計算機ネットワーク

開講クォーター: 1-2Q

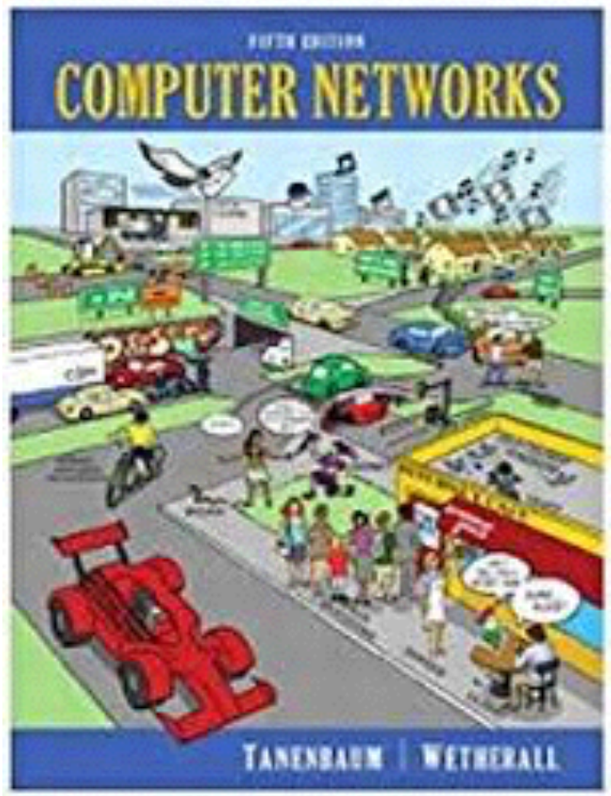
曜日・時限: 火7-8限

講義室: 1Q @ W834, 2Q @ W931

横田理央

rioyokota@gsic.titech.ac.jp

参考書

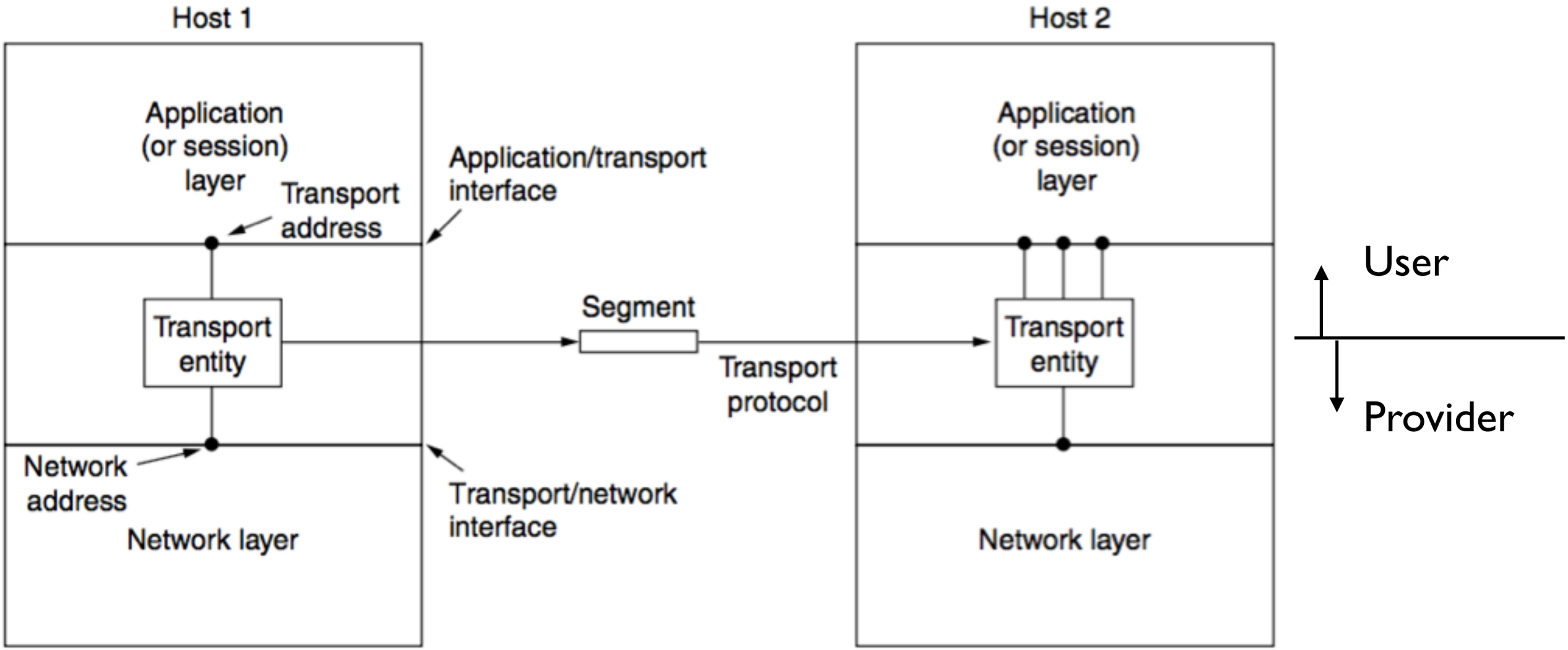


教科書

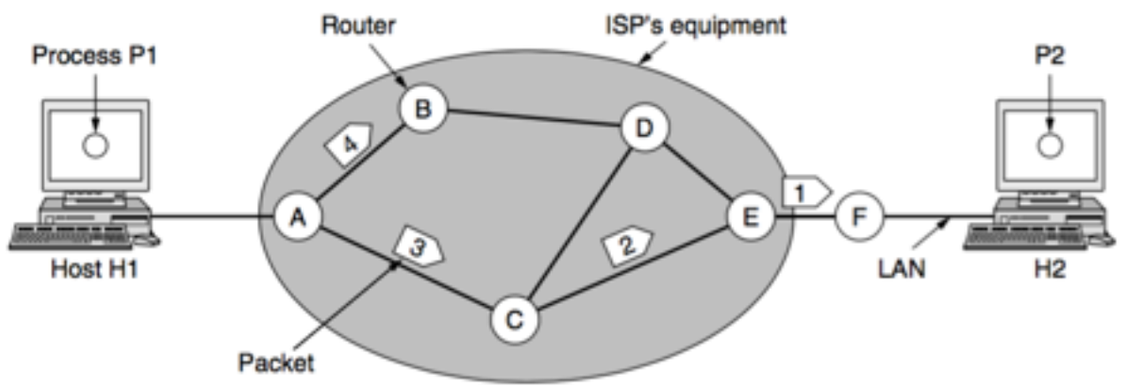
講義日程（2Q）

		授業計画		課題
06/14	第９回	ネットワーク層１ ルーティング・輻輳制御	５章	ルーティングの種類を理解し 輻輳制御手法を説明できる
06/21	第１０回	ネットワーク層２ インターネットとサービス品質	５章	インターネットの制御プロトコルを理解し ネットワーク間の接続について説明できる
06/28	第１１回	トランスポート層１ トランスポート・プロトコルの要素	６章	誤り制御とフロー制御を理解し 輻輳制御について説明できる
07/05	第１２回	トランスポート層２ UDP と TCP	６章	TCP の信頼性を理解し TCP のコネクション管理を説明できる
07/12	第１３回	アプリケーション層 DNS, 電子メール, www	７章	DNS, 電子メール, www のしくみを理解し ストリーミング，P2P について説明できる
07/26	第１４回	ネットワークセキュリティ１ 対称鍵暗号，公開鍵暗号	８章	暗号アルゴリズムを理解し SHA-1,2 と RSA について説明できる
08/02	第１５回	ネットワークセキュリティ２ デジタル署名，認証プロトコル	８章	電子メール,Web のセキュリティ の脅威について把握できる

The transport layer



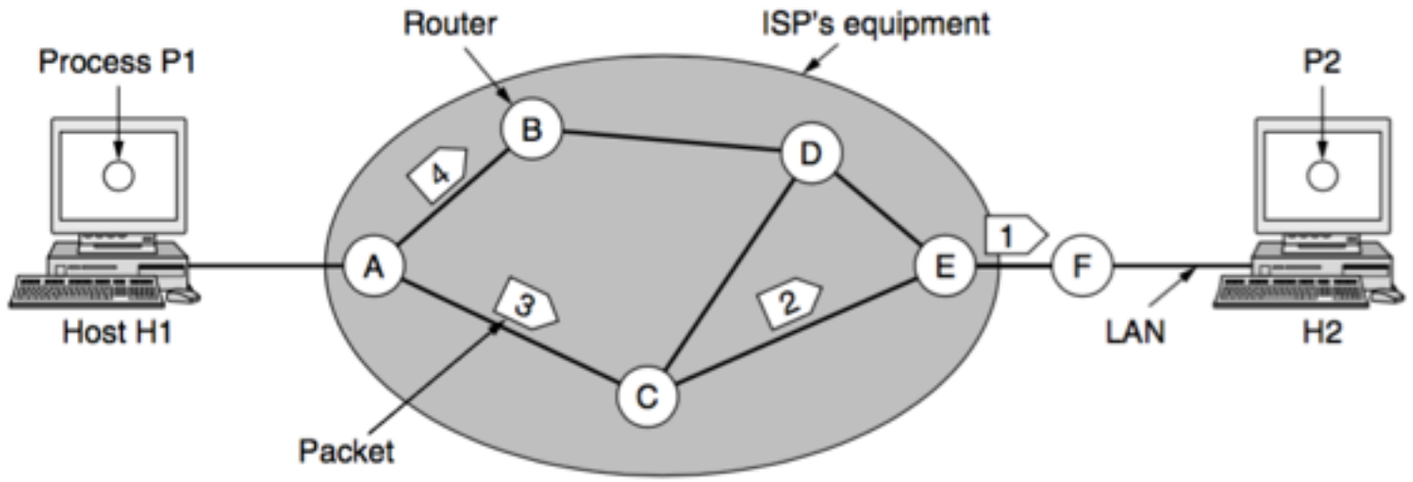
<u>Protocols</u>
TCP
UDP

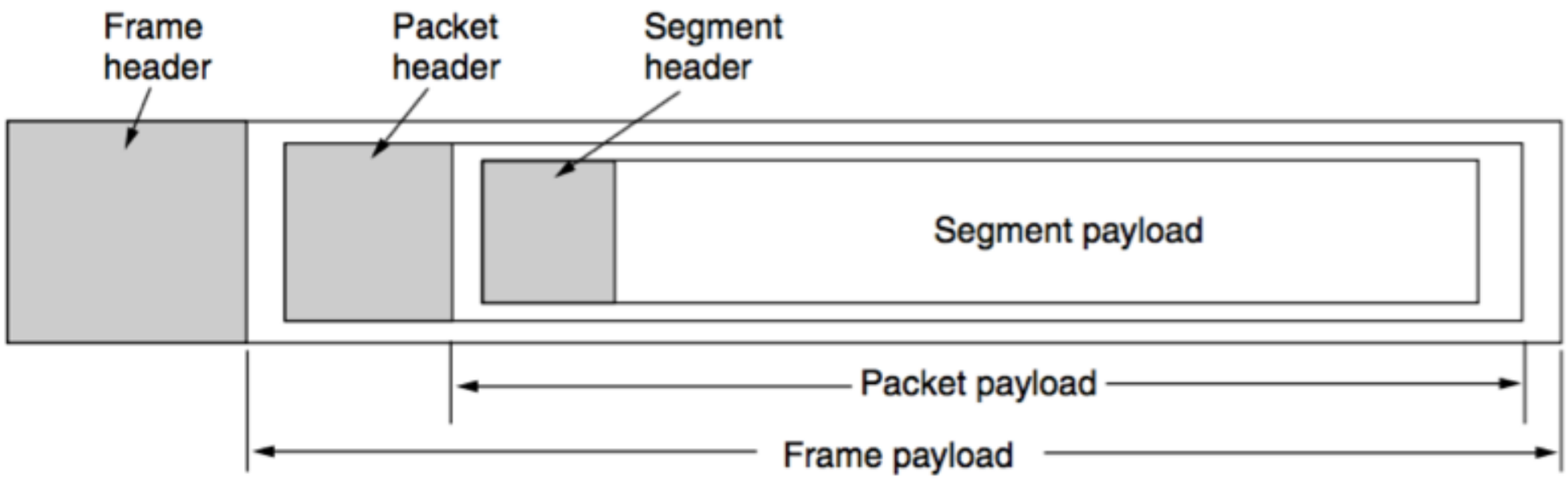


<u>End-to-end</u>
Error control
Flow control

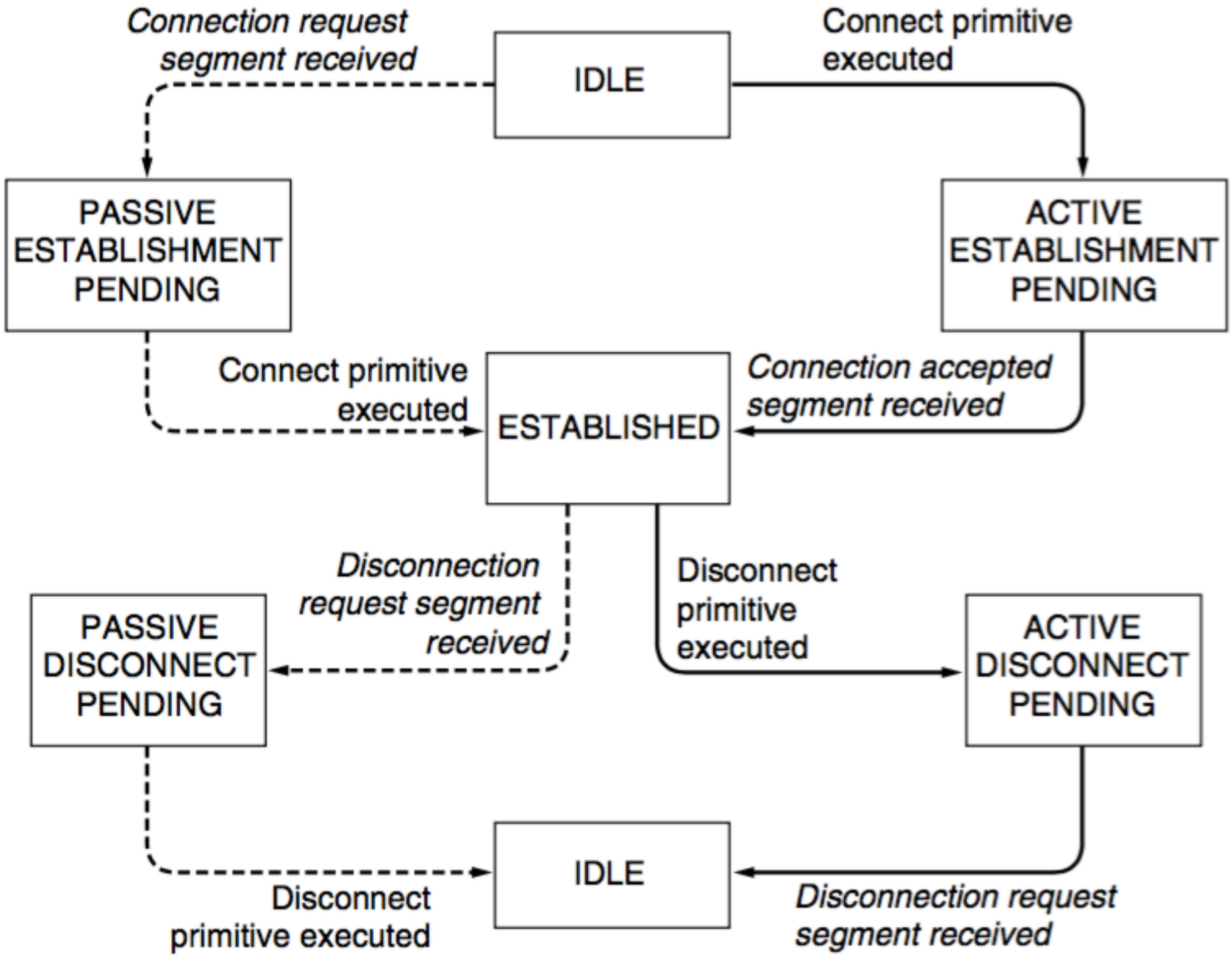
Transport protocol

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.	Request a release of the connection



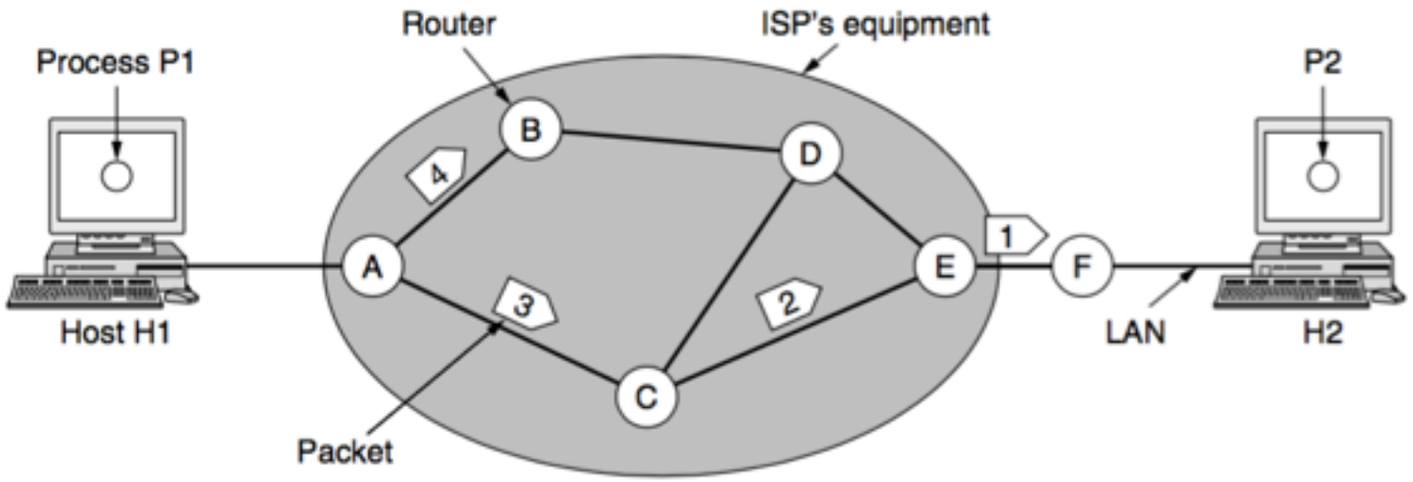


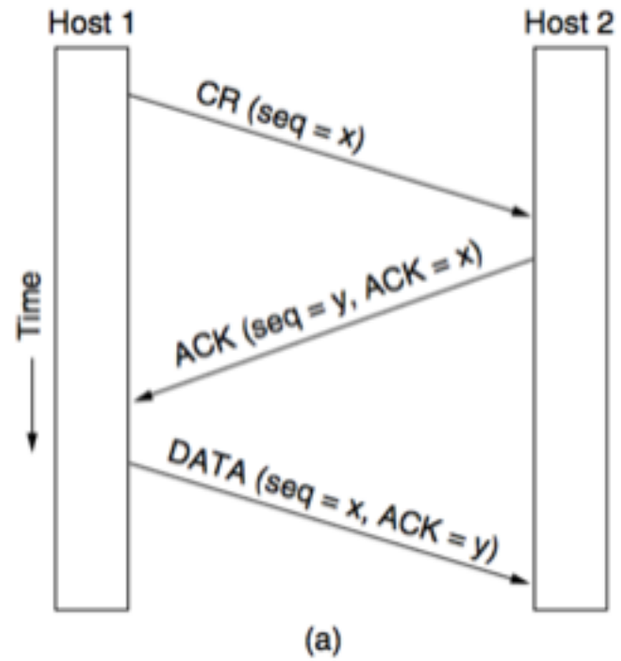
State diagram



The socket primitives for TCP

Primitive	Meaning
SOCKET	Create a new communication endpoint
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





Client code using sockets

```
#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 */

  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");

  /* 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);
}
```

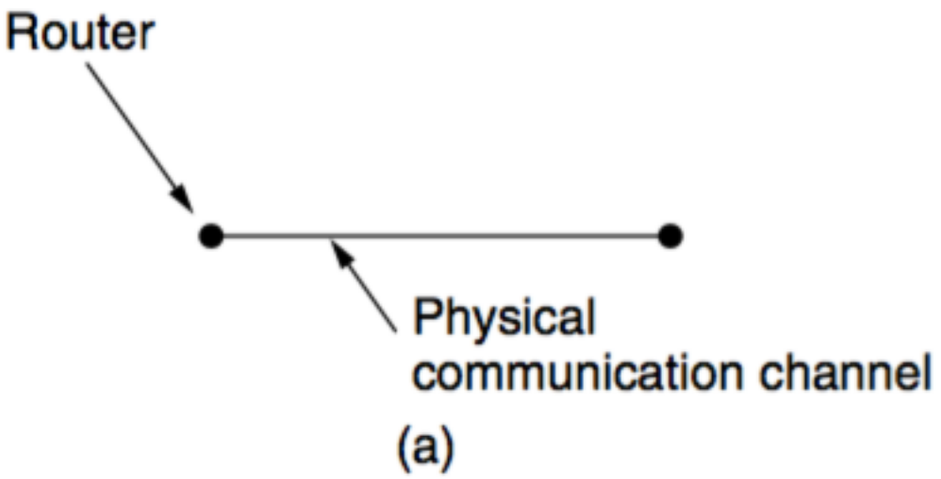
Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the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

Server code using sockets

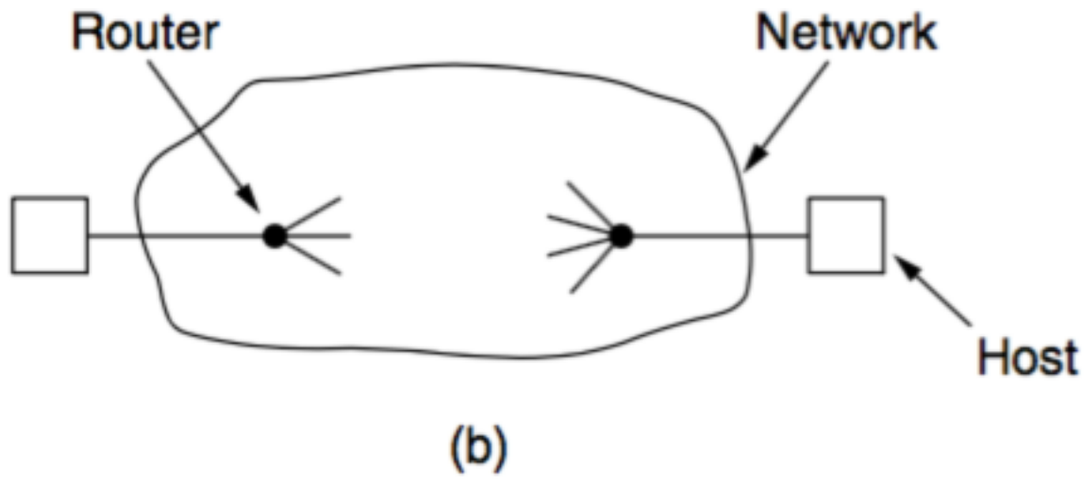
```
#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#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 buf[BUF_SIZE];                    /* buffer for outgoing file */
  struct sockaddr_in channel;            /* holds IP address */

  /* 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");

  /* 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 */
  }
}
```

Primitive	Meaning
SOCKET	Create a new communication endpoint
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

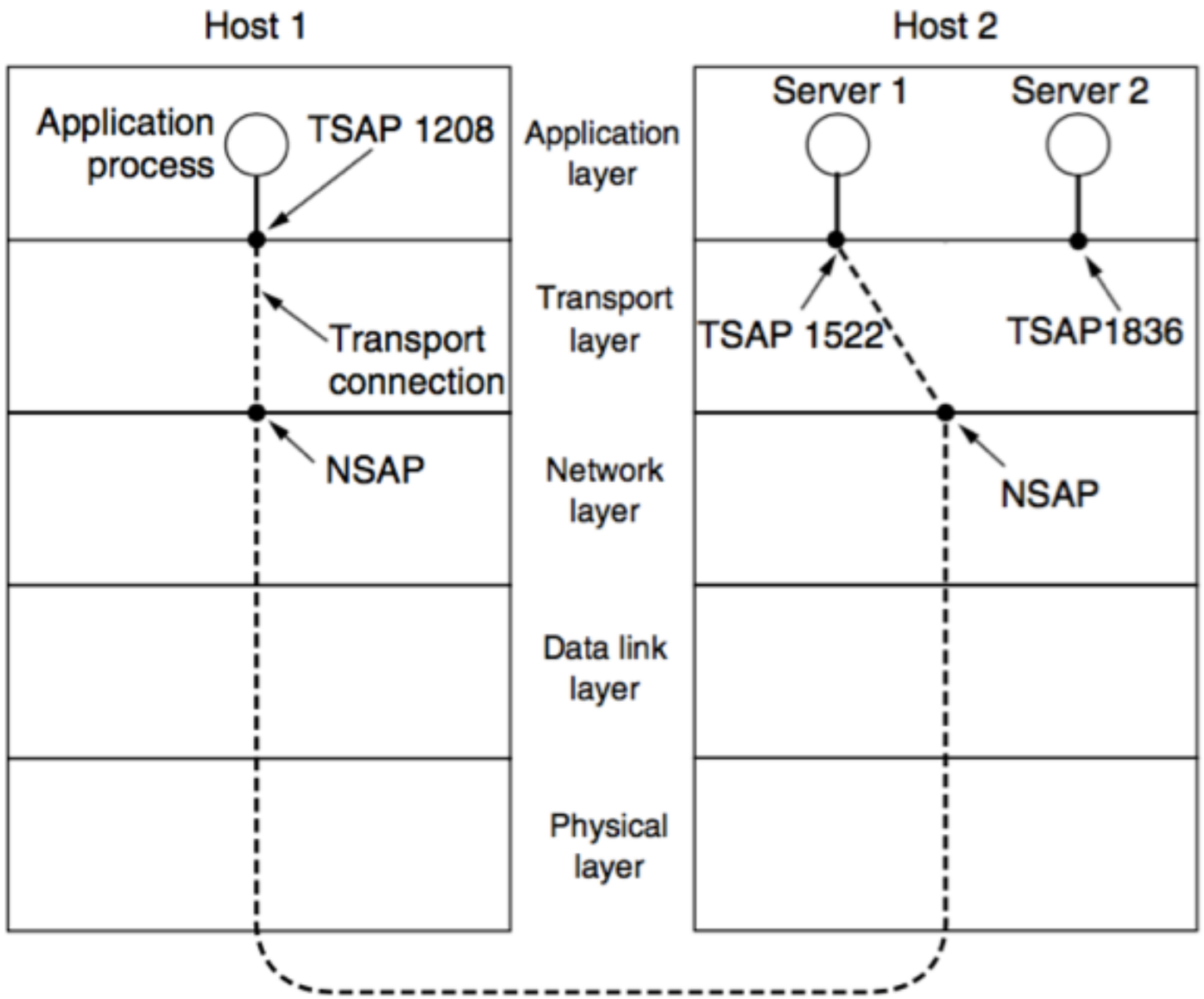
Transport Protocols



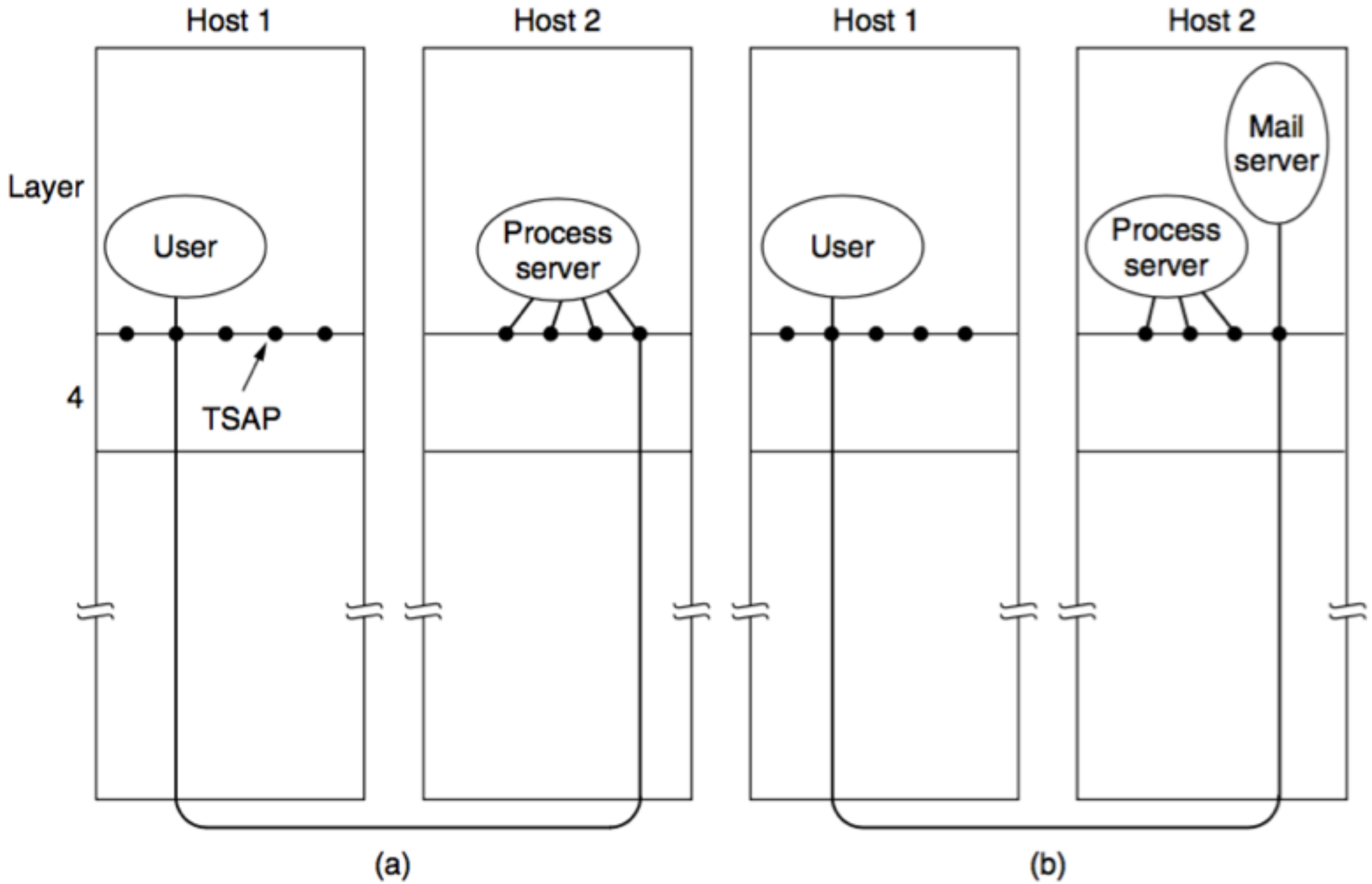
(a)



(b)



Initial Connection Protocol



Process Server

Instead of every conceivable server listening at a well-known TSAP, each machine that wishes to offer services to remote users has a special process server that acts as a proxy for less heavily used servers

Duplicate segments

1. How to prevent duplicate segments?

> Label segments with sequence numbers that will not be reused within *T* secs

2. What if the node crashes?

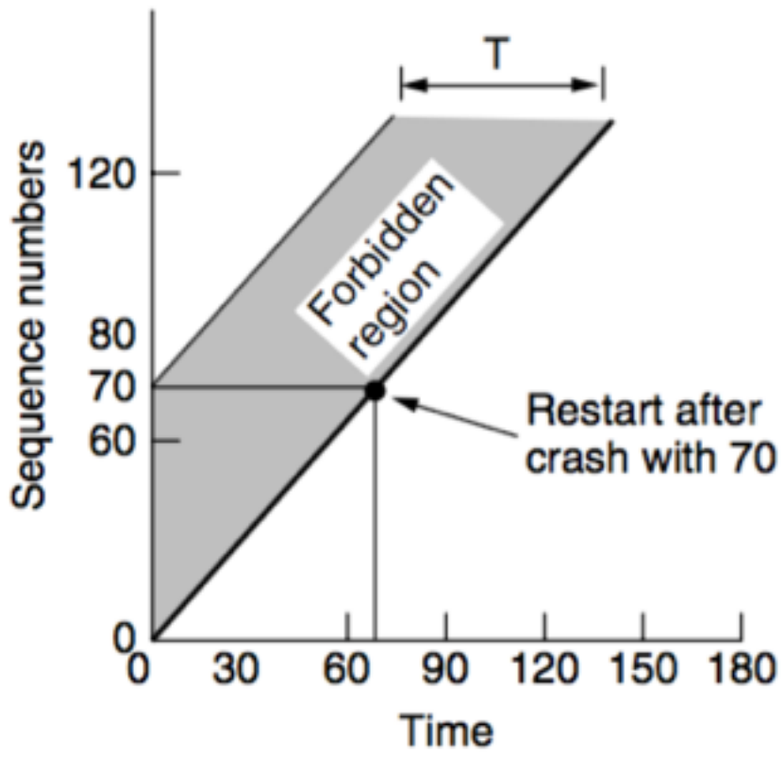
> Require transport entities to be idle for T secs after a recovery

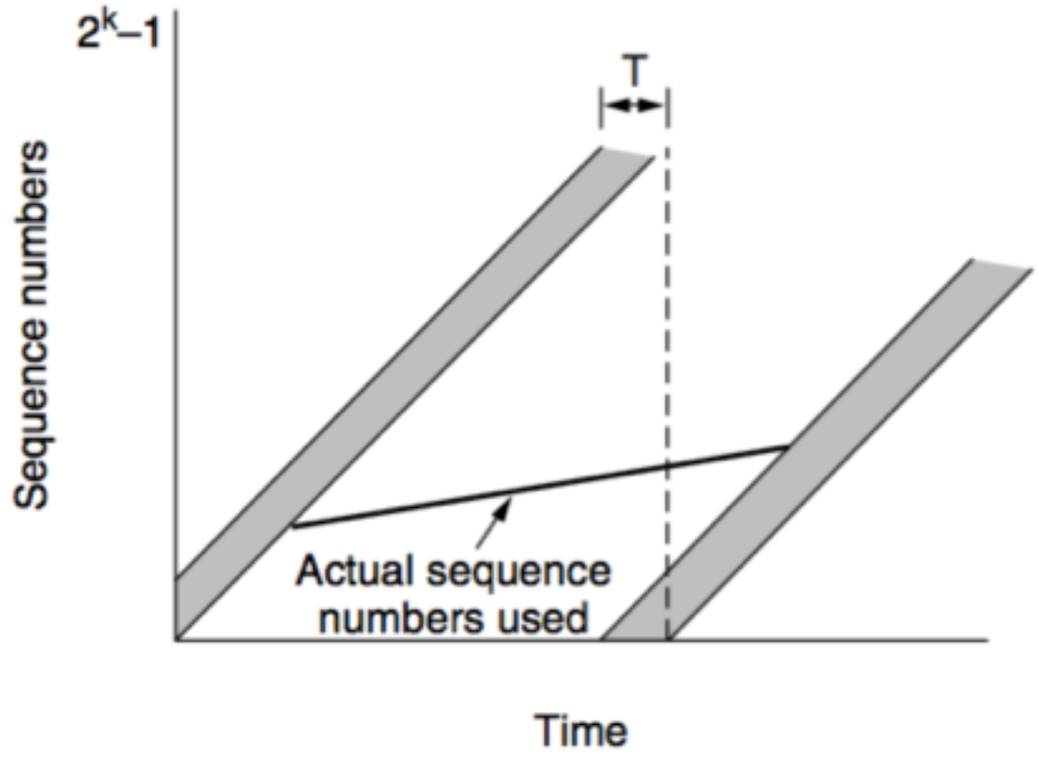
3. What if T is very large?

> Use the low-order k bits of a time-of-day clock

4. Is a clock-based scheme secure?

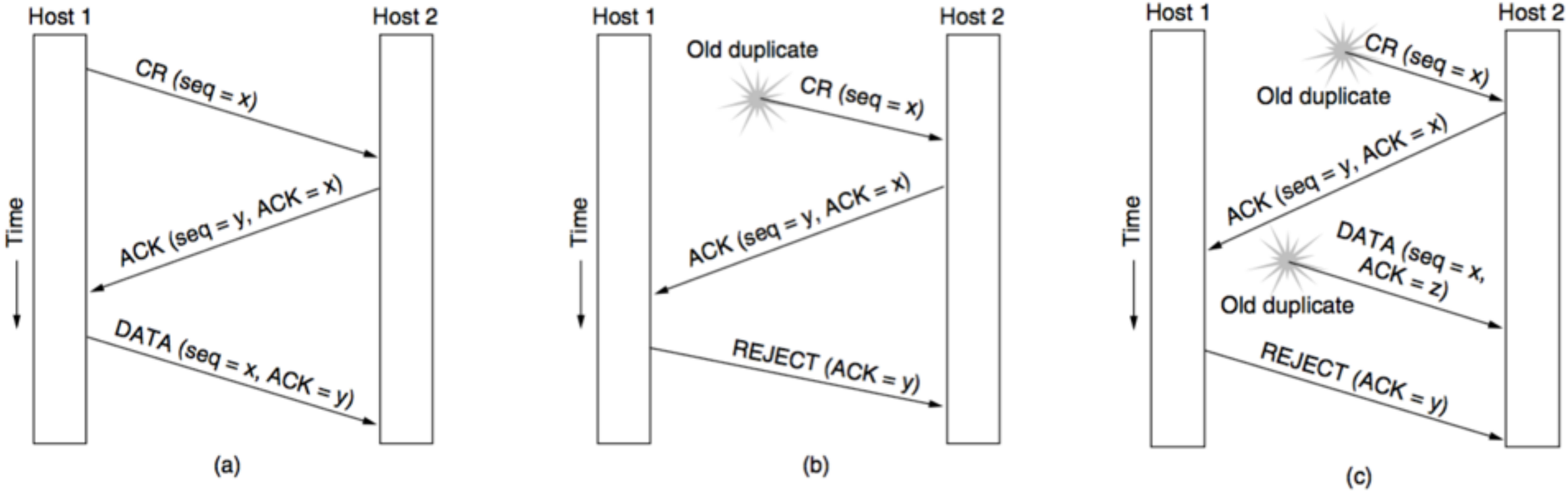
> Use pseudorandom initial sequence numbers





Connection Establishment

Three-way handshake

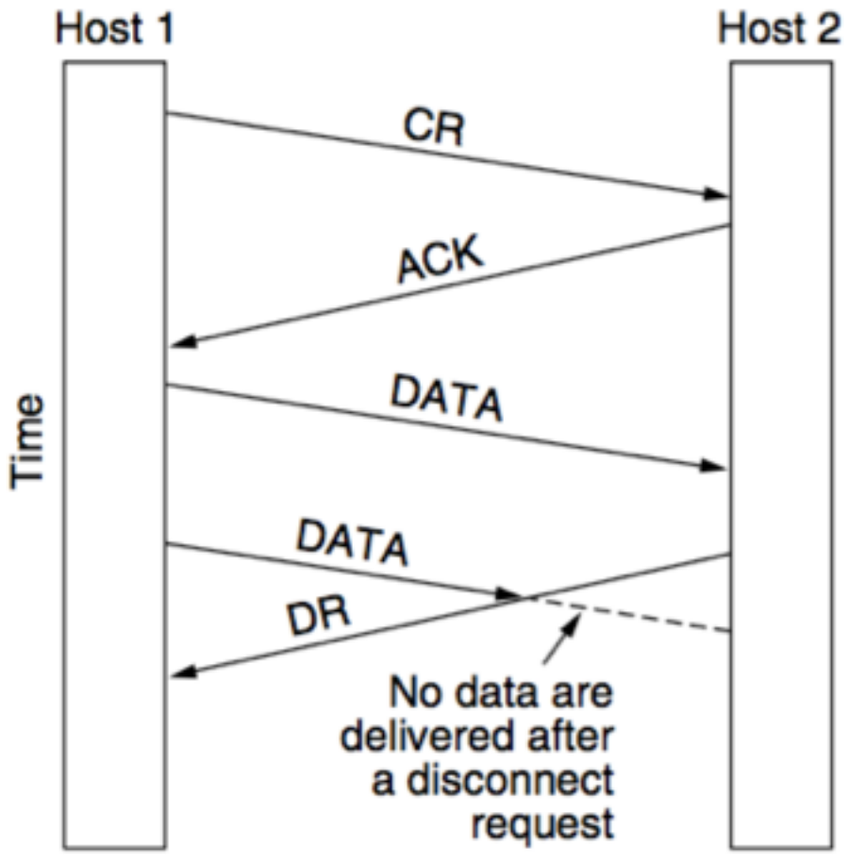


PAWS (Protection Against Wrapped Sequence numbers)

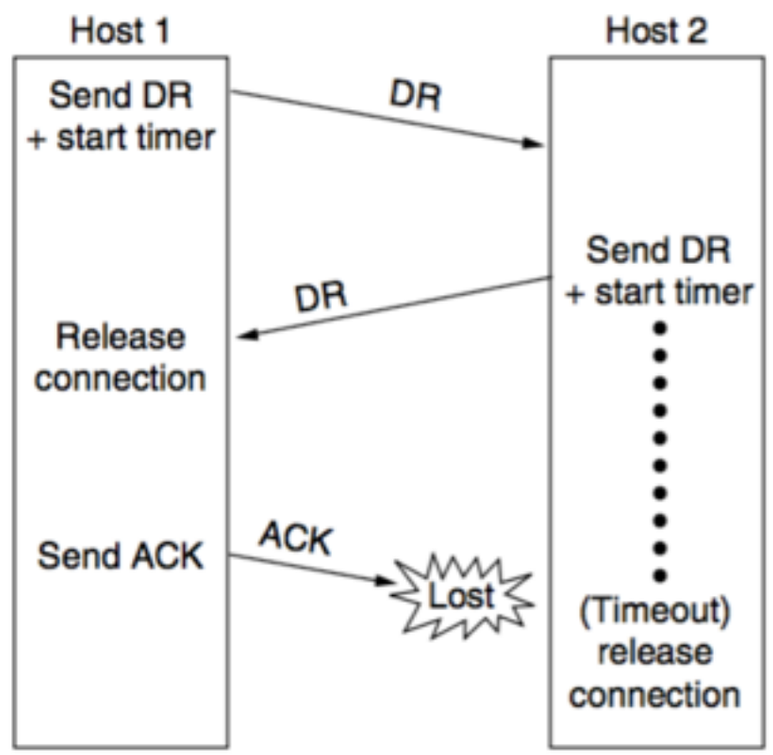
Extend the 32-bit sequence number so that it will not wrap within the maximum packet lifetime

Connection Release

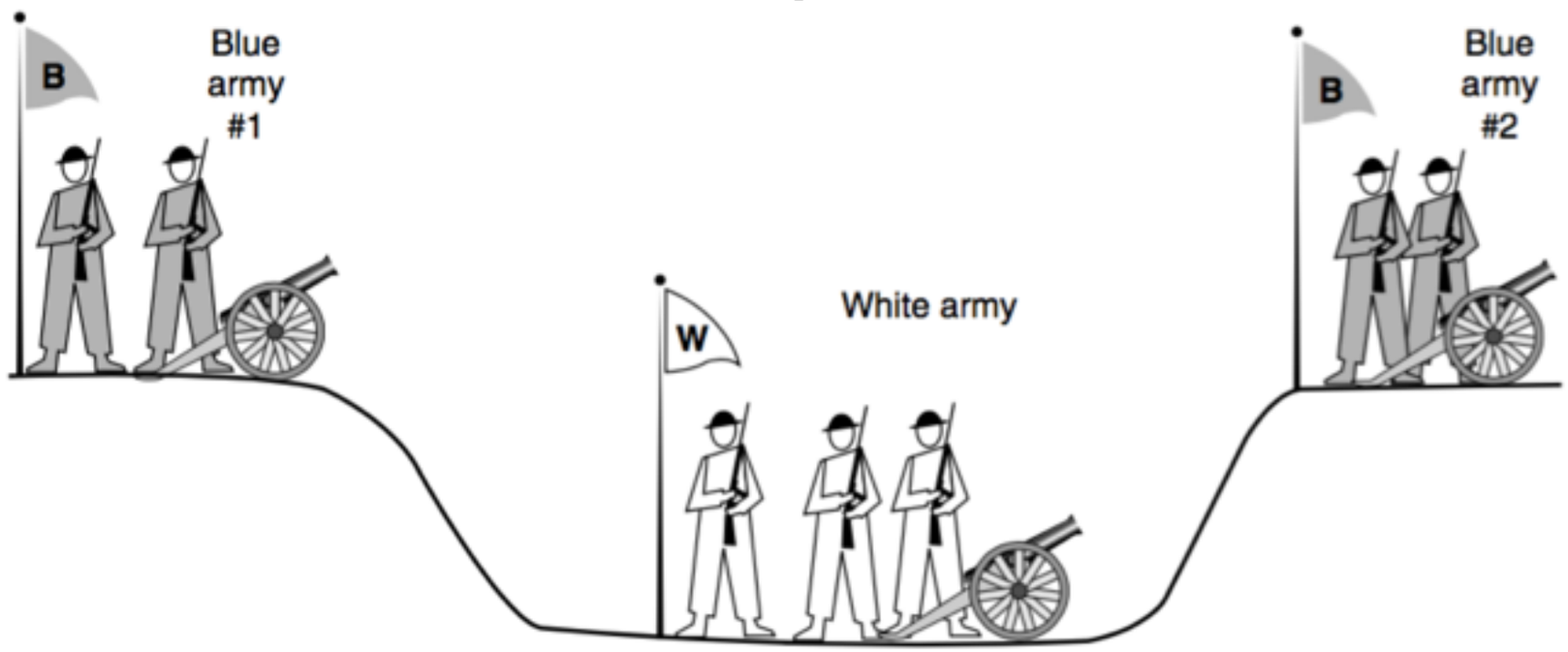
Asymmetric Release



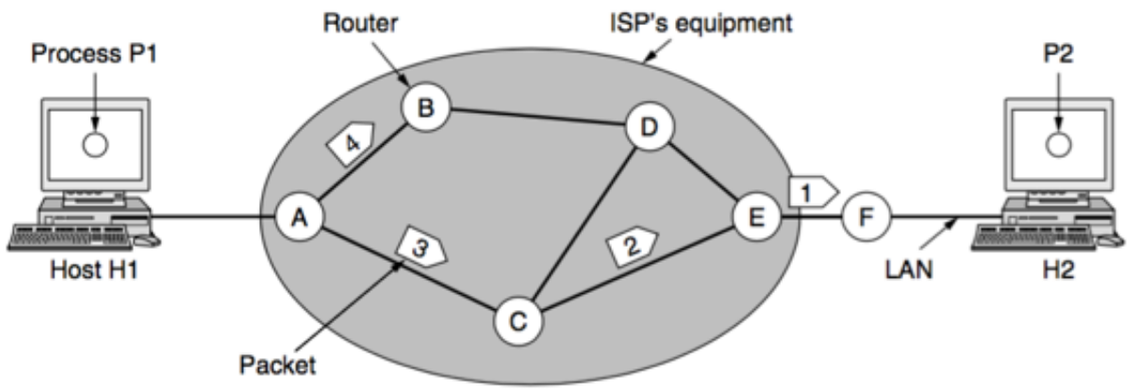
Symmetric Release

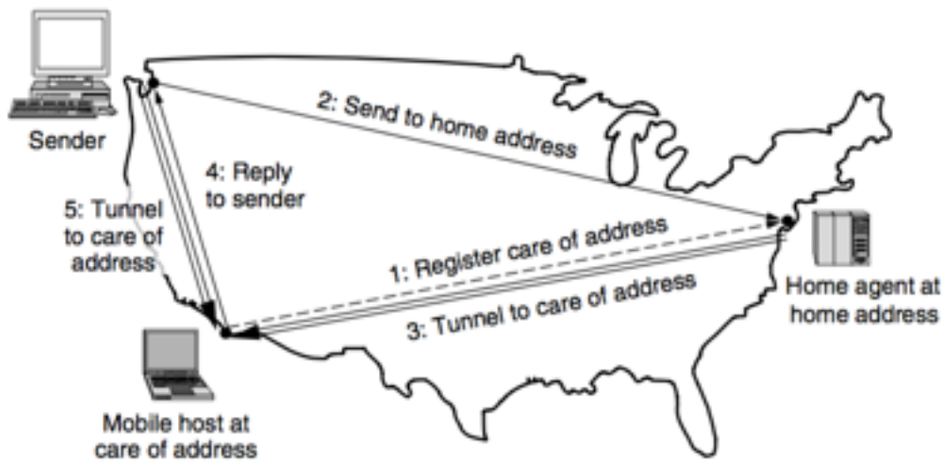


Two-army Problem

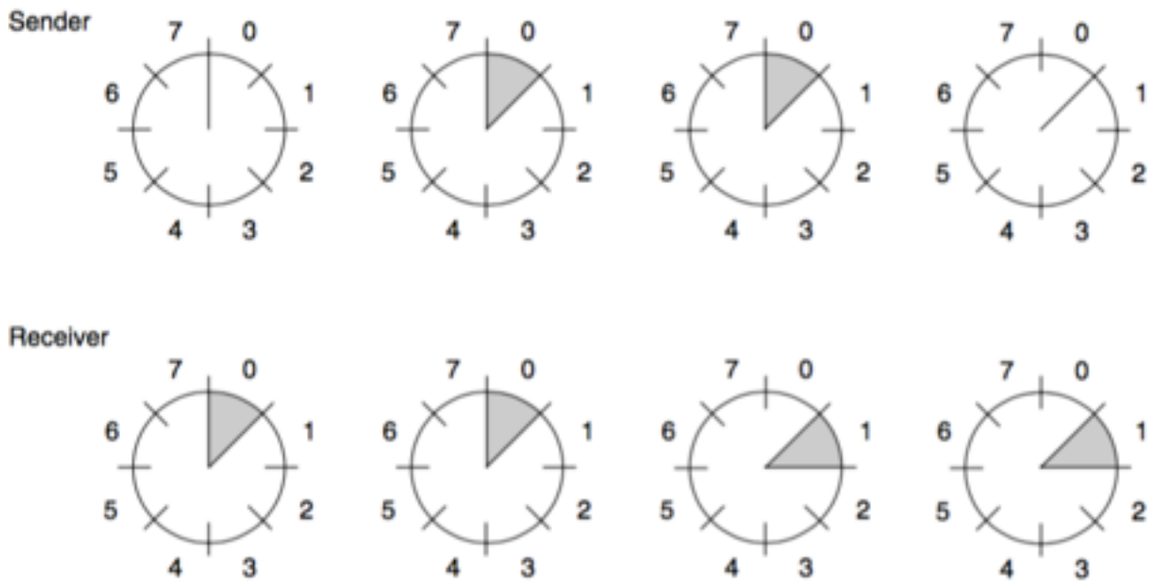


Error Correction





Sliding window



Data-link Layer

Hardware: router

Delay: 1μs <

Transport Layer

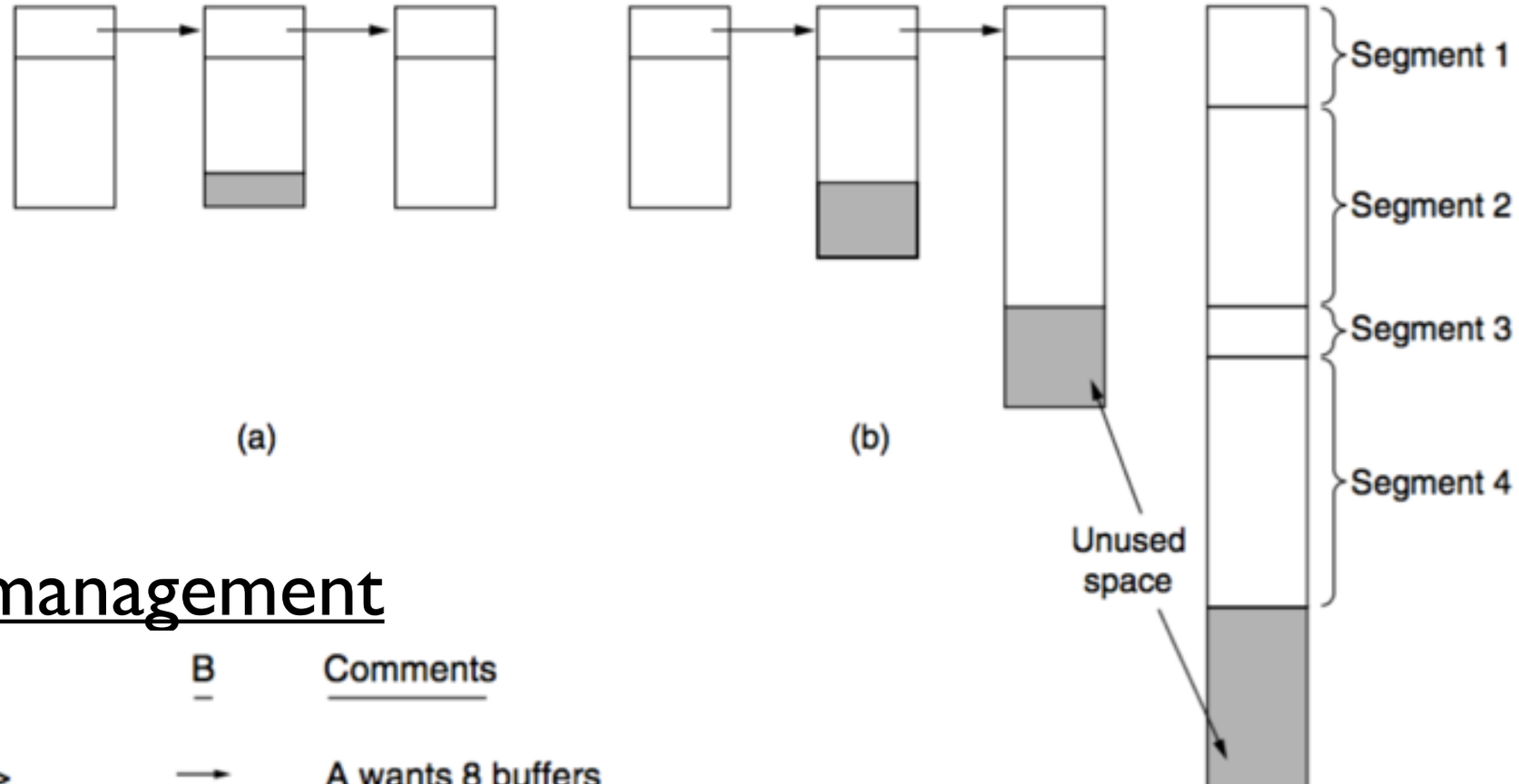
Hardware: host machine

Delay: 100ms >

Flow Control

Buffering

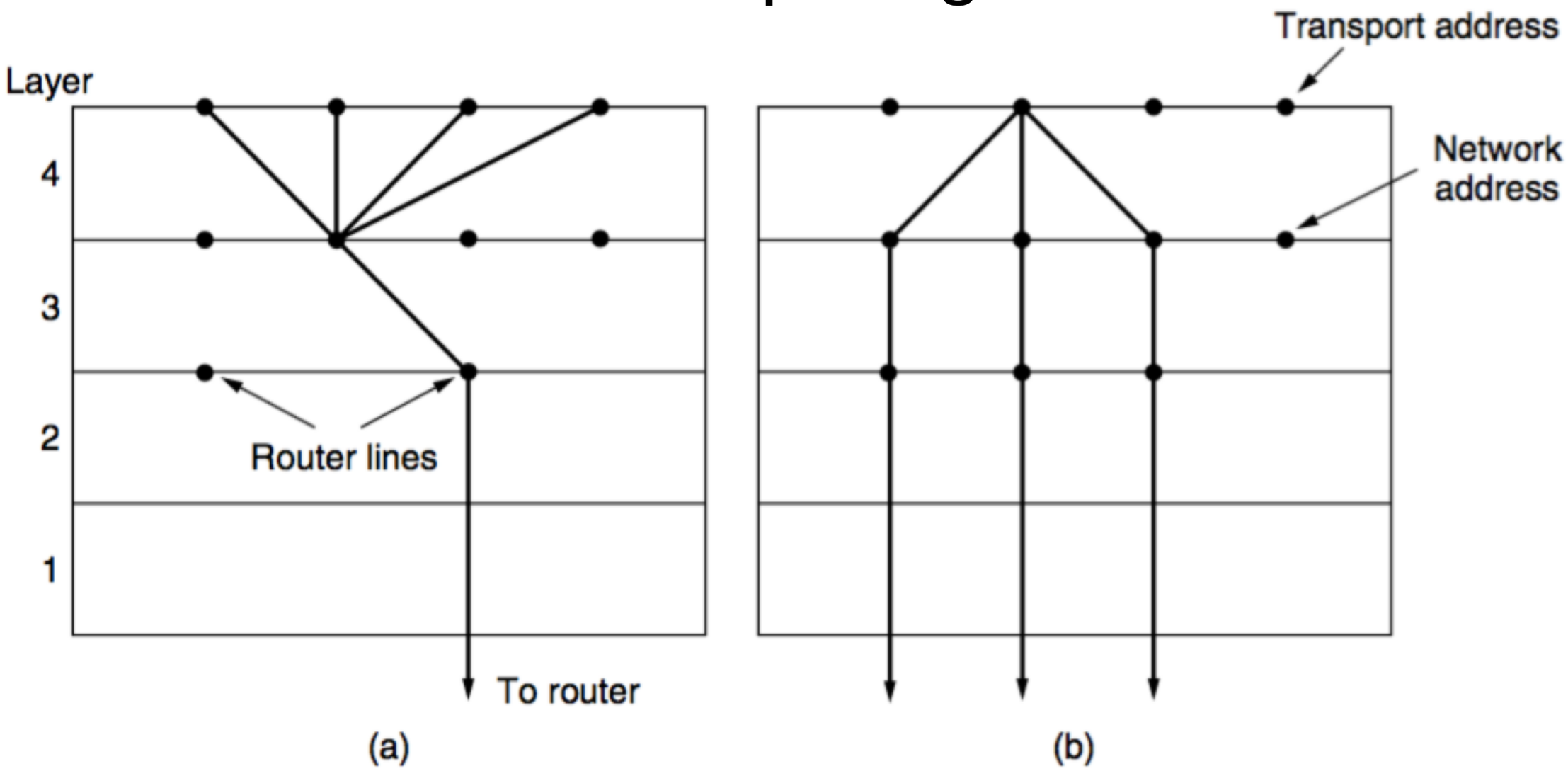
(a) fixed size
(b) variable size
(c) circular buffer



Dynamic window management

	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

Multiplexing



Multiplexing

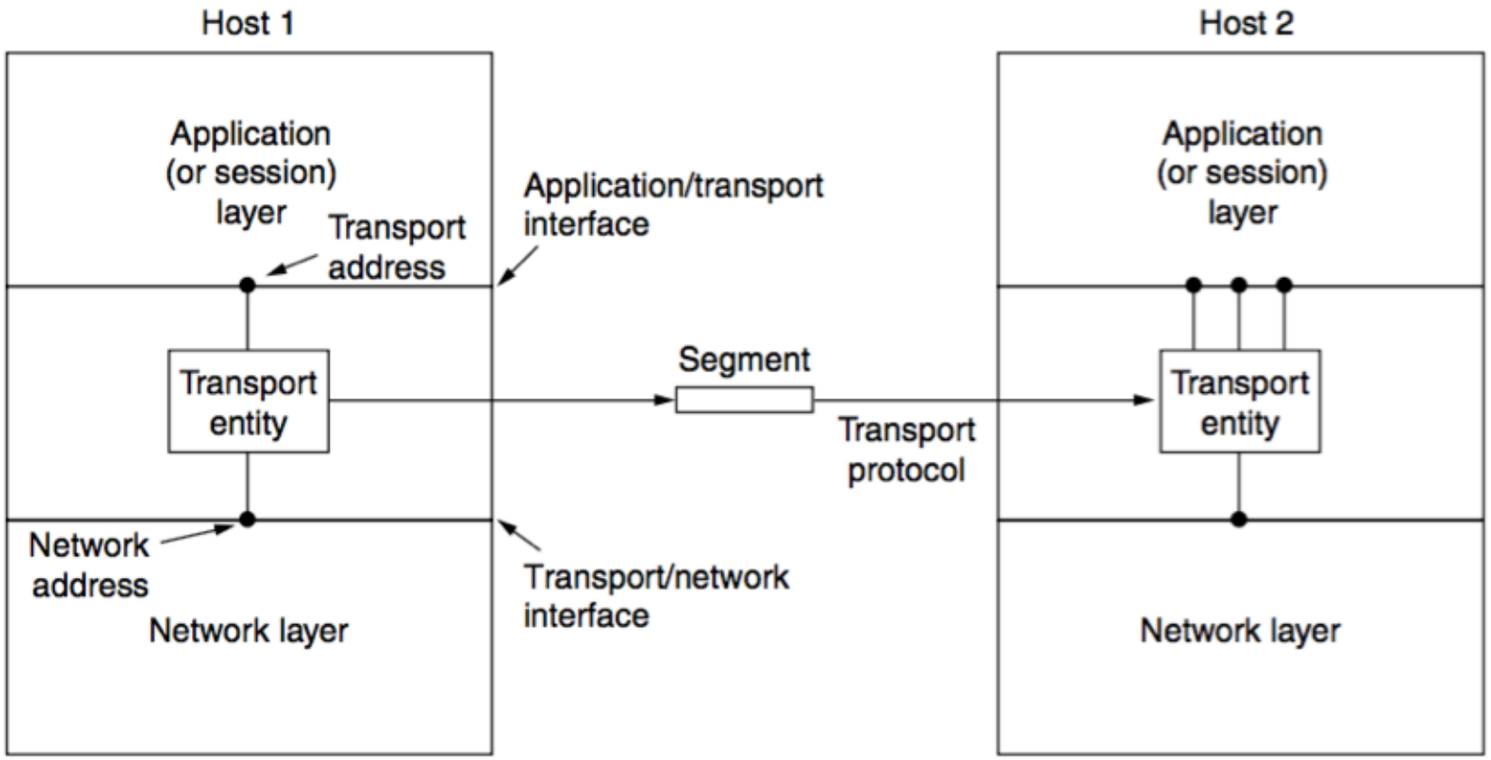
Inverse Multiplexing

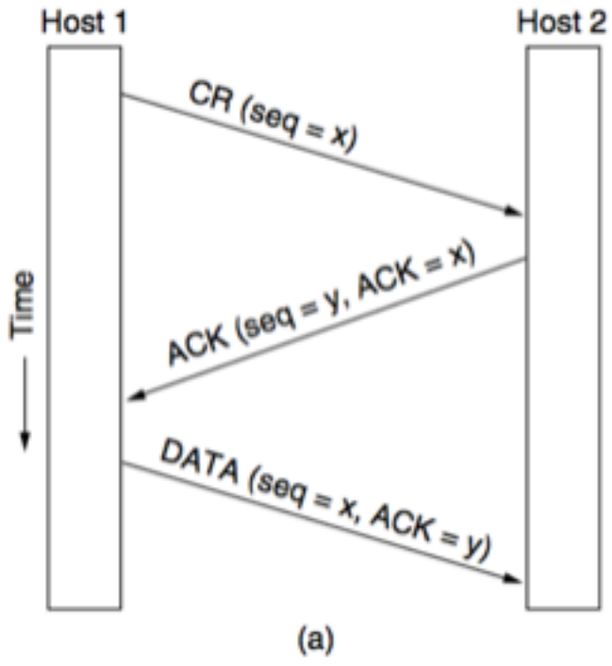
SCTP (Stream Control Transmission Protocol)

↕

TCP

Crash Recovery



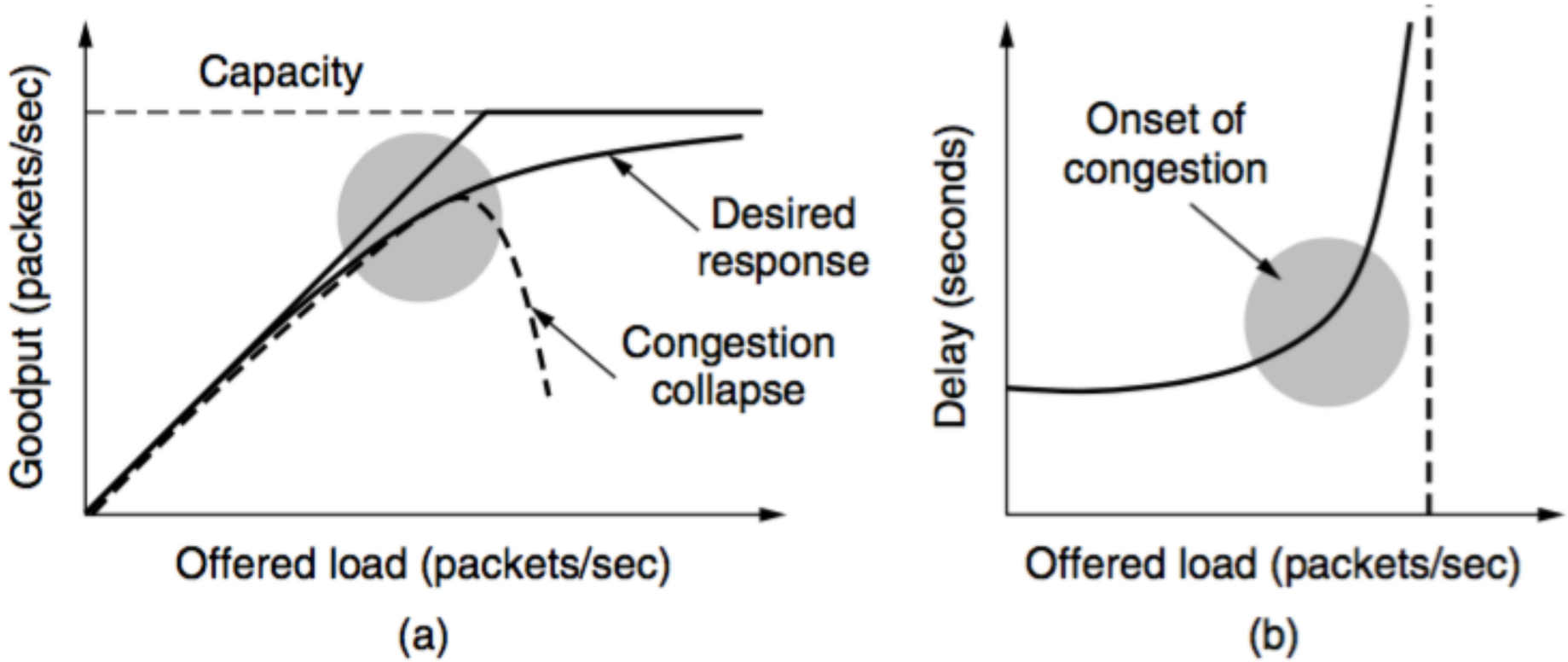


Strategy used by sending host	Strategy used by receiving host: First ACK, then write			First write, then ACK		
	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

S0 = no segments outstanding
S1 = one segment outstanding

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

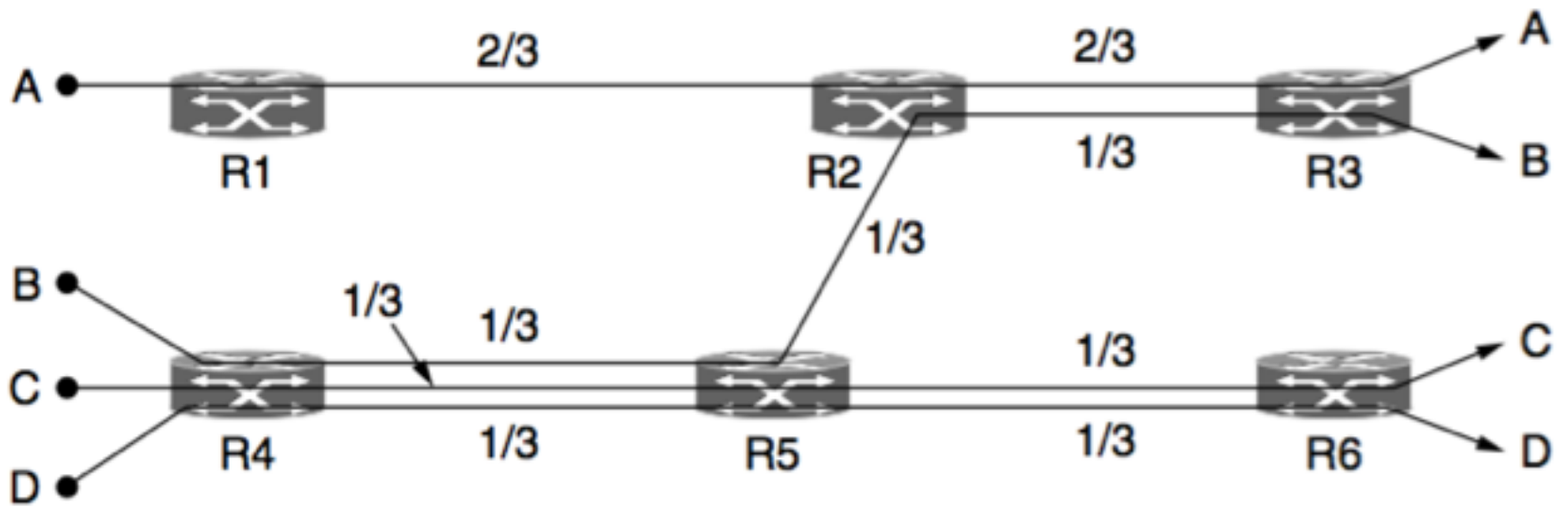
Congestion Control



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

The load with the highest power represents an efficient load for the transport entity to place on the network

Max-min fairness



Congestion Control

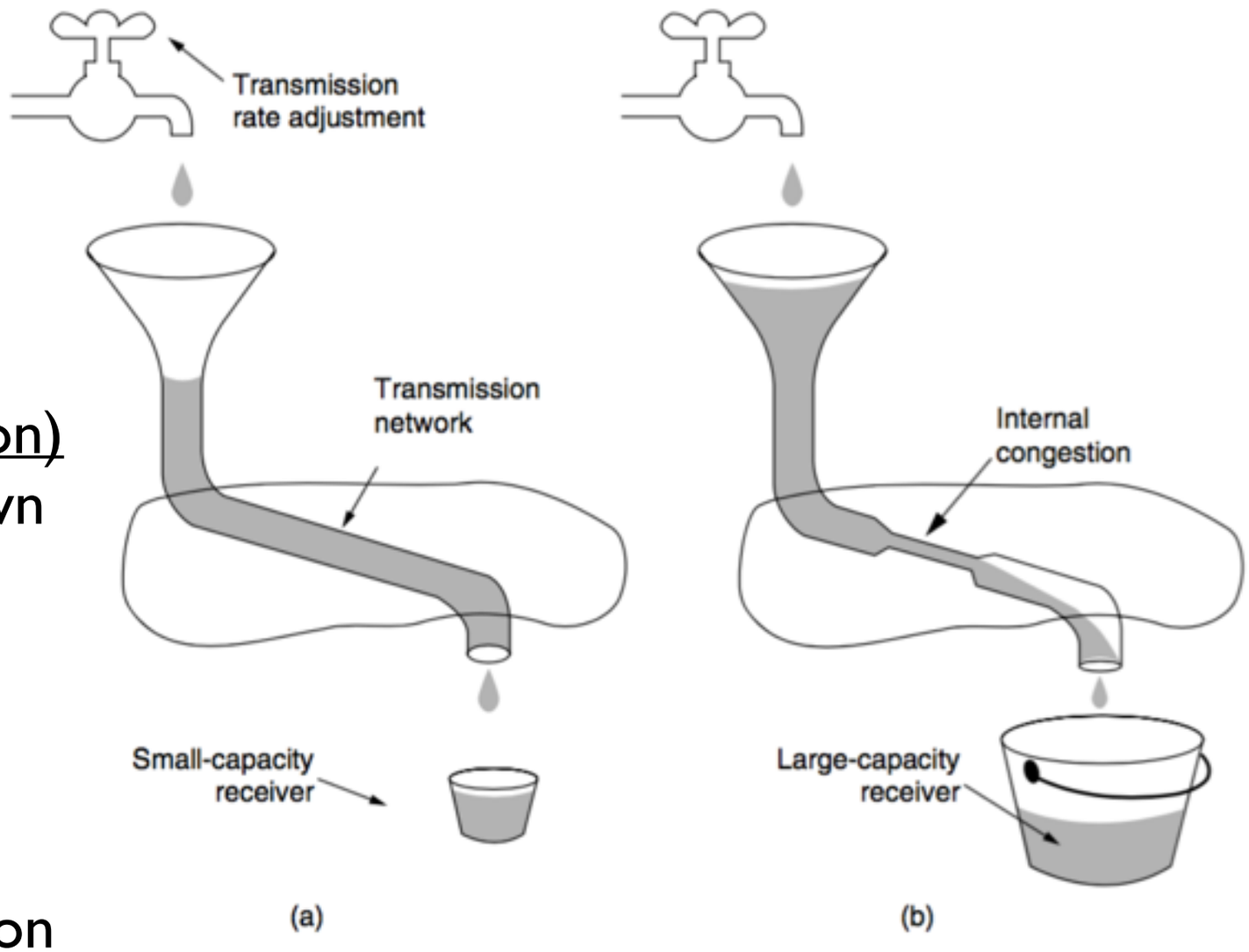
Regulating the sending rate

XCP (eXplicit Congestion Protocol)
Routers tell the sources the rate at which they may send

ECN (Explicit Congestion Notification)
Routers tell the sources to slow down

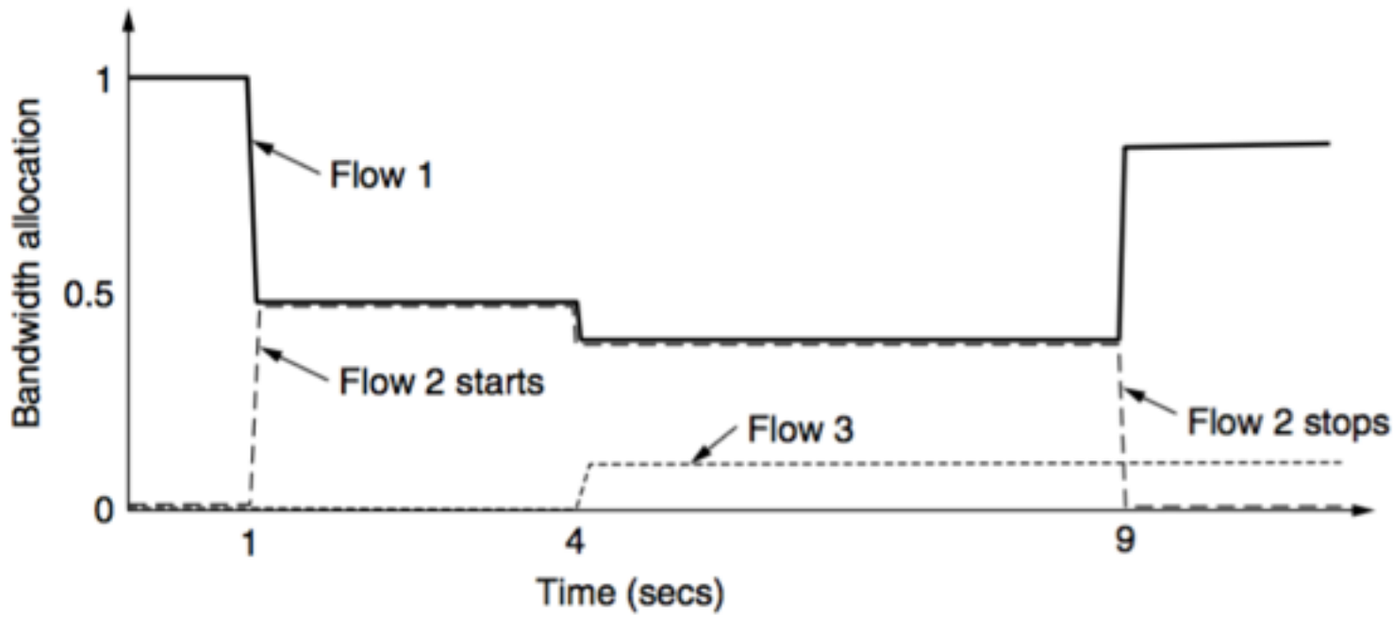
FAST TCP
Measures the roundtrip delay

TCP with RED routers
Packet loss is used to signal congestion



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
Compound TCP	Packet loss & end-to-end delay	No	Yes
CUBIC TCP	Packet loss	No	No
TCP	Packet loss	No	No

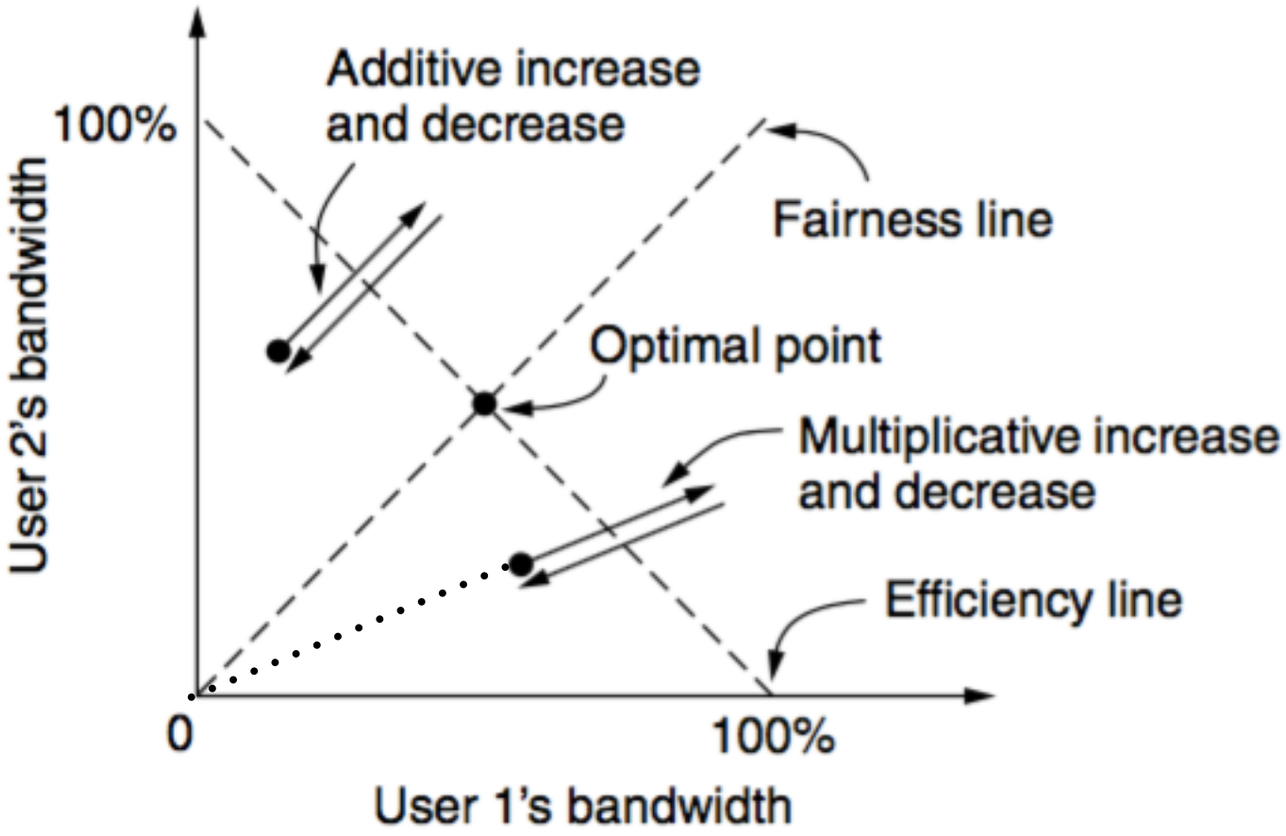
Control Law

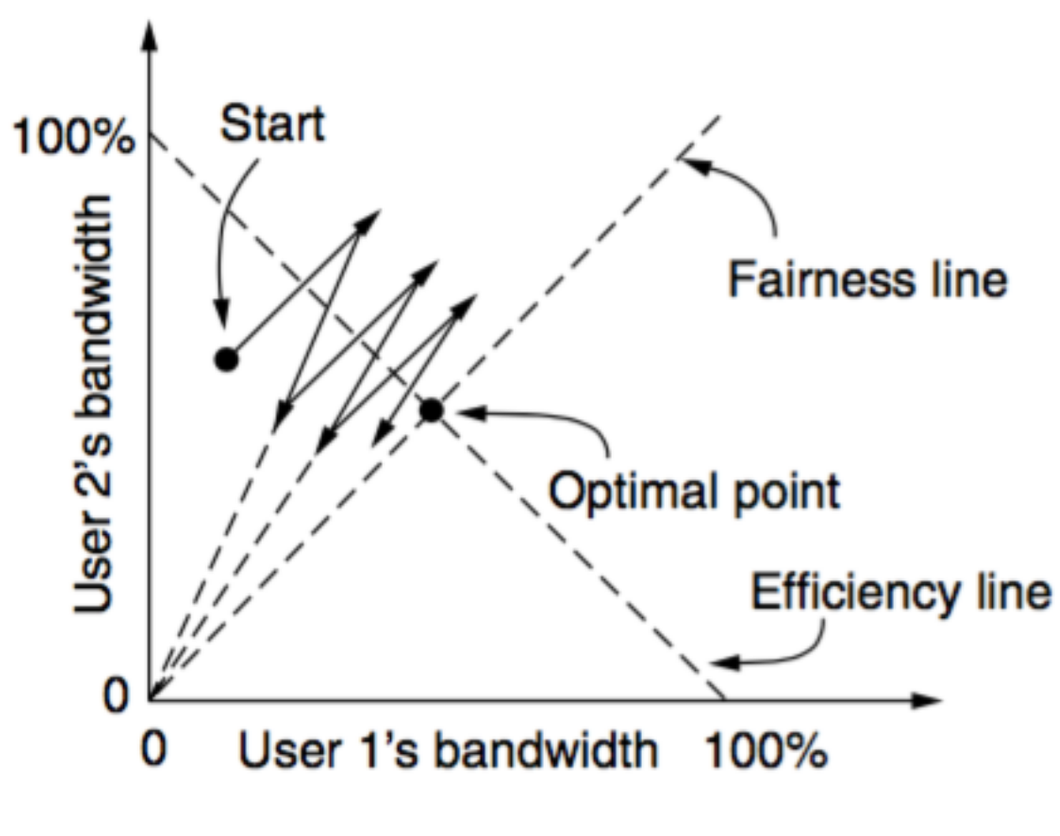


AIMD (Additive Increase Multiplicative Decrease)

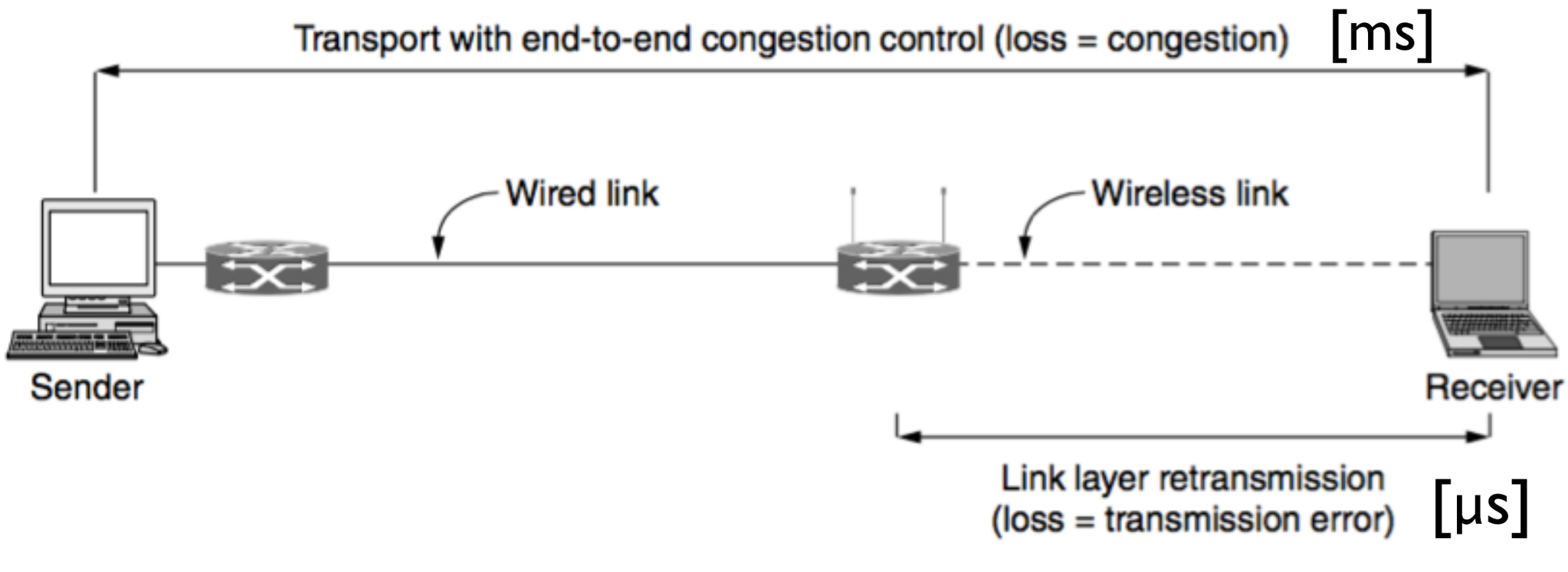
Additive: y=x+b

Multiplicative: y=ax



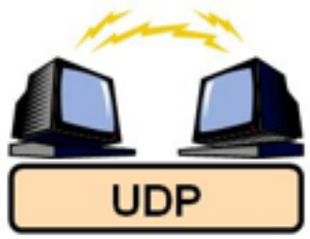


Wireless Issues

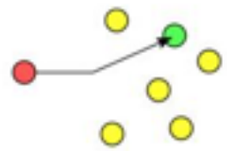


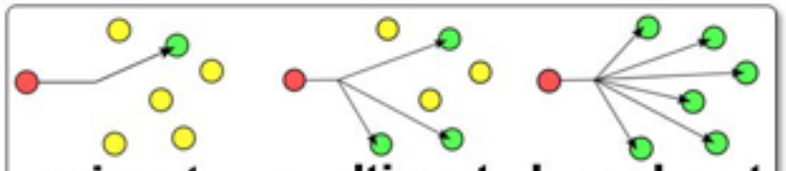
Internet Transport Protocols

- **Slower but reliable transfers**
- **Typical applications:**
  - Email
  - Web browsing

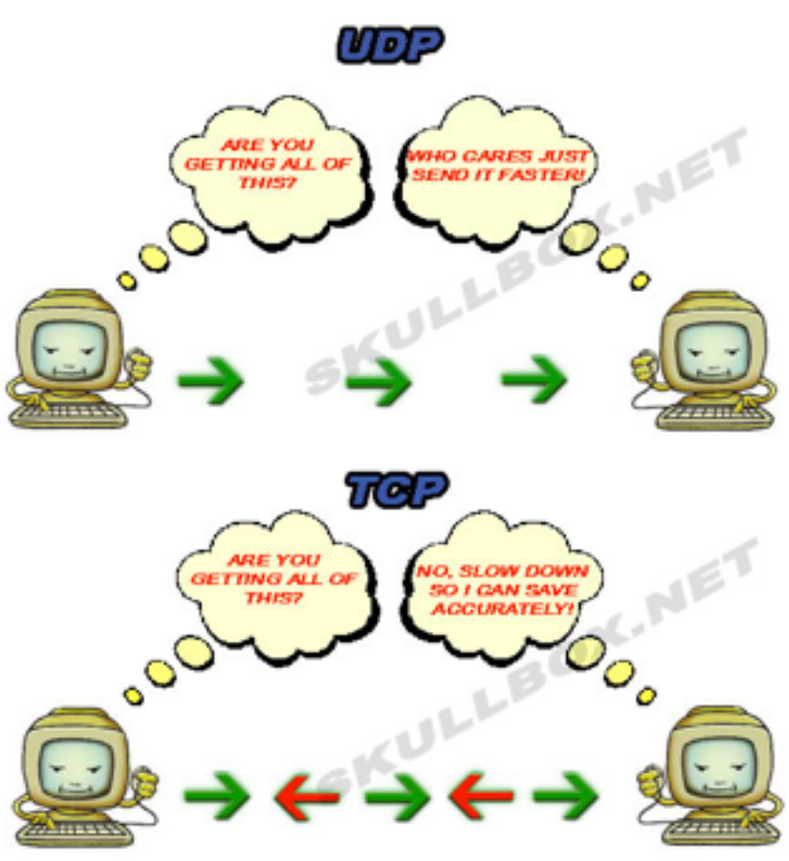


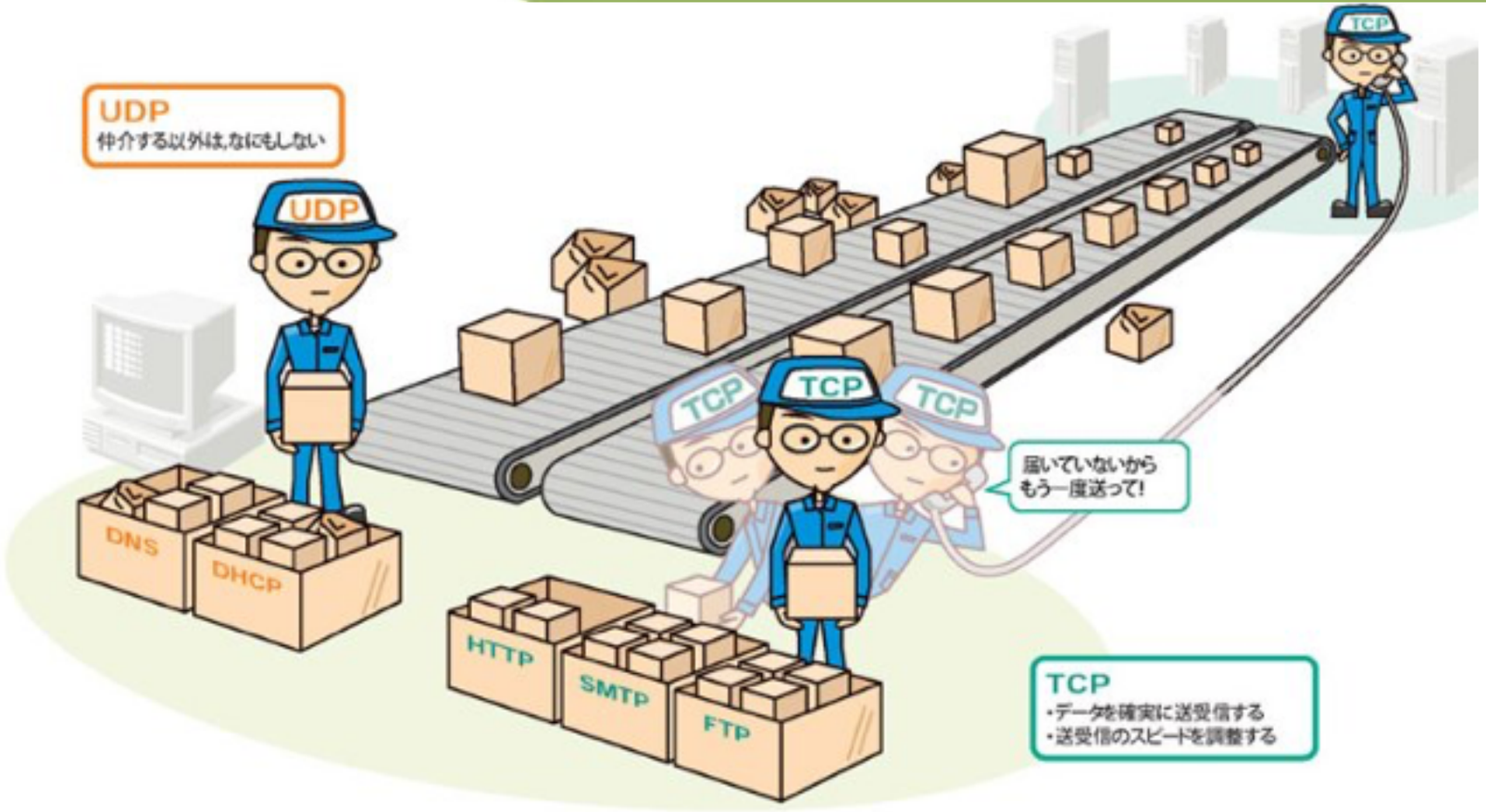
- **Fast but non-guaranteed transfers ("best effort")**
- **Typical applications:**
  - VoIP
  - Music streaming





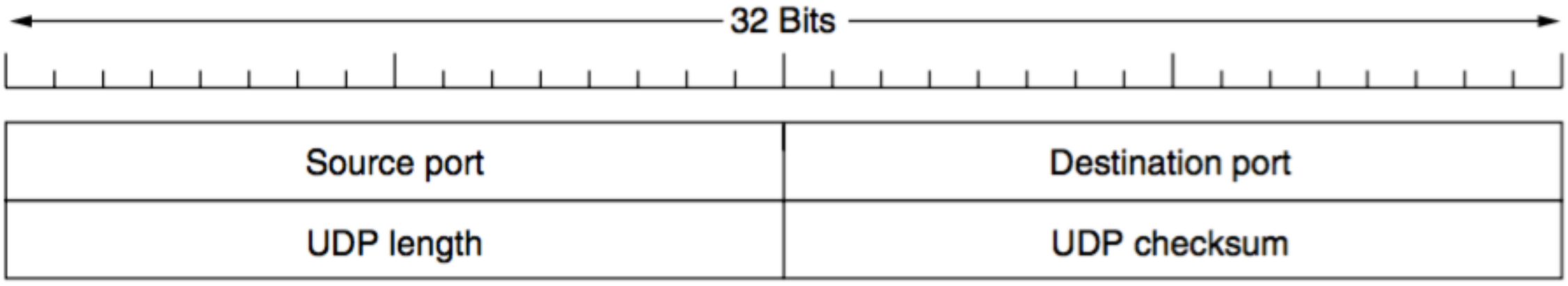
TCP	UDP
Reliable	Unreliable
Connection-oriented	Connectionless
Segment retransmission and flow control through windowing	No windowing or retransmission
Segment sequencing	No sequencing
Acknowledge segments	No acknowledgement



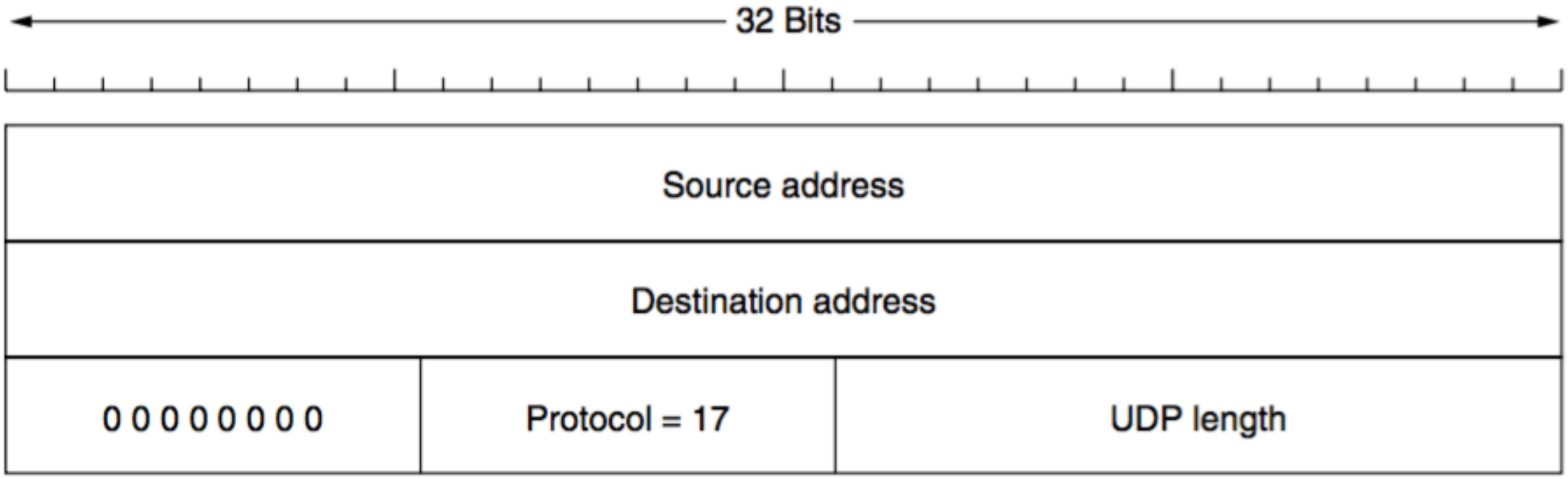


UDP (User Datagram Protocol)

Header

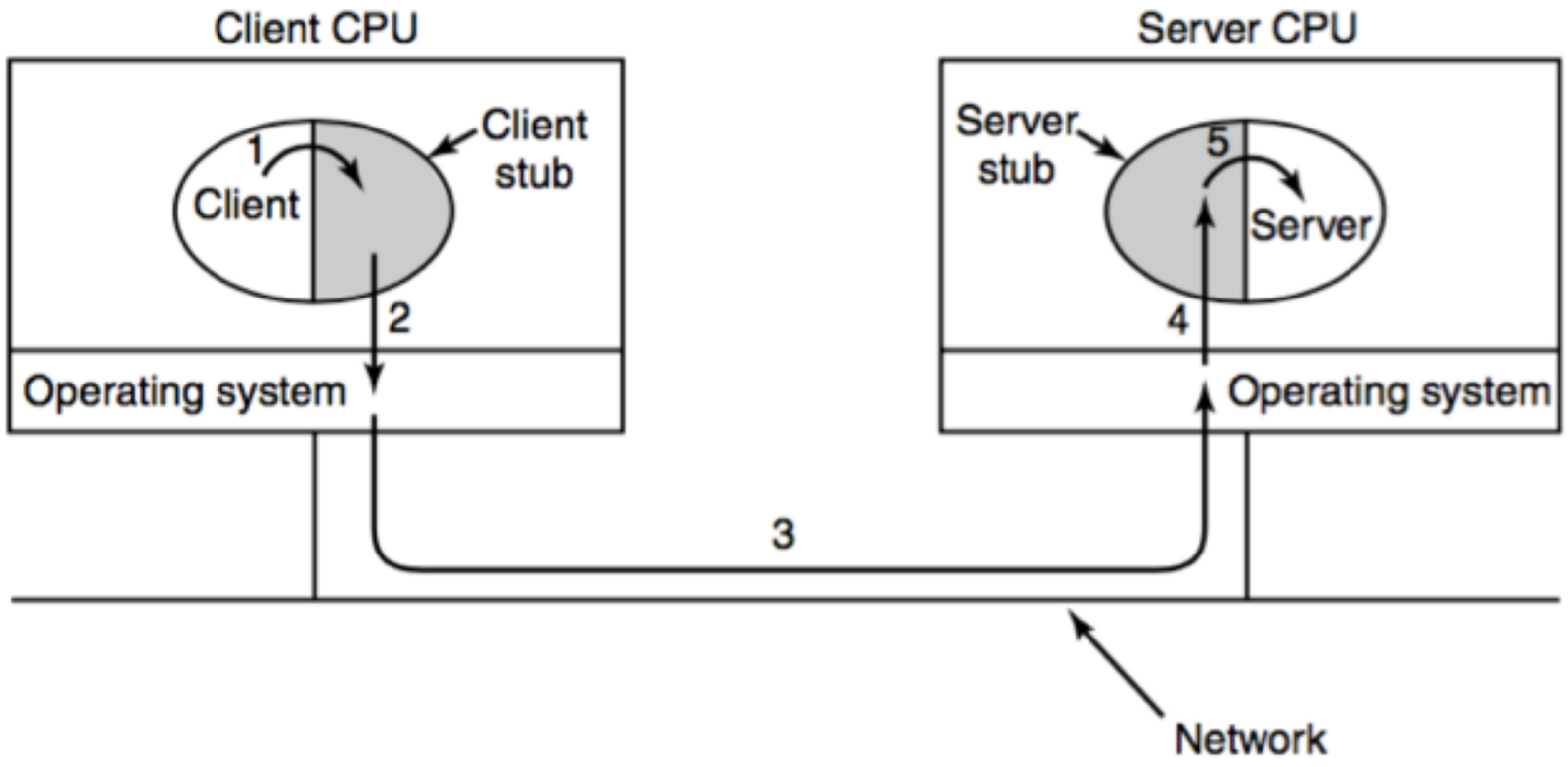


IPv4 Pseudoheader



1. Demultiplexing multiple processes using the ports
2. Optional end-to-end error detection

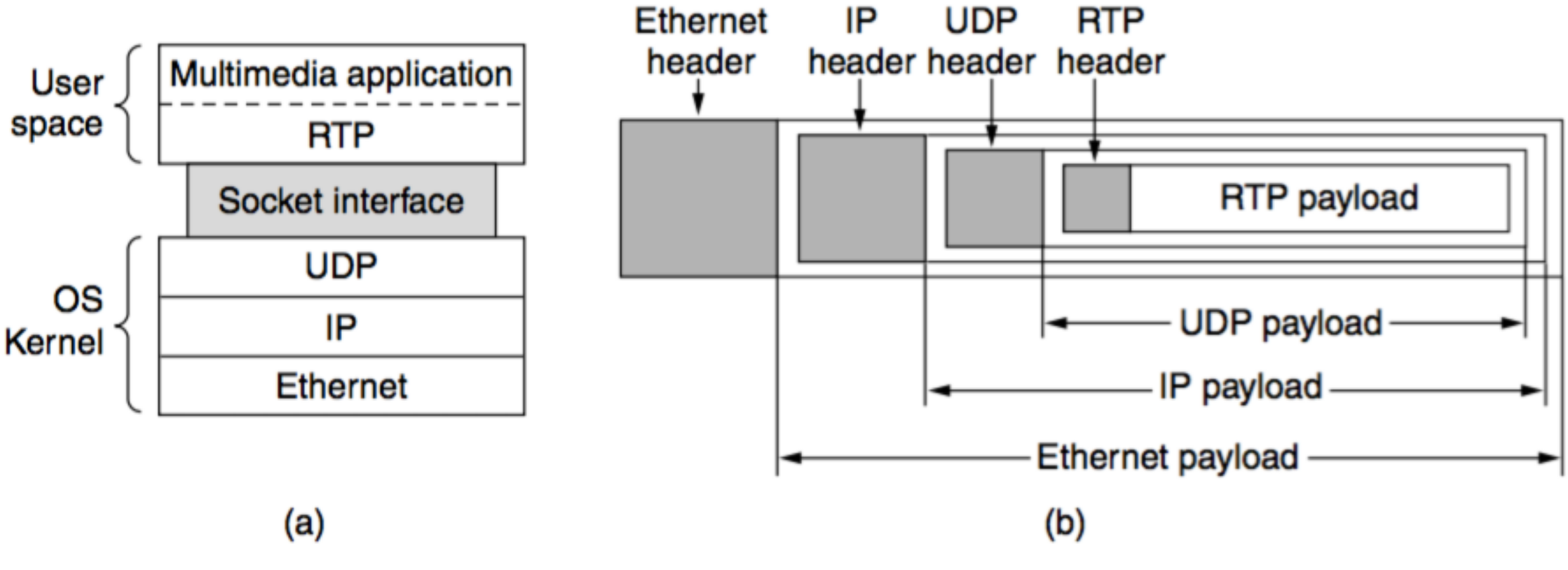
Remote Procedure Call



Limitations

1. Cannot pass pointers
2. Cannot pass arrays without specifying the size
3. Cannot call function with arbitrary parameters (*printf*)
4. Cannot use global variables
5. Operations must be idempotent (DNS)

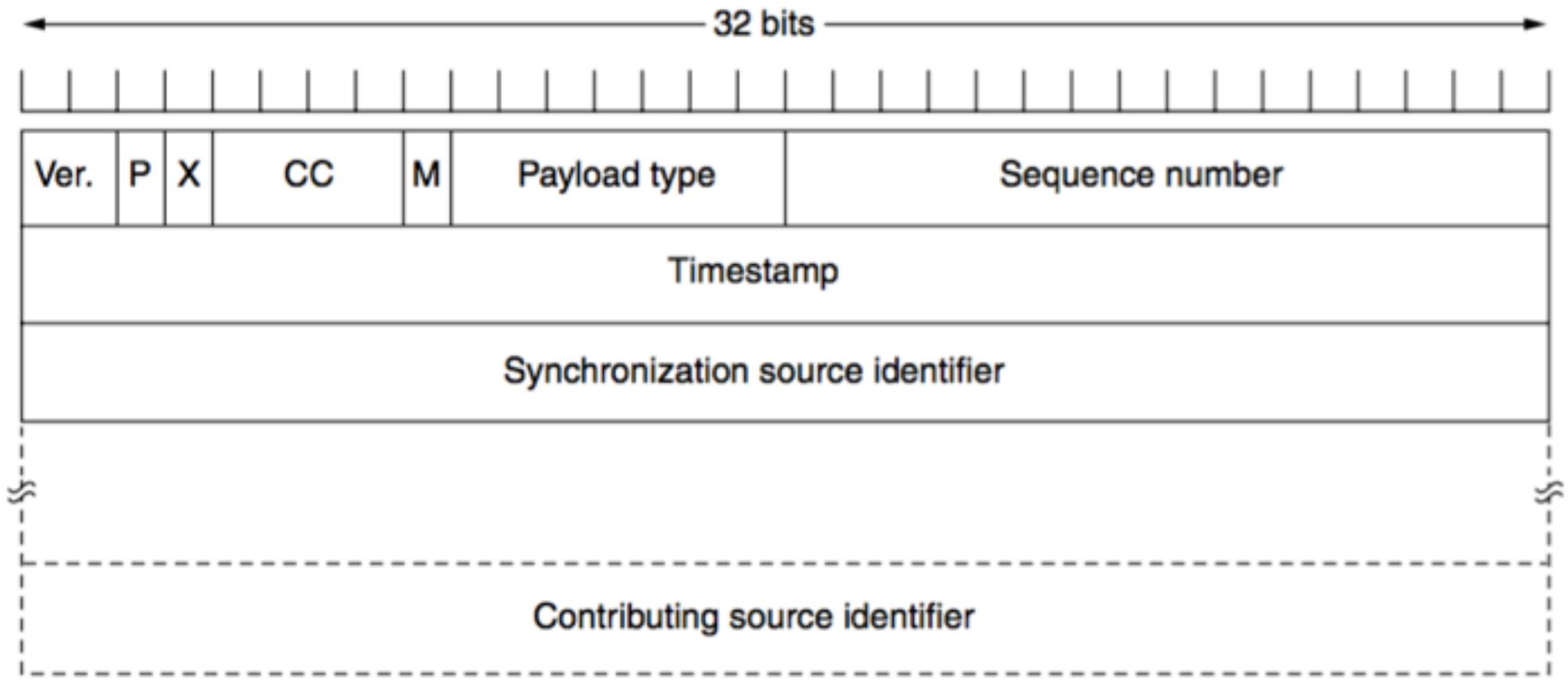
Realtime Transport Protocol



Service

1. Multiplex several real-time data streams onto a single stream of UDP packets
2. UDP stream can be sent to a single destination (unicasting) or to multiple destinations (multicasting)
3. Packets may be lost, delayed, corrupted
4. Retransmission is not a practical option since the retransmitted packet would probably arrive too late to be useful

Realtime Transport Protocol



Ver. : Version (currently at 2)
P: Indicates that the packet has been padded
X: Indicates that an extension header is present
CC: How many contributing sources are present
M: Application-specific marker bit
Payload type: Encoding algorithm (e.g., uncompressed 8-bit audio, MP3, etc.)
Sequence number: Used to detect lost packets
Timestamp: Reduce timing variability of streaming multimedia
Synchronization Source Identifier: Which stream the packet belongs to
Contributing Source Identifier: Streams being mixed are listed here

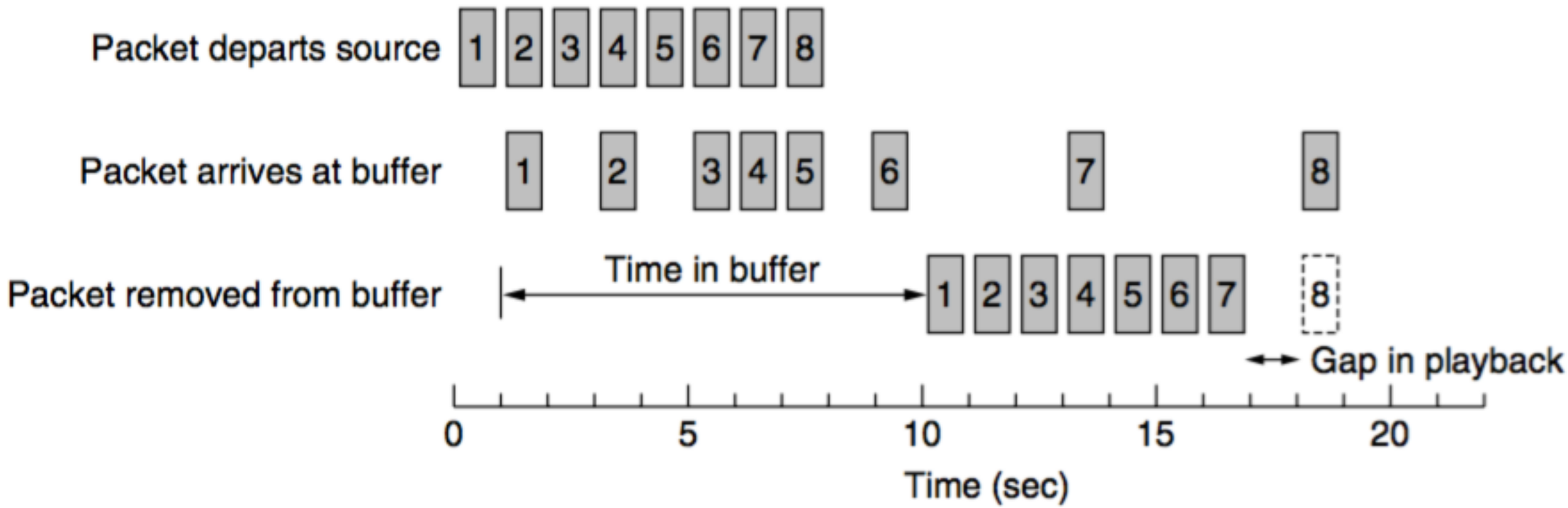
RTCP—The Real-time Transport Control Protocol

Provide feedback on delay, variation in delay or jitter, bandwidth, congestion, and other network properties to the sources.

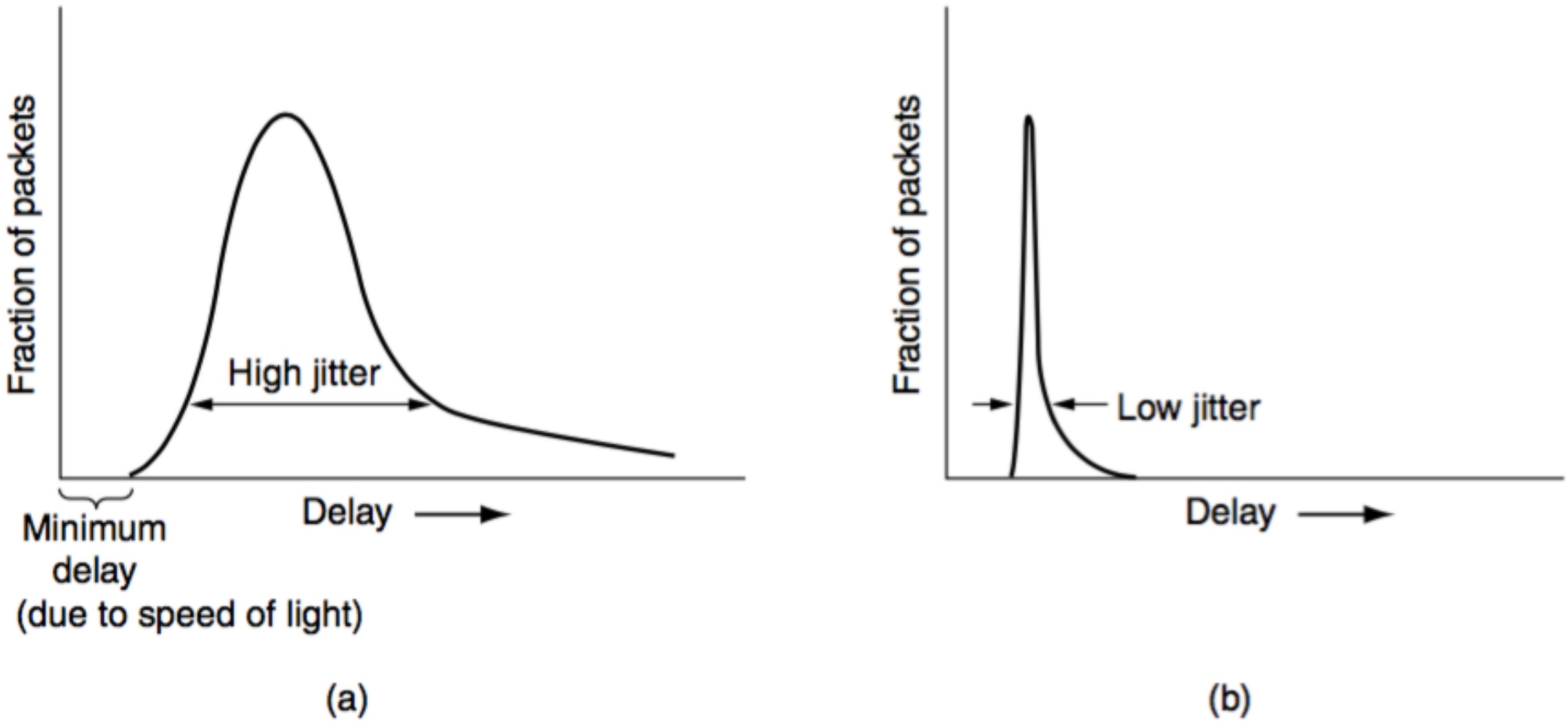
This information can be used by the encoding process to increase the data rate (and give better quality) when the network is functioning well and to cut back the data rate when there is trouble in the network.

For example, if the bandwidth increases or decreases during the transmission, the encoding may switch from MP3 to 8-bit PCM to delta encoding as required.

Playout with Buffering and Jitter Control



Playout with Buffering and Jitter Control



For a streaming audio or video player, buffers of about 10 seconds are often used to ensure that the player receives all of the packets (that are not dropped in the network) in time.

For live applications like videoconferencing, short buffers are needed for responsiveness.

講義日程（2Q）

		授業計画		課題
06/14	第９回	ネットワーク層１ ルーティング・輻輳制御	５章	ルーティングの種類を理解し 輻輳制御手法を説明できる
06/21	第１０回	ネットワーク層２ インターネットとサービス品質	５章	インターネットの制御プロトコルを理解し ネットワーク間の接続について説明できる
06/28	第１１回	トランスポート層１ トランスポート・プロトコルの要素	６章	誤り制御とフロー制御を理解し 輻輳制御について説明できる
07/05	第１２回	トランスポート層２ UDP と TCP	６章	TCP の信頼性を理解し TCP のコネクション管理を説明できる
07/12	第１３回	アプリケーション層 DNS, 電子メール, www	７章	DNS, 電子メール, www のしくみを理解し ストリーミング，P2P について説明できる
07/26	第１４回	ネットワークセキュリティ１ 対称鍵暗号，公開鍵暗号	８章	暗号アルゴリズムを理解し SHA-1,2 と RSA について説明できる
08/02	第１５回	ネットワークセキュリティ２ デジタル署名，認証プロトコル	８章	電子メール,Web のセキュリティ の脅威について把握できる