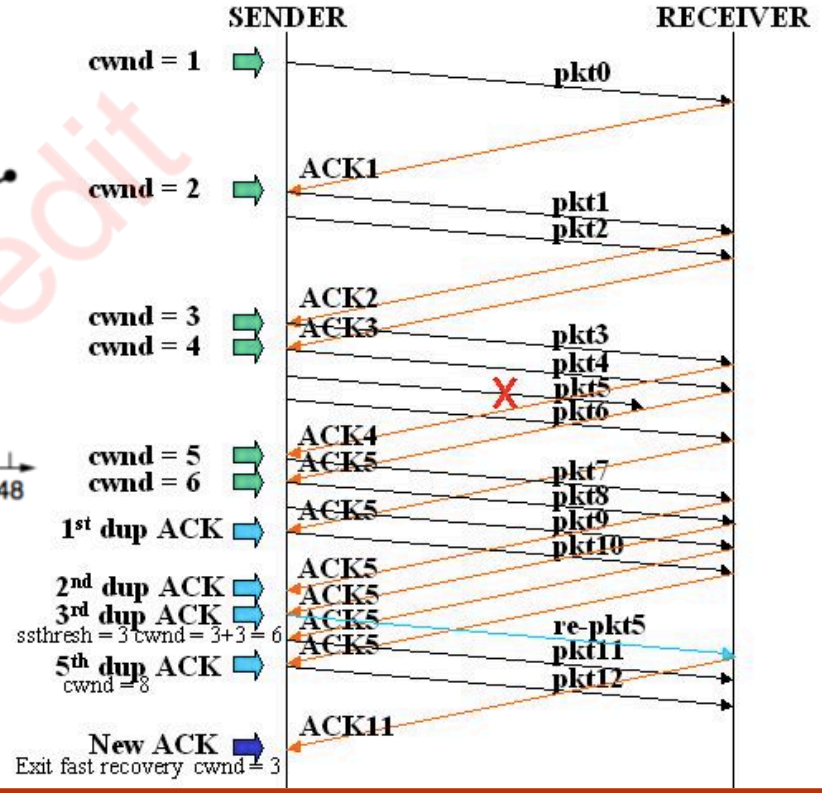
Fast Recovery – TCP Reno

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.

Fast Recovery – TCP Reno






thank you!

IIT KHARAGPUR

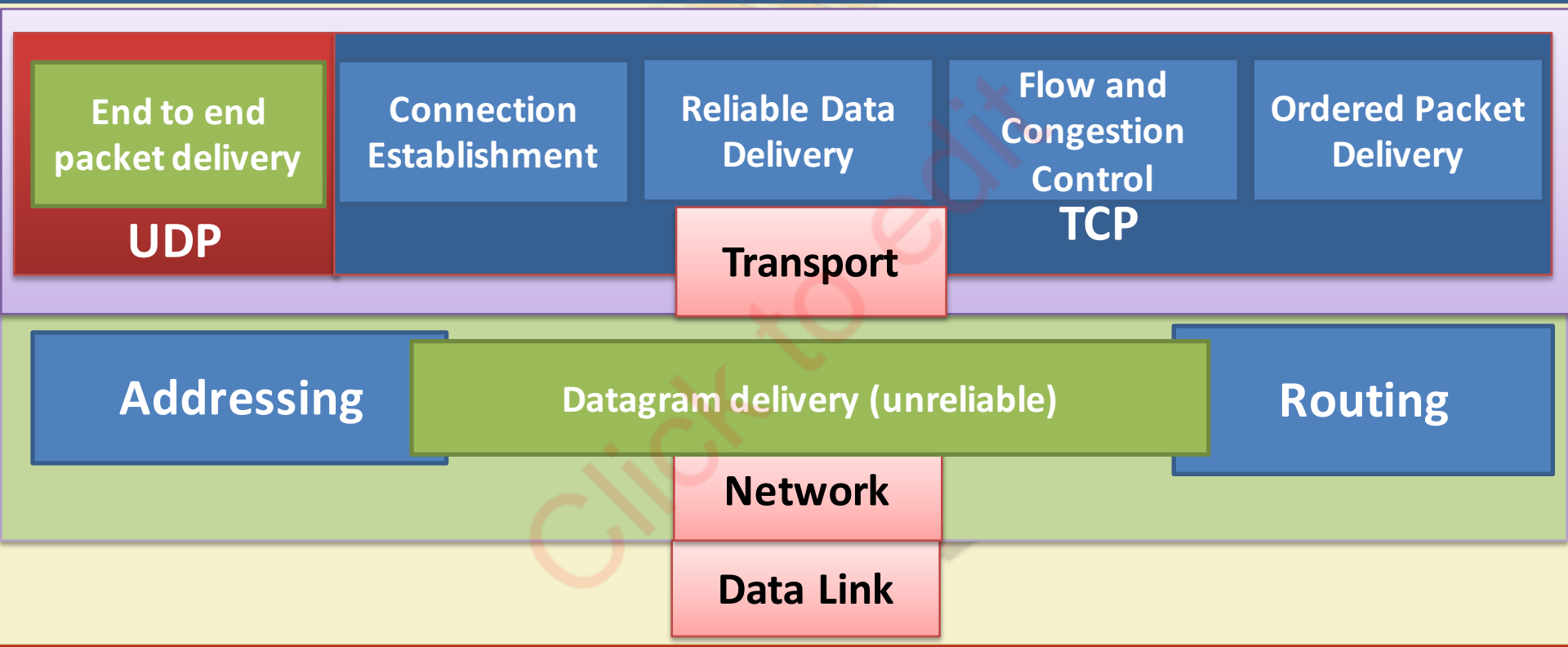
NPTEL ONLINE
CERTIFICATION COURSES

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)



UDP

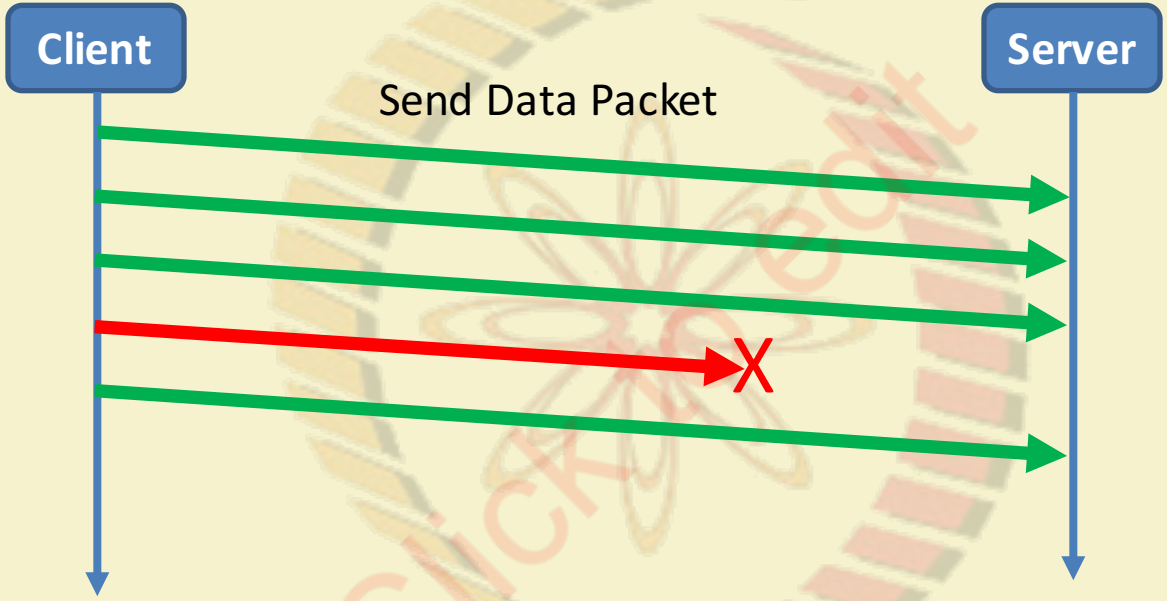
Features

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

Uses

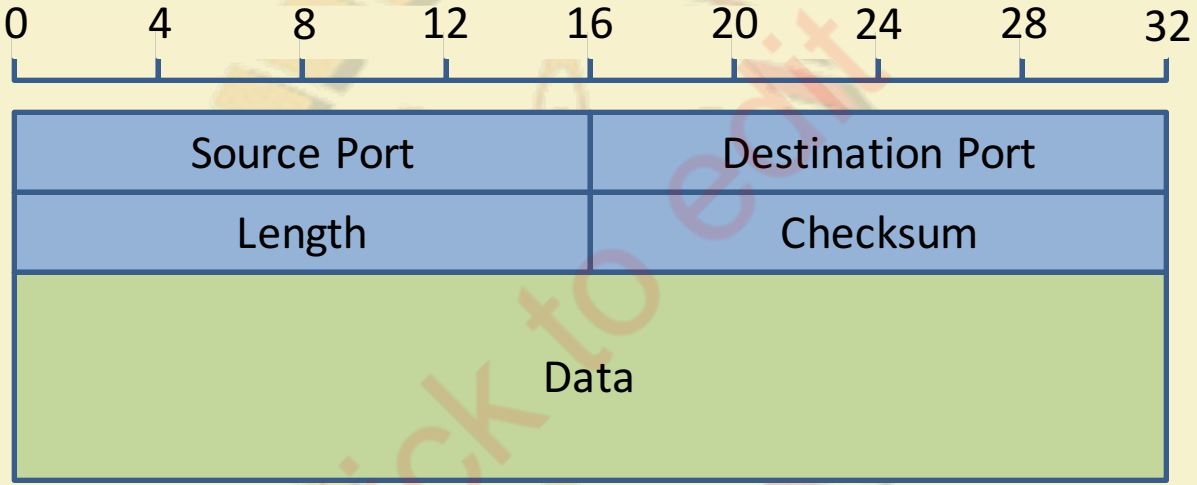
- **Provide performance**
  - No data holding in buffer like TCP
- **Short and sweet**
  - Have no overhead
  - Suitable for short messages

UDP is connection less and Unreliable



- No connection establishment, No reliability, No acknowledgement

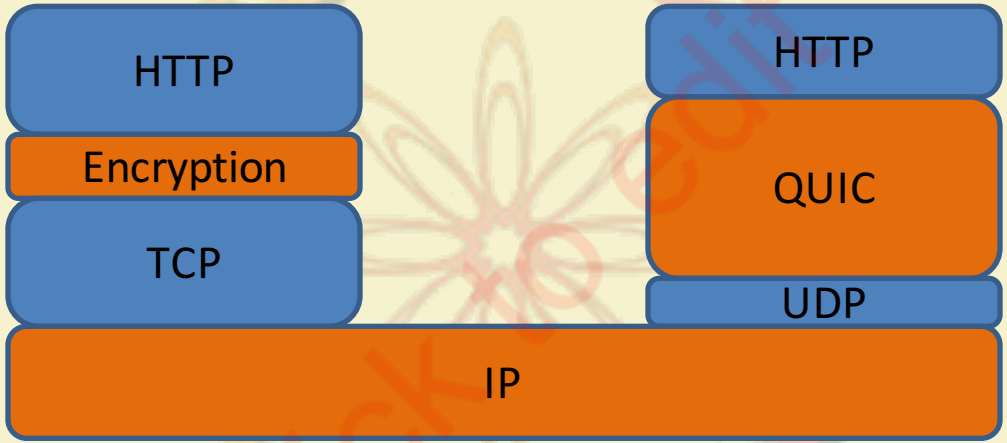
UDP Header



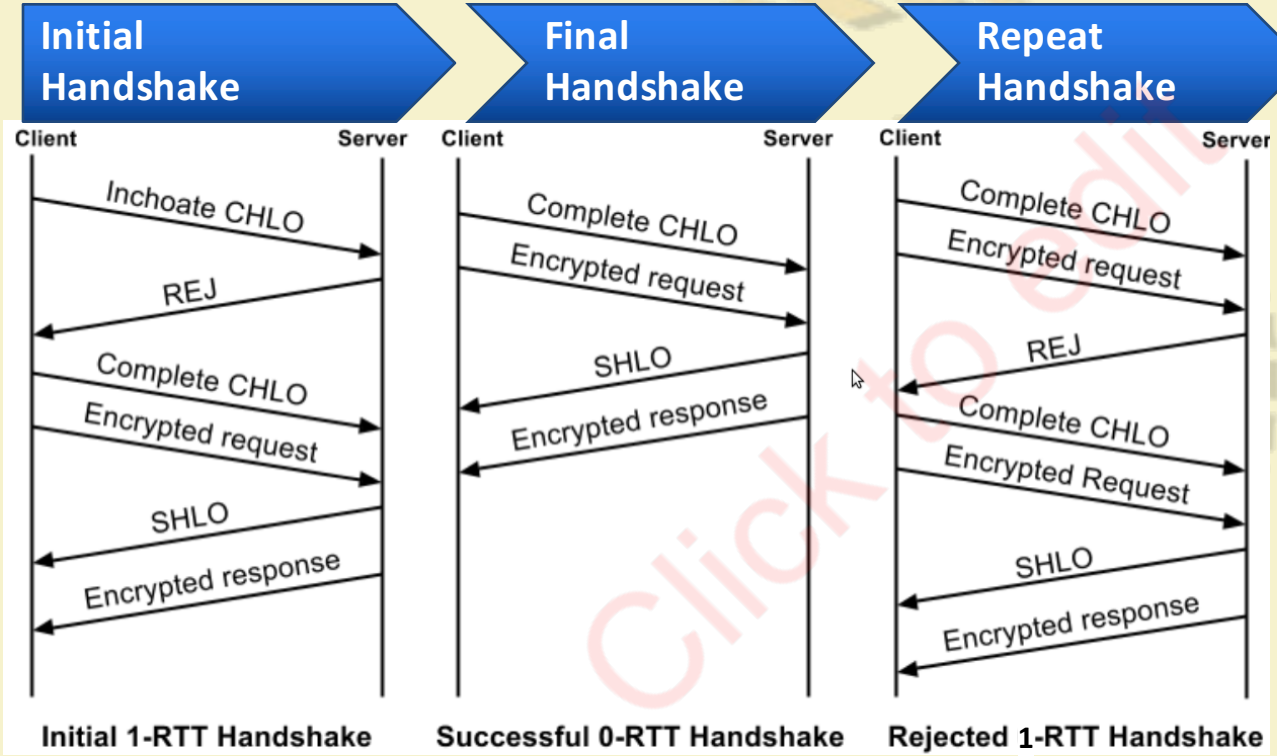
Applications

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

Quick UDP Internet Connections (QUIC)

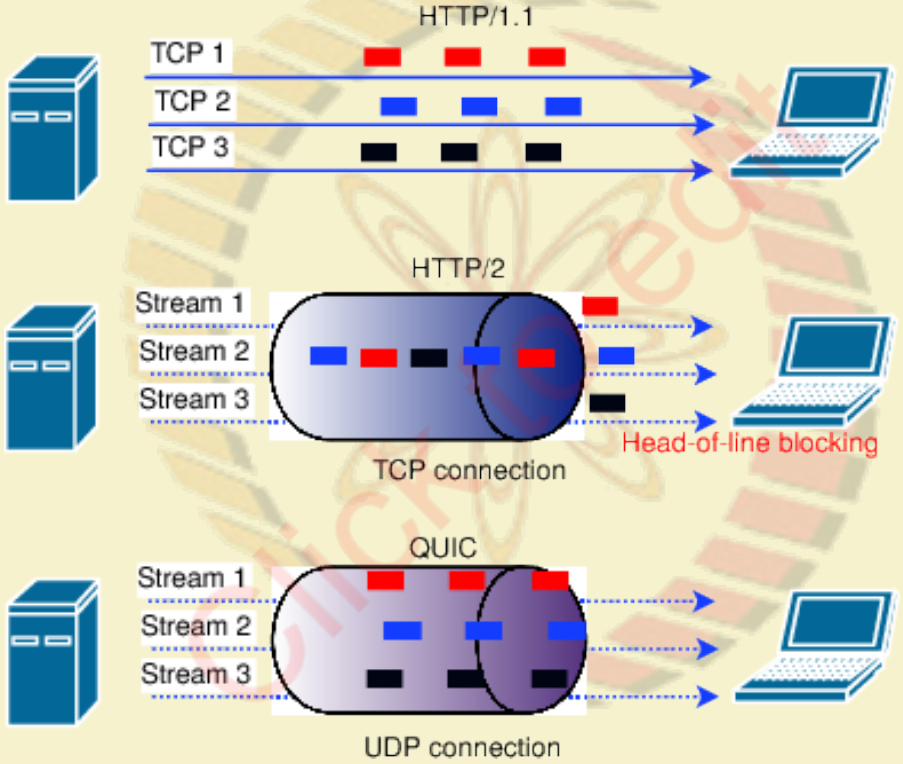


QUIC: 0-RTT Connection



Source: ACM SIGCOMM, 2017

QUIC: Multi Stream HoL Blocking Free Protocol




thank you!

Introduction to Socket Programming

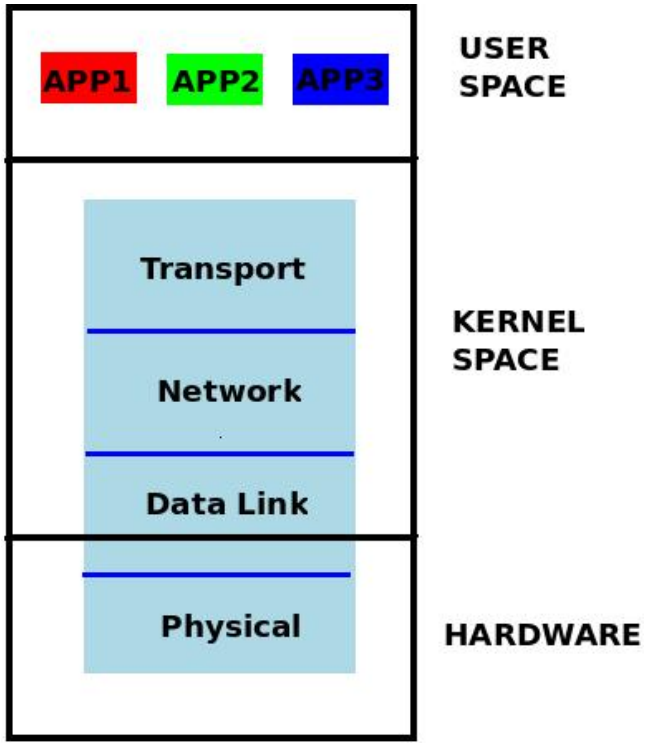
Sandip Chakraborty

Department of Computer Science and Engineering,

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

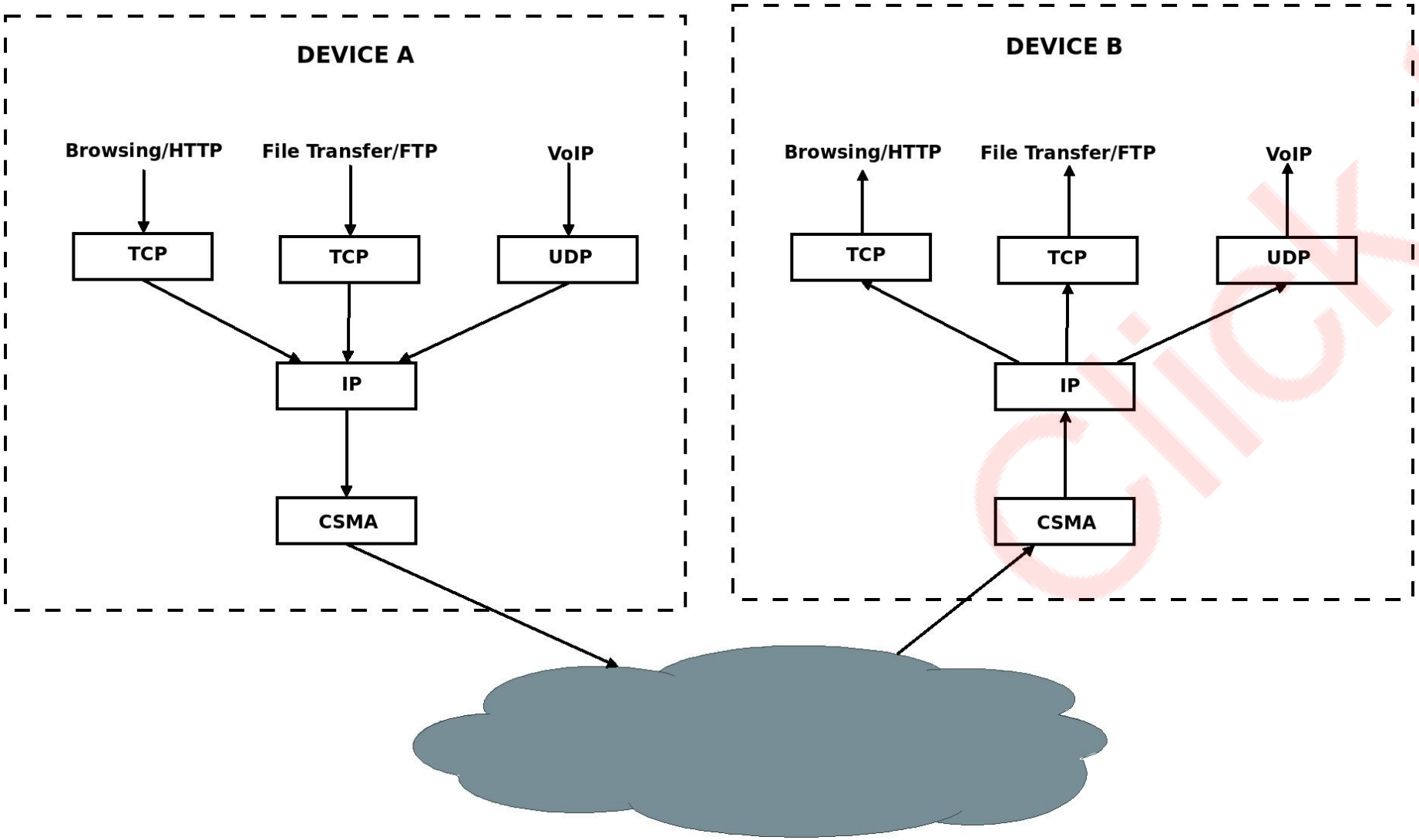
August 8, 2018

Connecting Network with Operating System

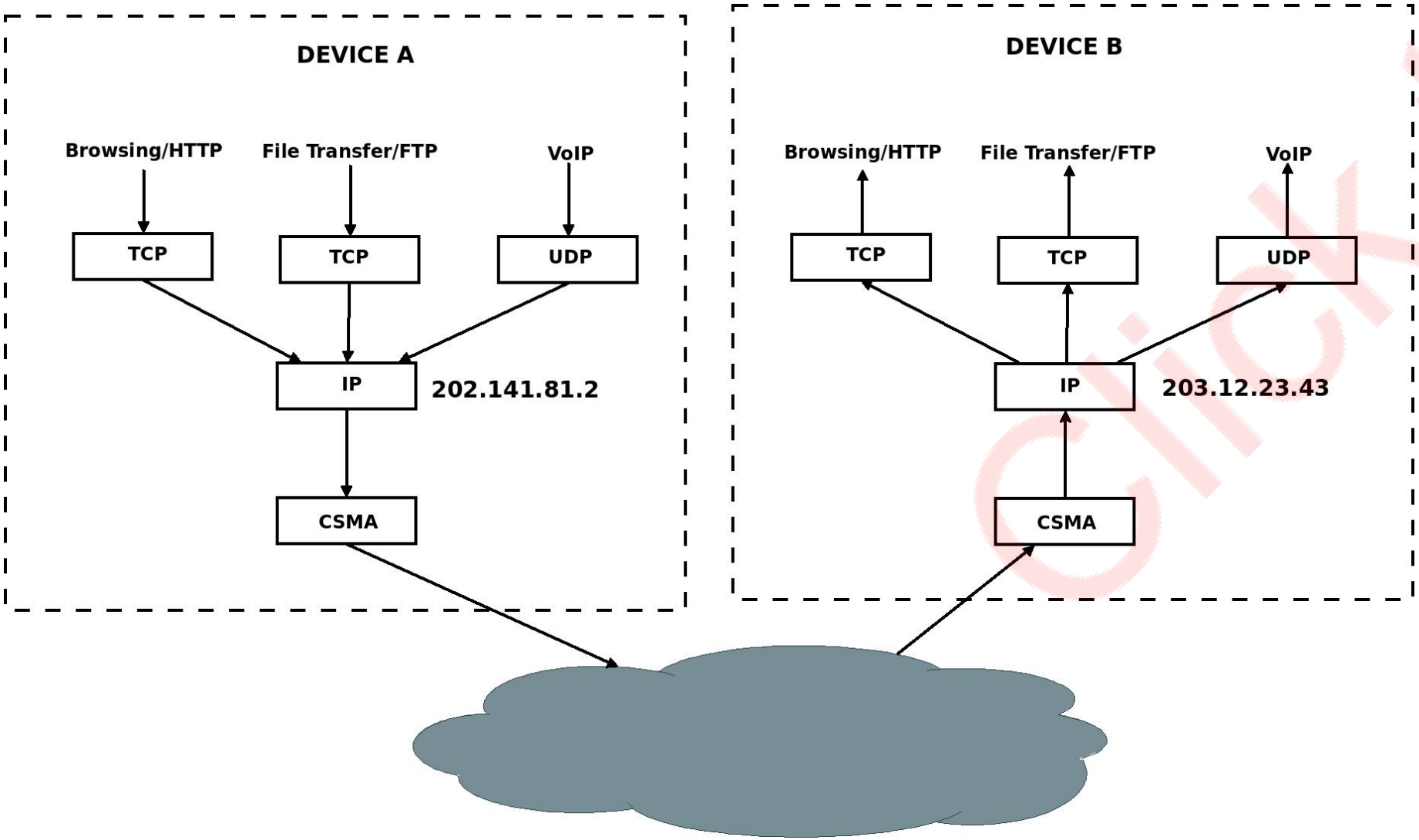


Check the `net` module (download Kernel source and check `/usr/src/linux/net`)!

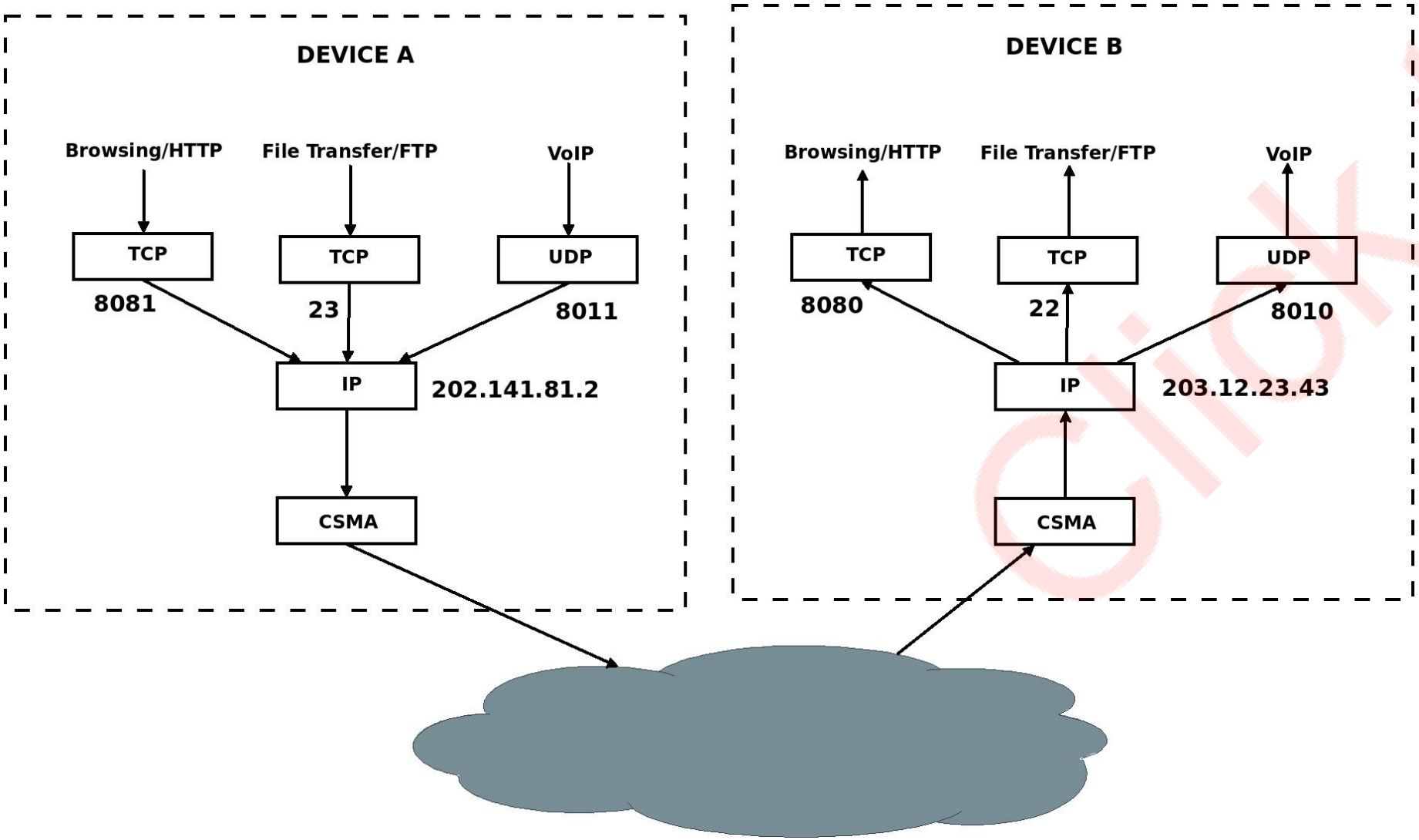
Application Multiplexing in TCP/IP



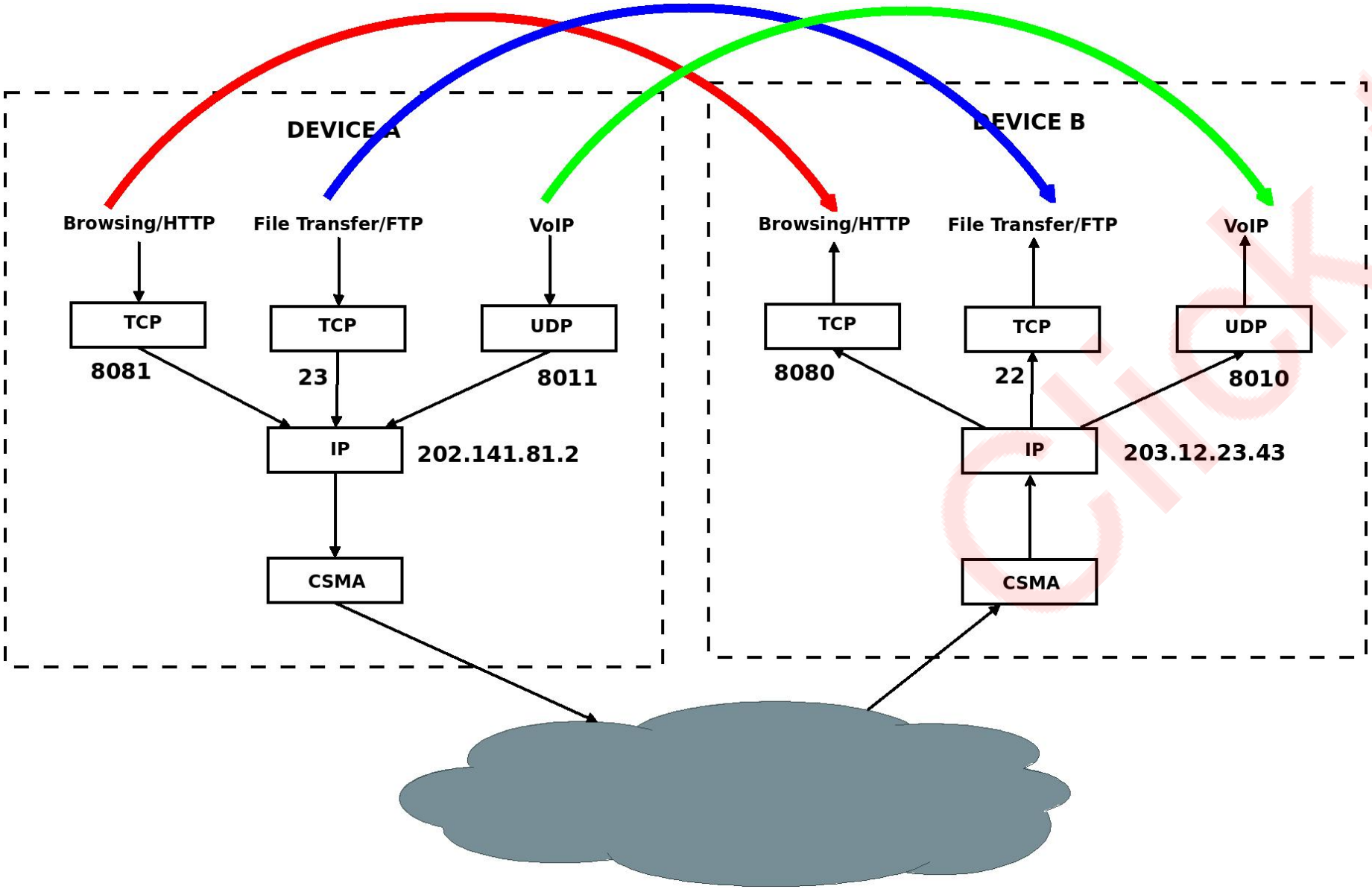
Application Multiplexing in TCP/IP



Application Multiplexing in TCP/IP

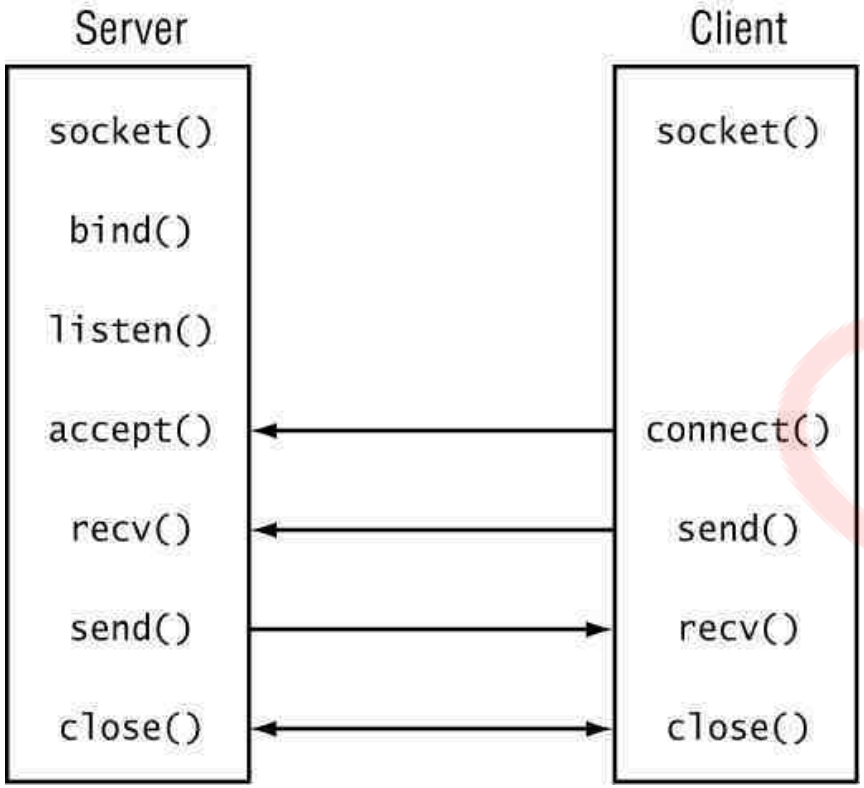


What are Sockets?



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**.

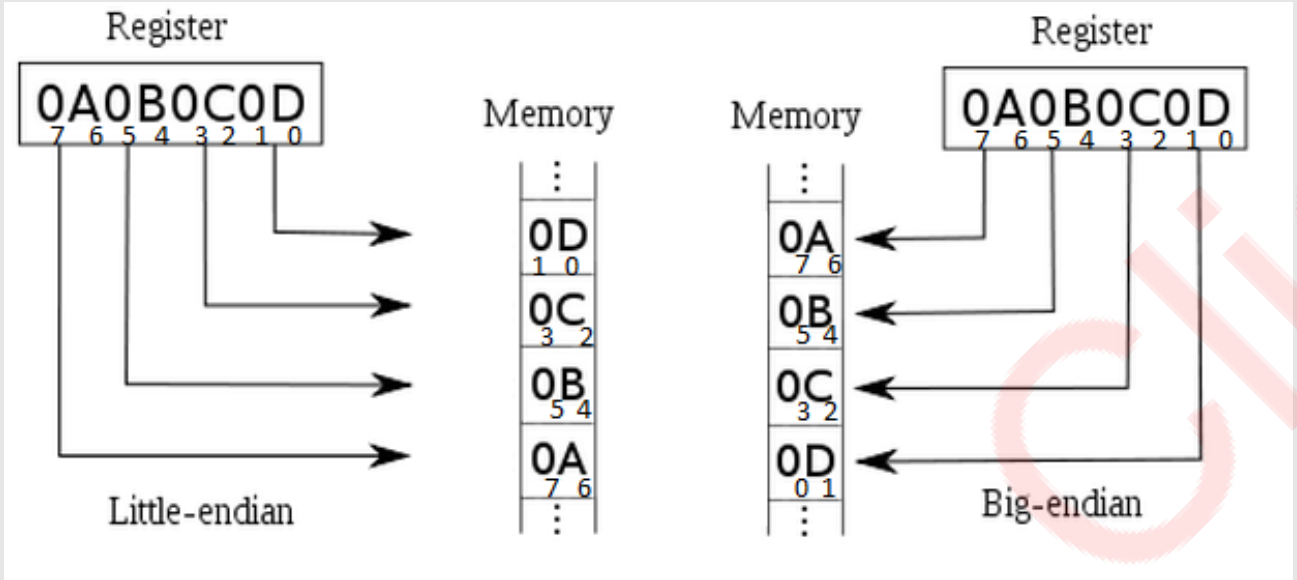
```
 struct sockaddr_in serveraddr;
int port = 3028;
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = INADDR_ANY;
serveraddr.sin_port = htons(port);
```

Host Byte Order to Network Byte Order - Why?

- Little Endian and Big Endian System

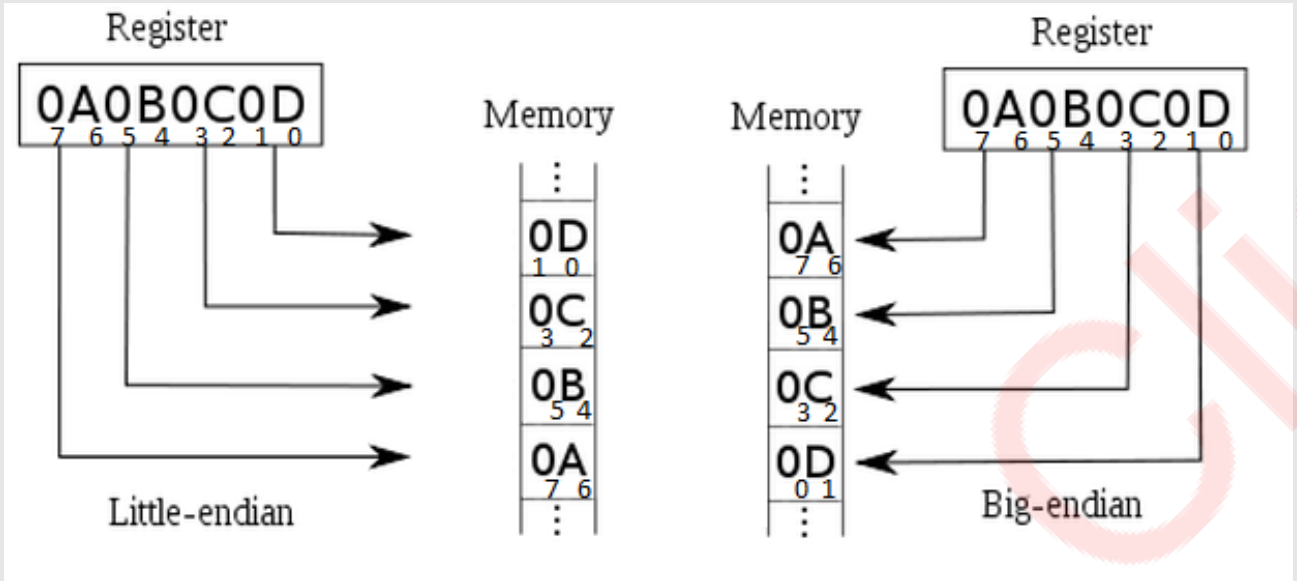
Host Byte Order to Network Byte Order - Why?

- Little Endian and Big Endian System



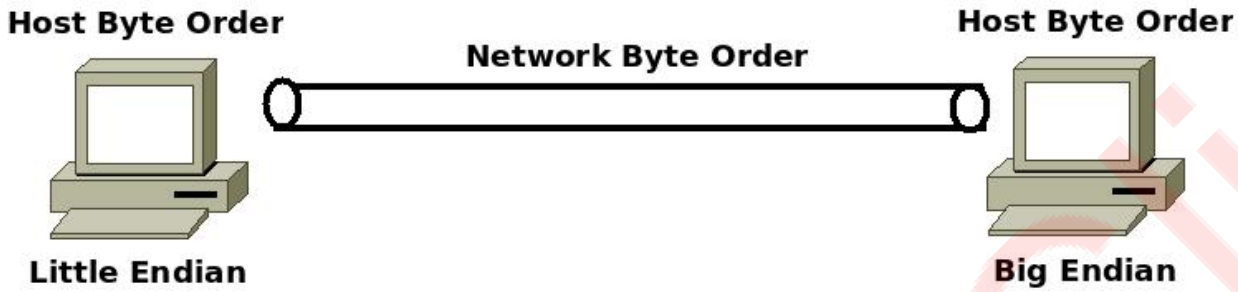
Host Byte Order to Network Byte Order - Why?

- 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

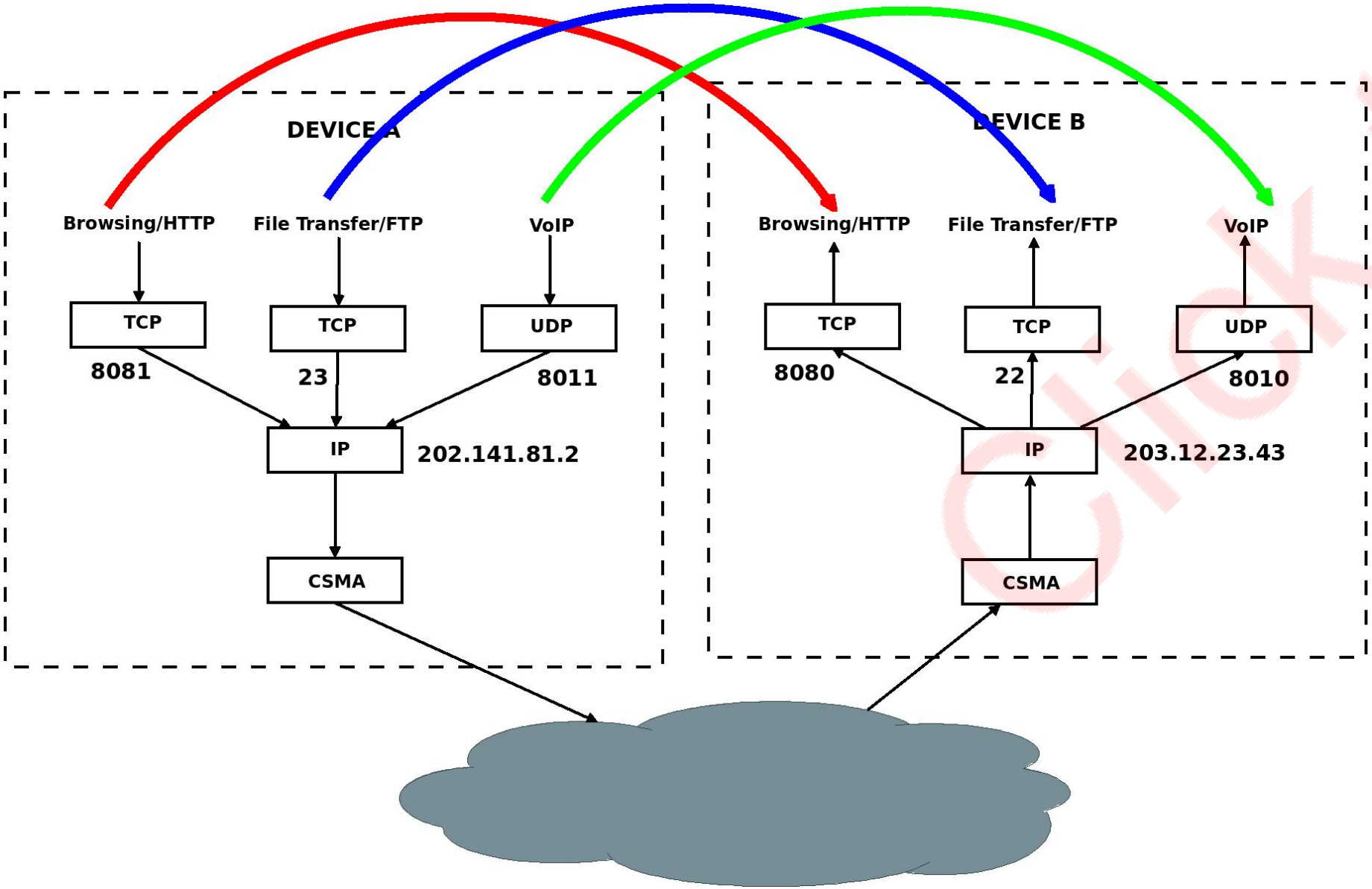
Sandip Chakraborty

Department of Computer Science and Engineering,

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

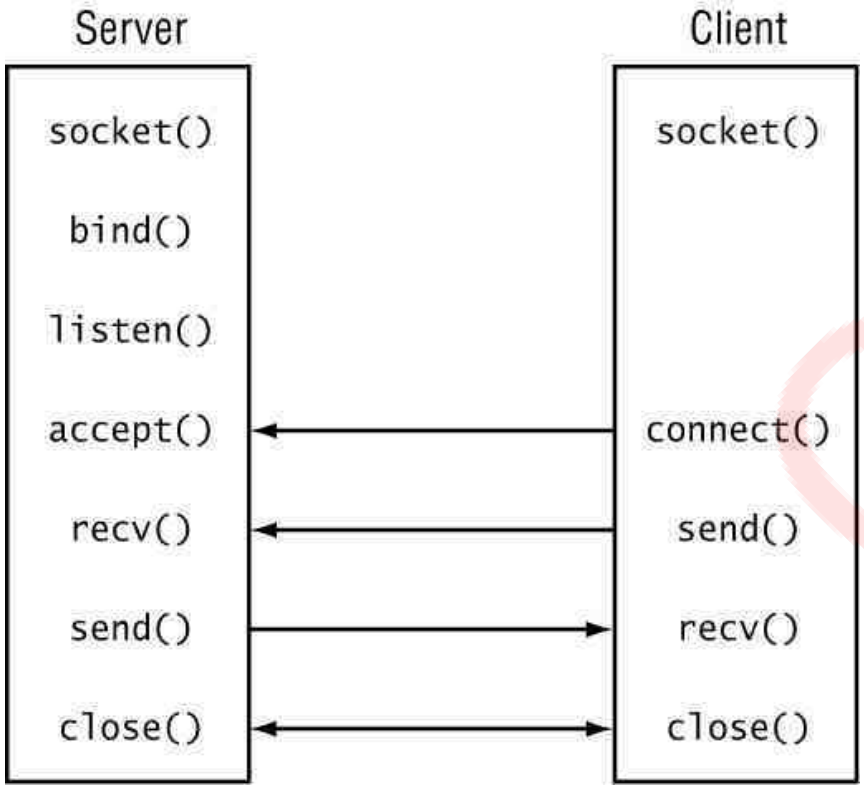
August 8, 2018

What are Sockets?

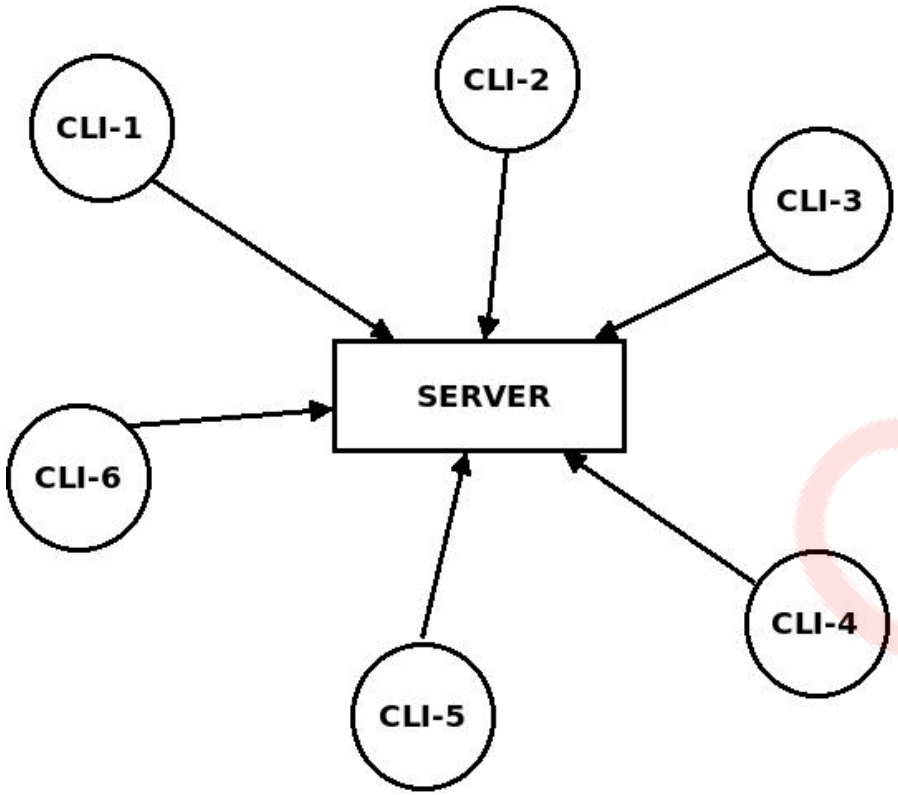


Socket Programming Framework/API

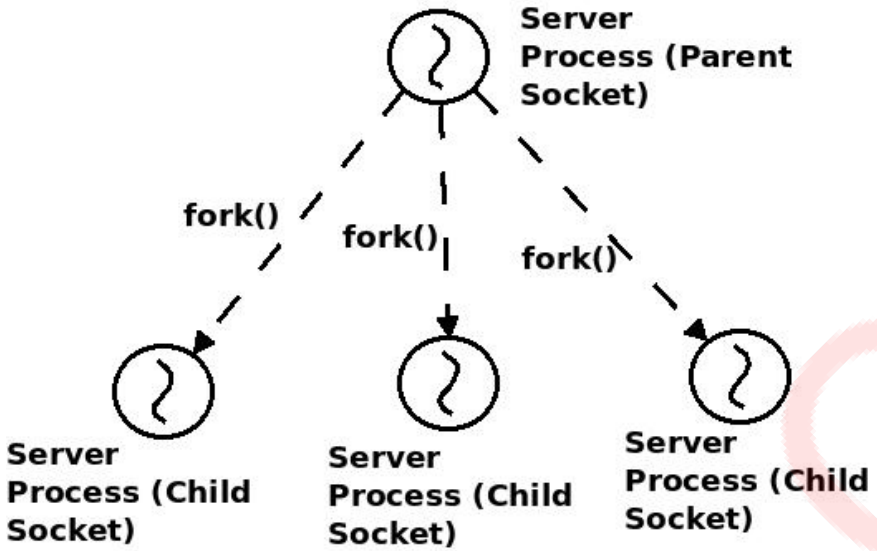
A set of **system calls** to get the service from TCP/IP protocol stack (`net` module in the OS kernel).



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.
        */
```

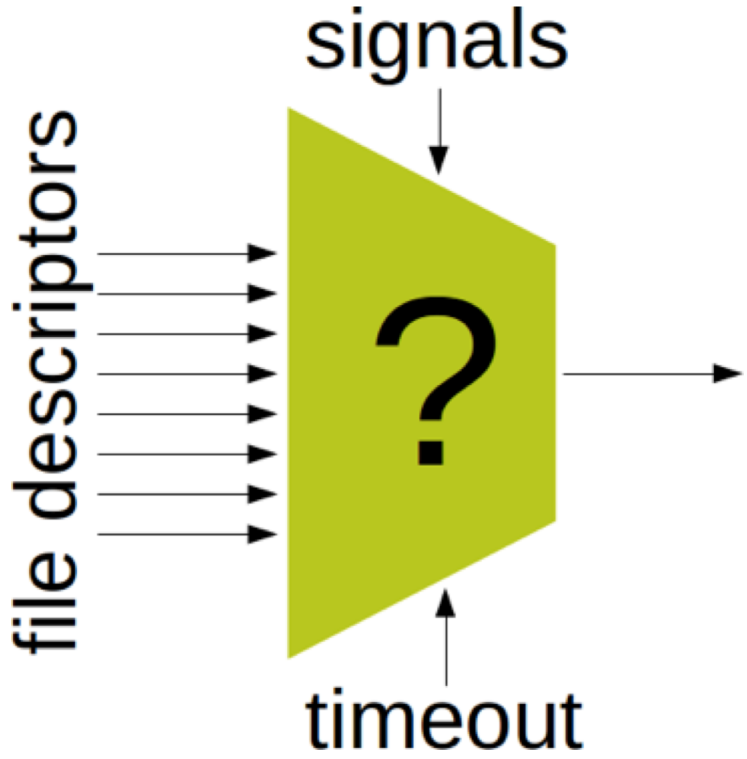
Thank You

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

Hello Bob, What's up

Hi Bob: How are you?

Hi Alice

Hi Jane, Bob is there

Peer to Peer Chat Application

- No central server to control chat message delivery
- Every user run's 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

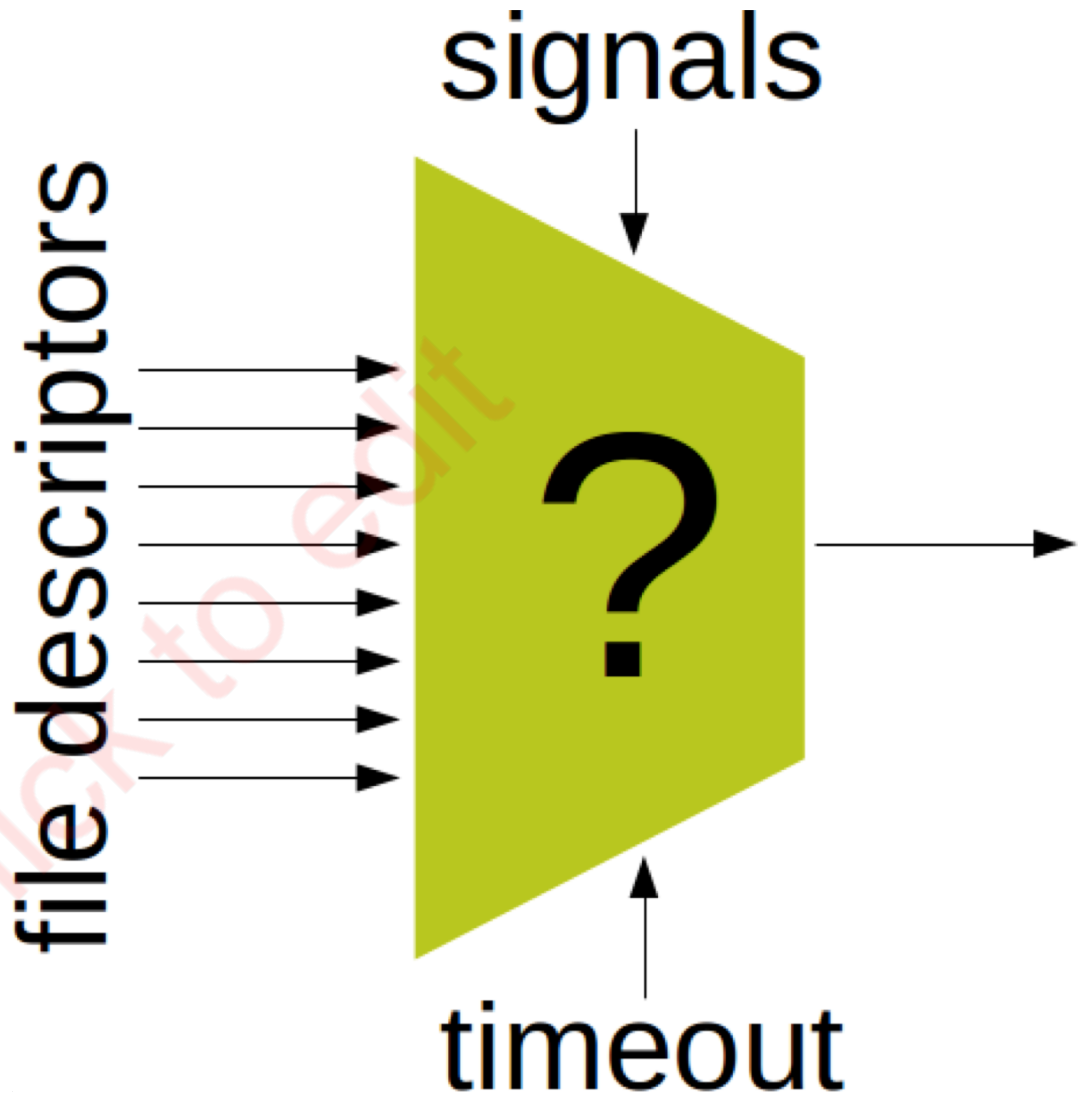
The select() system call

- 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

Read file descriptors

Write file descriptors

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

exception 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
- ***writefds*** 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_SET**s 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