




The Transport Layer



1

Introduction

- ❑ Together with the network layer, the transport layer is the heart of the protocol hierarchy.
 - ❑ The network layer provides end-to-end packet delivery using datagrams or virtual circuits.
 - ❑ The transport layer builds on the network layer to provide data transport from a process on a source machine to a process on a destination machine with a desired level of reliability that is independent of the physical networks currently in use.
 - ❑ It provides the abstractions that applications need to use the network.
 - ❑ Without the transport layer, the whole concept of layered protocols would make little sense.
- 

2

Services Provided to the Upper Layers

- ❑ The ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective data transmission service to its users, normally processes in the application layer.
- ❑ To achieve this, the transport layer makes use of the services provided by the network layer.
- ❑ The software and/or hardware within the transport layer that does the work is called the transport entity.
- ❑ The transport entity can be located in the operating system kernel, in a library package bound into network applications, in a separate user process, or even on the network interface card.

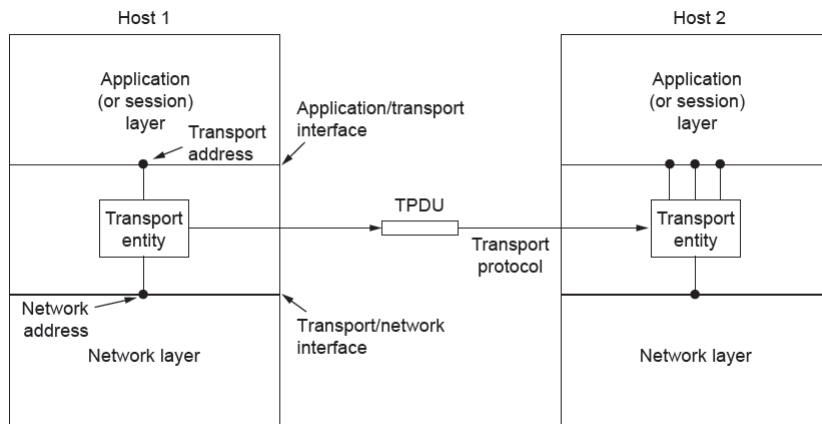
4

Services Provided to the Upper Layers

- ❑ The first two options are most common on the Internet.
- ❑ The (logical) relationship of the network, transport, and application layers is illustrated in Fig. on next slide.

5

Services Provided to the Upper Layers



The network, transport, and application layers

6

Services Provided to the Upper Layers

- ❑ Just as there are two types of network service, connection-oriented and connectionless, there are also two types of transport service.
- ❑ The connection-oriented transport service is similar to the connection-oriented network service in many ways.
- ❑ In both cases, connections have three phases: establishment, data transfer, and release.
- ❑ Addressing and flow control are also similar in both layers.
- ❑ Furthermore, the connectionless transport service is also very similar to the connectionless network service.
- ❑ However, note that it can be difficult to provide a connectionless transport service on top of a connection-oriented network service, since it is inefficient to set up a connection to send a single packet and then tear it down immediately afterwards.

7

Services Provided to the Upper Layers

- ❑ The obvious question is this: if the transport layer service is so similar to the network layer service, why are there two distinct layers? Why is one layer not adequate?
- ❑ The answer is subtle, but crucial. The transport code runs entirely on the users' machines, but the network layer mostly runs on the routers, which are operated by the carrier (at least for a wide area network).
- ❑ What happens if the network layer offers inadequate service?
- ❑ What if it frequently loses packets?
- ❑ What happens if routers crash from time to time?

8

Services Provided to the Upper Layers

- ❑ Problems occur, that's what. The users have no real control over the network layer, so they cannot solve the problem of poor service by using better routers or putting more error handling in the data link layer because they don't own the routers.
- ❑ The only possibility is to put on top of the network layer another layer that improves the quality of the service.
- ❑ If, in a connectionless network, packets are lost or mangled, the transport entity can detect the problem and compensate for it by using retransmissions.
- ❑ If, in a connection-oriented network, a transport entity is informed halfway through a long transmission that its network connection has been abruptly terminated, with no indication of what has happened to the data currently in transit, it can set up a new network connection to the remote transport entity.

9

Services Provided to the Upper Layers

- ❑ Using this new network connection, it can send a query to its peer asking which data arrived and which did not, and knowing where it was, pick up from where it left off.
- ❑ In essence, **the existence of the transport layer makes it possible for the transport service to be more reliable than the underlying network.**
- ❑ Furthermore, the transport primitives can be implemented as **calls to library procedures to make them independent** of the network primitives.
- ❑ The network service calls may vary considerably from one network to another (e.g., calls based on a connectionless Ethernet may be quite different from calls on a connection-oriented WiMAX network).

10

Services Provided to the Upper Layers

- ❑ Hiding the network service behind a set of transport service primitives ensures that changing the network merely requires replacing one set of library procedures with another one that does the same thing with a different underlying service.
- ❑ Because of the transport layer, application programmers can write code according to a standard set of primitives and have these programs work on a wide variety of networks, without having to worry about dealing with different network interfaces and levels of reliability.
- ❑ **If all real networks were flawless and all had the same service primitives and were guaranteed never, ever to change, the transport layer might not be needed.**

11

Services Provided to the Upper Layers

- ❑ For this reason, many people have made a qualitative distinction between layers 1 through 4 on the one hand and layer(s) above 4 on the other.
- ❑ The bottom four layers can be seen as the transport service provider, whereas the upper layer(s) are the transport service user.
- ❑ This distinction of provider versus user has a considerable impact on the design of the layers and puts the transport layer in a key position.
- ❑ Since it forms the major boundary between the provider and user of the reliable data transmission service. It is the level that applications see.

12

Transport Service Primitives

- ❑ To allow users to access the transport service, the transport layer must provide some operations to application programs, that is, a transport service interface.
- ❑ Each transport service has its own interface.
- ❑ The transport service is similar to the network service, but there are also some important differences.
- ❑ The main difference is that the network service is intended to model the service offered by real networks, warts and all.
- ❑ Real networks can lose packets, so the network service is generally unreliable.
- ❑ The connection-oriented transport service, in contrast, is reliable.
- ❑ Of course, real networks are not error-free, but that is precisely the purpose of the transport layer—to provide a reliable service on top of an unreliable network.

13

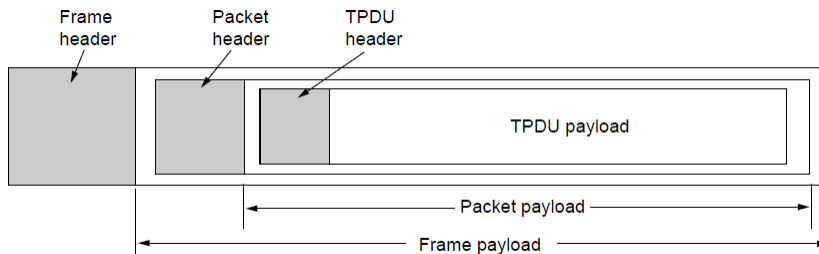
Transport Service Primitives

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

The primitives for a simple transport service

14

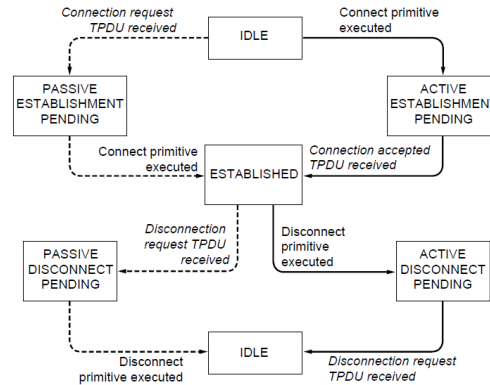
Transport Service Primitives



Nesting of TPDUs, packets, and frames.

15

Transport Service Primitives



A state diagram for a simple connection management scheme. Transitions labeled in *italics* are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.

16

Berkeley Sockets

- ❑ Sockets were first released as part of the Berkeley UNIX 4.2BSD software distribution in 1983.
- ❑ The primitives are now widely used for Internet programming on many operating systems, especially UNIX-based systems, and there is a socket-style API for Windows called "winsock."

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

The socket primitives for TCP

17

Example of Socket Programming: An Internet File Server (1)

```
/* This page contains a client program that can request a file from the server program
 * on the next page. The server responds by sending the whole file.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345          /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096             /* block transfer size */

int main(int argc, char **argv)
{
    int c, s, bytes;
    char buf[BUF_SIZE];            /* buffer for incoming file */
    struct hostent *h;             /* info about server */
    struct sockaddr_in channel;    /* holds IP address */

    . . .
```

Client code using sockets

18

Example of Socket Programming: An Internet File Server (2)

```
. . .

if (argc != 3) fatal("Usage: client server-name file-name");
h = gethostbyname(argv[1]);          /* look up host's IP address */
if (!h) fatal("gethostbyname failed");

s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (s < 0) fatal("socket");
memset(&channel, 0, sizeof(channel));
channel.sin_family = AF_INET;
memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
channel.sin_port = htons(SERVER_PORT);

c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
if (c < 0) fatal("connect failed");

. . .
```

Client code using sockets

19

Example of Socket Programming: An Internet File Server (3)

...

```
c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
if (c < 0) fatal("connect failed");

/* Connection is now established. Send file name including 0 byte at end. */
write(s, argv[2], strlen(argv[2])+1);

/* Go get the file and write it to standard output. */
while (1) {
    bytes = read(s, buf, BUF_SIZE);    /* read from socket */
    if (bytes <= 0) exit(0);           /* check for end of file */
    write(1, buf, bytes);              /* write to standard output */
}
}

fatal(char *string)
{
    printf("%s\n", string);
    exit(1);
}
```

Client code using sockets

20

Example of Socket Programming: An Internet File Server (4)

```
#include <sys/types.h>                /* This is the server code */
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345             /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096                /* block transfer size */
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{
    int s, b, l, fd, sa, bytes, on = 1;
    char buff[BUF_SIZE];              /* buffer for outgoing file */
    struct sockaddr_in channel;       /* holds IP address */

    ...
```

Server code

21

Example of Socket Programming: An Internet File Server (5)

. . .

```
/* Build address structure to bind to socket. */
memset(&channel, 0, sizeof(channel)); /* zero channel */
channel.sin_family = AF_INET;
channel.sin_addr.s_addr = htonl(INADDR_ANY);
channel.sin_port = htons(SERVER_PORT);

/* Passive open. Wait for connection. */
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); /* create socket */
if (s < 0) fatal("socket failed");
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));

b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
if (b < 0) fatal("bind failed");

l = listen(s, QUEUE_SIZE); /* specify queue size */
if (l < 0) fatal("listen failed");
```

. . .

Server code

22

Example of Socket Programming: An Internet File Server (6)

. . .

```
/* Socket is now set up and bound. Wait for connection and process it. */
while (1) {
    sa = accept(s, 0, 0); /* block for connection request */
    if (sa < 0) fatal("accept failed");

    read(sa, buf, BUF_SIZE); /* read file name from socket */

    /* Get and return the file. */
    fd = open(buf, O_RDONLY); /* open the file to be sent back */
    if (fd < 0) fatal("open failed");

    while (1) {
        bytes = read(fd, buf, BUF_SIZE); /* read from file */
        if (bytes <= 0) break; /* check for end of file */
        write(sa, buf, bytes); /* write bytes to socket */
    }
    close(fd); /* close file */
    close(sa); /* close connection */
}
```

Server code

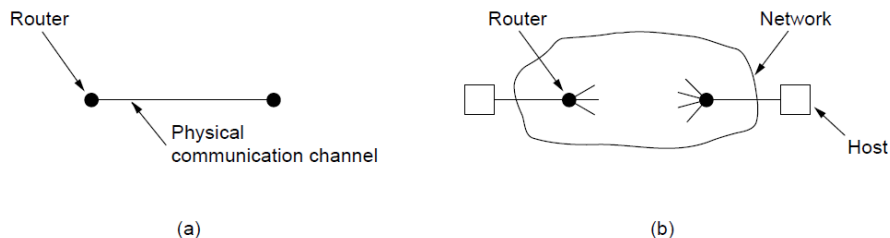
23

Elements of Transport Protocols

- ❑ The transport service is implemented by a transport protocol used between the two transport entities.
- ❑ In some ways, transport protocols resemble the data link protocols. Both have to deal with error control, sequencing, and flow control, among other issues.
- ❑ However, significant differences between the two also exist.
- ❑ These differences are due to major dissimilarities between the environments in which the two protocols operate, as shown in Fig. on next slide.

24

Elements of Transport Protocols



- (a) Environment of the data link layer.
- (b) Environment of the transport layer.

25

Elements of Transport Protocols

- ❑ At the data link layer, two routers communicate directly via a physical channel, whether wired or wireless, whereas at the transport layer, this physical channel is replaced by the entire network.
- ❑ This difference has many important implications for the protocols.



26

Elements of Transport Protocols

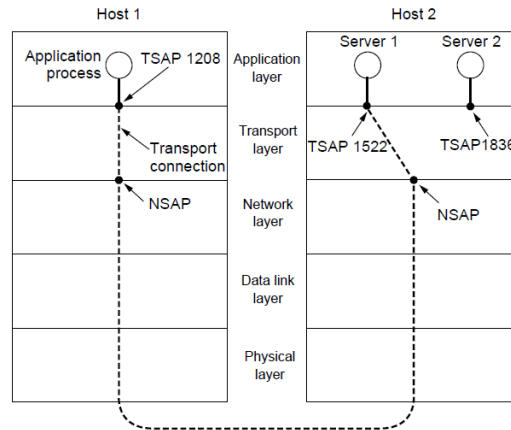
- ❑ Addressing
- ❑ Connection establishment
- ❑ Connection release
- ❑ Error control and flow control
- ❑ Multiplexing
- ❑ Crash recovery



27

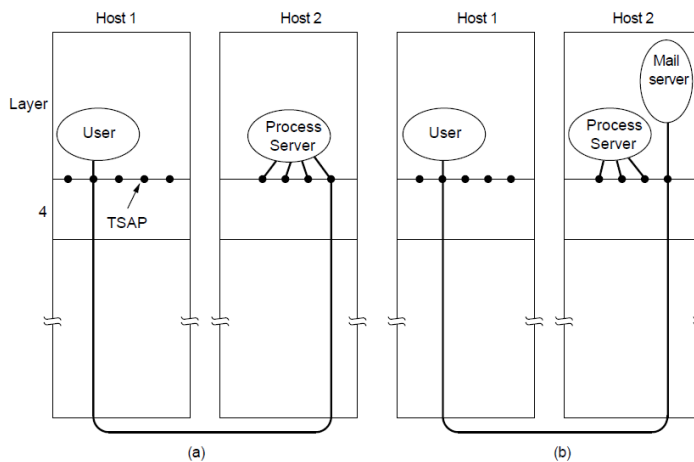
Addressing

TSAPs, NSAPs, and transport connections



28

Addressing



How a user process in host 1 establishes a connection with a mail server in host 2 via a process server.

29

Connection Establishment

- ❑ Establishing a connection sounds easy, but it is actually surprisingly tricky.
- ❑ At first glance, it would seem sufficient for one transport entity to just send a CONNECTION REQUEST segment to the destination and wait for a CONNECTION ACCEPTED reply.
- ❑ The problem occurs when the network can lose, delay, corrupt, and duplicate packets.
- ❑ This behavior causes serious complications.
- ❑ Imagine a network that is so congested that acknowledgements hardly ever get back in time and each packet times out and is retransmitted two or three times.

30

Connection Establishment

- ❑ Suppose that the network uses datagrams inside and that every packet follows a different route.
- ❑ Some of the packets might get stuck in a traffic jam inside the network and take a long time to arrive.
- ❑ That is, they may be delayed in the network and pop out much later, when the sender thought that they had been lost.
- ❑ Example: Problem of establishing a connection with Bank Server.
- ❑ The crux of the problem is that the delayed duplicates are thought to be new packets.
- ❑ We cannot prevent packets from being duplicated and delayed. But if and when this happens, the packets must be rejected as duplicates and not processed as fresh packets.

31

Connection Establishment

- ❑ The problem can be attacked in various ways, none of them very satisfactory.
- ❑ One way is to use throwaway transport addresses.
 - ❑ In this approach, each time a transport address is needed, a new one is generated.
 - ❑ When a connection is released, the address is discarded and never used again.
 - ❑ Delayed duplicate packets then never find their way to a transport process and can do no damage.
 - ❑ However, this approach makes it more difficult to connect with a process in the first place.
- ❑ Another possibility is to give each connection a unique identifier chosen by the initiating party and put in each segment, including the one requesting the connection.

32

Connection Establishment

- ❑ After each connection is released, each transport entity can update a table listing obsolete connections as (peer transport entity, connection identifier) pairs.
- ❑ Whenever a connection request comes in, it can be checked against the table to see if it belongs to a previously released connection.
- ❑ Unfortunately, this scheme has a basic flaw: it requires each transport entity to maintain a certain amount of history information indefinitely.
- ❑ This history must persist at both the source and destination machines. Otherwise, if a machine crashes and loses its memory, it will no longer know which connection identifiers have already been used by its peers.

33

Connection Establishment

- ❑ Instead, we need to take a different tack to simplify the problem.
- ❑ Rather than allowing packets to live forever within the network, we devise a mechanism to kill off aged packets that are still hobbling about.
- ❑ With this restriction, the problem becomes somewhat more manageable.
- ❑ Packet lifetime can be restricted to a known maximum using one (or more) of the following techniques:
 - ❑ Restricted network design.
 - ❑ Putting a hop counter in each packet.
 - ❑ Timestamping each packet.

34

Connection Establishment

- ❑ In practice, we will need to guarantee not only that a packet is dead, but also that all acknowledgements to it are dead, too.
- ❑ So, we will now introduce a period T , which is some small multiple of the true maximum packet lifetime.
- ❑ The maximum packet lifetime is a conservative constant for a network; for the Internet, it is somewhat arbitrarily taken to be 120 seconds.
- ❑ If we wait a time T secs after a packet has been sent, we can be sure that all traces of it are now gone and that neither it nor its acknowledgements will suddenly appear out of the blue to complicate matters.

35

Connection Establishment

- ❑ With packet lifetimes bounded, it is possible to devise a practical and foolproof way to reject delayed duplicate segments.
- ❑ The method described below is due to Tomlinson (1975), as refined by Sunshine and Dalal (1978).
- ❑ Variants of it are widely used in practice, including in TCP.
- ❑ The heart of the method is for the source to label segments with sequence numbers that will not be reused within T secs.
- ❑ The period, T , and the rate of packets per second determine the size of the sequence numbers.
- ❑ In this way, only one packet with a given sequence number may be outstanding at any given time.

36

Connection Establishment

- ❑ Duplicates of this packet may still occur, and they must be discarded by the destination.
- ❑ However, it is no longer the case that a delayed duplicate of an old packet may beat a new packet with the same sequence number and be accepted by the destination in its stead.
- ❑ To get around the problem of a machine losing all memory of where it was after a crash, one possibility is to require transport entities to be idle for T secs after a recovery.
- ❑ The idle period will let all old segments die off, so the sender can start again with any sequence number.
- ❑ However, in a complex internetwork, T may be large, so this strategy is unattractive.

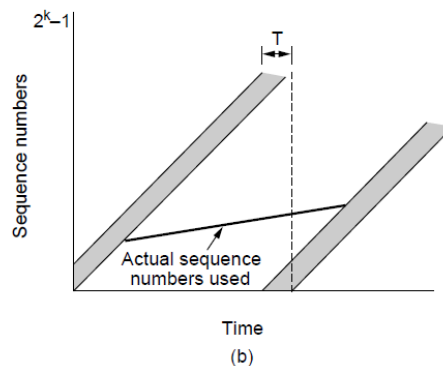
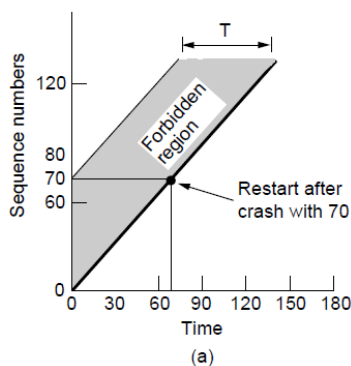
37

Connection Establishment

- ❑ Instead, Tomlinson proposed equipping each host with a time-of-day clock.
- ❑ The clocks at different hosts need not be synchronized. Each clock is assumed to take the form of a binary counter that increments itself at uniform intervals.
- ❑ Furthermore, the number of bits in the counter must equal or exceed the number of bits in the sequence numbers.
- ❑ Last, and most important, the clock is assumed to continue running even if the host goes down.
- ❑ When a connection is set up, the low-order k bits of the clock are used as the k -bit initial sequence number.
- ❑ The sequence space should be so large that by the time sequence numbers wrap around, old segments with the same sequence number are long gone.

38

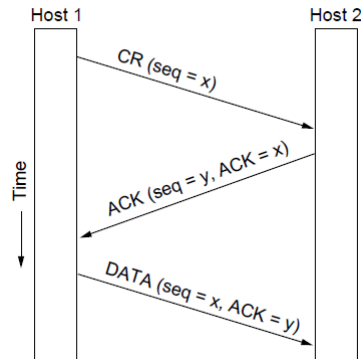
Connection Establishment



- (a) TPDUs may not enter the forbidden region.
- (b) The resynchronization problem.

39

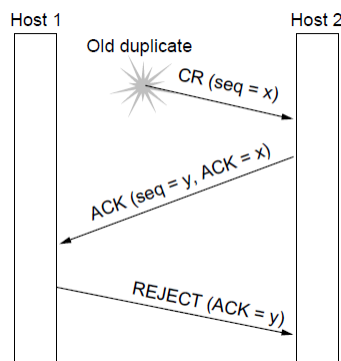
Connection Establishment



Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. Normal operation.

40

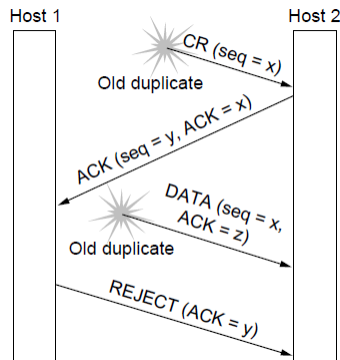
Connection Establishment



Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. Old duplicate CONNECTION REQUEST appearing out of nowhere.

41

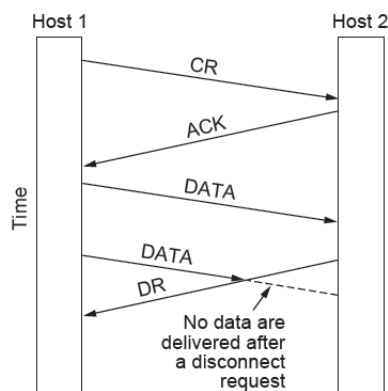
Connection Establishment



Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. Duplicate CONNECTION REQUEST and duplicate ACK

42

Connection Release

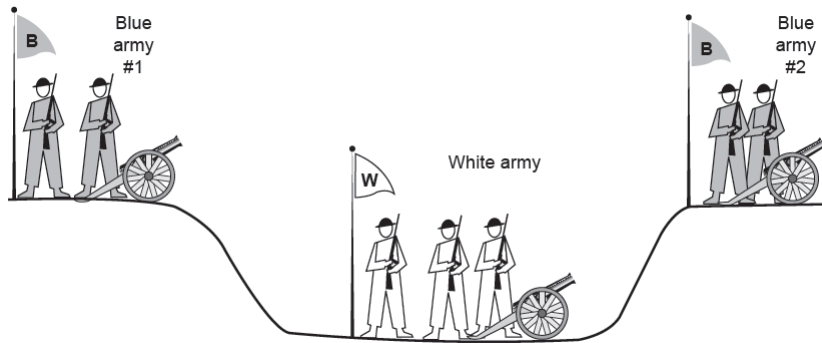


Abrupt disconnection with loss of data

43

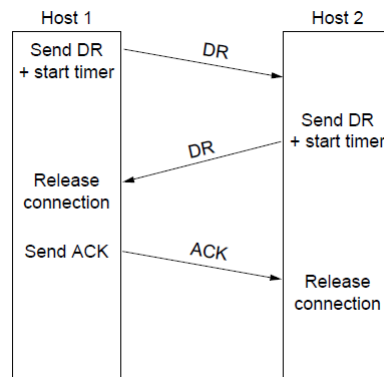
Connection Release

The two-army problem



44

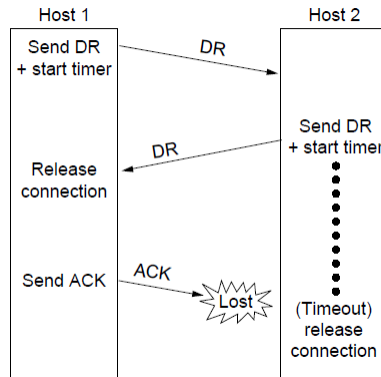
Connection Release



Four protocol scenarios for releasing a connection.
(a) Normal case of three-way handshake

45

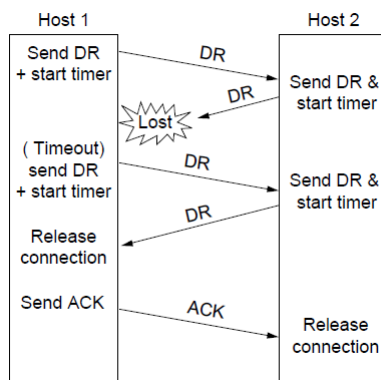
Connection Release



Four protocol scenarios for releasing a connection.
(b) Final ACK lost.

46

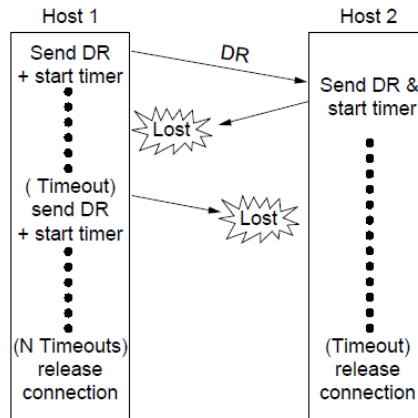
Connection Release



Four protocol scenarios for releasing a connection.
(c) Response lost

47

Connection Release



Four protocol scenarios for releasing a connection.
(d) Response lost and subsequent DRs lost.

48

Error Control and Flow Control

- ❑ Error control is ensuring that the data is delivered with the desired level of reliability, usually that all of the data is delivered without any errors.
- ❑ Flow control is keeping a fast transmitter from overrunning a slow receiver.
- ❑ Both of these issues have come up before, when we studied the data link layer.
- ❑ The solutions that are used at the transport layer are the same mechanisms that we studied in the data link layer.
- ❑ However, there is little duplication between the link and transport layers in practice.
- ❑ Even though the same mechanisms are used, there are differences in function and degree.

49

Error Control and Flow Control

- ❑ For a difference in function, consider error detection.
- ❑ The link layer checksum protects a frame while it crosses a single link.
- ❑ The transport layer checksum protects a segment while it crosses an entire network path.
- ❑ It is an end-to-end check, which is not the same as having a check on every link.
- ❑ Saltzer et al. (1984) describe a situation in which packets were corrupted inside a router.
- ❑ The link layer checksums protected the packets only while they traveled across a link, not while they were inside the router.
- ❑ Thus, packets were delivered incorrectly even though they were correct according to the checks on every link.

50

Error Control and Flow Control

- ❑ This and other examples led Saltzer et al. to articulate the **end-to-end argument**.
- ❑ According to this argument, “The transport layer check that runs end-to-end is essential for correctness, and the link layer checks are not essential but nonetheless valuable for improving performance (since without them a corrupted packet can be sent along the entire path unnecessarily)”.

51

Error Control and Flow Control

- ❑ As a difference in degree, consider retransmissions and the sliding window protocol.
- ❑ Most wireless links, other than satellite links, can have only a single frame outstanding from the sender at a time.
- ❑ That is, the bandwidth-delay product for the link is small enough that not even a whole frame can be stored inside the link.
- ❑ In this case, a small window size is sufficient for good performance.
- ❑ For wired and optical fiber links, such as (switched) Ethernet or ISP backbones, the error-rate is low enough that link-layer retransmissions can be omitted because the end-to-end retransmissions will repair the residual frame loss.

52

Error Control and Flow Control

- ❑ On the other hand, many TCP connections have a bandwidth-delay product that is much larger than a single segment.
- ❑ Consider a connection sending data across the U.S. at 1 Mbps with a round-trip time of 200 msec.
- ❑ Even for this slow connection, 200 Kbit of data will be stored at the sender in the time it takes to send a segment and receive an acknowledgement.
- ❑ For these situations, a large sliding window must be used.
- ❑ Stop-and-wait will cripple performance. In our example it would limit performance to one segment every 200 msec, or 5 segments/sec no matter how fast the network really is.
- ❑ Given that transport protocols generally use larger sliding windows.

53

Error Control and Flow Control

Data buffering for the sliding windows:

- ❑ Since a host may have many connections, each of which is treated separately, it may need a substantial amount of buffering for the sliding windows.
- ❑ The buffers are needed at both the sender and the receiver.
- ❑ Certainly they are needed at the sender to hold all transmitted but as yet unacknowledged segments.
- ❑ They are needed there because these segments may be lost and need to be retransmitted.
- ❑ However, since the sender is buffering, the receiver may or may not dedicate specific buffers to specific connections, as it sees fit.

54

Error Control and Flow Control

- ❑ The receiver may, for example, maintain a single buffer pool shared by all connections.
- ❑ When a segment comes in, an attempt is made to dynamically acquire a new buffer. If one is available, the segment is accepted; otherwise, it is discarded.
- ❑ Since the sender is prepared to retransmit segments lost by the network, no permanent harm is done by having the receiver drop segments, although some resources are wasted.
- ❑ The sender just keeps trying until it gets an acknowledgement.
- ❑ The best trade-off between source buffering and destination buffering depends on the type of traffic carried by the connection.
 - ❑ For low-bandwidth bursty traffic: not to dedicate any buffers, but rather to acquire them dynamically
 - ❑ or file transfer and other high-bandwidth traffic: it is better if the receiver does dedicate a full window of buffers

55

Error Control and Flow Control

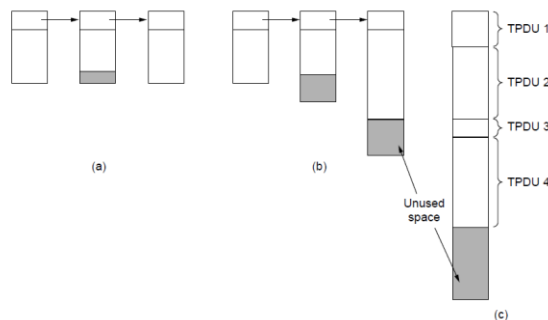
How to organize the buffer pool for the sliding windows?

- ❑ If most segments are nearly the same size, it is natural to organize the buffers as a pool of identically sized buffers, with one segment per buffer, as shown in Fig. (a) of next slide.
- ❑ However, if there is wide variation in segment size, from short requests for Web pages to large packets in peer-to-peer file transfers, a pool of fixed-sized buffers presents problems.
- ❑ Another approach to the buffer size problem is to use variable-sized buffers, as shown in Fig. (b) of next slide.
- ❑ The advantage here is better memory utilization, at the price of more complicated buffer management.

56

Error Control and Flow Control

- ❑ A third possibility is to dedicate a single large circular buffer per connection, as shown in Fig. (c).
- ❑ This system is simple and elegant and does not depend on segment sizes, but makes good use of memory only when the connections are heavily loaded.



(a) Chained fixed-size buffers. (b) Chained variable-sized buffers. (c) One large circular buffer per connection.

57

Error Control and Flow Control

- ❑ As connections are opened and closed and as the traffic pattern changes, the sender and receiver need to dynamically adjust their buffer allocations.
- ❑ Consequently, the transport protocol should allow a sending host to request buffer space at the other end.
- ❑ Buffers could be allocated per connection, or collectively, for all the connections running between the two hosts.
- ❑ Alternatively, the receiver, knowing its buffer situation (but not knowing the offered traffic) could tell the sender “I have reserved X buffers for you.”
- ❑ A reasonably general way to manage dynamic buffer allocation is to decouple the buffering from the acknowledgements, in contrast to the sliding window protocols

59

Error Control and Flow Control

- ❑ Dynamic buffer management means, in effect, a variable-sized window.
- ❑ Initially, the sender requests a certain number of buffers, based on its expected needs.
- ❑ The receiver then grants as many of these as it can afford.
- ❑ Every time the sender transmits a segment, it must decrement its allocation, stopping altogether when the allocation reaches zero.
- ❑ The receiver separately piggybacks both acknowledgements and buffer allocations onto the reverse traffic.
- ❑ TCP uses this scheme, carrying buffer allocations in a header field called Window size.
- ❑ Figure on next slide shows an example of how dynamic window management might work in a datagram network with 4-bit sequence numbers.

60

Error Control and Flow Control

A	Message	B	Comments
1 →	< request 8 buffers>	→	A wants 8 buffers
2 ←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3 →	<seq = 0, data = m0>	→	A has 3 buffers left now
4 →	<seq = 1, data = m1>	→	A has 2 buffers left now
5 →	<seq = 2, data = m2>	...	Message lost but A thinks it has 1 left
6 ←	<ack = 1, buf = 3>	←	B acknowledges 0 and 1, permits 2-4
7 →	<seq = 3, data = m3>	→	A has 1 buffer left
8 →	<seq = 4, data = m4>	→	A has 0 buffers left, and must stop
9 →	<seq = 2, data = m2>	→	A times out and retransmits
10 ←	<ack = 4, buf = 0>	←	Everything acknowledged, but A still blocked
11 ←	<ack = 4, buf = 1>	←	A may now send 5
12 ←	<ack = 4, buf = 2>	←	B found a new buffer somewhere
13 →	<seq = 5, data = m5>	→	A has 1 buffer left
14 →	<seq = 6, data = m6>	→	A is now blocked again
15 ←	<ack = 6, buf = 0>	←	A is still blocked
16 ...	<ack = 6, buf = 4>	←	Potential deadlock

Dynamic buffer allocation. The arrows show the direction of transmission. An ellipsis (...) indicates a lost TPDU

61

Error Control and Flow Control

- Until now we have tacitly assumed that the only limit imposed on the sender's data rate is the amount of buffer space available in the receiver.
- This is often not the case. Memory was once expensive but prices have fallen dramatically.
- Hosts may be equipped with sufficient memory that the lack of buffers is rarely, if ever, a problem, even for wide area connections.
- When buffer space no longer limits the maximum flow, another bottleneck will appear: the carrying capacity of the network.
- If adjacent routers can exchange at most x packets/sec and there are k disjoint paths between a pair of hosts, there is no way that those hosts can exchange more than kx segments/sec, no matter how much buffer space is available at each end.

62

Error Control and Flow Control

- ❑ If the sender pushes too hard (i.e., sends more than kx segments/sec), the network will become congested because it will be unable to deliver segments as fast as they are coming in.
- ❑ What is needed is a mechanism that limits transmissions from the sender based on the network's carrying capacity rather than on the receiver's buffering capacity.
- ❑ Belsnes (1975) proposed using a sliding window flow-control scheme in which the sender dynamically adjusts the window size to match the network's carrying capacity.
- ❑ This means that a dynamic sliding window can implement both flow control and congestion control.
- ❑ If the network can handle c segments/sec and the round-trip time (including transmission, propagation, queueing, processing at the receiver, and return of the acknowledgement) is r , the sender's window should be cr .

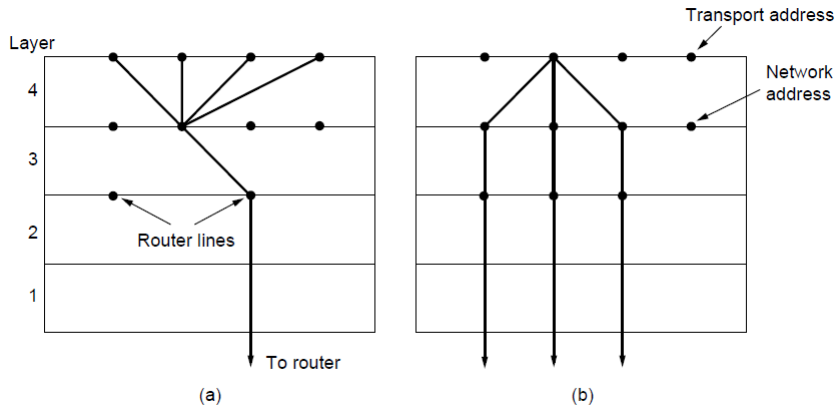
63

Error Control and Flow Control

- ❑ With a window of this size, the sender normally operates with the pipeline full.
- ❑ Any small decrease in network performance will cause it to block.
- ❑ Since the network capacity available to any given flow varies over time, the window size should be adjusted frequently, to track changes in the carrying capacity.
- ❑ TCP uses a similar scheme.

64

Multiplexing



(a) Multiplexing. (b) Inverse multiplexing.

65

Multiplexing

- ❑ Multiplexing, or sharing several conversations over connections, virtual circuits, and physical links plays a role in several layers of the network architecture.
- ❑ In the transport layer, the need for multiplexing can arise in a number of ways.
- ❑ For example, if only one network address is available on a host, all transport connections on that machine have to use it.
- ❑ When a segment comes in, some way is needed to tell which process to give it to. This situation, called multiplexing, is shown in Fig. (a) of previous slide.
- ❑ In this figure, four distinct transport connections all use the same network connection (e.g., IP address) to the remote host.

66

Multiplexing

- ❑ Multiplexing can also be useful in the transport layer for another reason. Suppose, for example, that a host has multiple network paths that it can use.
- ❑ If a user needs more bandwidth or more reliability than one of the network paths can provide, a way out is to have a connection that distributes the traffic among multiple network paths on a round-robin basis, as indicated in Fig. (b) of previous slide.
- ❑ This modus operandi is called **inverse multiplexing**. With k network connections open, the effective bandwidth might be increased by a factor of k .
- ❑ An example of inverse multiplexing is SCTP (Stream Control Transmission Protocol), which can run a connection using multiple network interfaces. In contrast, TCP uses a single network endpoint.

67

Crash Recovery

- ❑ If hosts and routers are subject to crashes or connections are long-lived (e.g., large software or media downloads), recovery from these crashes becomes an issue.
- ❑ If the transport entity is entirely within the hosts, recovery from network and router crashes is straightforward. The transport entities expect lost segments all the time and know how to cope with them by using retransmissions.
- ❑ A more troublesome problem is how to recover from host crashes. In particular, it may be desirable for clients to be able to continue working when servers crash and quickly reboot.
- ❑ To illustrate the difficulty, let us assume that one host (client), is sending a long file to another host (file server), using a simple stop-and-wait protocol.

68

Crash Recovery

- ❑ Partway through the transmission, the server crashes.
- ❑ When it comes back up, its tables are reinitialized, so it no longer knows precisely where it was.
- ❑ In an attempt to recover its previous status, the server might send a broadcast segment to all other hosts, announcing that it has just crashed and requesting that its clients inform it, about the status of all open connections.
- ❑ Each client can be in one of two states: one segment outstanding, S1, or no segments outstanding, S0.
- ❑ Based on only this state information, the client must decide whether to retransmit the most recent segment.

69

Crash Recovery

- ❑ At first glance, it is obvious, the client should retransmit if and only if it has an unacknowledged segment outstanding (i.e., is in state S1) when it learns of the crash.
- ❑ However, there are difficulties with this naive approach.
- ❑ Consider, for example, the situation in which the server's transport entity first sends an acknowledgement and then, when the acknowledgement has been sent, writes to the application process.
- ❑ Writing a segment onto the output stream and sending an acknowledgement are two distinct events that cannot be done simultaneously.
- ❑ The server can be programmed in one of two ways: acknowledge first or write first.

70

Crash Recovery

- The client can be programmed in one of four ways: always retransmit the last segment, never retransmit the last segment, retransmit only in state S0, or retransmit only in state S1.
- This gives eight combinations, but as we shall see, for each combination there is some set of events that makes the protocol fail.
- Three events are possible at the server: sending an acknowledgement (A), writing to the output process (W), and crashing (C).
- The three events can occur in six different orderings: AC(W), AWC, C(AW), C(WA), WAC, and WC(A), where the parentheses are used to indicate that neither A nor W can follow C (i.e., once it has crashed, it has crashed).
- Figure on next slide shows all eight combinations of client and server strategies and the valid event sequences for each one.

71

Crash Recovery

Different combinations of client and server strategy

Strategy used by sending host	Strategy used by receiving host					
	First ACK, then write			First write, then ACK		
	AC(W)	AWC	C(AW)	C(WA)	WAC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
 DUP = Protocol generates a duplicate message
 LOST = Protocol loses a message

72

Crash Recovery

- ❑ Making the protocol more elaborate does not help. Even if the client and server exchange several segments before the server attempts to write, so that the client knows exactly what is about to happen, the client has no way of knowing whether a crash occurred just before or just after the write.
- ❑ The conclusion is inescapable: under our ground rules of no simultaneous events—that is, separate events happen one after another not at the same time—host crash and recovery cannot be made transparent to higher layers.
- ❑ Put in more general terms, this result can be restated as “recovery from a layer N crash can only be done by layer $N + 1$,” and then only if the higher layer retains enough status information to reconstruct where it was before the problem occurred.

73

Congestion Control

- ❑ If the transport entities on many machines send too many packets into the network too quickly, the network will become congested, with performance degraded as packets are delayed and lost.
- ❑ Controlling congestion to avoid this problem is the combined responsibility of the network and transport layers.
- ❑ Congestion occurs at routers, so it is detected at the network layer. However, congestion is ultimately caused by traffic sent into the network by the transport layer.
- ❑ The only effective way to control congestion is for the transport protocols to send packets into the network more slowly.

74

Congestion Control

- ❑ The Internet relies heavily on the transport layer for congestion control, and specific algorithms are built into TCP and other protocols.
- ❑ Before we describe how to regulate traffic, we must understand what we are trying to achieve by running a congestion control algorithm.
- ❑ That is, we must specify the state in which a good congestion control algorithm will operate the network.
- ❑ The goal is more than to simply avoid congestion. It is to find a good allocation of bandwidth to the transport entities that are using the network.



75

Desirable bandwidth allocation

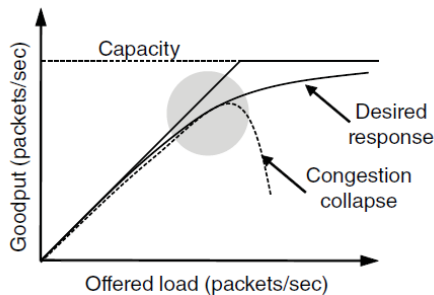
- ❑ A good allocation will deliver good performance because it uses all the available bandwidth but avoids congestion, it will be fair across competing transport entities, and it will quickly track changes in traffic demands.
- ❑ Efficiency and Power
- ❑ Max-Min Fairness
- ❑ Convergence



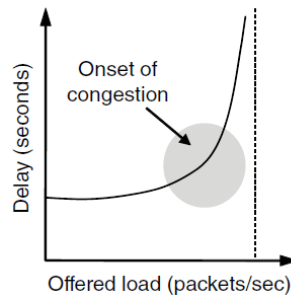
76

Desirable Bandwidth Allocation

- Efficiency allocation of bandwidth and Power



(a)



(b)

(a) Goodput and (b) delay as a function of offered load

77

Desirable bandwidth allocation

- For both goodput and delay, performance begins to degrade at the onset of congestion.
- Intuitively, we will obtain the best performance from the network if we allocate bandwidth up until the delay starts to climb rapidly. This point is below the capacity.
- To identify it, Kleinrock (1979) proposed the metric of power, where

$$power = \frac{load}{delay}$$

- Power will initially rise with offered load, as delay remains small and roughly constant, but will reach a maximum and fall as delay grows rapidly.
- The load with the highest power represents an efficient load for the transport entity to place on the network.

78

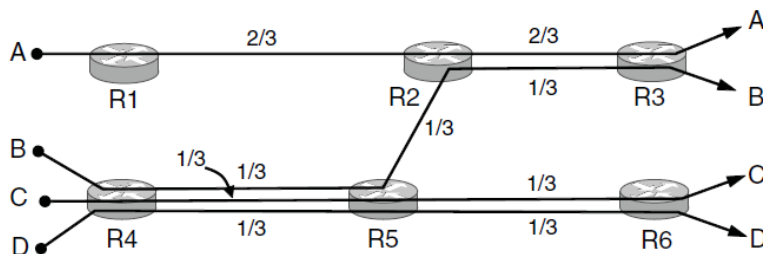
Desirable bandwidth allocation

- ❑ **Max-Min Fairness: how to divide bandwidth between different transport senders ?**
- ❑ This sounds like a simple question to answer—give all the senders an equal fraction of the bandwidth—but it involves several considerations.
 - ❑ what this problem has to do with congestion control ?
 - Networks do not have a strict bandwidth reservation for each flow or connection. They may for some flows if quality of service is supported, but many connections will seek to use whatever bandwidth is available or be lumped together by the network under a common allocation.
 - ❑ what a fair portion means for flows in a network ?
 - ❑ The form of fairness that is often desired for network usage is **max-min fairness**. An allocation is **max-min fair** if the bandwidth given to one flow cannot be increased without decreasing the bandwidth given to another flow with an allocation that is no larger. That is, increasing the bandwidth of a flow will only make the situation worse for flows that are less well off.

79

Desirable Bandwidth Allocation

- ❑ **Max-Min Fairness**



Max-min bandwidth allocation for four flows

80

Desirable bandwidth allocation

- ❑ **Max-Min Fairness: how to divide bandwidth between different transport senders ?**
- ❑ Max-min allocations can be computed given a global knowledge of the network.
- ❑ An intuitive way to think about them is to imagine that the rate for all of the flows starts at zero and is slowly increased. When the rate reaches a bottleneck for any flow, then that flow stops increasing.
- ❑ The other flows all continue to increase, sharing equally in the available capacity, until they too reach their respective bottlenecks.
- ❑ A third consideration is the level over which to consider fairness.
- ❑ A network could be fair at the level of connections, connections between a pair of hosts, or all connections per host.
- ❑ For example, defining fairness per host means that a busy server will fare no better than a mobile phone, while defining fairness per connection encourages hosts to open more connections.

81

Desirable bandwidth allocation

- ❑ Given that there is no clear answer, fairness is often considered per connection, but precise fairness is usually not a concern.
- ❑ It is more important in practice that no connection be starved of bandwidth than that all connections get precisely the same amount of bandwidth.
- ❑ In fact, with TCP it is possible to open multiple connections and compete for bandwidth more aggressively.
- ❑ This tactic is used by bandwidth-hungry applications such as **BitTorrent** for peer-to-peer file sharing.

82

Desirable bandwidth allocation

□ Convergence

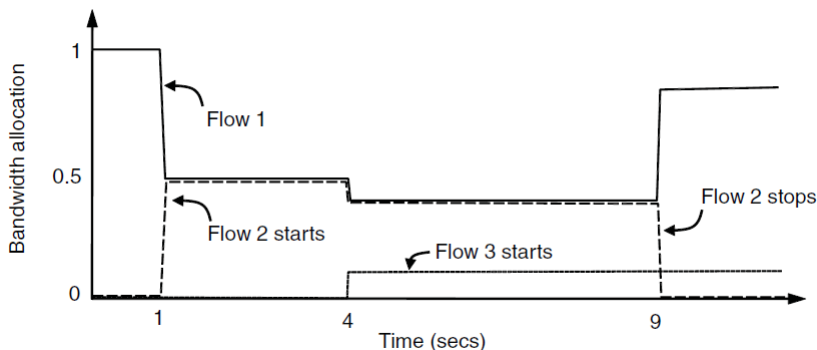
- Because of the variation in demand, the ideal operating point for the network varies over time.
- A good congestion control algorithm should rapidly converge to the ideal operating point, and it should track that point as it changes over time.
- If the convergence is too slow, the algorithm will never be close to the changing operating point.
- If the algorithm is not stable, it may fail to converge to the right point in some cases, or even oscillate around the right point



83

Desirable Bandwidth Allocation

□ Convergence



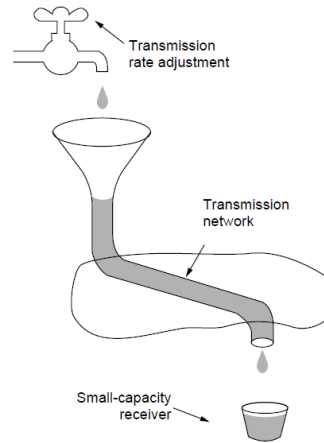
Changing bandwidth allocation over time



84

Regulating the Sending Rate

- ❑ The sending rate may be limited by two factors. The first is flow control, in the case that there is insufficient buffering at the receiver.
- ❑ The second is congestion, in the case that there is insufficient capacity in the network.
- ❑ In the figure, a thick pipe leading to a small-capacity receiver. This is a flow-control limited situation.
- ❑ As long as the sender does not send more water than the bucket can contain, no water will be lost.

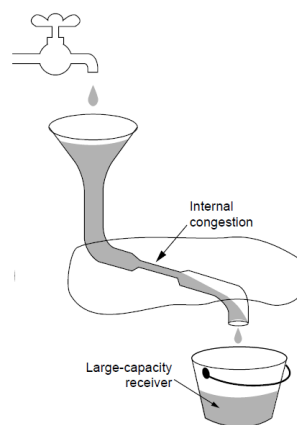


(a) A fast network feeding a low-capacity receiver

85

Regulating the Sending Rate

- ❑ In this Figure(b), the limiting factor is not the bucket capacity, but the internal carrying capacity of the network.
- ❑ If too much water comes in too fast, it will back up and some will be lost (in this case, by overflowing the funnel).
- ❑ These cases may appear similar to the sender, as transmitting too fast causes packets to be lost.
- ❑ However, they have different causes and call for different solutions.



(b) A slow network feeding a high-capacity receiver

86

Regulating the Sending Rate

- ❑ Now we will consider a congestion control solution.
- ❑ Since either of these problems can occur, the transport protocol will in general need to run both solutions and slow down if either problem occurs.
- ❑ The way that a transport protocol should regulate the sending rate depends on the form of the feedback returned by the network.
- ❑ Different network layers may return different kinds of feedback. The feedback may be explicit or implicit, and it may be precise or imprecise.
- ❑ An example of an explicit, precise design is when routers tell the sources the rate at which they may send. Designs in the literature such as XCP (eXplicit Congestion Protocol) operate in this manner (Katabi et al., 2002).
- ❑ An explicit, imprecise design is the use of ECN (Explicit Congestion Notification) with TCP.

87

Regulating the Sending Rate

- ❑ In other designs, there is no explicit signal. FAST TCP measures the roundtrip delay and uses that metric as a signal to avoid congestion (Wei et al., 2006).
- ❑ Finally, in the form of congestion control most prevalent in the Internet today, TCP with drop-tail or RED routers, packet loss is inferred and used to signal that the network has become congested.
- ❑ There are many variants of this form of TCP, including CUBIC TCP, which is used in Linux (Ha et al., 2008).
- ❑ Combinations are also possible. For example, Windows includes Compound TCP that uses both packet loss and delay as feedback signals (Tan et al., 2006).

88

Regulating the Sending Rate

Protocol	Signal	Explicit?	Precise?
XCP	Rate to use	Yes	Yes
TCP with ECN	Congestion warning	Yes	No
FAST TCP	End-to-end delay	No	Yes
CUBIC TCP	Packet loss	No	No
TCP	Packet loss	No	No

Some congestion control protocols

89

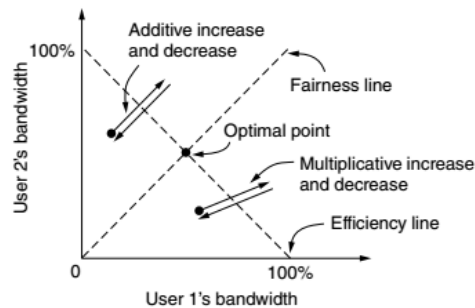
Regulating the Sending Rate

- ❑ If an explicit and precise signal is given, the transport entity can use that signal to adjust its rate to the new operating point.
- ❑ In the other cases, however, some guesswork is involved.
- ❑ In the absence of a congestion signal, the senders should increase their rates.
- ❑ When a congestion signal is given, the senders should decrease their rates.
- ❑ The way in which the rates are increased or decreased is given by a control law. These laws have a major effect on performance.

90

Regulating the Sending Rate

- ❑ **Additive Increase Multiplicative Decrease (AIMD)** control law (Chiu and Jain, 1989) is the appropriate control law to arrive at the efficient and fair operating point.
- ❑ Consider the simple case of two connections competing for the bandwidth of a single link.



91

Regulating the Sending Rate

- ❑ When the allocation is fair, both users will receive the same amount of bandwidth.
- ❑ This is shown by the dotted fairness line.
- ❑ When the allocations sum to 100%, the capacity of the link, the allocation is efficient.
- ❑ This is shown by the dotted efficiency line.
- ❑ A congestion signal is given by the network to both users when the sum of their allocations crosses this line.
- ❑ The intersection of these lines is the desired operating point, when both users have the same bandwidth and all of the network bandwidth is used.

92

Regulating the Sending Rate

- ❑ Consider what happens from some starting allocation if both user 1 and user 2 additively increase their respective bandwidths over time.
- ❑ For example, the users may each increase their sending rate by 1 Mbps every second.
- ❑ Eventually, the operating point crosses the efficiency line and both users receive a congestion signal from the network.
- ❑ At this stage, they must reduce their allocations. However, an additive decrease would simply cause them to oscillate along an additive line.
- ❑ This situation is shown in Fig. of previous slide.

93

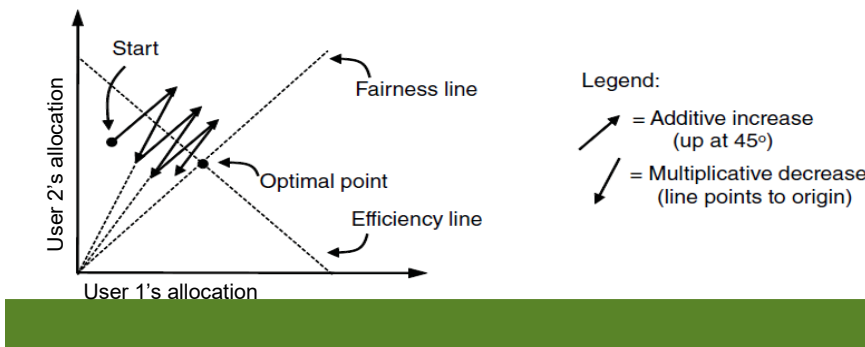
Regulating the Sending Rate

- ❑ Similarly, consider the case when both users multiplicatively increase their bandwidth over time until they receive a congestion signal.
- ❑ For example, the users may increase their sending rate by 10% every second. If they then multiplicatively decrease their sending rates, the operating point of the users will simply oscillate along a multiplicative line. This behavior is also shown in Fig. of previous slide.
- ❑ The multiplicative line has a different slope than the additive line. (It points to the origin, while the additive line has an angle of 45 degrees.)
- ❑ But it is otherwise no better. In neither case will the users converge to the optimal sending rates that are both fair and efficient.

94

Regulating the Sending Rate

- Now consider the case that the users additively increase their bandwidth allocations and then multiplicatively decrease them when congestion is signaled.
- This behavior is the AIMD control law, and it is shown in below Figure.
- It can be seen that the path traced by this behavior does converge to the optimal point that is both fair and efficient.
- This convergence happens no matter what the starting point, making AIMD broadly useful.



95

The Internet Transport Protocols

- The Internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one.
- The protocols complement each other.
- The connectionless protocol is UDP. It does almost nothing beyond sending packets between applications, letting applications build their own protocols on top as needed.
- The connection-oriented protocol is TCP. It does almost everything.
- It makes connections and adds reliability with retransmissions, along with flow control and congestion control, all on behalf of the applications that use it.

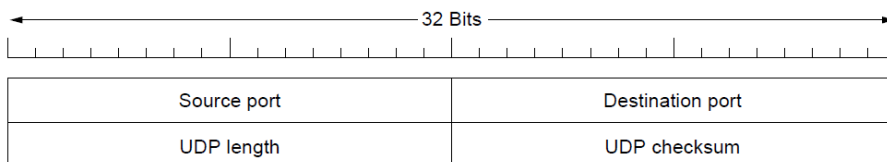
97

Introduction to UDP

- ❑ The Internet protocol suite supports a connectionless transport protocol called UDP (User Datagram Protocol).
- ❑ UDP provides a way for applications to send encapsulated IP datagrams without having to establish a connection.
- ❑ UDP is described in RFC 768.
- ❑ UDP transmits segments consisting of an 8-byte header followed by the payload. The header is shown in Fig. on next slide.
- ❑ The two ports serve to identify the endpoints within the source and destination machines.
- ❑ When a UDP packet arrives, its payload is handed to the process attached to the destination port.

98

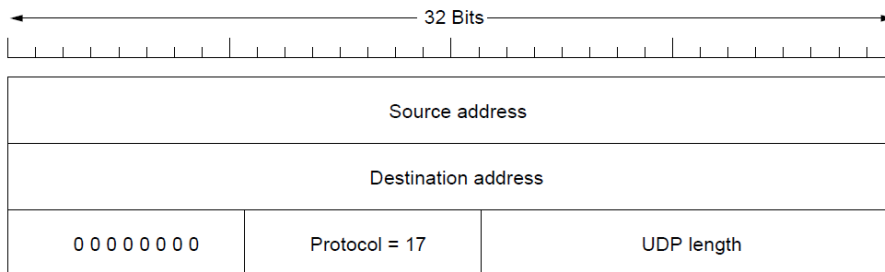
The UDP Header



The UDP header.

99

Introduction to UDP



The IPv4 pseudo header included in the UDP checksum.

100

The Internet Transport Protocols

- ❑ UDP is a simple protocol and it has some very important uses, such as client server interactions and multimedia.
- ❑ But for most Internet applications, reliable, sequenced delivery is needed.
- ❑ UDP cannot provide this, so another protocol is required.
- ❑ It is called TCP and is the main workhorse of the Internet.

107

Introduction to TCP

- ❑ TCP (Transmission Control Protocol) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork.
- ❑ An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters.
- ❑ TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.
- ❑ TCP was formally defined in RFC 793 in September 1981.
- ❑ As time went on, many improvements have been made, and various errors and inconsistencies have been fixed.

110

Introduction to TCP

- ❑ Each machine supporting TCP has a TCP transport entity, either a library procedure, a user process, or most commonly part of the kernel.
- ❑ In all cases, it manages TCP streams and interfaces to the IP layer.
- ❑ A TCP entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64 KB (in practice, often 1460 data bytes in order to fit in a single Ethernet frame with the IP and TCP headers), and sends each piece as a separate IP datagram.
- ❑ When datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which reconstructs the original byte streams.

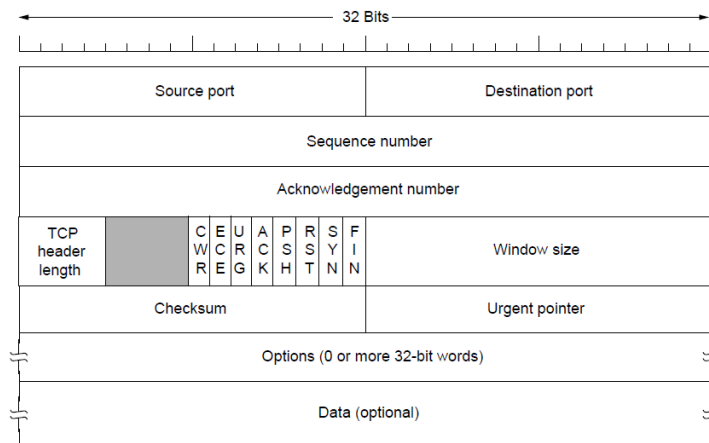
111

Introduction to TCP

- ❑ The IP layer gives no guarantee that datagrams will be delivered properly, nor any indication of how fast datagrams may be sent.
- ❑ It is up to TCP to send datagrams fast enough to make use of the capacity but not cause congestion, and to time out and retransmit any datagrams that are not delivered.
- ❑ Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence.
- ❑ In short, TCP must furnish good performance with the reliability that most applications want and that IP does not provide.

112

The TCP Segment Header



The TCP header.

115

The TCP Segment Header

- ❑ Every segment begins with a fixed-format, 20-byte header. The fixed header may be followed by header options.
- ❑ After the options, if any, up to $65,535 - 20 - 20 = 65,495$ data bytes may follow, where the first 20 refer to the IP header and the second to the TCP header.
- ❑ Segments without any data are legal and are commonly used for acknowledgements and control messages.
- ❑ The Source port and Destination port fields identify the local end points of the connection.
- ❑ A TCP port plus its host's IP address forms a 48-bit unique end point. The source and destination end points together identify the connection.
- ❑ This connection identifier is called a 5 tuple because it consists of five pieces of information: the protocol (TCP), source IP and source port, and destination IP and destination port.

116

The TCP Segment Header

- ❑ The Sequence number and Acknowledgement number fields perform their usual functions.
- ❑ Note that the latter specifies the next in-order byte expected, not the last byte correctly received. It is a cumulative acknowledgement because it summarizes the received data with a single number.
- ❑ It does not go beyond last data. Both are 32 bits because every byte of data is numbered in a TCP stream.
- ❑ The TCP header length tells how many 32-bit words are contained in the TCP header. This information is needed because the Options field is of variable length, so the header is, too.
- ❑ Next comes a 4-bit field that is not used.

117

The TCP Segment Header

- ❑ The Sequence number and Acknowledgement number fields perform their usual functions.
- ❑ Note that the latter specifies the next in-order byte expected, not the last byte correctly received. It is a cumulative acknowledgement because it summarizes the received data with a single number.
- ❑ It does not go beyond last data. Both are 32 bits because every byte of data is numbered in a TCP stream.
- ❑ The TCP header length tells how many 32-bit words are contained in the TCP header. This information is needed because the Options field is of variable length, so the header is, too.
- ❑ Next comes a 4-bit field that is not used.

118

The TCP Segment Header

- ❑ Now come eight 1-bit flags. CWR and ECE are used to signal congestion when ECN (Explicit Congestion Notification) is used, as specified in RFC 3168.
- ❑ ECE is set to signal an ECN-Echo to a TCP sender to tell it to slow down when the TCP receiver gets a congestion indication from the network.
- ❑ CWR is set to signal Congestion Window Reduced from the TCP sender to the TCP receiver so that it knows the sender has slowed down and can stop sending the ECN-Echo.
- ❑ URG is set to 1 if the Urgent pointer is in use. The Urgent pointer is used to indicate a byte offset from the current sequence number at which urgent data are to be found.

119

The TCP Segment Header

- ❑ The ACK bit is set to 1 to indicate that the Acknowledgement number is valid.
- ❑ This is the case for nearly all packets. If ACK is 0, the segment does not contain an acknowledgement, so the Acknowledgement number field is ignored.
- ❑ The PSH bit indicates PUSHed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency).
- ❑ The RST bit is used to abruptly reset a connection that has become confused due to a host crash or some other reason. It is also used to reject an invalid segment or refuse an attempt to open a connection.
- ❑ In general, if you get a segment with the RST bit on, you have a problem on your hands.

120

The TCP Segment Header

- ❑ The SYN bit is used to establish connections.
- ❑ The connection request has $\text{SYN} = 1$ and $\text{ACK} = 0$ to indicate that the piggyback acknowledgement field is not in use.
- ❑ The connection reply does bear an acknowledgement, however, so it has $\text{SYN} = 1$ and $\text{ACK} = 1$.
- ❑ In essence, the SYN bit is used to denote both CONNECTION REQUEST and CONNECTION ACCEPTED, with the ACK bit used to distinguish between those two possibilities.
- ❑ The FIN bit is used to release a connection. It specifies that the sender has no more data to transmit.

121

The TCP Segment Header

- ❑ However, after closing a connection, the closing process may continue to receive data indefinitely.
- ❑ Both SYN and FIN segments have sequence numbers and are thus guaranteed to be processed in the correct order.
- ❑ Flow control in TCP is handled using a variable-sized sliding window.
- ❑ The Window size field tells how many bytes may be sent starting at the byte acknowledged.
- ❑ A Window size field of 0 is legal and says that the bytes up to and including Acknowledgement number – 1 have been received, but that the receiver has not had a chance to consume the data and would like no more data for the moment, thank you.

122

The TCP Segment Header

- ❑ The receiver can later grant permission to send by transmitting a segment with the same Acknowledgement number and a nonzero Window size field.
- ❑ A Checksum is also provided for extra reliability. It checksums the header, the data, and a conceptual pseudo header in exactly the same way as UDP, except that the pseudo header has the protocol number for TCP (6) and the checksum is mandatory.
- ❑ The Options field provides a way to add extra facilities not covered by the regular header.
- ❑ Many options have been defined and several are commonly used.

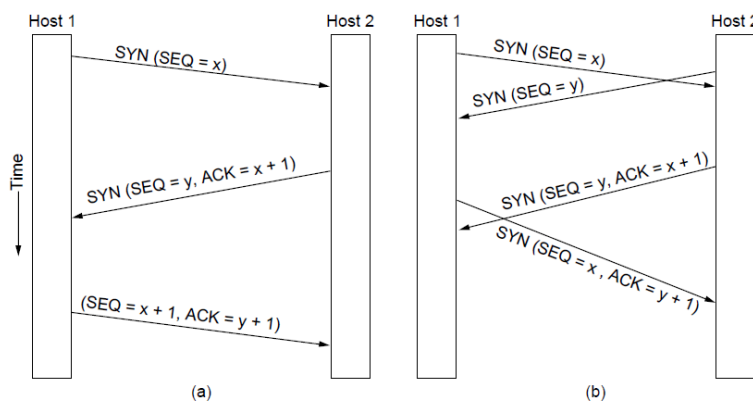
123

The TCP Segment Header

- ❑ The options are of variable length, fill a multiple of 32 bits by using padding with zeros, and may extend to 40 bytes to accommodate the longest TCP header that can be specified.
- ❑ Some options are carried when a connection is established to negotiate or inform the other side of capabilities.
- ❑ Other options are carried on packets during the lifetime of the connection.
- ❑ Each option has a Type-Length-Value encoding.

124

TCP Connection Establishment



- (a) TCP connection establishment in the normal case.
- (b) Simultaneous connection establishment on both sides.

125

TCP Connection Release

- ❑ Although TCP connections are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections.
- ❑ Each simplex connection is released independently of its sibling.
- ❑ To release a connection, either party can send a TCP segment with the FIN bit set, which means that it has no more data to transmit.
- ❑ When the FIN is acknowledged, that direction is shut down for new data.
- ❑ Data may continue to flow indefinitely in the other direction, however. When both directions have been shut down, the connection is released.

126

TCP Connection Release

- ❑ To avoid the two-army problem, timers are used.
- ❑ If a response to a FIN is not forthcoming within two maximum packet lifetimes, the sender of the FIN releases the connection.
- ❑ The other side will eventually notice that nobody seems to be listening to it anymore and will time out as well.
- ❑ While this solution is not perfect, given the fact that a perfect solution is theoretically impossible, it will have to do.
- ❑ In practice, problems rarely arise.

127

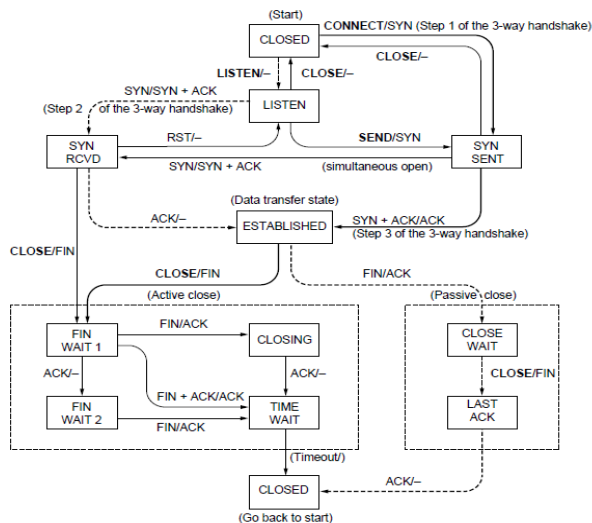
TCP Connection Management Modeling (1)

- The steps required to establish and release connections can be represented in a finite state machine with the 11 states as listed below:

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

128

TCP Connection Management Modeling (2)



TCP connection management finite state machine.

- The heavy solid line is the normal path for a client.
- The heavy dashed line is the normal path for a server.
- The light lines are unusual events.
- Each transition is labeled by the event causing it and the action resulting from it, separated by a slash.

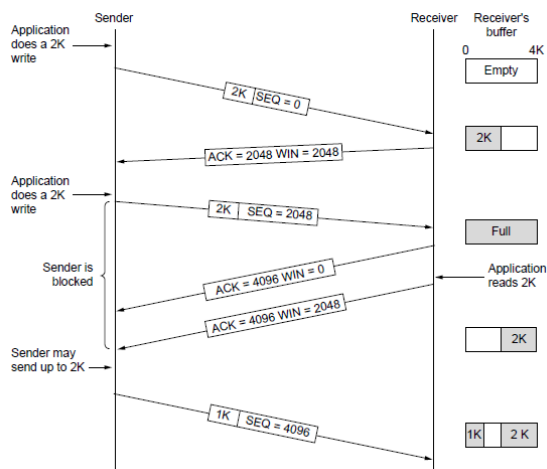
129

TCP Connection Management Modeling

- ❑ The event can either be a user-initiated system call (CONNECT, LISTEN, SEND, or CLOSE), a segment arrival (SYN, FIN, ACK, or RST), or, in one case, a timeout of twice the maximum packet lifetime.
- ❑ The action is the sending of a control segment (SYN, FIN, or RST) or nothing, indicated by —.
- ❑ Comments are shown in parentheses.

130

TCP Sliding Window



- ❑ Window management in TCP decouples the issues of acknowledgement of the correct receipt of segments and receiver buffer allocation unlike the Datalink Layer.

Window management in TCP

131

TCP Sliding Window

- ❑ When the window is 0, the sender may not normally send segments, with two exceptions.
- ❑ First, urgent data may be sent, for example, to allow the user to kill the process running on the remote machine.
- ❑ Second, the sender may send a 1-byte segment to force the receiver to reannounce the next byte expected and the window size. This packet is called a window probe.
- ❑ The TCP standard explicitly provides this option to prevent deadlock if a window update ever gets lost.
- ❑ Senders are not required to transmit data as soon as they come in from the application.
- ❑ Neither are receivers required to send acknowledgements as soon as possible.

132

Delayed Acknowledgements

- ❑ Consider a telnet connection, that reacts on every keystroke.
- ❑ In the worst case, whenever a character arrives at the sending TCP entity, TCP creates a 21 byte TCP segment, 20 bytes of header and 1 byte of data. For this segment, another ACK and window update is sent when the application reads that 1 byte. This results in a huge wastage of bandwidth.
- ❑ **Delayed acknowledgements:** Delay acknowledgement and window updates for up to 500 msec in the hope of receiving few more data packets within that interval.
- ❑ **However, the sender can still send multiple short data segments.**

133

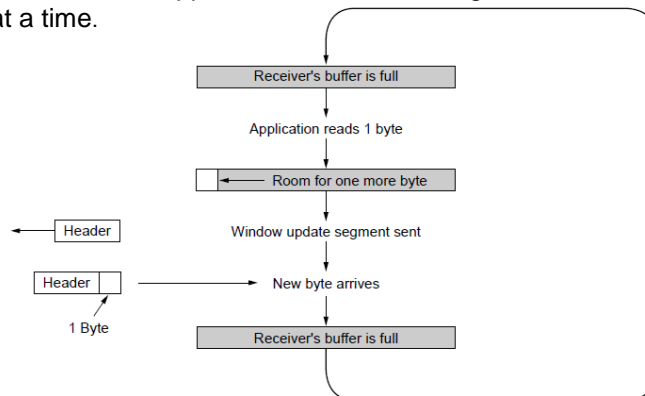
Nagle's Algorithm

- ❑ When data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged.
- ❑ Then send all buffered data in one TCP segment and start buffering again until the next segment is acknowledged.
 - ❑ **Only one short packet can be outstanding at any time.**
- ❑ The algorithm additionally says that a new segment should be sent if enough data have trickled in to fill a maximum segment.
- ❑ **Do we want Nagle's Algorithm all the time?**
 - ❑ **Not in Interactive Online Games**
- ❑ **Nagle's Algorithm and Delayed Acknowledgement**
 - ❑ Receiver waits for data and sender waits for acknowledgement – results in starvation

134

Silly Window Syndrome

- ❑ Another problem that can degrade TCP performance is the silly window syndrome (Clark, 1982).
- ❑ This problem occurs when data are passed to the sending TCP entity in large blocks, but an interactive application on the receiving side reads data only 1 byte at a time.



Silly window syndrome

135

Silly Window Syndrome

- ❑ Data are passed to the sending TCP entity in large blocks, but an interactive application on the receiver side reads data only 1 byte at a time.
 - ❑ Clark's solution: Do not send window update for 1 byte. Wait until sufficient space is available at the receiver buffer.
 - ❑ Specifically, the receiver should not send a window update until it can handle the maximum segment size it advertised when the connection was established or until its buffer is half empty, whichever is smaller.
 - ❑ Furthermore, the sender can also help by not sending tiny segments. Instead, it should wait until it can send a full segment, or at least one containing half of the receiver's buffer size.

136

Handling Short Segments – Sender and Receiver Together

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

137

Handling Out of Order in TCP

- ❑ TCP buffers out of order segments and forward a duplicate acknowledgement to the sender.
- ❑ **Acknowledgement in TCP – Cumulative acknowledgement**
- ❑ Receiver has received bytes 0, 1, 2, __, 4, 5, 6, 7
 - ❑ TCP sends a cumulative acknowledgement with ACK number 2, acknowledging everything up to byte 2
 - ❑ Once 4 is received, a duplicate ACK with ACK number 3 (next expected byte) is forwarded – **triggers congestion control**
 - ❑ After timeout, sender retransmits byte 3
 - ❑ Once byte 3 is received, it can send another cumulative ACK with ACK number 8 (next expected byte)

138

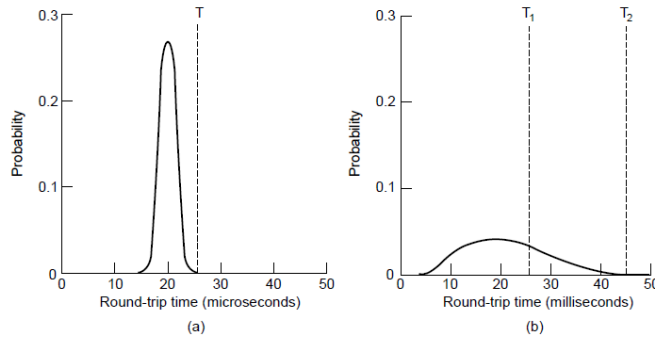
TCP Timer Management

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

139

TCP Timer Management

- (a) Probability density of acknowledgment arrival times in data link layer. (b) ... for TCP



RTT at Data Link Layer vs RTT at Transport Layer

140

RTT Estimation at the Transport Layer

- Use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance.
- **Jacobson's algorithm (1988) - used in TCP**
 - For each connection, TCP maintains a variable, **SRTT (smoothed Round Trip Time)** – best current estimate of the round trip time to the destination
 - When a segment is sent, a timer is started (both to see how long the acknowledgement takes and also to trigger a retransmission if it takes too long)
 - If the ACK gets back – measure the time (say, R)
 - Update SRTT as follows
$$SRTT = \alpha SRTT + (1 - \alpha)R$$
(Exponentially Weighted Moving Average – EWMA)
 - α is a smoothing factor that determines how quickly the old values are forgotten. Typically $\alpha = 7/8$

141

Problem with EWMA

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

142

RTO Estimation

- ❑ Update RTT variation ($RTTVAR$) as follows.
$$RTTVAR = \beta RTTVAR + (1 - \beta)|SRTT - R|$$
- ❑ Typically $\beta = \frac{3}{4}$
- ❑ RTO is estimated as follows,
$$RTO = SRTT + 4 \times RTTVAR$$
- ❑ **Why 4 ?**
 - ❑ Somehow arbitrary
 - ❑ Jacobson's paper is full of clever tricks – use integer addition, subtraction and shift – computation is lightweight

143

Karn's Algorithm

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

144

Other TCP Timers

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

145

TCP Congestion Control

- ❑ Based on implementation of AIMD using a window and with packet loss as the binary signal
- ❑ TCP maintains a **Congestion Window (CWnd)** – number of bytes the sender may have in the network at any time
- ❑ **Sending Rate = Congestion Window / RTT**
- ❑ Congestion window is maintained in addition to the flow control window (Receiver advertised window size, RWnd), which specifies the number of bytes that the receiver can buffer.
- ❑ Both windows are tracked in parallel, and the number of bytes that may be sent is the smaller of the two windows.
- ❑ **Sender Window (SWnd) = Min (CWnd, RWnd)**

150

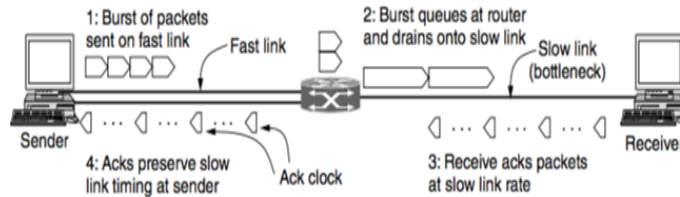
1986 Congestion Collapse

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

151

Adjust CWnd based on AIMD

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



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

152

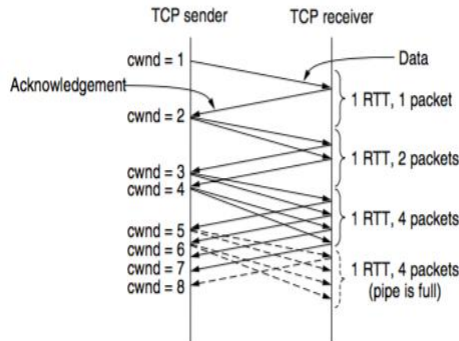
Increase Rate Exponentially at the Beginning – The Slow Start

- AIMD rule will take a very long time to reach a good operating point on fast networks if the CWnd is started from a small size.
- A 10 Mbps link with 100 ms RTT
 - Appropriate CWnd = BDP = 1 Mbit
 - 1250 byte packets -> 100 packets to reach BDP
 - CWnd starts at 1 packet, and increased 1 packet at every RTT
 - 100 RTTs are required 10 sec before the connection reaches to a moderate rate
- **Slow Start - Exponential increase of rate to avoid slow convergence**
 - Rate is not slow at all ! 😊
 - CWnd is doubled at every RTT

153

TCP Slow Start

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



154

Slow Start Threshold

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

155

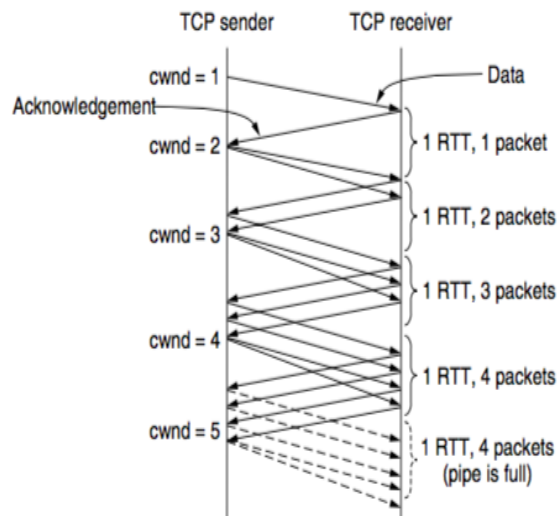
Additive Increase (Congestion Avoidance)

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

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

156

Additive Increase – Packet Wise Approximation



157

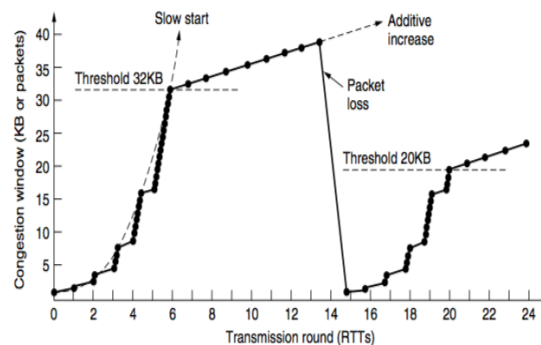
Triggering a Congestion

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

158

Fast Retransmission – TCP Tahoe

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



159

Fast Recovery – TCP Reno

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

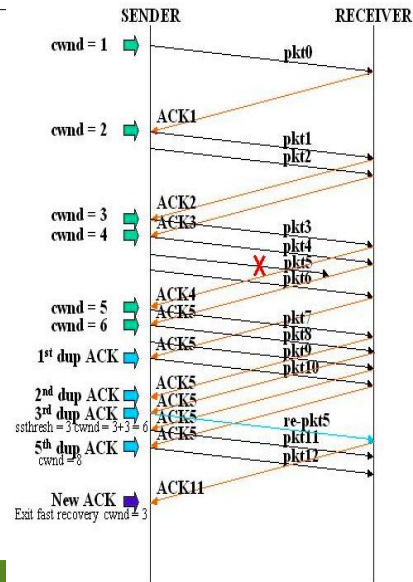
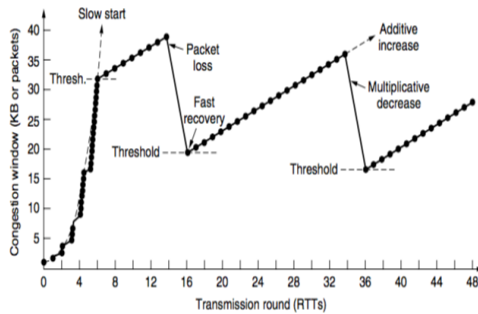
160

Fast Recovery – TCP Reno

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

161

Fast Recovery – TCP Reno



162

References

Text Book:

1. Computer Networks, Andrew S. Tanenbaum, David J. Wetherall, 5th Edition, Pearson/Prentice Hall Publication.

Reference Book:

1. Data and Computer Communication, William Stallings, 8th Edition, Pearson/Prentice Hall Publication.
2. Data Communications and Networking, Behrouz A. Forouzan, 3/e, McGrawHill Publication.
3. Computer Networks: A Systems Approach, Bruce S. Davie and Larry L. Peterson, 4e, Morgan Kaufmann Publication.
4. The TCP/IP Guide, by Charles M. Kozierok, Free online Resource <http://www.tcpipguide.com/free/>.

179

The End



180