# Today's Agenda

- Previous class review (refresher)
  - Routing
    - Introduction
    - Distance Vector
    - Link State
- Today's Class
  - UDP Protocol
  - TCP Protocol
    - Connection setup
    - State Transitions
    - Connection Teardown

# Due Dates

- Programming Assignment I (due March 29,11:59PM)
- ~~Term Paper Email (due February 5, 11:59PM)~~
- ~~Term Paper Proposal (due February 19, 11:59PM)~~
- **Midterm Examination (March 12)**
- Full term Paper (due May 6, 11:59PM)

# Previous Class

# Routing

## Forwarding versus Routing

- Forwarding:
  - to select an output port based on destination address and routing table
- Routing:
  - process by which routing table is built

# Routing

- Forwarding table VS Routing table
  - Forwarding table
    - Used when a packet is being forwarded and so must contain enough information to accomplish the forwarding function
    - A row in the forwarding table contains the mapping from a network number to an outgoing interface and some MAC information, such as Ethernet Address of the next hop
  - Routing table
    - Built by the routing algorithm as a precursor to build the forwarding table
    - Generally contains mapping from network numbers to next hops

# Routing

(a)

| Prefix/Length | Next Hop |
|---|---|
| 18/8 | 171.69.245.10 |

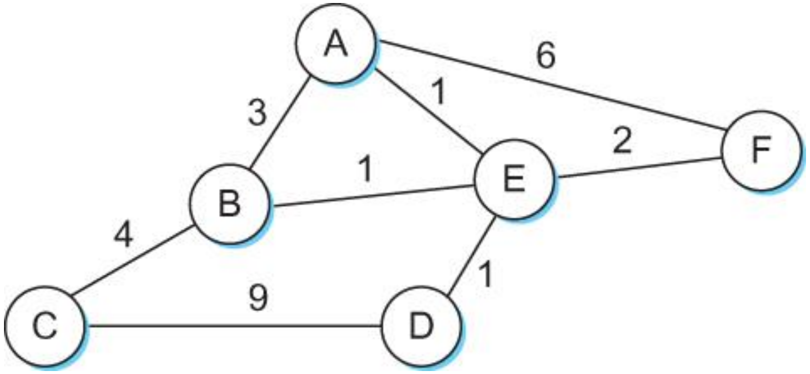
(b)

| Prefix/Length | Interface | MAC Address |
|---|---|---|
| 18/8 | if0 | 8:0:2b:e4:b:1:2 |

Example rows from (a) routing and (b) forwarding tables

# Routing

- Network as a Graph




- The basic problem of routing is to find the lowest-cost path between any two nodes
  - Where the cost of a path equals the sum of the costs of all the edges that make up the path

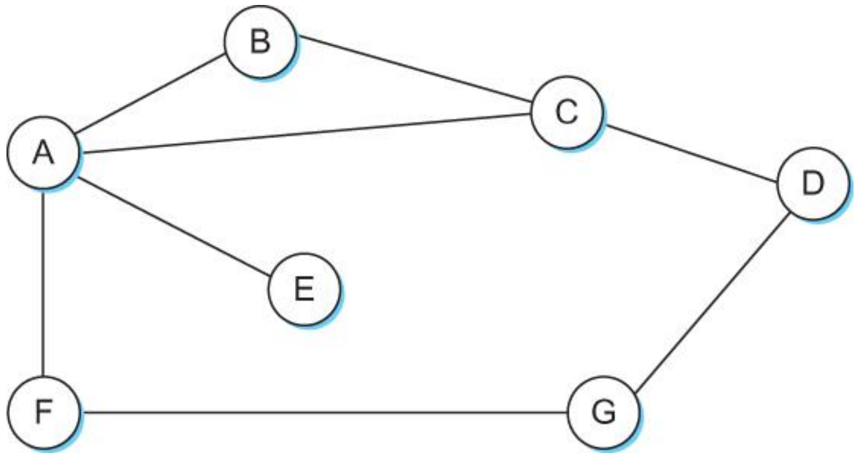
# Routing

- For a simple network, we can calculate all shortest paths and load them into some nonvolatile storage on each node.
- Such a static approach has several shortcomings
  - It does not deal with node or link failures
  - It does not consider the addition of new nodes or links
  - It implies that edge costs cannot change

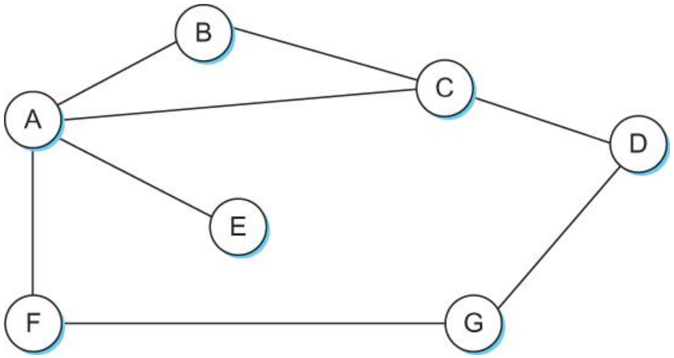
- What is the solution?
  - Need a distributed and dynamic protocol
  - Two main classes of protocols
    - Distance Vector
    - Link State

# Distance Vector

- Each node constructs a one dimensional array (a vector) containing the “distances” (costs) to all other nodes and distributes that vector to its immediate neighbors
- Starting assumption is that each node knows the cost of the link to each of its directly connected neighbors



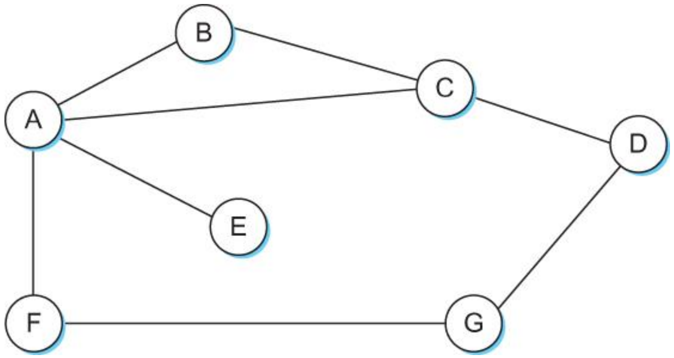
# Distance Vector




| Information Stored at Node | Distance to Reach Node | | | | | | |
|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G |
| A | 0 | 1 | 1 | ∞ | 1 | 1 | ∞ |
| B | 1 | 0 | 1 | ∞ | ∞ | ∞ | ∞ |
| C | 1 | 1 | 0 | 1 | ∞ | ∞ | ∞ |
| D | ∞ | ∞ | 1 | 0 | ∞ | ∞ | 1 |
| E | 1 | ∞ | ∞ | ∞ | 0 | ∞ | ∞ |
| F | 1 | ∞ | ∞ | ∞ | ∞ | 0 | 1 |
| G | ∞ | ∞ | ∞ | 1 | ∞ | 1 | 0 |

Initial distances stored at each node (global view)

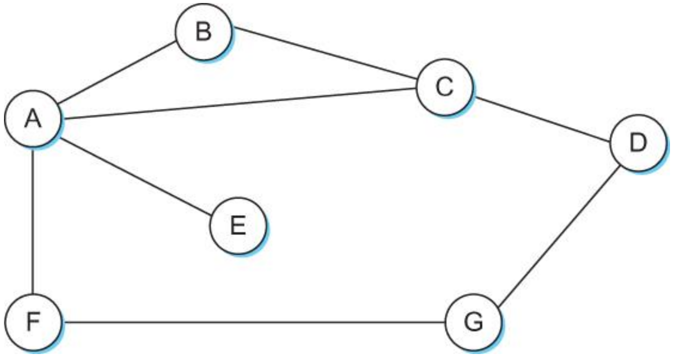
# Distance Vector




| Destination | Cost | NextHop |
|---|---|---|
| B | 1 | B |
| C | 1 | C |
| D | ∞ | — |
| E | 1 | E |
| F | 1 | F |
| G | ∞ | — |

Initial routing table at node A

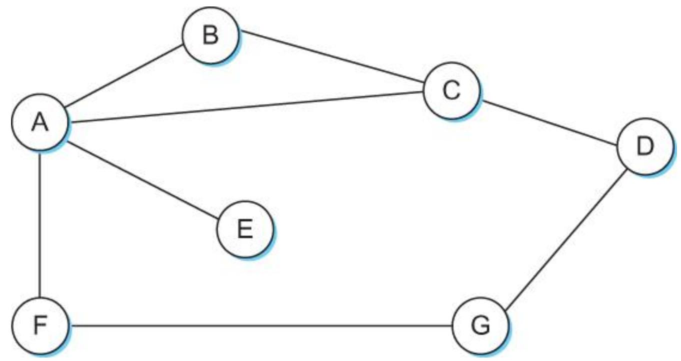
# Distance Vector




| Destination | Cost | NextHop |
|---|---|---|
| B | 1 | B |
| C | 1 | C |
| D | 2 | C |
| E | 1 | E |
| F | 1 | F |
| G | 2 | F |

Final routing table at node A

# Distance Vector




| Information Stored at Node | Distance to Reach Node | | | | | | |
|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G |
| A | 0 | 1 | 1 | 2 | 1 | 1 | 2 |
| B | 1 | 0 | 1 | 2 | 2 | 2 | 3 |
| C | 1 | 1 | 0 | 1 | 2 | 2 | 2 |
| D | 2 | 2 | 1 | 0 | 3 | 2 | 1 |
| E | 1 | 2 | 2 | 3 | 0 | 2 | 3 |
| F | 1 | 2 | 2 | 2 | 2 | 0 | 1 |
| G | 2 | 3 | 2 | 1 | 3 | 1 | 0 |

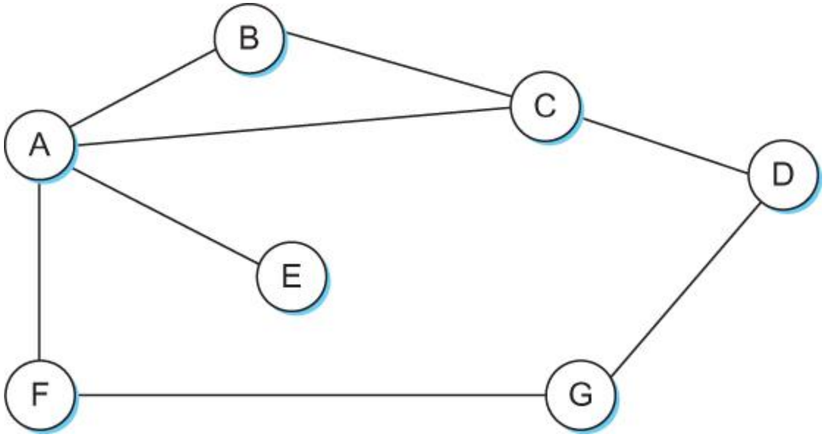
Final distances stored at each node (global view)

# Distance Vector

- The distance vector routing algorithm is sometimes called as Bellman-Ford algorithm
- Every T seconds each router sends its table to its neighbor each router then updates its table based on the new information
- Problems include fast response to good news and slow response to bad news. Also too many messages to update

# Distance Vector

- When a node detects a link failure
  - F detects that link to G has failed
  - F sets distance to G to infinity and sends update to A
  - A sets distance to G to infinity since it uses F to reach G
  - A receives periodic update from C with 2-hop path to G
  - A sets distance to G to 3 and sends update to F
  - F decides it can reach G in 4 hops via A



# Distance Vector

- Slightly different circumstances can prevent the network from stabilizing
  - Suppose the link from A to E goes down
  - In the next round of updates, A advertises a distance of infinity to E, but B and C advertise a distance of 2 to E
  - Depending on the exact timing of events, the following might happen
    - Node B, upon hearing that E can be reached in 2 hops from C, concludes that it can reach E in 3 hops and advertises this to A
    - Node A concludes that it can reach E in 4 hops and advertises this to C
    - Node C concludes that it can reach E in 5 hops; and so on.
    - This cycle stops only when the distances reach some number that is large enough to be considered infinite
      - **Count-to-infinity problem**

# Count-to-infinity Problem

- Use some relatively small number as an approximation of infinity
- For example, the maximum number of hops to get across a certain network is never going to be more than 16

- One technique to improve the time to stabilize routing is called *split horizon*
  - When a node sends a routing update to its neighbors, it does not send those routes it learned from each neighbor back to that neighbor
  - For example, if B has the route (E, 2, A) in its table, then it knows it must have learned this route from A, and so whenever B sends a routing update to A, it does not include the route (E, 2) in that update

# Count-to-infinity Problem

- In a stronger version of split horizon, called *split horizon with poison reverse*
  - B actually sends that back route to A, but it puts negative information in the route to ensure that A will not eventually use B to get to E
  - For example, B sends the route (E, ∞) to A

# EXERCISE 46, PAGE 294

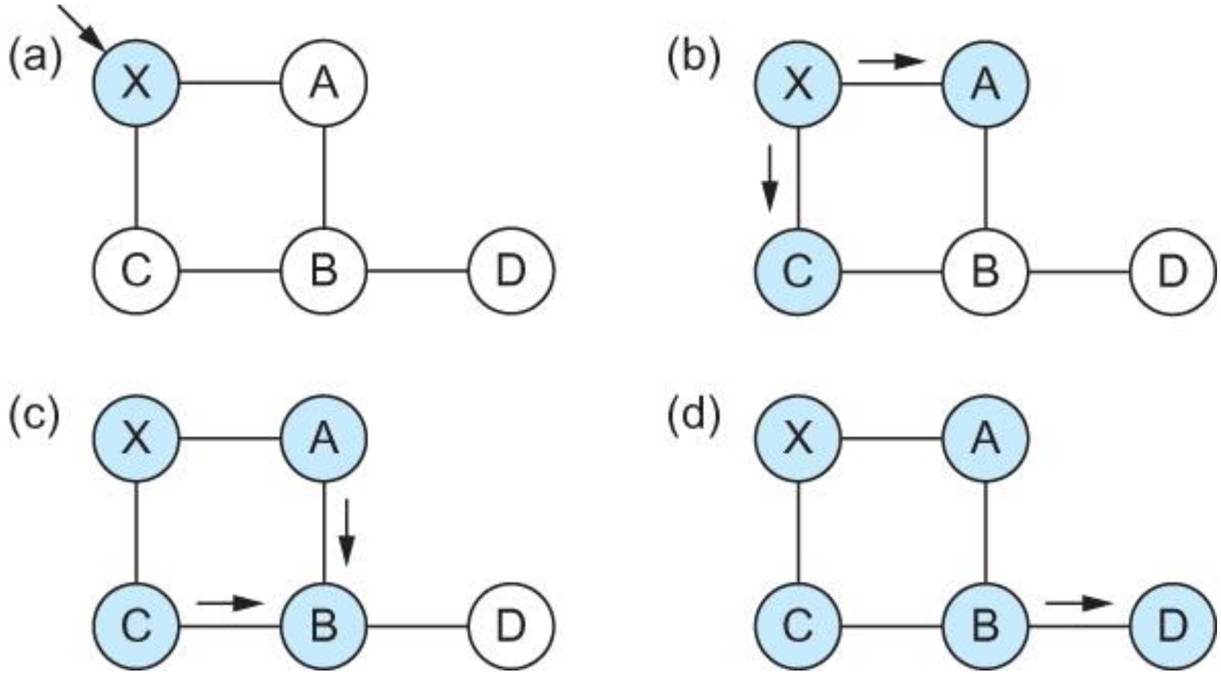
# Link State Routing

Strategy: Send to all nodes (not just neighbors) information about directly connected links (not entire routing table).

- Link State Packet (LSP)
  - id of the node that created the LSP
  - cost of link to each directly connected neighbor
  - sequence number (SEQNO)
  - time-to-live (TTL) for this packet
- Reliable Flooding
  - store most recent LSP from each node
  - forward LSP to all nodes but one that sent it
  - generate new LSP periodically; increment SEQNO
  - start SEQNO at 0 when reboot
  - decrement TTL of each stored LSP; discard when TTL=0

# Link State

## Reliable Flooding




Flooding of link-state packets. (a) LSP arrives at node X; (b) X floods LSP to A and C; (c) A and C flood LSP to B (but not X); (d) flooding is complete

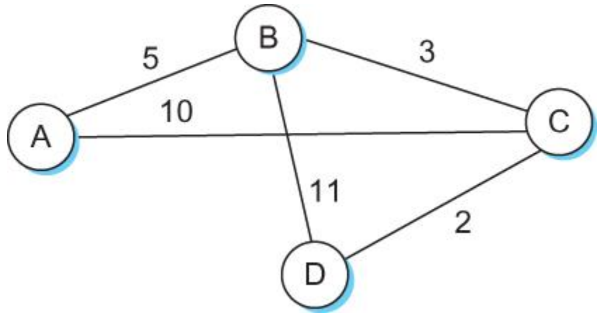
# Shortest Path Routing

- In practice, each switch computes its routing table directly from the LSP's it has collected using a realization of Dijkstra's algorithm called the *forward search algorithm*
- Specifically each switch maintains two lists, known as **Tentative** and **Confirmed**
- Each of these lists contains a set of entries of the form (Destination, Cost, NextHop)

# Shortest Path Routing

- The algorithm
  - Initialize the **Confirmed** list with an entry for myself; this entry has a cost of 0
  - For the node just added to the **Confirmed** list in the previous step, call it node **Next**, select its LSP
  - For each neighbor (Neighbor) of **Next**, calculate the cost (Cost) to reach this Neighbor as the sum of the cost from myself to Next and from Next to Neighbor
    - If Neighbor is currently on neither the **Confirmed** nor the **Tentative** list, then add (Neighbor, Cost, Nexthop) to the **Tentative** list, where Nexthop is the direction I go to reach Next
    - If Neighbor is currently on the **Tentative** list, and the Cost is less than the currently listed cost for the Neighbor, then replace the current entry with (Neighbor, Cost, Nexthop) where Nexthop is the direction I go to reach Next
  - If the **Tentative** list is empty, stop. Otherwise, pick the entry from the **Tentative** list with the lowest cost, move it to the **Confirmed** list, and return to Step 2.

# Shortest Path Routing




| Step | Confirmed | Tentative | Comments |
|---|---|---|---|
| 1 | (D,0,–) | | Since D is the only new member of the confirmed list, look at its LSP. |
| 2 | (D,0,–) | (B,11,B) (C,2,C) | D's LSP says we can reach B through B at cost 11, which is better than anything else on either list, so put it on Tentative list; same for C. |
| 3 | (D,0,–) (C,2,C) | (B,11,B) | Put lowest-cost member of Tentative (C) onto Confirmed list. Next, examine LSP of newly confirmed member (C). |
| 4 | (D,0,–) (C,2,C) | (B,5,C) (A,12,C) | Cost to reach B through C is 5, so replace (B,11,B). C's LSP tells us that we can reach A at cost 12. |
| 5 | (D,0,–) (C,2,C) (B,5,C) | (A,12,C) | Move lowest-cost member of Tentative (B) to Confirmed, then look at its LSP. |
| 6 | (D,0,–) (C,2,C) (B,5,C) | (A,10,C) | Since we can reach A at cost 5 through B, replace the Tentative entry. |
| 7 | (D,0,–) (C,2,C) (B,5,C) (A,10,C) | | Move lowest-cost member of Tentative (A) to Confirmed, and we are all done. |

# Shortest Path Routing

- Dijkstra's Algorithm - Assume non-negative link weights
  - N: set of nodes in the graph
  - l((i, j): the non-negative cost associated with the edge between nodes i, j ∈N and l(i, j) = ∝ if no edge connects i and j
  - Let s ∈N be the starting node which executes the algorithm to find shortest paths to all other nodes in N
  - Two variables used by the algorithm
    - M: set of nodes incorporated so far by the algorithm
    - C(n) : the cost of the path from s to each node n
    - The algorithm

```
M = {s}
For each n in N - {s}
    C(n) = l(s, n)
while ( N ≠ M)
  M = M ∪ {w} such that C(w) is the minimum
                             for all w in (N-M)
  For each n in (N-M)
         C(n) = MIN (C(n), C(w) + l(w, n))
```

# EXERCISE 63, PAGE 300

# Today's Class

# IPv6

# Major Features

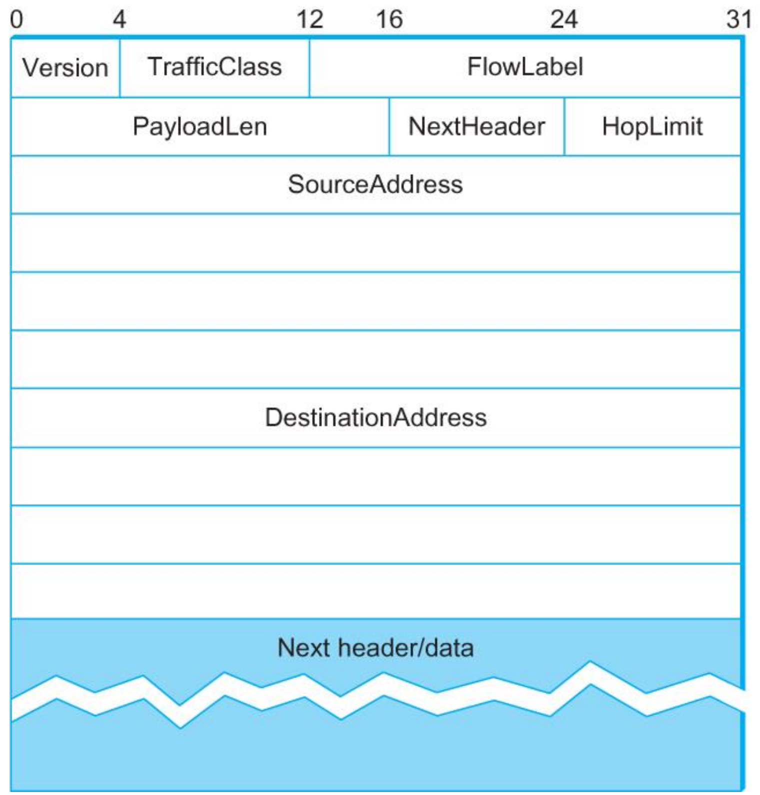
- 128-bit addresses
- Multicast
- Real-time service
- Authentication and security
- Auto-configuration
- End-to-end fragmentation
- Enhanced routing functionality, including support for mobile hosts

# IPv6 Addresses

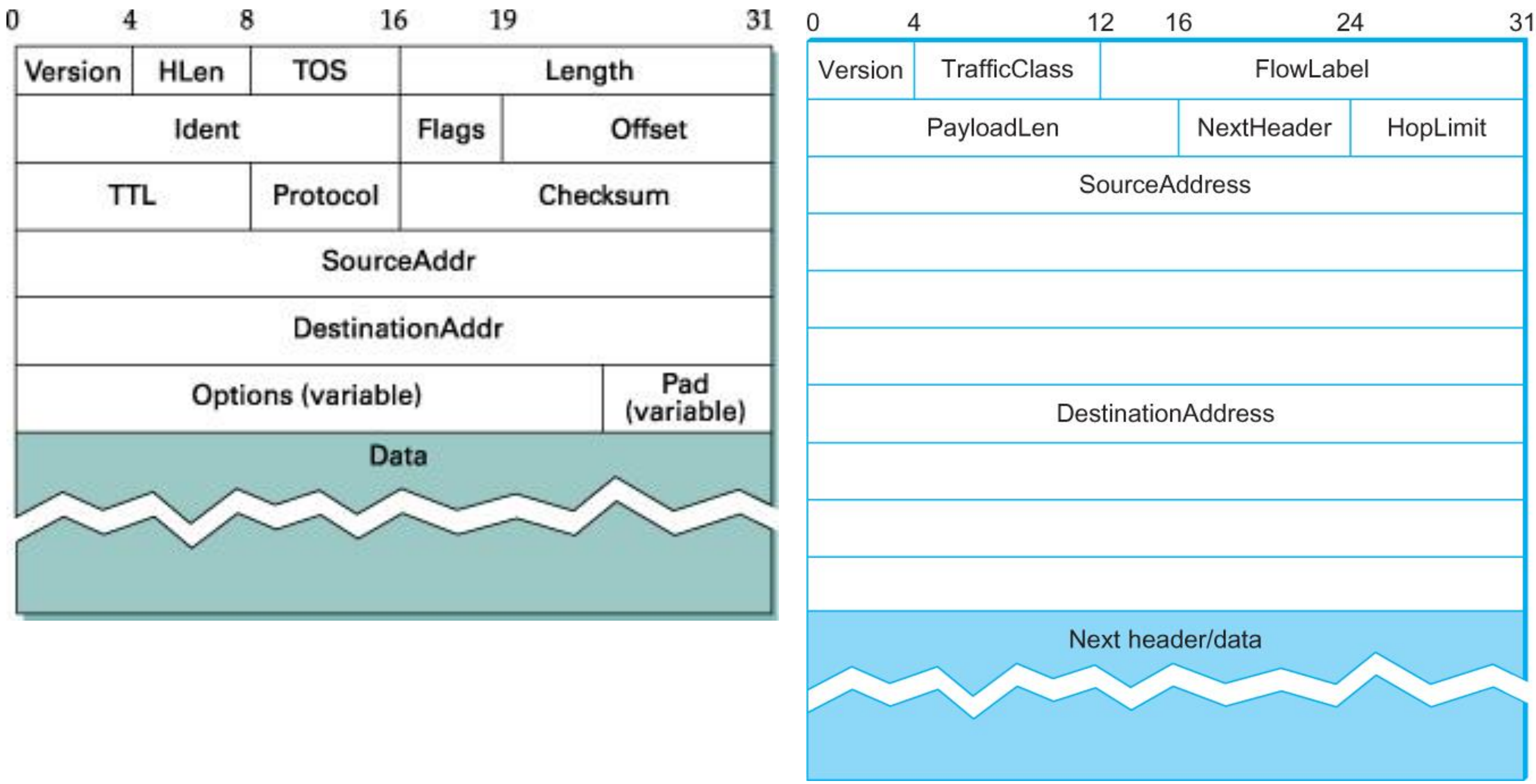
- Classless addressing/routing (similar to CIDR)
- Notation: x:x:x:x:x:x:x:x (x = 16-bit hex number)
  - contiguous 0s are compressed: 47CD::A456:0124
  - IPv6 compatible IPv4 address: ::128.42.1.87
- Address assignment
  - provider-based
  - geographic

# IPv6 Header

- 40-byte “base” header
- Extension headers (fixed order, mostly fixed length)
  - fragmentation
  - source routing
  - authentication and security
  - other options



# IPv4 versus IPv6



# End-to-end Protocols

- Common properties that a transport protocol can be expected to provide
  - Guarantees message delivery
  - Delivers messages in the same order they were sent
  - Delivers at most one copy of each message
  - Supports arbitrarily large messages
  - Supports synchronization between the sender and the receiver
  - Allows the receiver to apply flow control to the sender
  - Supports multiple application processes on each host

# End-to-end Protocols

- Typical limitations of the network on which transport protocol will operate
  - Drop messages
  - Reorder messages
  - Deliver duplicate copies of a given message
  - Limit messages to some finite size
  - Deliver messages after an arbitrarily long delay

# End-to-end Protocols

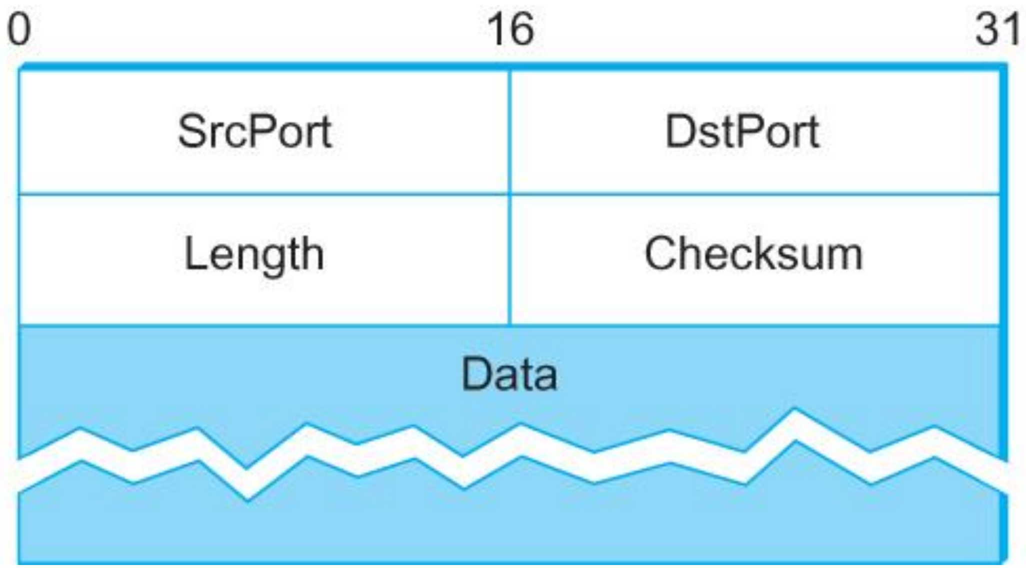
# End-to-end Protocols

- Challenge for Transport Protocols
  - Develop algorithms that turn the less-than-desirable properties of the underlying network into the high level of service required by application programs

# Simple Demultiplexer (UDP)

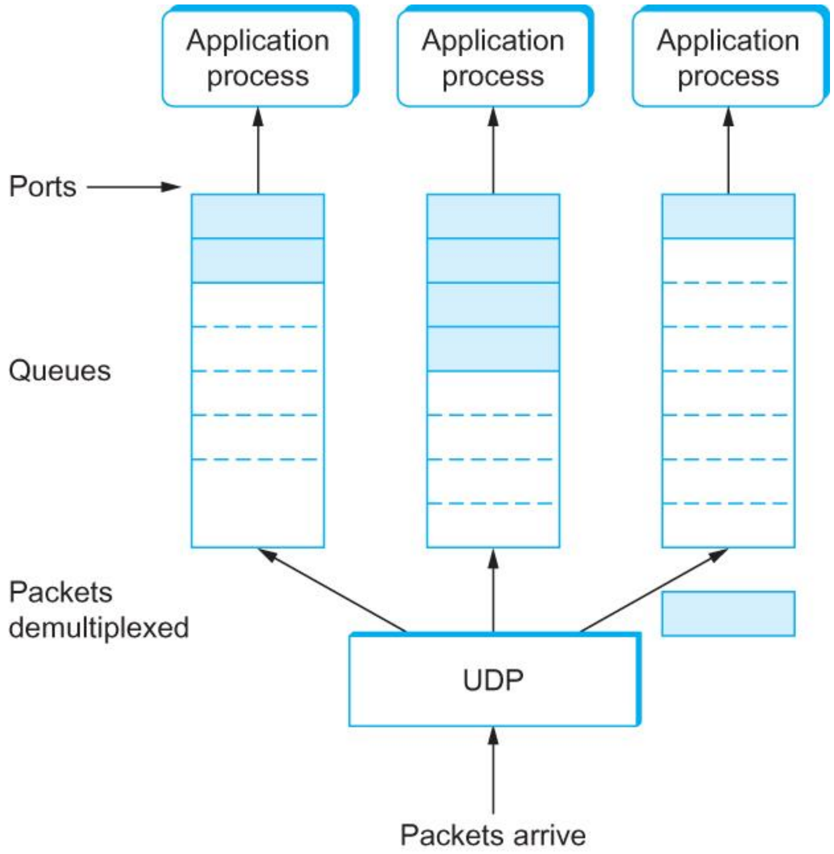
- Extends host-to-host delivery service of the underlying network into a process-to-process communication service
- Adds a level of demultiplexing which allows multiple application processes on each host to share the network

# Simple Demultiplexer (UDP)




Format for UDP header (Note: length and checksum fields should be switched)

# Simple Demultiplexer (UDP)




UDP Message Queue

# Reliable Byte Stream (TCP)

- In contrast to UDP, Transmission Control Protocol (TCP) offers the following services
  - Reliable
  - Connection oriented
  - Byte-stream service

# Flow control VS Congestion control

- Flow control involves preventing senders from overrunning the capacity of the receivers
- Congestion control involves preventing too much data from being injected into the network, thereby causing switches or links to become overloaded

# End-to-end Issues

- At the heart of TCP is the sliding window algorithm (discussed in Chapter 2)
- As TCP runs over the Internet rather than a point-to-point link, the following issues need to be addressed by the sliding window algorithm
  - TCP supports logical connections between processes that are running on two different computers in the Internet
  - TCP connections are likely to have widely different RTT times
  - Packets may get reordered in the Internet

# End-to-end Issues

- TCP needs a mechanism using which each side of a connection will learn what resources the other side is able to apply to the connection
- TCP needs a mechanism using which the sending side will learn the capacity of the network

# TCP Segment

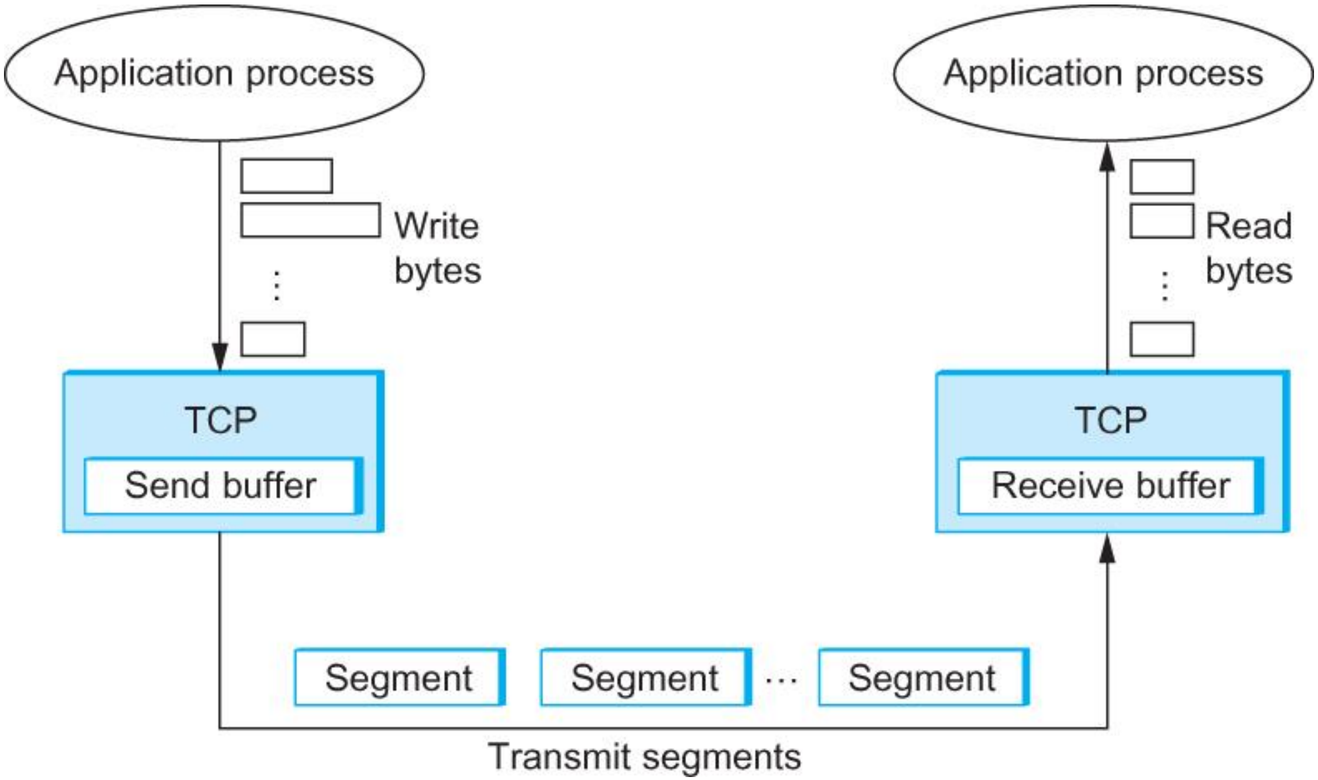
- TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection.

- Although "byte stream" describes the service TCP offers to application processes, TCP does not, itself, transmit individual bytes over the Internet.

# TCP Segment

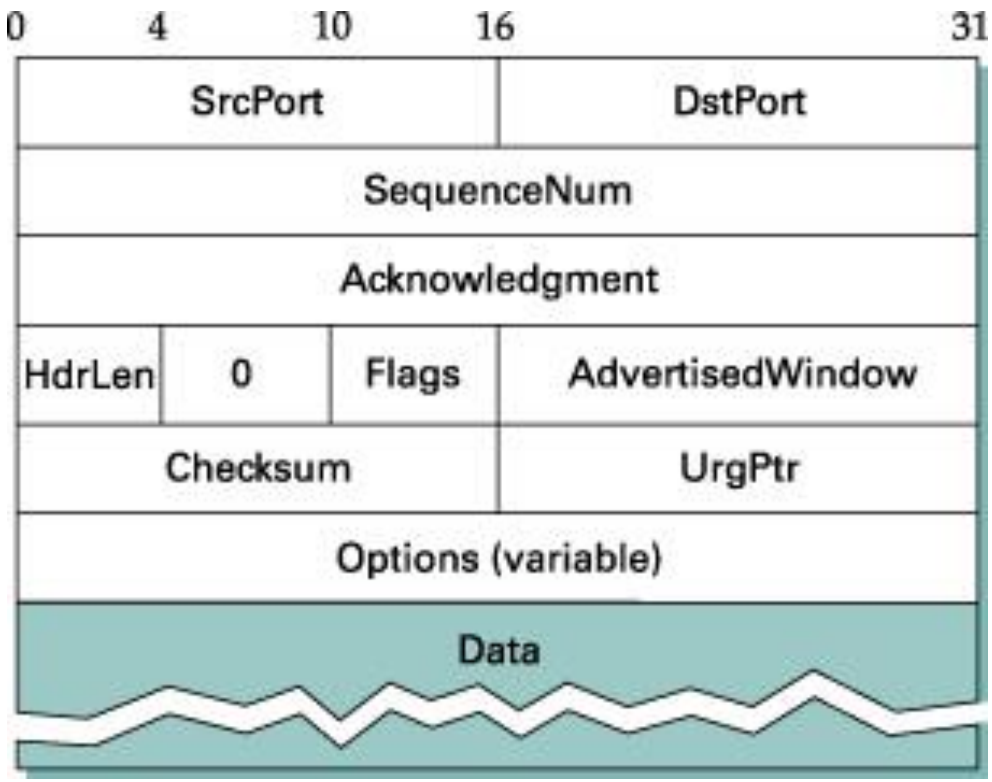
- TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet and then sends this packet to its peer on the destination host.
- TCP on the destination host then empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure.
- The packets exchanged between TCP peers are called *segments*.

# TCP Segment

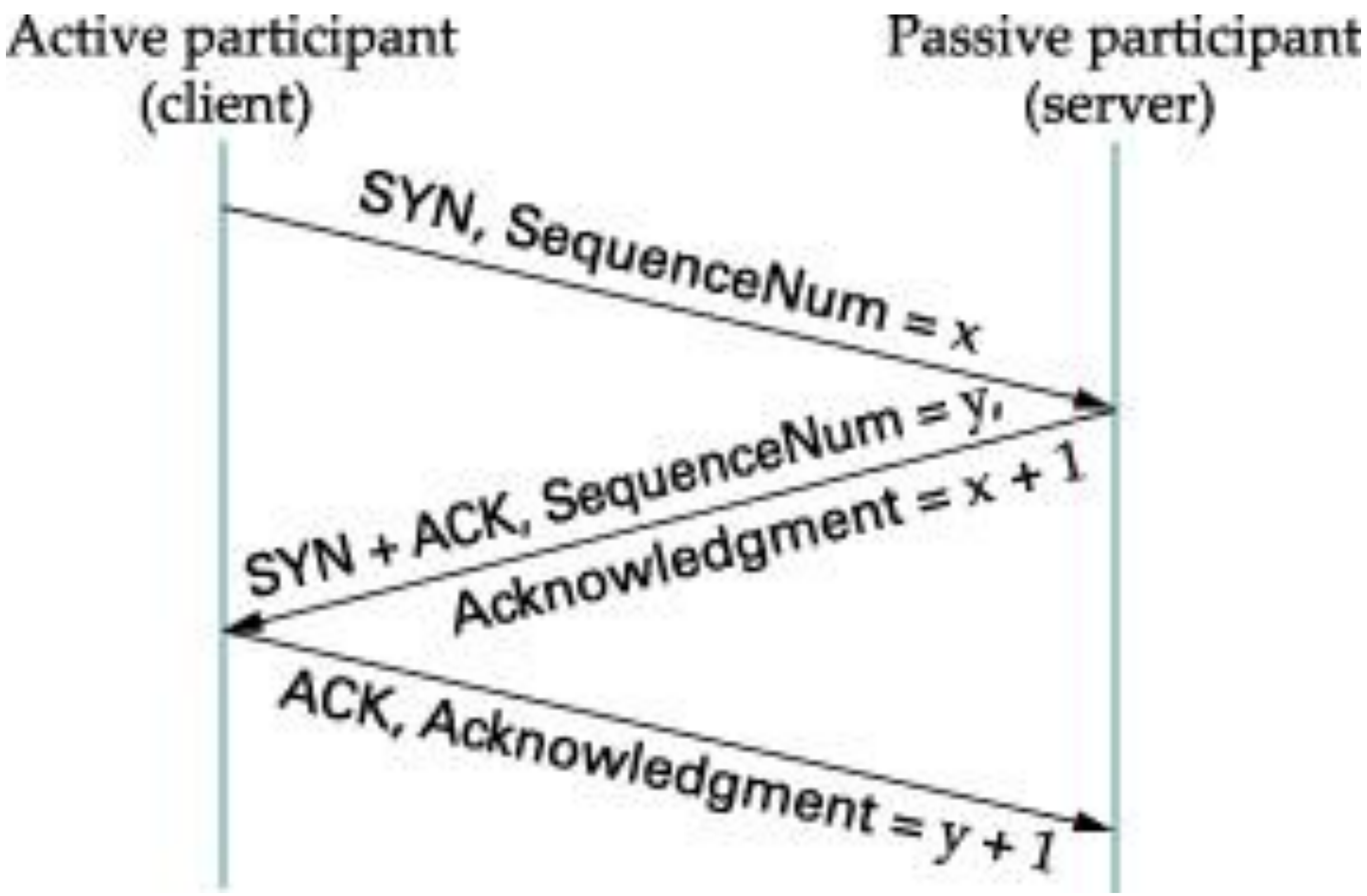



How TCP manages a byte stream.

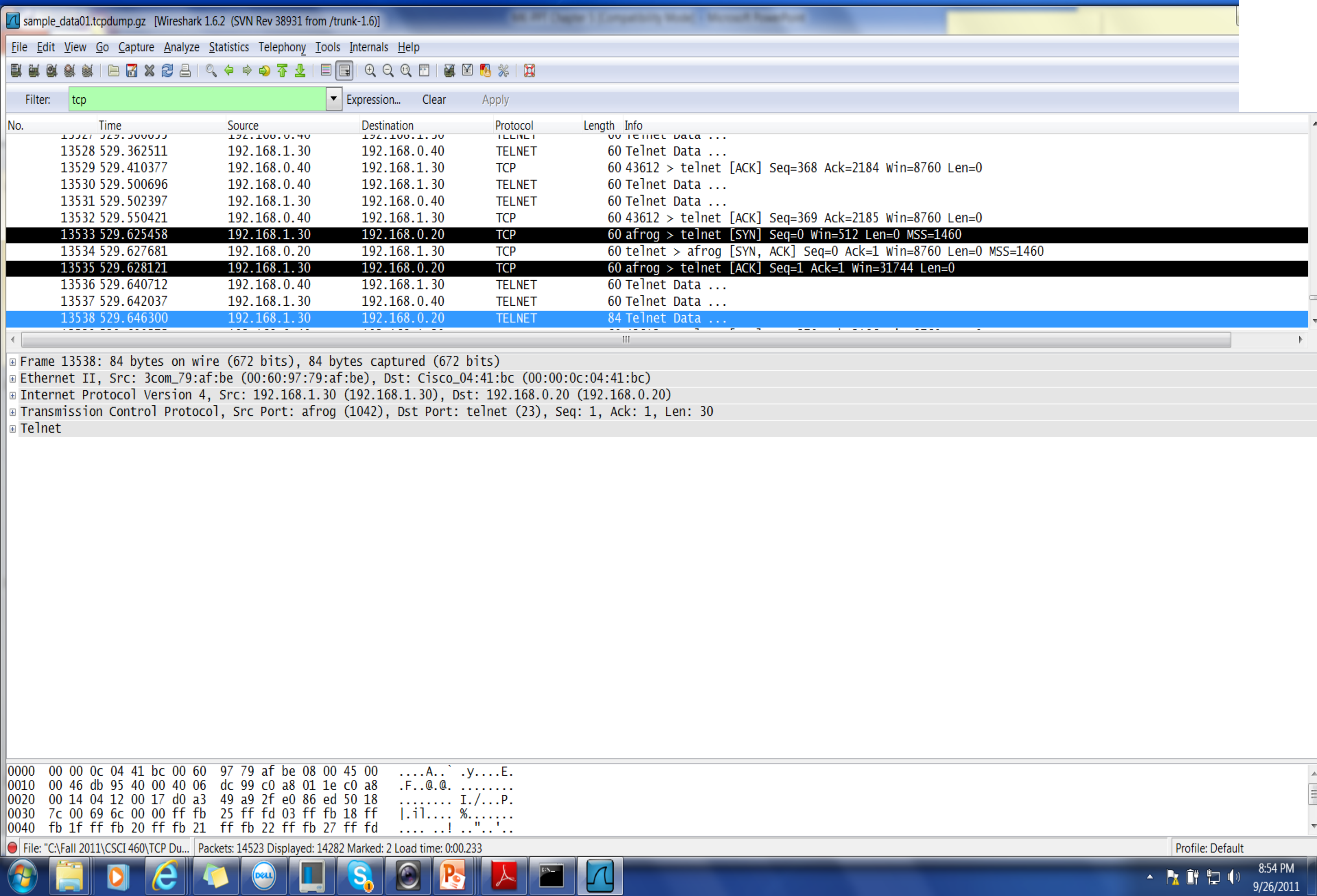
# TCP Packet


0
4
10
16
31
SrcPort
DstPort
SequenceNum
Acknowledgment
HdrLen
0
Flags
AdvertisedWindow
Checksum
UrgPtr
Options (variable)
Data

# Connection Establishment in TCP



# Connection Establishment in TCP



# Connection Tear-down in TCP

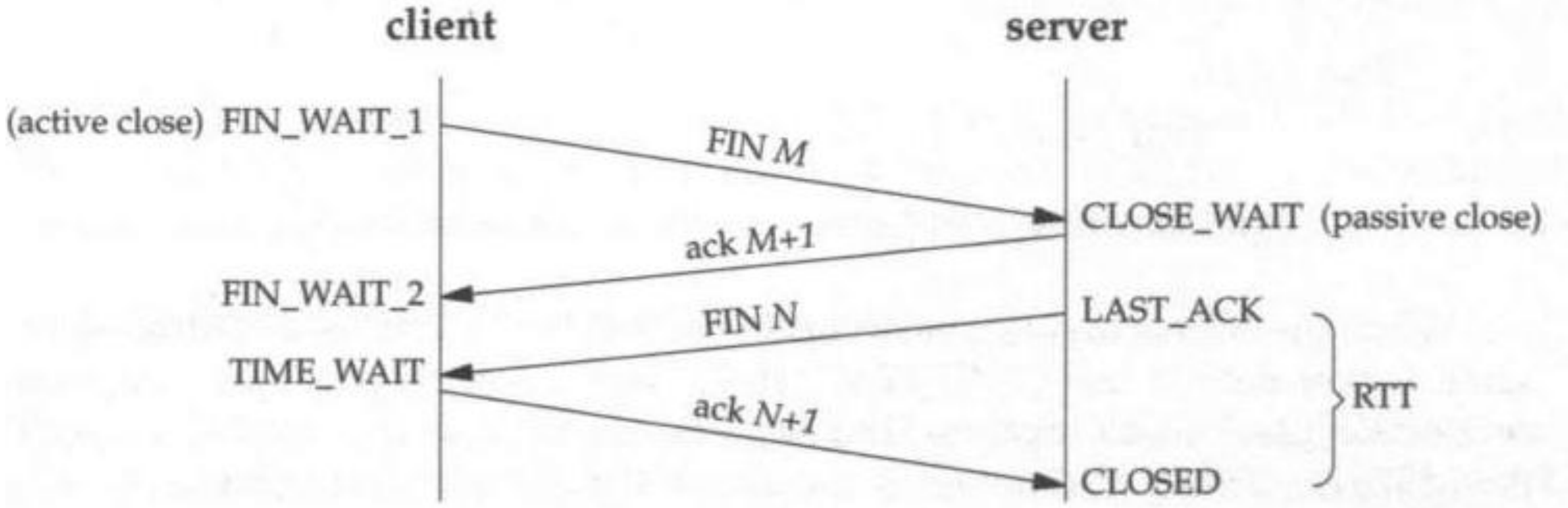
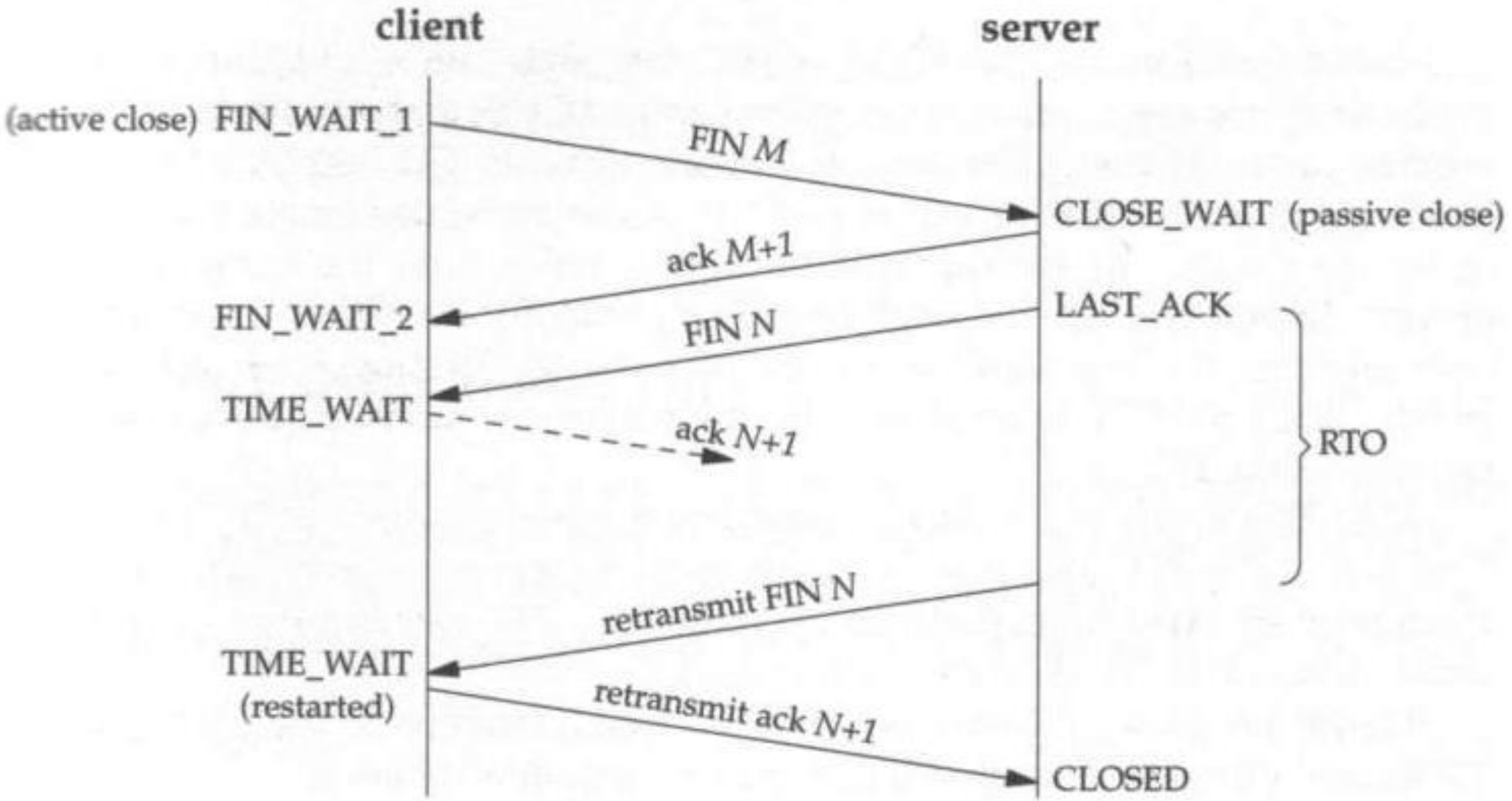
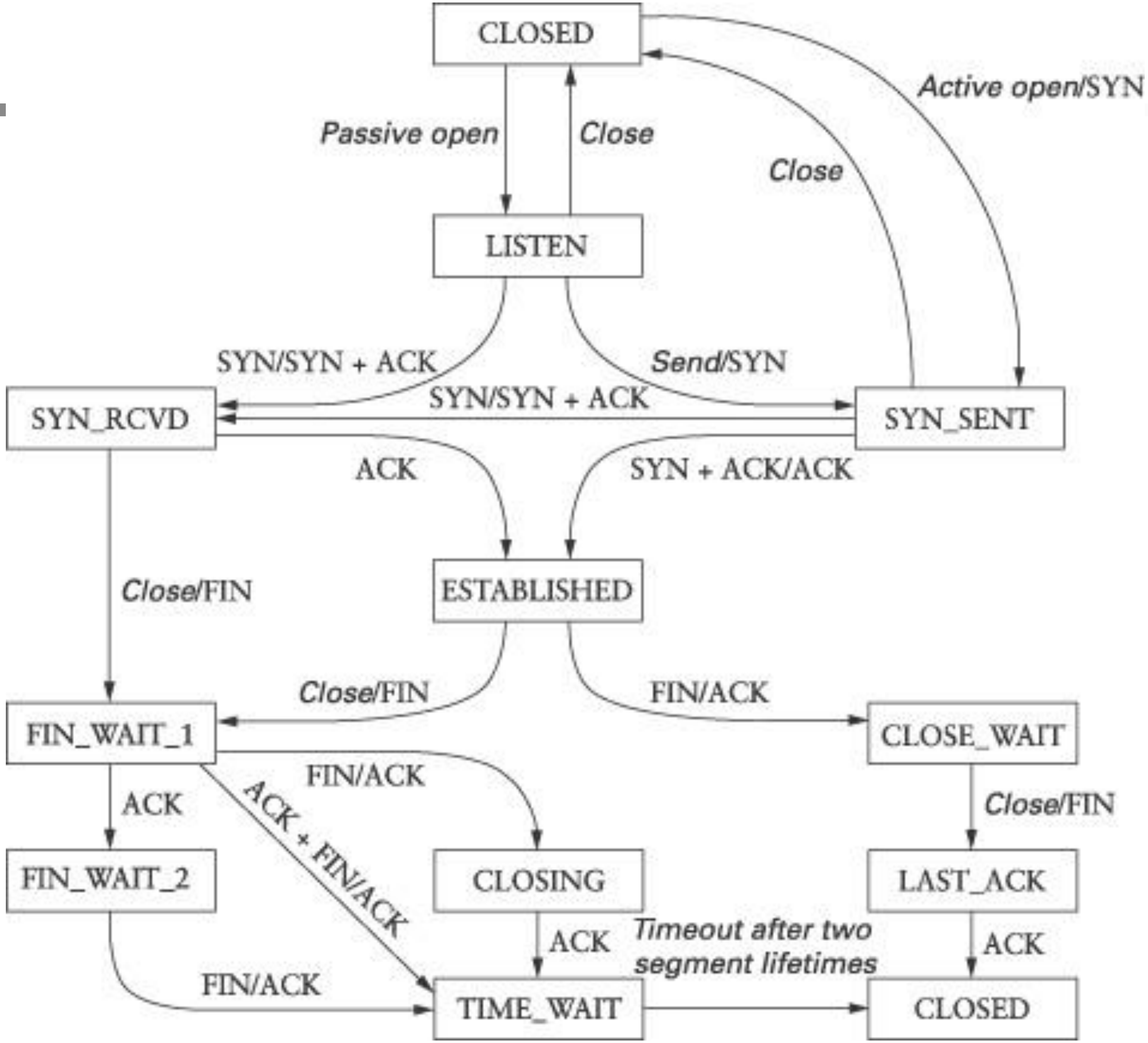



**Figure 4.4** Normal exchange of segments to close a connection.

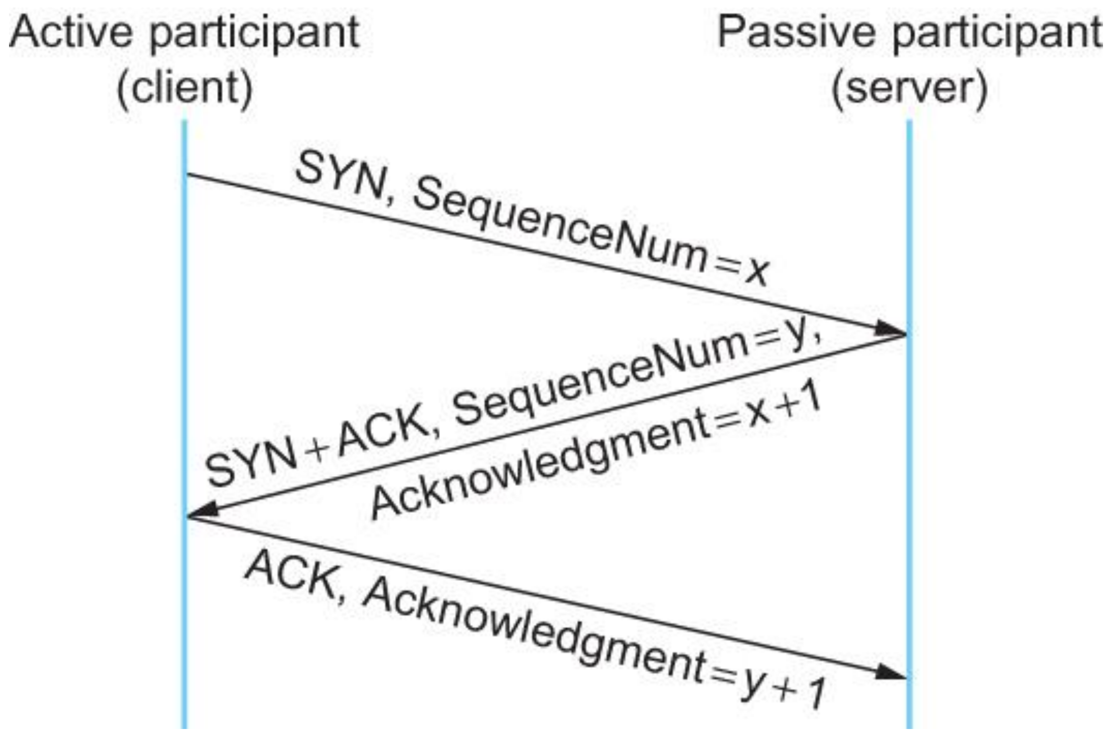
# Connection Tear-down in TCP





CLOSED
Passive open
Close
Active open/SYN
Close
LISTEN
SYN/SYN + ACK
Send/SYN
SYN/SYN + ACK
SYN_RCVD
SYN_SENT
ACK
SYN + ACK/ACK
Close/FIN
ESTABLISHED
Close/FIN
FIN/ACK
FIN_WAIT_1
CLOSE_WAIT
FIN/ACK
ACK
ACK + FIN/ACK
Close/FIN
FIN_WAIT_2
CLOSING
LAST_ACK
ACK
Timeout after two segment lifetimes
ACK
FIN/ACK
TIME_WAIT
CLOSED

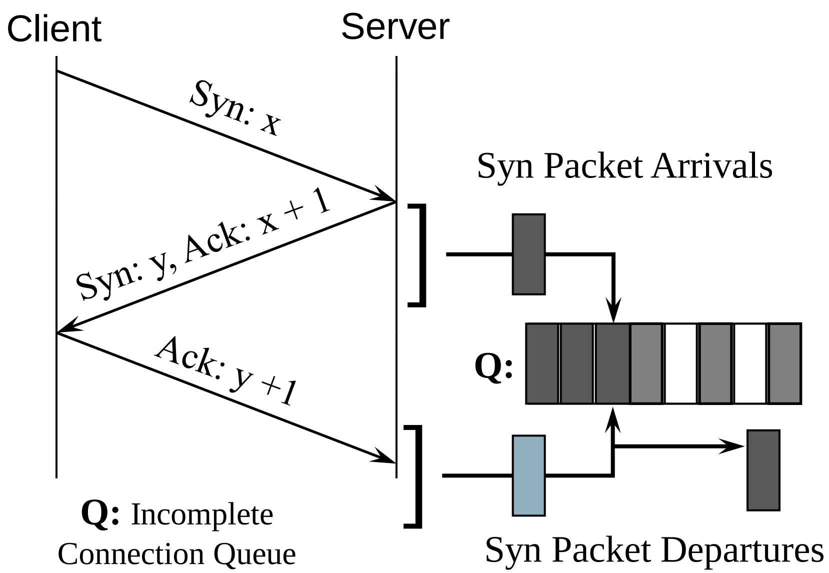
# Connection Establishment in TCP




Timeline for three-way handshake algorithm

# SYN Flooding in TCP

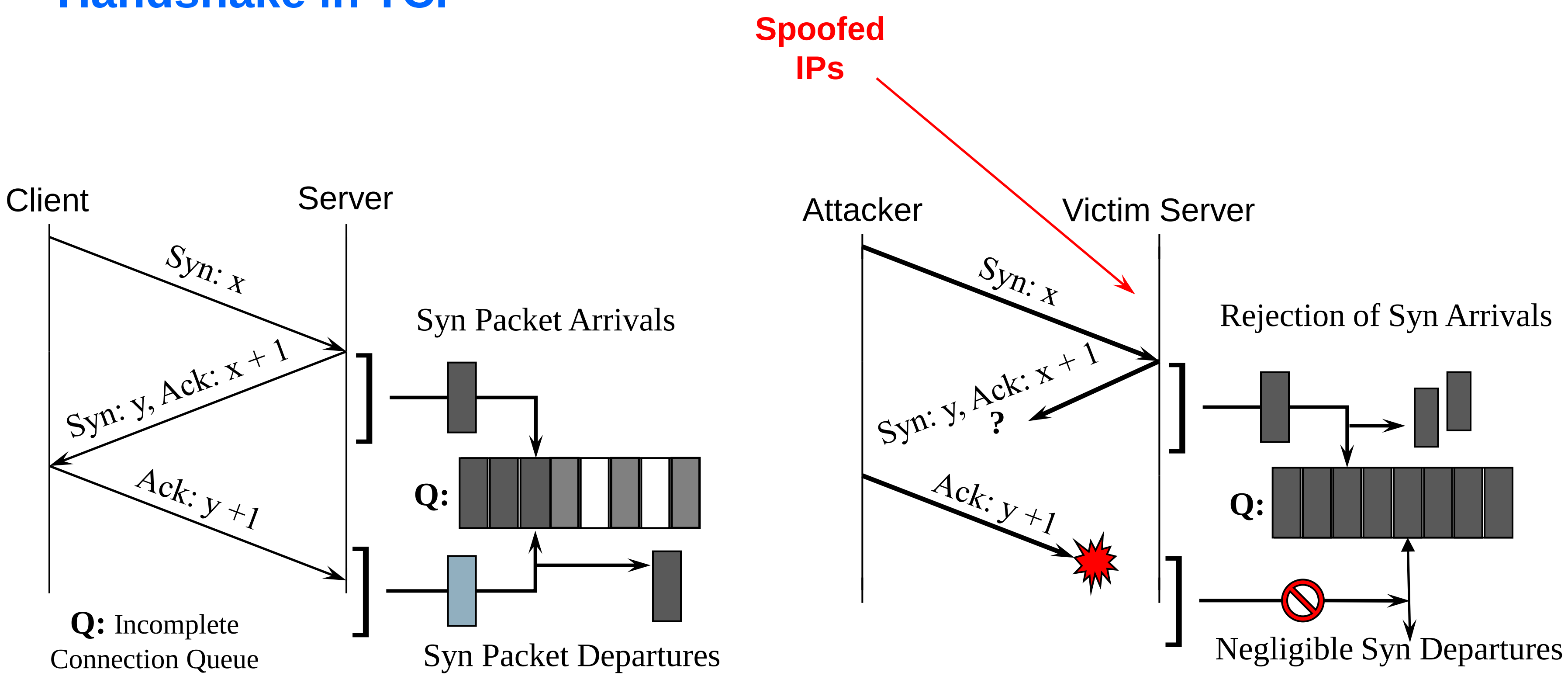
## Normal Three-way Handshake in TCP



# SYN Flooding in TCP

## Normal Three-way Handshake in TCP

## SYN Flooding Attack




**Attack Signature**

# SYN Flooding in TCP

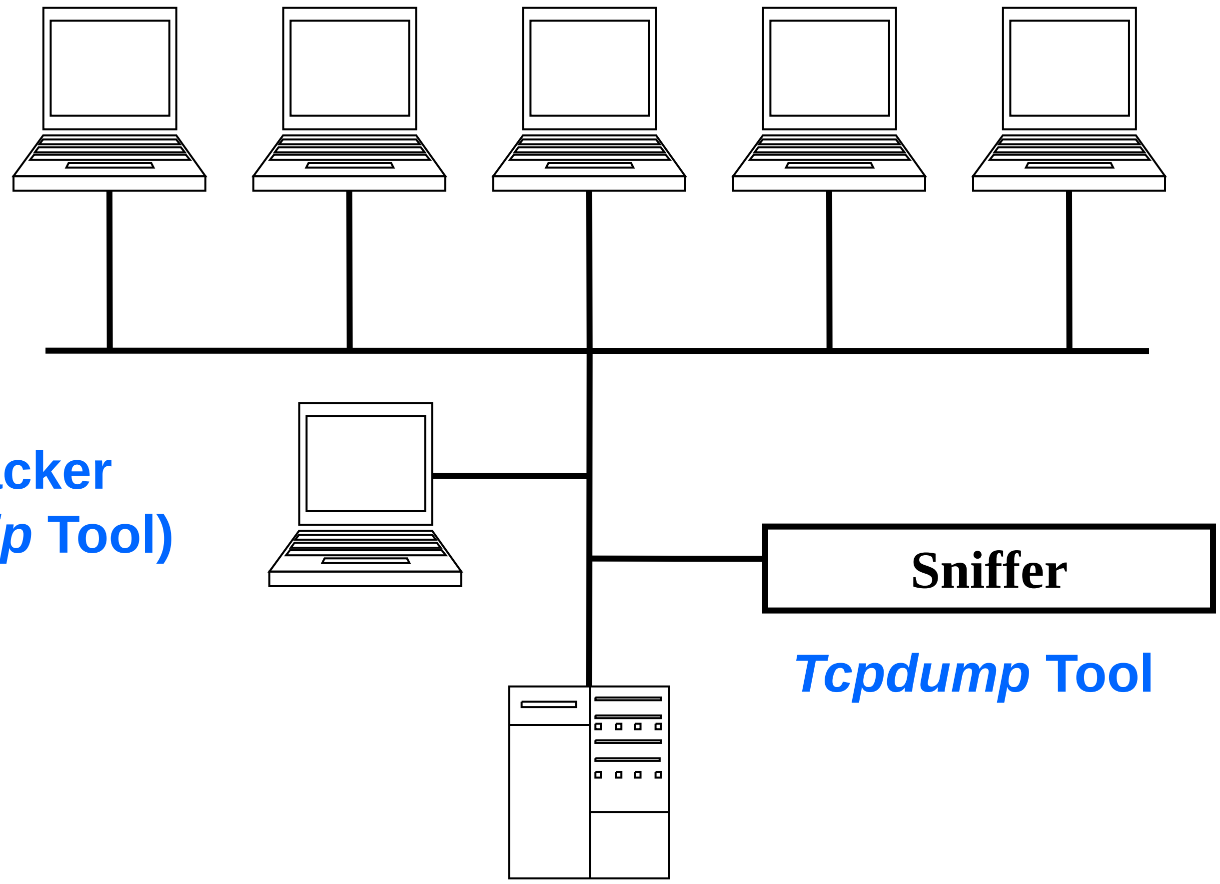
tcpdump.gz [Wireshark 1.6.2 (SVN Rev 38931 from /trunk-1.6)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

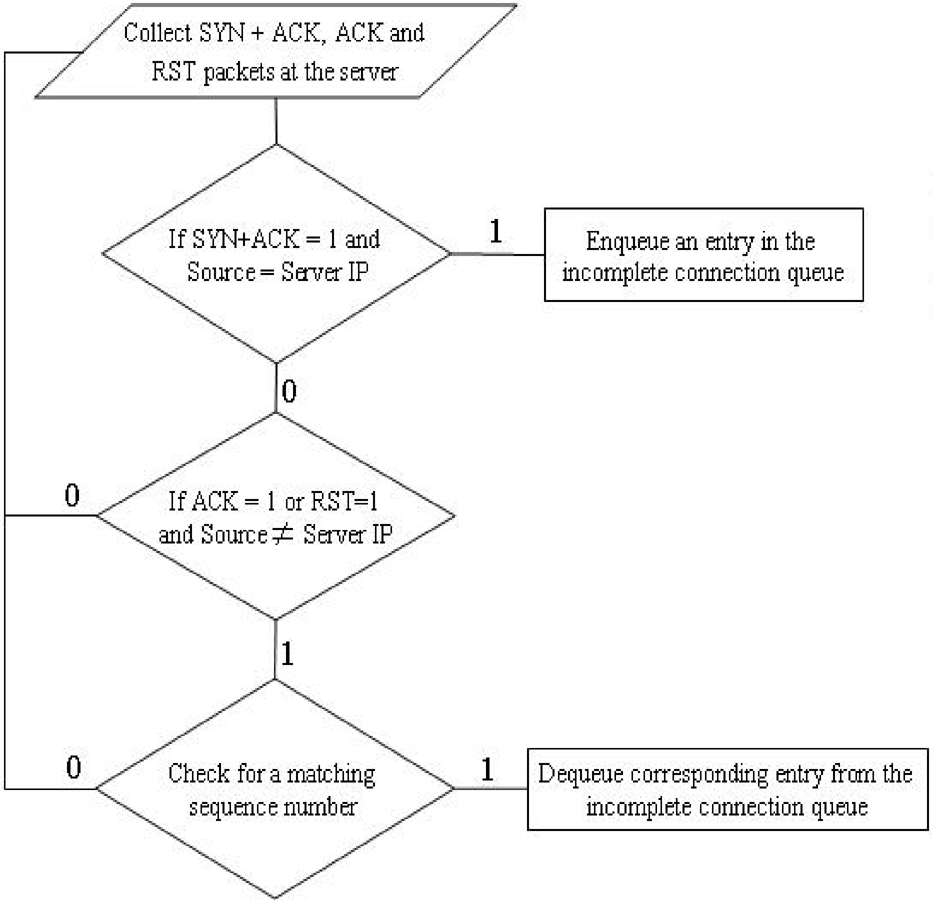
Filter: tcp Expression... Clear Apply

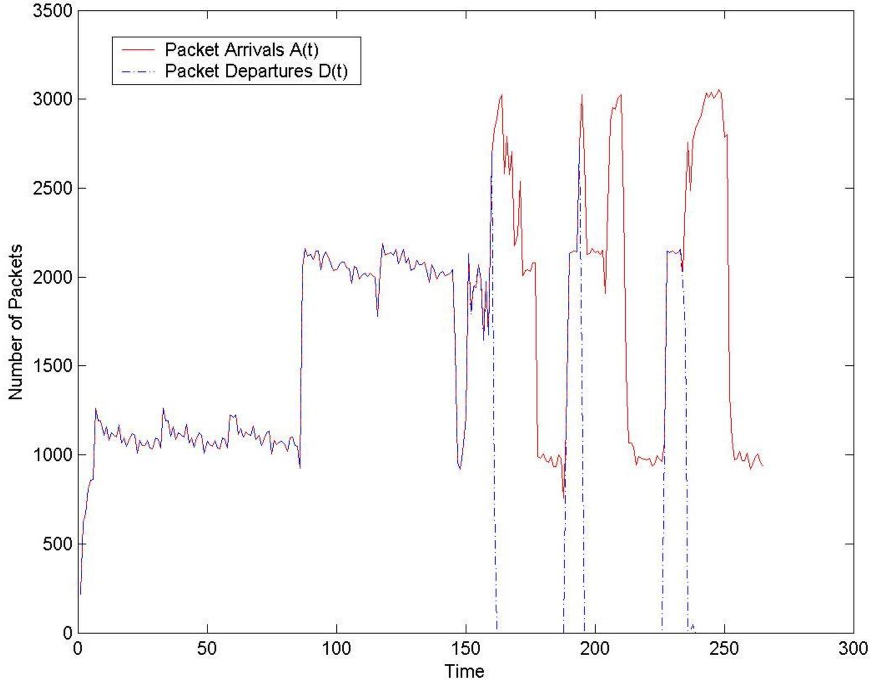
| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 276729 | 11:55:15.067167 | 135.8.60.182 | 172.16.114.207 | TCP | 60 | 15045 > smtp [FIN, ACK] Seq=997 Ack=329 Win=32120 Len=0 |
| 276730 | 11:55:15.067464 | 135.8.60.182 | 172.16.112.194 | TCP | 60 | 15051 > smtp [SYN] Seq=0 Win=512 Len=0 MSS=1460 |
| 276731 | 11:55:15.067588 | 172.16.114.207 | 135.8.60.182 | TCP | 60 | smtp > 15045 [ACK] Seq=329 Ack=998 Win=32735 Len=0 |
| 276732 | 11:55:15.067967 | 172.16.112.194 | 135.8.60.182 | TCP | 60 | smtp > 15051 [SYN, ACK] Seq=0 Ack=1 Win=32736 Len=0 MSS=1460 |
| 276733 | 11:55:15.068120 | 135.8.60.182 | 172.16.112.194 | TCP | 60 | 15051 > smtp [ACK] Seq=1 Ack=1 Win=32120 Len=0 |
| 276734 | 11:55:15.453647 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 18700 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276735 | 11:55:15.473565 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 18956 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276736 | 11:55:15.493560 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 19212 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276737 | 11:55:15.513563 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 19468 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276738 | 11:55:15.533564 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 19724 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276739 | 11:55:15.553575 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 19980 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276740 | 11:55:15.573562 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 20236 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276741 | 11:55:15.593562 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 20492 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276742 | 11:55:15.613564 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 20748 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276743 | 11:55:15.633565 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 21004 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276744 | 11:55:15.653571 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 21260 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276745 | 11:55:15.673570 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 21516 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276746 | 11:55:15.693713 | 1.2.3.4 | 172.16.112.50 | TCP | 60 | 21772 > telnet [SYN] Seq=0 Win=242 Len=0 |
| 276749 | 11:55:15.786412 | 172.16.112.149 | 195.115.218.108 | SMTP | 137 | S: 220 eagle.eyrie.af.mil ESMTP Sendmail 8.8.7/8.8.7; Wed, 3 Jun 1998 11:55:15 -0400 |
| 276750 | 11:55:15.789426 | 195.115.218.108 | 172.16.112.149 | SMTP | 77 | C: EHLO epsilon.pear.com |
| 276751 | 11:55:15.799483 | 172.16.112.149 | 195.115.218.108 | SMTP | 80 | S: 500 Command unrecognized |
| 276752 | 11:55:15.799784 | 195.115.218.108 | 172.16.112.149 | SMTP | 77 | C: HELO epsilon.pear.com |
| 276753 | 11:55:15.800322 | 172.16.112.149 | 195.115.218.108 | SMTP | 99 | S: 250 (epsilon.pear.com) pleased to meet you. |
| 276754 | 11:55:15.800579 | 195.115.218.108 | 172.16.112.149 | SMTP | 91 | C: MAIL From:<anguse@epsilon.pear.com> |
| 276755 | 11:55:15.801080 | 172.16.112.149 | 195.115.218.108 | SMTP | 98 | S: 250 <anguse@epsilon.pear.com>... Sender Ok |
| 276756 | 11:55:15.801375 | 195.115.218.108 | 172.16.112.149 | SMTP | 93 | C: RCPT To:<enriqueg@eagle.eyrie.af.mil> |
| 276757 | 11:55:15.801868 | 172.16.112.149 | 195.115.218.108 | SMTP | 92 | S: 250 <enriqueg@eagle.eyrie.af.mil> OK |
| 276758 | 11:55:15.802130 | 195.115.218.108 | 172.16.112.149 | SMTP | 60 | C: DATA |
| 276759 | 11:55:15.802844 | 172.16.112.149 | 195.115.218.108 | SMTP | 104 | S: 354 Enter mail, end with "." on a line by itself |

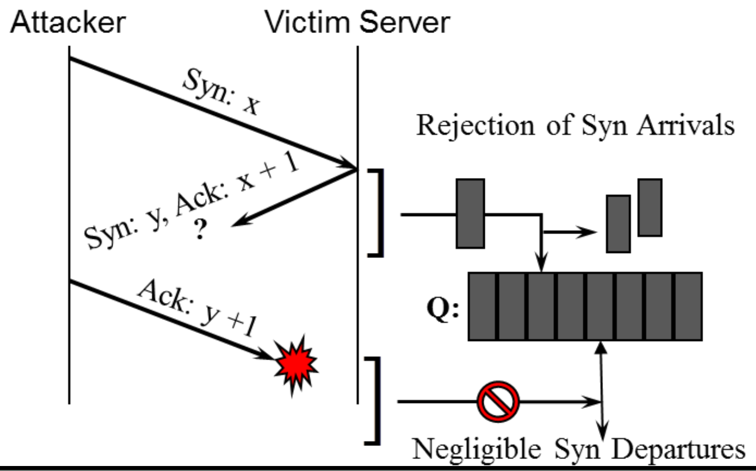
Frame 276735: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
Ethernet II, Src: DellComp_a3:58:23 (00:c0:4f:a3:58:23), Dst: Cisco_04:41:bc (00:00:0c:04:41:bc)
Internet Protocol Version 4, Src: 1.2.3.4 (1.2.3.4), Dst: 172.16.112.50 (172.16.112.50)
Transmission Control Protocol, Src Port: 18956 (18956), Dst Port: telnet (23), Seq: 0, Len: 0


Clients (*SURGE* Tool)
Attacker
(*Sendip* Tool)
Sniffer
*Tcpdump* Tool
*Apache Web Server*

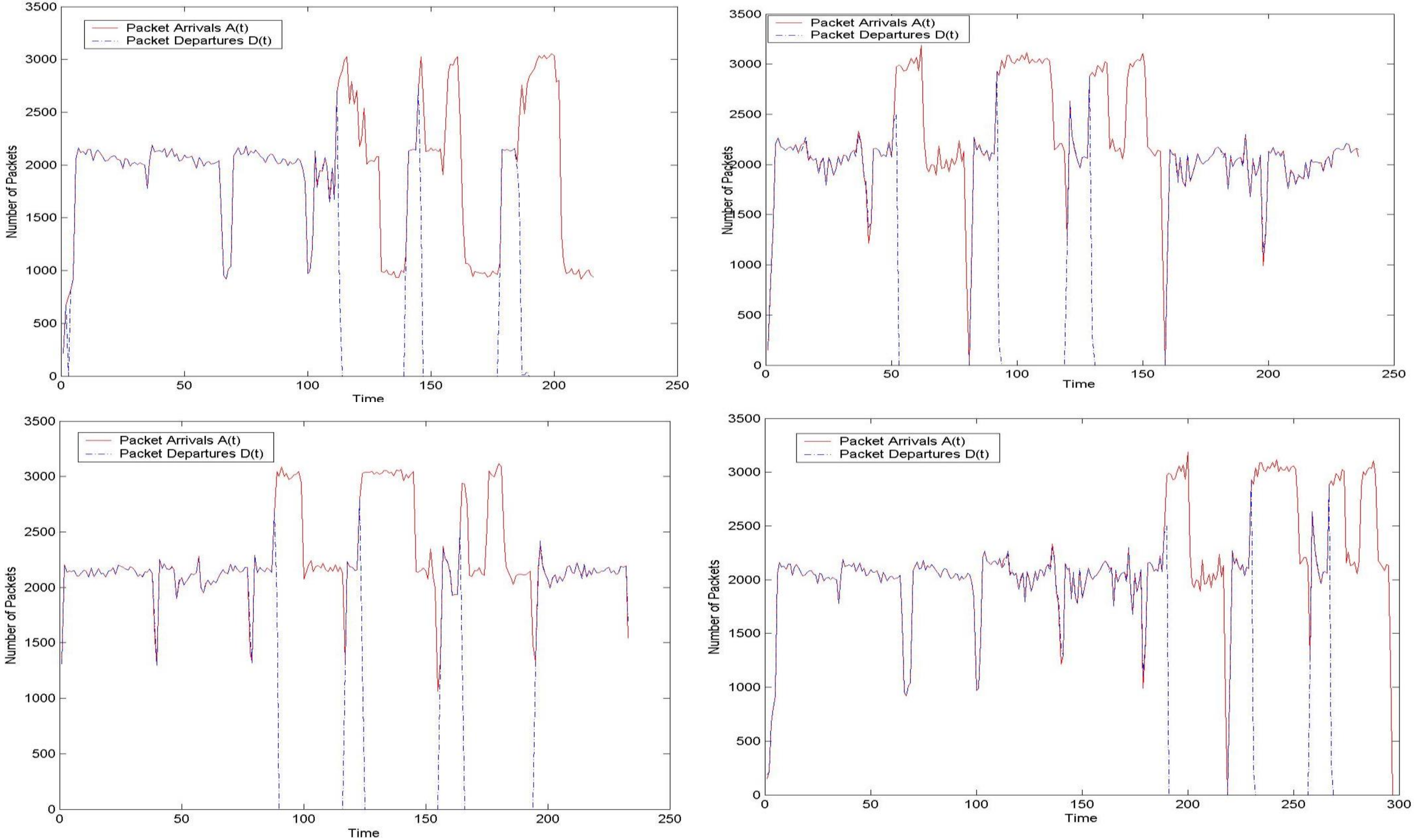
# Packet Classification and Counting



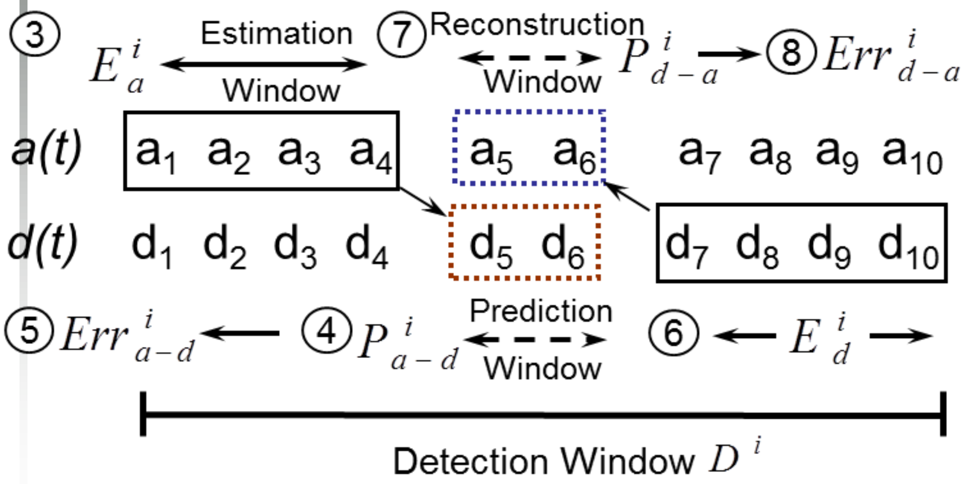



# Packet Counts



# SYNFloodAlert Algorithm




- Function Estimation by polynomial function approximation on $P_3 = a + bx + cx^2 + dx^3$
- Error

$$Err^i = \max\left(f_a^i(t) - A(t)\right) or \max\left(f_d^i(t) - D(t)\right)$$

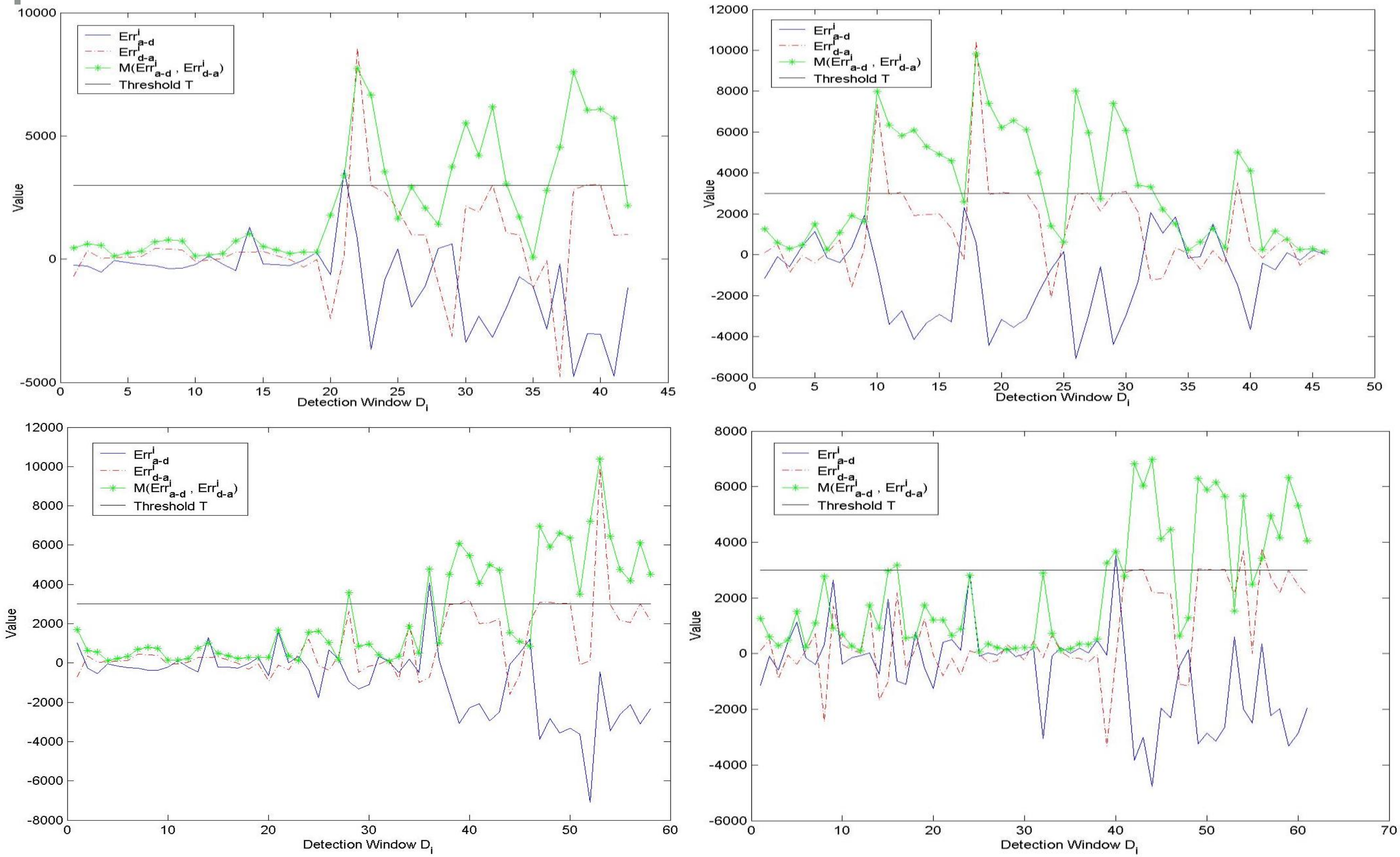
- Tunable Threshold ***T***

/* Set $Err_{a-d}^i$ and $Err_{d-a}^i$. */

/* Set the estimation, prediction, and reconstruction window sizes for Syn arrivals and departures. */

1. Estimate function $f_a^i$ using Syn arrivals in the Arrival Estimation Window.
2. Predict Syn departures using $f_a^i$ for the departures in the Prediction Window $P_{a-d}^i$. Compute error $Err_{a-d}^i$.
3. Estimate function $f_d^i$ for Syn departures in the Departure Estimation Window.
4. Reconstruct previous SYN arrivals using $f_d^i$ for all arrivals in the Reconstruction Window Compute error $P_{d-a}^i$.
5. If ***D***( $Err_{a-d}^i$ - $Err_{d-a}^i$ ) > ***T***, Flag an attack.

# SYNFloodAlert Algorithm



**THE END**

C→S :SIN (ISNc)

S →C :SYN (ISNs),ACK (ISNc)

C→S :ACK (ISNs)

C→S :data

and/or

S→C:data

***X→S :SYN (ISNx),SRC =T***

***S→T :SYN (ISNs),ACK (ISNx)***

***X→S :ACK (ISNs),SRC =T***

***X→S :ACK (ISNs),SRC =T, nasty-data***

**Explain the sequence number prediction attack on TCP (see 'Security Problems in the TCP/IP Protocol Suite' paper). Give detailed steps showing how an attacker might predict the sequence numbers of a server. Assume the following–1) the server increments the sequence number (SN) every millisecond by 1; 2) at the start (boot) time, t = 0 and SN = 0; 3) the time window is 1ms, implying at t = 1, SN = 1, at t = 2, SN = 2, at t = 3, SN = 3, and so on ( because the server increments SN every millisecond); and 4) the round-trip time between the attacker's computer and the server is exactly 1 second. Feel free to make reasonable assumptions to answer this question. State your assumptions.**

# Attack on RIP