ADVANCED COMPUTER NETWORKS

20IMCAT404

MODULE II

Syllabus
Introduction to Networks

- Transport layer protocols - Introduction to transport layer
- Multiplexing and Demultiplexing - Principles of reliable data transfer
- Stop-and-wait and Go-back- N design and evaluation
- Connection oriented transport TCP - Connectionless transport UDP
- Principles of congestion control - Efficiency and fairness

Transport Layer

- The transport layer is responsible for process-to-process delivery of the entire message.
- A process is an application program running on a host.
- Whereas the network layer oversees source-to-destination delivery of individual packets, it does not recognize any relationship between those packets.
- It treats each one independently, as though each piece belonged to a separate message, whether or not it does.
- The transport layer, on the other hand, ensures that the whole message arrives intact and in order, overseeing both error control and flow control at the source-to-destination level.

Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on and enhances network layer services

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Functions/services/responsibilities of transport layer

- **Service-point addressing**
- **TCP provides reliable data transfer.**
- **Segmentation and reassembly**
- **Connection control**
- **Flow control**
- **Error control**
- **Transport-layer multiplexing and demultiplexing.**

Functions/services/responsibilities of transport layer

Service-point Addressing:

- Computers often run several programs at the same time. For this reason, source-to-destination delivery means delivery not only from one computer to the next but also from a specific process (running program) on one computer to a specific process (running program) on the other.
- The transport layer header must therefore include a type of address called a service-point address (or port address).
- The network layer gets each packet to the correct computer; the transport layer gets the entire message to the correct process on that computer.

Functions/services/responsibilities of transport layer

Segmentation and Reassembly

- A message is divided into transmittable segments, with each segment containing a sequence number.
- These numbers enable the transport layer to reassemble the message correctly upon arriving at the destination and to identify and replace packets that were lost in transmission.

Functions/services/responsibilities of transport layer

Connection control

- The transport layer can be either connectionless or connection oriented.
- A connectionless transport layer treats each segment as an independent packet and delivers it to the transport layer at the destination machine.
- A connection oriented transport layer makes a connection with the transport layer at the destination machine first before delivering the packets. After all the data are transferred, the connection is terminated.

Functions/services/responsibilities of transport layer

Flow control.

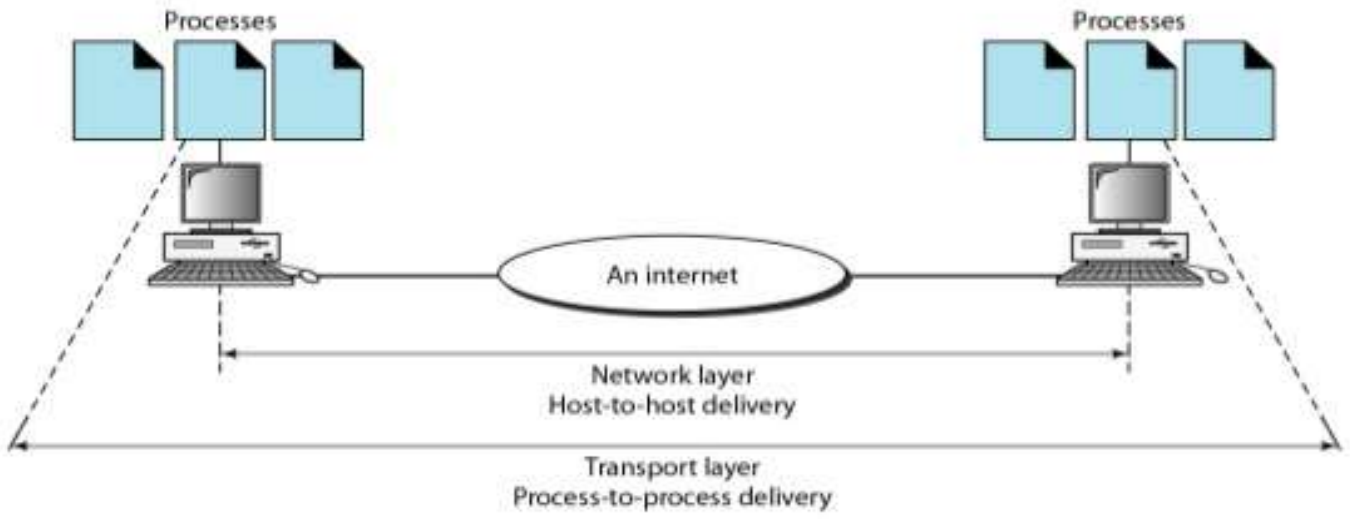
- Like the data link layer, the transport layer is responsible for flow control. However, flow control at this layer is performed end to end rather than across a single link.

Error control.

- Like the data link layer, the transport layer is responsible for error control.
- However, error control at this layer is performed process-to process rather than across a single link.
- The sending transport layer makes sure that the entire message arrives at the receiving transport layer without error (damage, loss, or duplication).
- Error correction is usually achieved through retransmission.

Process to process/Host to Host

Figure 2.11 *Reliable process-to-process delivery of a message*



Multiplexing and Demultiplexing

- A process (*as part of a network application*) can have one or more **sockets** - *doors through which data passes from the network* to the **process** and through which ***data*** passes from the process to the network.

- **The transport layer in the receiving host**
  - does not deliver data directly to a ***process***
  - but instead to an *intermediary socket*.
  - Because at any given time **there can be more than one socket** in the receiving host, each socket has a **unique identifier**.
  - The format of the identifier depends on whether the **socket** is a **UDP** or a **TCP socket**,

Multiplexing and Demultiplexing

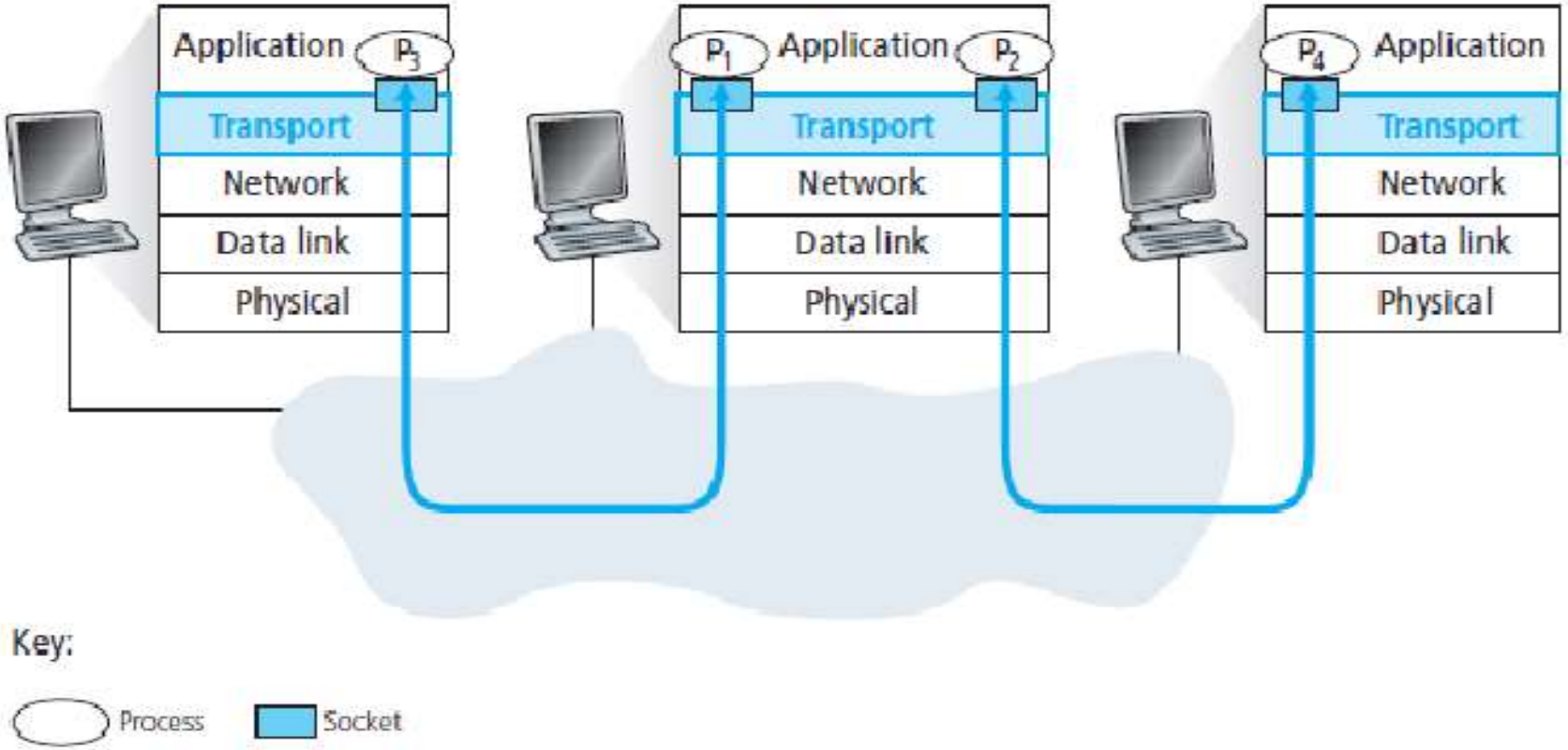


Figure 3.2 ♦ Transport-layer multiplexing and demultiplexing

Multiplexing and Demultiplexing

- How a receiving host directs an incoming transport-layer **segment** to the appropriate ***socket***?????
- Each transport-layer segment
  - has a **set of fields** in the segment for this purpose.
- At the receiving end,
  - the *transport layer* examines *these fields to* ***identify the receiving socket*** and then ***directs the segment*** to that *socket*.
  - This job of *delivering the data in a transport-layer segment to the correct socket* is called **demultiplexing.**
  - ***The job of gathering data chunks at the source host from different*** *sockets,*
  - *encapsulating each data chunk with header information* (that will later be used in demultiplexing)
  - To **create segments**, and *passing the segments to the network layer* is called **multiplexing.**

Multiplexing and Demultiplexing

- **The transport layer in the middle host in Figure (Previous)**
  - must **demultiplex segments arriving from the network layer below to either process P1 or P2 above**

  This is done by directing the arriving segment's data to the corresponding process's socket.

  The transport layer in the middle host must also **gather outgoing data from these sockets**, form transport-layer segments, and pass these segments down to the network layer.
- The roles of transport-layer multiplexing and demultiplexing:
  - we know that transport-layer multiplexing requires

    (1) that sockets have **unique identifiers**, and

    (2) that each segment have **special fields** that indicate the *socket to which the segment is to be delivered*. These special fields are the **source port number field and the destination port number field.**

  - (The UDP and TCP segments have other fields as well) **Each port number is a *16-bit number*, ranging from *0 to 65535*.**

Multiplexing and Demultiplexing

- The port numbers ranging **from 0 to 1023** are called **well-known port numbers,** and are restricted, which means that they are reserved for use by well-known application protocols such as HTTP (which uses port number **80**) and FTP (which uses port number **21**).
- The list of well-known port numbers is given in RFC 1700 and is updated at http://www.iana.org [RFC 3232].
- When we develop a new application, we must assign the application a port number.
- It should now be clear how the transport layer *could implement the* ***demultiplexing*** service:

- Each socket in the host could be assigned a port number, and
  - when a **segment** arrives at the host,
    - the transport layer examines the **destination port number** *in the segment* **directs the segment to the corresponding socket**.

- **The segment's data then passes through the socket into the attached process.**

Connectionless Transport UDP

Connectionless Multiplexing and Demultiplexing

Connectionless Transport UDP
Connectionless Multiplexing and Demultiplexing

- We can now precisely describe UDP multiplexing/demultiplexing.
- Suppose a process in **Host A** with UDP port **19157**, wants to send a chunk of application data to a process with UDP port **46428** in **Host B**
- The transport layer in **Host A**
  - creates a transport-layer segment that includes the *application data*
    - **the source port number (19157), the destination port number (46428)**
- The transport layer then passes the resulting segment to the network layer.
- The network layer encapsulates
  - the segment in an IP datagram and makes a best-effort attempt to deliver the segment to the receiving host.
- If the segment arrives at the receiving Host B,
  - the transport layer at the receiving host examines the destination port number in the segment (46428) and
    - delivers the segment to its socket identified by port 46428.

Connectionless Multiplexing and Demultiplexing

- Host B could be running **multiple processes**, each with its own UDP socket and associated port number.
- As UDP segments arrive from the network,
  - **Host B directs (demultiplexes) each segment to the appropriate socket by examining the segment's destination port number.**
- A UDP socket is fully identified by
  - a **two-tuple** consisting of a **destination IP address** and a **destination port number.**
- As a consequence,
  - if **two UDP segments have different source IP addresses and/or source port numbers**,
    - but have the ***same destination IP address and destination port number, then the two segments will be directed to the same destination process*** via the same destination socket.

Connectionless Multiplexing and Demultiplexing

- In the A-to-B segment
  - the **source port number** serves as part of a “return address”—when B wants to send a segment back to A,
  - the **destination port** in the ***B-to-A segment*** will *take its value* ***from the source port value*** *of the A-to-B segment*.

Connection-Oriented Transport TCP

Connection-Oriented Multiplexing and Demultiplexing

Connection-Oriented Multiplexing and Demultiplexing

- **TCP demultiplexing**
  - ***Depends on TCP sockets and TCP connection establishment***.
- One main difference between a TCP socket and a UDP socket is that
  - a **TCP socket** is **identified by a four-tuple:** (**source IP address, source port number, destination IP address, destination port number**).
- **when a TCP segment arrives from the network to a host:**
  - **the host uses all four values to direct (demultiplex) the segment to the appropriate socket.**
- **Two arriving TCP segments**
  - **with different *source IP addresses or source port numbers* will be directed to two different sockets.**

Connection-Oriented Multiplexing and Demultiplexing

- The TCP server application has a "welcoming socket,"
  - that **waits for connection establishment requests from TCP clients** say, port number **12000.**
- The **TCP client creates a socket and sends a connection establishment request segment** with the lines:
- ***A connection-establishment request a***
  - **TCP segment with destination port number 12000 and a special connection-establishment bit set in the TCP header.**
- The segment also **includes a source port number** that was chosen by the client.
- When the **host** operating system of the computer running the **server process**
  - receives the incoming **connection-request segment** with **destination port 12000**, it locates **the server process** that is **waiting to accept a connection on port number 12000**.
- The server process then creates a new socket.

Connection-Oriented Multiplexing and Demultiplexing

- The transport layer at the server notes the following **four values** in the connection- request segment:
  - (1) the source port number in the segment
  - (2) the IP address of the source host
  - (3) the destination port number in the segment
  - (4) its own IP address.
- The newly created connection socket is identified by these four values;
  - all subsequently arriving segments whose source port, source IP address, destination port, and destination IP address match these four values will be demultiplexed to this socket.
- With the TCP connection now in place,
  - the client and server can now send data to each other.

Connection-Oriented Multiplexing and Demultiplexing

- The server host may support many simultaneous TCP connection sockets,
  - with each **socket attached to a process,**
  - each **socket identified by its own four tuple.**

- When a **TCP segment** arrives at the **host**,
  - all four fields (source IP address, source port, destination IP address, destination port) are used to **direct (demultiplex) the segment to the appropriate socket**.

Two clients, using the same destination port number (80) to communicate with the same Web server application

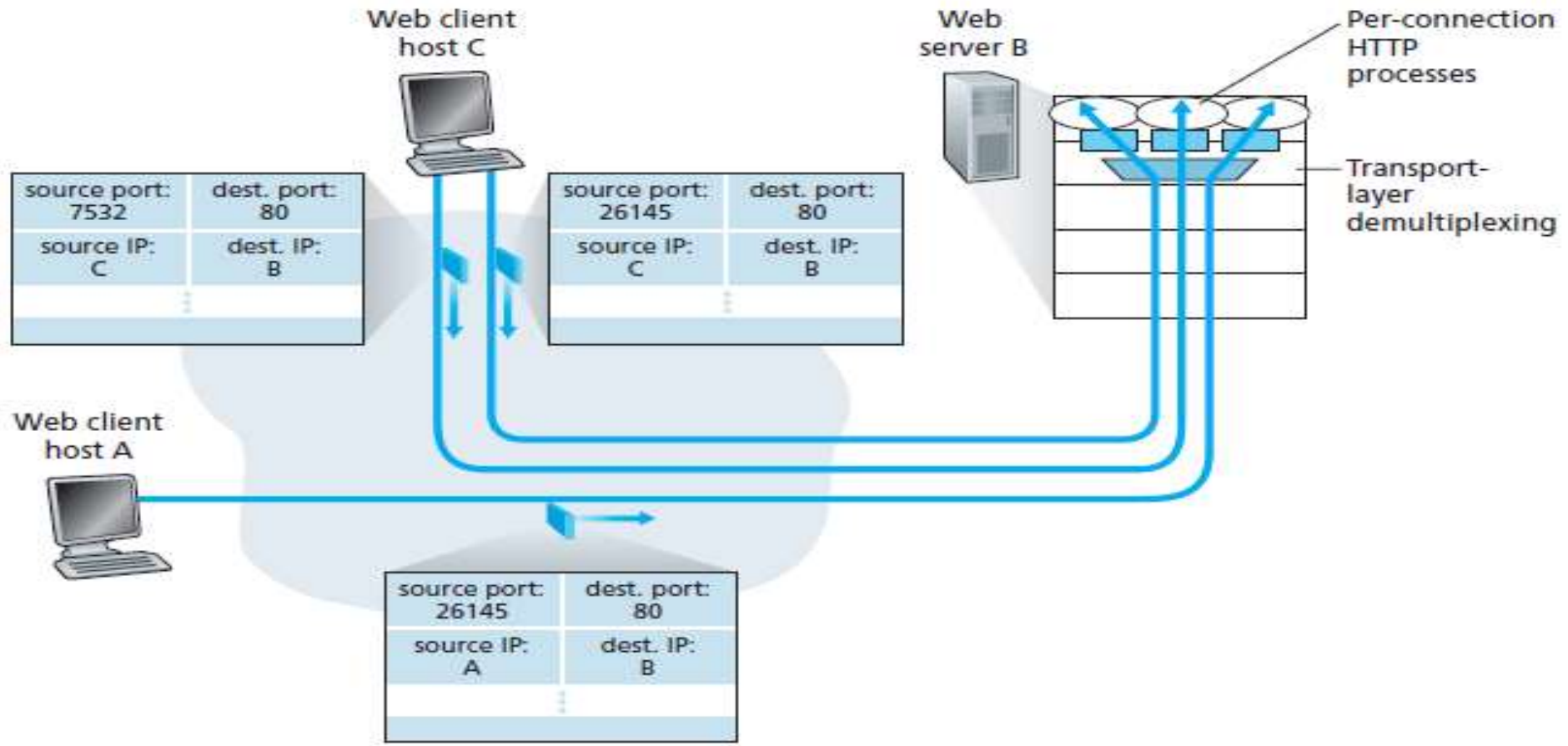


Figure 3.5 ♦ Two clients, using the same destination port number (80) to communicate with the same Web server application

Connection-Oriented Multiplexing and Demultiplexing

- Host C initiates two HTTP sessions to server B, and Host A initiates one HTTP session to B.
- Hosts A and C and server B each have their own unique IP address—A, C, and B, respectively.
- Host C assigns two different source port numbers (**26145 and 7532**) to its two HTTP connections.
- Because Host A is choosing source port numbers independently of C, it might also assign a source port of **26145 to its HTTP connection**.
- But this is not a problem—server B will still be able to correctly demultiplex the two connections having the same source port number, since the two connections have different source IP addresses.

Principles of Reliable Data Transfer

Principles of Reliable Data Transfer

- The service abstraction provided to the upper-layer entities is that of a reliable channel through which data can be transferred.
- With a reliable channel,
  - **no transferred data bits are corrupted** (flipped from 0 to 1, or vice versa) or lost, and **all are delivered in the order in which they were sent.**
- This is precisely the *service model* offered by **TCP**
  - to the Internet applications that invoke it.
- It is the responsibility of a
  - **reliable data transfer protocol to implement this** service **abstraction**.
  - This task is made difficult by the fact that *the layer below the reliable data transfer protocol may be unreliable.*
- For example, **TCP is a reliable data transfer protocol** that is **implemented on top of an** *unreliable (IP)* **end-to-end network layer.**
  - More generally, the layer beneath ***the two reliably communicating end points*** might consist of a single **physical link(link-level data transfer protocol**)
  - or **a global internetwork (a transport-level protocol**).
- This lower layer simply as an **unreliable point-to-point channel**.

Principles of Reliable Data Transfer

- Packets will be delivered
  - in **the order in which they were sent, with some packets possibly being lost**;
  - i.e, the *underlying channel will not reorder packets*.
- Figure 3.8(b)(BELOW☺) illustrates the interfaces for our **data transfer protocol.**
  - The **sending side of the data transfer protocol** will be invoked from above by a call to **rdt_send().**
  - It will pass the data to be delivered to
    - **upper layer at the receiving side**. (Here **rdt stands for *reliable data transfer protocol*)**

Principles of Reliable Data Transfer

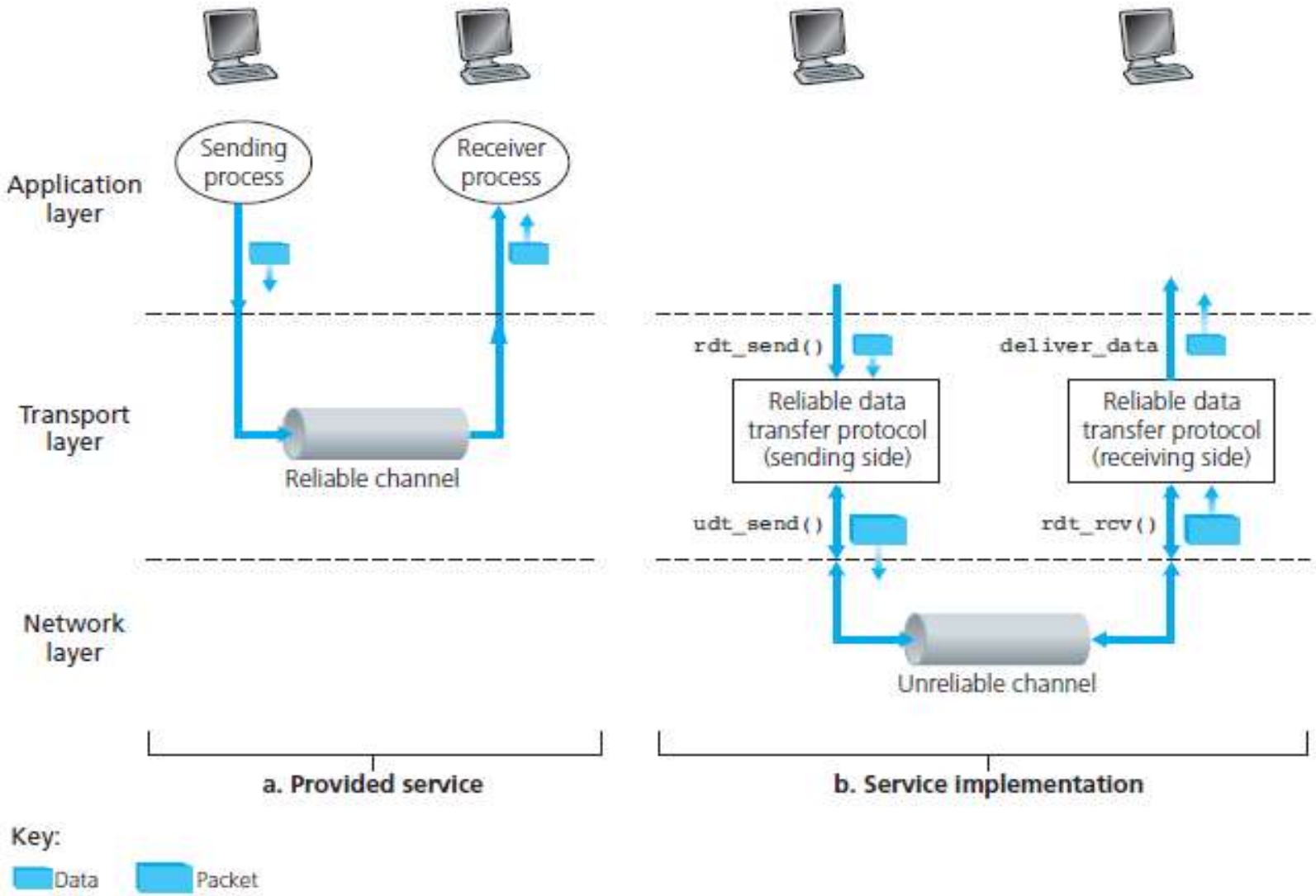


Figure 3.8 ♦ Reliable data transfer: Service model and service Implementation

Principles of Reliable Data Transfer

- The sending side
    - **rdt is being called.**
- On the receiving side, **rdt_rcv()** will be called
    - *when a packet arrives from the **receiving side** of the channel.*
- When the **rdt protocol** wants to **deliver data to the upper layer**,
    - it will do so by calling **deliver_data().**
    - In the following we use the terminology **“packet”** rather than transport-layer ***“segment”***
- In the case of **unidirectional data transfer,**
    - data transfer from the sending to the receiving side.

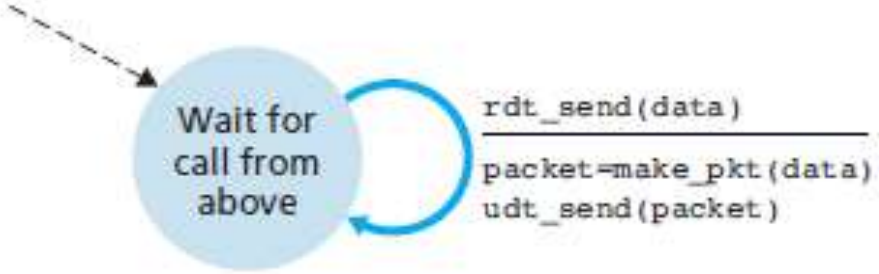
- In reliable **bidirectional** (that is, full-duplex) **data transfer**
    - *sending and receiving sides of **rdt*** will need to **exchange control packets** back and forth.
    - Both the **send and receive sides of rdt send packets**
        - to the other side by a call to **udt_send()** (*udt- unreliable data transfer)*

Building a Reliable Data Transfer Protocol

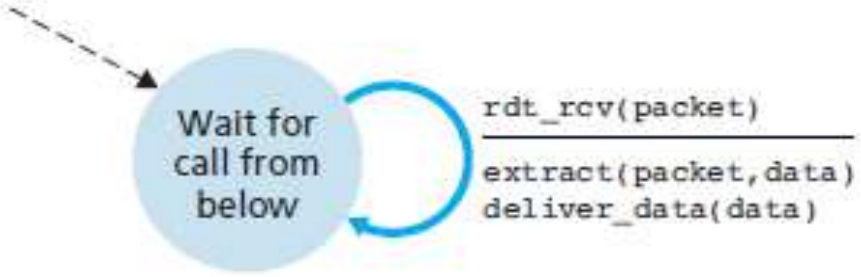
- **Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0**
  - Here underlying channel is *completely reliable.*
  - The **protocol** itself, rdt1.0, is **trivial.**
  - The **finite-state machine (FSM)** definitions for the *rdt1.0 sender and receiver* are shown in Figure (follows)
- The FSM in Figure (a)(follows)
  - defines the **operation of the sender**
  - FSM in Figure(b) (follows) defines the **operation of the receiver**.

  There are ***separate FSMs*** *for the sender and for the receiver.*
- *The **sender and** **receiver FSMs*** have just one state.
- The **arrows** in the FSM description indicate the **transition** of the **protocol** from **one state to another**.

Building a Reliable Data Transfer Protocol



a. rdt1.0: sending side



b. rdt1.0: receiving side

Figure 3.9 ♦ rdt1.0 – A protocol for a completely reliable channel

Building a Reliable Data Transfer Protocol

- The sending side of rdt simply
  - **accepts data from the upper layer** via the
    - **rdt_send(data)** *event, creates a packet containing the data* (via the action) **make_pkt(data))** and *sends the packet into the channel*.
    - **rdt_send(data)** event would result from a procedure call. Eg: **rdt_send()** by the upper-layer application.
- On the receiving side,
  - rdt receives a packet from the underlying channel via **rdt_rcv(packet) event**,
    - *removes the data from the packet* (via the action extract (packet, data))

    And passes the data up to the upper layer (via the action deliver_data(data)).
    - the *rdt_rcv(packet) event would result from a procedure call* (for example, to rdt_rcv()) from the lowerlayer protocol.

Building a Reliable Data Transfer Protocol

- All packet flow is from the sender to receiver;
  - with a reliable channel
  - there is *no need for the receiver side to provide any feedback* to the sender since nothing can go wrong.
- The **receiver** is able to
  - **receive data as fast as the sender happens to send data**.
  - Thus, there is **no need for the receiver** to **ask the sender** to slow down.
- A more realistic model - in which ***bits in a packet may be corrupted***.
- Such bit errors occur in the
  - physical ***components*** of a network *as a packet is* ***transmitted, propagates, or is buffered***.

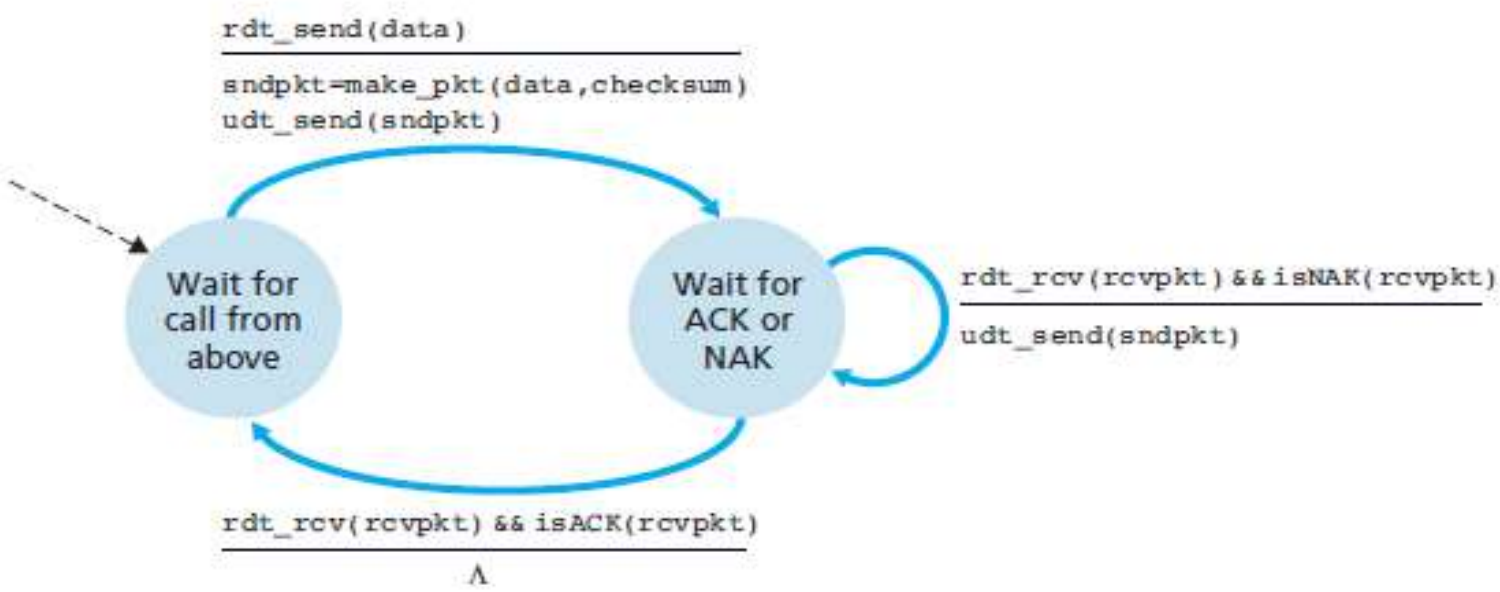
Building a Reliable Data Transfer Protocol

- Assume that all **transmitted packets are received in the order** *in which they were sent.* (***although their bits may be corrupted***)
- For reliably communicating over a channel:
  - first consider *how people might deal with such a situation.*
- Consider how you yourself might dictate a long message over the phone.
  - In a typical scenario, the message taker might say **"OK"** after each sentence
    - has been heard, understood, and recorded.
  - If the message taker hears a **garbled sentence**, you're asked to **repeat** the garbled sentence.
- This message-dictation protocol uses
  - **positive acknowledgments** ("OK")
  - **negative acknowledgments ("Please repeat that.").**

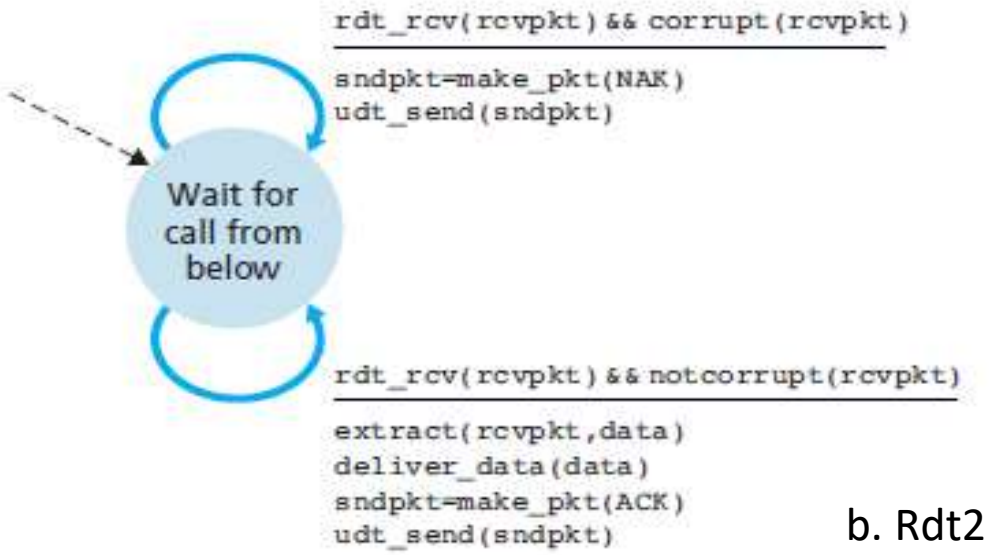
These control messages allow the receiver to let the sender
- know what has been received correctly,
- what has been received in error
- and thus requires repeating.

- In a computer network setting,
  - reliable data transfer protocols based on such **retransmission** are known as
    - **ARQ (Automatic Repeat reQuest) protocols**.

Building a Reliable Data Transfer Protocol



a. rdt2.0: sending side



b. Rdt2.0: Receiving Side

Building a Reliable Data Transfer Protocol

- The send side of **rdt2.0** has two states.
- In the leftmost state,
  - the **send-side protocol is waiting for data to be passed down from the upper layer**.
- When the **rdt_send(data)** event occurs,
  - **the sender will create a packet (sndpkt) containing the data to be sent**, along **with a packet checksum** and then send the packet via the **udt_send(sndpkt)** operation.
- In the rightmost state,
  - the sender protocol is waiting for an **ACK or a NAK** packet from the receiver.

Building a Reliable Data Transfer Protocol

- Protocol rdt2.0:
  - *The ACK or NAK packet could be corrupted*
  - Before proceeding on, you should think about how this problem may be fixed.
  - Minimally, we will need to **add checksum bits to ACK/NAK packets in order to detect such errors**.

  The more difficult question is "How *the protocol should recover from errors in ACK or NAK packets*???"

- The difficulty here is that **if an ACK or NAK is corrupted**,
  - ***the sender has no way of knowing whether or not the receiver has correctly received the last piece of transmitted data***.

Building a Reliable Data Transfer Protocol

- Consider three possibilities for handling corrupted ACKs or NAKs:
- If the *speaker* didn't understand the
  - "**OK**" or "**Please repeat that**" reply from the *receiver*,
  - the speaker would probably ask, "**What did you say**?" (thus introducing a new type of **sender-to-receiver packet** to our protocol).
- The **receiver would then repeat** the reply.
- But what if the speaker's
  - **"What did you say?" is corrupted?**
  - The receiver, having no idea whether the garbled sentence was part of the dictation
  - or a request to repeat the last reply, would probably then respond with **"What did *you say?"***
  - *And then, of course, that response* might be garbled.
  - Clearly, **we're heading down a difficult path.**
  - A second **alternative is to add enough checksum bits to allow the sender not only to detect, but also to recover from, bit errors**.
  - This solves the immediate problem for a channel that can corrupt packets but not lose them.

Building a Reliable Data Transfer Protocol

- A solution to this new problem :
  - is to add a new field to the data packet and have the sender number its data packets by putting
    - a **sequence number into this** field.
  - The **receiver then need only check this sequence number to determine**
    - **whether or not the received packet is a retransmission.**
  - For this simple case of a **stop-and wait protocol,**
    - **a 1-bit sequence number will suffice,**
    - **since it will allow the receiver to know whether the sender is resending the previously transmitted packet**
  - **(*the sequence number of the received packet has the same sequence number as the most recently received packet*)**
  - or a new packet (the sequence number changes)
  - Assuming a channel that does not lose packets, **ACK and NAK packets** do not themselves need to indicate the sequence number of the packet they are acknowledging.
  - The **sender** knows that a received **ACK or NAK** packet (whether **garbled or not**) was generated in response to its most recently transmitted data packet.

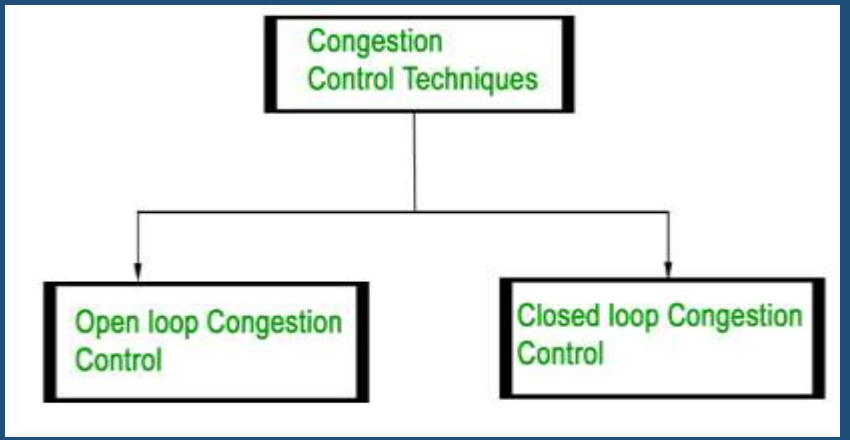
Principles of congestion control - efficiency and fairness

What is congestion?

- Congestion in the network occur if the load on the network, ie **the number of packets sent to the network is greater than the capacity of the network.** ie the number of packets a network can handle.
- Congestion in a network occurs because routers and switches have **input queues** and **output queues** that hold the packet before processing.
- The incoming packet is put at the end of the queue while waiting for processing.
- The packet at the front of the queue will be processed.
- After processing, the packet will be put in the output queue and waits its turn to be sent.
- If the rate of the packet arrival is greater than the packet processing rate, the input queue becomes congested. If the packet departure is less than the packet processing rate, the output queue becomes congested. All these results congestion.

Causes of Congestion:

- Packet arrival rate exceeds the outgoing link capacity.
- Insufficient memory to store arriving packets
- Bursty traffic
- Slow processor



Congestion control means keep the load below capacity.

Congestion control refers to the techniques used to control or prevent congestion.

Congestion control techniques can be broadly classified into two categories:

- **Open Loop Congestion Control**
- **Closed Loop Congestion Control**

Open Loop Congestion Control

- Open loop congestion control policies are applied to **prevent congestion before it happens.** The congestion control is handled either by the source or the destination.

- **Policies adopted by open loop congestion control are:**
  - Retransmission policy
  - Window policy
  - Acknowledgement policy
  - Discarding policy
  - Admission policy

Open Loop Congestion Control

1. Retransmission Policy

It is the policy in which retransmission of the packets are taken care. If the sender feels that a sent packet is lost or corrupted, the packet needs to be retransmitted. This transmission may increase the congestion in the network. To prevent congestion**,** **retransmission timers** must be designed to prevent congestion and also able to optimize efficiency.

2. Window Policy

The type of window at the sender side may also affect the congestion. Several packets in the Go-back-n window are resent, although some packets may be received successfully at the receiver side. This duplication may increase the congestion in the network and making it worse. Therefore, **Selective repeat window** should be adopted as it sends the specific packet that may have been lost.

Open Loop Congestion Control

3. Discarding Policy

A good discarding policy adopted by the routers is that the routers may prevent congestion and at the same time **partially discards the corrupted or less sensitive package** and also able to maintain the quality of a message. In case of audio file transmission, routers can discard less sensitive packets to prevent congestion and also maintain the quality of the audio file.

4. Acknowledgment Policy

Since acknowledgements are also the part of the load in network, the acknowledgment policy imposed by the receiver may also affect congestion. Several approaches can be used to prevent congestion related to acknowledgment. **The receiver should send acknowledgement for N packets rather than sending acknowledgement for a single packet.** The receiver should send a acknowledgment only if it has to sent a packet or a timer expires.

Open Loop Congestion Control

5. Admission Policy

In admission policy a mechanism should be used to prevent congestion. Switches in a flow should first check the resource requirement of a network flow before transmitting it further. If there is a chance of congestion or there is congestion in the network, router should deny establishing a virtual network connection to prevent further congestion.

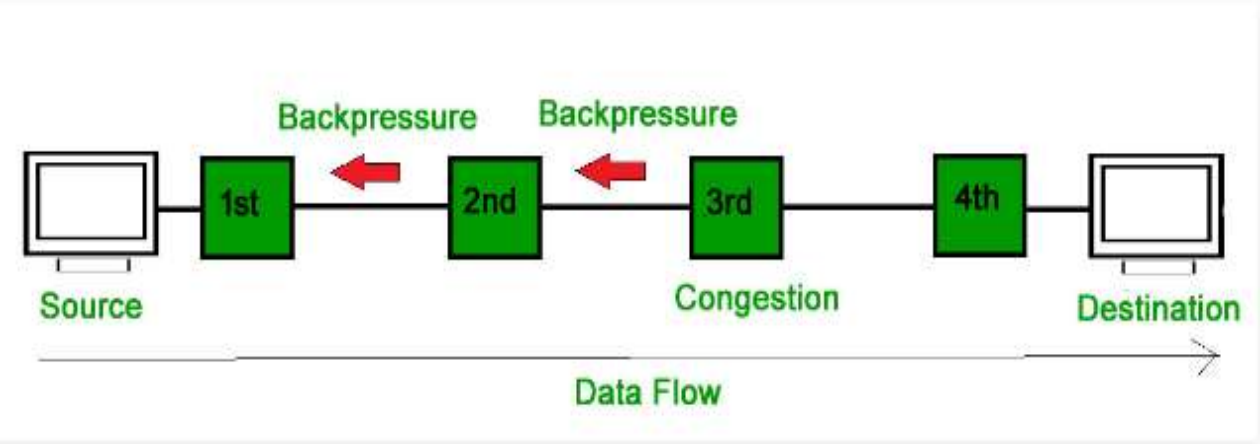
- All the above policies are adopted to prevent congestion before it happens in the network.

Closed Loop Congestion Control

- Closed loop congestion control technique is used to remove congestion after it happens.
- Several techniques are used by different protocols; some of them are:
    - Backpressure
    - Choke packet
    - Implicit signaling
    - Explicit signaling

Closed Loop Congestion Control

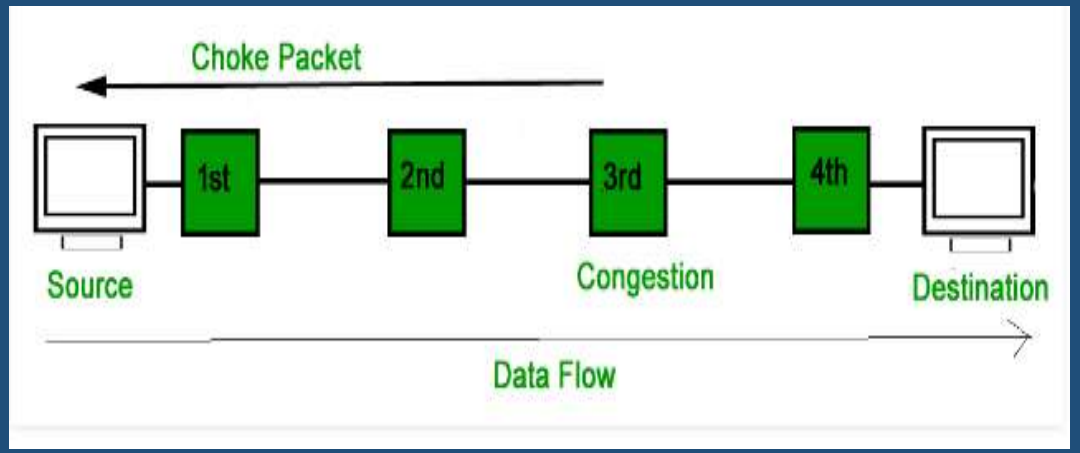
1. Backpressure is a technique in which a congested node stops receiving packet from upstream node. This may cause the upstream node or nodes to become congested and rejects receiving data from above nodes. Backpressure is a node-to-node congestion control technique that propagates in the opposite direction of data flow. The backpressure technique can be applied only to virtual circuit where each node has information of its above upstream node.



Closed Loop Congestion Control

2. Choke Packet Technique:

A choke packet is a packet sent by a node to the source to inform it of congestion. Each router monitors its resources and the utilization at each of its output lines. Whenever the **resource utilization exceeds the threshold value which is set by the administrator**, the router directly sends a choke packet to the source giving it a feedback to reduce the traffic. The intermediate nodes through which the packets has traveled are not warned about congestion



Closed Loop Congestion Control

3. Implicit Signaling:

In implicit signaling, there is no communication between the congested nodes and the source. **The source guesses that there is congestion in a network.** For example when sender sends several packets and there is no acknowledgment for a while, one assumption is that there is congestion.

Closed Loop Congestion Control

4. Explicit Signaling:

In explicit signaling, if a node experiences congestion it can explicitly sends a packet to the source or destination to inform about congestion. The difference between choke packet and explicit signaling is that, **the signal is included in the packets that carry data rather than creating different packet as in case of choke packet technique.**

Explicit signaling can occur in either forward or backward direction.

- **Forward Signaling:** In forward signaling signal is sent in the direction of the congestion. The destination is warned about congestion. The receiver in this case adopts policies to prevent further congestion.
- **Backward Signaling:** In backward signaling signal is sent in the opposite direction of the congestion. The source is warned about congestion and it needs to slow down.

The other congestion control mechanisms are:

- **Warning Bit**
- A special bit in the packet header is set by the router to warn the source when congestion is detected. The bit is copied and piggy-backed on the ACK and sent to the sender.
- The sender monitors the number of ACK packets it receives with the warning bit set and adjusts its transmission rate accordingly.

- **Traffic shaping** (Refer Module 1 Notes)
- Two traffic shaping algorithms are: LEAKY BUCKET AND TOKEN BUCKET

Efficiency

- Efficiency in computer networks refers to maximizing resource utilization while minimizing delays, congestion, and overhead. It ensures that:
- Bandwidth is fully utilized without excessive packet loss.
- Latency is minimized for a better user experience.
- Network congestion is managed effectively.
- Resources such as CPU, memory, and energy are used optimally.

Techniques that improve efficiency:

- **Traffic Engineering**: Optimizing data flow to prevent congestion.
- **Load Balancing**: Distributing network traffic across multiple paths or servers.
- **Quality of Service (QoS)**: Prioritizing critical applications (e.g., VoIP, gaming) over less time-sensitive ones.
- **Compression & Caching**: Reducing data size or storing frequently accessed content to lower bandwidth usage.

Fairness

- Fairness ensures equitable resource allocation among users or applications, preventing network monopolization by a few heavy users. Fairness can be defined in different ways:
- **Equal bandwidth allocation**: Every user gets the same bandwidth.
- **Proportional fairness**: Users receive resources based on their demands.
- **Weighted fairness**: Some users or applications (e.g., emergency services) are prioritized over others.

Mechanisms promoting fairness:

- **Fair Queueing Algorithms (e.g., Weighted Fair Queueing - WFQ)**: Ensures fair distribution of bandwidth among flows.
- **Congestion Control (e.g., TCP fairness)**: Prevents aggressive users from starving others.
- **Network Slicing**: Allocates dedicated resources for specific applications.

Balancing Efficiency and Fairness

- In real-world networks, there is often a trade-off between efficiency and fairness. Some approaches:
  - **Proportional fairness**: A balance where total efficiency is maximized while ensuring each user gets a fair share.
  - **Network Utility Maximization (NUM)**: Optimizes efficiency while maintaining fairness constraints.
  - **Active Queue Management (AQM)**: Techniques like Random Early Detection (RED) control congestion fairly without sacrificing efficiency.

THE END