NOTES ON

COMPUTER NETWORKS

Organized & Prepared By

Sharafat Ibn Mollah Mosharraf

12th Batch (05-06)

Dept. of Computer Science & Engineering

University of Dhaka

TABLE OF CONTENTS

NETWORK MODELS: OSI & TCP/IP – BASIC IDEA

What is this *Data Communication* or *Networking* course all about?

We want to transfer data between two computers. How can we do it? That's the question we'll try to find an answer for in the *Data Communication* or *Networking* courses – *how to communicate among computers*.

Designing the solution

So, how would we start? Let's try to visualize the situation. Suppose, you're using Yahoo Messenger. You want to say "hello" to your friend who also is using Yahoo Messenger. Then, how would you send this data (i.e., the text "hello")? Also, how would your friend receive that data?

The Application Layer

You're using Yahoo Messenger, which is an application, or, in other words, a running process. Similarly, your friend is also running the Messenger process in his computer. If we think in the simplest way, we can find that it's sufficient for the process from *your end* to just *send* the data, and similarly, it's sufficient for the process from *his end* to just *receive* the data – nobody needs to know how the data is traveling in-between.

So, the application just needs to prepare the *data* to be sent, or, prepare itself for receiving data. This is the first step of data transfer, which is known as the task of application layer.

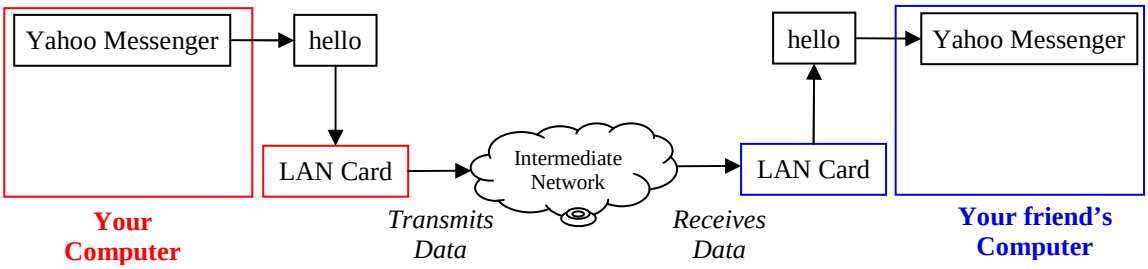


Figure 1: Application Layer Communication.

The Transport Layer

In the application layer, we didn't pay attention to one aspect. Problem arises when we consider the fact that *several processes run simultaneously* in a computer, but there is only a single device (e.g., LAN card) which receives or transmits data. How would your friend's LAN card know which process that data is to be forwarded to? This situation is depicted in *figure* 2.1 below.

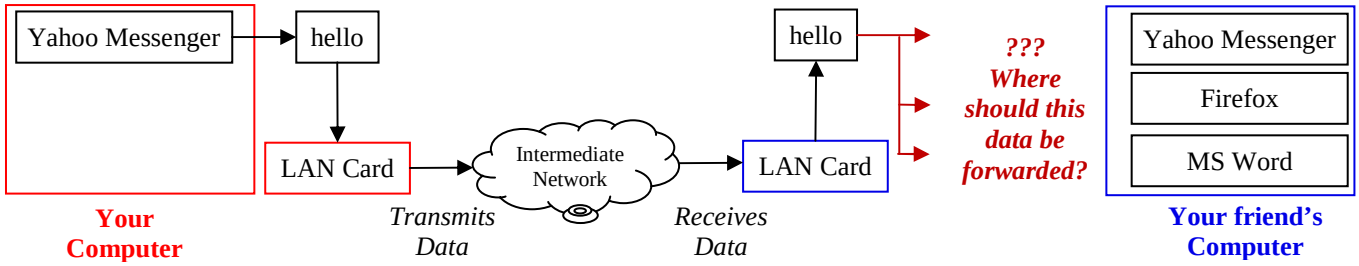


Figure 2.1: Problem with data transfer when there are multiple processes.

The solution? Well, we can send an extra info with the data which may contain something that would identify the destination process. Then the LAN card would know to which process that data is to be forwarded.

Now, what's that "*something*" that would identify the process? The name of the process? Well, there might be running several instances of the same program – all of whom have the same name. In that case, it'd sill be impossible to uniquely identify a process which would receive the data.

Well, how about process IDs? They're unique, of course. But the problem is, they're not *fixed* for each process. Suppose, right at this moment, your Yahoo Messenger has the PID 3483. When you'd restart the program, the PID may become 3521. So, how would the sending process in *your* computer know beforehand the PID of the receiving process in your friend's computer?

We need to come up with something else. And that *something else* is what we call "*ports*". You know what the term *port* means – a place (seaport or airport) where people and merchandise can enter or leave a country. Similarly, each program is assigned a *fixed* port number through which a data can enter to of leave from that program. No matter what the PID of a process is, it would bind itself to a *fixed* port number which must be known to the data sending process. Also, when a process binds itself to a particular port, no other process can bind itself to that port.

Now we get the idea. We shall send the destination port number along with the data, so that it can be received by the right process. We call this extra information a "*header*". This is the second layer of the data transfer process, which is called the *transport layer*.

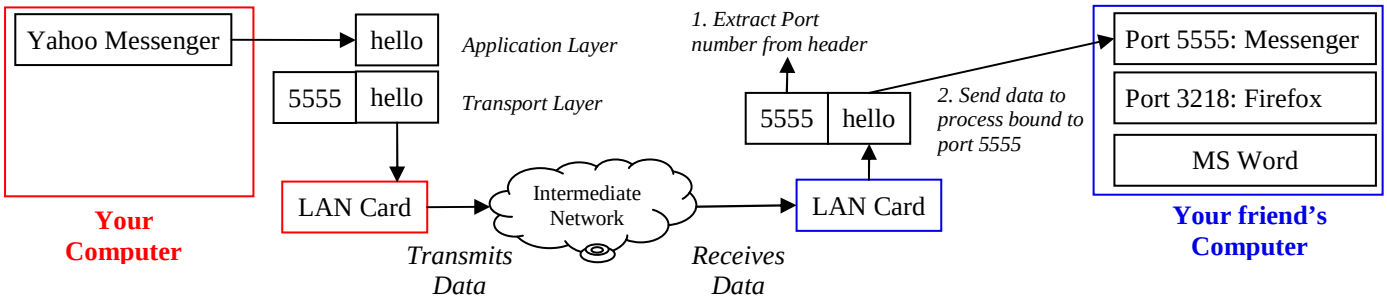


Figure 2.2: Data communication up to transport layer.

The Network Layer

Unfortunately, we forgot another fact. It's obvious that *more than one computers* may be running Yahoo Messenger. Then how it can be decided that which computer should receive the data? This situation is depicted in *figure* 3.1 below.

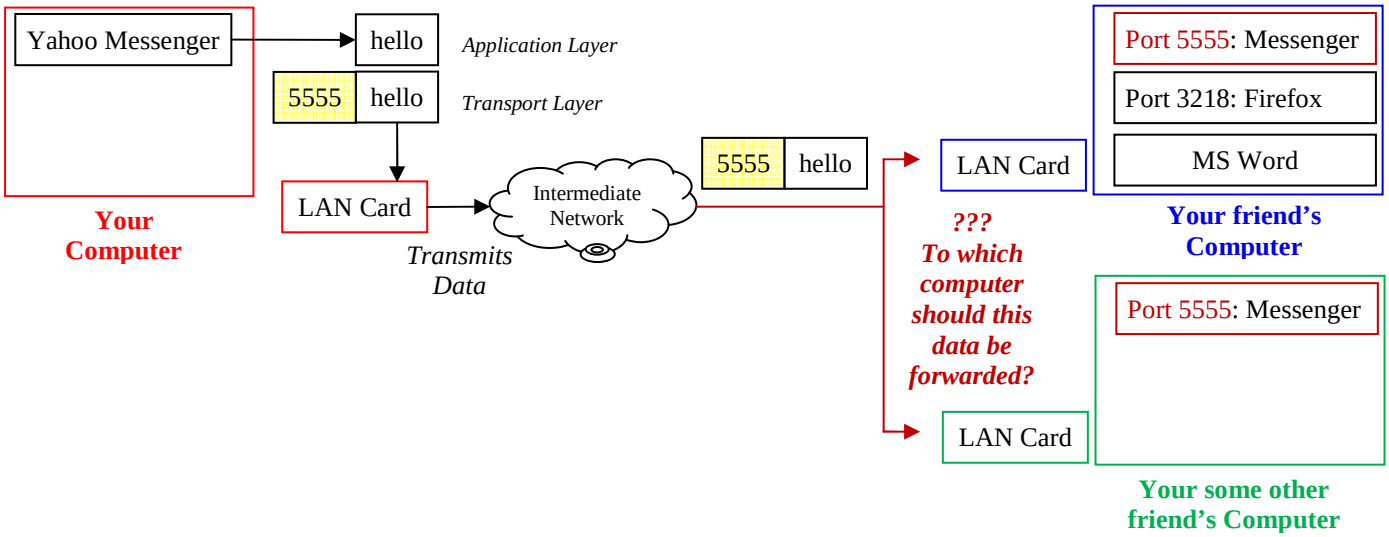


Figure 3.1: Problem with data transfer when there are multiple computers connected together in a network.

The solution is obvious. We'll assign each computer a *fixed* and *unique* ID, which is called a *Network Address* or *IP Address*, and add another header to the data packet which will contain the destination IP address. The layer in which this task is performed is called the *network layer*. Note that later, we'll study in details *how* the data will automatically reach the intended computer *from* the network.

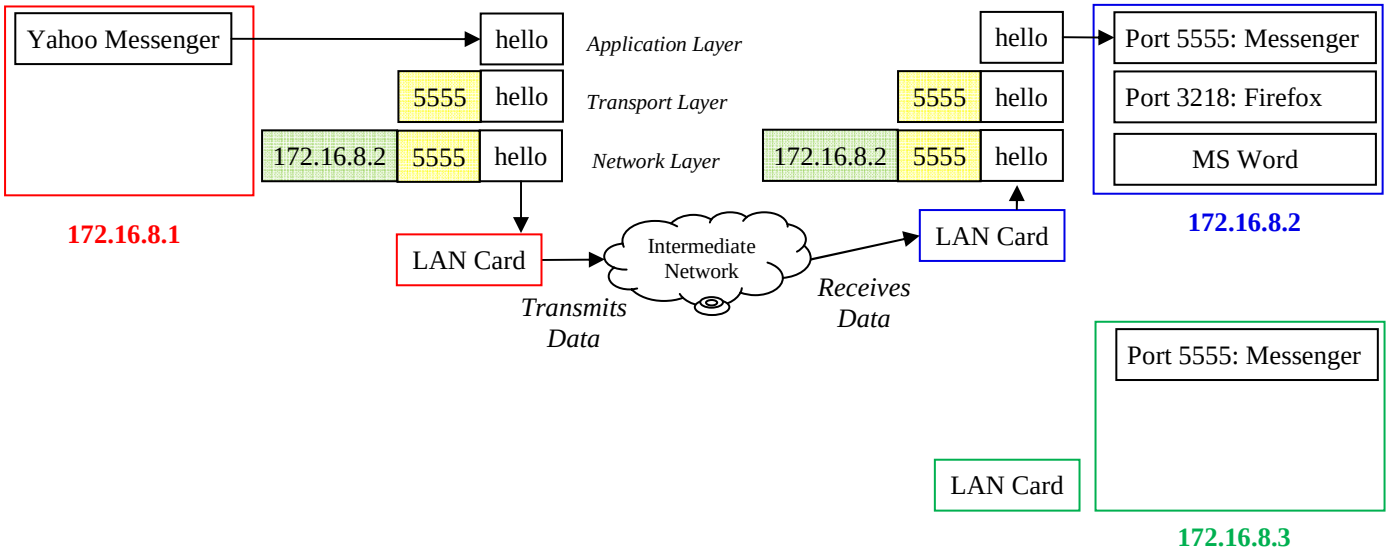


Figure 3.2: Data communication up to network layer.

The Data Link Layer

So far, we were concerned with just *sending* and *receiving* data only. Now it's time to examine the *intermediate network*.

Suppose you're in Bangladesh and your friend is in USA. Then how would your data travel such a long distance? There is no point mentioning the obvious fact that you cannot connect your computer to your friend's computer with cable *directly*. There must be some intermediate devices (called *hops* or *nodes*) which would carry on (or keep forwarding) your data until it reaches its destination. So, merely mentioning the IP address of the destination computer wouldn't help. You must send your data to your near-most hop, and it would relay it to his near-most hop (which is *towards* the destination computer, not on the *opposite* direction of it). Therefore, you'd need to know the address of your near-most hop. However, this address is not the IP address; this is called *Hardware Address* or *MAC* (*Machine Address Code*) *Address*. (Why a different identification system is used here will be explained shortly.)

This hardware address is added to the data as another header in the layer called the *data link layer*. However, a trailer is also added in this layer, which contains CRC (*Cyclic Redundancy Check*) code for verifying the integrity of the packet.

The hop receiving the data packet would change the header by putting the hardware address of *its* near-most hop's hardware address and pass the packet on. This process continues until the packet reaches the destination computer (called the *host*). The process is depicted in *figure* 4.

The Physical Layer

We've logically designed everything appropriately. But there remains the last question – how would the data *actually* move from hop-to-hop? Would it be transmitted *single-bit* at a time or *multiple-bits* at a time? Would it travel through *a piece of wire* or through *wind* (or, in other words, *wireless*)? Which transmission medium or transmission scheme would be *more efficient* or *more reliable*? These questions are answered in the *physical layer*. The physical layer is responsible for movements of individual bits from one hop (node) to the next.

Well, this concludes the network model we were trying to discover. This model is called the *TCP/IP Model*. Although we've observed that the TCP/IP model has five layers, actually, it has *four* layers. The *data link layer* and the *physical layer* are counted as a *single* layer.

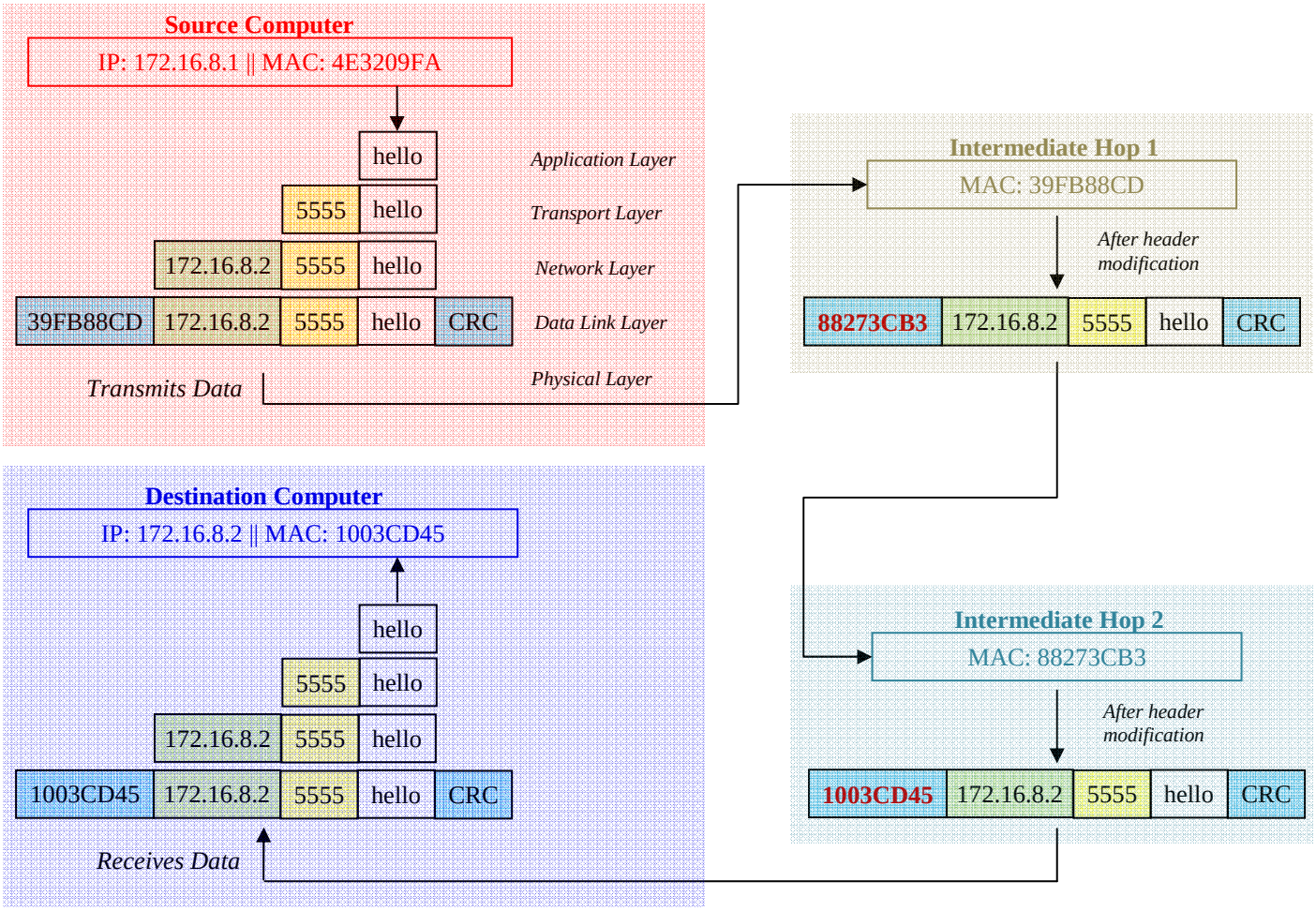


Figure 4: Complete Network Model for transferring data from one computer to another.

Next comes the *OSI* (*Open Systems Interconnection*) *Model*, which is the same as the TCP/IP model except that it splits the *application layer* into two *more* layers – one is called the *presentation layer*, and the other is called the *session layer*.

[*Omitted due to lack of time…*]

TASKS OF THE LAYERS AT A GLANCE

Tasks of Application layer:

How the receiver should interpret the incoming data.

Generally, we use application layer header fields to indicate the nature of the data contained in the packet. For example, in a field named 'error', we can set 1 to indicate that this is a packet containing error information.

Tasks of Presentation layer:

Concerned with the representation of data.

1. **Translation:** Converting system dependent data format into system independent data format and vice-versa.
2. **Encryption**
3. **Compression**

Tasks of Session layer:

1. **Dialog Control:** Half-duplex, full-duplex
2. **Synchronization:** Inserting a checkpoint after a certain amount of data so that when some data get lost or corrupted, only that amount of data may be resent instead of resending the whole message.

Tasks of Transport layer:

Responsible for *process-to-process* delivery of message.

1. **Port / Service-point Addressing**
2. **Providing Reliability**
    a. **Packet loss**
        i. General reliable data transfer protocol (*sending and receiving acknowledgement*)
        ii. Pipelined reliable data transfer protocol
            1. GBN (Go-Back-N)
            2. SR (Selective Repeat)
    b. **Packet corruption:** Checksum
3. **Flow Control:** Use a window.
4. **Congestion Control**

From reliability point of view, transport layer has the following tasks:

5. **Segmentation and Reassembly:** So that when a segment gets lost or corrupted, only that segment may be resent instead of resending the whole message.
6. **Connection Control:** TCP (connection-oriented) [*provides reliability of packet loss*], UDP (connectionless) [*if no packet-loss reliability is needed*]. [*However, both must provide reliability of packet corruption.*]

Tasks of Network layer:

Responsible for *host-to-host* delivery of message.

1. **Logical Addressing**
2. **Routing:** Deciding through which *path* a datagram should go from source to destination.

Tasks of Link Layer:

Responsible for *hop-to-hop* / *node-to-node* delivery of message.

1. **Physical / MAC Addressing**
2. **Reliability**
    a. **Packet loss :** GBN, SR (*mainly used for wireless communication*)
    b. **Packet corruption:** CRC
3. **Flow Control** (*adapter buffer may overflow*): Ethernet sends a PAUSE frame
4. **Multiple access control** (*multiple nodes send frames simultaneously through a single channel. How the receiver would distinguish among the frames?*)

From reliability point of view, link layer has the following task:

5. **Framing:** So that when a frame gets lost or corrupted, only that segment may be resent instead of resending the whole message.

Tasks of Physical Layer:

1. **Representation / Encoding of Bits:** How 0's and 1's are changed into signals.
2. **Data / Transmission Rate.** For example, Ethernet has a rate of 10/100/1000 Mbps.
3. **Physical Topology:** Star, Bus, Ring, Mesh, Hybrid topologies.

CHAPTER 3
TRANSPORT LAYER

Concepts

3.1 TCP and UDP Segment Structures

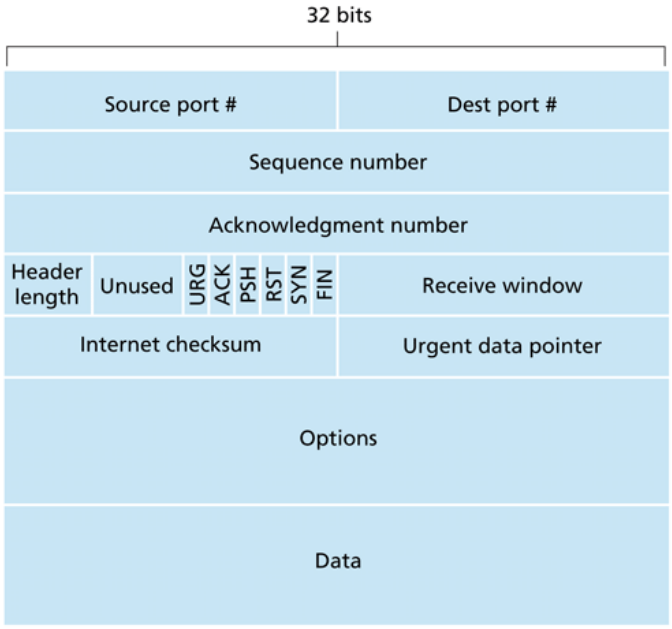


Figure 3.29 ♦ TCP segment structure

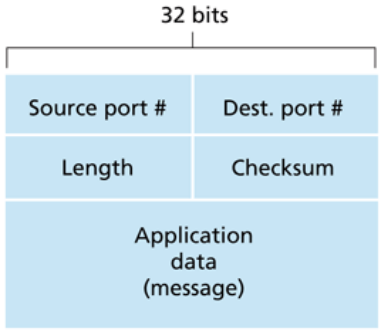


Figure 3.7 ♦ UDP segment structure

3.2 UDP Checksum (*a.k.a. Internet Checksum*)

1. Add all the 16-bit words in the UDP segment.
2. If the addition has overflow, then wrap it around.
3. The checksum is the 1's complement of the final sum.

3.3 Building a reliable data transfer protocol

1. **Reliable data transfer over a perfectly reliable channel (rdt 1.0)**
   - Just send and receive packets
2. **Reliable data transfer over a channel with bit errors (rdt 2.0)**
   - Use retransmission (ARQ / Stop-and-wait protocol).
   - **How?** – By feedbacking ACK and NACK.
   - **Problem:** The ACK/NACK itself could be corrupted.

     Solutions:

     1. Add enough checksum bits to allow the not only to detect, but also to recover from bit errors.
     2. Resend the *current* data packet.

        Problem with the 2nd solution: Receiver will receive duplicate packets.

        Solution to this new problem: Put a sequence number (0 or 1) in each packet. **(rdt 2.1)**
   - Instead of sending a NACK, we can resend the ACK of the last received non-corrupt packet. **(rdt 2.2)**
3. **Reliable data transfer over a *Lossy* channel with bit errors (rdt 3.0)**
   - Retransmit after timeout.
   - **Timeout interval?** – Should be at least as long as a round-trip delay between sender and receiver.

- **Problem:** Performance is poor.

  Solution: Use pipelining for sending packets.

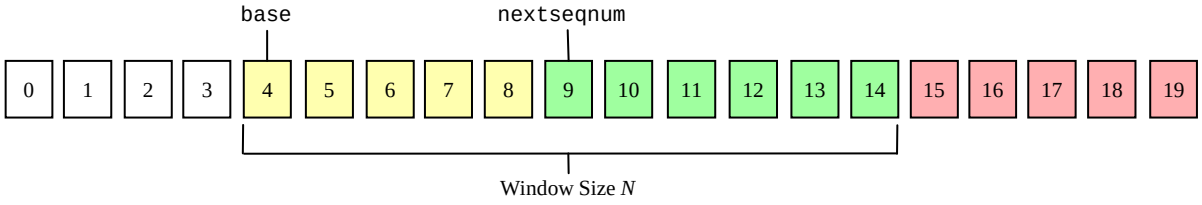
  Consequences of this solution:

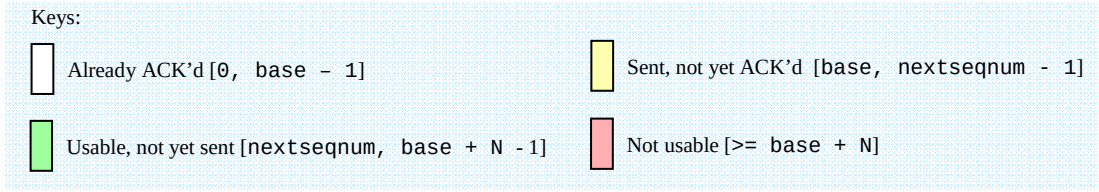
  1. Range of sequence numbers must be increased.
  2. Sender and receiver may have to buffer packets (which are sent/received, but not yet acknowledged).
  3. The above two depends on the manner in which a data transfer protocol responds to lost, corrupted and overly delayed packets.

3.4 Approaches toward pipelined error recovery:

1. Go-Back-N (GBN) [*Sliding Window Protocol*]

- Sender can transmit multiple packets without waiting for an acknowledgement.
- But sender is constrained to have no more than some maximum allowable number, *N*, of unacknowledged packets in the pipeline.
- An acknowledgement for packet with sequence number *n* will be taken to be a ***cumulative acknowledgement***, indicating that all packets with a sequence number up to and including *n* have been correctly received at the receiver.
- Range of sequence numbers: $[0, 2^k - 1]$, where k = Number of bits in the packet sequence number field.
- Sender's view of sequence numbers in GBN:





- ***Problem with GBN:***

  Poor performance when window size and bandwidth-delay product are both large. In that case, many packets can be in the pipeline. A single packet error can thus cause GBN to retransmit a large number of packets, many unnecessarily.

2. Selective Repeat (SR)

- Avoids unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error at the receiver.
- Acknowledge each correctly received packet whether or not it is in order.
- Out-of-order packets are buffered until any missing packets are received, at which point a batch of packets can be delivered *in order* to the upper layer.
- The receiver must re-acknowledge already received packets with certain sequence numbers *below* the current window base.
- ***Problem with SR:***

  Lack of synchronization between sender and receiver windows has important consequences: how can the receiver tell whether a received packet is of the current sequence number, or of an already ACK'ed sequence number?

  Solution:

  Set the window size to less than or equal to *half* the size of the sequence number space for SR protocols.

3.5 Maximum Segment Size (MSS) of a TCP Packet

- The maximum amount of data that can be grabbed and placed in a segment is limited by the MSS (maximum segment size).
- The MSS is typically set by first determining the length of the largest link-layer frame that can be sent by the local sending host (the so-called MTU – maximum transmission unit), and then setting the MSS to ensure that a TCP segment (when encapsulated in an IP datagram) will fit into a single link-layer frame. Common values for the MTU are 1,460 bytes, 536 bytes, and 512 bytes.
- The MSS is the maximum amount of application-layer data in the segment, not the maximum size of the TCP segment including headers.

3.6 Sequence Numbers and Acknowledgement Numbers

- Sequence numbers are over the stream of transmitted *bytes* and *not* over the series of transmitted *segments*. The sequence number for a segment is therefore the bytes-stream number of the *first byte* in the segment.
- Both sides of a TCP connection randomly choose an initial sequence number. This is done to minimize the possibility that a segment that is still present in the network from an earlier, already-terminated connection between two hosts is mistaken for a valid segment in a later connection between the same two hosts (which also happen to be using the same port numbers as the old connection).
- The acknowledgement number that the receiver puts in its segment is the sequence number of the *next* byte the receiver is expecting from the sender.
- TCP provides *cumulative acknowledgement* as it only acknowledges bytes up to the first missing byte in the stream.
- ***What does a host do when it receives out-of-order packets?***

  TCP RFCs do not impose any rules here. Possible solutions:

  1. The receiver immediately discards those segments.
  2. The receiver keeps those segments into the buffer.

  The latter choice is more efficient in terms of network bandwidth, and is the approach taken in practice.

3.7 Round-Trip Time Estimation and Timeout

```
EstimatedRTT = (1 – α) × EstimatedRTT + α × SampleRTT
DevRTT = (1 – β) × DevRTT + β × | SampleRTT – EstimatedRTT |
TimeoutInterval = EstimatedRTT + 4 × DevRTT
```

3.8 TCP's Reliable Data Transfer Mechanism

Consider what happens when the sender sends a sequence of segments 1, 2, … N, and all of the segments arrive in order without error at the receiver. Further suppose that the acknowledgement for packet $n < N$ gets lost, but the remaining $N – 1$ acknowledgements arrive at the sender before their respective timeouts. TCP would retransmit at most, one segment, namely, segment n. Moreover, TCP would not even retransmit segment n if the acknowledgement for segment $n + 1$ arrived before the timeout for segment n.

TCP's error recovery mechanism is probably best categorized as a hybrid of GBN and SR protocols.

TCP ACK generation:

Event	TCP Receiver Action
Arrival of in-order segment with expected sequence number. All data up to expected sequence number already acknowledged.	Delayed ACK. Wait up to 500 msec for arrival of another in-order segment. If next in-order segment does not arrive in this interval, send an ACK.
Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK transmission.	Immediately send single cumulative ACK, ACKing both in-order segments.
Arrival of out-of-order segment with higher-than-expected sequence number. Gap detected.	Immediately send duplicate ACK, indicating sequence number of next expected byte (which is the lower end of the gap).
Arrival of segment that partially or completely fills in gap in received data.	Immediately send ACK, provided that segment starts at the lower end of gap.

3.9 Doubling the Timeout Interval

Whenever the timeout event occurs, TCP retransmits the not yet acknowledged segment with the smallest sequence number. But each time TCP retransmits, it sets the next timeout interval to twice the previous value, rather than deriving it from the last `EstimatedRTT` and `DevRTT`.

However, whenever the timer is started after either of the two other events (i.e., data received from application above, and ACK received), the `TimeoutInterval` is derived from the most recent values of `EstimatedRTT` and `DevRTT`.

This modification provides a limited form of congestion control.

3.10 Fast Retransmit

If the TCP sender receives three duplicate ACKs for the same data, it takes this as an indication that the segment following the segment that has been ACKed three times has been lost. In that case, the TCP sender performs a fast retransmit – retransmitting the missing segment *before* that segment's timer expires.

3.11 Flow Control

Because TCP is not permitted to overflow the allocated buffer, we must have

```
LastByteRcvd – LastByteRead ≤ RcvBuffer
```

The receive window, denoted RcvWindow, is set to the amount of spare room in the buffer:

```
RcvWindow = RcvBuffer – [LastByteRcvd – LastByteRead]
```

The TCP specification requires the sender to continue to send segments with one data byte when the receiver's receive window is zero.

3.12 TCP Connection Management

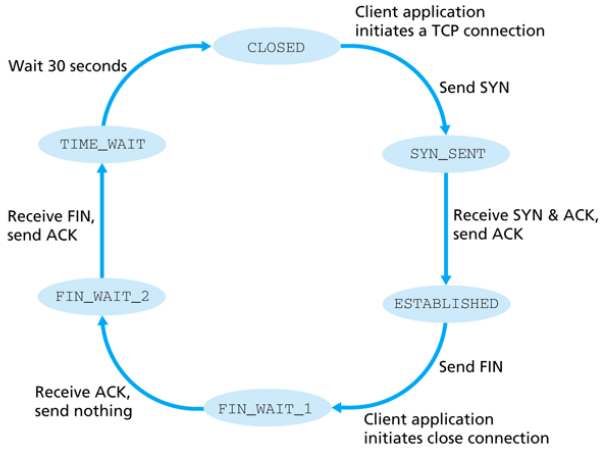


Figure 3.41 ♦ A typical sequence of TCP states visited by a client TCP

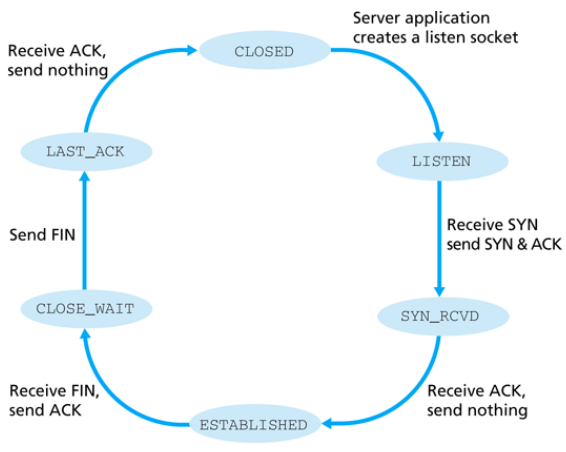


Figure 3.42 ♦ A typical sequence of TCP states visited by a server-side TCP

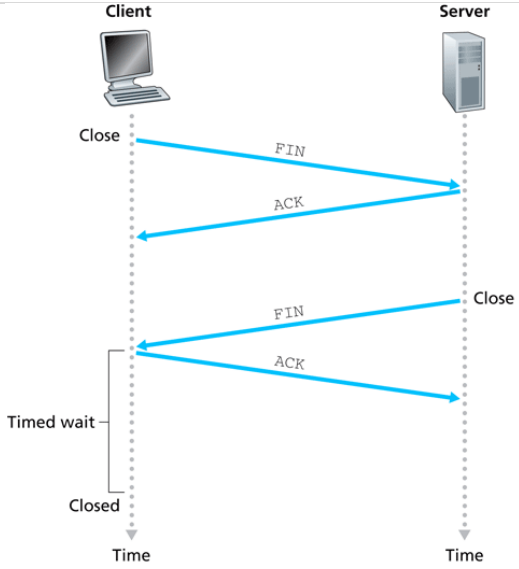


Figure 3.40 ♦ Closing a TCP connection

3.13 Principles of Congestion Control

The Causes and the Costs of Congestion

1. Scenario 1: *Two* senders, a Router with *Infinite* Buffers

Two hosts (A and B) each have a connection that shares a single hop between source and destination.

- ***Assume:***
  - λ_{in} bytes/sec = The average rate Host A is sending *original* data into the connection. Original means each unit of data is sent into the socket only *once*.
  - Data is encapsulated and sent – no error recovery (e.g., retransmission), flow control or congestion control is performed.
  - Host B operates in a similar manner and it too is sending at a rate of λ_{in} bytes/sec.
  - Outgoing link capacity of router = *R*. The link is shared.
  - The router has an infinite amount of buffer space.

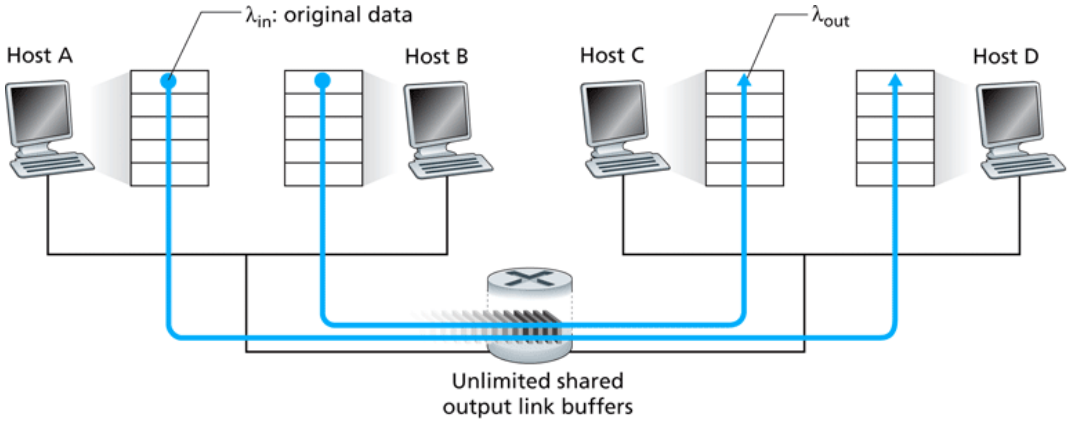


Figure 3.43 ♦ Congestion scenario 1: Two connections sharing a single hop with infinite buffers

- ***Performance of Host A's connection under this first scenario:***
  - ***Per-Connection* Throughput** (*number of bytes/sec at the receiver*):
    - Sending rate between 0 and *R*/2: Throughput = Sender's sending rate
    - Sending rate above *R*/2: Throughput = *R*/2

      Reason: The link simply can't deliver packets to receiver at a steady-state rate that exceeds *R*/2.

- Achieving a per-connection throughput of $R/2$ might actually appear to be a good thing, because the link is fully utilized in delivering packets to their destinations. However, operating near link capacity consequences in larger delay.

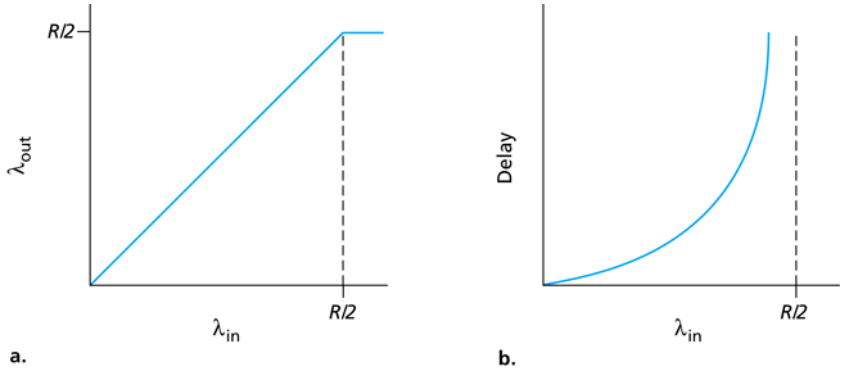


Figure 3.44 ♦ Congestion scenario 1: Throughput and delay as a function of host sending rate

- **Average Delay:**
  - As the sending rate approaches $R/2$, the average delay becomes larger and larger.
  - When the sending rate exceeds $R/2$, the average number of queued packets in the router is unbounded, and the average delay between source and destination becomes infinite.
- **Cause of congestion as learnt from this scenario:**

  Operating at an aggregate throughput of near or above R.
- **Cost of congestion as learnt from this scenario:**

  Large queuing delays are experienced.

2. Scenario 2: *Two* senders, a Router with *Finite* Buffers

- ***Assume:***
  - The router has an infinite amount of buffer space.
    Consequence: Packets will be dropped when arriving to an already-full buffer.
  - Each connection is reliable – retransmission on packet loss is performed.
  - λ_{in} bytes/sec = The average rate Host A is sending *original* data into the connection.
  - λ'_{in} bytes/sec = The *offered load* to the network (*original* + *retransmitted* data)
- ***Performance***

  Depends on how retransmission is performed.

1. Sender is able to somehow determine whether or not a buffer is free in the router and thus sends a packet only when a buffer is free.

- No loss would occur: $\lambda_{in} = \lambda'_{in}$.
- **Throughput = λ_{in}**
- From a throughput standpoint, performance is ideal – everything that is sent is received.
- Note that the average host sending rate cannot exceed $R/2$ under this situation, since packet loss is assumed never to occur.

2. Sender retransmits only when a packet is *known for certain* to be lost.

Let, $\lambda'_{in} = R/2$.

Let, at this value of offered load, **throughput = $R/3$**.

$\therefore$ The rate of actual data = throughput = $R/3$

$\therefore$ The rate of retransmitted data = $R/2 - R/3 = R/6$

- **Cost of congestion as learnt from this scenario:**

  The sender must perform retransmissions in order to compensate for dropped (lost)

packets due to buffer overflow.

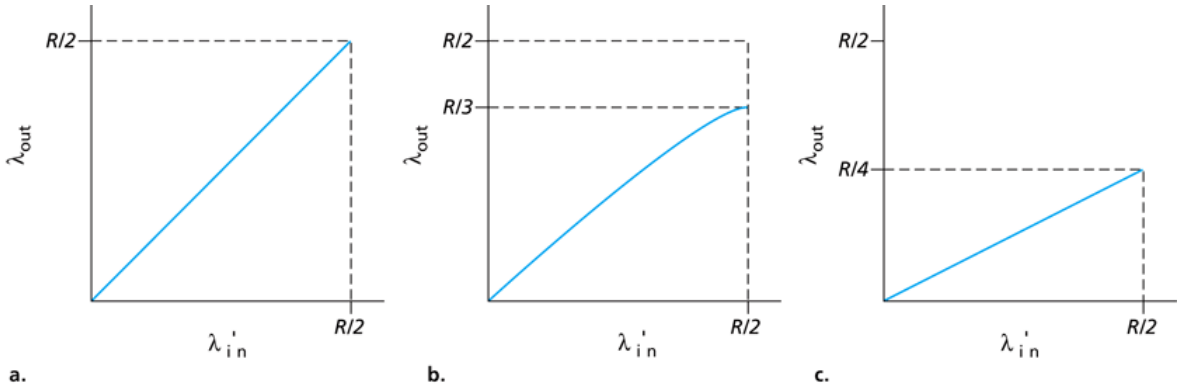


Figure 3.46 ♦ Scenario 2 performance with finite buffers

3. **The sender may time out prematurely and retransmit a packet that has been delayed in the queue but not yet lost.**

In this case, the work done by the router in forwarding the retransmitted copy of the original packet was wasted, as the receiver will have already received the original copy of this packet. The router would have better used the link transmission capacity to send a different packet instead.

- **Throughput: *R*/4**

  Reason: Since each packet is forwarded twice (on average), the throughput will have an asymptotic value of *R*/4 as the offered load approach *R*/2.

- **Cost of congestion as learnt from this scenario:**

  Unneeded retransmissions by the sender in the face of large delays may cause a router to use its link bandwidth to forward unneeded copies of a packet.

3. Scenario 3: Four Senders, Routers with Finite Buffers, and Multi-hop Paths

Four hosts transmit packets, each over overlapping two-hop paths.

- ***Assume:***
  - Each host uses a time-out/retransmission mechanism to implement reliable data transfer service.
  - All hosts have the same value of λ_{in}.
  - All router links have capacity R bytes/sec.
  - The A-C connection shares router R1 with D-B connection, and R2 with B-D connection.

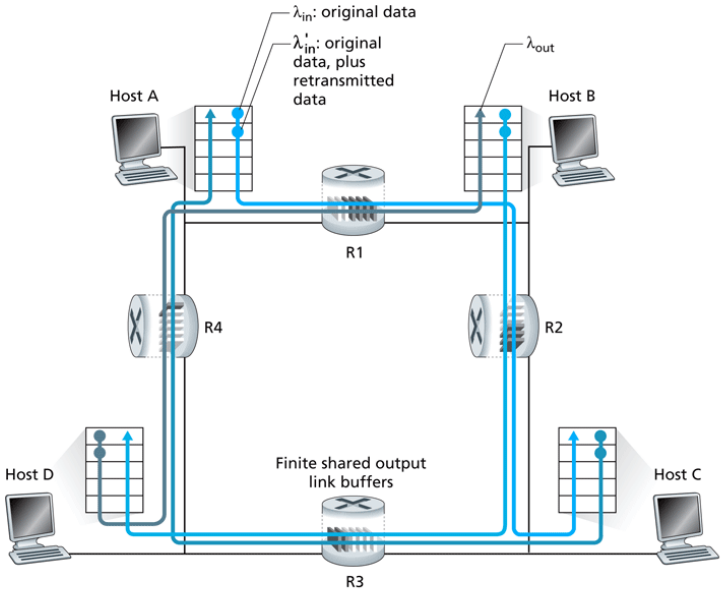


Figure 3.47 ♦ Four senders, routers with finite buffers, and multihop paths

- ***For extremely small values of λ_{in},***
  - Buffer overflows are rare.
  - The throughput approximately equals the offered load.
- ***For slightly larger values of λ_{in},***
  - The corresponding throughput is also larger, since more original data is being transmitted into the network and delivered to the destination, and overflows are still rare.
- ***Thus, for small values of λ_{in}, an increase in λ_{in} results in an increase in λ_{out}.***
- ***For extremely large values of λ_{in} (and hence λ'_{in}),***
  - Consider the router R2.
  - The A-C traffic arriving from router R1 to R2 can have an arrival rate at R2 that is at most *R*, the capacity of the link from R1 to R2, regardless of the value of λ_{in}.
  - The arrival rate of B-D traffic at R2 can be much larger than that of the A-C traffic.
  - Because the A-C and B-D traffic must compete at router R2 for the limited amount of buffer space, the amount of A-C traffic that successfully gets through R2 (i.e., is not lost due to buffer overflow) becomes smaller and smaller as the offered load from B-D gets larger and larger.
  - In the limit, as the offered load approaches infinity, an empty buffer at R2 is immediately filled by a B-D packet, and the throughput of the A-C connection at R2 goes to zero in the limit of heavy traffic.
- **Cost of congestion as learnt from this scenario:**

  When a packet is dropped along a path, the transmission capacity that was used at each of the upstream links to forward that packet to the point at which it is dropped ends up having been wasted.

3.14 Approaches to Congestion Control

Congestion control approaches can be distinguished based on the whether or not the network layer provides any explicit assistance to the transport layer for congestion control purposes:

1. ***End-to-End Congestion Control***
   - The network layer provides no explicit support to the transport layer for congestion control purposes.
   - Even the presence of congestion in the network must be inferred by the end systems based only on observed network behavior (e.g., packet loss and delay).
   - Implemented by TCP.
2. ***Network-Assisted Congestion Control***
   - Network-layer components (i.e., routers) provide explicit feedback to the sender regarding the congestion state in the network.
   - Implemented by ATM [*the ATM ABR (Available Bit-Rate) congestion control mechanism*].

3.15 TCP Congestion Control

The approach taken by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion. If little congestion is perceived, the send rate is increased; if much congestion is perceived, the send rate is decreased.

But this approach raises three questions:

- **How does a TCP sender limit the rate at which it sends traffic into its connection?**

  Introduce a *congestion window*, which imposes a constraint on the rate of transmission. Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of *congestion window* and *receive window*. That is:

  $$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \min(\texttt{CongWin}, \texttt{RcvWindow})$$

Ignoring the RcvWindow constraint, roughly, at the beginning of every RTT, the constraint CongWin permits the sender to send CongWin bytes of data into the connection; and at the end of the RTT, the sender receives the acknowledgements for the data.

Thus, The sender's send rate is roughly CongWin/RTT bytes/sec. By adjusting the value of CongWin, the sender can therefore adjust the rate at which it sends data into its connection.

- **How does a TCP sender perceive that there is congestion on the path between itself and the destination?**
  - ***How congestion is detected***

    When there is excessive congestion, then one (or more) router buffers along the path overflows, causing a datagram to be dropped. The dropped datagram, in turn, results in a loss event at the sender – either a timeout or the receipt of three duplicate ACKs – which is taken by the sender to be an indication of congestion on the sender-to-receiver path.
  - ***How congestion-freeness is detected***

    In this case, acknowledgements for previously unacknowledged segments will be received at the TCP sender. TCP will take the arrival of these acknowledgements to increase its congestion window size (and hence its transmission rate).
- **What algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?**

  It's called TCP congestion control algorithm.

3.16 TCP Congestion Control Algorithm

The algorithm has three major components:

1. Additive-Increase, Multiplicative Decrease (AIMD)
2. Slow Start (SS)
3. Reaction to Timeout Events

Additive-Increase, Multiplicative Decrease (AIMD)

- **Approach:** Increase transmission rate (window size), probing for usable bandwidth, until loss occurs.
- **Additive Increase:** Increase `CongWin` by 1 MSS every RTT until loss detected.
  - This can be accomplished in several ways; a common approach is for the TCP sender to increase its `CongWin` by MSS (MSS/`CongWin`) bytes whenever a new acknowledgement arrives.

    For example, if

    MSS = 1,460 bytes

    `CongWin` = 14,600 bytes

    ∴ No. of segments sent within an RTT = 10

    Each arriving ACK increases the congestion window size by 1/10 MSS.

    ∴ After acknowledgements for all 10 segments have been received, the value of `CongWin` will have increased by MSS, as desired.
  - This linear increase phase is known as *congestion avoidance* (CA).
- **Multiplicative Decrease:** Cut `CongWin` in half after loss. However, `CongWin` is not allowed to drop below 1 MSS.

Slow Start (SS)

- When a TCP connection begins, the value of `CongWin` is typically initialized to 1 MSS.
- Instead of increasing its rate linearly during this initial phase, a TCP sender continues to increase its rate exponentially by doubling its value of `CongWin` every RTT until there is a loss event, at which time `CongWin` is cut in half and then grows linearly, as described above.

Reaction to Loss Events

- **Timeout Events**
  - TCP sender enters a slow-start phase – i.e., sets the congestion window to 1 MSS and then grows the window exponentially. The window continues to grow exponentially until `CongWin` reaches one half of the value it had before the timeout event. At that point, `CongWin` grows linearly.
  - TCP manages these more complex dynamics by maintaining a variable called `Threshold`, which determines the window size at which slow start will end and congestion avoidance will begin. The variable is initially set to a large value (65 Kbytes in practice) so that it has no initial effect. Whenever a loss event occurs, the value of `Threshold` is set to one half of the current value of `CongWin`.
  - Thus, while in SS, a TCP sender increases the value of `CongWin` exponentially fast until `CongWin` reaches Threshold. When `CongWin` reaches Threshold, TCP enters the CA phase, during which `CongWin` ramps up linearly as described earlier.
  - Both TCP Tahoe and TCP Reno acts in this way.
- **Triple Duplicate ACKs**
  - **TCP Tahoe –** behaves as described in the timeout event – enters an SS phase.
  - **TCP Reno –** the congestion window is cut in half and then increases linearly.
    - ***Reason*:** Even though a packet has been lost, the arrival of three duplicate ACKs indicates that some segments have been received at the sender. Thus, unlike the case of a timeout, the network is showing itself to be capable of delivering at least some segments, even if other segments are being lost to congestion.
    - This canceling of the SS phase after a triple duplicate ACK is called *fast recovery*.

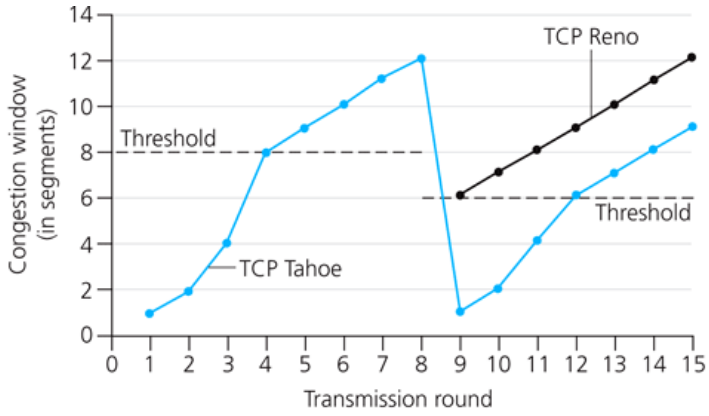


Figure 3.53 ♦ Evolution of TCP's congestion window (Tahoe and Reno)

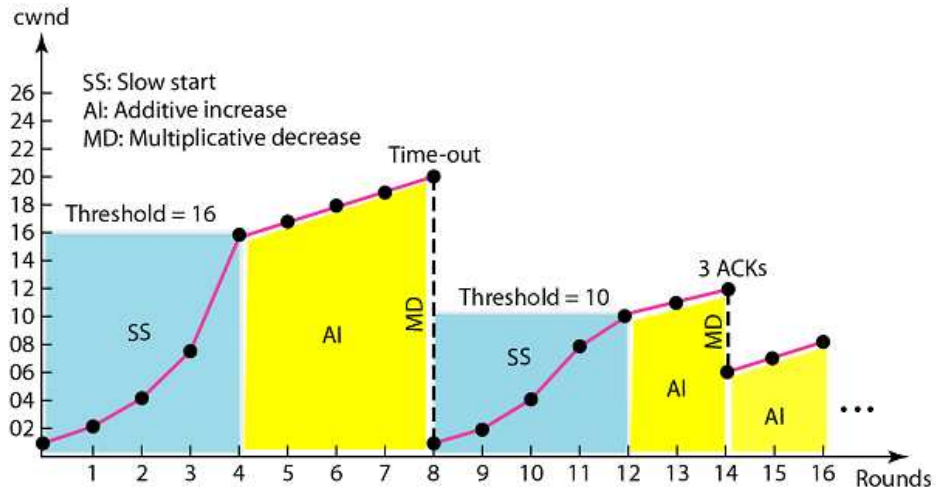


Figure: TCP Reno's Reaction to loss events.

3.17 TCP Throughput

- Ignore the SS phases, as they are typically very short (since the sender grows out of the phase exponentially fast).
- Let,

  Current window size = *w* bytes

  Current round-trip time = *RTT* sec

  ∴ Transmission rate = *w*/*RTT* (roughly)

  Let, *W* be the value of *w* when a loss event occurs.

  Assuming that RTT and W are approximately constant over the duration of the connection, the TCP transmission rate ranges from $\frac{1}{2} \times \frac{W}{RTT}$ to $\frac{W}{RTT}$.

  Because TCP's throughput increases linearly between the two extreme values, we have

  Average throughput of connection = $\left(\frac{1}{2} \times \frac{W}{RTT} + \frac{W}{RTT}\right) / 2 = \frac{3}{4} \times \frac{W}{RTT} = \frac{0.75\,W}{RTT}$

CHAPTER 4
NETWORK LAYER

4.1 Routing and Forwarding

Routing refers to the network-wide process that determines the end-to-end paths that packets take from source to destination.

Forwarding refers to the router-local action of transferring a packet from an input link interface to the appropriate output link interface.

How forwarding works

Every router has a forwarding table. A router forwards a packet by examining the value of a field in the arriving packet's header, and then using this value to index into the router's forwarding table. The result from the forwarding table indicates to which of the router's link interfaces the packet is to be forwarded.

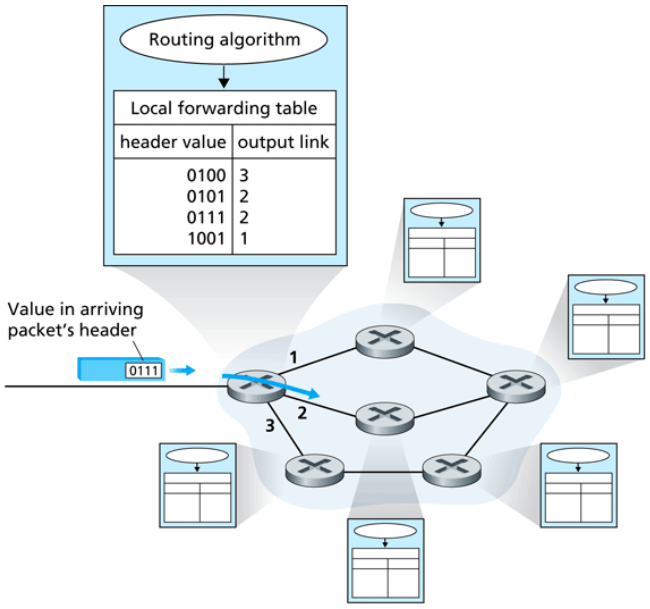


Figure 4.2 ♦ Routing algorithms determine values in forwarding tables.

How forwarding tables are configured?

Using routing algorithms. *Coming soon…*

4.2 Network Service Model

Network Architecture	Service Model	Bandwidth Guarantee	No-Loss Guarantee	Ordering	Timing	Congestion Control
Internet	Best Effort	None	None	Any Order	Not Maintained	None
ATM (*Asynchronous Transfer Mode*)	ABR (*Available Bit Rate*)	Guaranteed Minimum	None	In Order	Not Maintained	Congestion Indication Provided
	CBR (*Constant Bit Rate*)	Guaranteed Constant Rate	Yes	In Order	Maintained	Congestion Will Not Occur

4.3 Virtual Circuit (VC) Networks

A VC consists of

1. A path (i.e., a series of links and routers) between the source and destination hosts
2. VC numbers, one number for each link along the path
3. Entries in the forwarding table in each router along the path.

A packet belonging to a VC will carry a VC nunber in its header. Because a VC may have a different VC number on each link, each intervening router must replace the VC number of each traversing paket with a new one. The new VC number is obtained from the forwarding table.

In a VC network, the network's routers must maintain connection state information for the ongoing connections. Specifically, each time a new connection is established across a router, a new connection entry must be added to the router's forwarding table; and each time a connection is released, an entry must be removed from the table.

Example of forwarding in a VC network

Incoming Interface	Incoming VC#	Outgoing Interface	Outgoing VC#
1	12	2	22
2	63	1	18
3	7	2	17
1	97	3	87
…	…	…	…

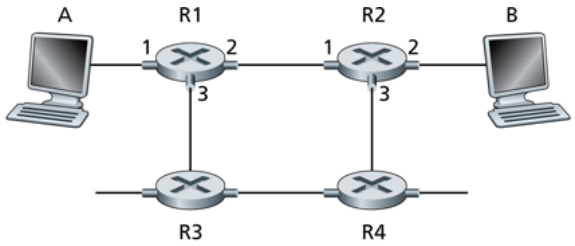


Figure 4.3 ♦ A simple virtual circuit network

Why doesn't a packet just keep the same VC number on each of the links along its route?

1. Replacing the number from link to link reduces the length of the VC number field in packet header.
2. VC setup is simplified. Specifically, with multiple VC numbers, each link in the path can choose a VC number independently of the VC number chosen at other links along the path. If a common VC number were required for all links aong the path, the routers would have to exchange and process a substantial number of messages to agree on a common VC number to be used for connection.

Phases in VC

1. **VC Setup**
   - ➢ Sending transport layer contacts the network layer, specifies the receiver's address, and waits for the network to set up the VC.
   - ➢ Network layer
     - determines the path between sender and receiver, and the VC number for each link along the path.
     - adds an entry in the forwarding table in each router along the path.
     - may also reserve resources (e.g. bandwidth) along the path of the VC.
2. **Data Transfer**
3. **VC Teardown**
   - ➢ Sender (or receiver) informs the network layer of its desire to terminate the VC.
   - ➢ Network layer
     - informs the end system on the other side of the network of the call termination.
     - updates the forwarding tables in each of the routers on the path to indicate that the VC no longer exists.

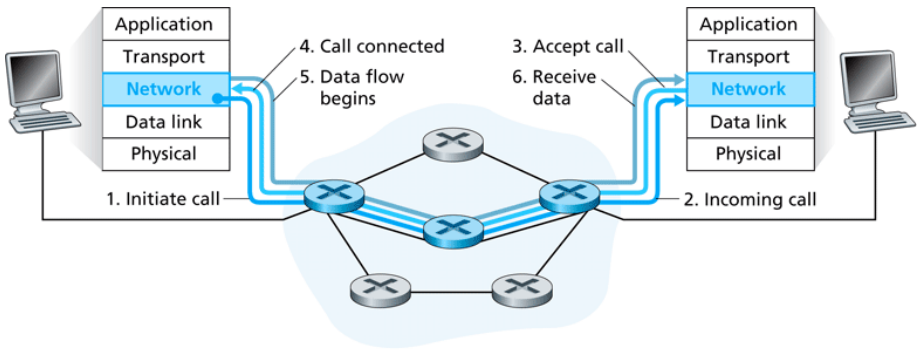


Figure 4.4 ♦ Virtual-circuit setup

Signaling Messages and Protocols

The messages that the end systems send into the network to initiate or terminate a VC, and the messages pased between the routers to set up the VC are known as ***signaling messages***, and the protocols used to exchange these messages are often referred to as ***signaling protocols***.

4.4 Datagram Networks

In a datagram network, each time an end system wants to send a packet, it stamps the packet with the address of the destination end system and then pops the packet into the network. This is done without any VC setup. Routers in a datagram network do not maintain any state information about VCs.

A packet passes through a series of routers. When a packet arrives at the router, the router uses the packet's destination address to lookup the appropriate output link interface in the forwarding table. It then forwards the packet to that output link interface.

For example, a router with 4 links may have the following forwarding table:

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0
11001000 00010111 00011000 00000000 through 11001000 00010111 00011000 11111111	1
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2
otherwise	3

Problem: There have to be 2^{32} (≈ 4 billion) entries in each router's forwarding table!

Solution: Use *Longest Prefix Matching*.

When there are multiple matches, the router uses the ***longest prefix matching rule***; that is, it finds the longest matching entry in the table and forwards the packet to the link interface associated with the longest prefix match. The table might look like the following:

Prefix Match	Link Interface
11001000 00010111 00010	0
11001000 00010111 00011000	1
11001000 00010111 00011	2
otherwise	3

As an example, the destination address 11001000 00010111 00011000 1010101 is mapped to the link interface 1.

Although routers in datagram networks maintain no connection state information, they nevertheless maintain forwarding state information in their forwarding tables. Because forwarding tables in datagram networks can be modified at any time, a series of packets sent from one end system to another may follow different paths through the network and may arrive out of order.

4.5 Inside the Router

High-Level View of a Generic Router Architecture

There are four components of a router:

1. **Input Ports**
   - Physical layer functions: terminating an incoming physical link to a router.
   - Data link layer functions: protocol, decapsulation.
   - Lookup and forwarding function:
     - Data packets are forwarded into the switching fabric of the router to place at the appropriate outupt port.
     - Control packets (e.g., packets carrying routing protocol information) are forwarded from input port to the routing processor.
2. **Switching Fabric**

   Connects the router's input ports to its output ports.
3. **Output Ports**
   - Stores the packets that have been forwarded to it through the switching fabric and then transmits the packets on the outgoing link.
   - The output port thus performs the reverse data link and physical layer functionalities of the input port.
4. **Routing Processor**
   - Executes routing protocols
   - Maintains the routing information and forwarding tables
   - Performs network management functions within the router

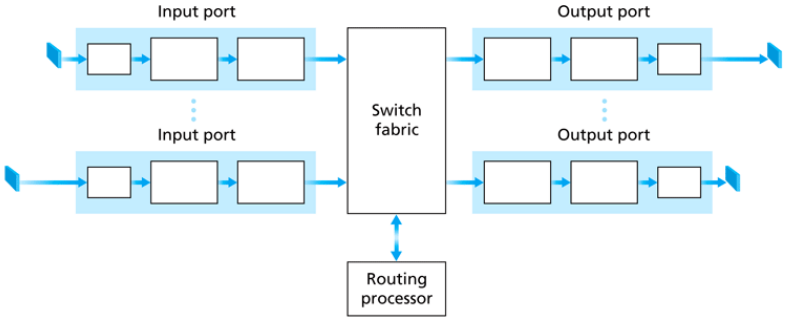


Figure 4.6 ♦ Router architecture

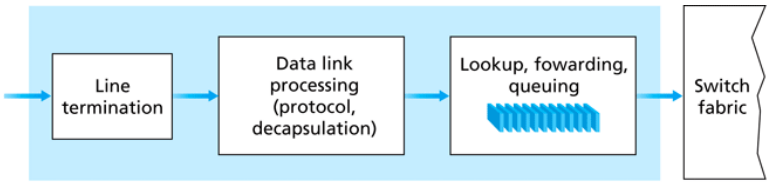


Figure 4.7 ♦ Input port processing

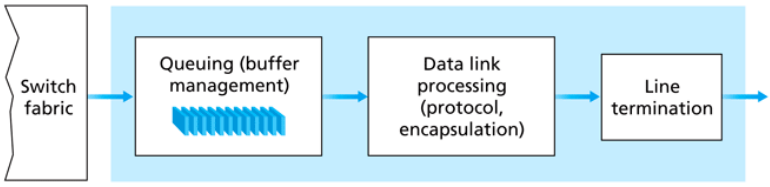


Figure 4.9 ♦ Output port processing

Switching Techniques

1. **Switching via memory**
   - An input port with an arriving packet signals the routing processor via an interrupt.
   - The packet is then copied from the port into memory.
   - The routing processor then extracts the destination address from the header, looks up the appropriate output port in the forwarding table, and copies the packet to the output port buffers.
2. **Switching via a bus**
   - The input ports transfer a packet directly to the output port over a shared bus, without the intervention by the routing processor.
   - As the bus is shared, only one packet at a time can be transferred over the bus. Thus, the bandwidth of the router is limited to the bus speed.
3. **Switching via an interconnection network / crossbar switch**
   - A crossbar switch is an interconnection netgwork consisting of 2*n* buses that connect *n* input ports to *n* output ports.
   - A packet arriving at an input port travels along the horizontal bus attached to the input port until it intersects with the vertical bus leading to the desired output port.
   - If the vertical bus is being used to transfer a packet from another input port to this same output port, the arriving packet is blocked and must be queued at the input port.

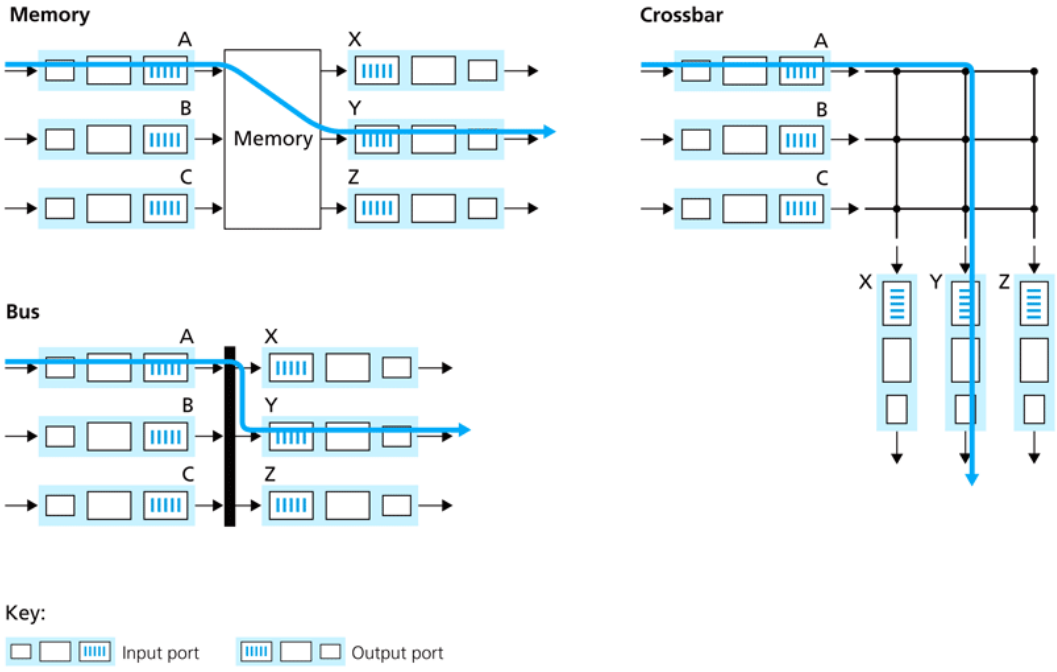


Figure 4.8 ♦ Three switching techniques

4.6 IP Datagram Fragmentation

The problem

Not all link layer protocols can carry network layer packets of the same size. For example, Ethernet frames can carry up to 1,500 bytes of data, whereas frames for some wide-area links can carry no more than 576 bytes. The maximum amount of data that a link layer frame can carry is called the ***maximum transmission unit*** (MTU). While forwarding packets, if a router receives a large frame and finds out that the output link the packet is destined to has a smaller MTU, then what will happen?

The solution – Datagram Fragmentation

- When a datagram is created, the sending host stamps the datagram with source and destination IP addresses as well as an identification number. Typically, the sending host increments the identification number for each datagram it sends.

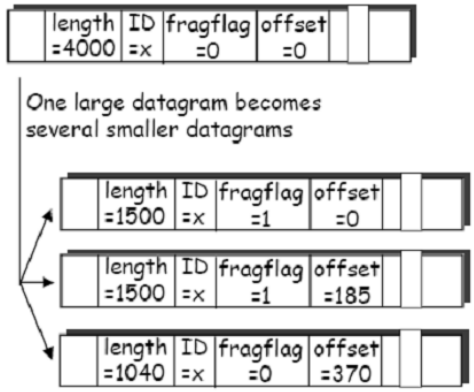
- When a router needs to fragment a datagram, each resulting datagram (i.e., fragment) is stamped with the source and destination IP and identication number of the original datagram.

  Also, in order to for the destination host to determine whether a fragment is missing (and also to be able to reassemble the fragments in their proper order), an offset field is used to specify where the fragment fits within the original IP datagram.

  Because IP is an unreliable service, one or more of the fragments may never arrive at the destination. For this reason, in order for the destination host to be absolutely sure it has received the last fragment of the original datagram, the last fragment has a flag bit set to 0, whereas all the other fragments have this flag bit set to 1.

- The job of datagram reassembly is implemented in the end systems rather than in network routers in order to keep the network core simple.

- The payload of the datagram is passed to the transport layer at the destination only after the IP layer has fully reconstructed the original IP datagram. If one or more of the fragments does not arrive at the destination, the incomplete datagram is discarded and not passed to the transport layer.



Consider an example illustrated in the figure on the right. A datagram of 4,000 bytes (20 bytes of IP header + 3,980 bytes of payload) arrives at a router and mulst be forwarded to a link with an MTU of 1,500 bytes. Suppose that the original datagram is stamped with an identification number of *x*. The characteristics of the three fragments are shown below:

Fragment	Bytes	ID	Offset	Flag
1st	Data = 1,480 bytes Header = 20 bytes	*x*	0 (means the data should be inserted beginnning at byte 0)	1
2nd	Data = 1,480 bytes Header = 20 bytes	*x*	185 (means the data should be inserted beginnning at byte 1480. Note that 185 × 8 = 1480)	1
3rd	Data = 1,020 bytes (= 3980 - 1480 - 1480) Header = 20 bytes	*x*	370 (means the data should be inserted beginnning at byte 2960. Note that 370 × 8 = 2960)	0

The values in the above table reflect the requirement that the amount of original payload data in all but the last fragment be a multiple of 8 bytes, and that the offset value be specified in units of 8-byte chunks.

4.7 IPv4 Datagram Format

Version: Protocol version: IPv4, IPv6.

Type of Service: Vendor Specific Service.

Datagram Length: (Header + Data) length.

Identifier, Flags, Offset: For fragmentation.

TTL: Hop count.

Upper-layer Protocol: TCP, UDP etc.

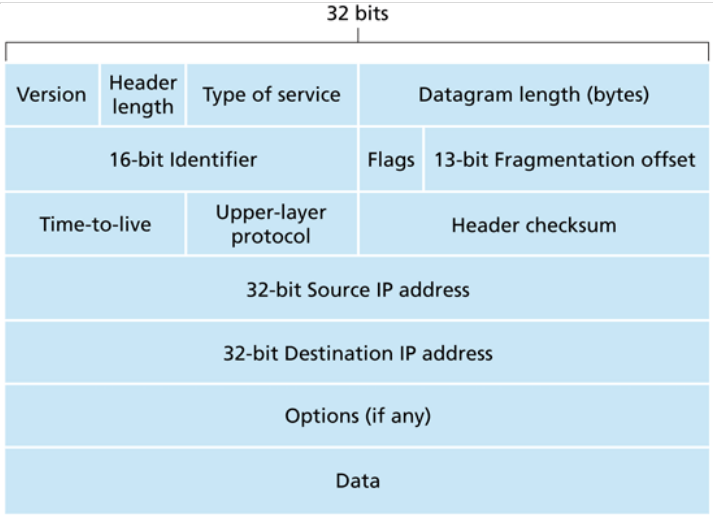


Figure 4.13 ♦ IPv4 datagram format

4.8 **Why error checking is performed at both transport and network layers?**

1. Only the IP header is checksummed at the IP layer, while the TCP/UDP checksum is computed over the entire TCP/UDP segment.
2. TCP/UDP and IP do not necessarily both have to belong to the same protocol stack. TCP can, in principle, run over a different protocol (e.g. ATM) and IP can carry data that will not be passed to TCP/UDP.

4.9 **IPv4 Addressing**

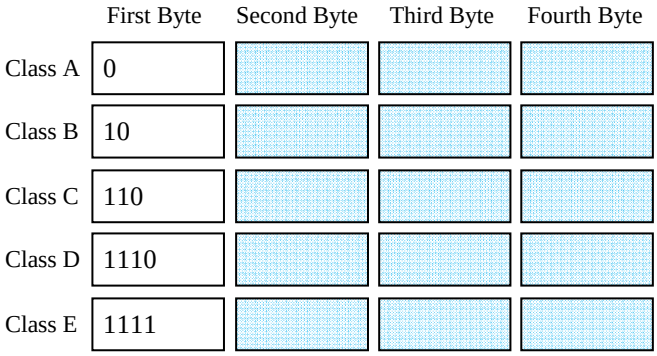
Notation

1. Binary: 10000000 00001011 00000011 00011111
2. Dotted-Decimal: 128 . 11 . 3 . 31
3. Hexadecimal: 0x 80 0B 03 1F

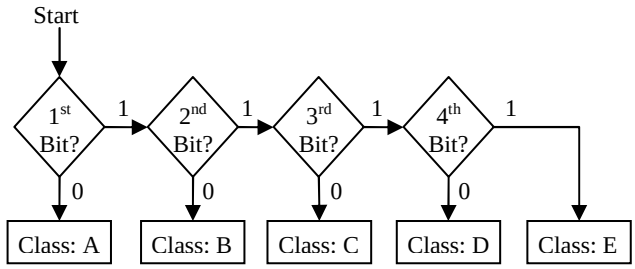
4.10 **Classful Addressing**

In classful addressing, the IP address space is divided into five classes: A, B, C, D and E. Each class occupies some part of the whole address space.

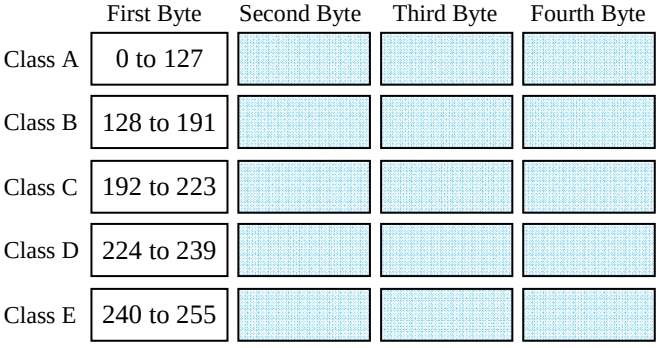
Recognizing Classes



Finding the class in binary notation



Finding the address class



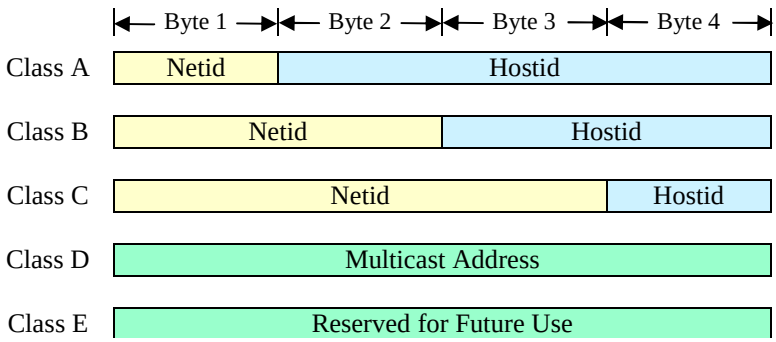
Finding the class in decimal notation

Class	Number of Addresses	Percentage
A	2^{31}	50%
B	2^{30}	25%
C	2^{29}	12.5%
D	2^{28}	6.25%
E	2^{28}	6.25%

Addresses per class

Netid, Hostid and Blocks

In classful addressing, an IP address in classes A, B and C is divided into ***netid*** and ***hostid***.



Each class is divided into a fixed number of ***blocks*** with each block having a different *netid*. For example, class A is divided into 128 blocks:

The 1st block covers addresses from **0**.0.0.0 to **0**.255.255.255 (netid 0).
The 2nd block covers addresses from **1**.0.0.0 to **1**.255.255.255 (netid 1).
The last block covers addresses from **127**.0.0.0 to **127**.255.255.255 (netid 127).

Similarly, class B is divided into 128 × 128 = 16,384 blocks:

The 1st block covers addresses from **128.0**.0.0 to **128.0**.255.255 (netid 128.0).
The last block covers addresses from **191.255**.0.0 to **191.255**.255.255 (netid 191.255).

Note that for each block of addresses, the *netid* is the same, but the *hostid* can take any value in the given range.

Special Addresses

Special Address	Netid	Hostid	Source or Destination	Example
Network address	Specific	All 0s	None	141.14.0.0
Direct broadcast address	Specific	All 1s	Destination	141.14.3.255
Limited broadcast address	All 1s	All 1s	Destination	255.255.255.255
This host on this network	All 0s	All 0s	Source	0.0.0.0
Specific host on this network	All 0s	Specific	Destination	0.0.3.64
Loopback address	127	Any	Destination	127.0.0.1

Network Address

The first address in a block (in classes A, B and C) is called the *network address* and is used to identify the organization to the rest of the internet. It defines the network of the organization, not individual hosts.

Properties of Network Addresses

1. The network address is the first address in the block.
2. The network address defines the network to the rest of the internet.
3. Given the network address, we can find the class of the address, the block, and the range of the addresses in the block.

Finding class, netid of the block and range of addresses from a network address

Consider the network address 132.21.0.0.

The class is B, because the first byte is between 128 and 191.
The block has a netid of 132.21.
The addresses range from132.21.0.0 to 132.21.255.255.

Masking – Finding the network address from a given address

A *mask* is a 32-bit number that gives the first address in the block (the network address) when bitwise ANDed with an address in the block. In the AND operation for classful addressing, there are three masks, one for each class. The following table shows the ***default mask*** for each class:

Class	Mask in Binary	Mask in Dotted-Decimal
A	11111111 00000000 00000000 00000000	255.0.0.0
B	11111111 11111111 00000000 00000000	255.255.0.0
C	11111111 11111111 11111111 00000000	255.255.255.0

Direct Broadcast Address

In classes A, B and C, if the hostid is all 1s, the address is called a *direct broadcast address*. It is used by a router to send a packet to all hosts in a specifc network.

Limited Broadcast Address

In classes A, B and C, an address with all 1s for the netid and hostid defines a broadcast address in

the current network. A host that wants to send a message to every other host can use this address as a destination address in an IP packet. However, a router will block a packet having this type of address to confine the broadcasting to the local network. Note that this address belongs to class E.

This host on this network

If an IP address is composed of all zeroes, it means *this host on this network*. This is used by a host at bootstrap time when it does not know its IP address. The host sends an IP packet to a bootstrap server using this address as the source address and a limited broadcast address as the destination address to find its own address.

Specific host on this network

An IP address with a netid of all zeroes means a specific host on this network. It is used by a host to send a message to another host on the same network.

Loopback Address

The IP address with the first byte equal to 127 is used for the loopback address, which is an address used to test the software on a machine. When this address is used, a packet never leaves the machine; it simply returns to the protocol software.

Private Addresses

A number of blocks in each class are assigned for private use. They are not recognized globally. These blocks are depicted in the following table:

Class	Netids	Blocks
A	10.0.0	1
B	172.16 to 172.31	16
C	192.168.0 to 192.168.255	256

Unicast, Multicast and Broadcast Addresses

Unicast communication is *one-to-one*. When a packet is sent from an individual source to an individual destination, a unicast communication takes place.

Multicast communication is *one-to-many*. When a packet is sent from an individual source to a group of destinations, a multicast communication takes place. The entire address defines a groupid. A system on the internet can have one or more class D multicast addresses (in addition to its unicast adderss or addresses). If a system has seven multicast addresses, it means that it belongs to seven different groups.

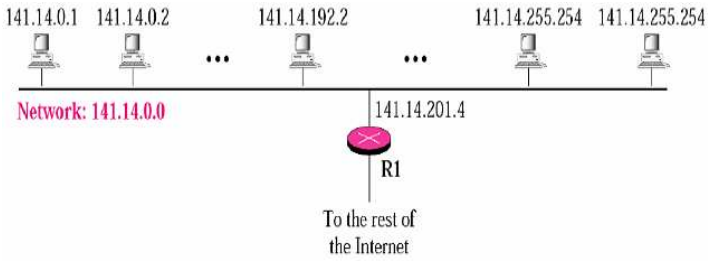
Broadcast communication is *one-to-all*. The internet allows broadcasting only at the local level.

The Problem with Classful Addressing

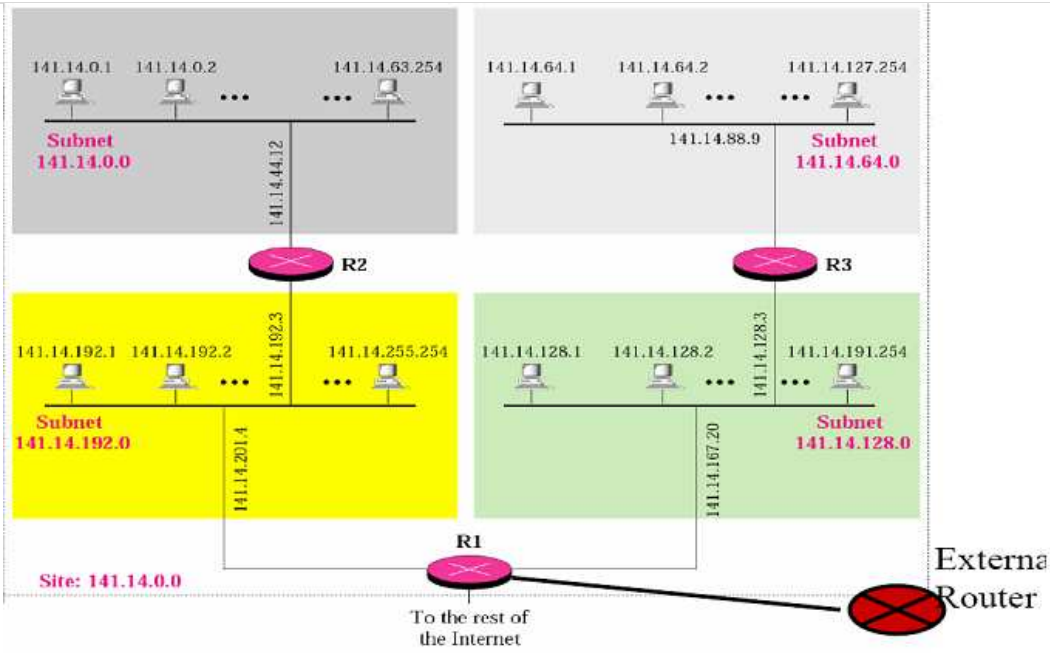
Classful addressing means network must be of size 65536 (2^{16}) for class B and 256 (2^{8}) for class C. If an organization wants 4 networks each with 300 hosts, then it needs 4 class B networks. Consequently, more than 200,000 addresses are wasted.

Subnetting

Subnetting was introduced in 1980s to solve this problem. In subnetting, a network is divided into *subnets* (short for *subnetworks*).



Example of a network which is not subnetted



Example of a subnetted network

In the example above, only R1, R2 and R3 need to know about the subnets. Each subnet requires an entry in their routing table. Routers external to the network need not to know about subnetting. In the figure, the external router requires only an entry for the network address (141.14.0.0) in its routing table.

Subnetid

141.14.64.1 is an address in the subnet 141.14.64.0 (this is called the subnet address).

Before subnetting

netid

After subnetting

netid subnetid hostid

Subnet Mask

When we configure a host, we must specify both the IP address of the host and the subnet mask. The subnet mask has the same role as the network mask; however, it has more 1s.

Example

Given the following netid, subnetid and hostid
(Note: 1st 2 bytes are in dotted decimal, last 2 bytes in binary)

netid subnetid hostid

The corresponding subnet mask is

In dotted decimal notation, it is 255.255.192.0

Finding the Subnet Address from a Given Address

Consider an IP address 200.45.34.56 and the subnet mask 255.255.240.0. To get the subnet address, we apply the AND operation on the IP address and the subnet mask.

IP Address → 11001000 00101101 00100010 00111000
Subnet Mask → 11111111 11111111 11110000 00000000
Subnet Address → 11001000 00101101 00100000 00000000

The subnet address is 200.45.32.0.

Number of subnetworks

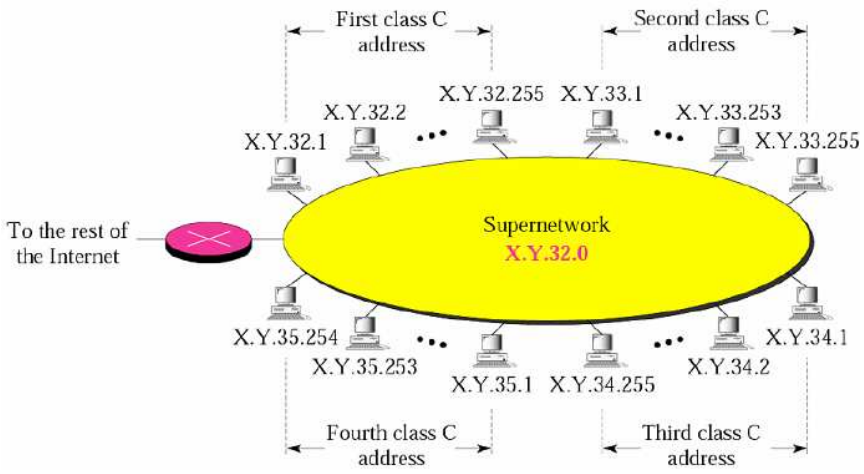
The number of subnetworks can be found by counting the extra 1s that are added to the default mask to make the subnet mask. For example, in the above example, the number of extra 1s is 3; therefore, the number of subnets is 2^4 or 16.

Number of addresses per subnet

The number of addresses per subnet can be found by counting the 0s in the subnet mask. For example, in the above example, the number of 0s is 12; therefore, the number of possible addresses in each subnet is 2^{12} or 4096.

Supernetting

If an organization wants 1000 addresses, supernetting allows 4 class C networks to be merged to form a supernet with 1024 addresses.



A supernetwork

Supernet address assignment rules

If a supernet consists of 100 class C networks and if the addresses are *randomly* chosen, then the routers external to the supernet will require 100 entries for the supernet. It would be desirable if only 1 entry is required. This can be achieved by carefully assigning addresses:

1. The number of blocks must be a power of 2 (1, 2, 4, 8, 16, …).
2. The blocks must be contiguous in the address space (i.e., no gaps between the blocks).
3. The third byte of the first address in the superblock must be evenly divisible by the number of blocks. In other words, if the number of blocks is *N*, the 3rd byte must be divisible by *N*.

Example

A company needs 600 addresses. Which of the following set of class C blocks can be used to form a supernet for this company?

a. 198.47.32.0 198.47.33.0 198.47.34.0
Not acceptable. Number of blocks is 3, which is not a power of 2.

b. 198.47.32.0 198.47.42.0 198.47.52.0 198.47.62.0
Not acceptable. Blocks are not contiguous.

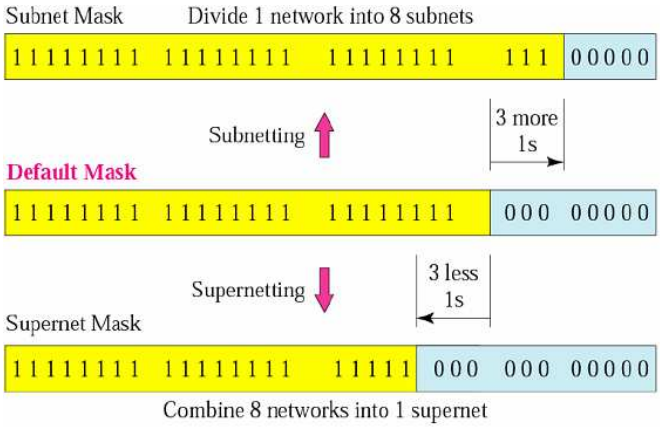
c. 198.47.31.0 198.47.32.0 198.47.33.0 198.47.34.0
Not acceptable. 3rd byte of first address is not divisible by 4 (number of blocks).

d. 198.47.32.0 198.47.33.0 198.47.34.0 198.47.35.0
Acceptable.

Supernet Mask

A supernet mask is the reverse of a subnet mask. A subnet mask for class C has more 1s than the default mask for this class. A supernet mask for class C has less 1s than the default mask for this class.

The following figure shows the difference between a subnet mask and a supernet mask. A subnet mask that divides a block into eight subblocks has 3 more 1s ($2^3 = 8$) than the default mask; a supernet mask that combines eight blocks into one superblock has 3 less 1s than the default mask.



Example 1

We need to make a supernetwork out of 16 class C blocks. What is the supernet mask?

Solution: For 16 blocks, we need to change four 1s to 0s in the default mask. So, the mask is: 11111111 11111111 1111**0000** 00000000 or 255.255.240.0.

Example 2

Given the first address of a supernet 205.16.32.0 and a supernet mask of 255.255.248.0, how many blocks are in this supernet and what is the range of addresses?

The supernet mask has 21 1s. The default mask has 24 1s. Since the difference is 3, there are 2^3 or 8 blocks in this supernet.

The blocks are 205.16.32.0, 205.16.33.0, …, 205.16.39.0.

The range of addresses is: from 205.16.32.0 to 205.16.39.255.

4.11 Classless Addressing

The problem with classful addressing

The use of classful addressing has created many problems. Until the mid 1990s, a range of addresses meant a block of addresses in class A, B or C. The minimum number of addresses granted to an organization was 256 (class C); the maximum was 16, 777, 216 (class A). In between these limits, an organization could have a class B block or several class C blocks. These address assignments were always in multiples of 256. However, what about a small business that needed only 16 addresses? Or a household that needed only two addresses?

To resolve the problems of classful addressing, in 1996, the Internet authorities announced a new architecture called *classless addressing*.

Restrictions

1. The number of *addresses* in a block must be a power of 2 (1, 2, 4, 8, 16, …).
2. The *addresses* in a block must be contiguous.
3. The first address must be evenly divisible by the number of addresses.
   If the address has ≤ 256 addresses, we need to divide the right-most byte only.
   If the address has ≥ 256 and ≤ 256^2 addresses, we need to check only the two right-most bytes and so on.

Example 1

Which of the following can be the beginning address of a block that contains 16 addresses?

a. 205.16.37.32 **b.** 190.16.42.44 **c.** 17.17.33.80 **d.** 123.45.24.52

Only two are eligible (*a* and *c*), because both 32 and 80 are divisible by 16.

Example 2

Which of the following can be the beginning address of a block that contains 1024 addresses?

a. 205.16.37.32 **b.** 190.16.42.0 **c.** 17.17.32.0 **d.** 123.45.24.52

In this case, we need to check two bytes because 1024 = 4 × 256. The right-most byte must be divisible by 256, and the second byte from the right must be divisible by 4. Only (*c*) is eligible.

CIDR (Classless Inter-Domain Routing) / Slash Notation

A classless network is usually written as *A.B.C.D*/*n* where *n* specifies the number of 1s in the mask. The first *n* bits of A.B.C.D is known as the ***prefix*** and *n* is known as the ***prefix length***. The last (32 – *n*) bits are known as the ***suffix***.

Finding the Network address, range of a block etc.

These all are similar to the techniques used in *subnetting*.

Subnetting

We can, of course, have subnetting with classless addressing. The prefix length (*n*) increases to define the subnet prefix length.

Finding the Subnet Mask

The number of desired subnets defines the subnet prefix. If the number of subnets is *s*, the number of extra 1s in the prefix length is $\log_2 s$, where $s = 2^{\text{number of extra 1s}}$.

Example

An organization is granted the block 130.34.12.64/26. The organization needs 4 subnets. What are the subnet addresses and the range of addresses for each subnet?

First, we find the number of addresses that can be assigned to each subnet. Since the prefix length is 26, the last 6 bits are available as hostid. Hence, the total number of addresses in the block is 64 (2^6). If we create 4 subnets, each subnet will have 16 addresses.

Now, let us find the subnet prefix (subnet mask). We need four subnets which means we need to add two more 1s to the site (netid) prefix. The subnet prefix is then /28.

Therefore, the range of addresses for each subnet is:

Subnet 1: 130.34.12.64/28 to 130.34.12.79/28.
Subnet 2: 130.34.12.80/28 to 130.34.12.95/28.
Subnet 3: 130.34.12.96/28 to 130.34.12.111/28.
Subnet 4: 130.34.12.112/28 to 130.34.12.127/28.

4.12 Network Address Translation (NAT)

Suppose that an ISP has allocated a contiguous block of addresses to an organization. What if the subnet of the organization grows bigger? Fortunately, there is a simpler approach to address allocation that has found increasingly widespread use in such scenarios: NAT.

A NAT-enabled router does not *look* like a router to the outside world. Instead, the NAT router behaves to the outside world as a *single* device with a *single* IP address.

How NAT Works

Consider the following example.

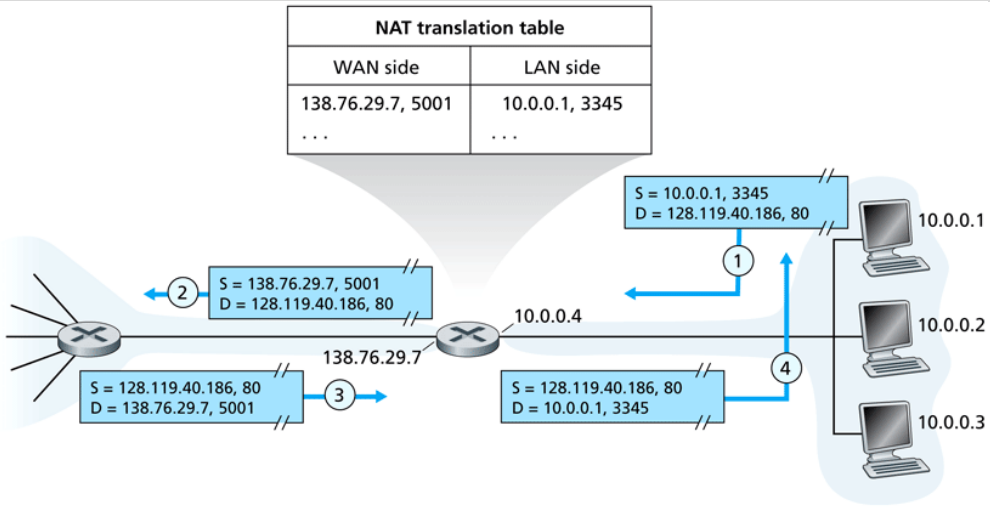


Figure 4.22 ♦ Network address translation

1. Suppose, host 10.0.0.1 requests a web page on some web server with IP 128.119.40.186. The host 10.0.0.1 assigns the (arbitrary) source port number 3345 and sends the datagram into the LAN.
2. The NAT router receives the datagram, generates a new source port number 5001 for the datagram, replaces source IP with its WAN-side IP 138.76.29.7, and replaces the original source port number 3345 with the new source port number 5001. An entry into the NAT translation table is also entered.
3. The web server, unaware that the arriving datagram containing the HTTP request has been manipulated by the NAT router, responds with a datagram whose destination IP is the IP of the NAT router and destination port number is 5001.
4. When this datagram arrives at the NAT router, the router indexes the NAT translation table using the destination IP and port number to obtain the appropriate IP address (10.0.0.1) and destination port number (3345) for the browser. The router then rewrites the datagram's destination address and destination port number, and forwards the datagram into the home network.

Objections to NAT

1. Port numbers are meant to be used for addressing *processes*, not for addressing *hosts*.
2. Routers are supposed to process packets up to layer 3.
3. NAT protocol violates the so-called end-to-end argument; i.e., host should be talking directly with each other, without interfering nodes modifying IP addresses and port numbers.
4. We should use IPv6 to solve the shortage of IP addresses, rather than recklessly patching up the problem with a stopgap solution like NAT.

Problem of NAT with P2P Applications

- In a P2P application, any participating peer A should be able to initiate a TCP connection to any other participating peer B.
- Now, if peer B is behind a NAT, it cannot act as a server and accept TCP connections (unless the NAT is specifically configured for the P2P application).
- This problem can be circumvented if peer A first contacts peer B through an intermediate peer C which is not behind a NAT and to which B has established an on-going TCP connection. Peer A can then ask peer B, via peer C, to initiate a TCP connection directly back to peer A.
- This hack, called ***connection reversal***, is actually used by many P2P applications. But if both peer A and peer B are behind their own NATs, then, for all practical purposes, it is impossible to establish a TCP connection between the two peers without application-specific NAT configuration.

4.13 Internet Control Message Protocol (ICMP)

ICMP is used by hosts and routers to communicate network-layer information to each other. The most typical use of ICMP is for error reporting.

ICMP messages have a type and a code field, and contain the header and the first 8 bytes of the IP datagram that caused the ICMP message to be generated in the first place (so that the sender can determine the datagram that caused the error).

4.14 IPv6

Important changes introduced in IPv6 datagram format:

1. **Expanded addressing capabilities.**
   - Size of IP address is increased from 32 to 128 bits.
   - In addition to unicast and multicast addresses, IPv6 has introduced a new type of address, called an ***anycast address***, which allows a datagram to be delivered to any one of a group of hosts that have the same prefix. (This feature could be used, for example, to send an HTTP GET to the nearest of a number of mirror sites that contain a given document.)
2. **A streamlined 40-byte fixed-length header.**
3. **Flow labeling and priority.**

   This allows labeling of packets belonging to particular flows for which the sender requests special handling, such as nondefault quality of service or real-time service.

IPv6 Datagram Format

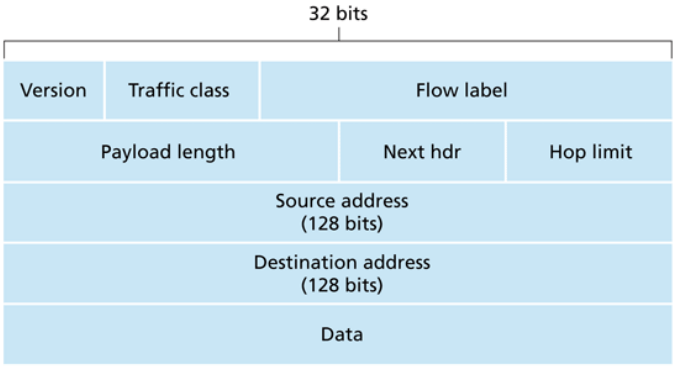


Figure 4.24 ♦ IPv6 datagram format

Traffic class: Similar to TOC field in IPv4.

Next header: Similar to protocol field in IPv4.

Hop limit: Similar to TTL field in IPv4.

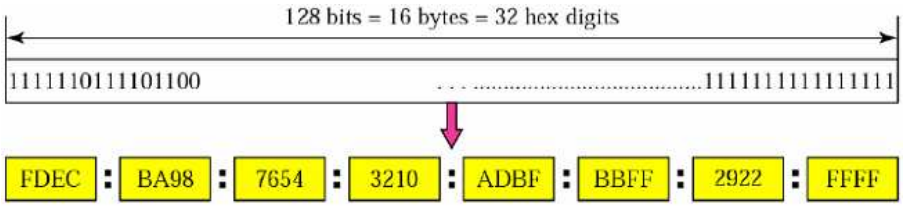
Fields no longer present in IPv6 datagram

- **Fragmentation / Reassembly**

If an IPv6 datagram received by a router is too large to be forwarded over the outgoing link, the router simply drops the datagram and sends a "Packet Too Big" ICMP error message back to the sender. The sender can then resend the data using a smaller IP datagram size.

- **Header Checksum**
- **Options**

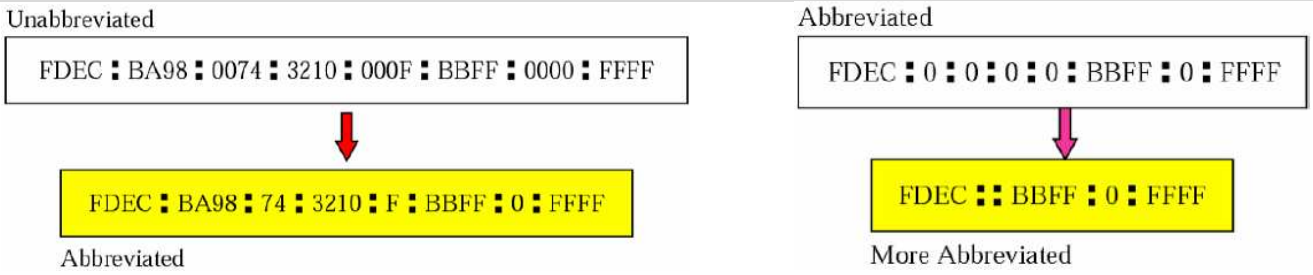
IPv6 Address Format: Hexadecimal Colon Notation



Abbreviation of Addresses

1. Leading zeroes of a section (four digits between two colons) can be omitted. Only the leading zeroes can be omitted, not the trailing zeroes.

   Using this form of abbreviation, 0074 can be written as 74, 000F as F, and 0000 as 0. Note that 3210 cannot be abbreviated.



2. If there are consecutive sections consisting of zeroes only, then the zeroes can be removed altogether and they are replaced with a double semicolon.

   Note that this type of abbreviation is allowed only once per address. If there are two runs of zero sections, only one of them can be abbreviated.

Transitioning from IPv4 to IPv6

How will the public internet, which is based on IPv4, be transitioned to IPv6? The problem is that while new IPv6-capable systems can be made backward-compatible, that is, can send, route and receive IPv4 datagrams, already deployed IPv4-capable systems are not capable of handling IPv6 datagrams.

Two techniques are as follows:

Dual-Stack

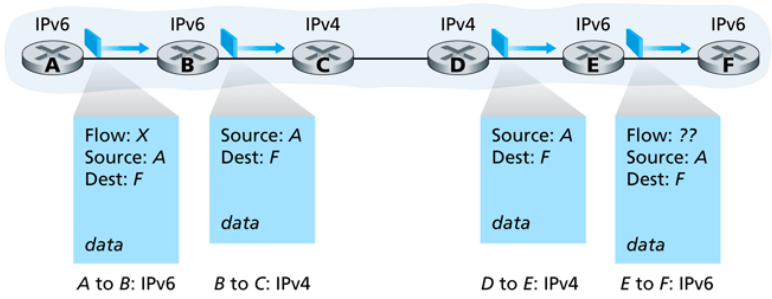


Figure 4.25 ♦ A dual-stack approach

- Suppose node A wants to send an IP datagram to node F, both of whom are IPv6-capable. However, node B must create an IPv4 datagram to send to C.
- The data field of the IPv6 datagram is copied into the data field of the IPv4 datagram and appropriate address mapping is done.
- However, in performing the conversion from IPv6 to IPv4, there will be IPv6-specific fields in the IPv6 datagram that have no counterpart in IPv4. The information in these fields will be lost.

Tunneling

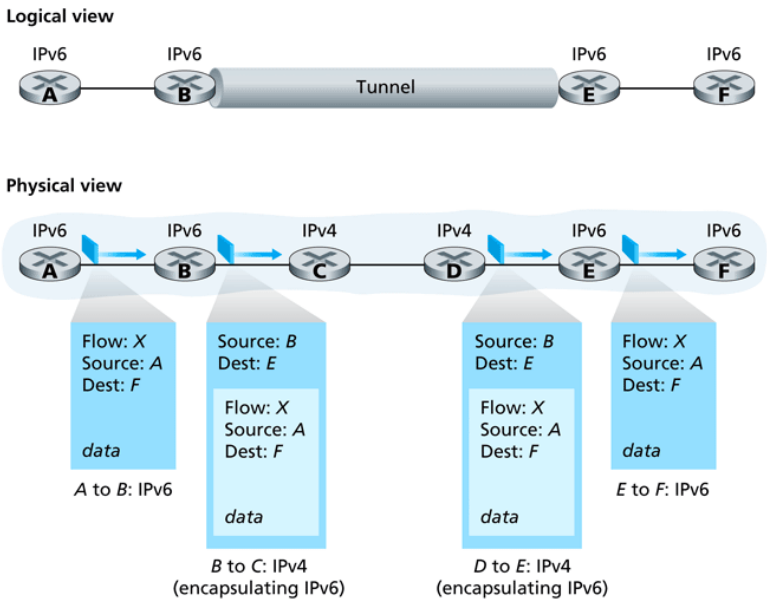


Figure 4.26 ♦ Tunneling

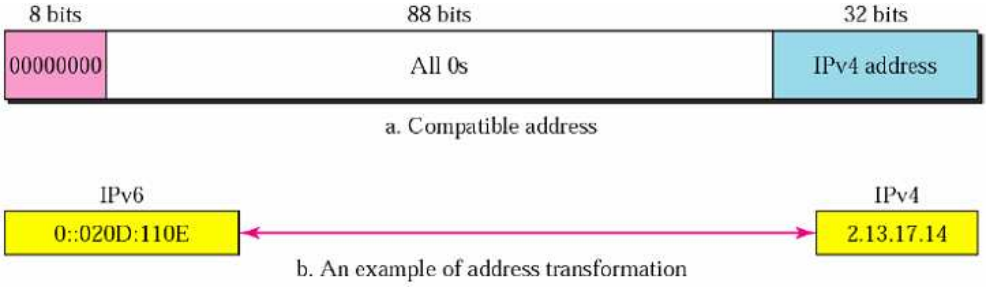
- With tunneling, the IPv6 node on the sending side of the tunnel (for example, B) takes the entire IPv6 datagram and puts it in the data (payload) field of an IPv4 datagram.
- This IPv4 datagram is then sent to the first node in the tunnel (for example, C).
- The intervening IPv4 routers in the tunnel route this IPv4 datagram among themselves.
- The IPv6 node on the receiving side of the tunnel eventually receives the IPv4 datagram (it is the destination of the IPv4 datagram), determines that the IPv4 datagram contains an IPv6 datagram, extracts the IPv6 datagram, and then routes the IPv6 datagram exactly as it would if it had received the IPv6 datagram from a directly connected IPv6 neighbor.

Converting IPv6 address into IPv4 address

During transition from IPv4 to IPv6, hosts can use their IPv4 addressees embedded in IPv6 addresses. Two formats have been designed for this purpose: *compatible* and *mapped*.

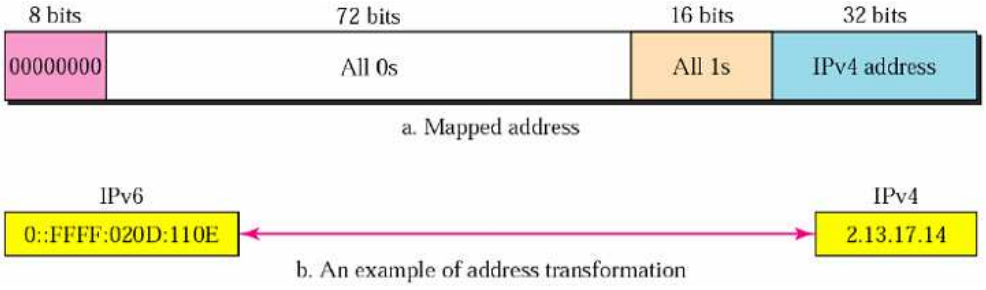
Compatible Address

Used when an IPv6-compatible node wants to send packets to another IPv6-compatible node via IPv4-compatible nodes.



Mapped Address

Used when an IPv6-compatible node wants to send a packet to a node capable of IPv4 only.



4.15 Routing Algorithms

The purpose of a routing algorithm is simple: given a set of routers, with links connecting the routers, a routing algorithm finds a "good" path from source router to destination router. Typically, a good path is one that has the least cost.

Classifications of Routing Algorithms

Global and *Decentralized* Routing Algorithms

A *global routing algorithm* computes the least-cost path between a source and destination using complete, global knowledge about the network. In practice, algorithms with global state information are often referred to as *link-state* (LS) algorithm, since the algorithm must be aware of the cost of each link in the network.

In a *decentralized routing algorithm*, the calculation of the least-cost path is carried out in an iterative, distributed manner. No node has complete information about the costs of all network links. Instead, each node begins with only the knowledge of the costs of its own directly attached links. Then, through an iterative process of calculation and exchange of information with its neighboring nodes, a node gradually calculates the least-cost path to a destination. The decentralized routing algorithm we'll study is called a *distance-vector* (DV) algorithm, because each node maintains a vector of estimates of the costs (distances) to all other nodes in the network.

Static and *Dynamic* Routing Algorithms

In *static routing algorithms*, routes change very slowly over time, often as a result of human intervention (for example, a human manually editing a router's forwarding table).

Dynamic routing algorithms change the routing paths as the network traffic loads or topology change. A dynamic routing algorithm can be run either periodically or in direct response to topology or link cost changes. While dynamic algorithms are more responsive to network changes, they are also more susceptible to problems such as routing loops and oscillation in routes.

Load-sensitive and *Load-insensitive* Routing Algorithms

In a *load-sensitive algorithm*, link costs vary dynamically to reflect the current level of cogestion in the underlying link. If a high cost is associated with a link that is currently congested, a routing algorithm will tend to choose routes around such a congested link.

Today's internet routing algorithms are *load-insensitive*, as a link's cost does not explicitly reflect its current (or recent past) level of congestion.

4.16 The Link-State (LS) Routing Algorithm

The link-state routing algorithm is known as the Dijkstra's algorithm. Dijkstra's algorithm is iterative and has the property that after the k^{th} iteration of the algorithm, the least-cost paths are know to k destination nodes, and among the least-cost paths to all destination nodes, these k paths will have the k smallest costs.

Complexity of LS Algorithm

Total number of nodes searched through over all the iterations = $n + (n-1) + (n-2) + \ldots + 1$

$$= \frac{n(n+1)}{2}$$

$\therefore$ Worst-case complexity = $O(n^2)$.

A more sophisticated implementation of this algorithm, using a data structure known as a heap, can find the minimum cost in logarithmic rather than linear time, thus reducing the complexity to $O(n\log n)$.

4.17 The Distance-Vector (DV) Routing Algorithm

The distance vector (DV) algorithm is iterative, asynchronous, and distributed.

It is *distribute*d in that each node receives some information from one or more of its directly attached neighbors, performs a calculation, and may then distribute the results of its calculation back to its neighbors.

It is *iterative* in that this process continues on until no more information is exchanged between neighbors.

The algorithm is *asynchronous* in that it does not require all of the nodes to operate in lock step with each other.

The Bellman-Ford equation

$$d_x(y) = min_v \{c(x, v) + d_v(y)\}$$

Here, $d_x(y)$ = cost of the least-cost path from node x to node y

The min_v in the equation is taken over all of x's neighbors, v.

Basic Idea of DV algorithm

- Each node x begins with $D_x(y)$, an estimate of the cost of the least-cost path from itself to node y, for all nodes in N.

  Let, $D_x = [D_x(y) : y \in N]$ be node x's distance vector, which is the vector of cost estimates from x to all other nodes y in N.

- Each node x maintains the following routing data:
  - For each neighbor v, the cost $c(x, v)$ from x to directly attached neghbor, v.
  - Node x's distance vector, that is, $D_x = [D_x(y) : y \in N]$, containing x's estimate of its cost to all destinations, y in N.
  - The distance vectors of each of its neighbors, i.e., $D_v = [D_v(y) : y \in N]$ for each neighbor v of x.

- In the distributed, asynchronous algorithm, from time to time, each node sends a copy of its distance vector to each of its neighbors. When a node *x* receives a new distance vector from any of its neighbors, *v*, it saves *v*'s distance vector, and then uses the Bellman-Ford equation to update its own distance vector as follows:

$$D_x(y) = min_v \{c(x, v) + D_v(y)\} \qquad \text{for each node } y \in N$$

- If node *x*'s distance vector has changed as a result of this update step, node x will then send its updated distance vector to each of its neighbors, which can in turn update their own distance vectors.

Example

The leftmost column of the figure displays three initial routing tables for each of the three nodes *x*, *y* and *z*.

Within a specific routing table, each row is a distance vector – specifically, each node's routing table includes its own distance vector and that of each of its neighbors. Thus, the first row in node *x*'s initial routing table is $D_x = [D_x(x), D_x(y), D_x(z)] = [0, 2, 7]$. The second and third rows in this table are the most recently received distance vectors from nodes *y* and *z*, respectively. Because at initialization, node *x* has not received anything from *y* or *z*, the entries in the 2nd and 3rd rows are initialized to infinity.

After initialization, each node sends its distance vector to each of its two neighbors. For example, node *x* sends its distance vector $D_x = [0, 2, 7]$ to both *y* and *z*. After receiving the updates, each node recomputes its own distance vector. For example, node *x* computes:

$$D_x(x) = 0$$

$$D_x(y) = \min \{c(x, y) + D_y(y), c(x, z) + D_z(y)\}$$
$$= \min \{2 + 0, 7 + 1\} = 2$$

$$D_x(z) = \min \{c(x, y) + D_y(z), c(x, z) + D_z(z)\} = \min \{2 + 1, 7 + 0\} = 3$$

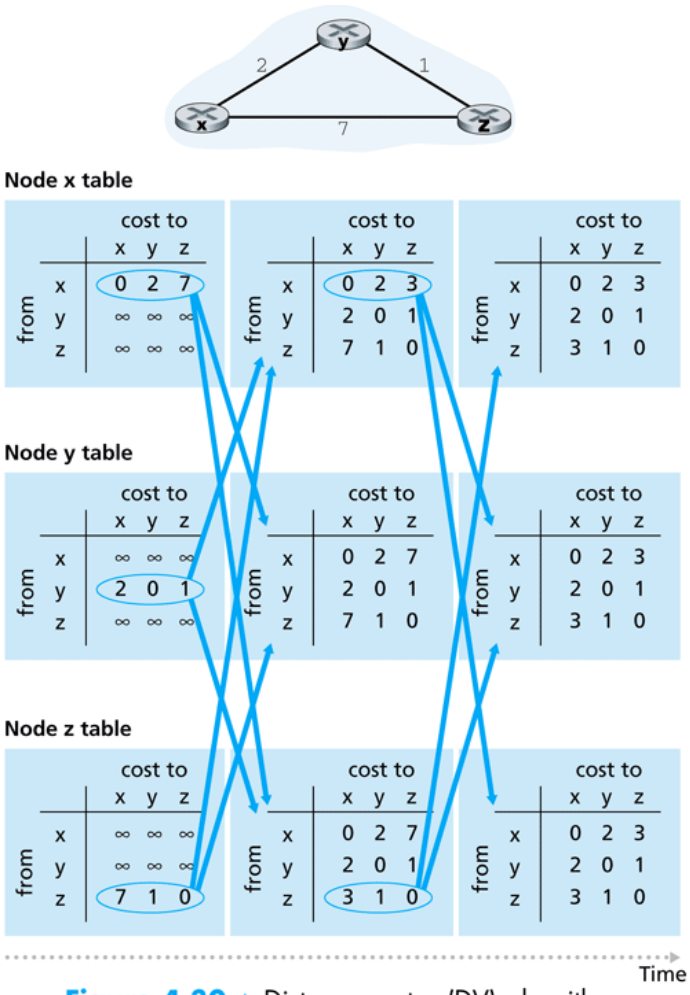


Figure 4.30 ♦ Distance-vector (DV) algorithm

The second column therefore displays, for each node, the node's new distance vector along with distance vectors just received from its neighbors. Note, for example, that node *x*'s estimate for the least cost to node *z*, $D_x(z)$ has changed from 7 to 3. Also note that for both nodes *y* and *z*, node *y* achieves the corresponding minimums. Thus, at this stage of the algorithm, the next-hop routers are $v^*(y) = y$ and $v^*(z) = y$, where v^* is the neighboring node that achieves the minimum.

After the nodes recompute their distance vectors, they again send their updated distance vectors to their neighbors (if there has been a change). This is illustrated in the figure by the arrows from the 2nd column of tables to the 3rd column of tables. Note that only nodes *x* and *z* send updates: node *y*'s distance vector didn't change so node *y* doesn't send an update. After receiving the updates, the nodes then recompute their distance vectors and update their routing tables, which are shown in the 3rd column.

The process of receiving updated distance vectors from neighbors, recomputing routing table entries, and informing neighbors of changed costs of the least-cost path to a destination continues until no update messages are sent. At this point, the algorithm will enter a quiescent state and remain there until a link cost changes.

Changes in Link Cost and Link Failure

Increase in Link Cost

The following figure (*a*) illustrates a scenario where the link cost from *y* to *x* changes from 4 to 1. We focus here only on *y* and *z*'s distance table entries to destination *x*. The DV algorithm causes the following sequence of events to occur:

- At time t_0, *y* detects the link cost change (the cost has changed from 4 to 1), updates its distance vector and informs its neighbors of this change since its distance vector has changed.
- At time t_1, *z* receives the update from *y* and then updates its table. Since it computes a new least cost to *x* (it has decreased from a cost of 5 to a cost of 2), it informs its neighbors.
- At time t_2, *y* has receives *z*'s update and updates its distance table. *Y*'s least costs do not change and hence *y* does not send any message to *z*. The algorithm comes to a quiescent state.

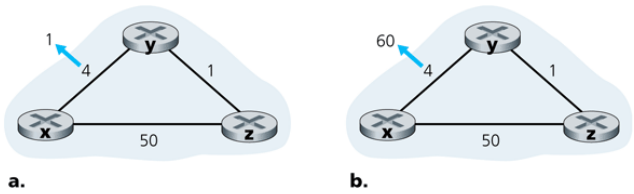


Figure 4.31 ♦ Changes in link cost

Decrease in Link Cost: Emergence of Routing Loop and Count-to-Infinity Problem

Suppose that the link cost between *x* and *y* increases from 4 to 60 as shown in the above figure (*b*).

- Before the link cost changes, $D_y(x) = 4$, $D_y(z) = 1$, $D_z(y) = 1$ and $D_z(x) = 5$. At time t_0, *y* detects the link cost change (the cost has changed from 4 to 60). *y* computes its new minimum cost path to *x* to have a cost of

  $$D_y(x) = \min \{c(y, x) + D_x(x), c(y, z) + D_z(x)\} = \min \{60 + 0, 1 + 5\} = 6$$

  Of course, with our global view of the network, we can see that this new cost via *z* is wrong. But the only information node *y* has is that its direct cost to *x* is 60 and that *z* has last told *z* that *z* could get to *x* with a cost of 5. So, in order to get to *x*, *y* would now route through *z*, fully expecting that *z* will be able to get to *z* with a cost of 5. As of t_1, we have a ***routing loop*** – in order to get to *x*, *y* routes through *z*, and *z* routes through *y*. A routing loop is like a black hole – a packet arriving at *y* or *z* as of t_1 will bounce back and forth between these two nodes forever (or until the routing tables are changed).
- Since node *y* has computed a new minimum cost to *x*, it informs *z* of this new cost at time t_1.
- Sometime after t_1, *z* receives the new least cost to *x* via *y* (*y* has told *z* that *y*'s new minimum cost is 6). *z* knows it can get to *y* with a cost of 1 and hence computes a new least cost to *z* (still via *y*) of 7. Since *y*'s least cost to *x* has increased, it then informs *y* of its new cost at t_2.
- In a similar manner, *y* then updates its table and informs *z* of a new cost of 8. *z* then updates its table and informs *y* of a new cost of 9 and so on…

This process will persist for 44 iterations until *z* eventually computes the cost of its path via *y* to be greater than 50. At this point, *z* will determine that its least-cost path to *x* is via its direct connection to *x*. *y* will then route to *x* via *z*.

What would have happened if the link cost $c(y, x)$ had changed from 4 to 10,000 and the cost $c(z, x)$ had been 9,999? This problem is sometimes referred to as count-to-infinity problem.

Poisoned Reverse

The specific looping scenario just described can be avoided using a technique known as *poisoned reverse*. The idea is simple – if *z* routes through *y* to get to destination *x*, then *z* will advertise to *y* that its (*z*'s) distance to *x* is infinity. *z* will continue telling this little "white lie" to *y* as long as it routes to *x* via *y*. Since *y* believes that *z* has no path to *x*, *y* will never attempt to route to *x* via *z*, as long as *z*

continues to route to x via y (and lie about doing so).

Let's now see how poisoned reverse solves the particular looping problem we encountered before. As a result of the poisoned reverse, y's distance table indicates an infinite cost when routing to x via Z (the result of z having informed y that z's cost to x was infinity). When the cost of the xy link changes from 4 to 60 at time t_0, y updates its table and continues to route directly to x, albeit at a higher cost of 60, and informs z of this change in cost. After receiving the update at t_1, z immediately shifts it route to x to be via the direct zx link at a cost of 50. Since this is a new least cost to x, and since the path no longer passes through y, z informs y of this new least cost path to x at t_2. After receiving the update from z, y updates its distance table to route to x via z at a least cost of 51. Also, since z is now on y's least path to x, y poisons the reverse path from z to x by informing z at time t_3 that it (y) has an infinite cost to get to x.

Does poison reverse solve the general count-to-infinity problem? It does not. Loops involving three or more nodes (rather than simply two immediately neighboring nodes, as we saw in Figure 4.31) will not be detected by the poison reverse technique.

4.18 Comparison of LS and DV Routing Algorithms

	LS	DV
Message complexity	With N nodes and E links, $O(\|N\|\|E\|)$ messages need to be sent.	Messages need to be exchanged among neighbors only.
Speed of convergence	$O(\|N\|^2)$ algorithm requiring $O(\|N\|\|E\|)$ messages.	- Can converge slowly. - Can have *routing loops* while converging. - Can suffer from *count-to-infinity* problem.
Robustness: what happens if a router malfunctions?	As an LS node is only computing its own routing table, route calculations are somewhat separated under LS, providing a degree of robustness.	A node can advertise incorrect least path costs to *any/all* destinations.

4.19 Hierarchical Routing

In our study of LS and DV algorithms, our view of the network simply as a collection of interconnected routers has two limitations:

1. **Scale.**

   As the number of routers becomes large, the overhead involved in computing, storing and communicating the routing table information (e.g., link state updates or least cost path changes) becomes prohibitive. Today's public Internet consists of millions of interconnected routers and more than 50 million hosts. Storing routing table entries to each of these hosts and routers would clearly require enormous amounts of memory.

2. **Administrative Autonomy.**

   Although engineers tend to ignore issues such as a company's desire to run its routers as it pleases (e.g., to run whatever routing algorithm it chooses), or to *hide* aspects of the networks' internal organization from the outside, these are important considerations. Ideally, an organization should be able to run and administer its network as it wishes, while still being able to connect its network to other *outside* networks.

Autonomous Systems (AS)

Both of these problems can be solved by organizing routers into autonomous systems (AS), with each AS consisting of a group of routers that are typically under the same administrative control.

- Routers within the same AS all run the same routing algorithm.
- The routing algorithm running within an autonomous system is called an ***intra-autonomous system routing protocol*** or ***interior gateway protocols***.

- It will be necessary, of course, to connect Ass to each other, and thus one or more of the routers in an AS will have the added task of being responsible for forwarding packets to destinations outside the AS; these routers are called ***gateway routers***.

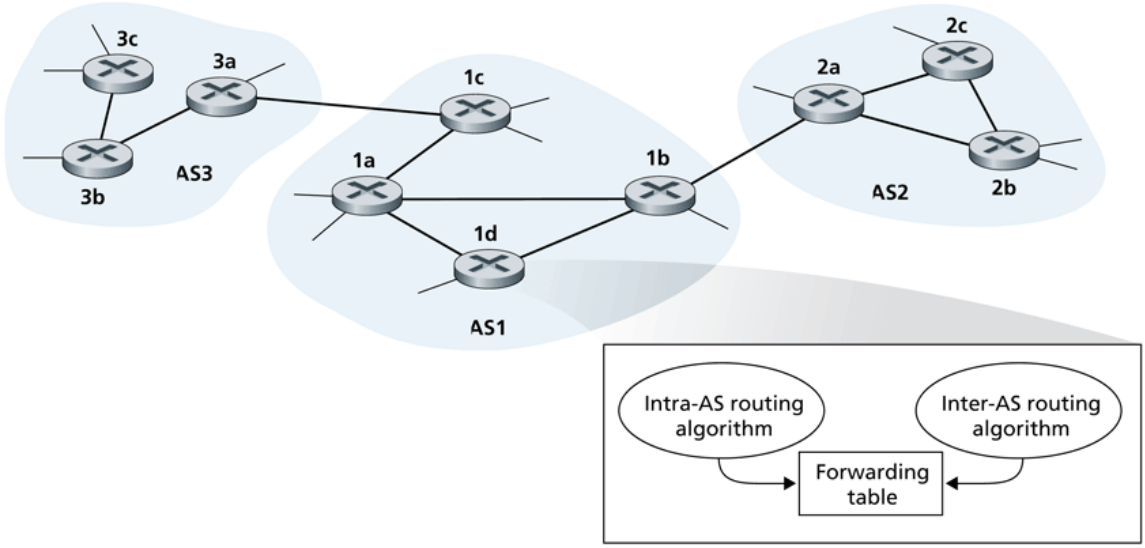


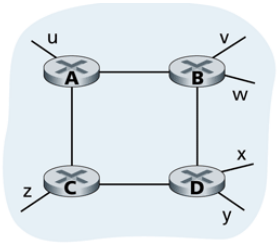
Figure 4.32 ♦ An example of interconnected autonomous systems

How does a router within some AS know how to route a packet to a destination that is outside the AS?

- It's easy to answer this question if the AS has only one gateway router that connects to only one other AS. The router would simply choose the least-cost path to the gateway router and send the packet to it. The gateway router would forward the packet outside the AS.
- If the AS has more than one gateways (e.g., AS1 in the above figure), then the problem of knowing where to forward the packet becomes more challenging. To solve this, the AS needs
  1. to learn which destinations are reachable via which AS (e.g., which are via AS2 and which are via AS3).
  2. to propagate this reachability information to all the routers within the AS (AS1), so that each router can configure its forwarding table to handle external-AS destinations.
- These two tasks are handled by the ***inter-AS routing protocol***.
- Since the inter-AS routing protocol involves communication between two Ass, the two communicating Ass must run the same inter-AS routing protocol. In fact, in the internet, all Ass run the same inter-AS routing protocol called BGP4.
- Each router receives information from an intra-AS routing protocol and an inter-AS routing protocol and uses the information from both protocols to configure its forwarding table.
  - For example, if AS1 learns from inter-AS routing protocol that subnet *x* is reachable via *only* AS3, all the routers in AS1 will add an entry to their table that to send a packet to destination *x*, it must be forwarded to gateway 1c.
  - Again, if AS1 learns from inter-AS routing protocol that subnet *x* is reachable via both AS2 and AS3, all the routers in AS1 will add an entry to their table that to send a packet to destination *x*, it must be forwarded to the gateway *having the least-cost path* from the router.
- When an AS learns about a destination from a neighboring AS, the AS can advertise this routing information to some of its other neighboring ASs.
  - For example, suppose AS1 learns from AS2 that subnet *x* is reachable via AS2. AS1 could then tell AS3 that *x* is reachable via AS1.

4.20 Intra-AS Routing in the Internet: RIP (Routing Information Protocol)

- RIP is a DV protocol.
- RIP uses hop count as cost metric; i.e., each link has a cost of 1. Costs are actually from source router to a destination subnet rather than between a pair of routers. RIP uses the term *hop*, which is the number of subnets traversed along the shortest path from source router to destination subnet. The figure beside illustrates an AS with six leaf subnets. The table in the figure indicates the number of hops from the source A to all the leaf subnets.



Destination	Hops
u	1
v	2
w	2
x	3
y	3
z	2

Figure 4.34 ♦ Number of hops from source router A to various subnets

- The maximum cost of a path is limited to 15, thus limiting the use of RIP to autonomous systems that are fewer than 15 hops in diameter.
- Routing updates are exchanged between neighbors approximately every 30 seconds using a ***RIP response message***. The response message sent by a router or host contains a list of up to 25 destination subnets within the AS, as well as the sender's distance to each of those subnets. Response messages are also known as ***RIP advertisements***.
- If a router does not hear from its neghbor at least once every 180 seconds, that neighbor is considered to be no longer reachable. When this happens, RIP modifies the local routing table and then propagates this information by sending advertisements to its neighboring routers.
- A router can also request information about its neighbor's cost to a given destination using RIP's request message. Routers send RIP request and response messages to each other over UDP using port number 520. (In this case, RIP is implemented as an application layer protocol in a UNIX system serving as a router.)

How RIP Advertisement Works

Consider the portion of an AS shown in the figure beside. Each router maintains a RIP table known as a routing table. A router's routing table includes both the router's distance vector and the router's forwarding table. Figure 4.36 shows the routing table for router *D*.

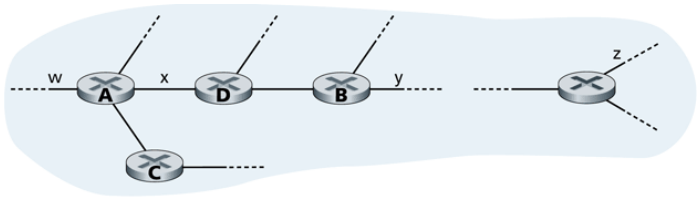


Figure 4.35 ♦ A portion of an autonomous system

Now suppose that 30 seconds later, router *D* receives from router *A* the advertisement shown in figure 4.37. Note that this advertisement is nothing other than the routing table information from router *A*.

Destination Subnet	Next Router	Number of Hops to Destination
w	A	2
y	B	2
z	B	7
x	—	1
....	....	....

Figure 4.36 ♦ Routing table in router *D* before receiving advertisement from router *A*

Router *D*, upon receiving this advertisement, merges the advertisement with the old routing table. In particular, router *D* learns that there is now a path through router *A* to subnet *z* that is shorter than the path through router *B*. Thus, router *D* updates its routing table to account for the shorter shortest path.

Destination Subnet	Next Router	Number of Hops to Destination
z	C	4
w	—	1
x	—	1
....	....	....

Figure 4.37 ♦ Advertisement from router A

4.21 **Intra-AS Routing in the Internet: OSPF (Open Shortest Path First)**

- OSPF and its closely related cousin, IS-IS, are typically deployed in upper-tier ISPs whereas RIP is deployed in lower-tier ISPs and enterprise networks.
- A router broadcasts a link's state periodically (at least every 30 minutes) to all other routers in the AS, even if the link's state has not changed.
- OSPF advertisements are contained in OSPF messages that are carried directly by IP, with an upper-layer protocol of 89 for OSPF.
- An OSPF AS can be configured into areas. Each area runs its own OSPF link-state routing algorithm.
- Within each area, one or more ***area border routers*** are responsible for routing packets outside the area.
- Exactly one OSPF area in the AS is configured to be the ***backbone*** area. The primary role of the backbone area is to route traffic between the other areas in the AS. The backbone always contains all area border routers in the AS and may contain nonborder routers as well.
- Inter-area routing within the AS requires that the packet be first routed to an area border router (intra-area routing), then routed through the backbone to the area border router that is in the destination area, and then routed to the final destination.
- A ***boundary router*** exchanges routing information with routers belonging to other ASs. This router might, for example, use BGP to perform inter-AS routing. It is through such a boundary router that other router learn about paths to external networks.

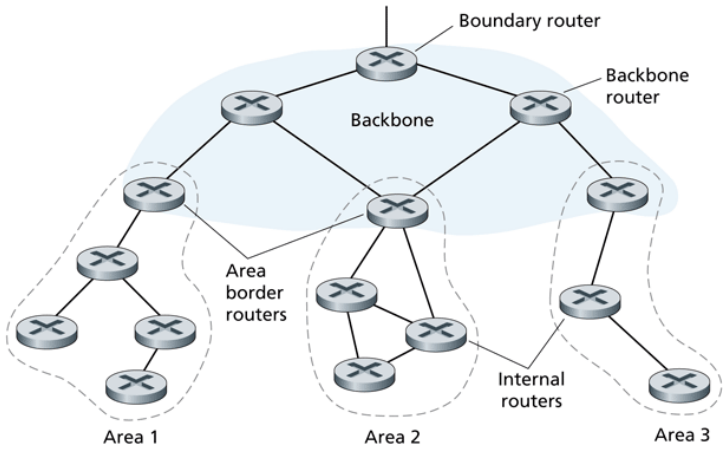


Figure 4.40 ♦ Hierarchically structured OSPF AS with four areas

CHAPTER 5
LINK LAYER

What we are going to discuss in this chapter

1. **The services provided by link layer**
2. **Error detection and correction techniques**
    a. Parities and FEC (Forward Error Correction)
    b. Checksum
    c. CRC (Cyclic Redundancy Check)
3. **Multiple Access Protocols**
    a. Channel Partitioning Protocols
        i. TDMA (Time Division Multiple Access)
        ii. FDMA (Frequency Division Multiple Access)
        iii. CDMA (Code Division Multiple Access)
    b. Random Access Protocols
        i. Slotted ALOHA
        ii. ALOHA
        iii. CSMA
    c. Taking Turns Protocol
        i. Reservation
        ii. Polling
        iii. Token Passing
4. **Link-layer addressing**
    a. MAC (Media Access Control)
    b. ARP (Address Resolution Protocol)
    c. DHCP (Dynamic Host Configuration Protocol)
5. **Ethernet:** Details of the implementation of a multiple access protocol (CSMA/CD)
6. **Hubs and Switches:** Interconnections among LANs
7. **PPP** (Point-to-point Protocol)

5.1 Types of link-layer channels

There are two fundamentally different types of link-layer channels:

1. **Broadcast Channels:**

   Many hosts are connected to the same communication channel, and a so-called medium access protocol is needed to coordinate transmissions and avoid collisions.

   Examples: LANs, wireless LANs, satellite networks, HFC (Hybrid Fiber Coaxial Cable) access networks.

2. **Point-to-point Communication Link:**

   Examples: Between two routers, between a residential dial-up modem and as ISP router.

5.2 Link-layer Protocols

1. Ethernet
2. Wi-Fi (802.11 wireless LAN)
3. Token Ring
4. PPP (Point-to-Point Protocol)

5.3 Some Important Notes

An important characteristic of the link layer is that a datagram may be handled by different link-layer protocols on the different links in the path. For example, a datagram may be handled by Ethernet on the first link, PPP on the last link, and frame relay on all intermediate links.

It is important to note that the services provided by the different link-layer protocols may be different. For example, a link-layer protocol may or may not provide reliable delivery. Thus, the network layer must be able to accomplish its end-to-end job in the face of a varying set of individual link-layer services.

5.4 Adapters / Network Interface Cards (NICs)

For a given communication link, the link-layer protocol is for the most part implemented in an adapter. Adapters are also commonly known as network interface cards or NICs.

As shown in the figure, the main components of an adapter are the bus interface and the link interface.

The bus interface is responsible for communicating with the adapter's parent node. It sends to and receives from the parent node network-layer datagram and control information.

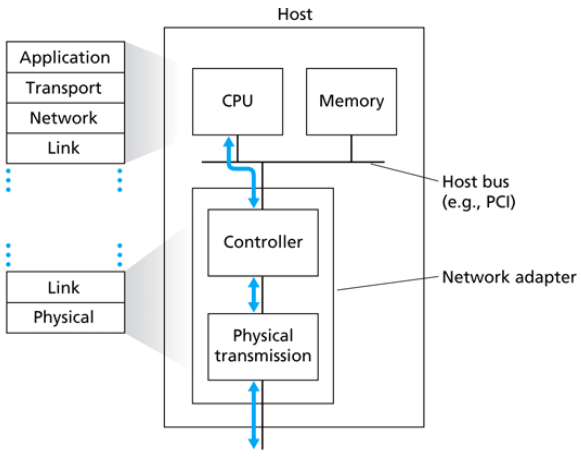


Figure 5.2 ♦ Network adapter: its relationship to other host components, and to protocol stack functionality

The link interface is responsible for implementing the link-layer protocol. In addition to framing and de-framing datagram, it may provide error detection, random access and other link-layer functions. It also includes the transmit and receive circuitry.

5.5 Error Detection and Correction Techniques

General Technique

Before sending, compute the error detection bits and append them to the message. After receiving, check the whole message and determine whether error has occurred.

5.6 Parity Checks

In an even parity scheme, the sender simply includes one additional bit and chooses its value such that the total number of 1's in the $d + 1$ bits (the original information plus a parity bit) is even. For odd parity schemes, the parity bit value is chosen such that there are an odd number of 1's.

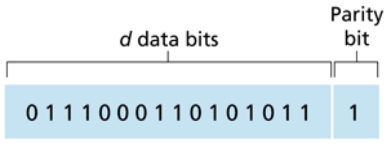


Figure 5.5 ♦ One-bit even parity

Receiver operation is also simple with a single parity bit. The receiver need only count the number of 1's in the received $d + 1$ bits. If an odd number of 1-valued bits are found with an even parity scheme, the receiver knows that at least one bit error has occurred. More precisely, it knows that some *odd* number of bit errors has occurred.

Disadvantage

Occurrence of an even number of bit errors cannot be detected.

5.7 FEC (Forward Error Correction)

Figure 5.2-3 shows a two-dimensional generalization of the single-bit parity scheme. Here, the d bits in D are divided into i rows and j columns. A parity value is computed for each row and for each column. The resulting $i + j + 1$ parity bits are the data link frame's error detection bits.

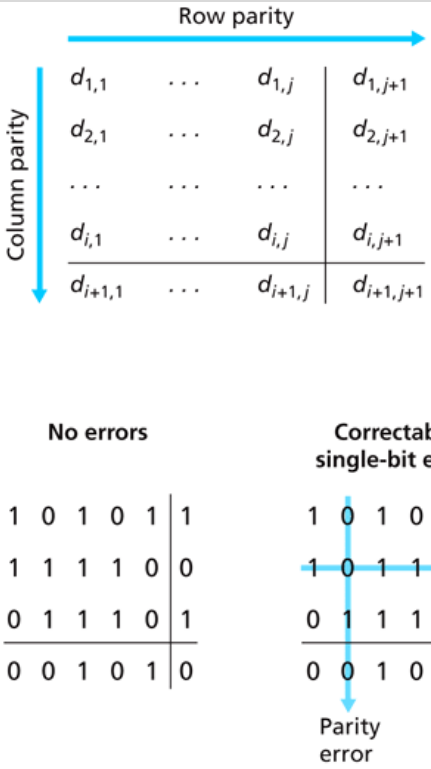


Figure 5.6 ♦ Two-dimensional even parity

Suppose now that a single bit error occurs in the original d bits of information. With this two-dimensional parity scheme, the parity of both the column and the row containing the flipped bit will be in error. The receiver can thus not only detect the fact that a single bit error has occurred, but can use the column and row indices of the column and row with parity errors to actually identify the bit that was corrupted and correct that error!

The ability of the receiver to both detect and correct errors is known as *forward error correction* (FEC).

Application

These techniques are commonly used in audio storage and playback devices such as audio CD's.

Advantages

1. Decreases the number of sender retransmissions required.
2. Allows for immediate correction of errors at the receiver. This avoids having to wait the round-trip propagation delay needed for the sender to receive a NAK packet and for the retransmitted packet to propagate back to the receiver – a potentially important advantage for real-time network applications.

5.8 Internet Checksum

5.9 Cyclic Redundancy Check (CRC)

5.10 Multiple Access Protocols

The Problem

Because all nodes are capable of transmitting frames, more than two nodes can transmit frames at the same time. When this happens, all of the nodes receive multiple frames at the same time, that is, the transmitted frames collide at all of the receivers. Typically, when there is a collision, none of the receiving nodes can make any sense of any of the frames that were transmitted; in a sense, the signals

of the colliding frame become inextricably tangled together. Thus, all the frames involved in the collision are lost, and the broadcast channel is wasted during the collision interval. Clearly, if many nodes want to frequently transmit frames, many transmissions will result in collisions, and much of the bandwidth of the broadcast channel will be wasted.

So, how can we coordinate the access of multiple sending and receiving nodes to a shared broadcast channel?

General Idea for a Solution

In order to ensure that the broadcast channel performs useful work when multiple nodes are active, it is necessary to somehow coordinate the transmissions of the active nodes. This coordination job is the responsibility of the multiple access protocol.

What should be the general characteristics of a multiple access protocol

A multiple access protocol for a broadcast channel of rate R bits per second should have the following desirable characteristics:

1. When only *one* node has data to send, that node has a throughput of R bps.
2. When M nodes have data to send, each of these nodes has a throughput of R/M bps. This need not necessarily imply that each of the M nodes always have an instantaneous rate of R/M , but rather that each node should have an average transmission rate of R/M over some suitably-defined interval of time.
3. The protocol is decentralized, i.e., there are no master nodes that can fail and bring down the entire system.
4. The protocol is simple, so that it is inexpensive to implement.

Some multiple access protocols

1. **Channel Partitioning Protocols:** TDMA, FDMA and CDMA
2. **Random Access Protocols:** Slotted ALOHA, ALOHA, CSMA
3. **Taking-Turns Protocols:** Polling, Token Passing

5.11

Channel Partitioning Protocols

The broadcast channel's bandwidth is partitioned among all nodes sharing that channel.

TDMA (Time-Division Multiple Access)

Suppose the channel supports N nodes and that the transmission rate of the channel is R bps.

TDMA divides time into time frames and further divides each time frame into N time slots. Each slot time is then assigned to one of the N nodes.

Whenever a node has a frame to send, it transmits the frame's bits during its assigned time slot in the revolving TDMA frame. Typically, frame sizes are chosen so that a single frame can be transmitting during a slot time.

Advantages

1. Eliminates collision.
2. Perfectly fair: each node gets a dedicated transmission rate of R/N bps during each slot time.

Disadvantages

1. A node is limited to this rate of R/N bps over a slot's time even when it is the *only* node with frames to send.

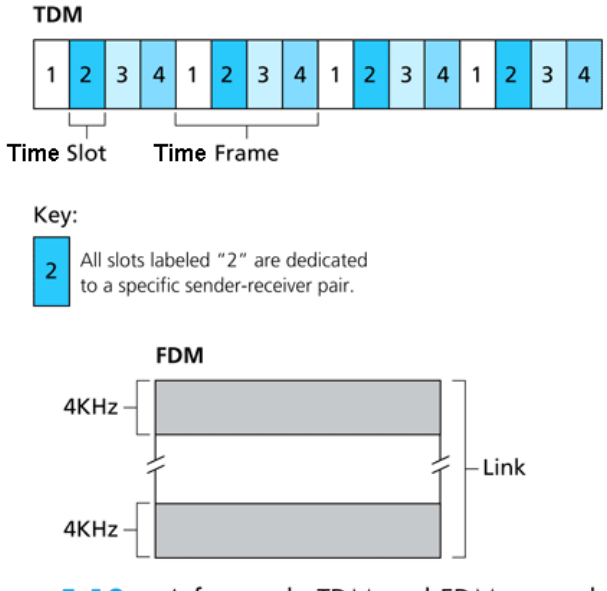


Figure 5.10 ♦ A four-node TDM and FDM example

2. A node must always wait for its turn in the transmission sequence – again, even when it is the *only* node with a frame to send.

FDMA (Frequency-Division Multiple Access)

FDMA divides the *R* bps channel into different frequencies (each with a bandwidth of *R*/*N*) and assigns each frequency to one of the *N* nodes. FDMA thus creates *N* "smaller" channels of *R*/*N* bps out of the single, "larger" *R* bps channel.

Advantages and Disadvantages

Same as the advantages and disadvantages of TDMA except the *second* disadvantage.

Code Division Multiple Access (CDMA)

CDMA assigns a different *code* to each node. Each node then uses its unique code to encode the data bits it sends. Because CDMA's use is so tightly tied to wireless channels, we'll save our discussion of the technical details of CDMA until chapter 6 (Wireless and Mobile Networks).

5.12 **Random Access Protocols**

In a random access protocol, a transmitting node always transmits at the full rate of the channel, namely, R bps. When there is a collision, each node involved in the collision repeatedly retransmits its frame until the frame gets through without a collision. But when a node experiences a collision, it doesn't necessarily retransmit the frame right away. Instead it waits a random delay before retransmitting the frame. Each node involved in a collision chooses independent random delays. Because after a collision the random delays are independently chosen, it is possible that one of the nodes will pick a delay that is sufficiently less than the delays of the other colliding nodes, and will therefore be able to "sneak" its frame into the channel without a collision.

Some random access protocols

1. Pure ALOHA (Areal Locations Of Hazardous Atmospheres)
2. Slotted ALOHA
3. CSMA (Carrier-Sense Multiple Access)
4. CSMA/CD (CSMA with Collision Detection)
5. CSMA/CA (CSMA with Collision Avoidance)

5.13 **General Idea of the ALOHA Protocols**

In ALOHA protocols, a node waits for an arbitrary amount of time before sending a new frame or before retransmitting a frame after a collision.

The amount of time to be waited is said to be a probability, *p*. *p* is the probability of taking the decision that it should send the frame. It's like tossing a coin; if it's the head, the node will send the frame, if it's the tail, it will wait for some fixed amount of time, and then toss again to decide whether it should send the frame this time.

Pure ALOHA

In pure ALOHA, when a frame first arrives (i.e., a network layer datagram is passed down from the network layer at the sending node), the node *immediately* transmits the frame in its entirety into the broadcast channel. If a transmitted frame experiences a collision with one or more other transmissions, the node will then immediately (after completely transmitting its collided frame) retransmit the frame with probability *p*. Otherwise, the node waits for a frame transmission time. After this wait, it then transmits the frame with probability *p*, or waits (remaining idle) for another frame time with probability 1 – *p*.

Slotted ALOHA

The operation of slotted ALOHA in each node is simple:

1. When the node has a fresh frame to send, it waits until the beginning of the next slot and

transmits the entire frame in the slot.

2. If there isn't a collision, the node won't consider retransmitting the frame. (The node can prepare a new frame for transmission, if it has one.)
3. If there is a collision, the node detects the collision before the *end* of the slot. The node retransmits its frame in each subsequent slot with probability *p* until the frame is transmitted without a collision.

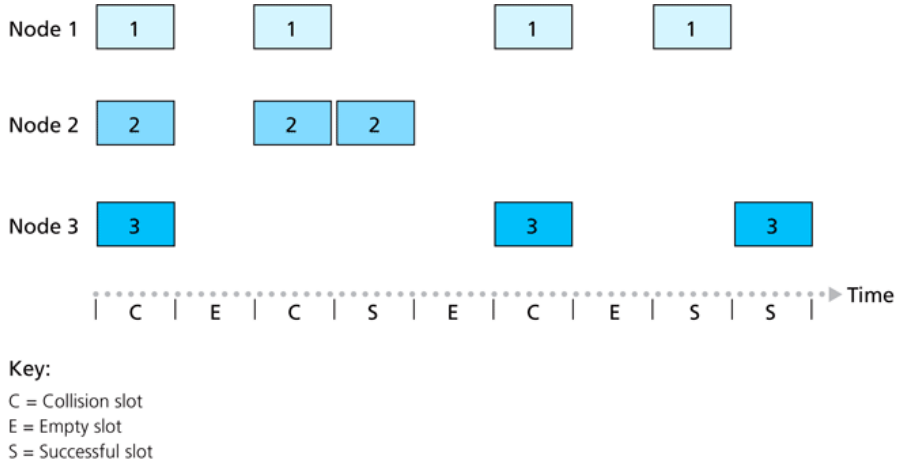


Figure 5.11 ♦ Nodes 1, 2, and 3 collide in the first slot. Node 2 finally succeeds in the fourth slot, node 1 in the eighth slot, and node 3 in the ninth slot.

Advantages of slotted ALOHA

1. Allows a node to transmit continuously at full rate even when that node is the only active node.
2. Highly decentralized, because each node detects collisions and independently decides when to retransmit.
3. Extremely simple protocol.

Efficiency Concerns with ALOHA

Slotted ALOHA also works great when there is only one active node, but how efficient is it when there are multiple active nodes? There are two possible efficiency concerns here.

1. As shown in Figure 5.11, when there are multiple active nodes, a certain fraction of the slots will have collisions and will therefore be "wasted."
2. The second concern is that another fraction of the slots will be empty because all active nodes refrain from transmitting as a result of the probabilistic transmission policy.

The only "unwasted" slots will be those in which exactly one node transmits. A slot in which exactly one node transmits is said to be a *successful slot*.

The efficiency of a slotted multiple access protocol is defined to be the long-run fraction of successful slots when there are a large number of active nodes, with each node having a large number of frames to send.

Note that if no form of access control were used, and each node were to immediately retransmits after each collision, the efficiency would be zero. Slotted ALOHA clearly increases the efficiency beyond zero, but by how much?

Maximum Efficiency of Slotted ALOHA

To keep the derivation of maximum efficiency of Slotted ALOHA simple, let's modify the protocol a little and assume that each node attempts to transmit a frame in each slot with probability *p*. (That is, we assume that each node always has a frame to send and that the node transmits with probability *p* for a fresh frame as well as for a frame that has already suffered a collision.)

Let, the number of nodes = *N*.

Then the probability that a given slot is a successful slot is the probability that one of the nodes transmits and that the remaining $N - 1$ nodes do not transmit.

Let, Probability that a given node transmits = *p*

∴ Probability that the remaining $N - 1$ nodes do not transmit = $(1 - p)^{N-1}$

∴ Probability that a given node has success = $p(1 - p)^{N-1}$

∴ Probability that an arbitrary node (among N nodes) has success = $N\,p(1 - p)^{N-1}$

∴ Efficiency of Slotted ALOHA = $N\,p(1 - p)^{N-1}$

∴ Maximum efficiency for N active nodes?

We have to find p^* that maximizes the expression $N\,p^*(1 - p^*)^{N-1}$.

∴ Maximum efficiency for a large number of nodes?

Let N = ∞

After performing these calculations, we'll find that the maximum efficiency of the protocol is given by 1/e = .37. That is, when a large number of nodes have many frames to transmit, then (at best) only 37% of the slots do useful work. Thus the effective transmission rate of the channel is not R bps but only .37 R bps!

Maximum Efficiency of Pure ALOHA

We focus on an individual node. We'll make the same assumptions as in our slotted ALOHA analysis and take the frame transmission time to be the unit of time.

At any given time, the probability that a node is transmitting a frame is p.

Suppose this frame begins transmission at time t_0.

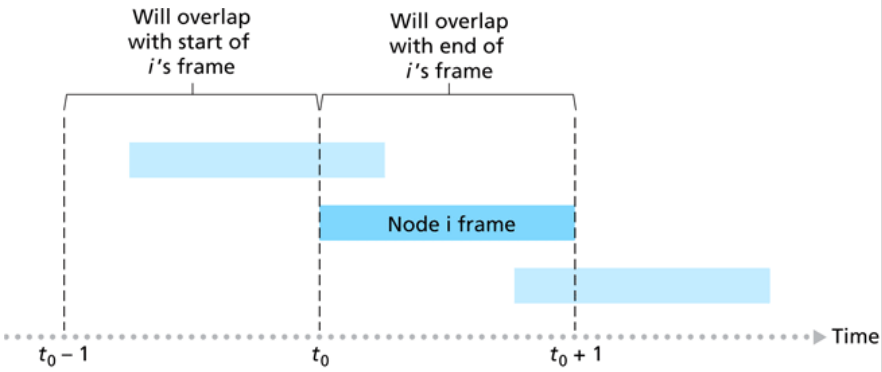


Figure 5.12 ♦ Interfering transmissions in pure ALOHA

As shown in Figure 5.12, in order for this frame to be successfully transmitted, no other nodes can begin their transmission in the interval of time $[t_0\text{-}1, t_0]$. Such a transmission would overlap with the beginning of the transmission of node i's frame.

The probability that all other nodes do not begin a transmission in this interval is $(1 - p)^{N-1}$.

Similarly, no other node can begin a transmission while node i is transmitting, as such a transmission would overlap with the latter part of node i's transmission.

The probability that all other nodes do not begin a transmission in this interval is also $(1 - p)^{N-1}$.

Thus, the probability that a given node has a successful transmission is $p(1 - p)^{2(N-1)}$.

By taking limits as in the slotted ALOHA case, we find that the maximum efficiency of the pure ALOHA protocol is only 1/(2e) – exactly half that of slotted ALOHA.

Disadvantages of ALOHA

In both slotted and pure ALOHA, a node neither pays attention to whether another node happens to be transmitting when it begins to transmit, nor stops transmitting if another node begins to interfere with its transmission.