Chapter 06
Transport Layer

Associate Prof. Hong Zheng（郑宏）
Computer School
Beijing Institute of Technology

Key Points

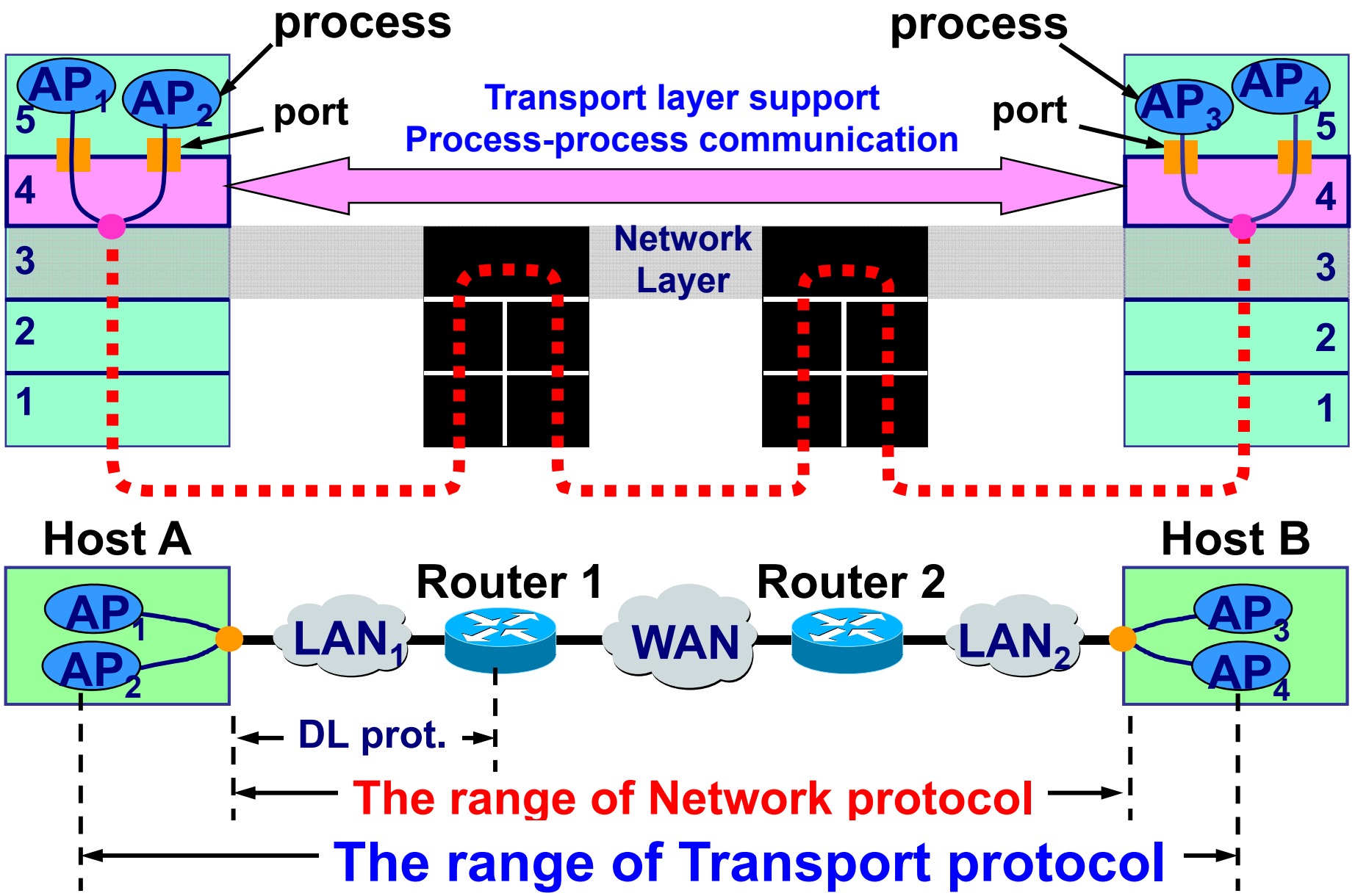
功能及机制	服务，连接建立与释放，TSAP	掌握
Internet Transport Layer	Port, Socket, Multiplexing	熟练掌握
UDP	Services, datagram format, checksum, Port, Socket	熟练掌握
TCP	Services, segment format, checksum, Port, Socket	熟练掌握
	Connection Management (3-way handshake & half close), Reliable Data transmission (Error control, Flow control, sliding window), Congestion control (AIMD, Slow Start, Congestion Avoidance, Fast retransmission, Fast recovery)	

Chapter 6: Roadmap

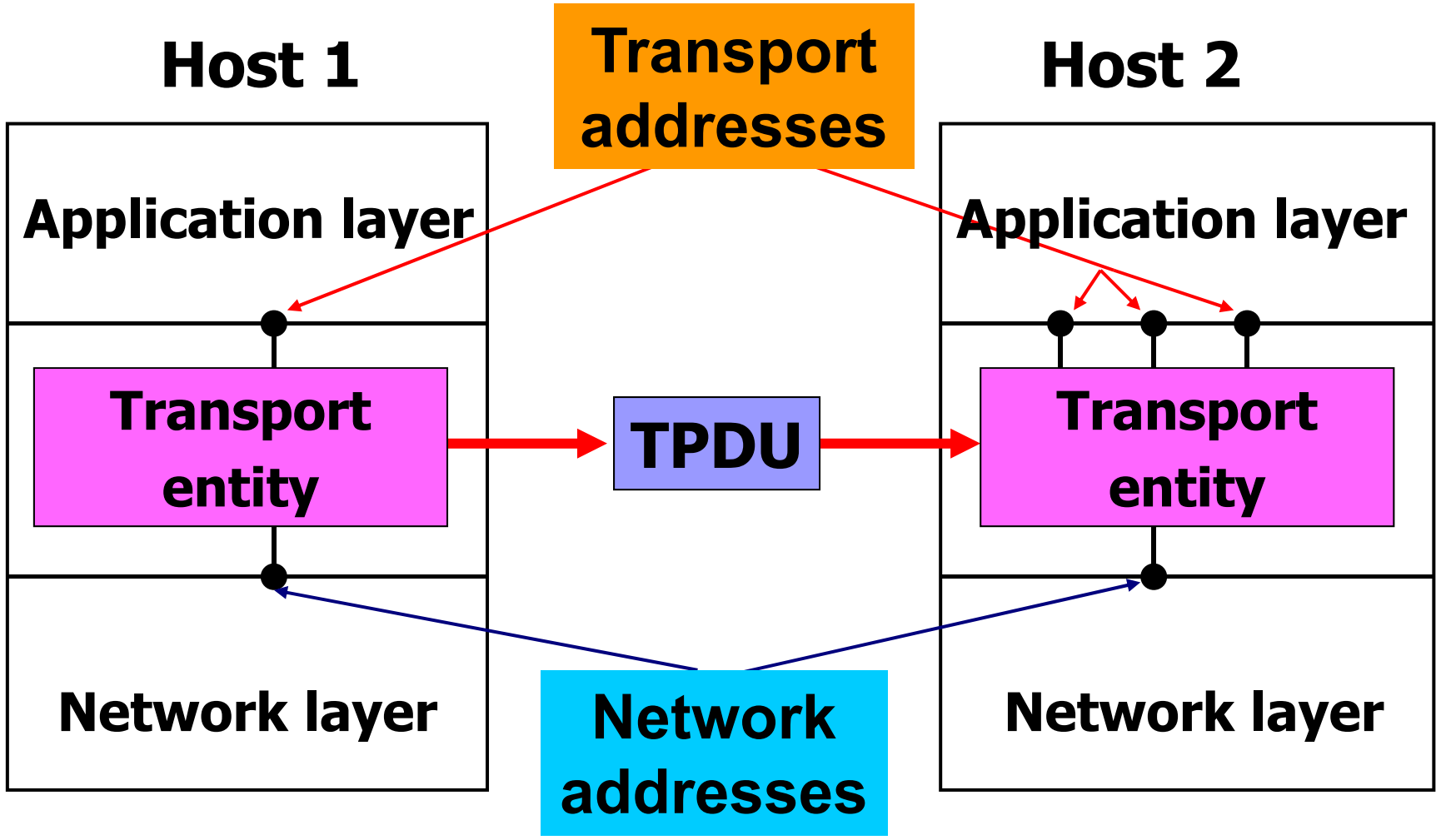
- 6.1 Transport Services
- 6.2 Elements of transport protocol
- 6.3 Simple transport protocol
- 6.4 The Internet Transport Layer
  - UDP
  - TCP

Transport Layer Overview

- Transport layer acts as a liaison between the upper-layer protocols and the lower-layer protocols.
- Transport layer is responsible for process-to-process (end-to-end) communication
  - Support logical communication between processes
  - relies on, enhances, network layer services


process
process
AP1
AP2
AP3
AP4
port
port
Transport layer support
Process-process communication
5
4
3
2
1
Network
Layer
Host A
Host B
Router 1
Router 2
LAN1
WAN
LAN2
DL prot.
The range of Network protocol
The range of Transport protocol

Transport Layer Overview



Transport Services

- To upper layer
- efficient, reliable, cost-effective service
- 2 types:
  - connection-oriented services
    - Establishment
    - Data transfer
    - Release
  - unreliable connectionless services

Transport Service Primitives

- **Simple primitives:**
  - **connect**
  - **send**
  - **receive**
  - **disconnect**
- **Q: How to handle incoming connection request in server process?**
  - **Wait for connection request from client!**
  - **listen**

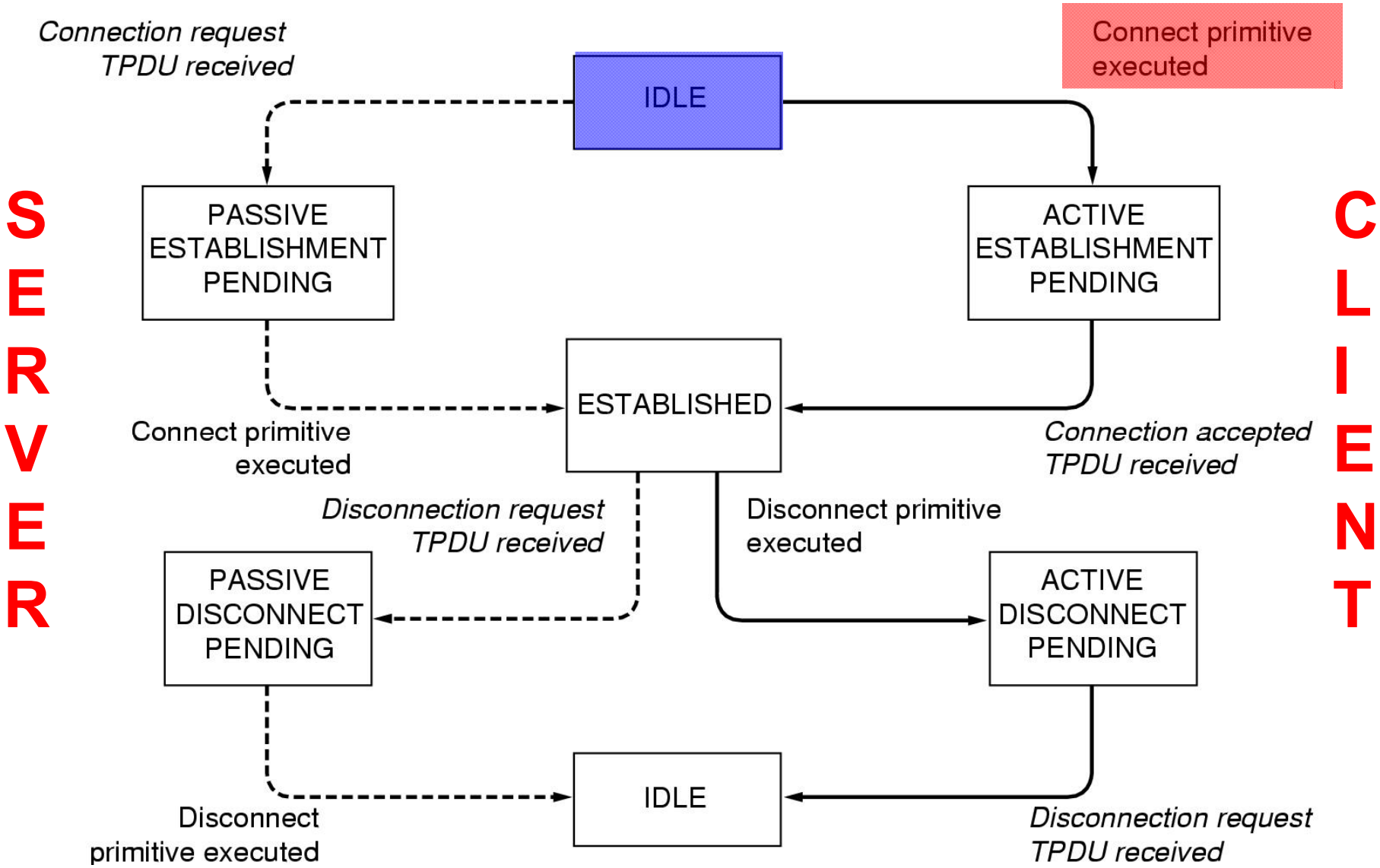
Simple service: primitives

listen	**Wait till a process wants a connection**
connect	**Try to setup a connection**
send	**Send data packet**
receive	**Wait for arrival of data packet**
disconnect	**Calling side breaks up the connection**

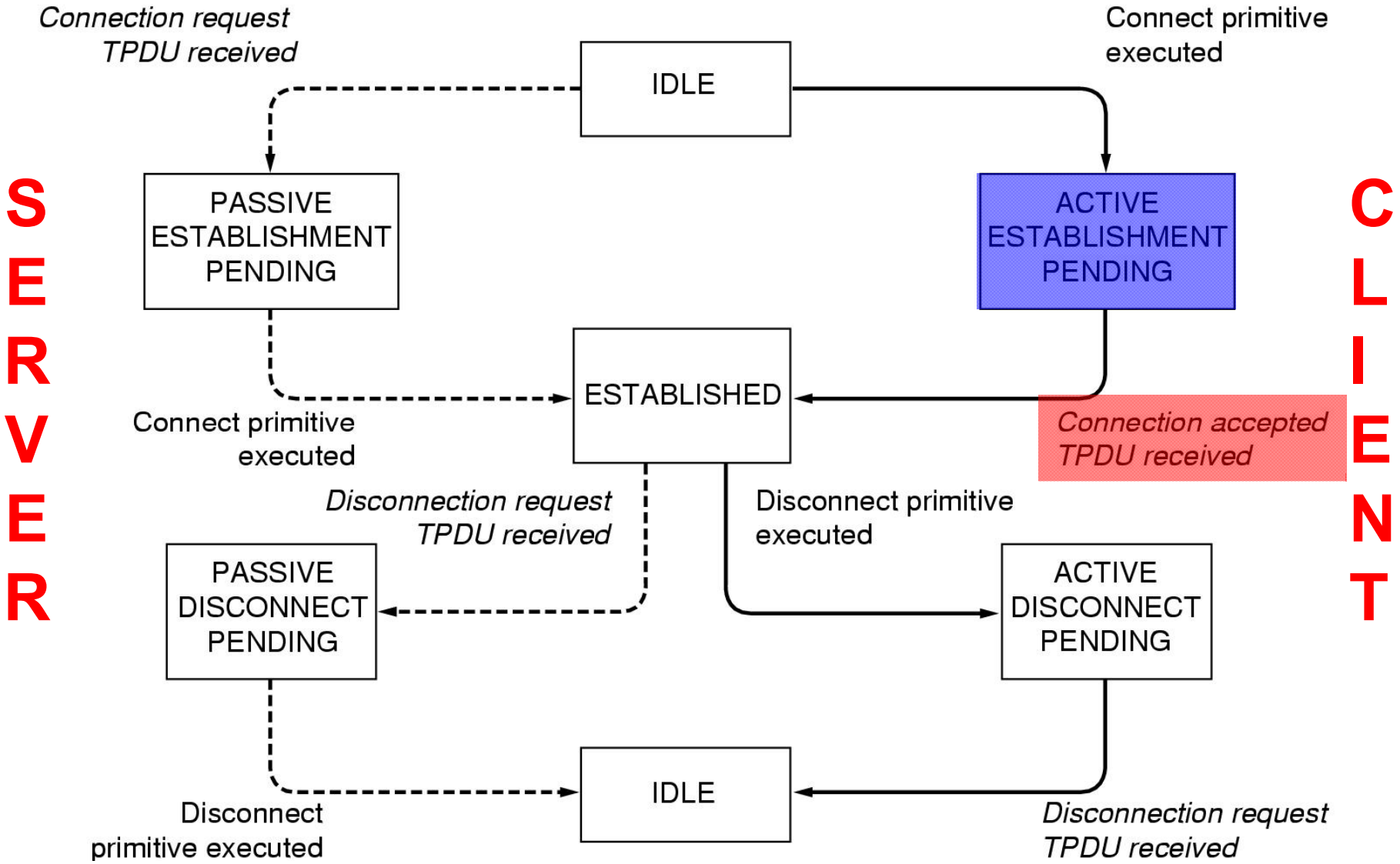
Simple service: primitives

listen	No TPDU
connect	Connection Request TPDU
send	Data TPDU
receive	No TPDU
disconnect	Disconnect TPDU

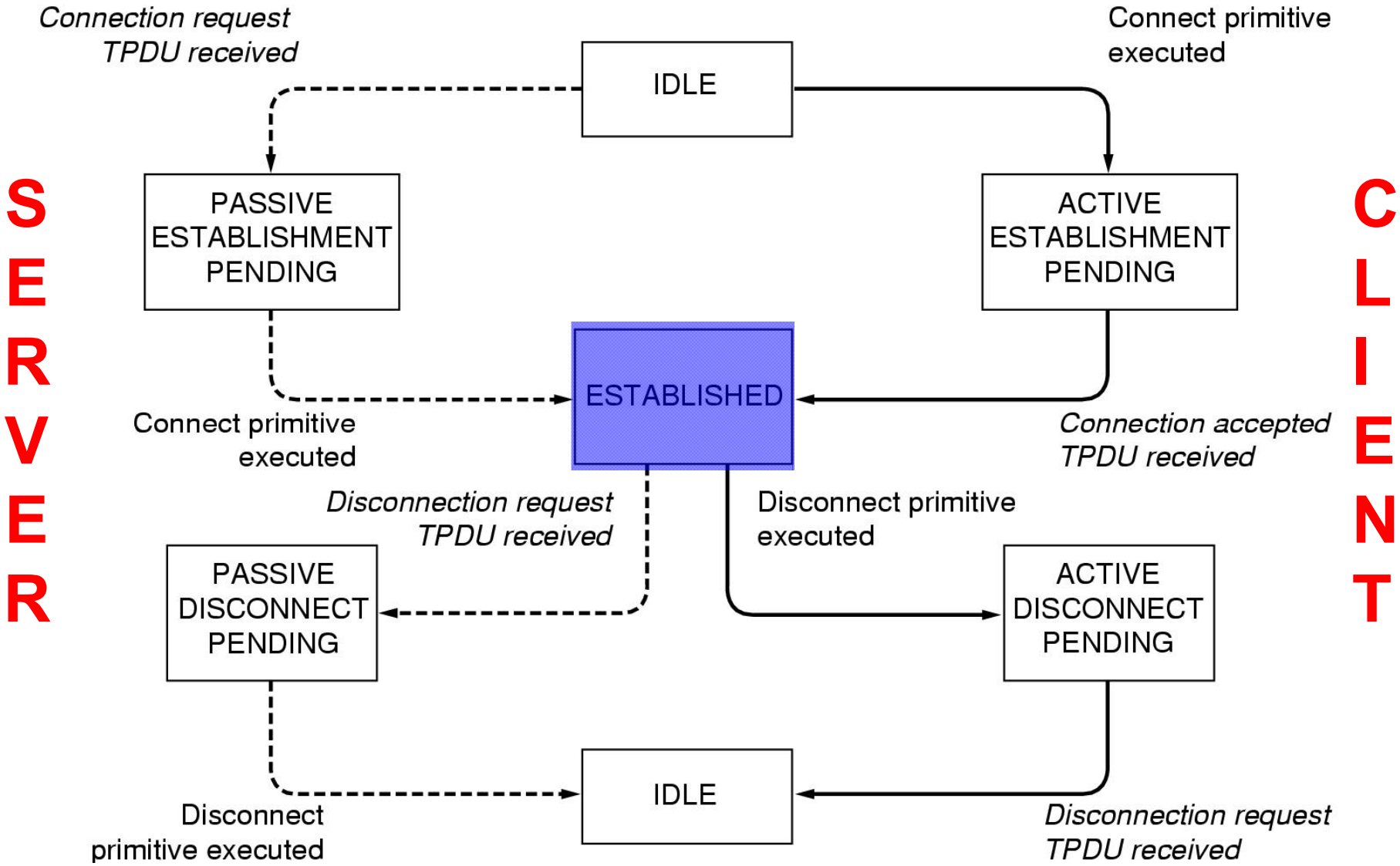
Simple service: state diagram



Simple service: state diagram



Simple service: state diagram



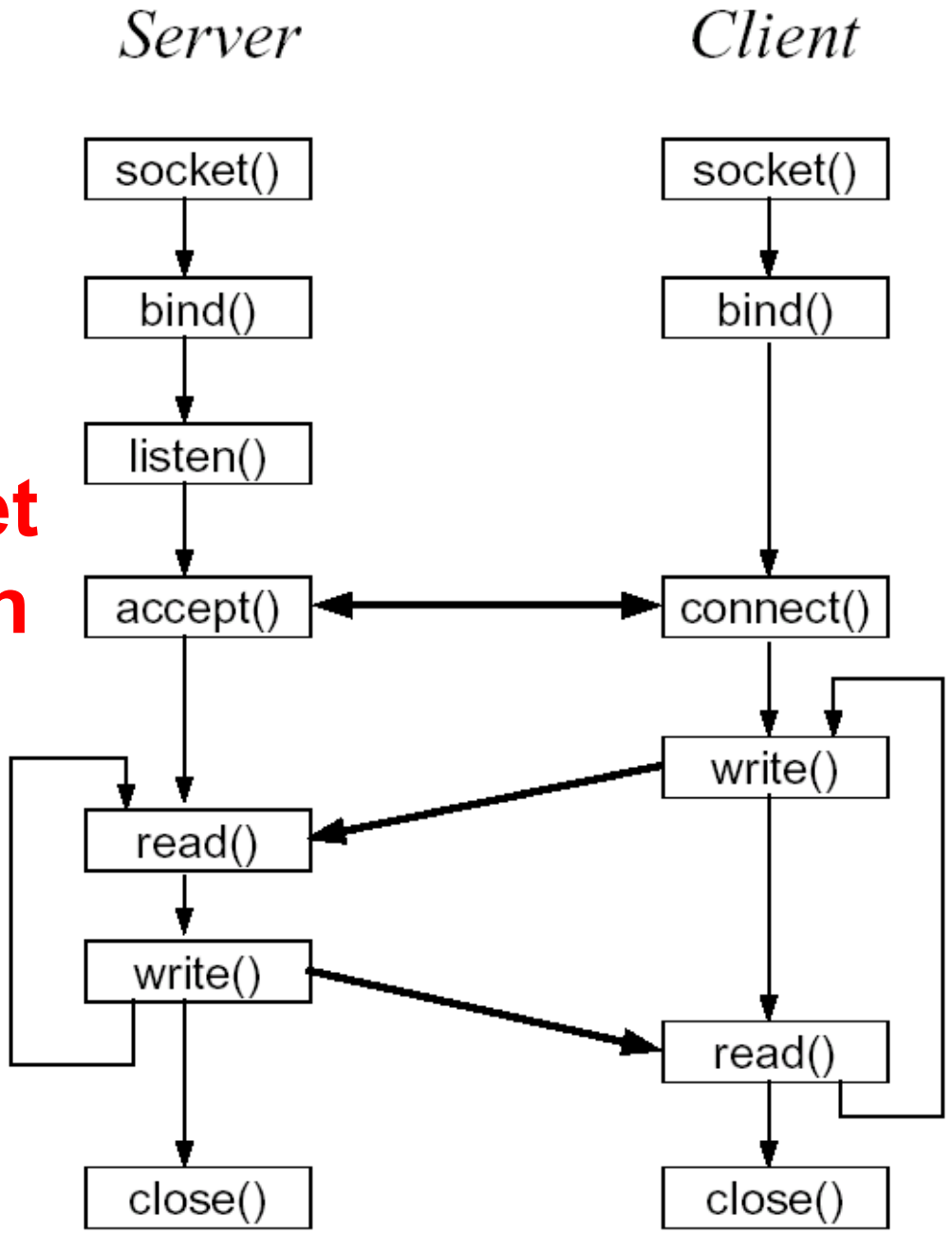
Berkeley service primitives

- Used in Berkeley UNIX for TCP
- Addressing primitives:
  - socket
  - bind
- Server primitives:
  - listen
  - accept
  - send + receive
  - close
- Client primitives:
  - connect
  - send + receive
  - close

Berkeley service primitives

socket	**create new communication end point**
bind	**attach a local address to a socket**
listen	**announce willingness to accept connections; give queue size**
accept	**block caller until a connection request arrives**
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**

Connection-Oriented Socket Communication



Sockets – Server Side

```
serverAddress : TransportAddress /* Publicly known address */
...
PROCESS Server IS
  clientSocket : Socket; /* Private socket */
  ...
  BEGIN
    serverSocket := NEW Socket;
    serverSocket.bind(serverAddress);
    serverSocket.listen(maxConnections);
    LOOP
        serverSocket.accept(clientSocket);
        clientSocket.recv(request);
        clientSocket.send(answer);
        clientSocket.close();
    END LOOP;
END Server;
```

Sockets – Client Side

```
serverAddress : TransportAddress /* Publicly known address */
...
PROCESS Client IS
    clientAddress : TransportAddress; /* Private address */
    clientSocket : Socket;      /* Private socket  */
    ...
    BEGIN
    clientAddress := NEW TransportAddress;
    clientSocket := NEW Socket;

    clientSocket.bind(clientAddress);
    LOOP
        IF clientSocket.connect(serverAddress)
            THEN EXIT;
            ELSE sleep(1);
        END IF;
    END LOOP;

    clientSocket.recv(request);
    clientSocket.send(answer);
    clientSocket.close();
    END Client;
```

Chapter 6: Roadmap

- 6.1 Transport Services
- 6.2 Elements of transport protocol
- 6.3 Simple transport protocol
- 6.4 The Internet Transport Layer
  - UDP
  - TCP

Elements of Transport Protocols

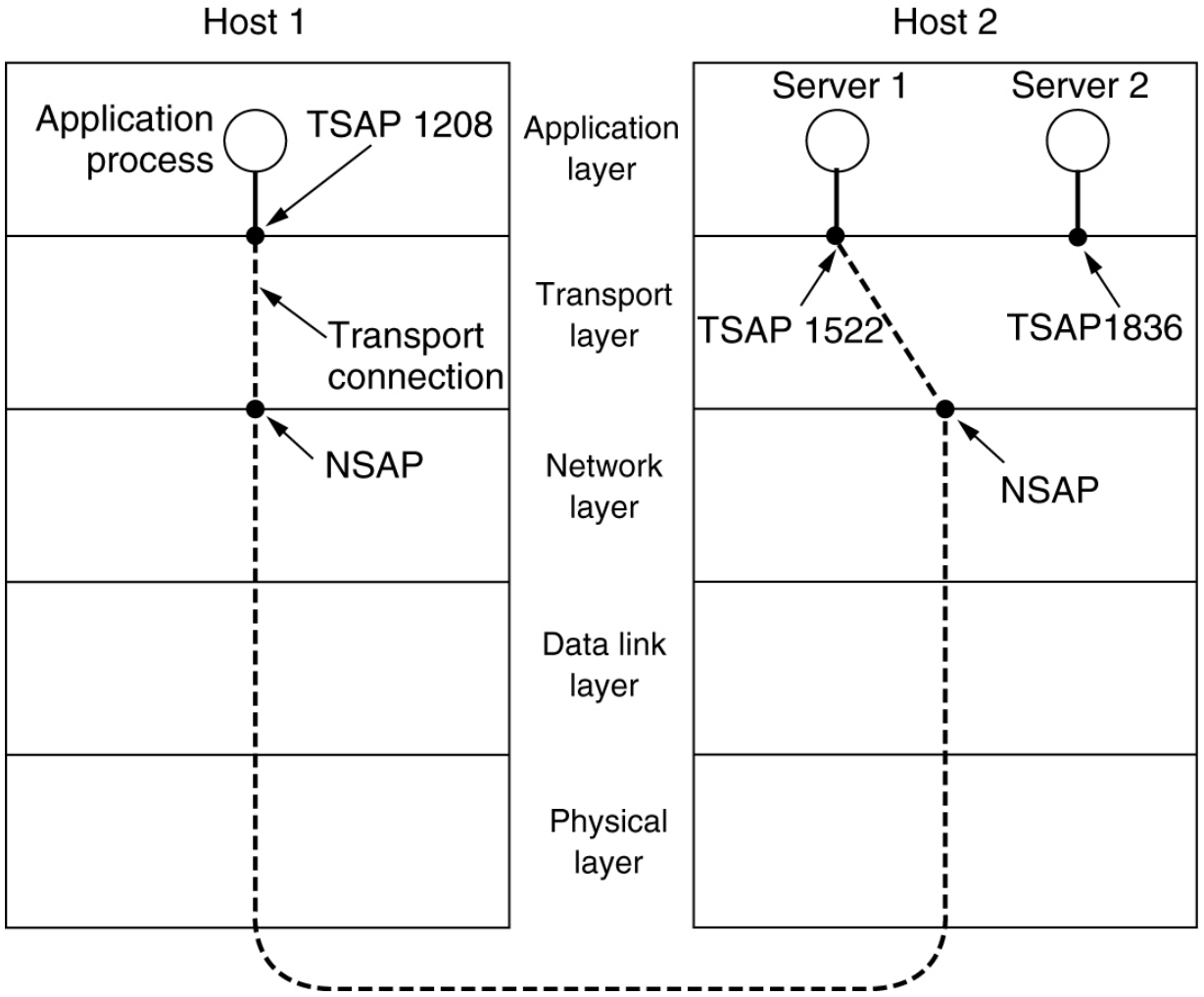
- Addressing
- Connection Establishment
- Connection Release
- Flow Control and Buffering
- Multiplexing
- Crash Recovery

Addressing

- **TSAP** = transport service access point
- **Problems:**
  - Getting TSAP addresses?
  - From TSAP address to NSAP address?

Addressing

- **Connection scenario**



Addressing

- Connection scenario
  - Host 2 (server)
    - Time-of-day server attaches itself to TSAP 1522
  - Host 1 (client)
    - Connect from TSAP 1208 to TSAP 1522
    - Setup network connection to host 2
    - Send transport connection request
  - Host 2
    - Accept connection request

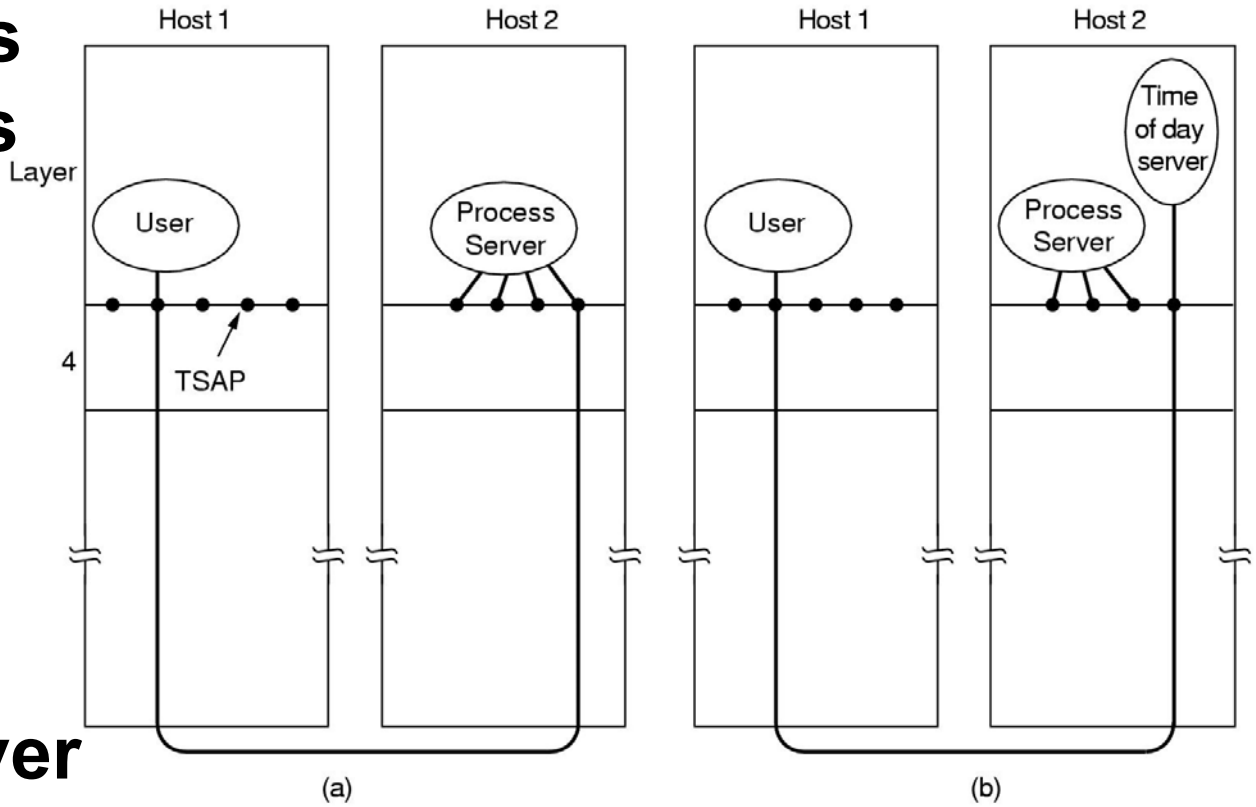
Addressing

- Getting TSAP addresses?
  - Stable TSAP addresses
    - For key services
    - Not for user processes
      - active for a short time
      - number of addresses limited
  - Name servers
    - to find existing servers
    - map service name into TSAP address
  - Initial connection protocol

Addressing

Initial connection protocol

- to avoid many waiting servers ➔ one process server (with a well-known address)
- waits on many TSAPs
- creates requested server



Addressing

- From TSAP address to NSAP address?
  - hierarchical addresses
    - address = <country> <network> <host> <port>
      - Examples: IP address + port
        Telephone numbers (<> number portability?)
    - Disadvantages:
      - TSAP bound to host!
  - flat address space
    - Advantages:
      - Independent of underlying network addresses
      - TSAP address not bound to host
    - Mapping to network addresses:
      - Name server
      - broadcast

Connection Establishment

- **Problem:** delayed duplicates!
- same packets are received in same order a second time!

Recognized the duplicated?

Main cause: The network has storage capabilities, and unpredictable delays.

Connection Establishment

- **Unsatisfactory solutions:**
  - **throwaway TSAP addresses**
    - **need unlimited number of addresses?**
    - **process server solution impossible**
  - **connection identifier**
    - **Never reused!**
      - **Maintain state in hosts**

Connection Establishment

- **Satisfactory solutions**
  - **Ensure limited packet lifetime (incl. Acks)**
  - **Mechanisms**
    - **prevent packets from looping + bound congestion delay**
    - **Hop-counter in each packet**
    - **timestamp in each packet**
  - **Basic assumption**

Maximum packet lifetime T

If we wait a time T after sending a packet, all traces of it (including Acks) are gone

Connection Establishment

- **Tomlinson's method**
  - **Basic idea:**

    2 identically numbered TPDUs are never outstanding at the same time!

  - **Requires: clock in each host**
    - **Clock keeps running, even when a hosts crashes**

Connection Establishment

- **Tomlinson's method**

  Never reuse a sequence number x within the lifetime T for the packet with x

  - **Problems to solve**
    - **Selection of the initial sequence number for a new connection**
    - **Wrap around of sequence numbers for an active connection**
    - **Handle host crashes**

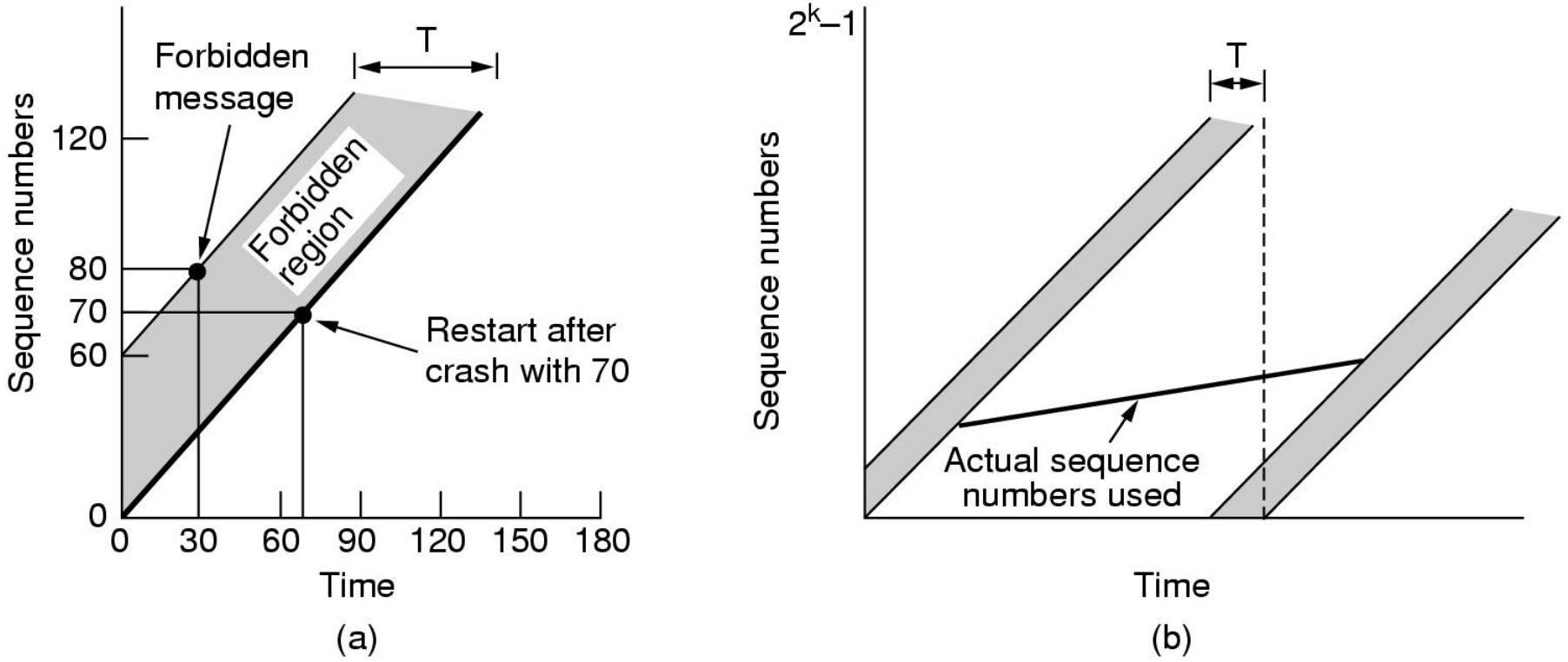
  → Forbidden region

Connection Establishment

- **Tomlinson's method**
  - Initial sequence number
    = lower order bits of clock
  - Ensure initial sequence numbers are always OK
    → forbidden region
  - Wrap around
    - Idle
    - Resynchronize sequence numbers

Connection Establishment

- **Tomlinson - forbidden region**



Linear relation between time and initial sequence numbers
(a) TPDUs may not enter forbidden reg. (b) resynchronization problem

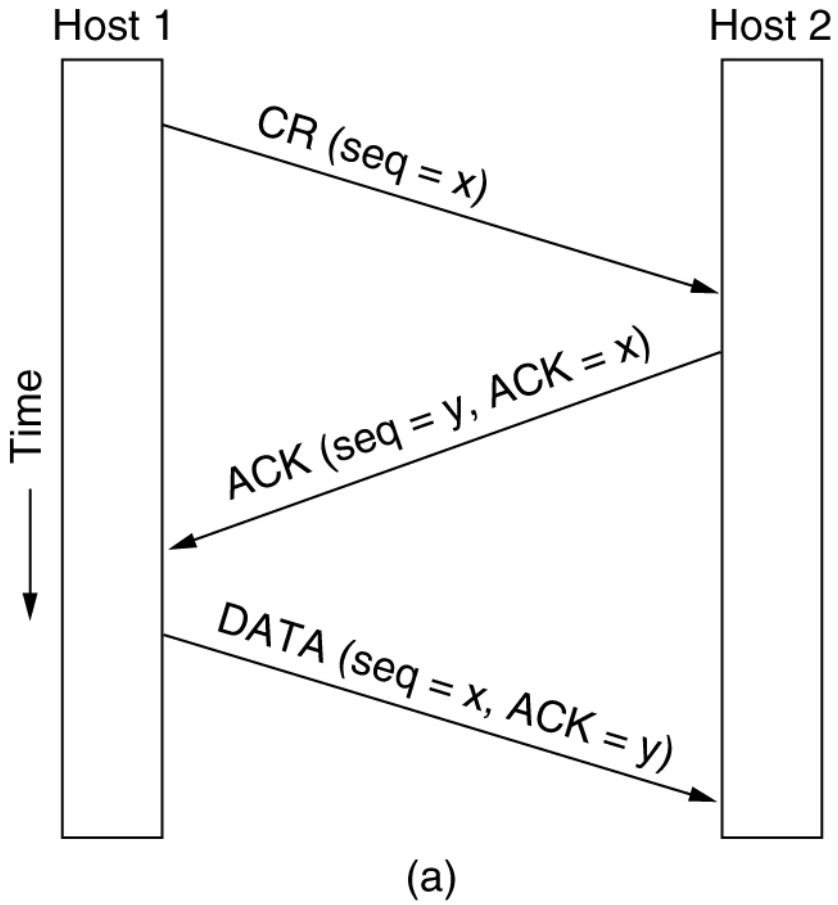
Error-Free Connection Establishment

- **Problem:** we have a way of avoiding duplicates, but how do we get a connection in the first place?
- **Note:** One way or the other we have to get the sender and receiver **to agree on initial sequence numbers.** We need to avoid that an old (unnumbered) connection request pops up.
- **Solution: Three-way handshake.**

Error-Free Connection Establishment

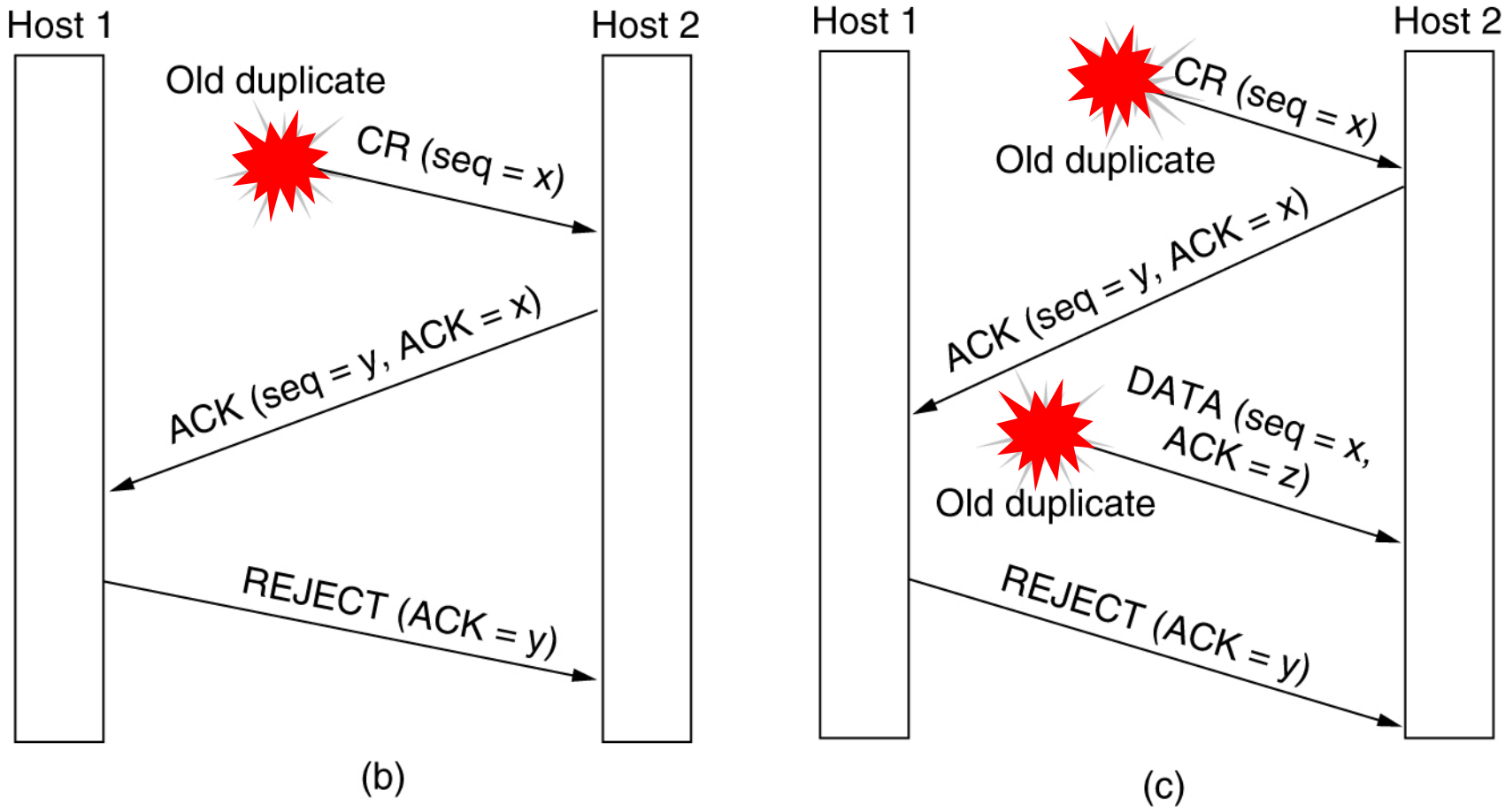
- three-way-handshake

No combination of delayed packets can cause the protocol to fail



Error-Free Connection Establishment

- three-way-handshake



Connection Release

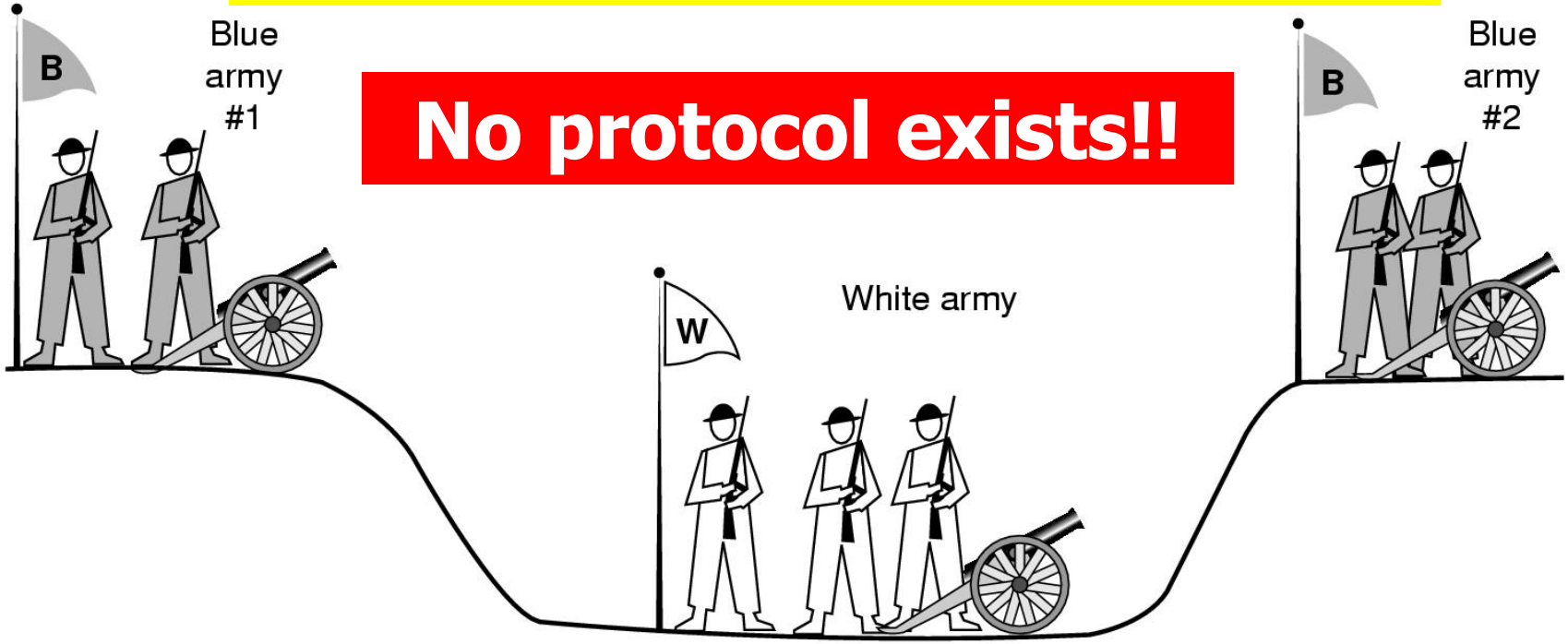
- 2 styles:
  - Symmetric
    - Both parties should agree to release connection
    - How to reach agreement? Two-army problem
    - Solution: three-way-handshake. OK?
  - Asymmetric
    - Connection broken when one party hangs up
    - Abrupt! ➔ may result in data loss
- Pragmatic approach
  - Connection = 2 unidirectional connections
  - Sender can close unidirectional connection

Connection Release

- **Symmetric: two-army-problem**

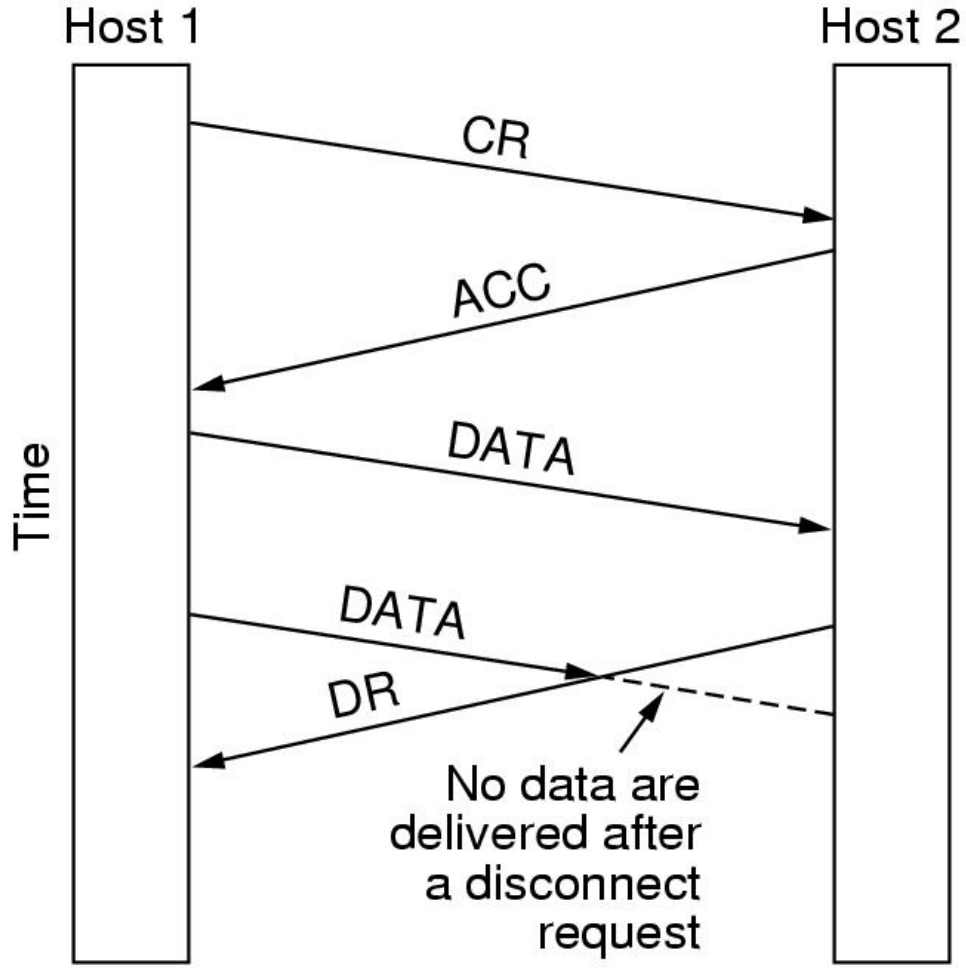
Simultaneous attack by blue army

Communication is unreliable



Connection Release

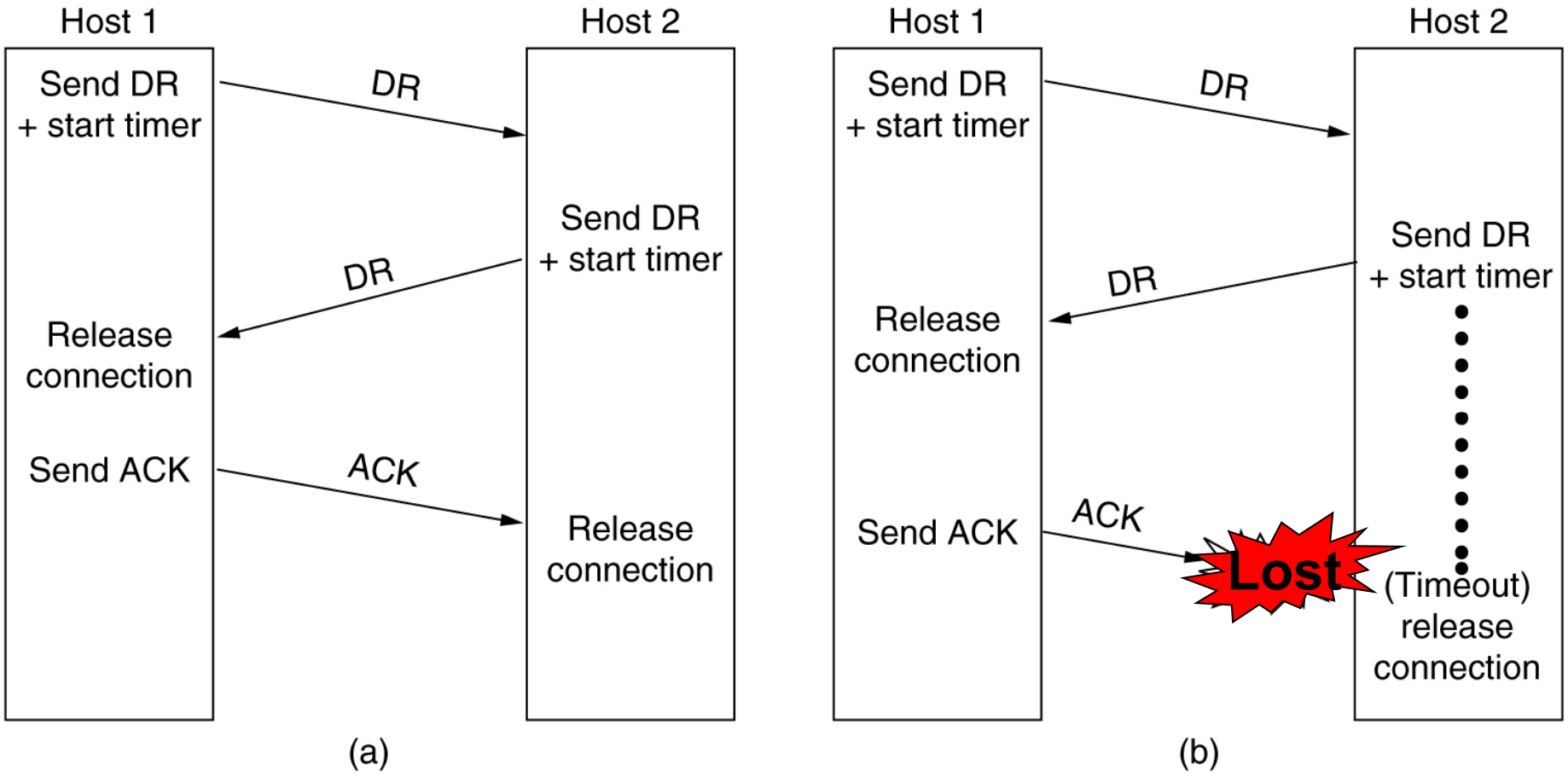
- **Symmetric: data loss**



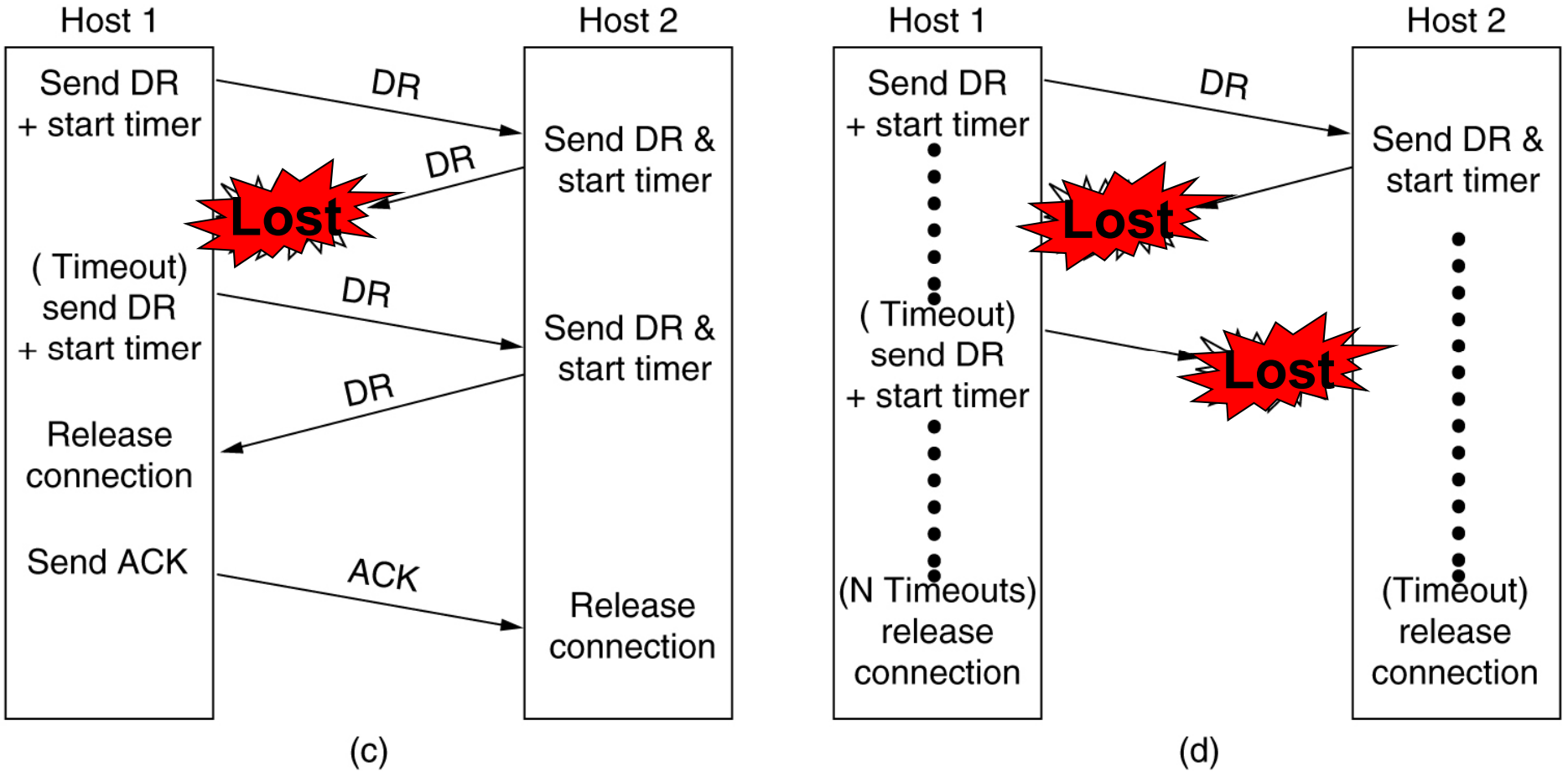
Connection Release

- **Three-way-handshake + timers**
  - **Send disconnection request**
    + start timer to resend (at most N times) the disconnection request
  - **ACK disconnection request**
    + start timer to release connection

Connection Release



Connection Release



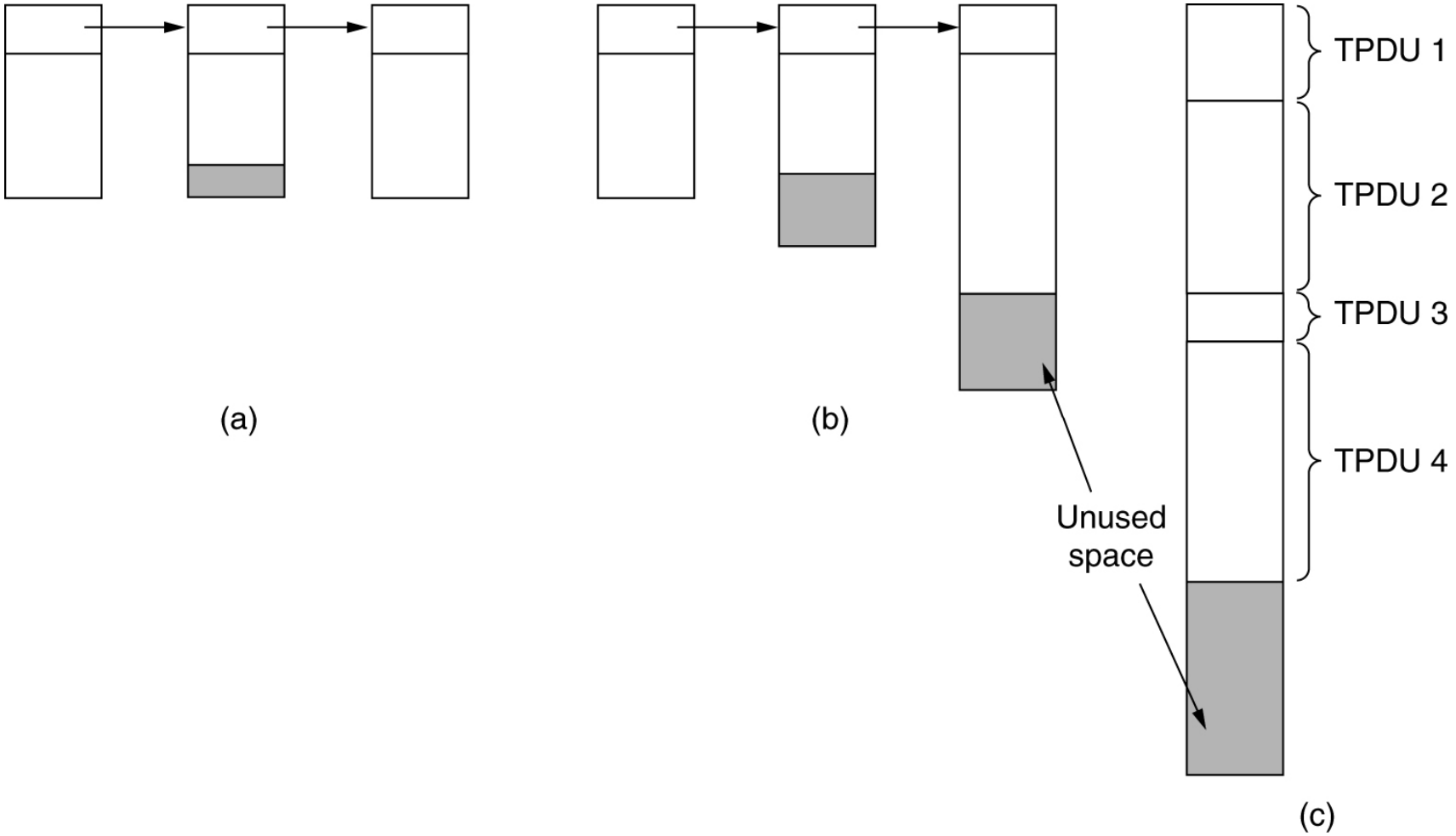
Flow Control and Buffering

- **problem:**
  - Hosts may have so many connections that it becomes infeasible to allocate a fixed number of buffers per connection to implement a proper sliding window protocol
- **Solution:**
  - **dynamic** buffer allocation scheme.
  - **In general:** the sender and receiver need to negotiate the number of TPDUs that can be transmitted in sequence.

Flow Control and Buffering

	Transport	Data link
connections, lines	many varying	few fixed
(sliding) window size	varying	fixed
buffer management	different sizes?	fixed size

Flow Control and Buffering



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

Flow Control and Buffering

Dynamic Buffer Allocation: decouple buffering from ACKs

	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 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

Flow Control and Buffering

Dynamic Buffer Allocation: decouple buffering from ACKs

	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 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
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

Question:

what can we do about the potential deadlock?

Flow Control and Buffering

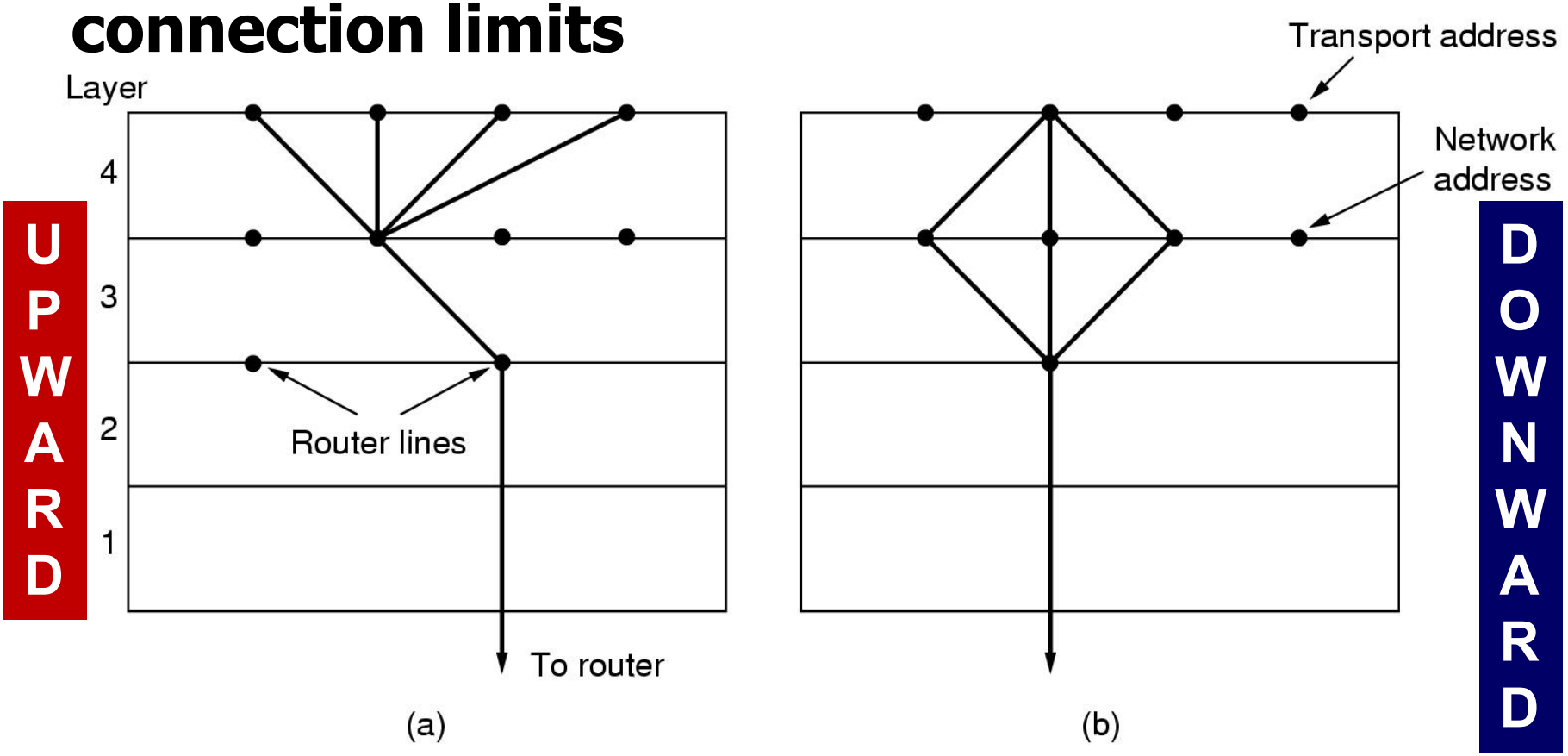
- **Where to buffer?**
  - **datagram network ➔ @ sender**
  - **reliable network**
  - **+ Receiver process guarantees free buffers?**
    - **No: for low-bandwidth bursty traffic ➔ @ sender**
    - **Yes: for high-bandwidth smooth traffic ➔ @ receiver**

Flow Control and Buffering

- **Window size?**
  - **Goal:**
    - Allow sender to continuously send packets
    - Avoid network congestion
  - **Approach:**
    - maximum window size = c * r
      - network can handle c TPDUs/sec
      - r = cycle time of a packet
    - measure c & r and adapt window size

Multiplexing

- **Upward:** reduce number of network connections to reduce cost
- **Downward:** increase bandwidth to avoid per connection limits



Crash Recovery

- **Recovery from network, router crashes?**
  - **No problem**
    - **Datagram network:** loss of packet is always handled
    - **Connection-oriented network:** establish new connection + use state to continue service
- **Recovery from host crash?**
  - **server crashes, restarts:** implications for client?

How should the sending host respond when the receiving host crashes before, during, or after its response?

  - **assumptions:**
    - no state saved at crashed server
    - no simultaneous events
  - **NOT POSSIBLE !!!**

Crash Recovery

- Illustration of problem: File transfer
  - Sender: 1-bit window protocol
    - States S0: had no outstanding (unacknowledged) TPDUs
    - States S1: had one outstanding TPDU
    - packet with seq number 0 transmitted; wait for ack
  - Receiver: actions
    - Ack packet
    - Write data to disk
    - Order ?

Different combinations of client and server strategy

A: ACK
W: Write
C: Crash

Strategy used by sending host	Strategy used by receiving host: First ACK, then write			Strategy used by receiving host: 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

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

Crash Recovery

- **Recovery from network, router crashes?**
  - **No problem**
    - **Datagram network: loss of packet is always handled**
    - **Connection-oriented network: establish new connection + use state to continue service**
- **Recovery from host crash?**

Recovery from a layer N crash can only be done by layer N+1 and only if the higher layer retains enough status information.

Chapter 6: Roadmap

- 6.1 Transport Services
- 6.2 Elements of transport protocol
- 6.3 Simple transport protocol
- 6.4 The Internet Transport Layer
  - UDP
  - TCP

Simple Transport Protocol

- **Service primitives:**
  - connum = **LISTEN** (local)
    - Caller is willing to accept connection
    - Blocked till request received
  - connum = **CONNECT** (local, remote)
    - Tries to establish connection
    - Returns identifier (nonnegative number)
  - status = **SEND** (connum, buffer, bytes)
    - Transmits a buffer
    - Errors returned in status
  - status = **RECEIVE** (connum, buffer, bytes)
    - Indicates caller's desire to get data
  - status = **DISCONNECT** (connum)
    - Terminates connection

Simple Transport Protocol

- Transport entity
  - Uses a connection-oriented reliable network
  - Programmed as a library package
  - Network interface
    - ToNet(…)
    - FromNet(…)
    - Parameters:
      - Connection identifier (connum = VC)
      - Q bit: 1 = control packet
      - M bit: 1 = more data packets to come
      - Packet type
      - Pointer to data
      - Number of bytes of data

Simple Transport Protocol

- **Transport entity: packet types**

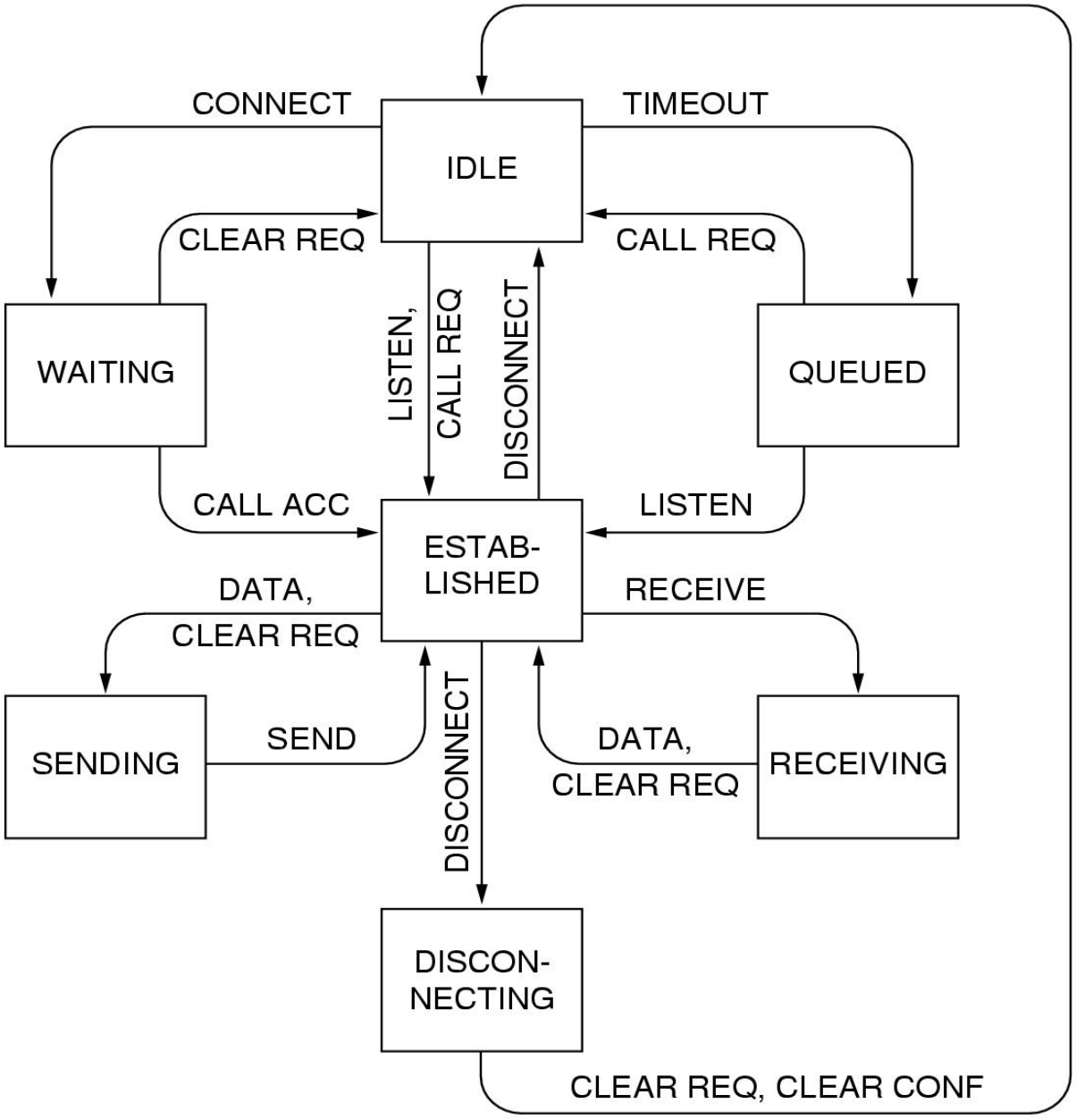
Network packet	Meaning
Call request	Sent to establish a connection
Call accepted	Response to Call Request
Clear Request	Sent to release connection
Clear confirmation	Response to Clear request
Data	Used to transport data
Credit	Control packet to manage window

Simple transport protocol

- Transport entity: state of a connection

State	Meaning
Idle	Connection not established
Waiting	CONNECT done; Call Request sent
Queued	Call Request arrived; no LISTEN yet
Established	
Sending	Waiting for permission to send a packet
Receiving	RECEIVE has been done
Disconnecting	DISCONNECT done locally

The example protocol in graphical form



Finite State Machine

		State						
		Idle	Waiting	Queued	Established	Sending	Receiving	Dis-connecting
Primitives	LISTEN	P1: ~/Idle P2: A1/Estab P2: A2/Idle		~/Estab				
	CONNECT	P1: ~/Idle P1: A3/Wait						
	DISCONNECT				P4: A5/Idle P4: A6/Disc			
	SEND				P5: A7/Estab P5: A8/Send			
	RECEIVE				A9/Receiving			
Incoming packets	Call_req	P3: A1/Estab P3: A4/Queu'd						
	Call_acc		~/Estab					
	Clear_req		~/Idle		A10/Estab	A10/Estab	A10/Estab	~/Idle
	Clear_conf							~/Idle
	DataPkt						A12/Estab	
	Credit				A11/Estab	A7/Estab		
Clock	Timeout			~/Idle				

Predicates	Actions	
P1: Connection table full	A1: Send Call_acc	A7: Send message
P2: Call_req pending	A2: Wait for Call_req	A8: Wait for credit
P3: LISTEN pending	A3: Send Call_req	A9: Send credit
P4: Clear_req pending	A4: Start timer	A10: Set Clr_req_received flag
P5: Credit available	A5: Send Clear_conf	A11: Record credit
	A6: Send Clear_req	A12: Accept message

Simple transport protocol

- Transport entity: code
  - To read and study at home!
  - Questions?
    - Is it acceptable not to use a transport header?
    - How easy would it be to use another network protocol?

Example Transport Entity (1)

```
#define MAX_CONN 32                     /* max number of simultaneous connections */
#define MAX_MSG_SIZE 8192               /* largest message in bytes */
#define MAX_PKT_SIZE 512                /* largest packet in bytes */
#define TIMEOUT 20
#define CRED 1
#define OK 0

#define ERR_FULL –1
#define ERR_REJECT –2
#define ERR_CLOSED –3
#define LOW_ERR –3

typedef int transport_address;
typedef enum {CALL_REQ,CALL_ACC,CLEAR_REQ,CLEAR_CONF,DATA_PKT,CREDIT} pkt_type;
typedef enum {IDLE,WAITING,QUEUED,ESTABLISHED,SENDING,RECEIVING,DISCONN}  cstate;

/* Global variables. */
transport_address listen_address;       /* local address being listened to */
int listen_conn;                        /* connection identifier for listen */
unsigned char data[MAX_PKT_SIZE];       /* scratch area for packet data */

struct conn {
  transport_address local_address, remote_address;
  cstate state;                         /* state of this connection */
  unsigned char *user_buf_addr;         /* pointer to receive buffer */
  int byte_count;                       /* send/receive count */
  int clr_req_received;                 /* set when CLEAR_REQ packet received */
  int timer;                            /* used to time out CALL_REQ packets */
  int credits;                          /* number of messages that may be sent */
} conn[MAX_CONN + 1];                   /* slot 0 is not used */
```

Example Transport Entity (2)

```
void sleep(void);                                   /* prototypes */
void wakeup(void);
void to_net(int cid, int q, int m, pkt_type pt, unsigned char *p, int bytes);
void from_net(int *cid, int *q, int *m, pkt_type *pt, unsigned char *p, int *bytes);

int listen(transport_address t)
{ /* User wants to listen for a connection. See if CALL_REQ has already arrived. */
  int i, found = 0;

  for (i = 1; i <= MAX_CONN; i++)                   /* search the table for CALL_REQ */
       if (conn[i].state == QUEUED && conn[i].local  address == t) {
              found = i;
              break;
       }

  if (found == 0) {
       /* No CALL_REQ is waiting.  Go to sleep until arrival or timeout. */
       listen_address = t;  sleep();  i = listen_conn ;
  }
  conn[i].state = ESTABLISHED;                      /* connection is ESTABLISHED */
  conn[i].timer = 0;                                /* timer is not used */
```

Example Transport Entity (3)

```
  listen_conn = 0;                          /* 0 is assumed to be an invalid address */
  to_net(i, 0, 0, CALL_ACC, data, 0);       /* tell net to accept connection */
  return(i);                                /* return connection identifier */
}

int connect(transport_address l, transport_address r)
{ /* User wants to connect to a remote process;  send CALL_REQ packet. */
  int i;
  struct conn *cptr;

  data[0] = r;   data[1] = l;               /* CALL_REQ packet needs these */
  i = MAX_CONN;                             /* search table backward */
  while (conn[i].state != IDLE && i > 1) i = i  –1;
  if (conn[i].state == IDLE) {
       /* Make a table entry that CALL_REQ has been sent. */
       cptr = &conn[i];
       cptr->local  address = l; cptr->remote_address = r;
       cptr->state = WAITING; cptr->clr_req_received = 0;
       cptr->credits = 0; cptr->timer = 0;
       to_net(i, 0, 0, CALL_REQ, data, 2);
       sleep();                             /* wait for CALL_ACC or CLEAR_REQ */
       if (cptr->state == ESTABLISHED) return(i);
       if (cptr->clr_req_received) {
            /* Other side refused call. */
            cptr->state = IDLE;             /* back to IDLE state */
            to  net(i, 0, 0, CLEAR_CONF, data, 0);
            return(ERR_REJECT);
       }
  } else return(ERR_FULL);                  /* reject CONNECT: no table space */
}
```

Example Transport Entity (4)

```
int send(int cid, unsigned char bufptr[], int bytes)
{ /* User wants to send a message. */
  int i, count, m;
  struct conn *cptr = &conn[cid];

  /* Enter SENDING state. */
  cptr->state = SENDING;
  cptr->byte_count = 0;                          /* # bytes sent so far this message */
  if (cptr->clr_req_received == 0 && cptr->credits == 0) sleep();
  if (cptr->clr_req_received == 0) {
       /* Credit available; split message into packets if need be. */
       do {
            if (bytes – cptr->byte_count > MAX_PKT_SIZE) {/* multipacket message */
                 count = MAX_PKT_SIZE;  m = 1;   /* more packets later */
            } else {                               /* single packet message */
                 count = bytes – cptr->byte_count;  m = 0;    /* last pkt of this message */
            }
            for (i = 0; i < count; i++) data[i] = bufptr[cptr->byte_count + i];
            to_net(cid, 0, m, DATA_PKT, data, count);   /* send 1 packet */
            cptr->byte_count = cptr->byte_count + count;      /* increment bytes sent so far */
       } while (cptr->byte_count < bytes);         /* loop until whole message sent */
```

Example Transport Entity (5)

```
        cptr->credits--;                    / * each message uses up one credit */
        cptr->state = ESTABLISHED;
        return(OK);
    } else {
        cptr->state = ESTABLISHED;
        return(ERR_CLOSED);                 /* send failed: peer wants to disconnect */
    }
}

int receive(int cid, unsigned char bufptr[], int *bytes)
{ /* User is prepared to receive a message. */
    struct conn *cptr = &conn[cid];

    if (cptr->clr_req_received == 0) {
        /* Connection still established; try to receive. */
        cptr->state = RECEIVING;
        cptr->user_buf_addr = bufptr;
        cptr->byte_count = 0;
        data[0] = CRED;
        data[1] = 1;
        to_net(cid, 1, 0, CREDIT, data, 2);     /* send credit */
        sleep();                                /* block awaiting data */
        *bytes = cptr->byte_count;
    }
    cptr->state = ESTABLISHED;
    return(cptr->clr_req_received ? ERR_CLOSED : OK);
}
```

Example Transport Entity (6)

```
int disconnect(int cid)
{ /* User wants to release a connection. */
  struct conn *cptr = &conn[cid];

  if (cptr->clr_req_received) {                  /* other side initiated termination */
       cptr->state = IDLE;                       /* connection is now released */
       to_net(cid, 0, 0, CLEAR_CONF, data, 0);
  } else {                                       /* we initiated termination */
       cptr->state = DISCONN;                    /* not released until other side agrees */
       to_net(cid, 0, 0, CLEAR_REQ, data, 0);
  }
  return(OK);
}

void packet_arrival(void)
{ /* A packet has arrived, get and process it. */
  int cid;                                       /* connection on which packet arrived */
  int count, i, q, m;
  pkt_type ptype;     /* CALL_REQ, CALL_ACC, CLEAR_REQ, CLEAR_CONF, DATA_PKT, CREDIT */
  unsigned char data[MAX_PKT_SIZE];              /* data portion of the incoming packet */
  struct conn *cptr;

  from_net(&cid, &q, &m, &ptype, data, &count);   /* go get it */
  cptr = &conn[cid];
```

Example Transport Entity (7)

```
switch (ptype) {
  case CALL_REQ:                          /* remote user wants to establish connection */
    cptr->local_address = data[0];  cptr->remote_address = data[1];
    if (cptr->local_address == listen_address) {
          listen_conn = cid;  cptr->state = ESTABLISHED;  wakeup();
    } else {
          cptr->state = QUEUED;  cptr->timer = TIMEOUT;
    }
    cptr->clr_req_received = 0;   cptr->credits = 0;
    break;
  case CALL_ACC:                          /* remote user has accepted our CALL_REQ */
    cptr->state = ESTABLISHED;
    wakeup();
    break;
  case CLEAR_REQ:                         /* remote user wants to disconnect or reject call */
    cptr->clr_req_received = 1;
    if (cptr->state == DISCONN) cptr->state = IDLE;  /* clear collision */
    if (cptr->state ==  WAITING || cptr->state == RECEIVING || cptr->state == SENDING) wakeup();
    break;
  case CLEAR_CONF:                        /* remote user agrees to disconnect */
    cptr->state = IDLE;
    break;
  case CREDIT:                            /* remote user is waiting for data */
    cptr->credits += data[1];
    if (cptr->state == SENDING) wakeup();
    break;
  case DATA_PKT:                          /* remote user has sent data */
    for (i = 0; i < count; i++) cptr->user_buf_addr[cptr->byte_count + i] = data[i];
    cptr->byte_count += count;
    if (m == 0 ) wakeup();
}
Cor}
```

Example Transport Entity (8)

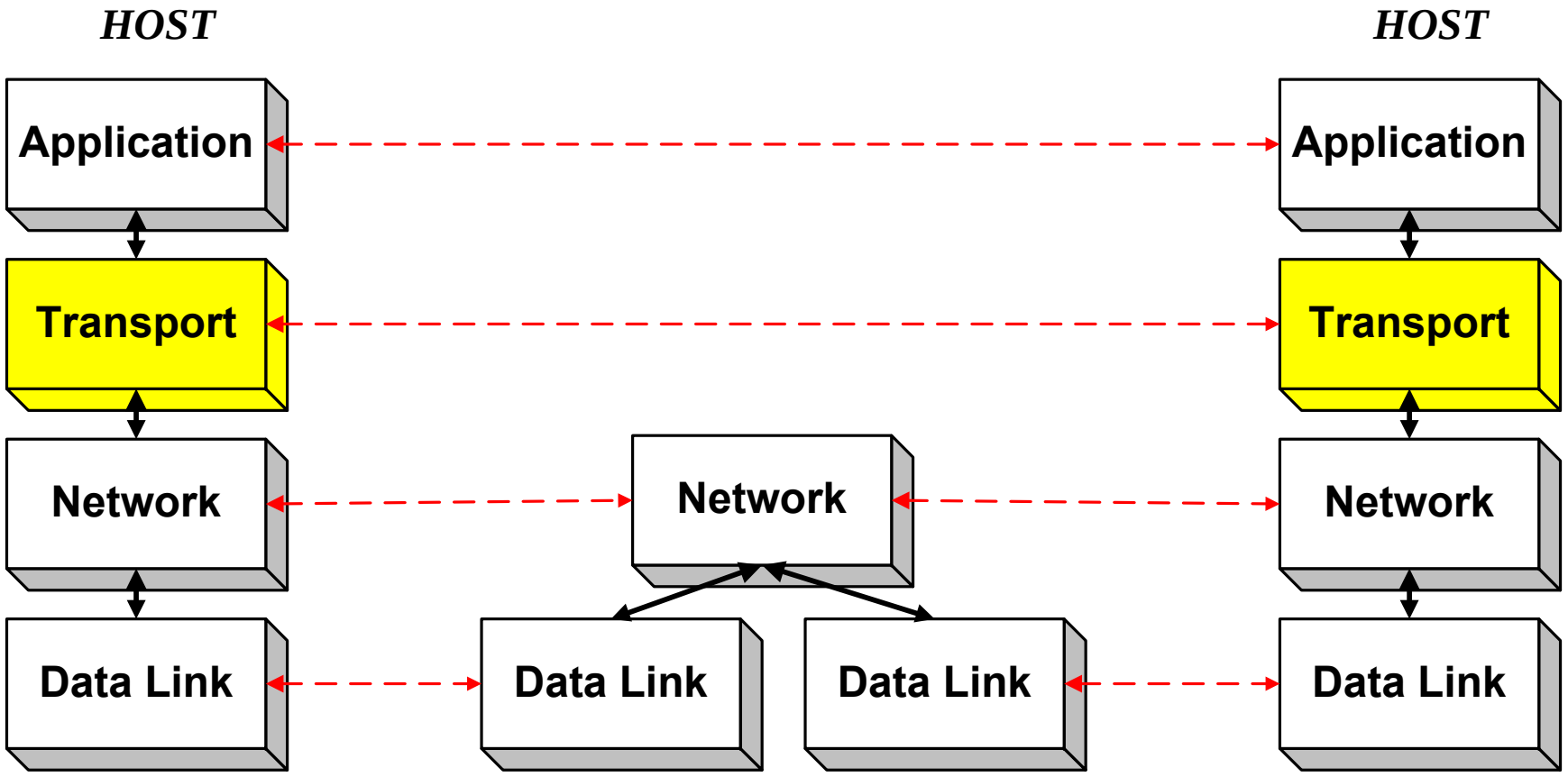
```
}

void clock(void)
{ /* The clock has ticked, check for timeouts of queued connect requests. */
  int i;
  struct conn *cptr;

  for (i = 1; i <= MAX_CONN; i++) {
        cptr = &conn[i];
        if (cptr->timer > 0) {                    /* timer was running */
              cptr->timer--;
              if (cptr->timer == 0) {             /* timer has now expired */
                    cptr->state = IDLE;
                    to_net(i, 0, 0, CLEAR_REQ, data, 0);
              }
        }
  }
}
```

Chapter 6: Roadmap

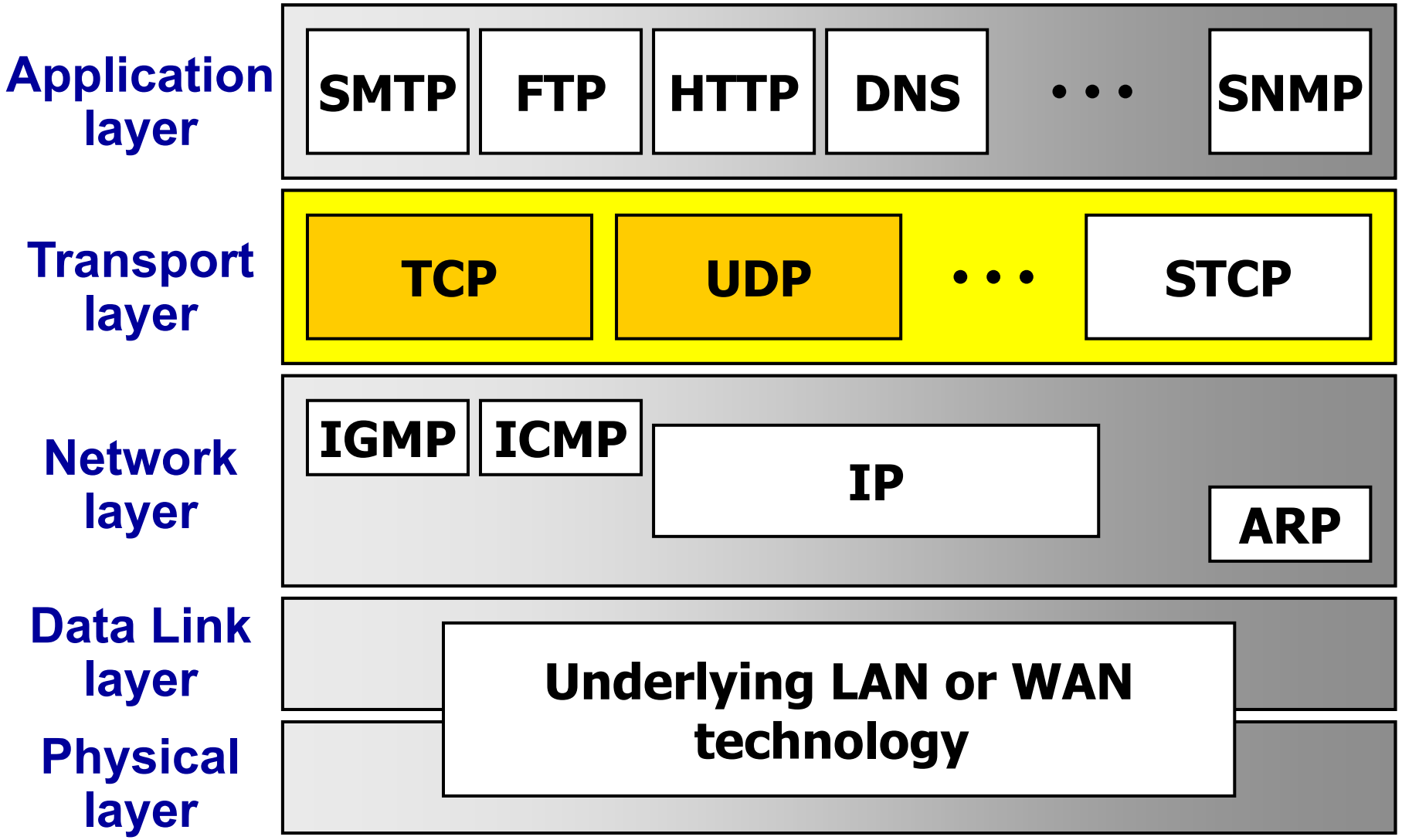
- 6.1 Transport Services
- 6.2 Elements of transport protocol
- 6.3 Simple transport protocol
- 6.4 The Internet Transport Layer
  - UDP
  - TCP

The Internet Transport Layer

- Transport layer protocols are end-to-end protocols
- They are only implemented at the hosts



The Internet Transport Layer



The Internet Transport Layer

The Internet supports 2 transport protocols

UDP - User Datagram Protocol

- datagram oriented
- unreliable, connectionless, simple
- unicast and multicast
- useful only for few applications, e.g., multimedia applications
- used a lot for services
  - network management (SNMP), routing (RIP), naming (DNS), etc.

TCP - Transmission Control Protocol

- Byte stream oriented
- reliable, connection-oriented
- complex
- only unicast
- used for most Internet applications:
  - web (http), email (smtp), file transfer (ftp), terminal (telnet), etc.

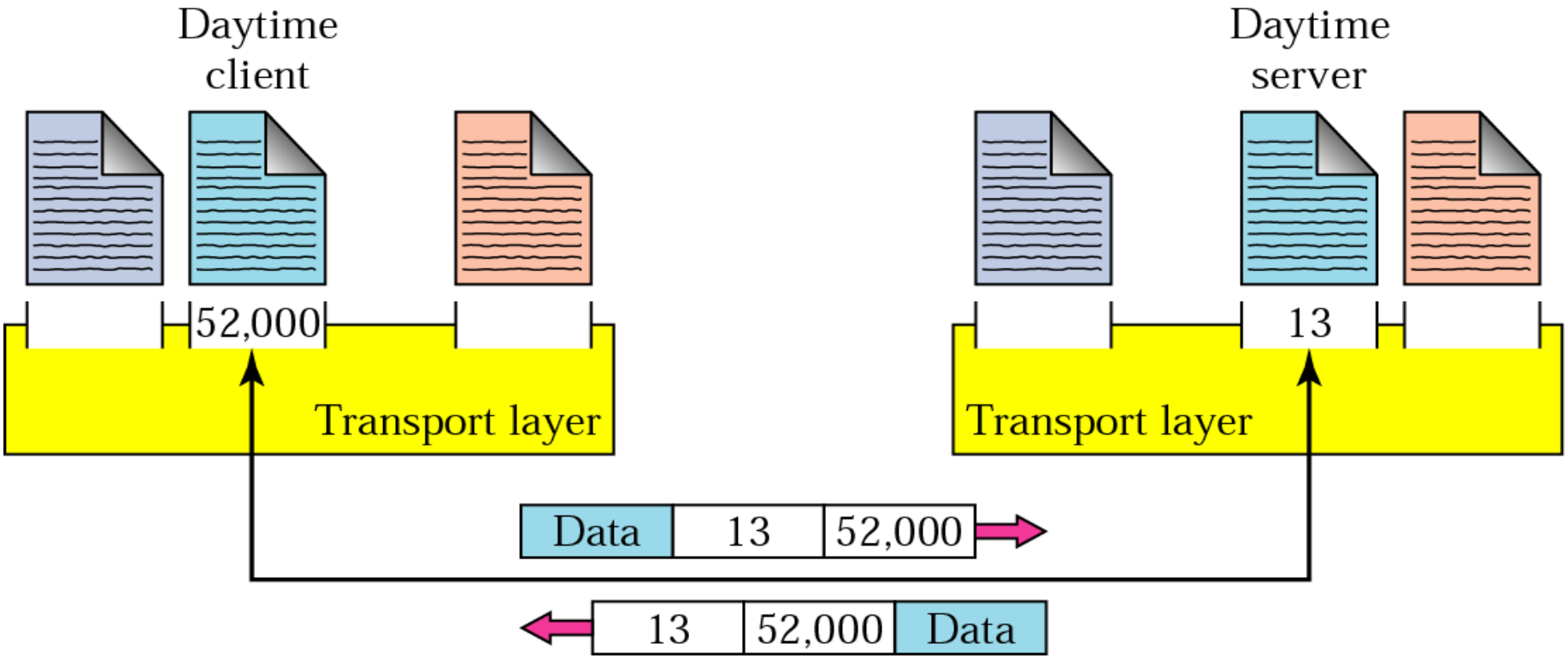
UDP vs TCP

- **UDP**
  - Low-level, connectionless
  - No reliability guarantee
- **TCP**
  - Connection-oriented
  - Reliability guarantee
  - Not as efficient as UDP

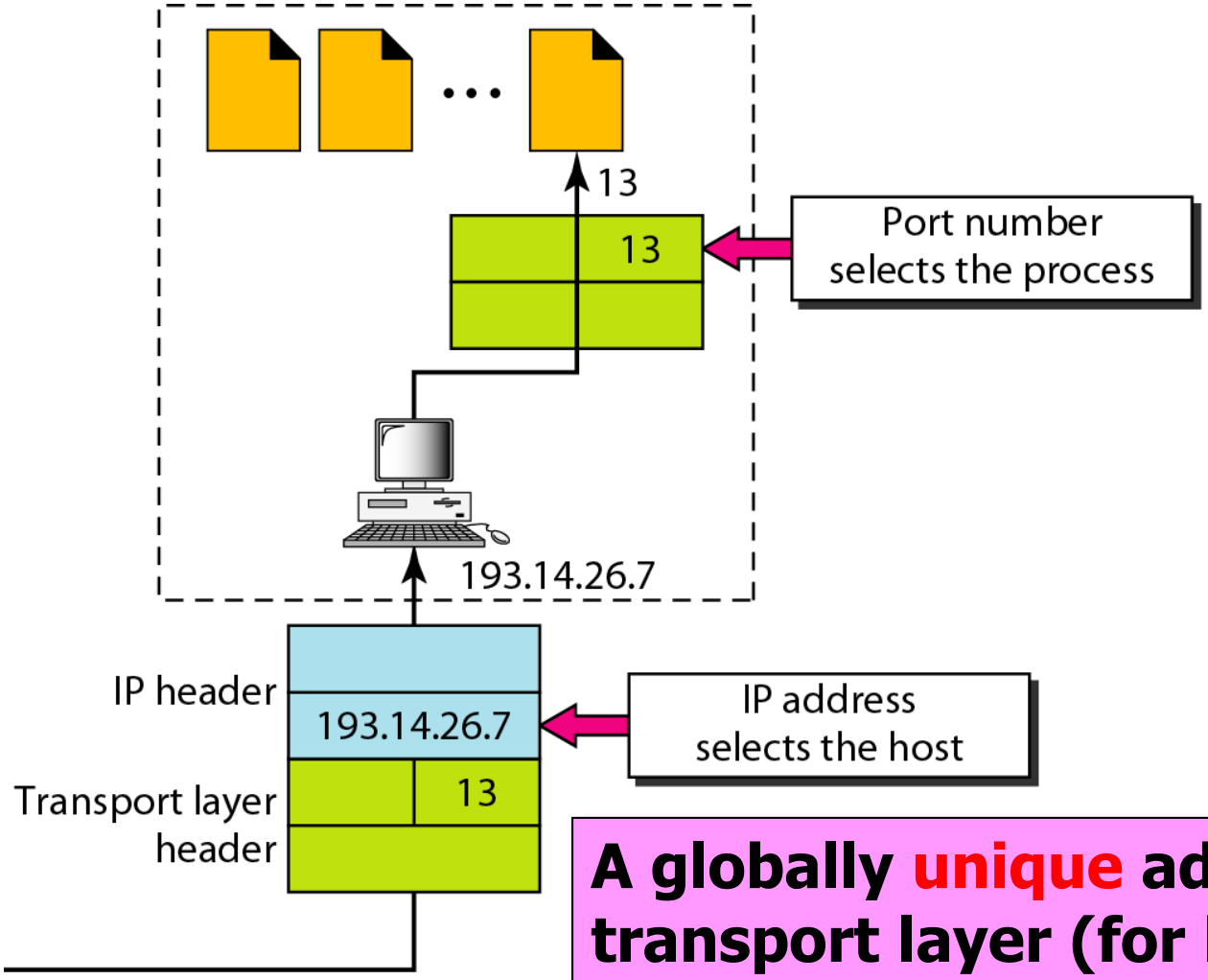
Transport Layer Addressing

Addresses

- **Data link layer → MAC address**
- **Network layer → IP address**
- **Transport layer → *Port number:* identifies the process**



IP addresses *vs.* port numbers

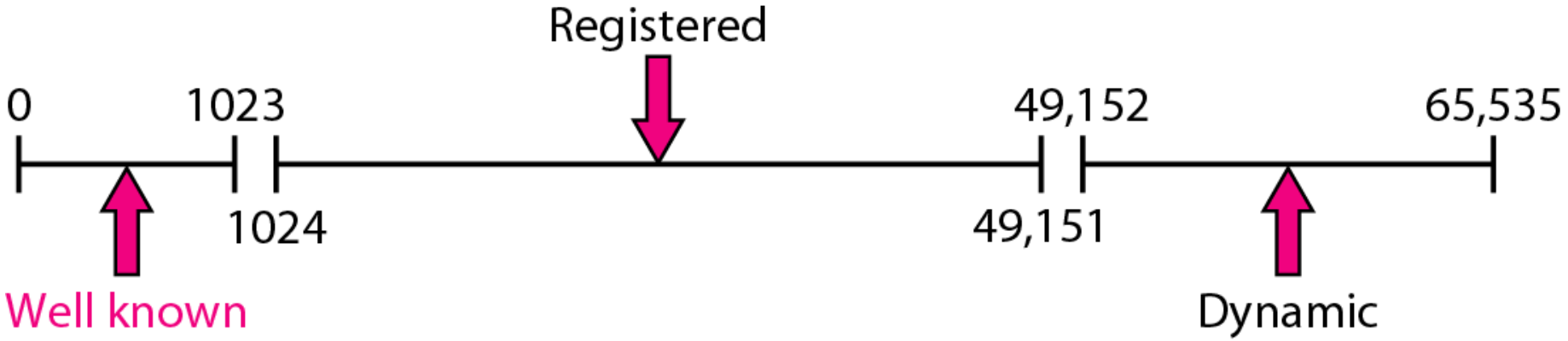


A globally unique address at the transport layer (for both UDP and TCP) is a tuple <IP address, port number>

Port Numbers

- Port numbers are 16-bit integers (0 → 65,535)
  - Servers use well know ports, 0-1023 are privileged
  - Clients use ephemeral (short-lived) ports

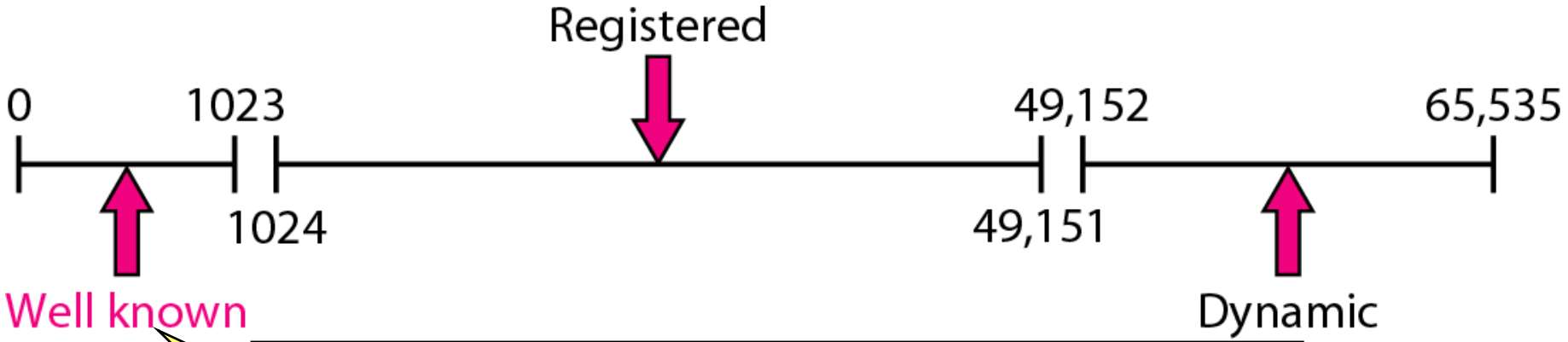
Internet Assigned Numbers Authority (IANA) maintains a list of port number assignment



Port Numbers

- Port numbers are 16-bit integers (0 → 65,535)
  - Servers use well know ports, 0-1023 are privileged
  - Clients use ephemeral (short-lived) ports

Internet Assigned Numbers Authority (IANA) maintains a list of port number assignment

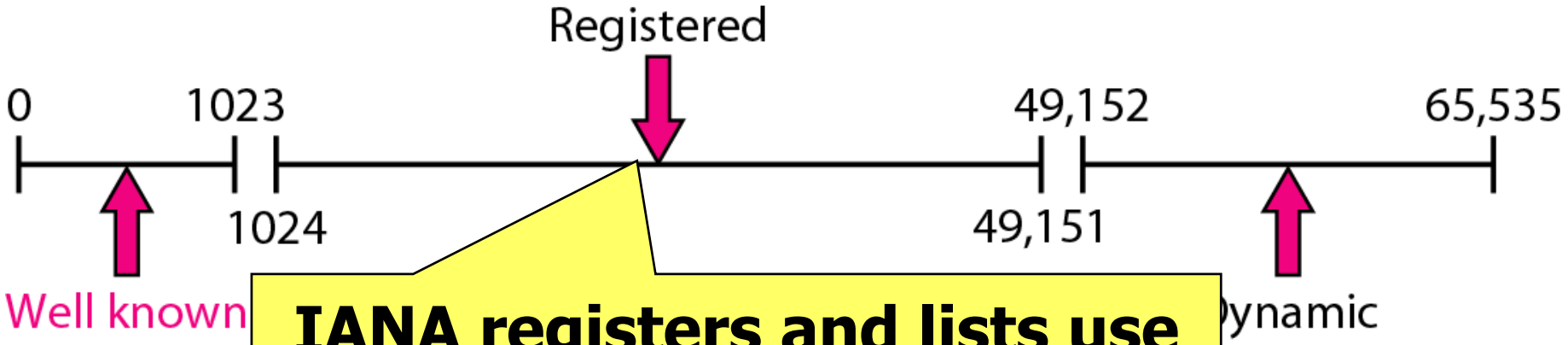


controlled and assigned by IANA

Port Numbers

- Port numbers are 16-bit integers (0 → 65,535)
  - Servers use well know ports, 0-1023 are privileged
  - Clients use ephemeral (short-lived) ports

Internet Assigned Numbers Authority (IANA) maintains a list of port number assignment

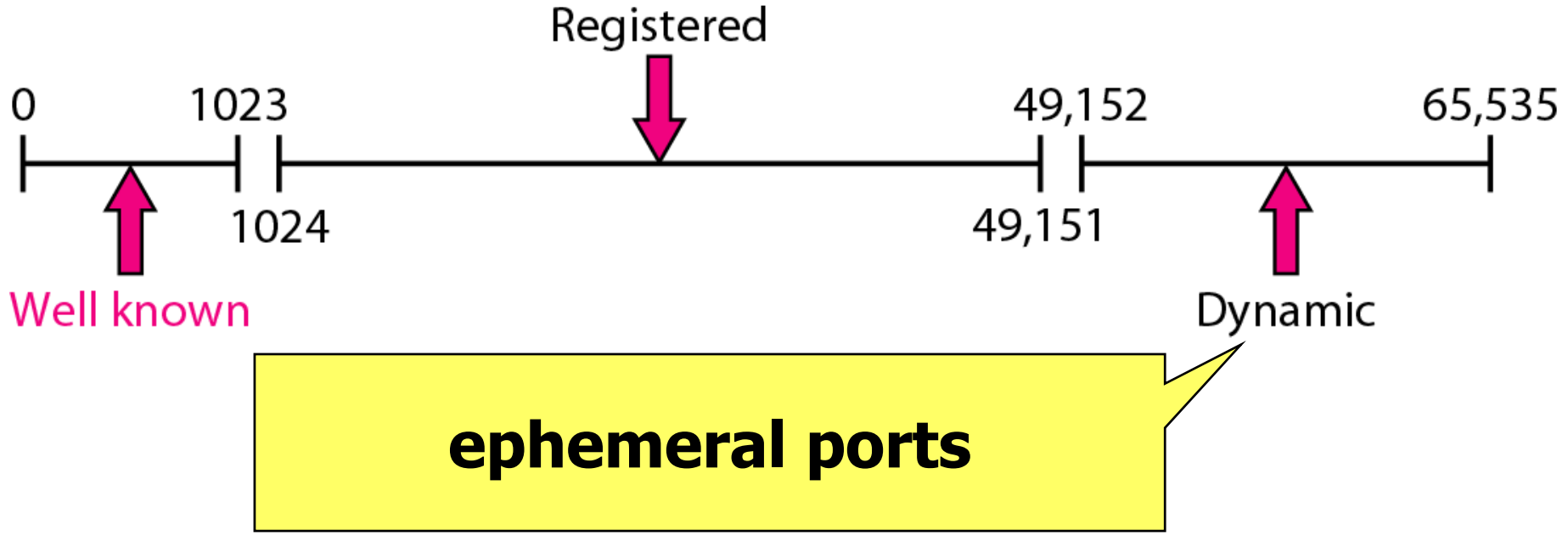


IANA registers and lists use of ports as a convenience (49151 is ¾ of 65536)

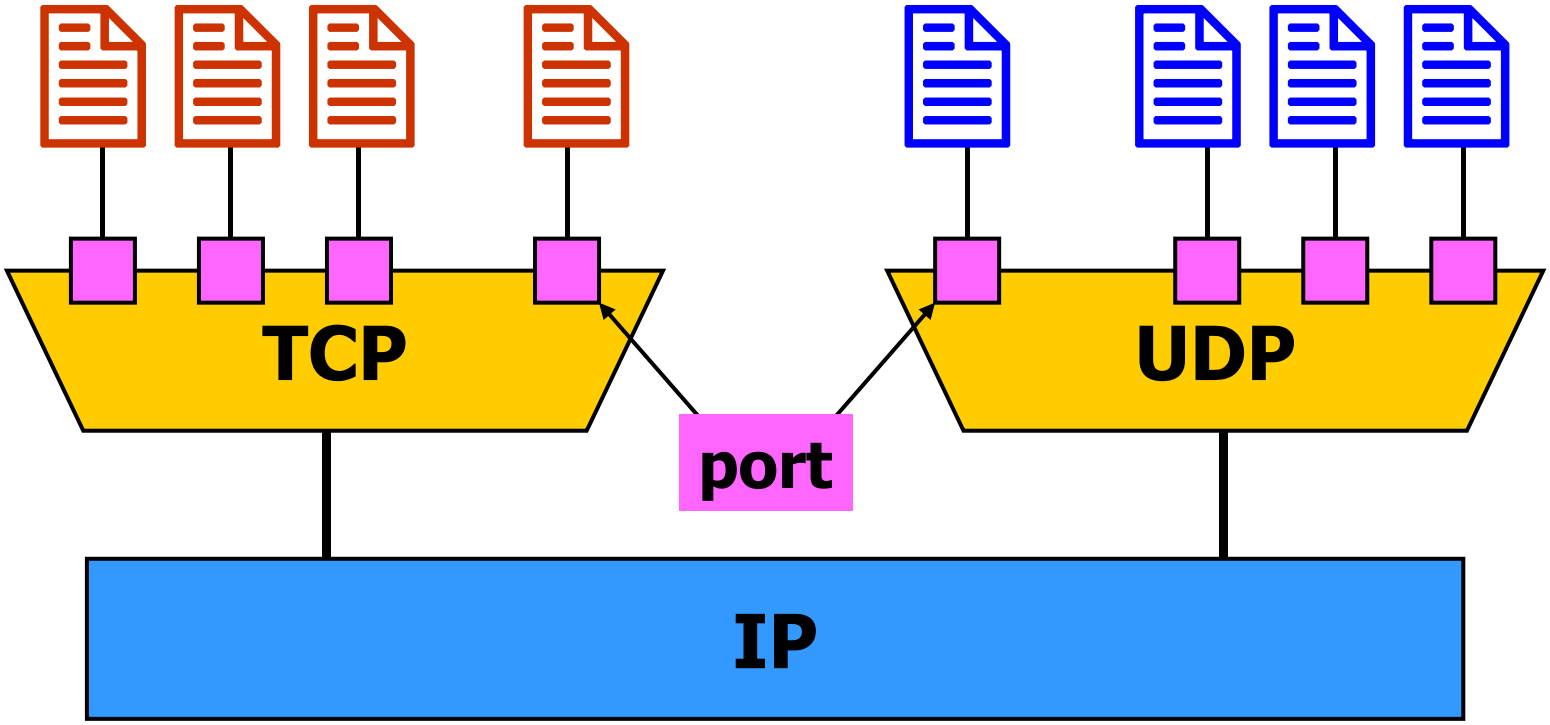
Port Numbers

- Port numbers are 16-bit integers (0 → 65,535)
  - Servers use well know ports, 0-1023 are privileged
  - Clients use ephemeral (short-lived) ports

Internet Assigned Numbers Authority (IANA) maintains a list of port number assignment



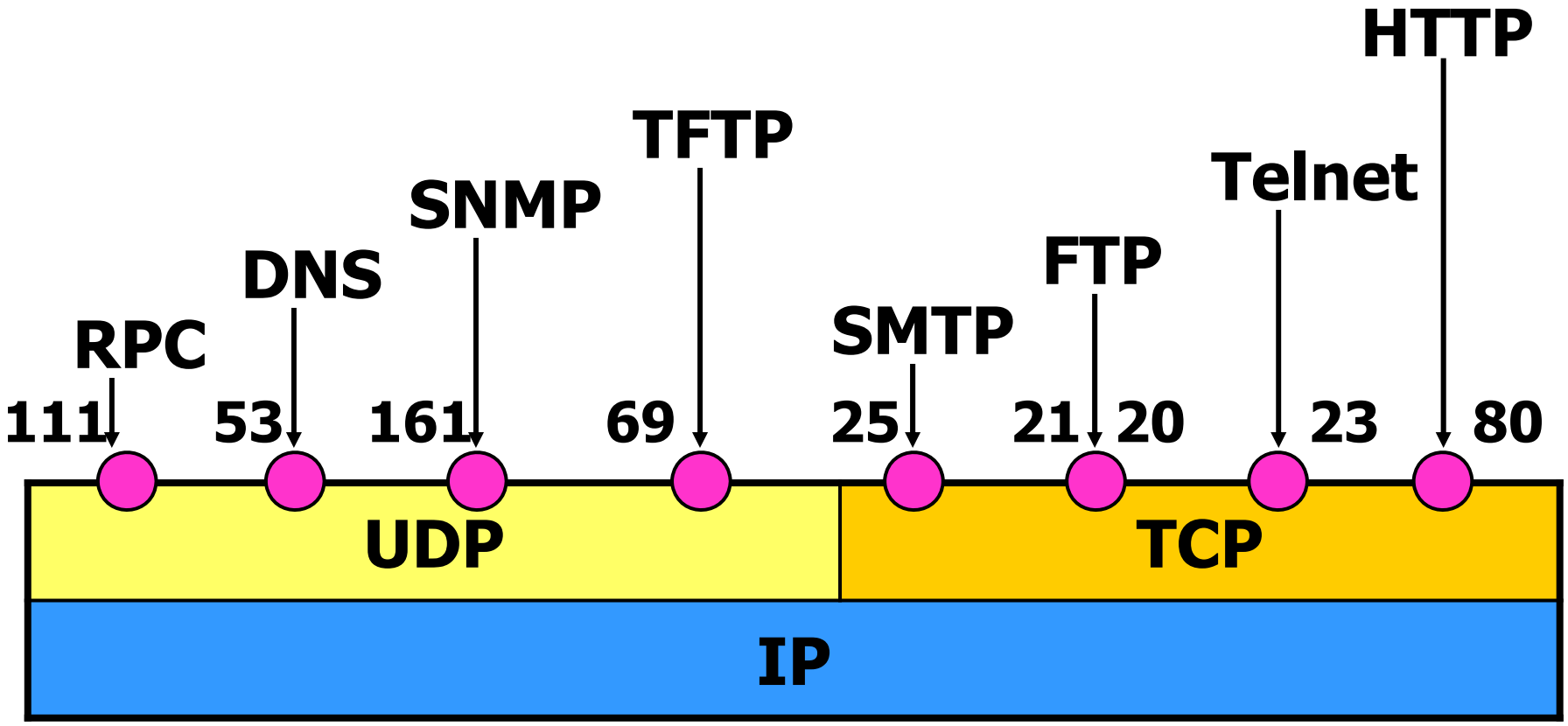
Port Numbers



Port number = TSAP地址

UDP和TCP使用相同的端口规则，但它们是不同的

Port Numbers



UDP and TCP Port

Socket Addressing

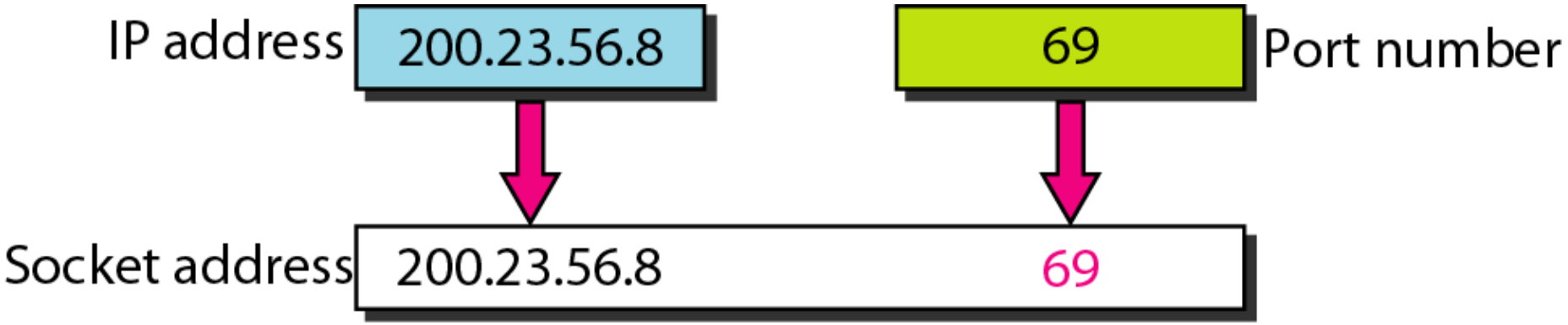
- **Process-to-process delivery needs *two* identifiers: IP address and Port number**
  - **Combination of IP address and port number is called a socket address (a socket is a communication endpoint)**
  - **Client socket address uniquely identifies client process**
  - **Server socket address uniquely identifies server process**

Socket Addressing

- Transport-layer protocol needs a *pair* of socket addresses
  - Client socket address
  - Server socket address
  - For example, socket pair for a TCP connection is a 4-tuple

```
<
    Local IP address,
    Local port,
    Remote IP address,
    Remote port
>
```

Socket Addressing

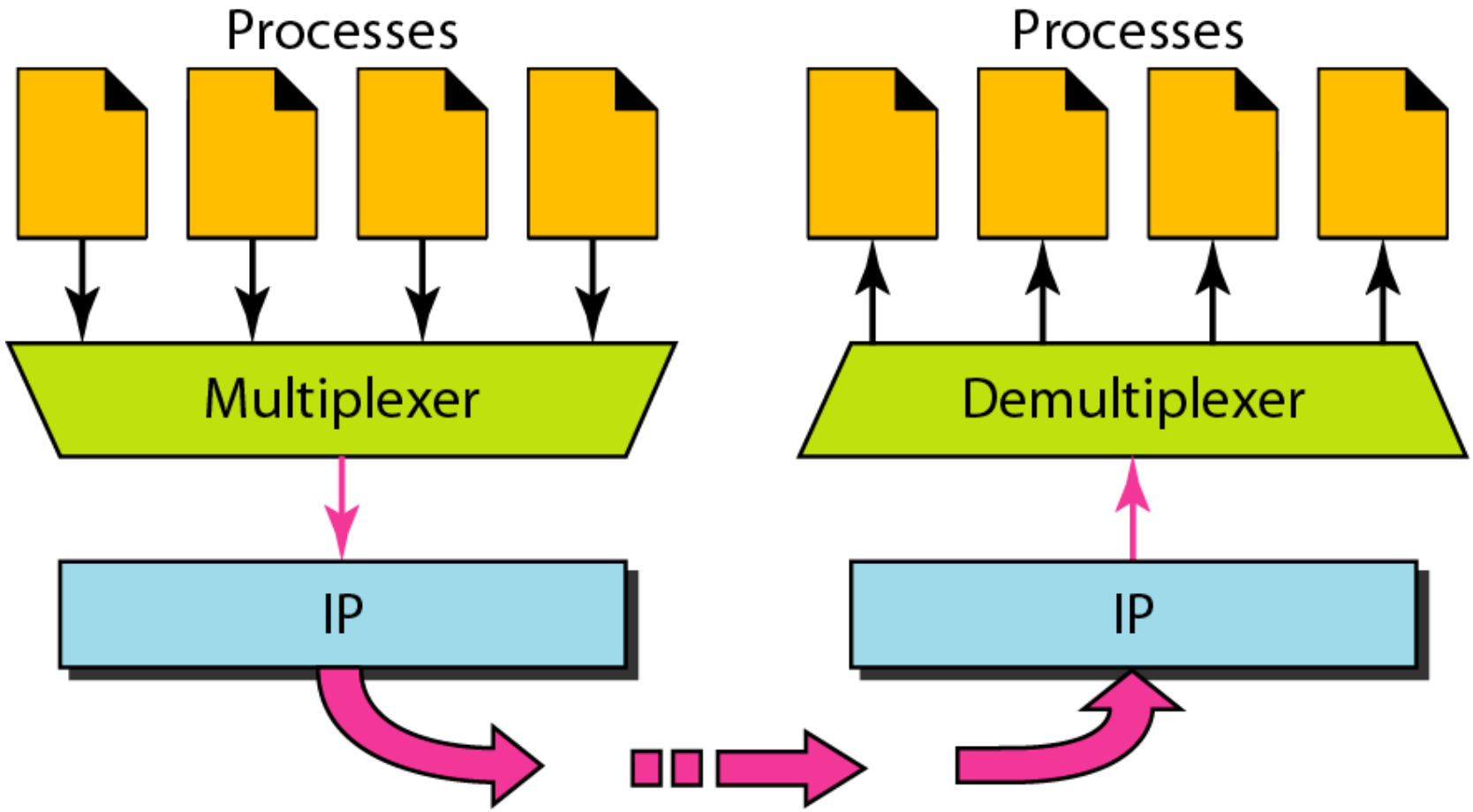


A socket provides an interface to send data to/from the network through a port

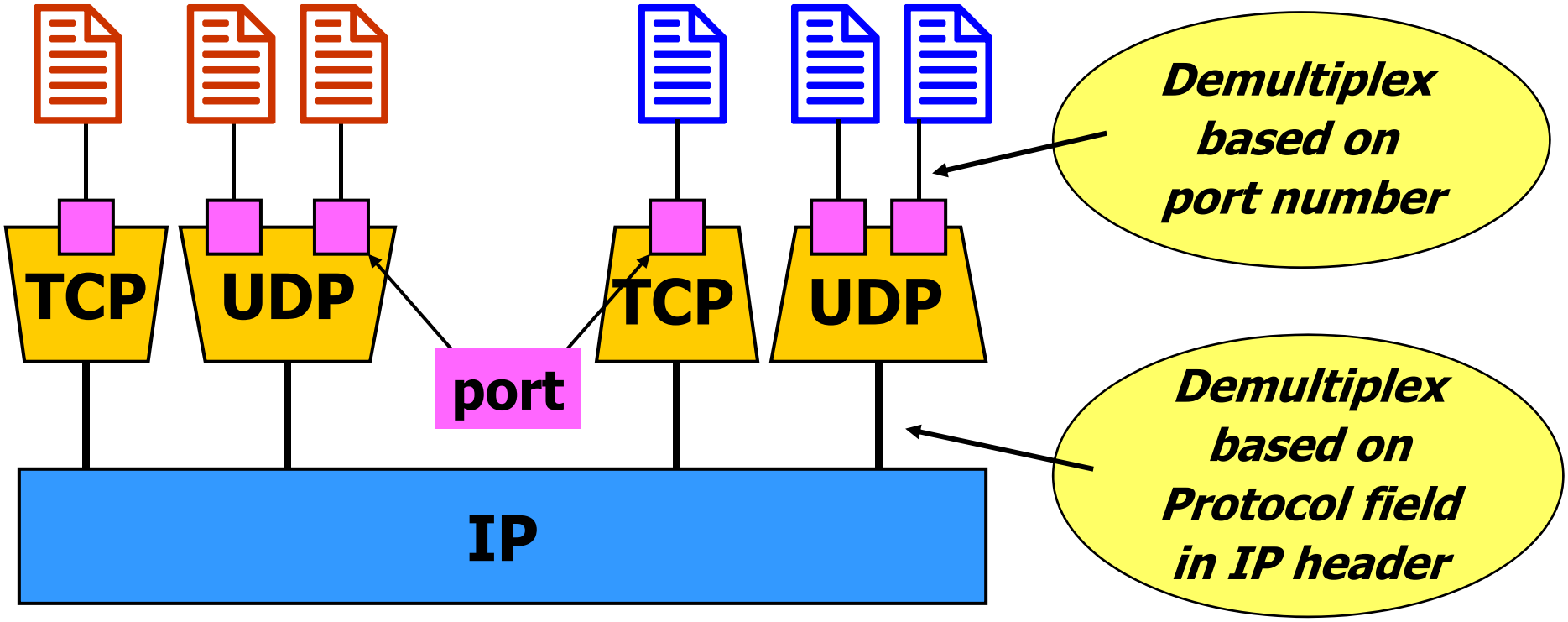
Multiplexing and Demultiplexing

- **Multiplexing**
  - Sender side may have several processes that need to send packets (albeit only 1 transport-layer protocol)
- **Demultiplexing**
  - At receiver side, after error checking and header dropping, transport-layer delivers each message to the appropriate process

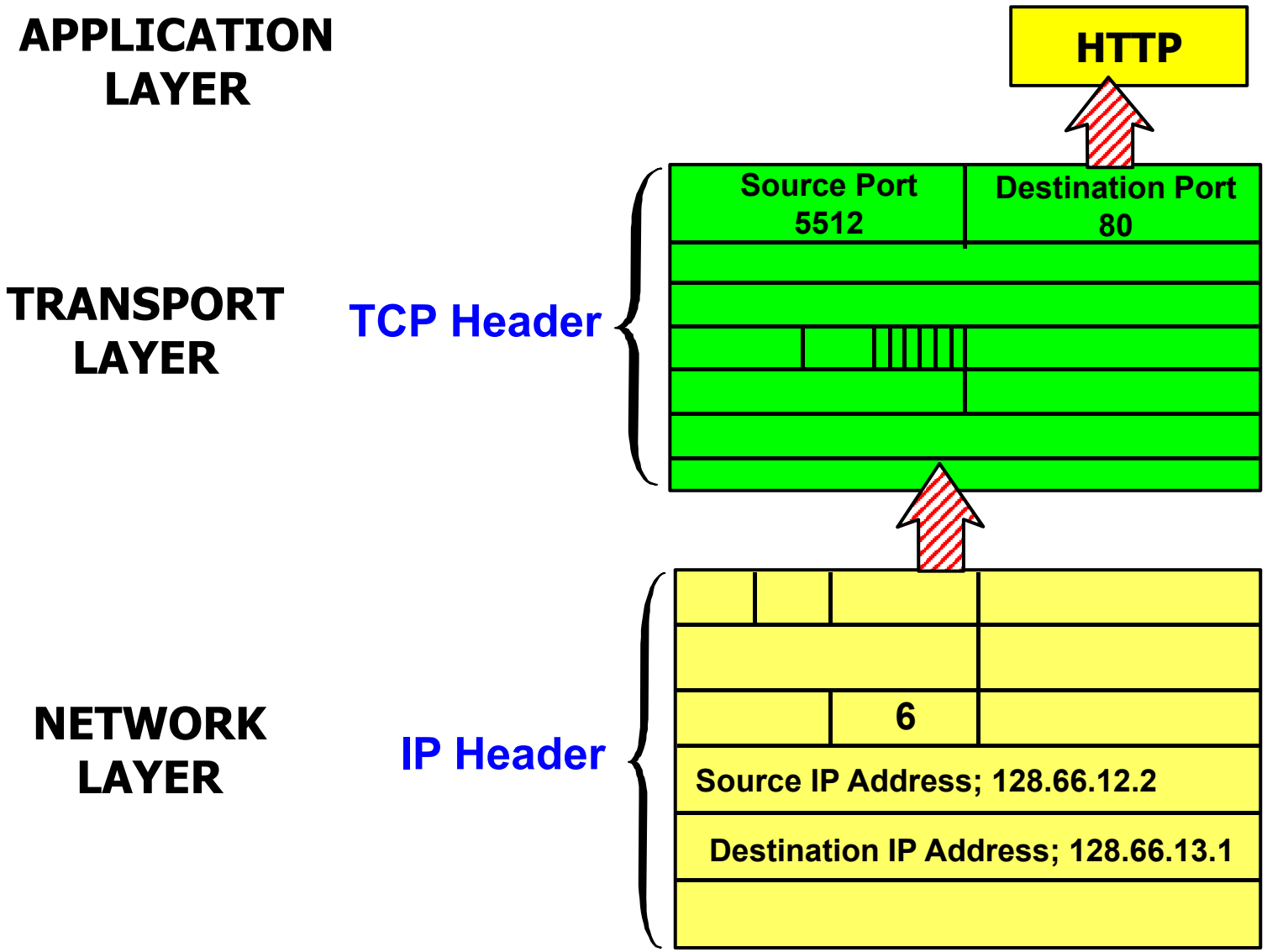
Multiplexing and Demultiplexing



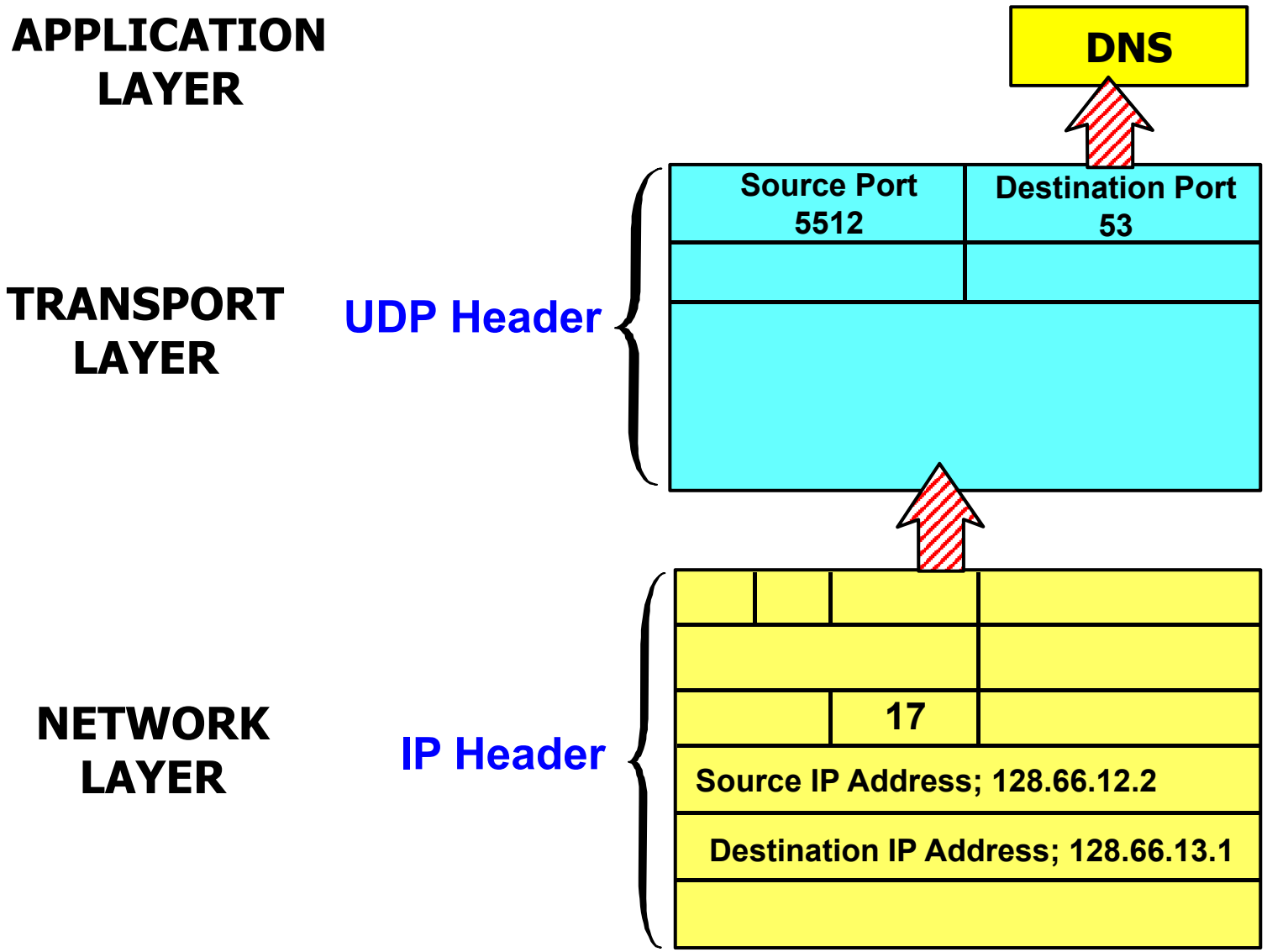
Multiplexing and Demultiplexing



Multiplexing and Demultiplexing



Multiplexing and Demultiplexing



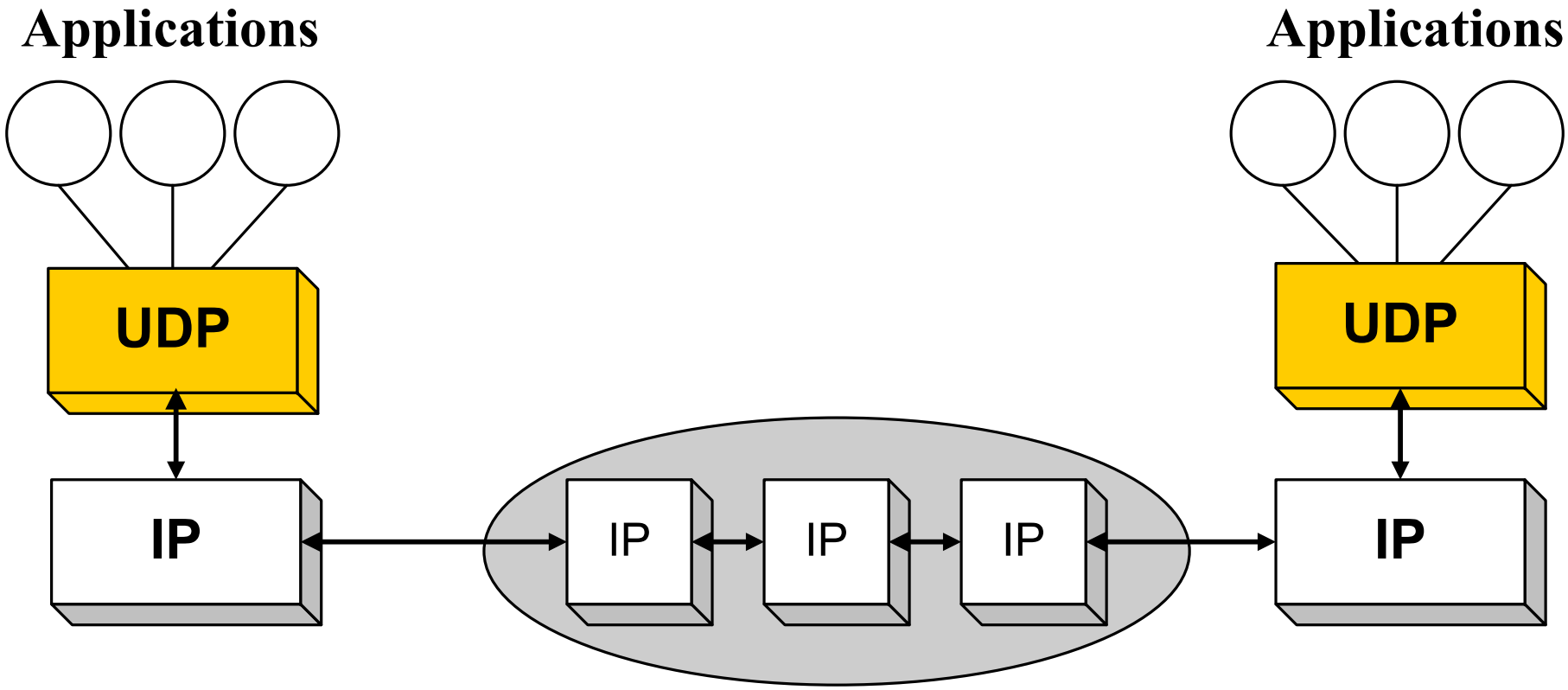
Chapter 6: Roadmap

- 6.1 Transport Services
- 6.2 Elements of transport protocol
- 6.3 Simple transport protocol
- 6.4 The Internet Transport Layer
  - UDP
  - TCP

UDP: User Datagram Protocol [RFC 768]

- "best effort" transport protocol
  - unreliable
  - connectionless
    - no handshaking between UDP sender, receiver before packets start being exchanged
    - each UDP packet handled independently of others
  - Just provides multiplexing and demultiplexing

UDP: User Datagram Protocol [RFC 768]



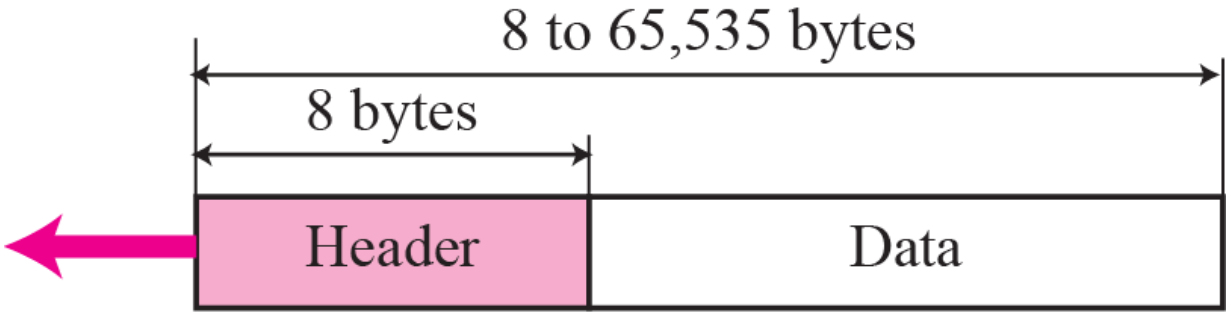
The only thing that UDP adds is multiplexing and demultiplexing

UDP: User Datagram Protocol [RFC 768]

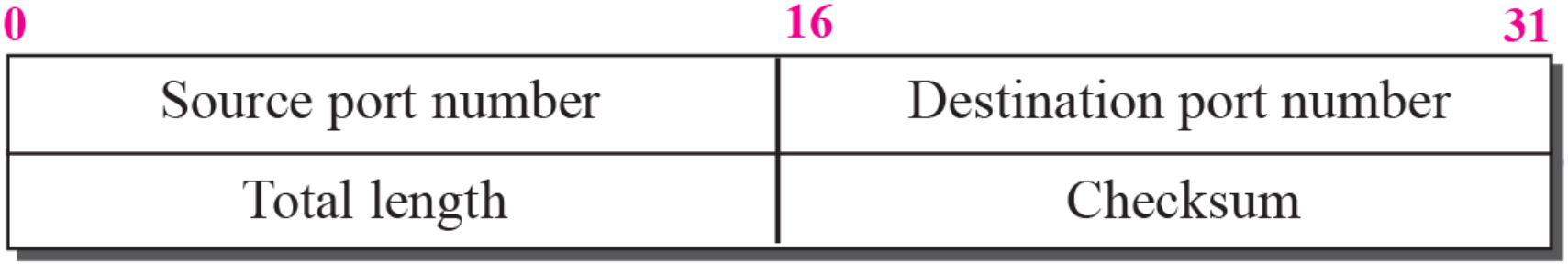
- Often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- Other UDP uses
  - DNS
  - SNMP

UDP: User Datagram Protocol [RFC 768]

- **UDP packets, called user datagrams, have a fixed-size header of 8 bytes.**

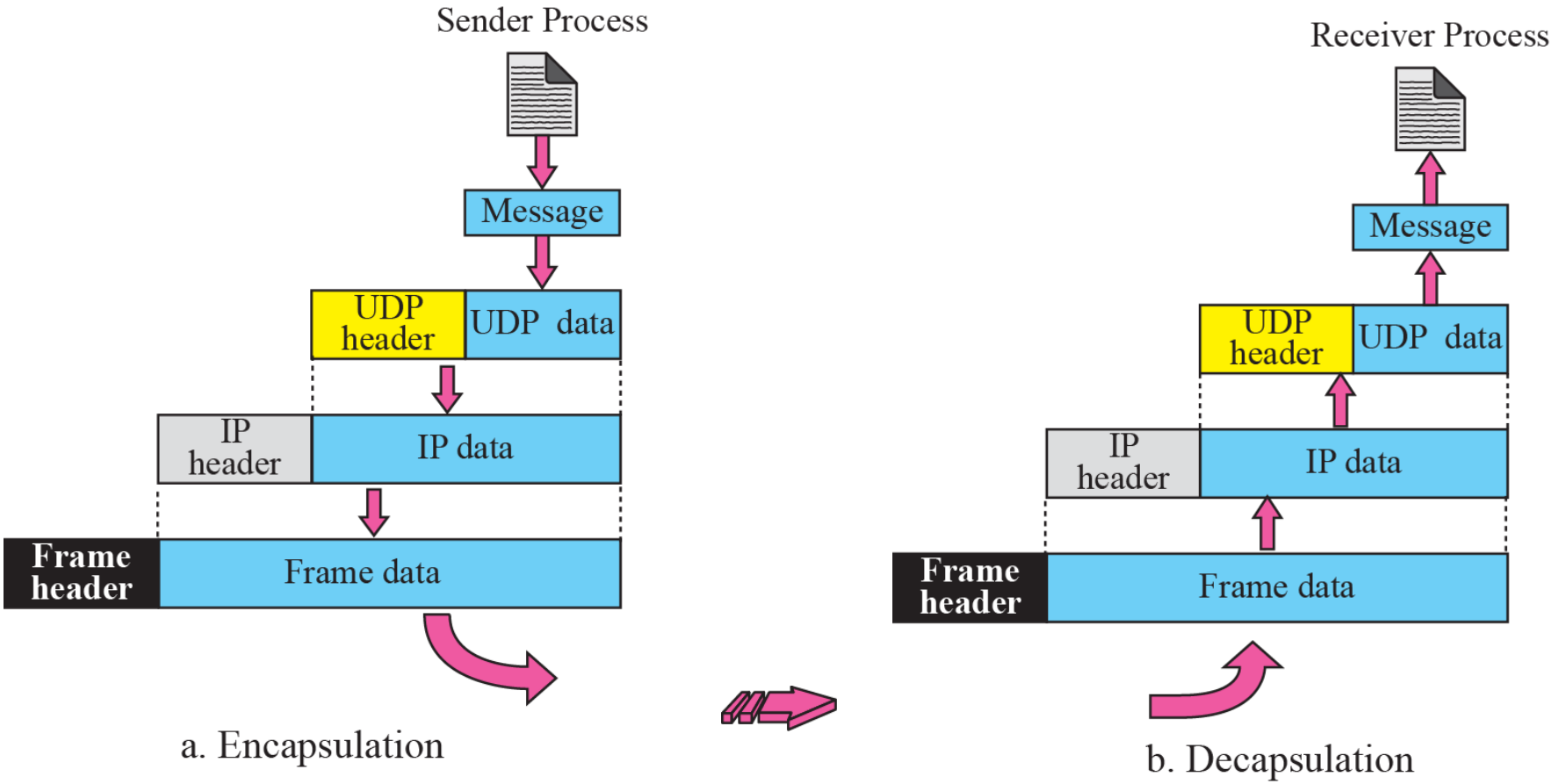


a. UDP user datagram



b. Header format

UDP: User Datagram Protocol [RFC 768]



Encapsulation and decapsulation

UDP: User Datagram Protocol [RFC 768]

Note

UDP length
= IP length – IP header's length

Example

The following is a dump of a UDP header in hexadecimal format.

CB84000D001C001C

a. What is the source port number?
b. What is the destination port number?
c. What is the total length of the user datagram?
d. What is the length of the data?
e. Is the packet directed from a client to a server or vice versa?

Example

Solution

a. The source port number is the first four hexadecimal digits $(CB84)_{16}$ or 52100.

b. The destination port number is the second four hexadecimal digits $(000D)_{16}$ or 13.

c. The third four hexadecimal digits $(001C)_{16}$ define the length of the whole UDP packet as 28 bytes.

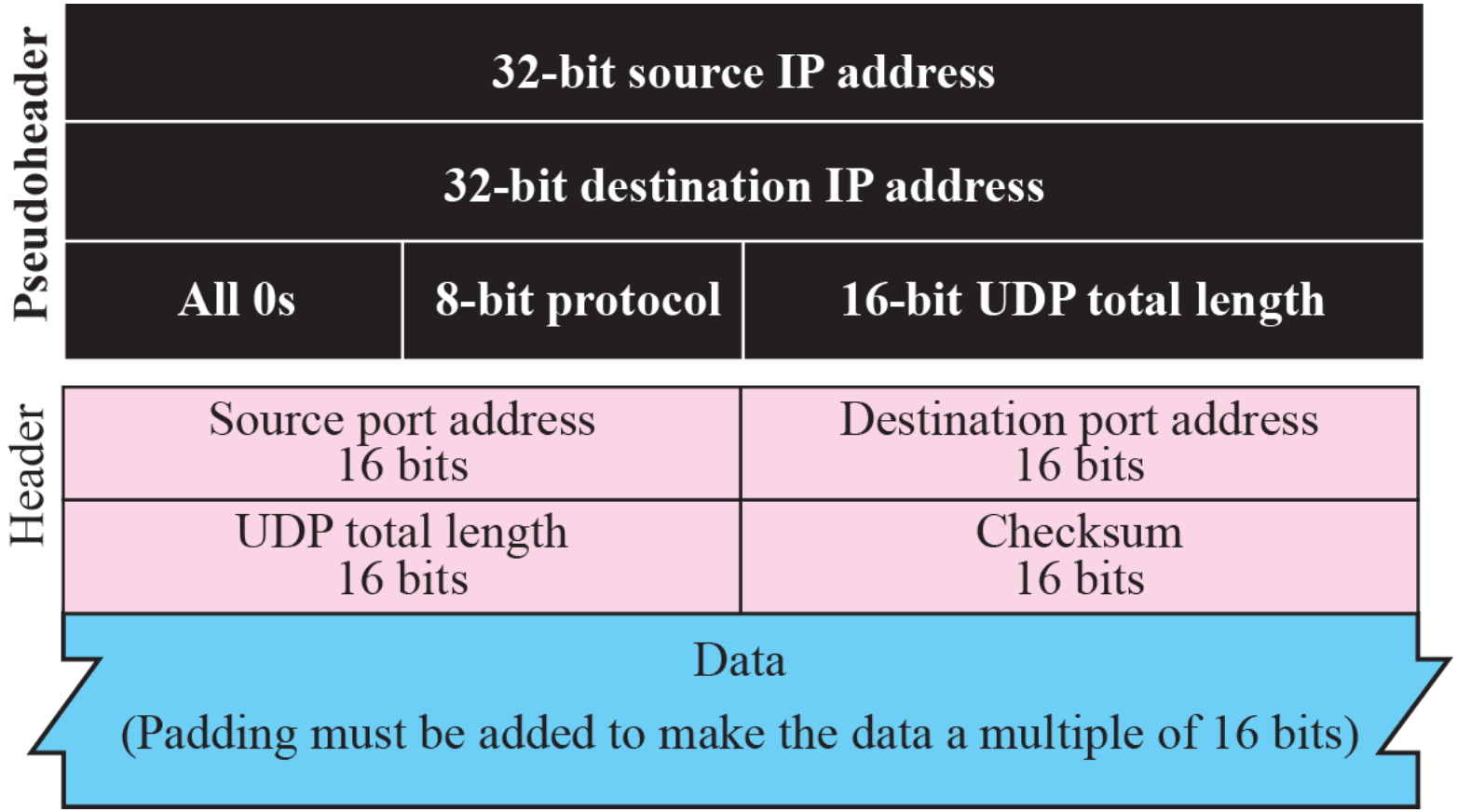
d. The length of the data is the length of the whole packet minus the length of the header, or 28 – 8 = 20 bytes.

e. Since the destination port number is 13 (well-known port), the packet is from the client to the server.

UDP: Well-known Port Number

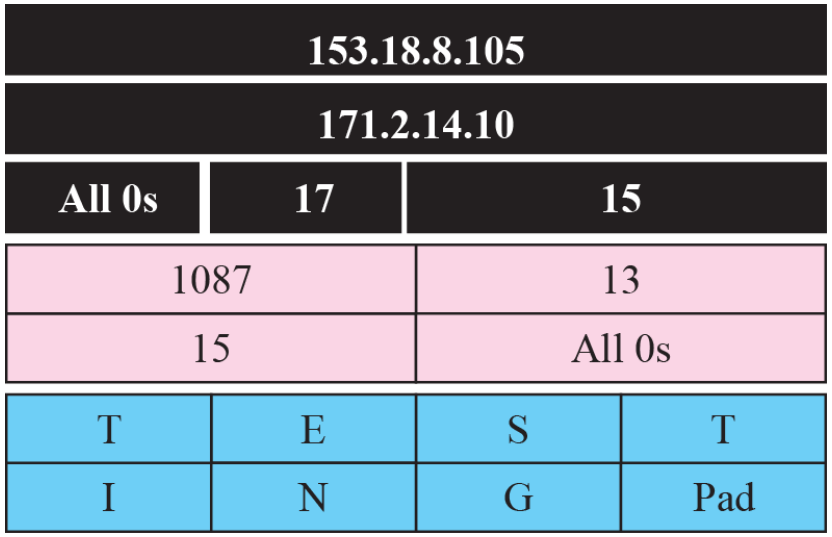
Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Domain	Domain Name Service (DNS)
67	Bootps	Server port to download bootstrap information
68	Bootpc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
111	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

UDP: Checksum



Pseudoheader for checksum calculation

UDP: checksum

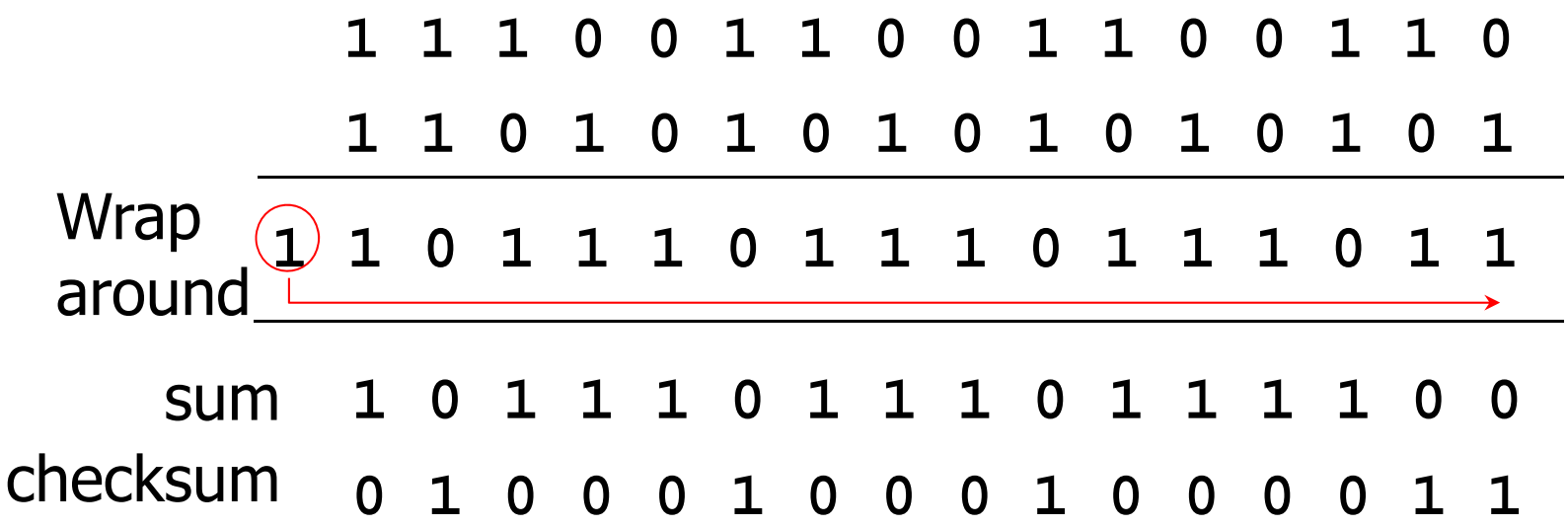


10011001	00010010	153.18
00001000	01101001	8.105
10101011	00000010	171.2
00001110	00001010	14.10
00000000	00010001	0 and 17
00000000	00001111	15
00000100	00111111	1087
00000000	00001101	13
00000000	00001111	15
00000000	00000000	0 (checksum)
01010100	01000101	T and E
01010011	01010100	S and T
01001001	01001110	I and N
01000111	00000000	G and 0 (padding)
10010110	11101011	Sum
01101001	**00010100**	Checksum

Checksum calculation for a simple UDP user datagram

Internet checksum: an example

Example: add two 16-bit integers

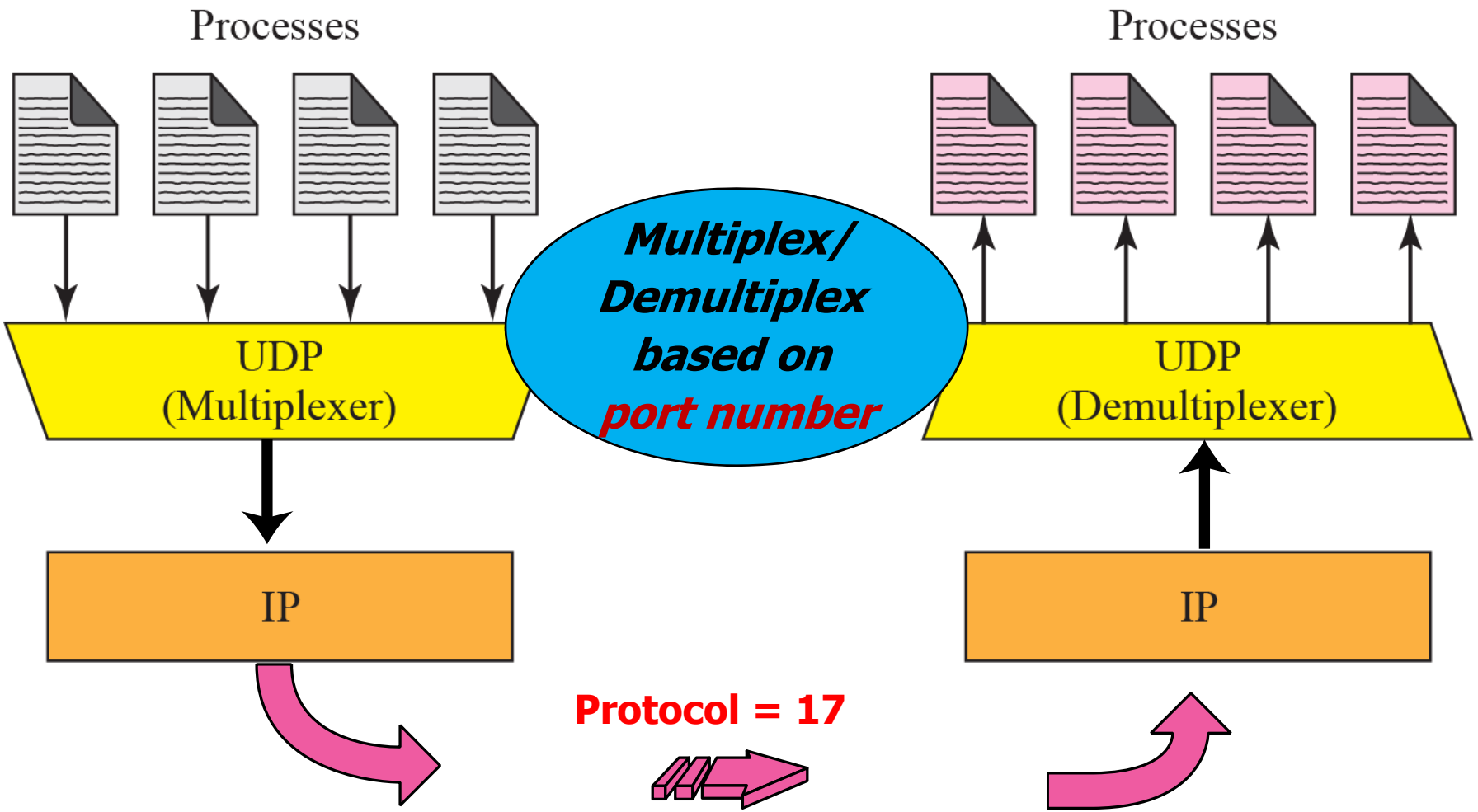


Note: when adding numbers, a carryout from the most significant bit needs to be added to the result.

Internet checksum: weak protection!

Example: add two 16-bit integers

UDP: Multiplexing and demultiplexing

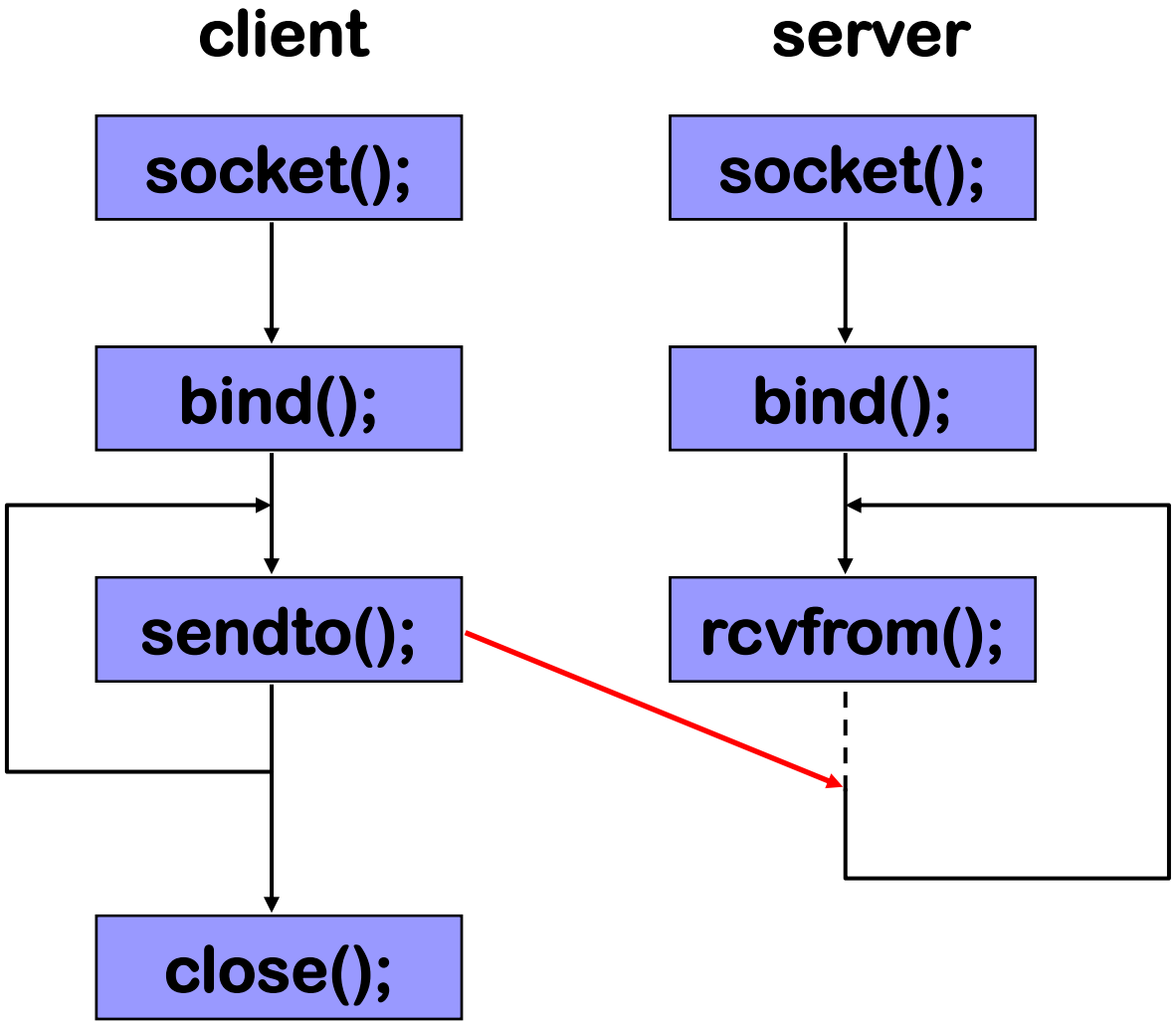


Multiplexing and demultiplexing

The UDP service

- **UDP service interface**
  - one message, up to 8K
  - destination address, destination port, source address, source port
- **UDP service is message oriented**
  - delivers exactly the message or nothing
  - several messages may be delivered in disorder
  - Message may be lost, application must implement loss recovery.
- **If a UDP message is larger than MTU, then fragmentation occurs at the IP layer**

UDP is used via a Socket Library



UDP Client/Server Interaction

Server starts by getting ready to receive client messages…

Client

1. Create a UDP socket
2. Communicate (send/receive messages)
3. When done, close the socket

Server

1. Create a UDP socket
2. Assign a port to socket
3. Communicate (receive/send messages)
4. When done, close the socket

UDP Client/Server Interaction

```
/* Create socket for incoming messages */
if ((servSock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    Error("socket() failed");
```

Client

1. Create a UDP socket
2. Communicate (send/receive messages)
3. When done, close the socket

Server

1. Create a UDP socket
2. Assign a port to socket
3. Communicate (receive/send messages)
4. When done, close the socket

UDP Client/Server Interaction

```
ServAddr.sin_family = PF_INET;                    /* Internet address family */
   ServAddr.sin_addr.s_addr = htonl(INADDR_ANY);     /* Any incoming interface */
   ServAddr.sin_port = htons(20000);                 /* Local port 20000 */
   if (bind(servSock, (struct sockaddr *) &ServAddr, sizeof(ServAddr)) < 0)
      Error("bind() failed");
```

Client

1. Create a UDP socket
2. Communicate (send/receive messages)
3. When done, close the socket

Server

1. Create a UDP socket
2. Assign a port to socket
3. Communicate (receive/send messages)
4. When done, close the socket

Specifying Addresses

Generic

- struct sockaddr
  {
  unsigned short sa_family; /* Address family (e.g., PF_INET) */
  char sa_data[14]; /* Protocol-specific address information */
  };

IP Specific

- struct sockaddr_in
  {
  unsigned short sin_family; /* Internet protocol (PF_INET) */
  unsigned short sin_port; /* Port (16-bits) */
  struct in_addr sin_addr; /* Internet address (32-bits) */
  char sin_zero[8]; /* Not used */
  };
- struct in_addr
  {
  unsigned long s_addr; /* Internet address (32-bits) */
  };

UDP Client/Server Interaction

```
struct sockaddr_in  peer;
    int peerSize = sizeof(peer);
    char buffer[65536];

    recvfrom(servSock, buffer, 65536, 0, (struct sockaddr *)&peer, &peerSize);
```

Client

1. Create a UDP socket
2. Communicate (send/receive messages)
3. When done, close the socket

Server

1. Create a UDP socket
2. Assign a port to socket
3. Communicate (receive/send messages)
4. When done, close the socket

UDP Client/Server Interaction

Server is now blocked waiting for a message from a client

Client

1. Create a UDP socket
2. Communicate (send/receive messages)
3. When done, close the socket

Server

1. Create a UDP socket
2. Assign a port to socket
3. Communicate (receive/send messages)
4. When done, close the socket

UDP Client/Server Interaction

Later, a client decides to talk to the server…

Client

1. Create a UDP socket
2. Communicate (send/receive messages)
3. When done, close the socket

Server

1. Create a UDP socket
2. Assign a port to socket
3. Communicate (receive/send messages)
4. When done, close the socket

UDP Client/Server Interaction

```
/* Create socket for outgoing messages */
    if ((clientSock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        Error("socket() failed");
```

Client

1. **Create a UDP socket**
2. **Communicate (send/receive messages)**
3. **When done, close the socket**

Server

1. **Create a UDP socket**
2. **Assign a port to socket**
3. **Communicate (receive/send messages)**
4. **When done, close the socket**

UDP Client/Server Interaction

```
// Initialize server's address and port
   struct sockaddr_in server;
   server.sin_family = AF_INET;
   server.sin_addr.s_addr = inet_addr("10.10.100.37");
   server.sin_port = htons(20000);
   // Send it to the server
   sendto(clientSock, buffer, msgSize, 0, (struct sockaddr *)&server, sizeof(server));
```

Client

1. Create a UDP socket
2. Communicate (send/receive messages)
3. When done, close the socket

Server

1. Create a UDP socket
2. Assign a port to socket
3. Communicate (receive/send messages)
4. When done, close the socket

UDP Client/Server Interaction

close(clientSock);

close(serverSock);

Client

1. Create a UDP socket
2. Communicate (send/receive messages)
3. When done, close the socket

Server

1. Create a UDP socket
2. Assign a port to socket
3. Communicate (receive/send messages)
4. When done, close the socket

Chapter 6: Roadmap

- 6.1 Transport Services
- 6.2 Elements of transport protocol
- 6.3 Simple transport protocol
- 6.4 The Internet Transport Layer
  - UDP
  - TCP

TCP: Transmission Control Protocol

- Originally described in RFC 793, 1981
- Connection-oriented protocol
  - it creates a virtual connection between two TCPs to send data.
    - Reliable
    - In-order
  - In addition, TCP uses flow and error control mechanisms at the transport level.

TCP Services

- Process-to-Process Communication
- Multiplexing and Demultiplexing
- Connection-Oriented Service
- Byte Stream Delivery Service
- Reliable, in-order Delivery Service
- Full-Duplex Communication

TCP Features

- **Connection-oriented**
- **Byte-stream**
  - app writes bytes
  - TCP sends *segments*
  - app reads bytes
- **Reliable and in-order data transfer**
- **Full duplex**
- **Flow control**: keep sender from overrunning receiver
- **Congestion control**: keep sender from overrunning network

TCP: Well-known Port Number

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
20 and 21	FTP	File Transfer Protocol (Data and Control)
23	TELNET	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol

Multiplexing and Demultiplexing

- **Like** UDP, TCP uses *protocol ports* to identify the ultimate destination within a machine
- **Unlike** UDP, TCP defines a *connection* as the fundamental abstraction for data transfer
- Connection:
  - *Virtual circuit*
  - Identified by a pair of *endpoints*

TCP Connection Endpoint

- An *endpoint* for a TCP connection is defined by a socket - a (*host, port*) pair
  - *Host* = the IP address of for a host
  - *Port* = a TCP port on that host
- A TCP *connection* is defined by a pair of endpoints (4-tuple) :
  - Port 1037 on www.redhouse.gov and port 76 on www.cs.openuniv.edu:

    (198.137.240.91, 1037) and (128.143.137.17, 76)

TCP Connection Endpoint

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

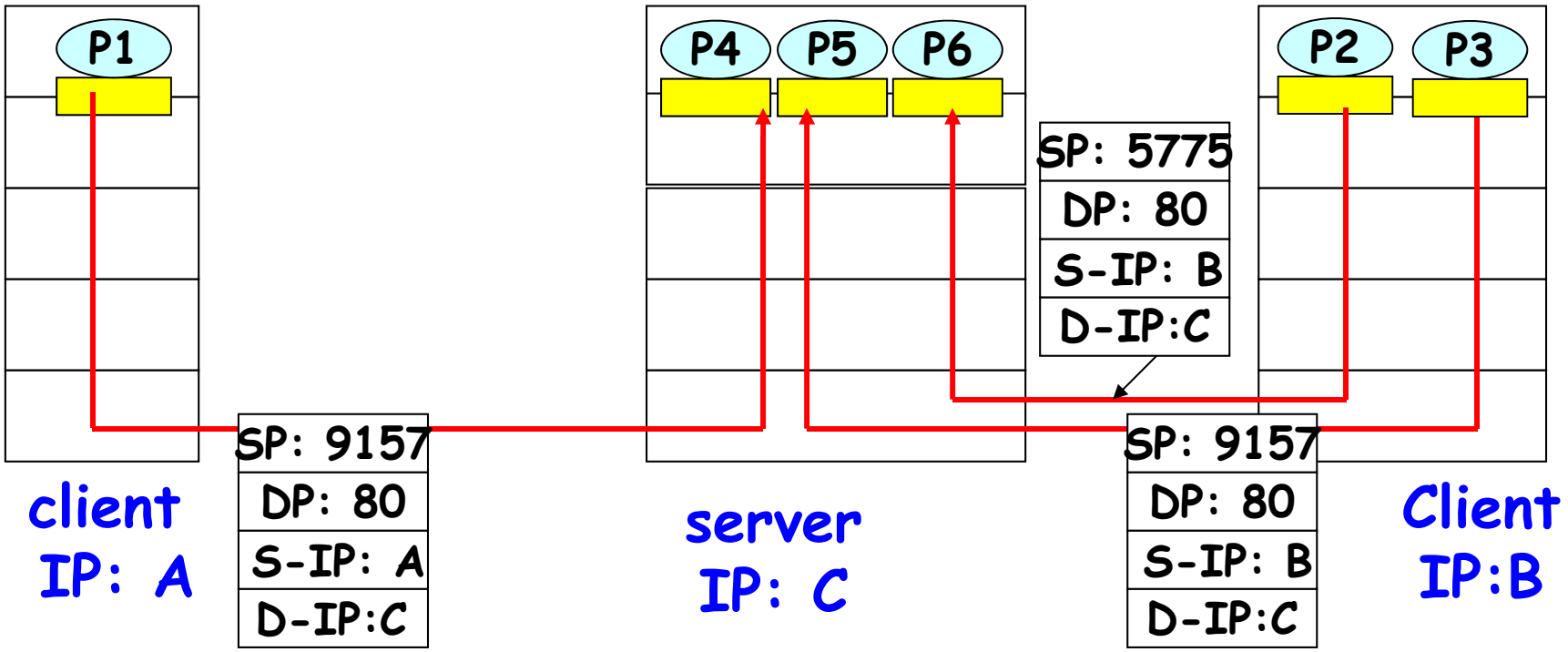
TCP Connection Endpoint

- **Multiple** connections to same port (e.g. server):
  - Port 1037 on www.redhouse.gov and port 444 on on www.cs.openuniv.edu :

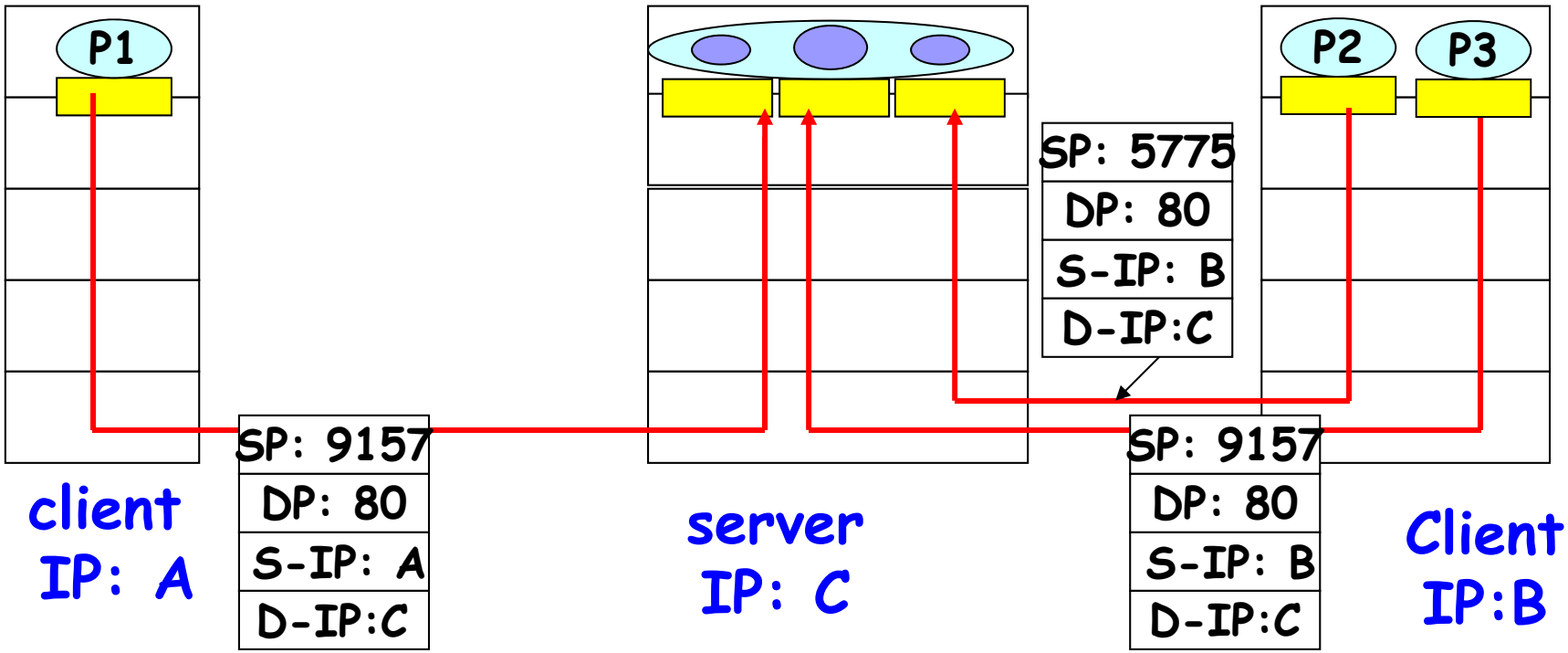
    (198.137.240.91, 1037) and (128.143.137.17, 444)
  - Port 3553 on falcon.cs.jmu.edu and port 444 on on www.cs.openuniv.edu :

    (134.126.10.30, 3553) and (128.143.137.17, 444)
- **No ambiguity** - connection identified by **both** endpoints

Connection-oriented demux



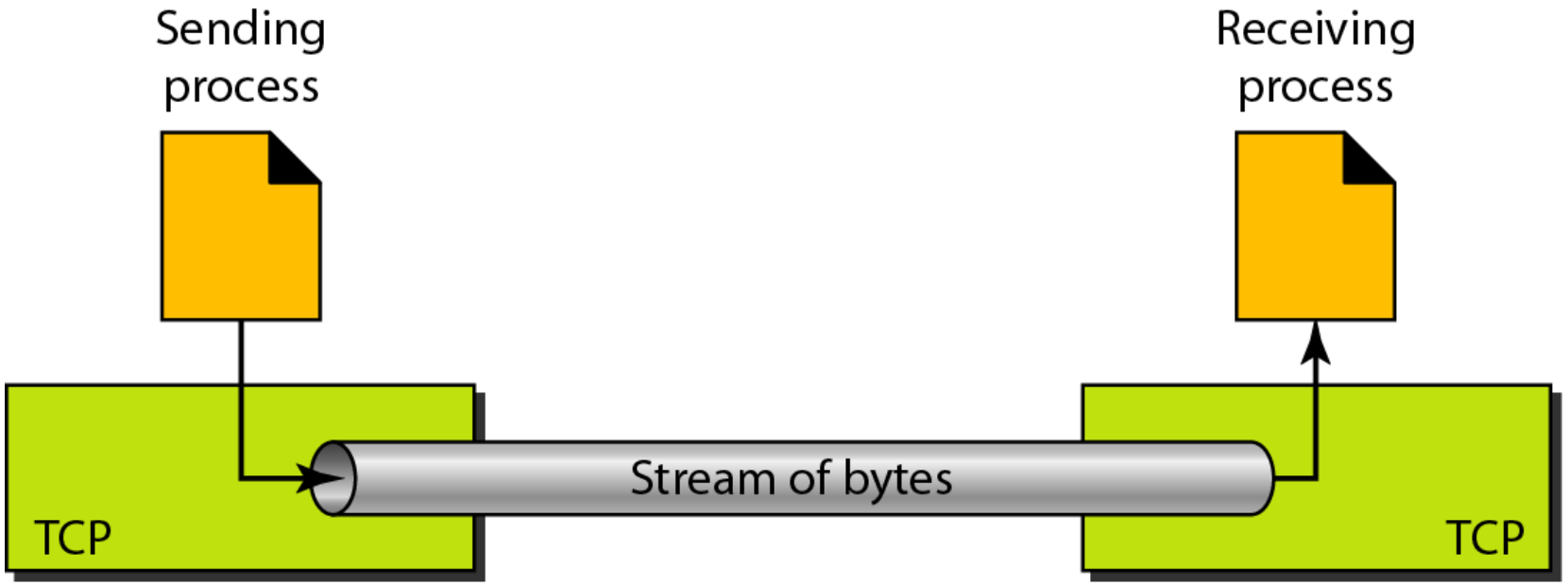
Connection-oriented demux: Threaded Server



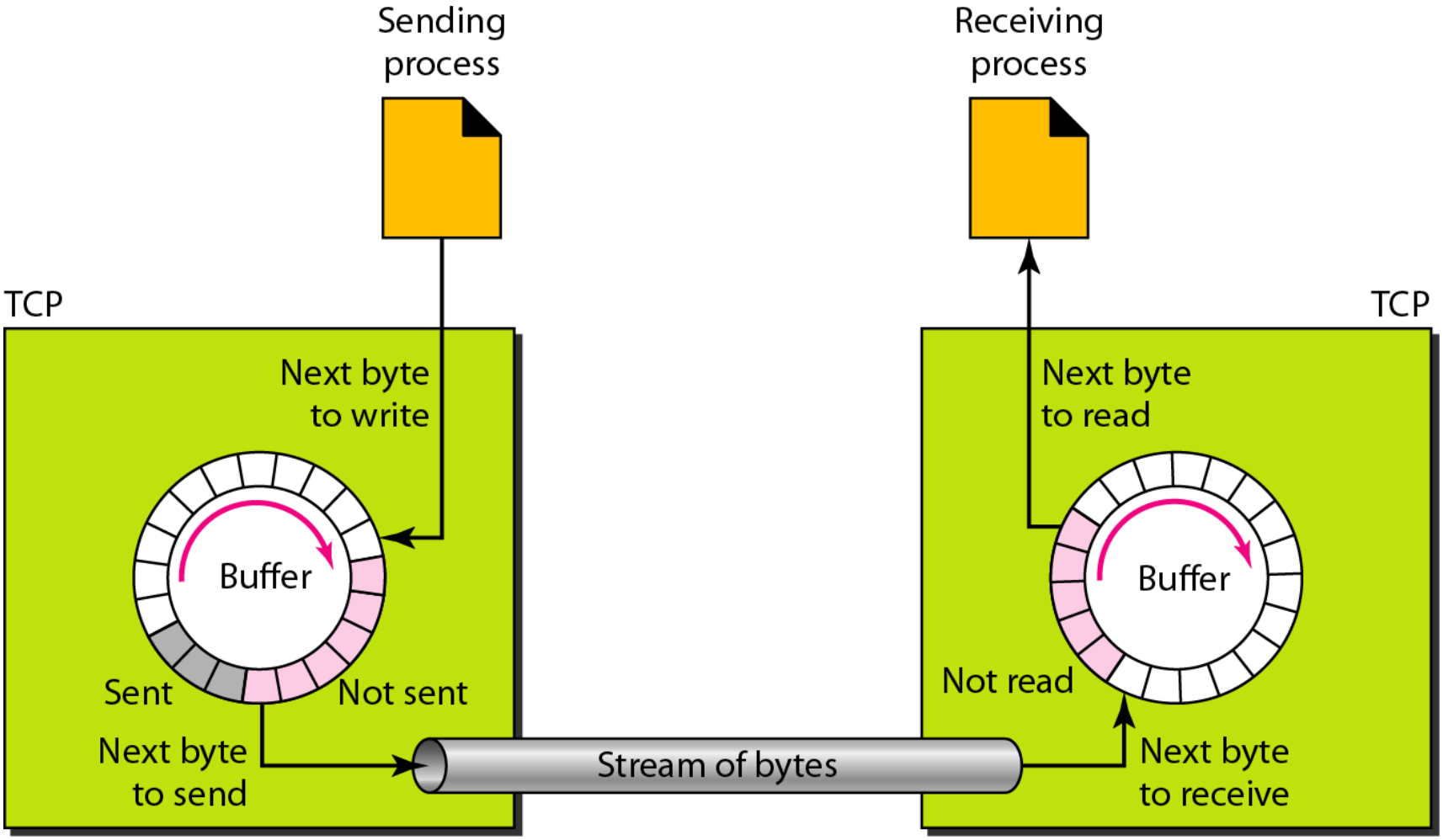
TCP Support for Reliable Delivery

- **Checksum**
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet
- **Sequence numbers**
  - Used to detect missing data
  - … and for putting the data back in order
- **Retransmission**
  - Sender retransmits lost or corrupted data
  - Timeout based on estimates of round-trip time
  - Fast retransmit algorithm for rapid retransmission

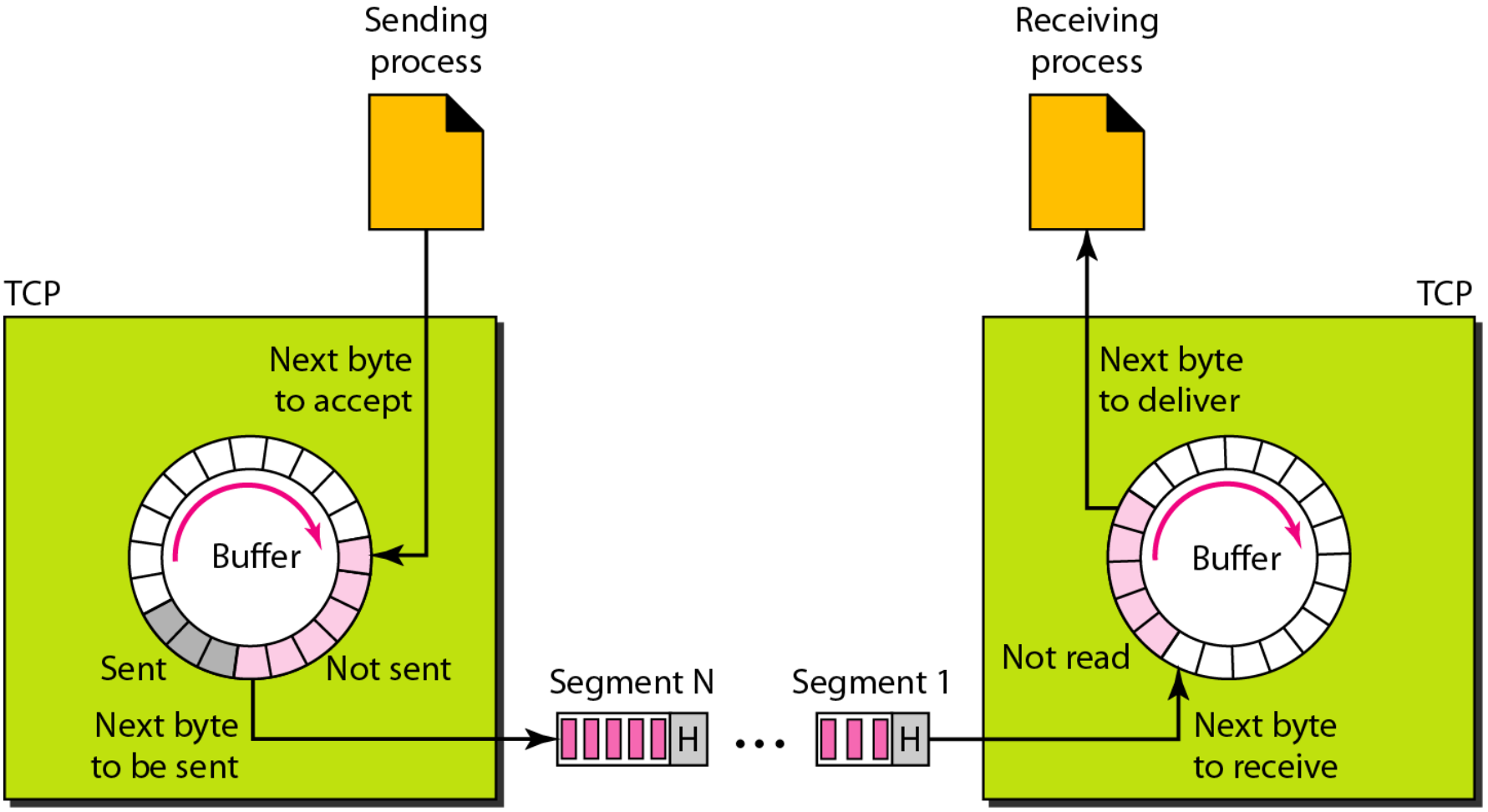
Stream delivery



Sending and receiving buffers



TCP segments



Sequence Number

- TCP breaks byte stream from application into segments (limited by Max. Segment Size - MSS)
- Each byte in the data stream is considered
- Each segment has a sequence number
- 32-bit sequence numbers
  - Wrap around supported

Sequence Number

Note

The bytes of data being transferred in each connection are numbered by TCP.

The numbering starts with a randomly generated number.

Sequence Number

Note

The value in the sequence number field of a segment defines the number of the first data byte contained in that segment.

Sequence Number

The following shows the sequence number for each segment (segment size=1000B):

Segment 1	→	Sequence Number: 10,001 (range: 10,001 to 11,000)
Segment 2	→	Sequence Number: 11,001 (range: 11,001 to 12,000)
Segment 3	→	Sequence Number: 12,001 (range: 12,001 to 13,000)
Segment 4	→	Sequence Number: 13,001 (range: 13,001 to 14,000)
Segment 5	→	Sequence Number: 14,001 (range: 14,001 to 15,000)

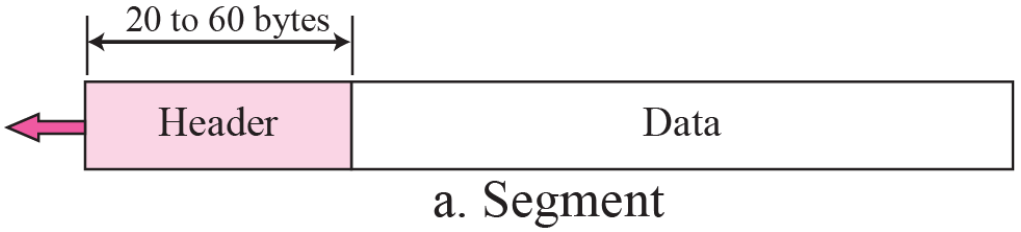
Sequence Number

Note

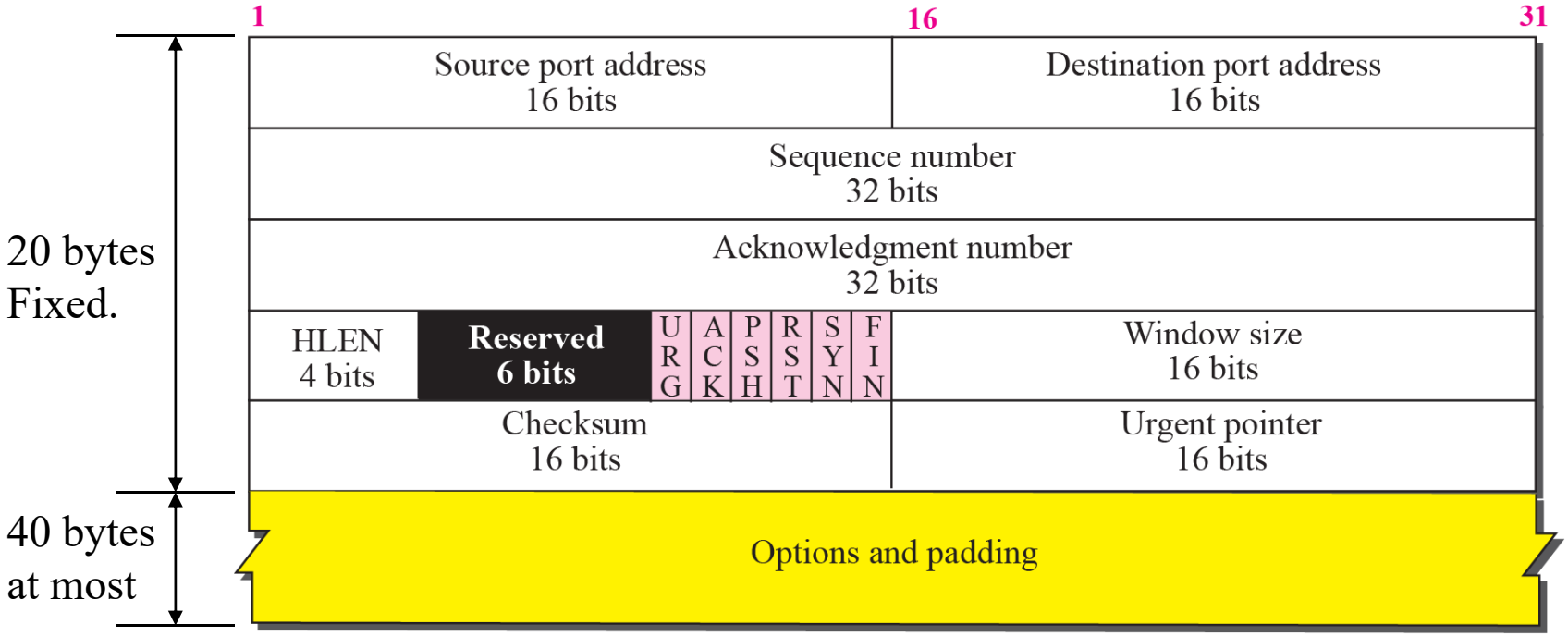
The value of the acknowledgment field in a segment defines the number of the next byte a party expects to receive.

The acknowledgment number is cumulative.

TCP Segment Format



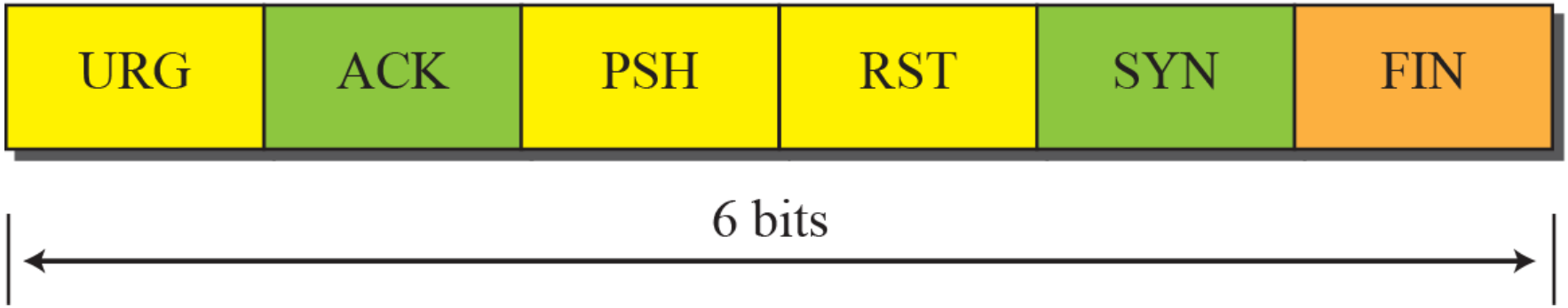
a. Segment



b. Header

Control field

URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection



TCP Segment Format

Example

The following is a dump of a TCP header in hexadecimal format :

05320017 00000001 00000000 500207FF 00000000

What is the source port number?
What is the destination port number?
What is sequence number?
What is the acknowledgment number?
What is the length of the header?
What is the type of the segment?
What is the window size?

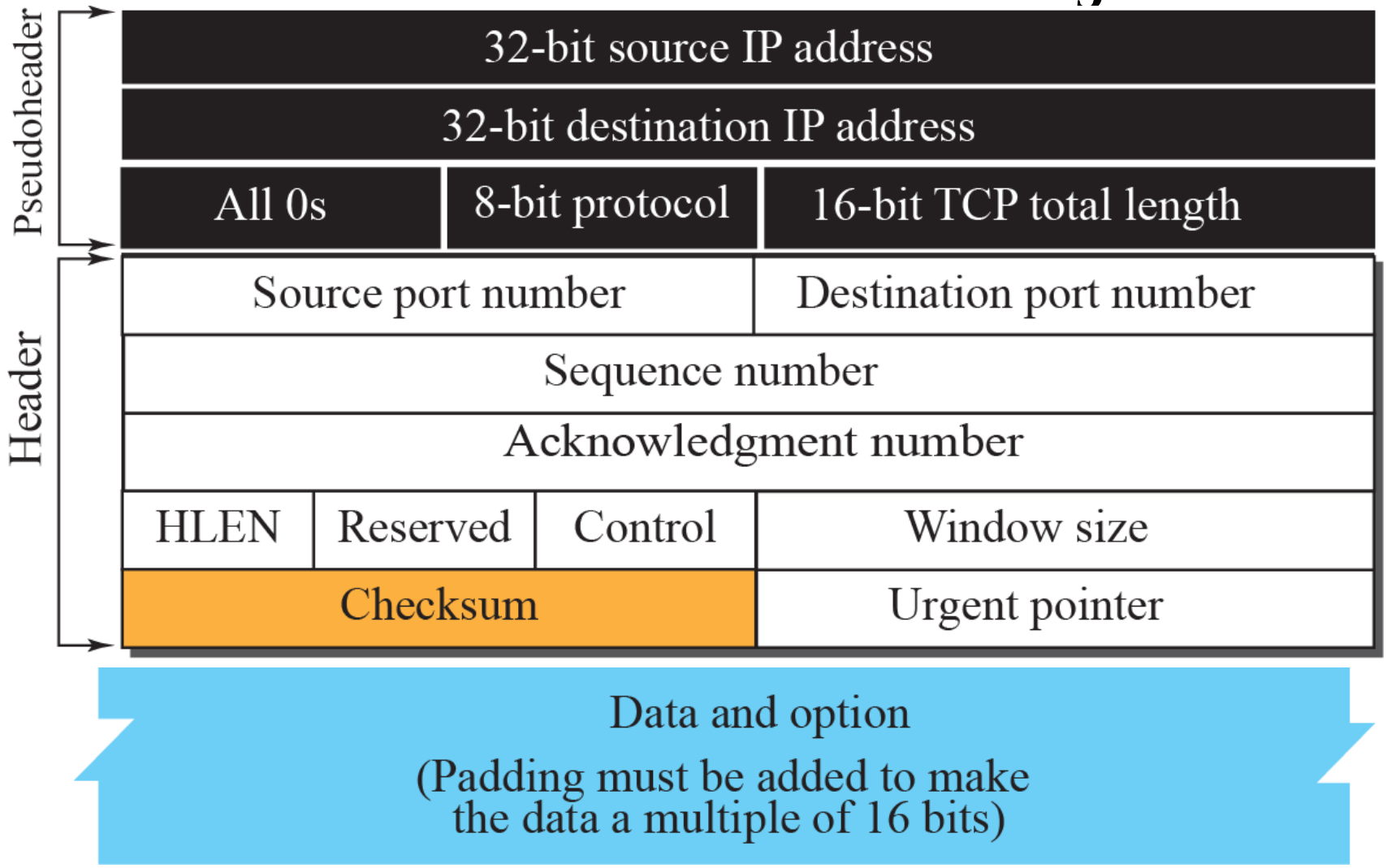
TCP Segment Format

Example

To make the initial sequence number, a random number, most systems start the counter at 1 and increase the counter by 64000 every 0.5s, how long does it take for the counter to wrap around?

TCP Checksum

Pseudoheader added to the TCP segment

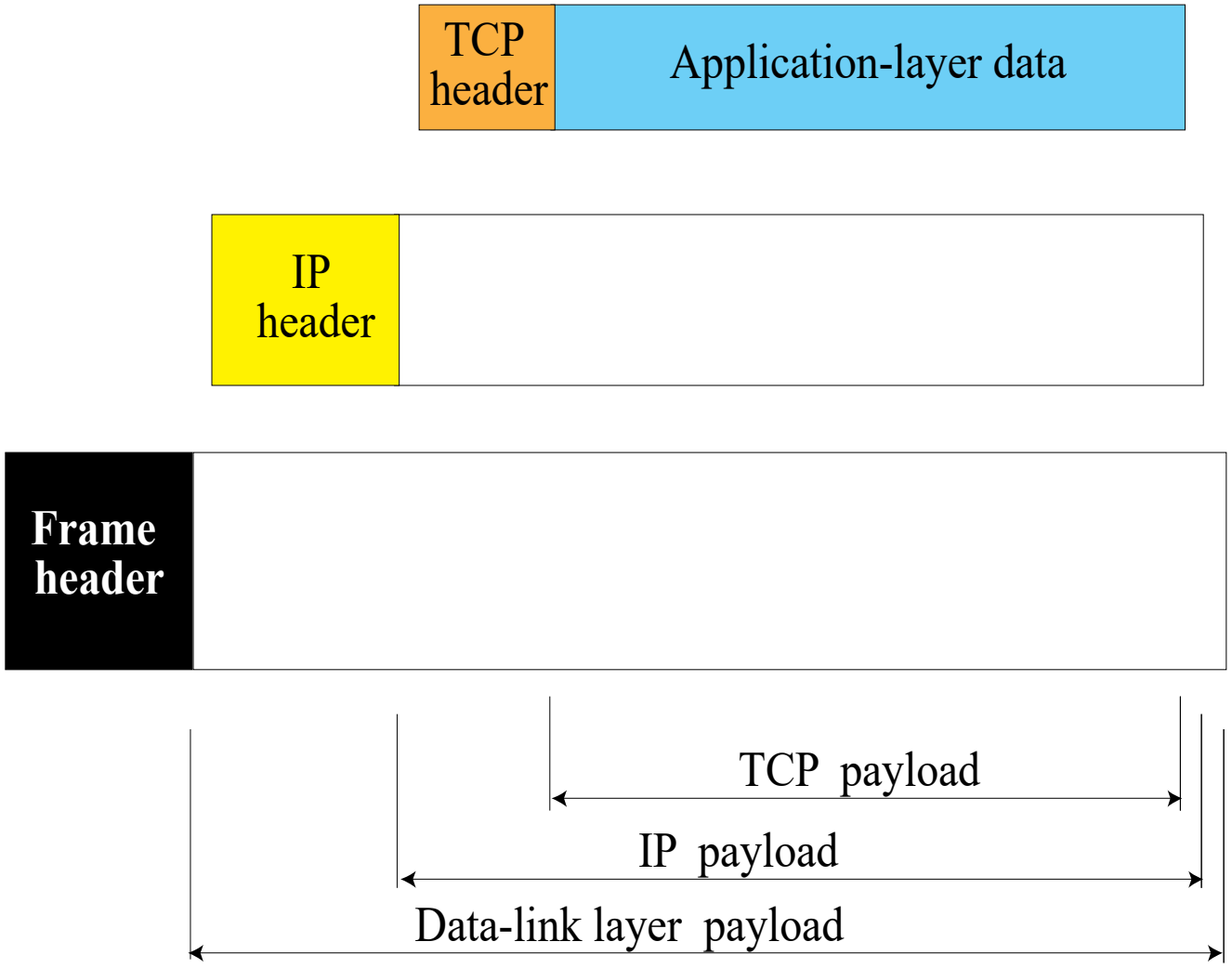


TCP Checksum

Note

The use of the checksum in TCP is mandatory.

Encapsulation

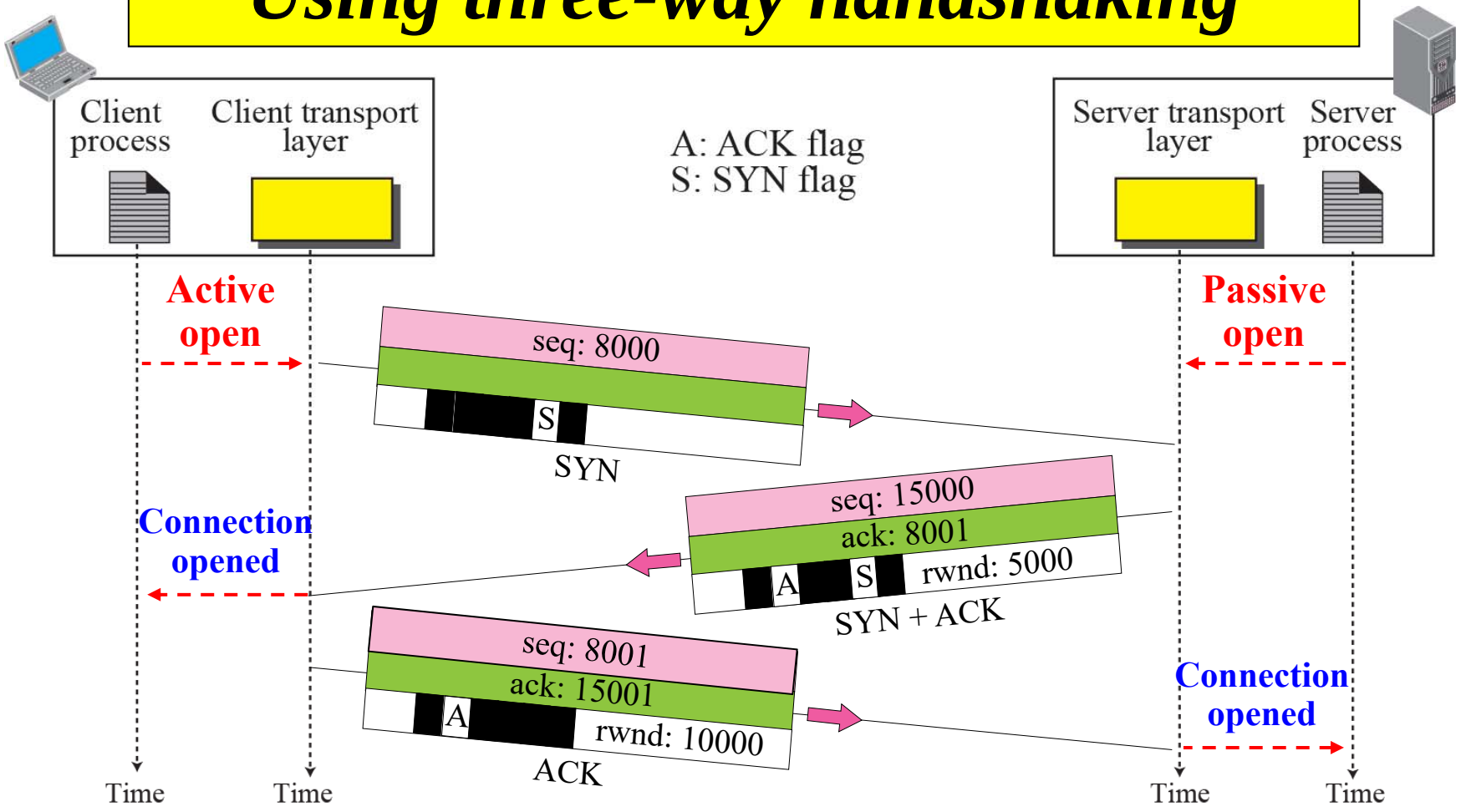


TCP Connection

- TCP is connection-oriented.
- All of the segments belonging to a message are then sent over this virtual path.
- Connection Establishment
- Data Transfer
- Connection Termination
- Connection Reset

Connection Establishment

Using three-way handshaking



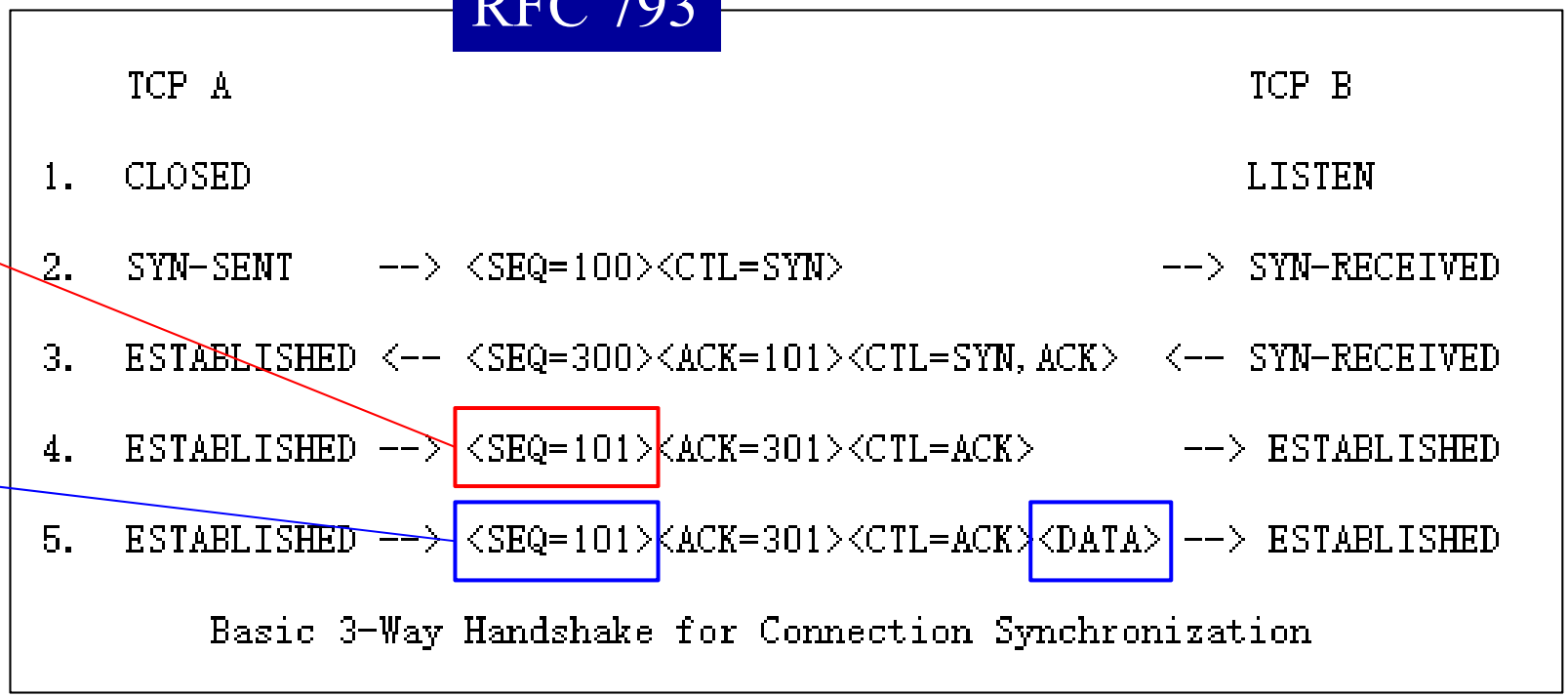
Connection Establishment

Using three-way handshaking

TCP A responds with an empty segment containing an ACK

TCP A sends some data

RFC 793

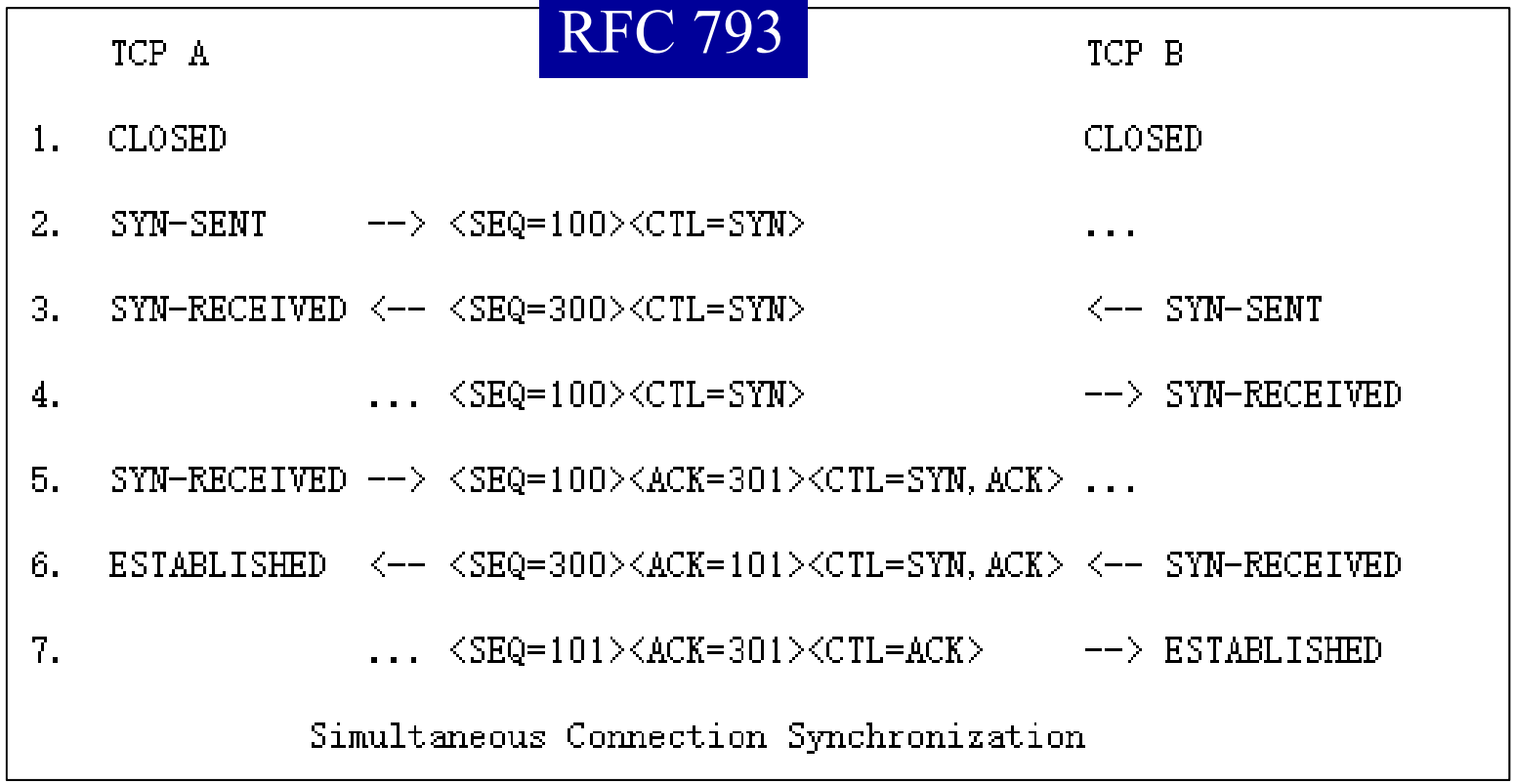


The sequence number of the segment in line 5 is the same as in line 4 because the ACK does not occupy sequence number space.

Connection Establishment

Using three-way handshaking

RFC 793

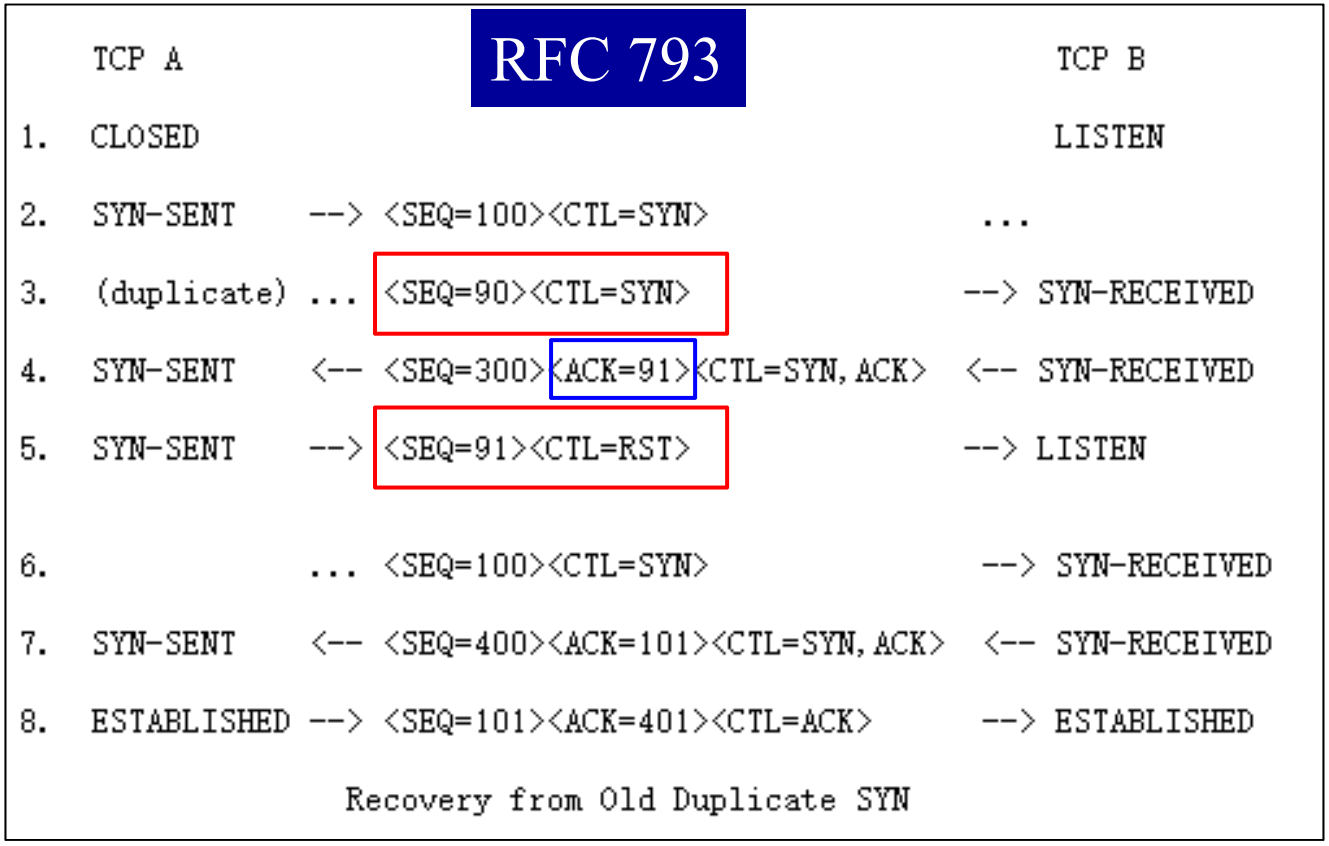


Simultaneous initiation is only slightly more complex.

Connection Establishment

Using three-way handshaking

RFC 793



The principle reason for the three-way handshake is to prevent old duplicate connection initiations from causing "half-open" connections.

Connection Establishment

- TCP uses a three-way handshake to open a connection:
  - (1) ACTIVE OPEN: Client sends a segment with
    - SYN bit set
    - port number of client
    - initial sequence number (ISN) of client
  - (2) PASSIVE OPEN: Server responds with a segment with
    - SYN bit set
    - initial sequence number of server
    - ACK for ISN of client
  - (3) Client Acknowledges by sending a segment with:
    - ACK ISN of server

Connection Establishment

Note

A SYN segment cannot carry data, but it consumes one sequence number.

A SYN + ACK segment cannot carry data, but does consume one sequence number.

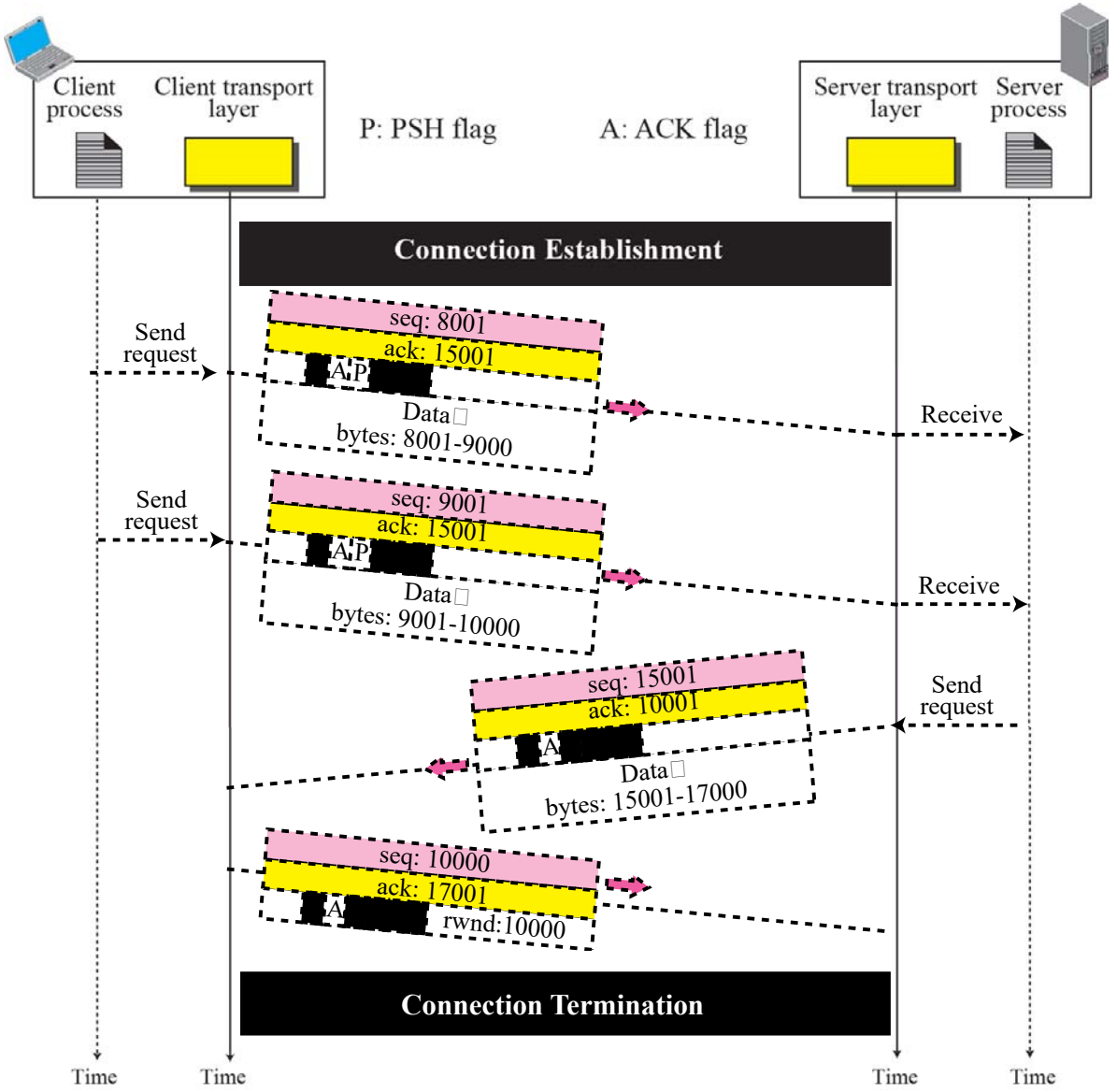
An ACK segment, if carrying no data, consumes no sequence number.

Connection Establishment

Example

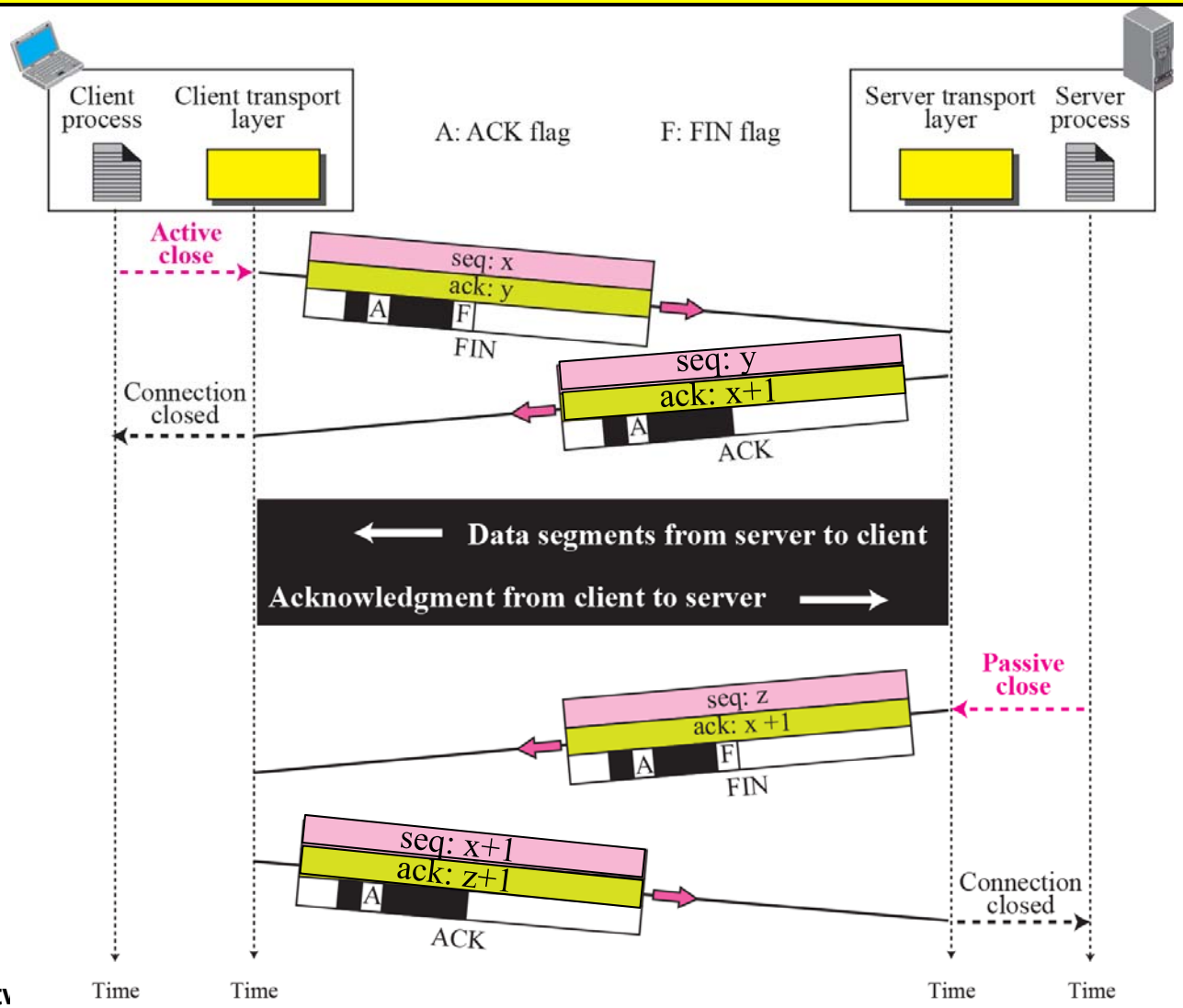
TCP opens a connections using an initial sequence number (ISN) of 14534. The other party opens the connection with an ISN of 21732. Show the three TCP segment during the connection establishment.

Data Transfer



Connection Termination

Half-Close, Four-way handshaking



Connection Termination

Half-Close, Four-way handshaking

RFC 793

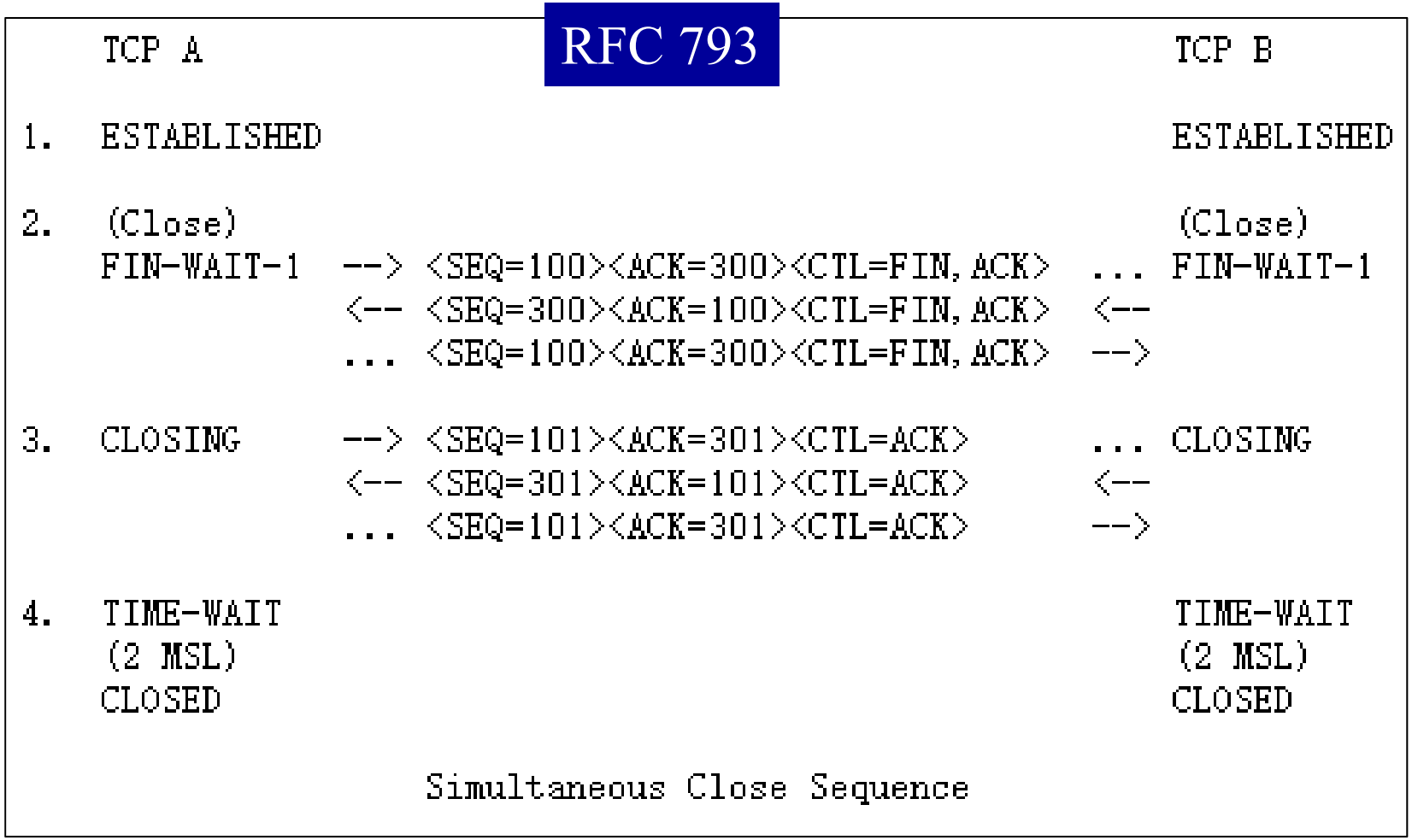
```
    TCP A                                                TCP B

1.  ESTABLISHED                                          ESTABLISHED

2.  (Close)
    FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  --> CLOSE-WAIT

3.  FIN-WAIT-2  <-- <SEQ=300><ACK=101><CTL=ACK>      <-- CLOSE-WAIT

4.                                                       (Close)
    TIME-WAIT   <-- <SEQ=300><ACK=101><CTL=FIN,ACK>  <-- LAST-ACK

5.  TIME-WAIT   --> <SEQ=101><ACK=301><CTL=ACK>      --> CLOSED

6.  (2 MSL)
    CLOSED
```

Normal Close Sequence

Connection Termination

Half-Close, Four-way handshaking

RFC 793



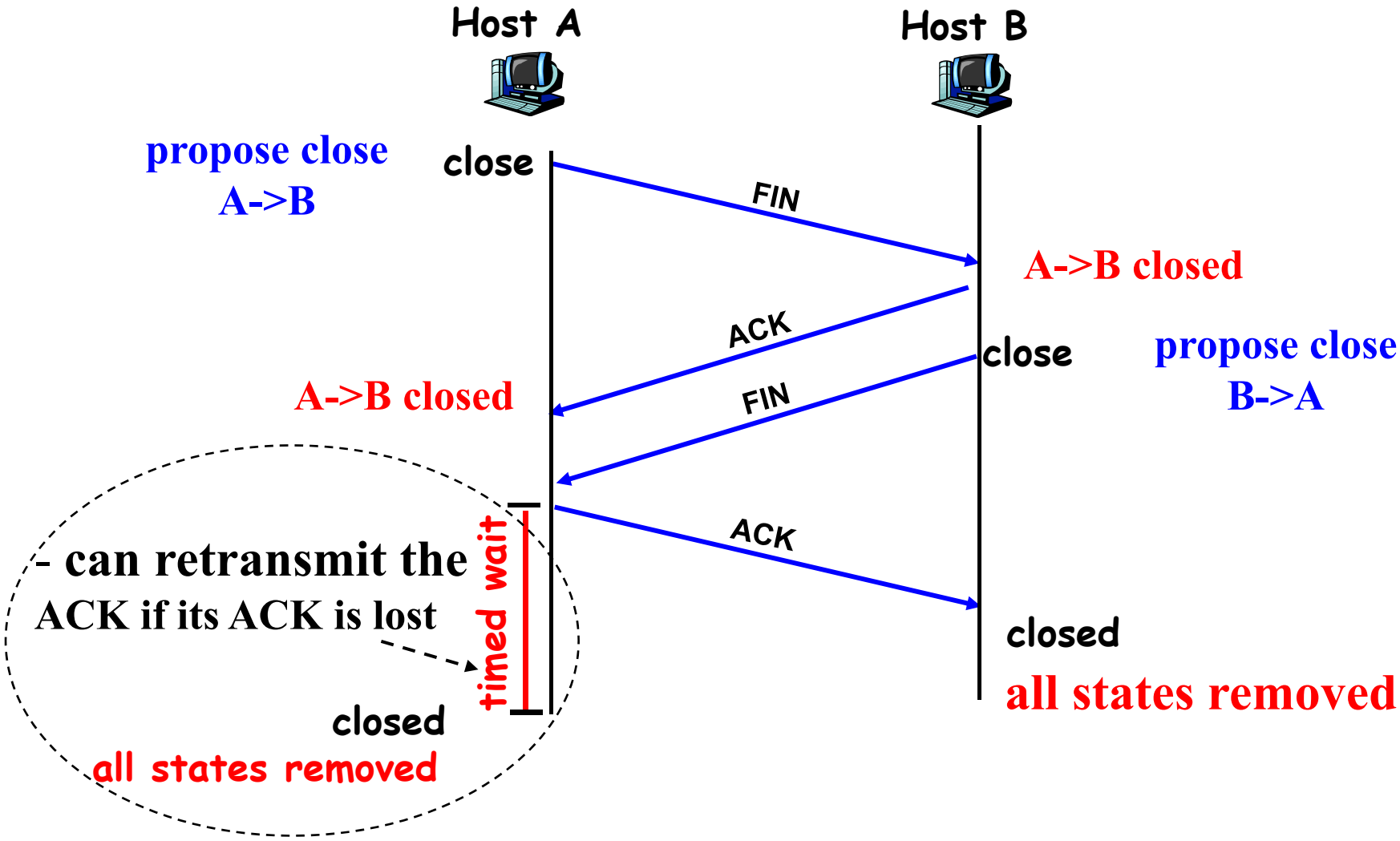
Connection Termination

Note

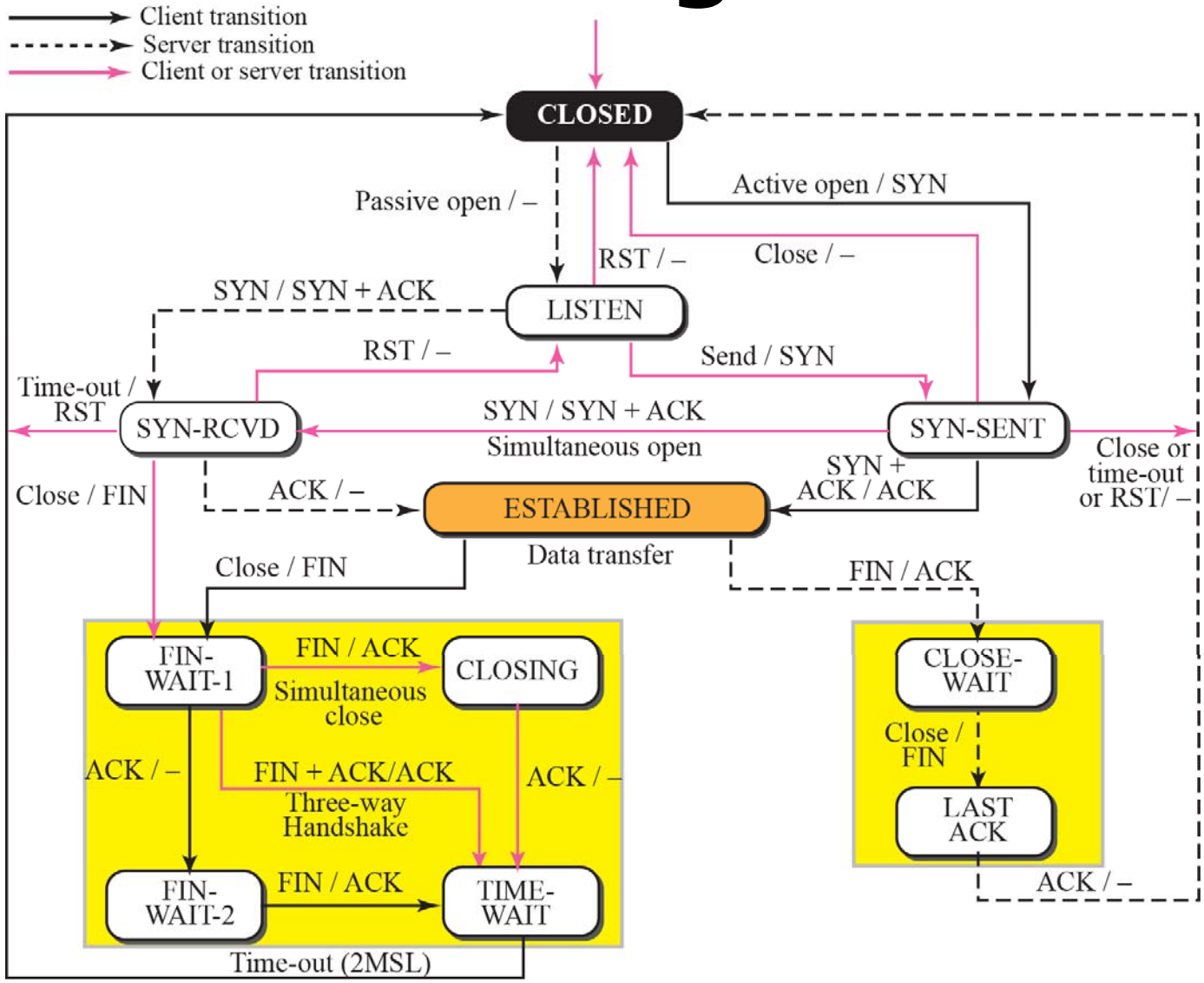
The FIN segment consumes one sequence number if it does not carry data.

The FIN + ACK segment consumes one sequence number if it does not carry data.

Four Way Teardown



Connection Management



State transition diagram

Connection Management

Note

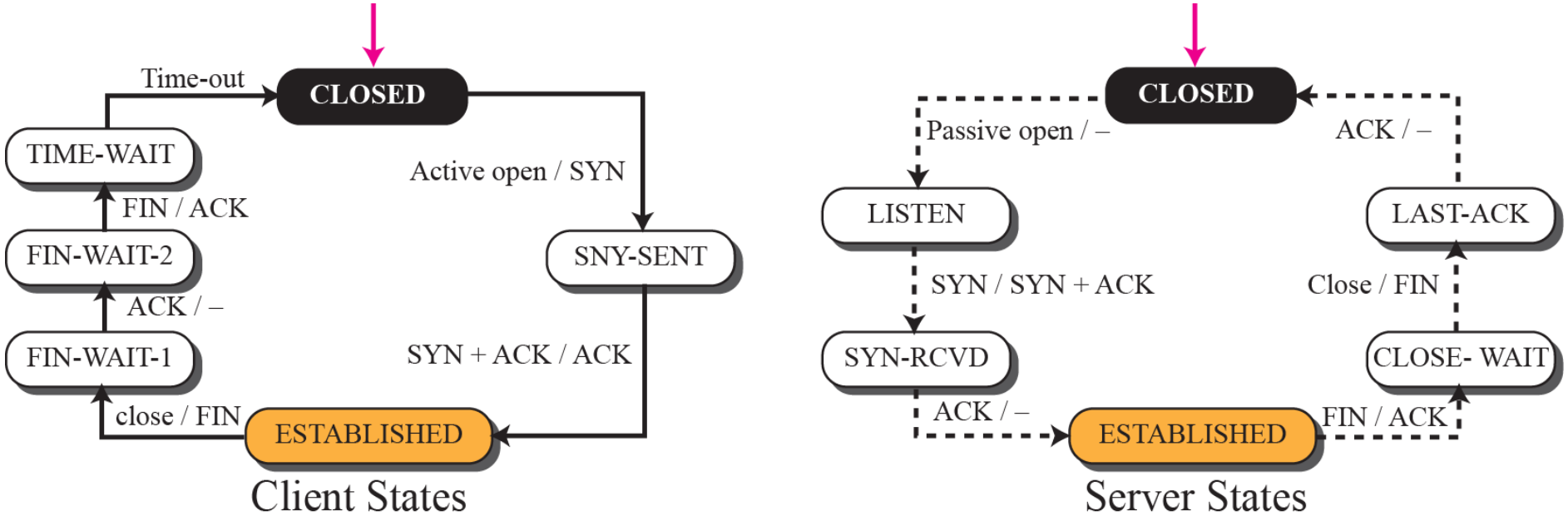
The state marked as ESTBLISHED in the FSM is in fact two different sets of states that the client and server undergo to transfer data.

Connection Management

States for TCP

State	*Description*
CLOSED	No connection exists
LISTEN	Passive open received; waiting for SYN
SYN-SENT	SYN sent; waiting for ACK
SYN-RCVD	SYN+ACK sent; waiting for ACK
ESTABLISHED	Connection established; data transfer in progress
FIN-WAIT-1	First FIN sent; waiting for ACK
FIN-WAIT-2	ACK to first FIN received; waiting for second FIN
CLOSE-WAIT	First FIN received, ACK sent; waiting for application to close
TIME-WAIT	Second FIN received, ACK sent; waiting for 2MSL time-out
LAST-ACK	Second FIN sent; waiting for ACK
CLOSING	Both sides decided to close simultaneously

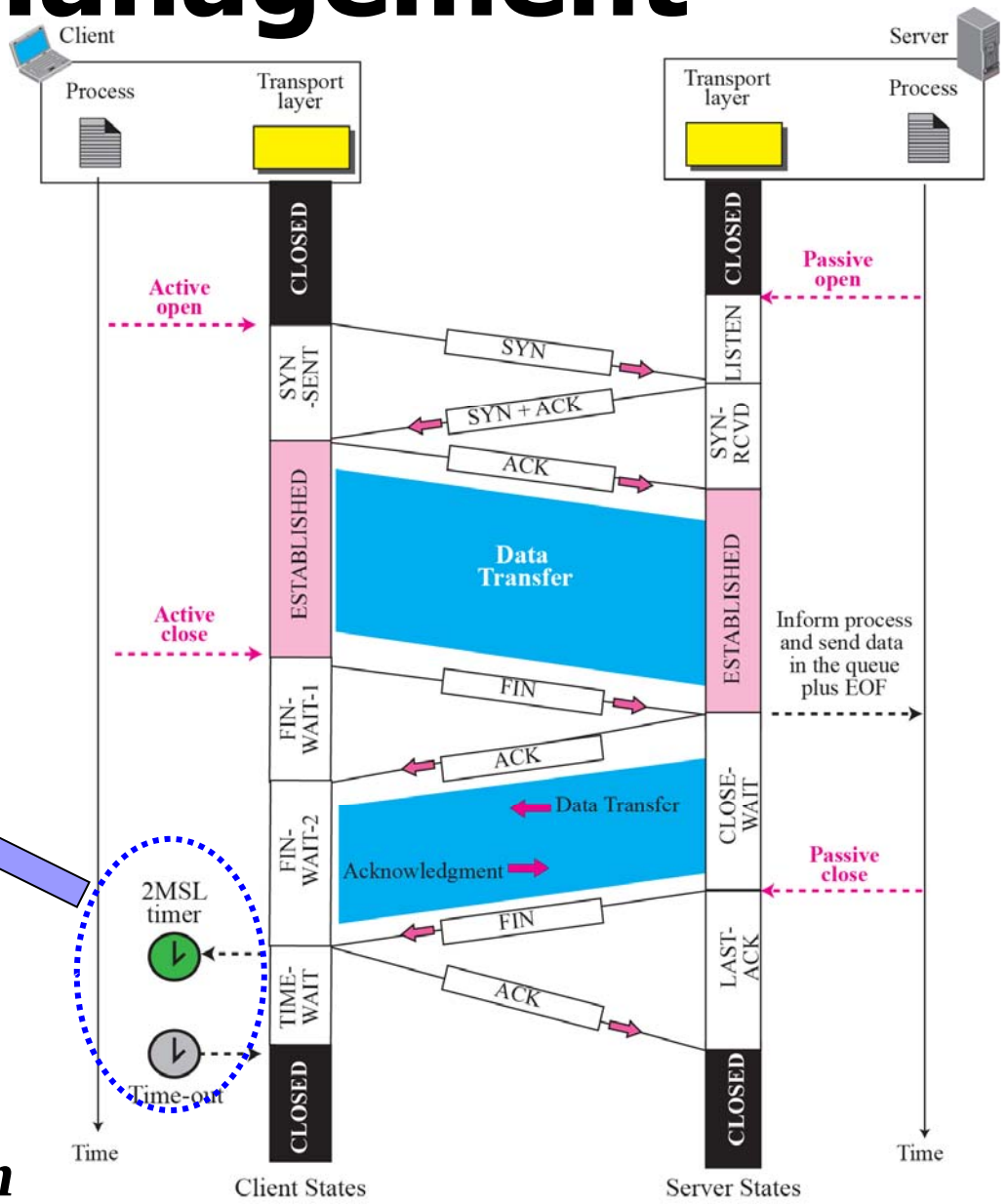
Connection Management



Transition diagram for connection and half-close termination

Connection Management

1. Enough time for an ACK to be lost and a new FIN to arrive. If during the TIME-WAIT state, a new FIN arrives, the client sends a new ACK and restarts the 2MSL timer
2. To prevent a duplicate segment from one connection appearing in the next one, TCP requires that incarnation cannot take place unless 2MSL amount of time has elapsed.



Time-line diagram

2MSL Wait State

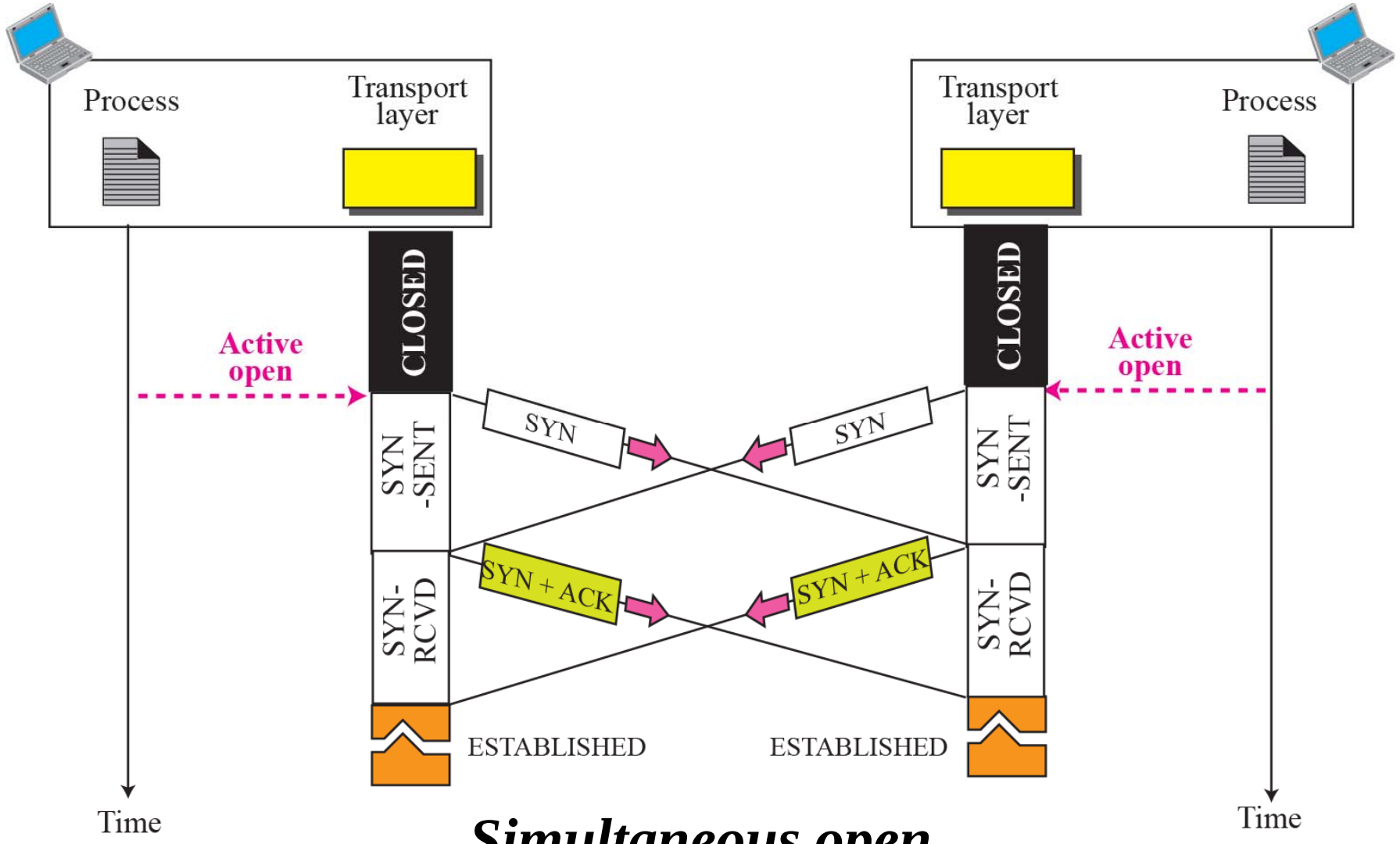
2MSL Wait State = TIME_WAIT

- When TCP does an active close, and sends the final ACK, the connection must stay in in the TIME_WAIT state for twice the maximum segment lifetime.

  2MSL= 2 * Maximum Segment Lifetime

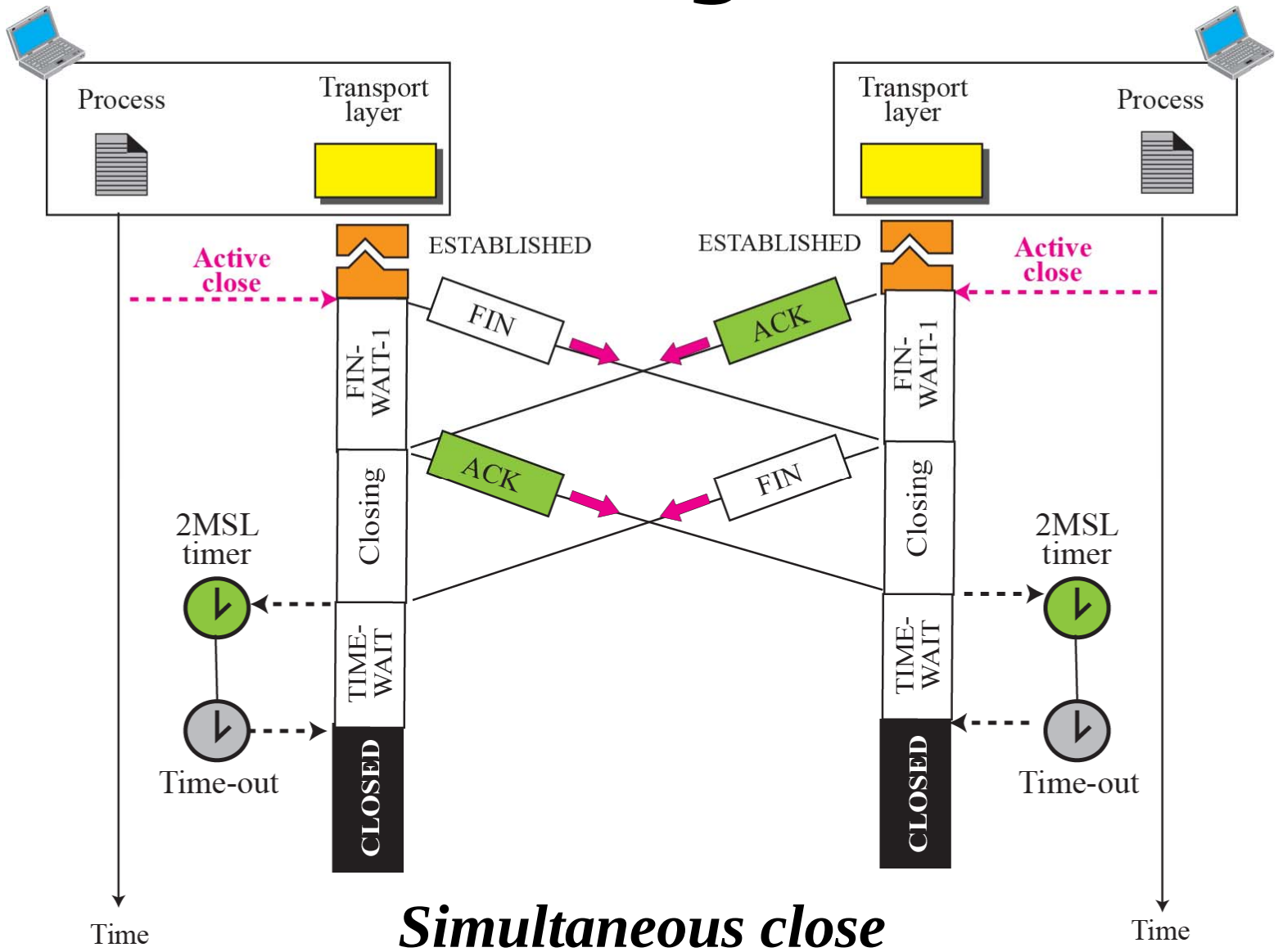
- Why?
  TCP is given a chance to resent the final ACK. (Server will timeout after sending the FIN segment and resend the FIN)
- The MSL is set to 2 minutes or 1 minute or 30 seconds.

Connection Management



Simultaneous open

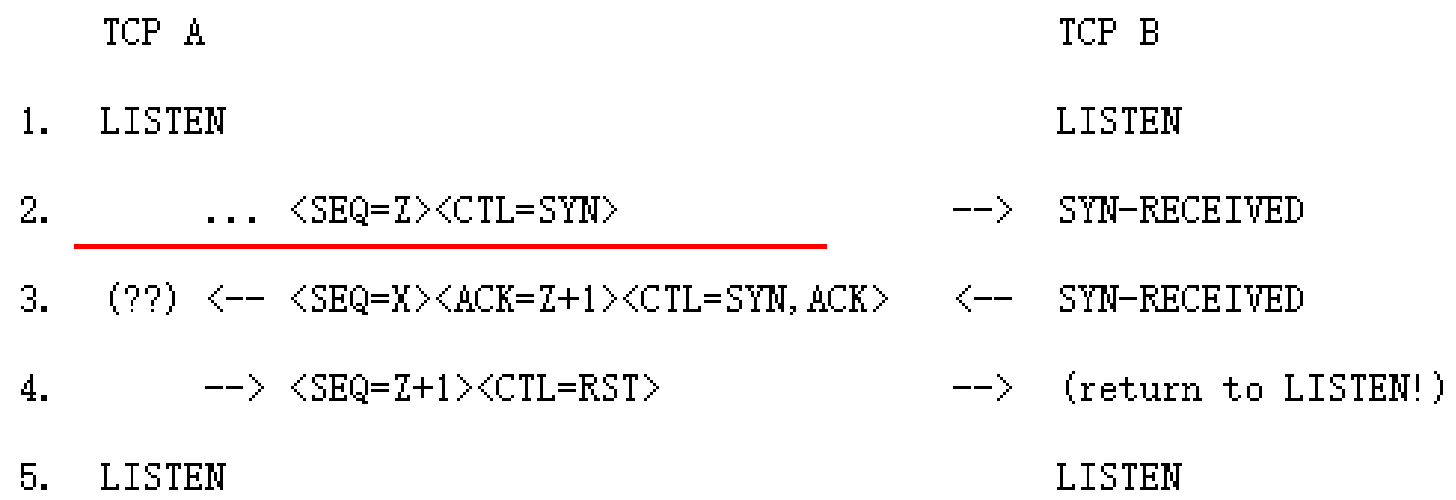
Connection Management



Simultaneous close

Connection Management

In figure 12, we find the two TCPs A and B with passive connections waiting for SYN. An old duplicate arriving at TCP B (line 2) stirs B into action. A SYN-ACK is returned (line 3) and causes TCP A to generate a RST (the ACK in line 3 is not acceptable). TCP B accepts the reset and returns to its passive LISTEN state.

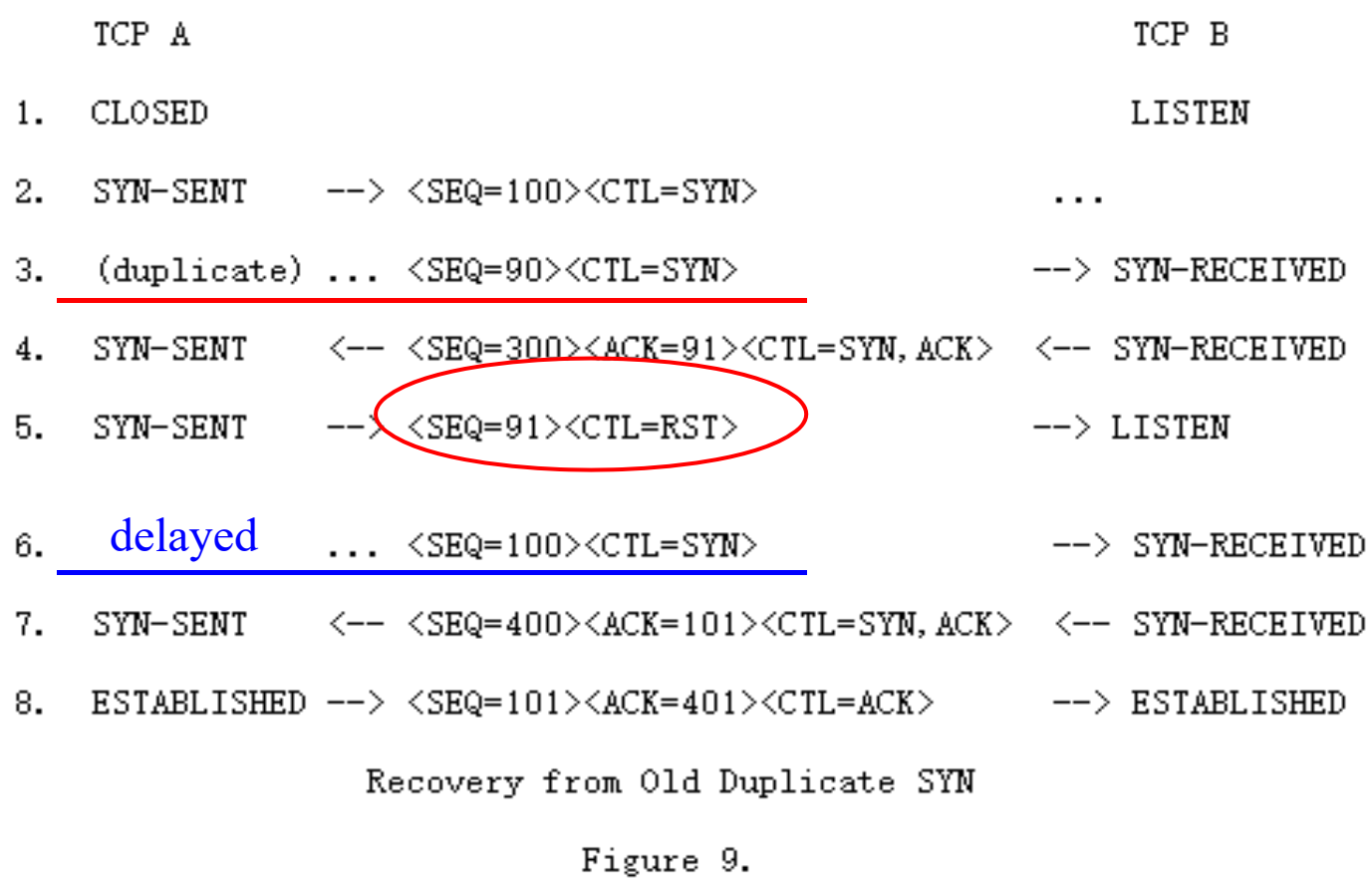


Old Duplicate SYN Initiates a Reset on two Passive Sockets

Figure 12.

Connection Management

As a simple example of recovery from old duplicates, consider figure 9. At line 3, an old duplicate SYN arrives at TCP B. TCP B cannot tell that this is an old duplicate, so it responds normally (line 4). TCP A detects that the ACK field is incorrect and returns a RST (reset) with its SEQ field selected to make the segment believable. TCP B, on receiving the RST, returns to the LISTEN state. When the original SYN (pun intended) finally arrives at line 6, the synchronization proceeds normally. If the SYN at line 6 had arrived before the RST, a more complex exchange might have occurred with RST's sent in both directions.



Recovery from Old Duplicate SYN

Figure 9.

Connection Management

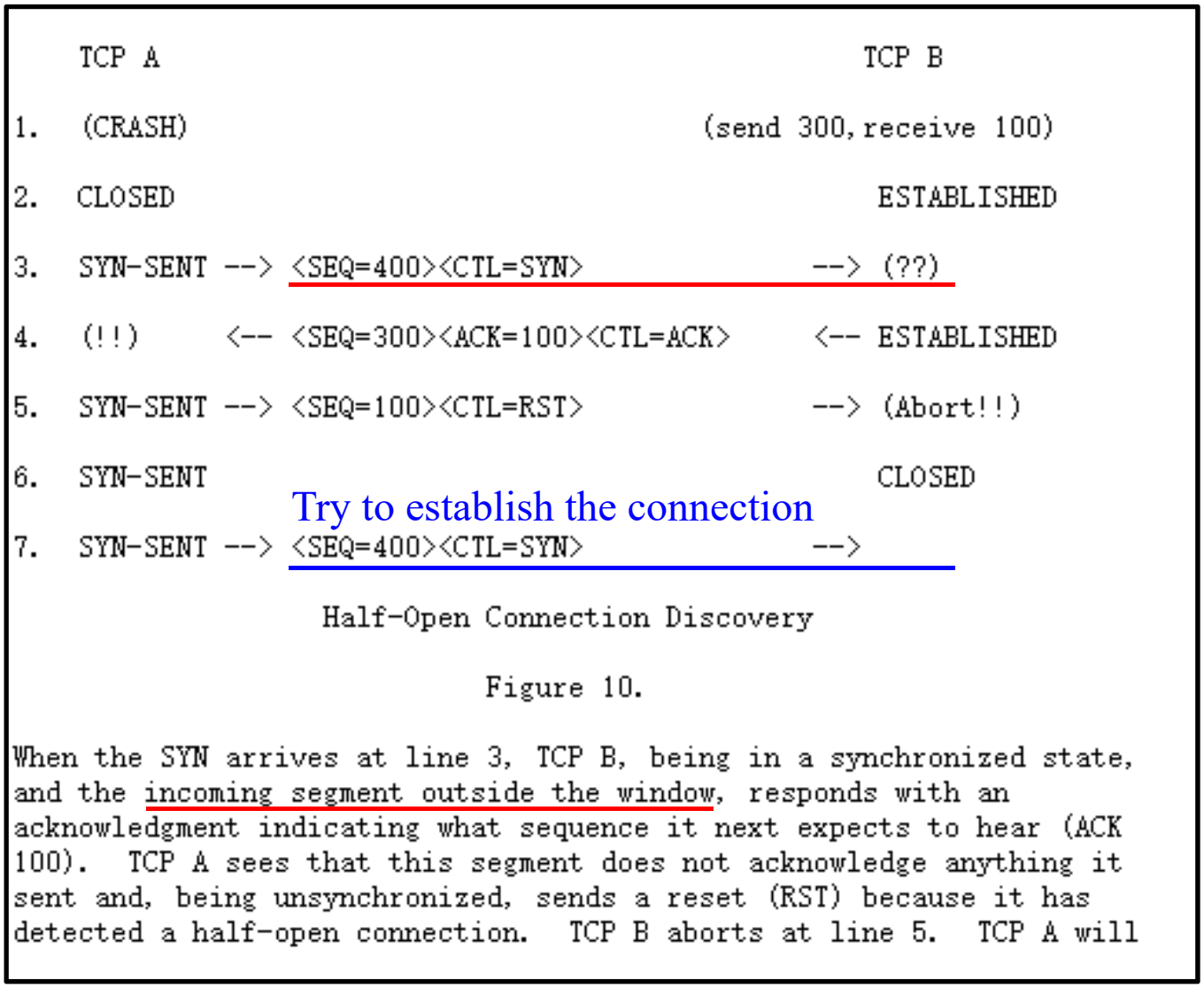


Figure 10.

When the SYN arrives at line 3, TCP B, being in a synchronized state, and the incoming segment outside the window, responds with an acknowledgment indicating what sequence it next expects to hear (ACK 100). TCP A sees that this segment does not acknowledge anything it sent and, being unsynchronized, sends a reset (RST) because it has detected a half-open connection. TCP B aborts at line 5. TCP A will

Connection Management

An interesting alternative case occurs when TCP A crashes and TCP B tries to send data on what it thinks is a synchronized connection. This is illustrated in figure 11. In this case, the data arriving at TCP A from TCP B (line 2) is unacceptable because no such connection exists, so TCP A sends a RST. The RST is acceptable so TCP B processes it and aborts the connection.

```
      TCP A                                                TCP B

1.   (CRASH)                                   (send 300, receive 100)

2.   (??)     <-- <SEQ=300><ACK=100><DATA=10><CTL=ACK> <-- ESTABLISHED

3.            --> <SEQ=100><CTL=RST>                    --> (ABORT!!)
```

Active Side Causes Half-Open Connection Discovery

Figure 11.

Resetting Connections

- Resetting connections is done by setting the RST flag
- When is the RST flag set?
  - Connection request arrives and no server process is waiting on the destination port
  - Abort (Terminate) a connection causes the receiver to throw away buffered data. Receiver does not acknowledge the RST segment

What is Flow/Congestion/Error Control ?

- **Error Control:**
  - Algorithms to recover or conceal the effects from packet losses
- **Flow Control:**
  - Algorithms to prevent that the sender overruns the receiver with information
- **Congestion Control:**
  - Algorithms to prevent that the sender overloads the network (adapt to network conditions)

- The goal of each of the control mechanisms are different.
- In TCP, the implementation of these algorithms is combined – Sliding windows

Unreliable IP Packet Delivery Service

- **Benefits:**
  - Fast
  - Cost-effective
- **Drawbacks:**
  - Packet loss, corruption, delay, duplication, out-of-order delivery
  - Sender might transmit faster than receiver can receive

The Need for a Reliable Stream Delivery Service

- **Application-level programmers don't want:**
  - To have to provide reliability in each application
  - To be constrained by fixed-size packets

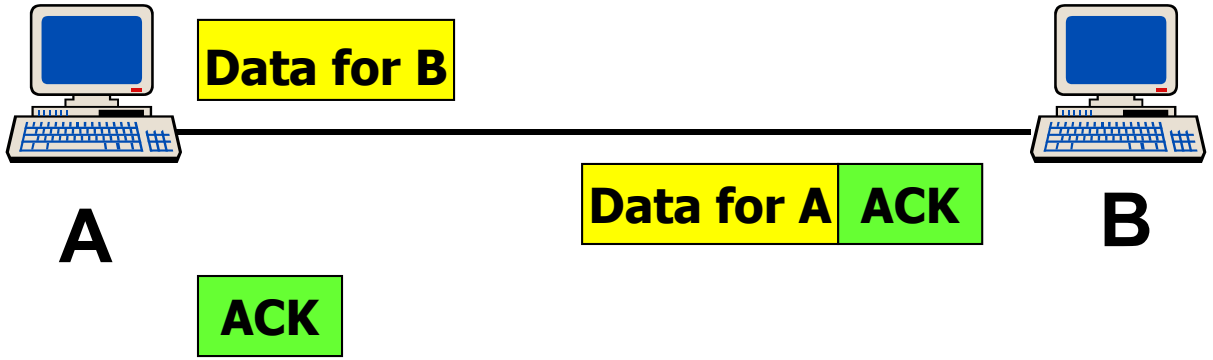
- **Application-level programmers do want:**
  - Reliability
  - Unstructured, stream-oriented service
  - Full duplex virtual circuit connection
  - Buffered transfer

Reliability in TCP

- **Checksum** used to detect bit level errors
- **Sequence numbers** used to detect sequencing errors
  - Duplicates are ignored
  - Reordered packets are reordered (or dropped)
  - Lost packets are retransmitted
- **Acknowledgement**
- **Timeouts** used to detect lost packets
  - Requires RTO calculation
  - Requires sender to maintain data until it is ACKed
- **Retransmission** used to correct errors

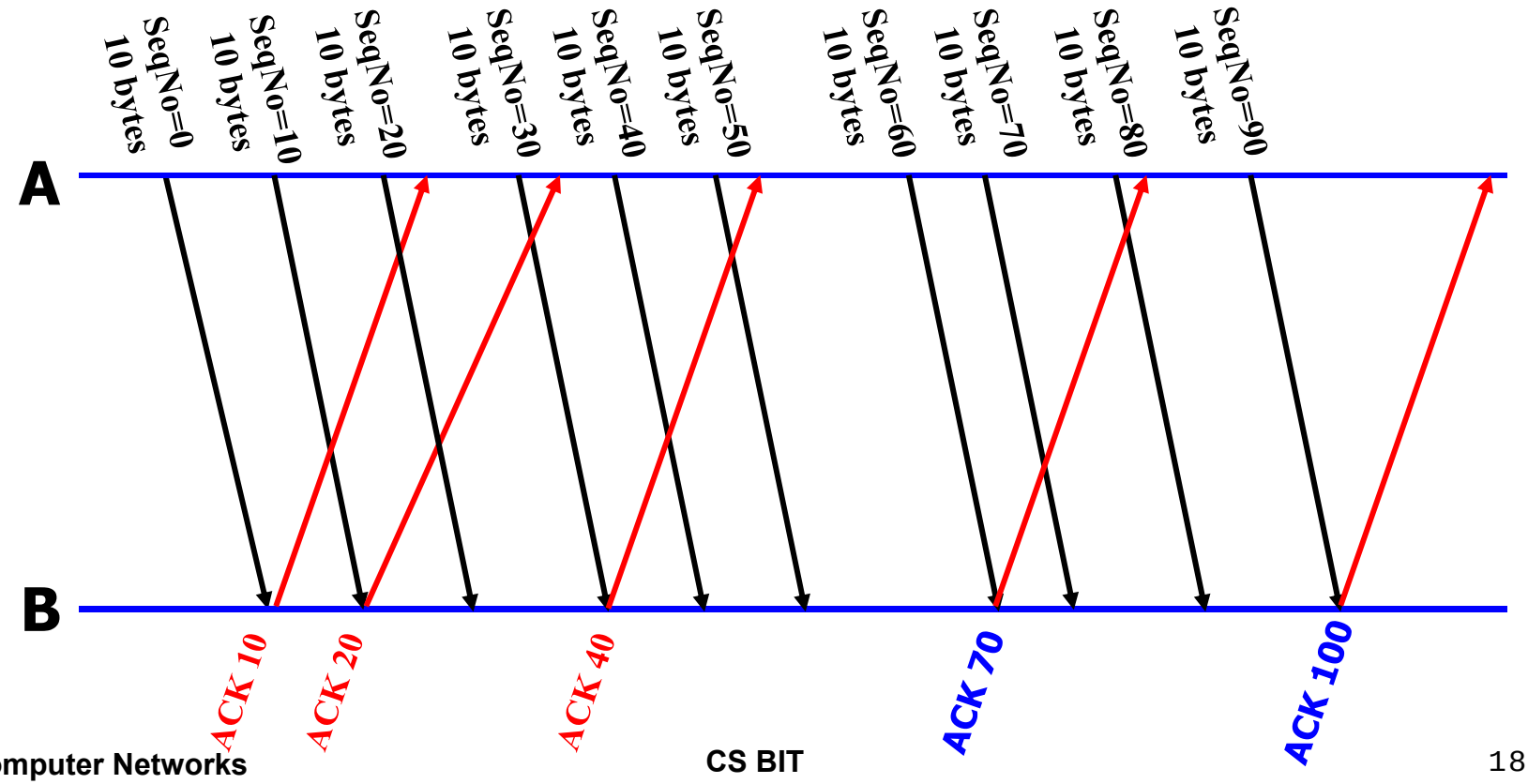
Acknowledgements in TCP

- TCP receivers use acknowledgments (ACKs) to confirm the receipt of data to the sender
- Acknowledgment can be added ("piggybacked") to a data segment that carries data in the opposite direction
- ACK information is included in the the TCP header
- Acknowledgements are used for flow control, error control, and congestion control

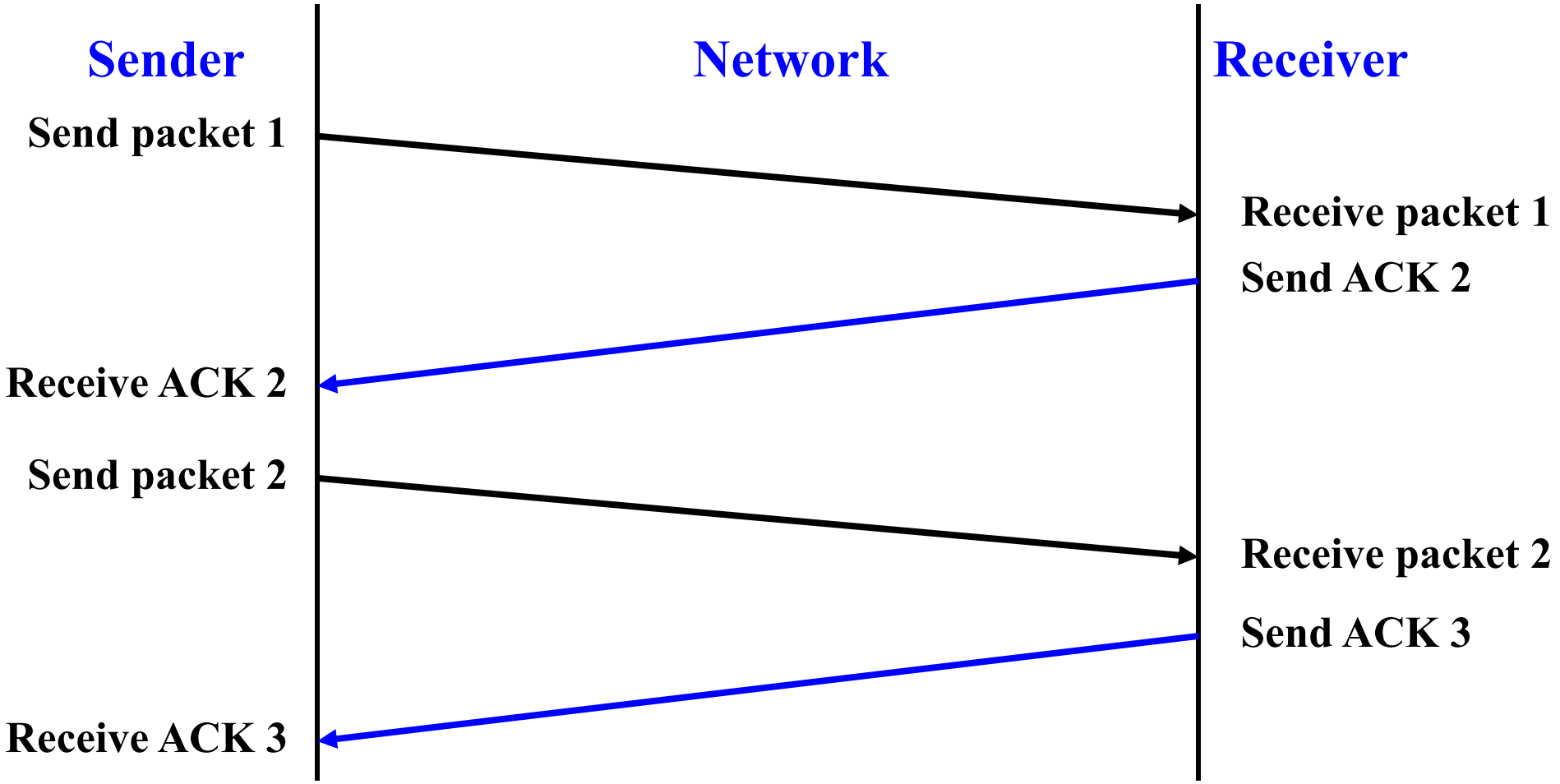


Cumulative Acknowledgements

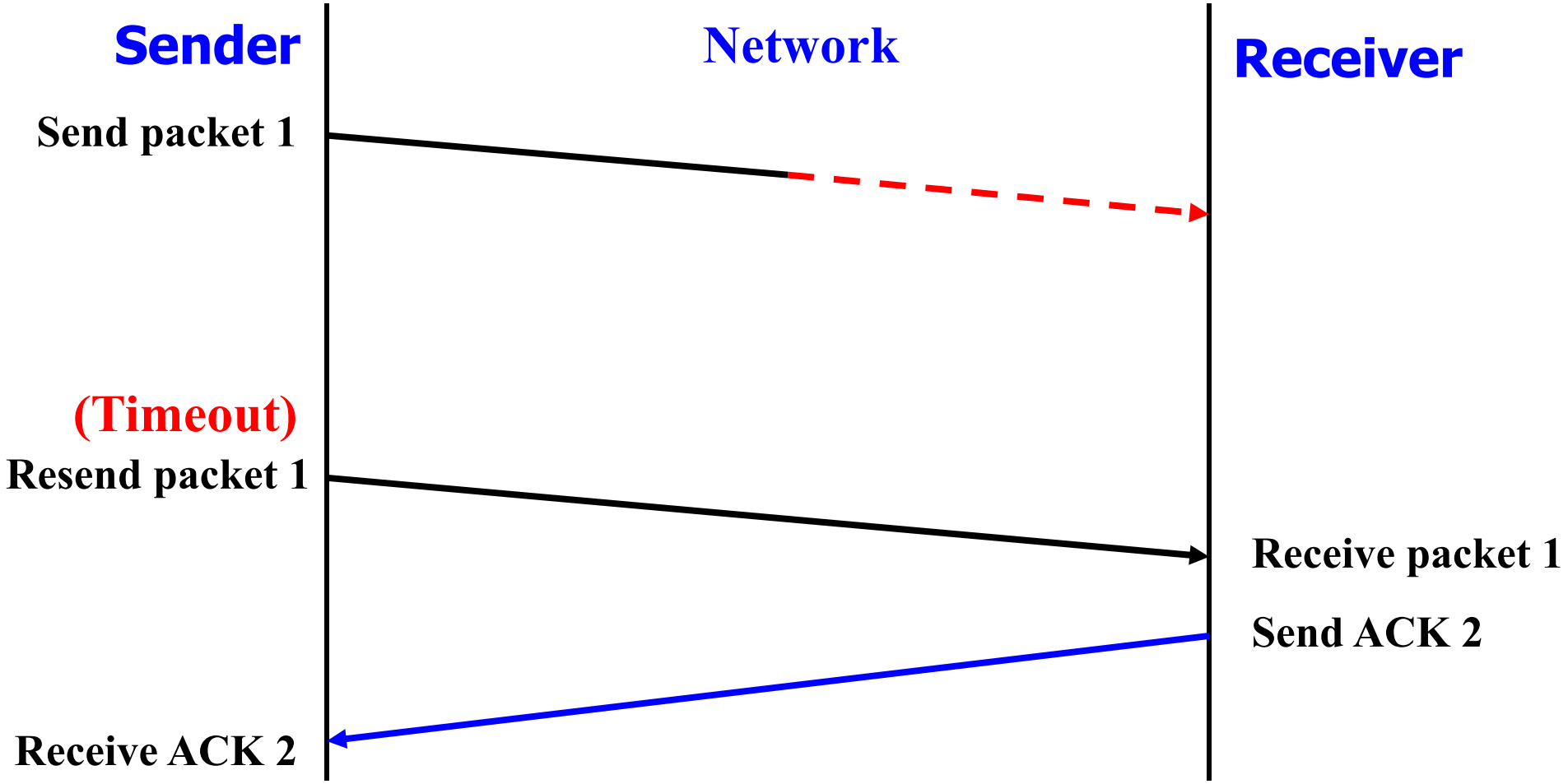
- **TCP has cumulative acknowledgements:**
  An acknowledgment confirms the receipt of all unacknowledged data with a smaller sequence number



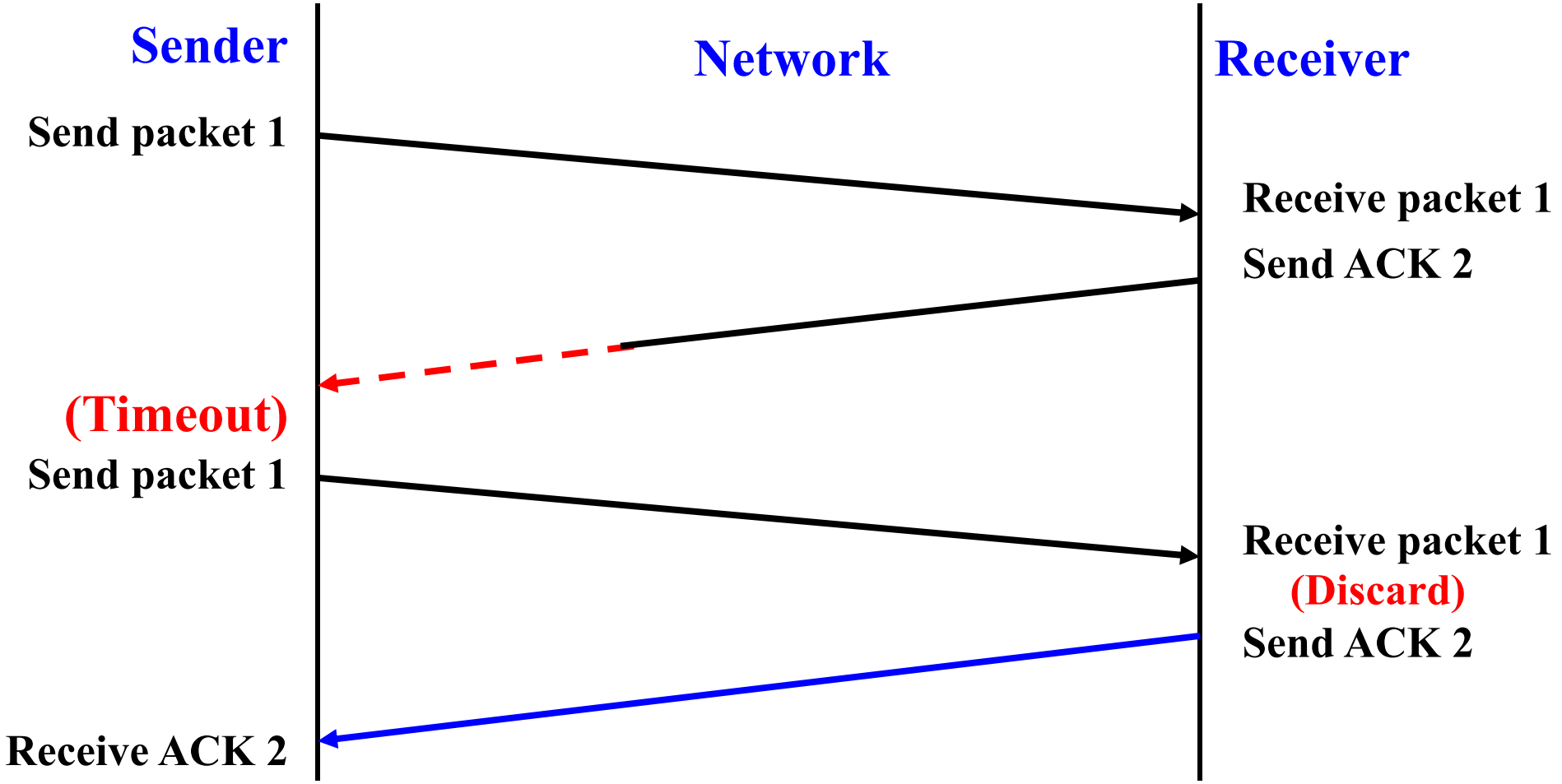
Providing Reliability with Acknowledgments and Retransmissions



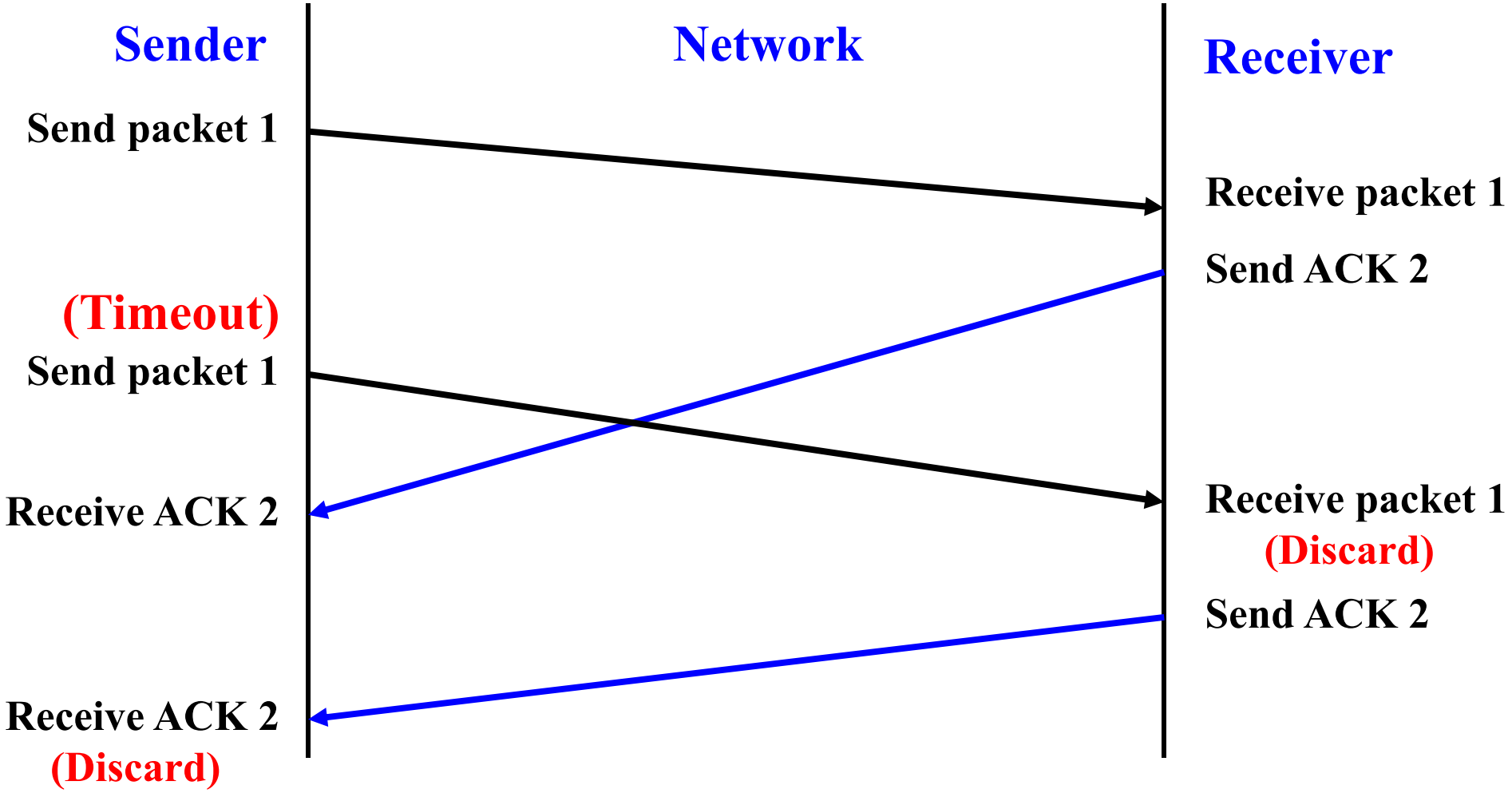
Packet Loss


Sender
Network
Receiver
Send packet 1
(Timeout)
Resend packet 1
Receive packet 1
Send ACK 2
Receive ACK 2

ACK Loss


Sender
Network
Receiver
Send packet 1
Receive packet 1
Send ACK 2
(Timeout)
Send packet 1
Receive packet 1
(Discard)
Send ACK 2
Receive ACK 2

ACK Delayed


Sender
Network
Receiver
Send packet 1
Receive packet 1
Send ACK 2
(Timeout)
Send packet 1
Receive ACK 2
Receive packet 1
(Discard)
Send ACK 2
Receive ACK 2
(Discard)

Acknowledgements

Note

ACK segments do not consume sequence numbers and are not acknowledged.

No retransmission timer is set for an ACK segment.

Acknowledgements

Note

Data may arrive out of order and be temporarily stored by the receiving TCP, but TCP guarantees that no out-of-order data are delivered to the process.

TCP Round Trip Time and Timeout

Q: How to set TCP timeout value?

- **longer than RTT**
  - but RTT varies
- **too short:** premature timeout
  - unnecessary retransmissions
- **too long:** slow reaction to segment loss

TCP Round Trip Time and Timeout

Q: How to estimate RTT?

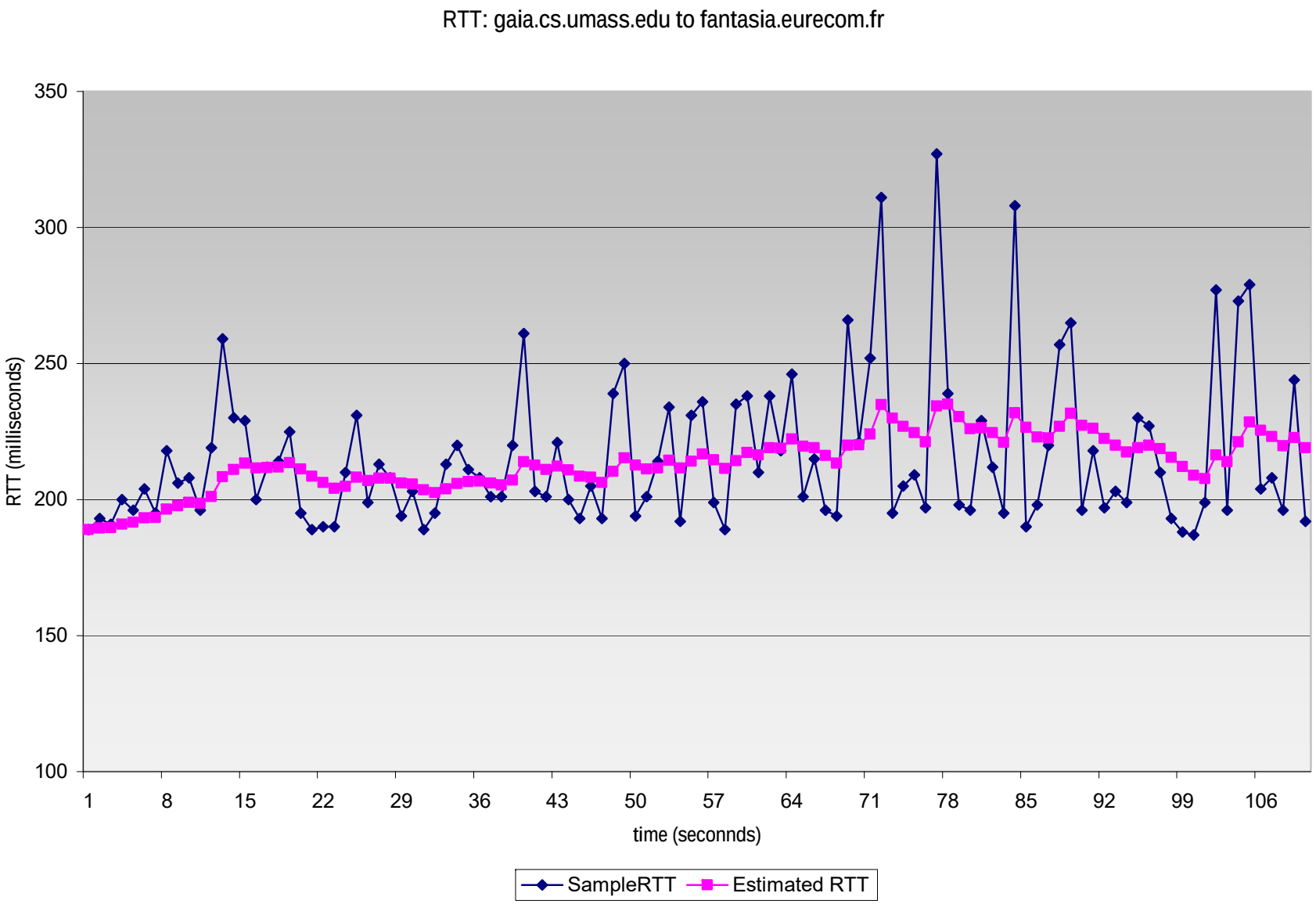
- **SampleRTT:** measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout

$$\text{EstimatedRTT} = (1-\alpha)*\text{EstimatedRTT} + \alpha*\text{SampleRTT}$$

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout:

EstimatedRTT plus "safety margin"

- First estimate of how much SampleRTT deviates from EstimatedRTT:

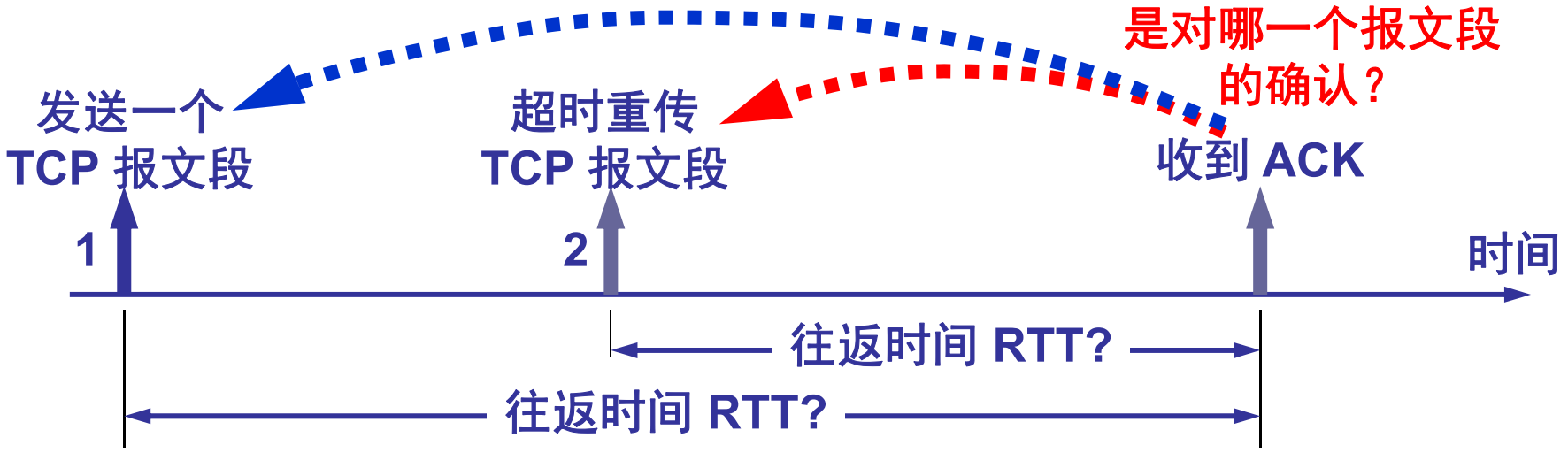
```
DevRTT = (1-β)*DevRTT +
                β*|SampleRTT-EstimatedRTT|
(typically, β = 0.25)
```

- Then set timeout interval:

```
Timeout = EstimatedRTT + 4*DevRTT
```

RTT测量相当复杂

- TCP 报文段1没有收到确认。重传（即报文段1）后，收到了确认报文段 ACK。
- 如何判定此确认报文段是对原来的报文段 1 的确认，还是对重传的报文段 1 的确认？

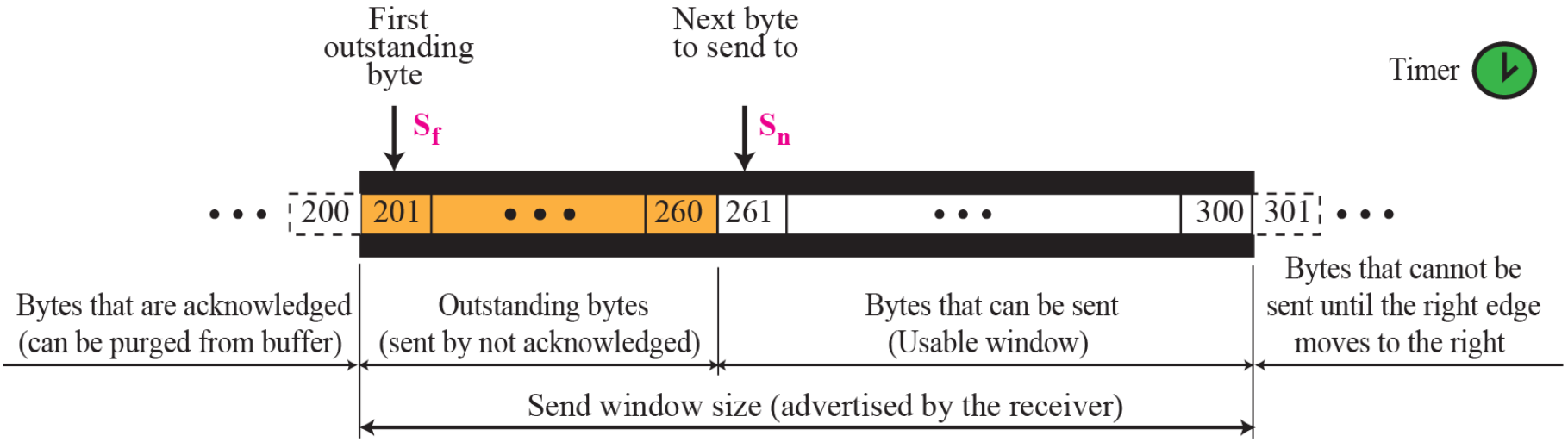


TCP Sliding Window

- TCP uses 2 windows (send window and receive window) for each direction of data transfer
  - which means 4 windows for a bidirectional communication.

- Used for error control, flow control and congestion control

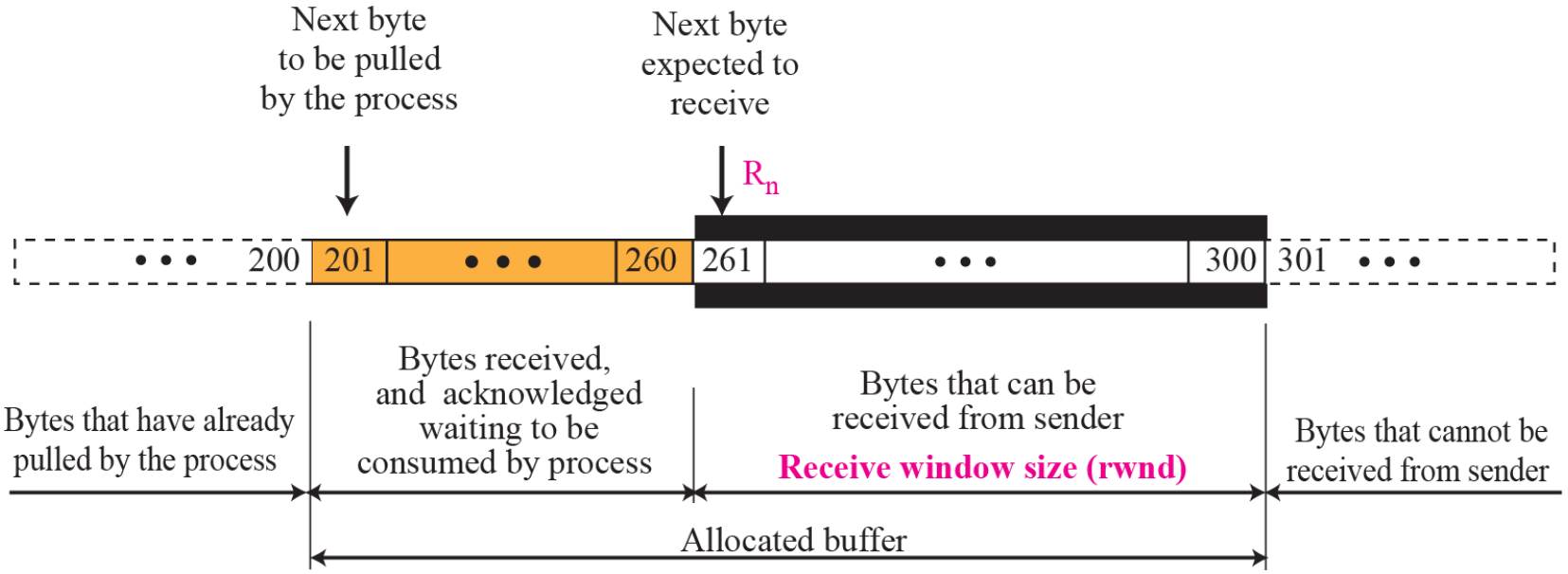
Send window



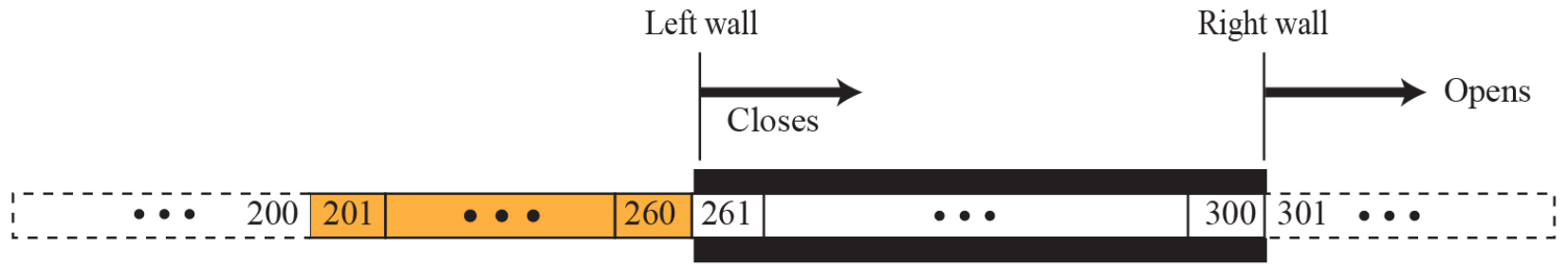
a. Send window

b. Opening, closing, and shrinking send window

Receive window



a. Receive window and allocated buffer



b. Opening and closing of receive window

TCP Flow Control

- Tell peer exactly how many bytes it is willing to accept (advertised window → sender can not overflow receiver buffer)
  - Sender window includes bytes sent but not acknowledged
  - Receiver window (number of empty locations in receiver buffer)
  - Receiver advertises window size in ACKs

TCP Flow Control

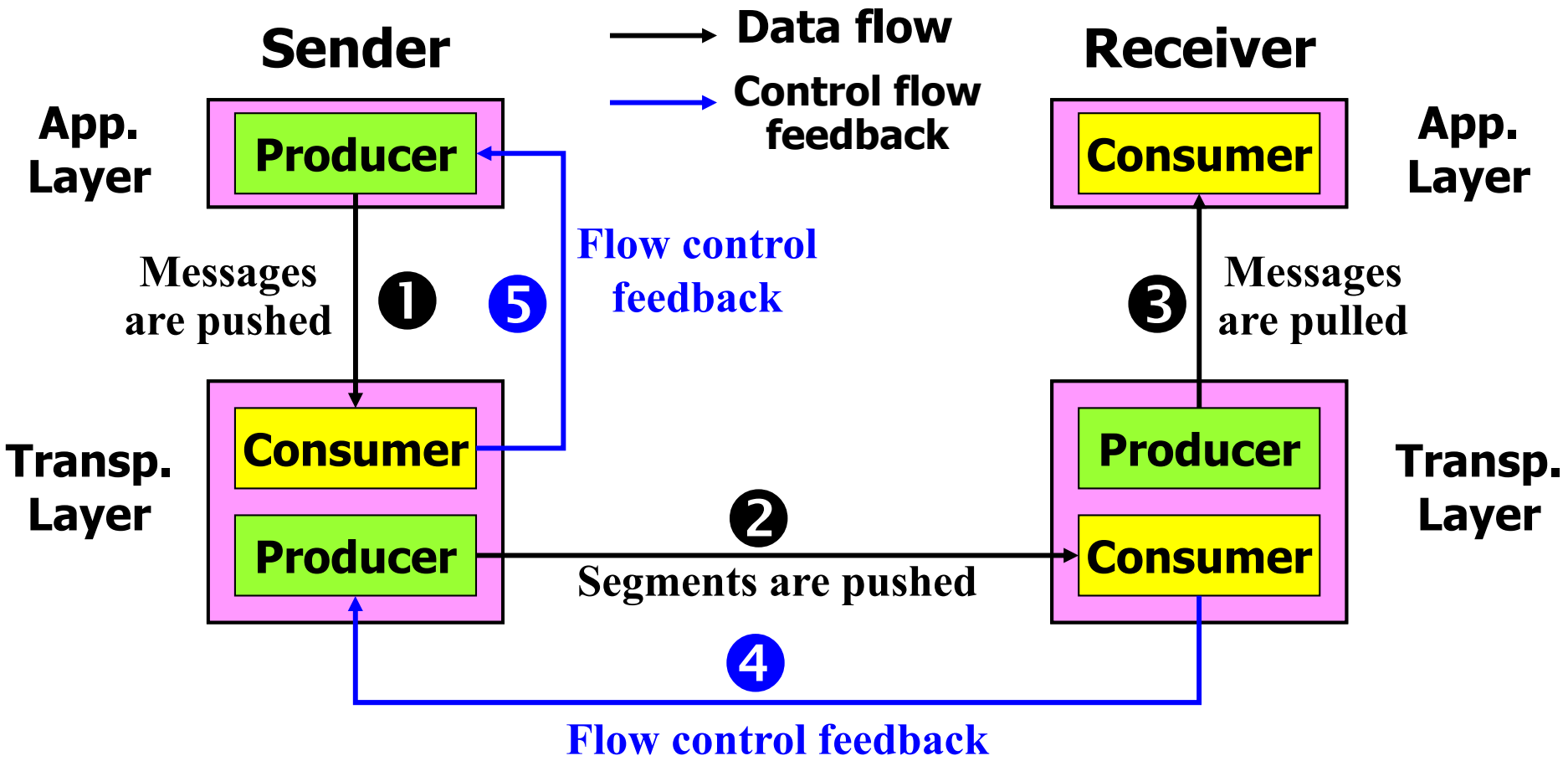
- **Sender window <= receiver window (flow control)**
  - **Sliding** sender window (without a change in receiver's advertised window)
  - **Expanding** sender window (receiving process consumes data faster than it receives → receiver window size increases)
  - **Shrinking** sender window (receiving process consumes data more slowly than it receives → receiver window size reduces)
  - **Closing** sender window (receiver advertises a window of zero)

TCP Flow Control

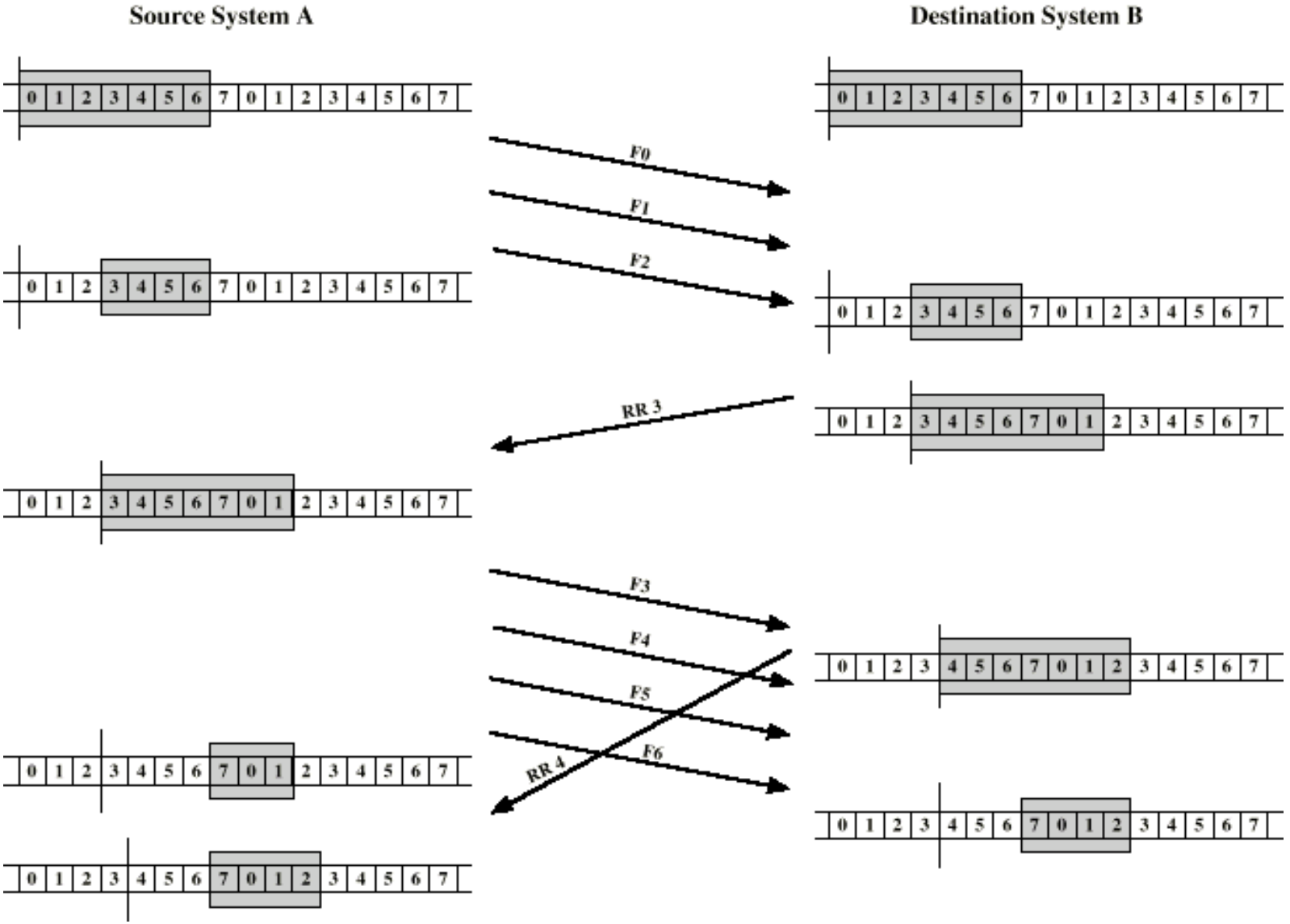
Note

A sliding window is used to make transmission more efficient as well as to control the flow of data so that the destination does not become overwhelmed with data.
TCP sliding windows are byte-oriented.

TCP Flow Control



Example Sliding Window

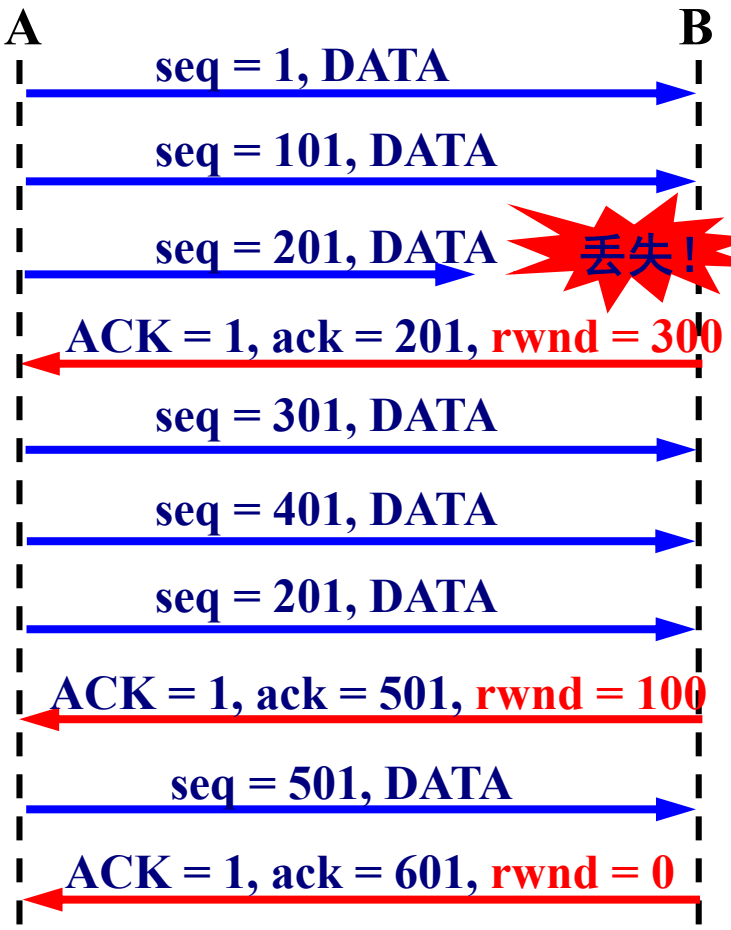


TCP Flow Control

- Each window can vary in size over time
  - Each ACK contains a *window advisement* Specifing how many additional octets of data the receiver is willing to accept
  - Sender increases or decreases sending window sized based on the receiver's advise
- Provides *end-to-end flow control*

TCP Flow Control

A 向 B 发送数据。在连接建立时，
B 告诉 A：“我的接收窗口 rwnd = 400（字节）”。



A 发送了序号 1 至 100，还能发送 300 字节

A 发送了序号 101 至 200，还能发送 200 字节

允许 A 发送序号 201 至 500 共 300 字节

A 发送了序号 301 至 400，还能再发送 100 字节新数据

A 发送了序号 401 至 500，不能再发送新数据了

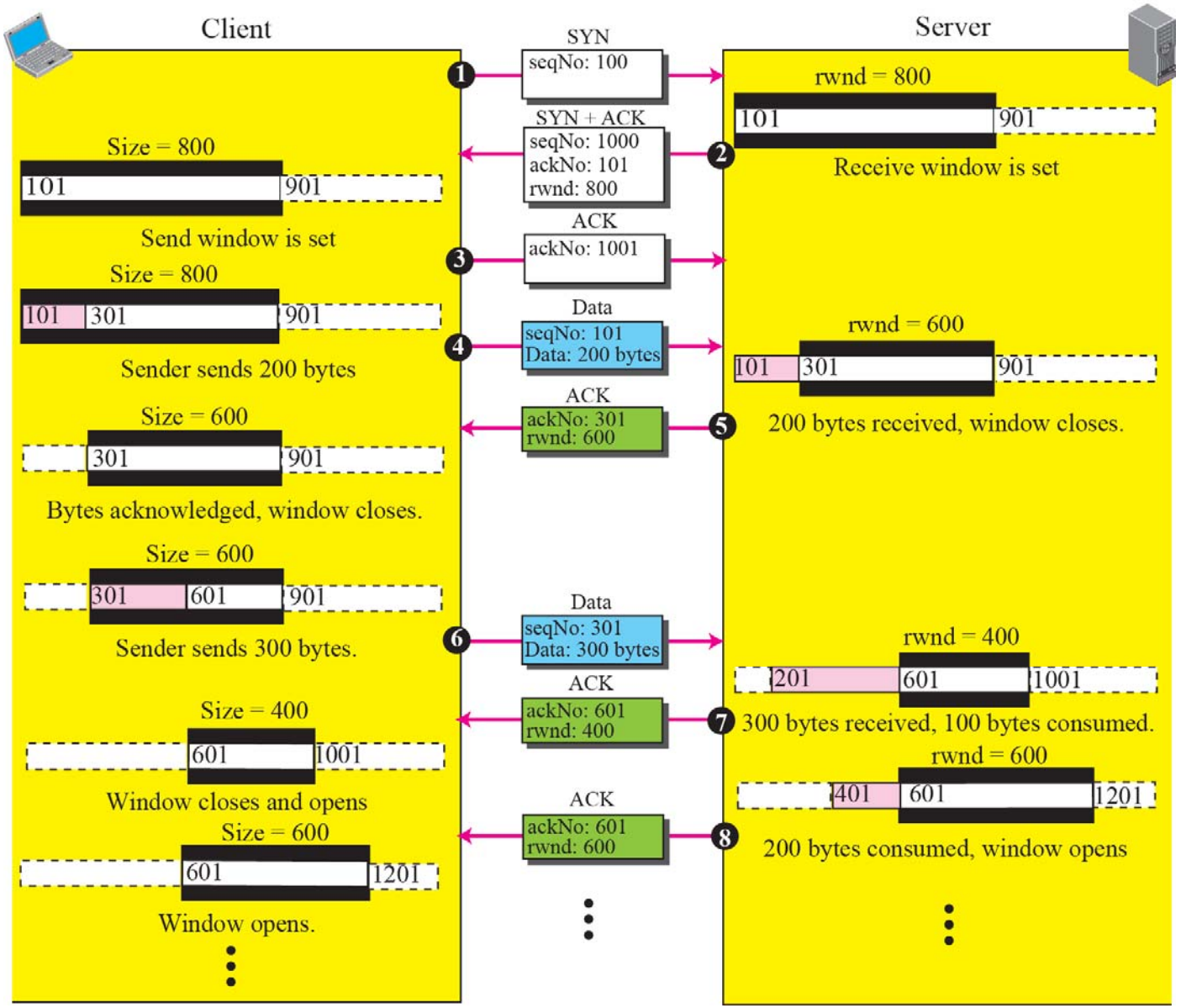
A 超时重传旧的数据，但不能发送新的数据

允许 A 发送序号 501 至 600 共 100 字节

A 发送了序号 501 至 600，不能再发送了

不允许 A 再发送（到序号 600 为止的数据都收到了）

TCP Flow Control



Silly Window Syndrome (1/2)

Sending data in very small segments

1. Syndrome created by the Sender

- Sending application program creates data slowly (e.g. 1 byte at a time)
- Wait and collect data to send in a larger block
- How long should the sending TCP wait?
- **Solution:** Nagle's algorithm

Sender accumulates data until "enough" data to send

Silly Window Syndrome (2/2)

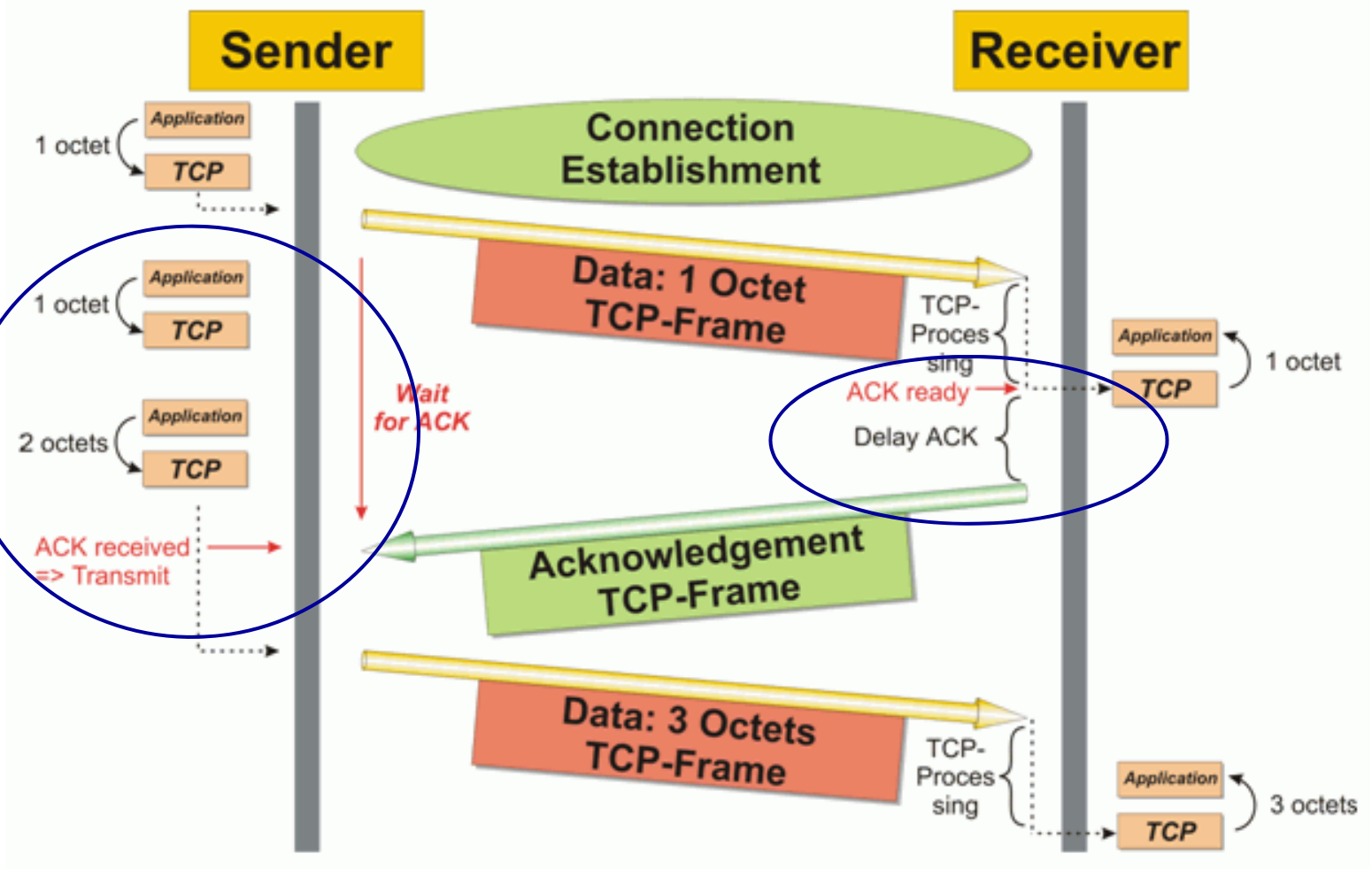
2. Syndrome created by the Receiver

- Receiving application program consumes data slowly (e.g. 1 byte at a time)
- The receiving TCP announces a window size of 1 byte. The sending TCP sends only 1 byte…
- **Solution 1:** Clark's solution
- **Solution 2:** Delayed Acknowledgement

Clark's solution:

receiver consumes data until "enough" space available to advertise

Can be used together



TCP Congestion control

- TCP assumes the cause of a lost segment is due to congestion in the network
- If the cause of the lost segment is congestion, retransmission of the segment does not remove the problem, it actually aggravates it
- The network needs to tell the sender to slow down (affects the sender window size in TCP)

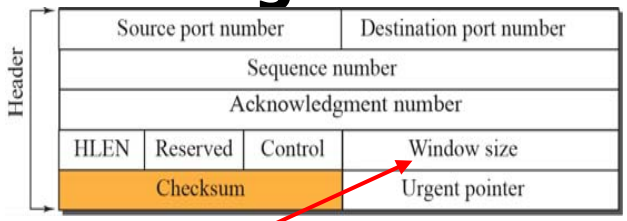
TCP Congestion control

- TCP has a mechanism for congestion control. The mechanism is implemented at the sender
- TCP uses a congestion window and a congestion policy that avoid congestion and detect and alleviate congestion after it has occurred.
- congestion policy: AIMD

Additive Increase Multiplicative Decrease

TCP Congestion control

- **Congestion window**
  - Indicates the **maximum amount** of data that can be sent out on a connection without being acknowledged.
  - An **estimate** of network capacity.



- **Send window size is set as follows:**

Send Window = Min (flow control window, congestion window)

- **flow control window** is advertised by the receiver
- **congestion window** is adjusted based on feedback from the network

TCP Congestion control

- **Congestion detection: (by Sender)**
  - **Implicit feedback**
  - **Loss (Retransmission timer times out)**

  OR
  - **3 duplicated ACK received**
- **Action**
  - **Decrease the congestion window to reduce the amount of data to be sent.**

TCP Congestion Controls

- Tahoe (Jacobson 1988)
  - Slow Start + Congestion Avoidance + Fast Retransmit
- Reno (Jacobson 1990)
  - Tahoe + Fast Recovery
- New Reno (Janey Hoe 1996)
  - Reno with modified fast recovery
- SACK (an option in TCP)
  - Reno + selective ACKs
- Vegas (Brakmo & Peterson 1994)
  - New Congestion Avoidance
- RED (Floyd & Jacobson 1993)
  - Probabilistic marking
- REM (Athuraliya & Low 2000)
  - Clear buffer, match rate

TCP Congestion control

- TCP congestion control is governed by two parameters:
  - Congestion Window (cwnd)
  - Slow-start threshhold Value (ssthresh)

    (Initial value is 2^{16}-1)
- Congestion control works in two phases:
  - slow start (cwnd < ssthresh)
  - congestion avoidance (cwnd ≥ ssthresh)

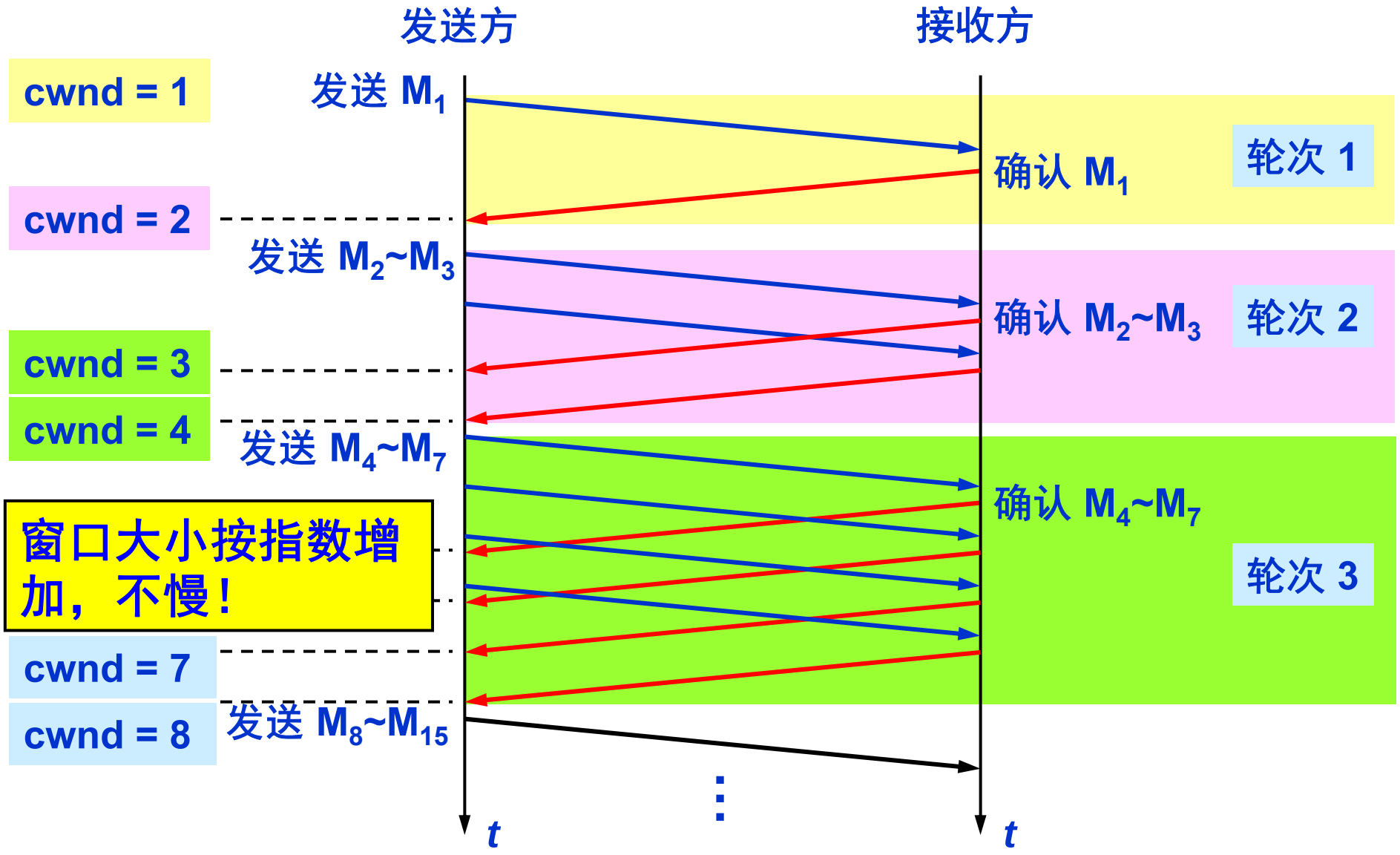
Slow Start

- **Objective:** Determine initial available capacity
- Idea
  - Begin with `CongestionWindow` = 1 segment size
  - For each ACKed segment, increases it by 1 segment size

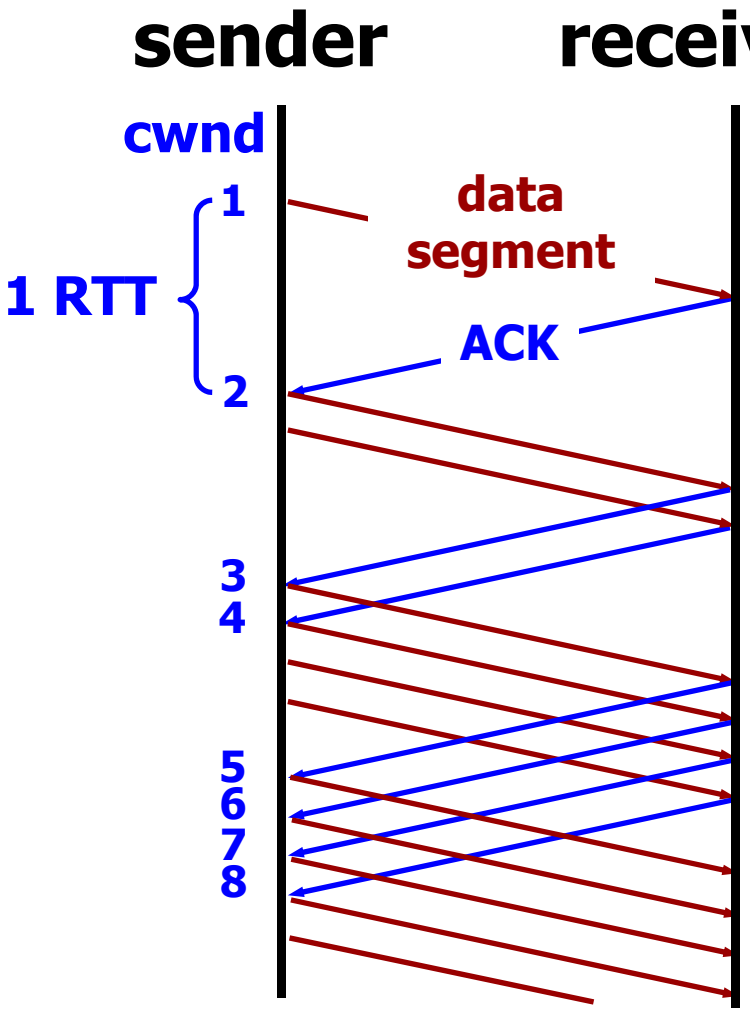
    `CongestionWindow++`
  - Continue increasing until (loss or cwnd ≥ ssthresh)
- Result : Exponential growth of cwnd

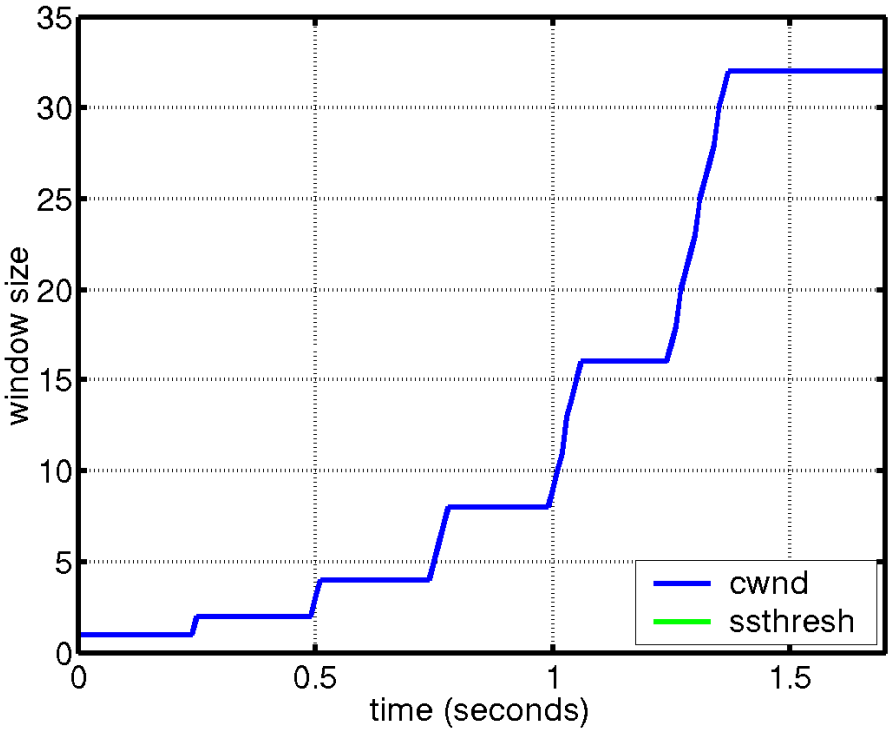
  each RTT: cwnd ← 2 x cwnd
- Used
  - When first starting connection
  - When connection times out

发送方每收到一个对新报文段的确认（重传的不算在内）就使 cwnd 加 1。



Slow Start





cwnd ← cwnd + 1 (for each ACK)

Congestion Avoidance

- When cwnd ≥ ssthresh, enter CA.
  - Slows down the increase of cwnd
- On each successful ACK:

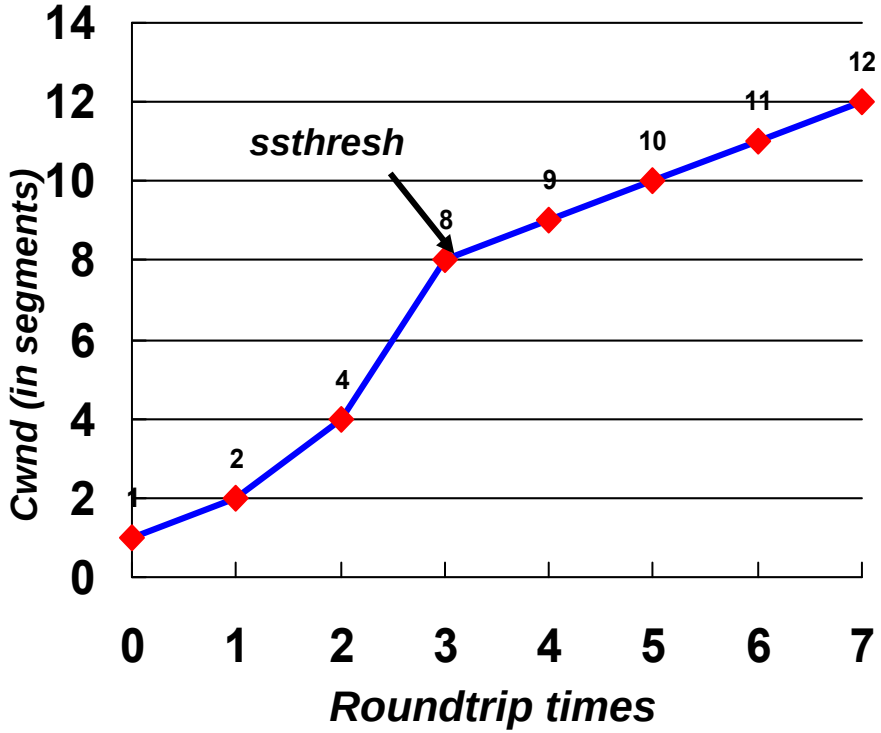
  cwnd ← cwnd + 1/cwnd
- Result : Linear growth of cwnd

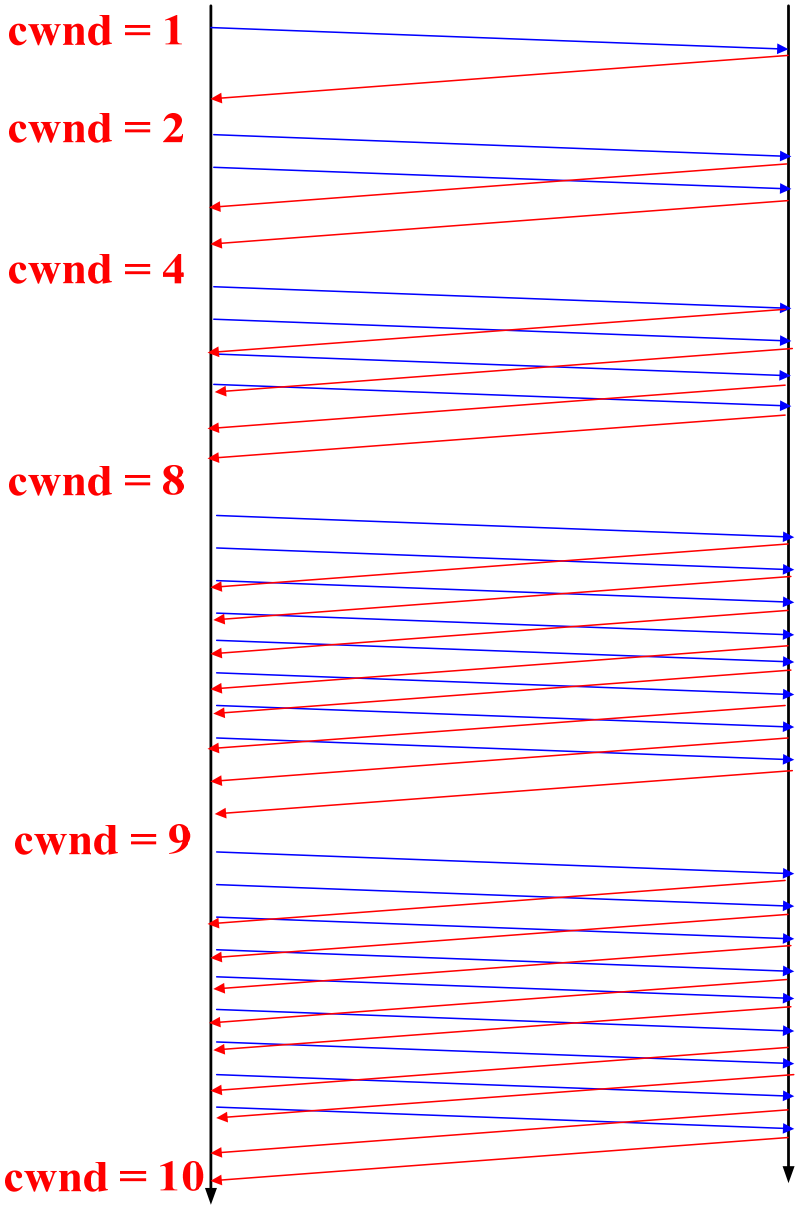
  each RTT: cwnd ← cwnd + 1

cwnd is increased by one only if all *cwnd* segments have been acknowledged.

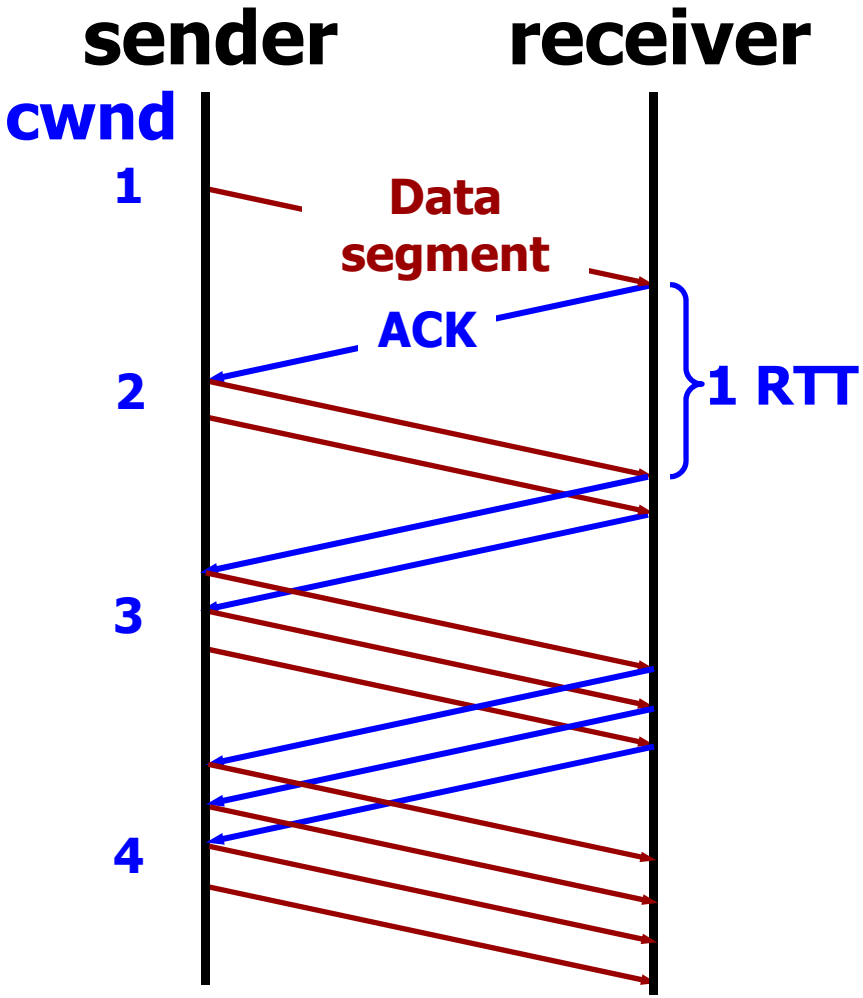
Slow Start/Congestion Avoidance Example

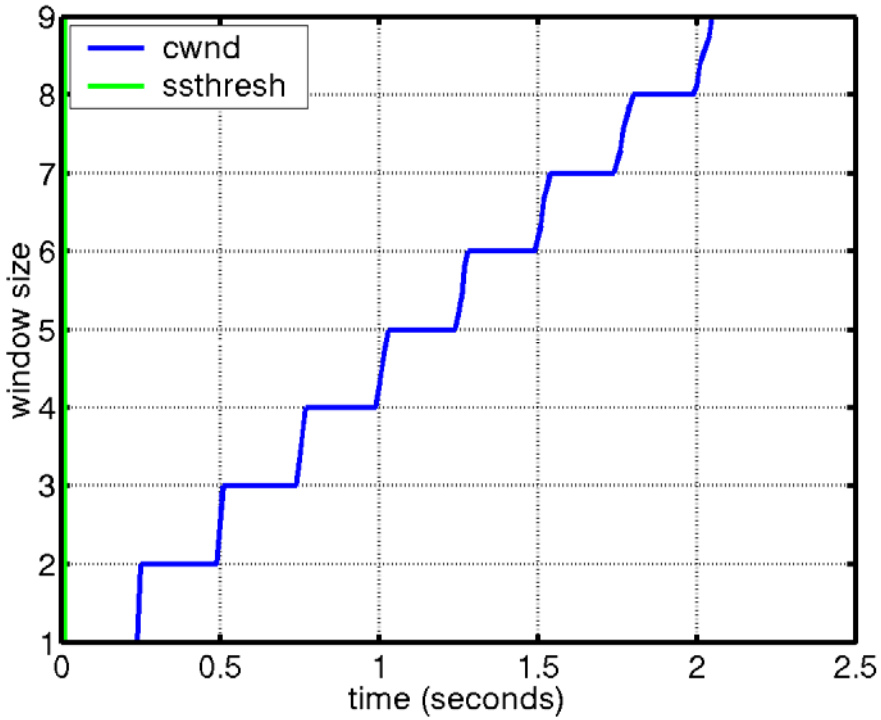
Assume that *ssthresh = 8*





Congestion Avoidance



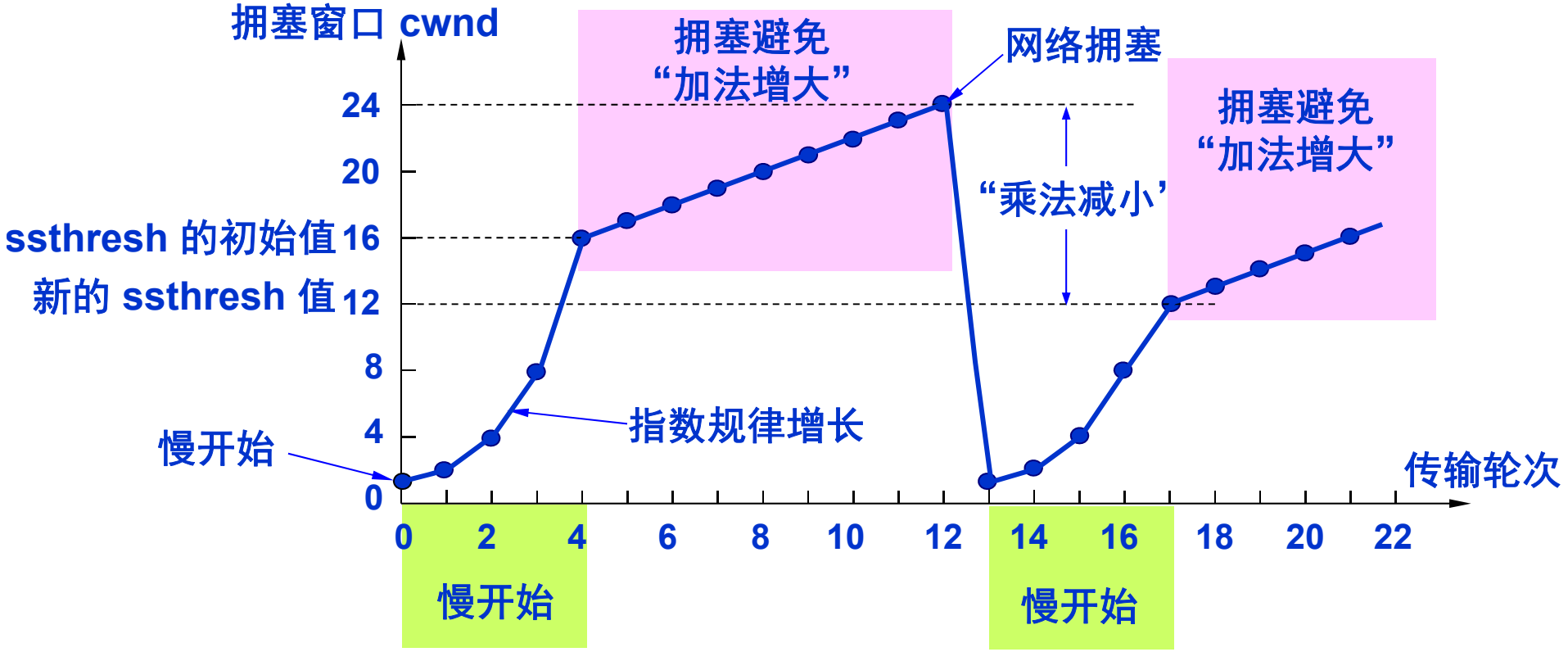


cwnd ← cwnd + 1 (for each cwnd ACKS)

Retransmission Timeout

- When retransmission timeout (Packet loss)
  - ssthresh = cwnd/2
  - cwnd = 1
  - Enter Slow Start

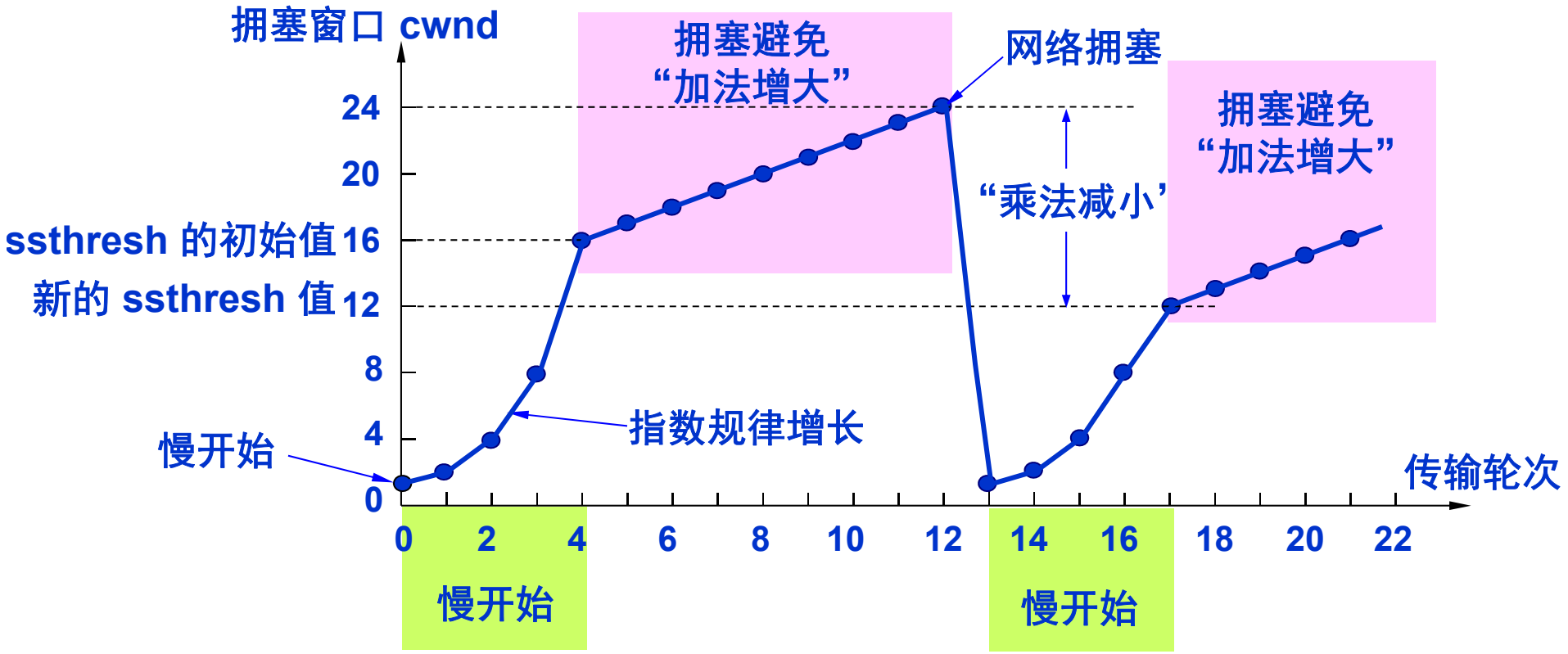
慢开始和拥塞避免算法的实现举例



当 TCP 连接进行初始化时，将拥塞窗口置为 1。图中的窗口单位不使用字节而使用报文段。

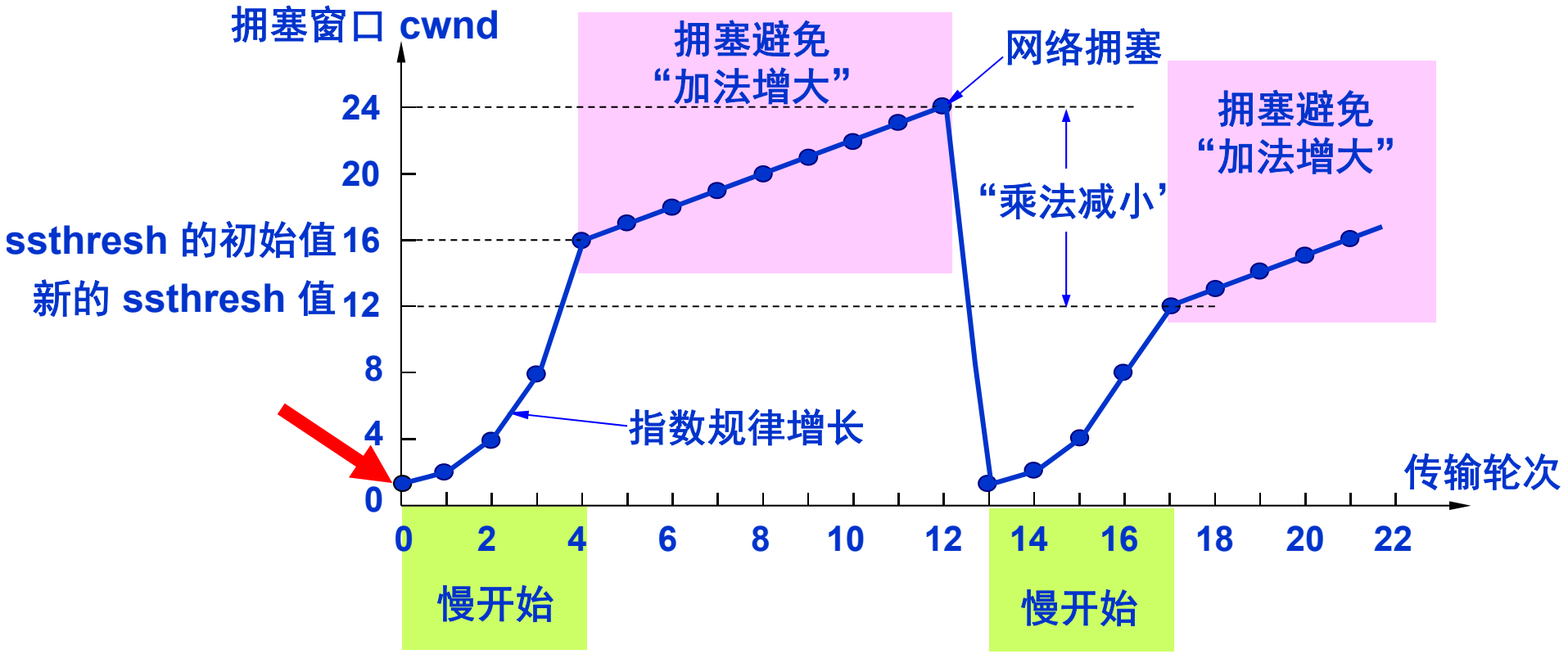
慢开始门限的初始值设置为 16 个报文段，即 ssthresh = 16。

慢开始和拥塞避免算法的实现举例



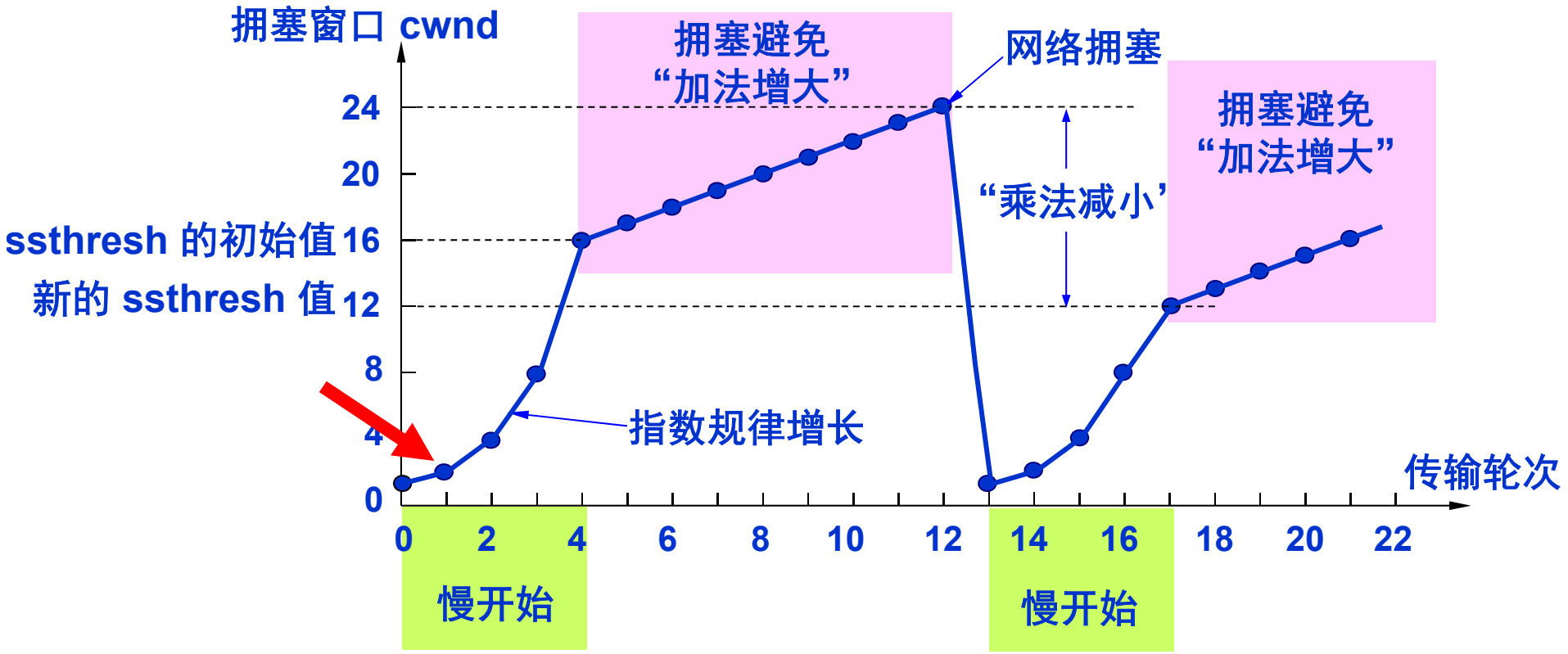
发送端的发送窗口不能超过拥塞窗口 cwnd 和接收端窗口 rwnd 中的最小值。我们假定接收端窗口足够大，因此现在发送窗口的数值等于拥塞窗口的数值。

慢开始和拥塞避免算法的实现举例



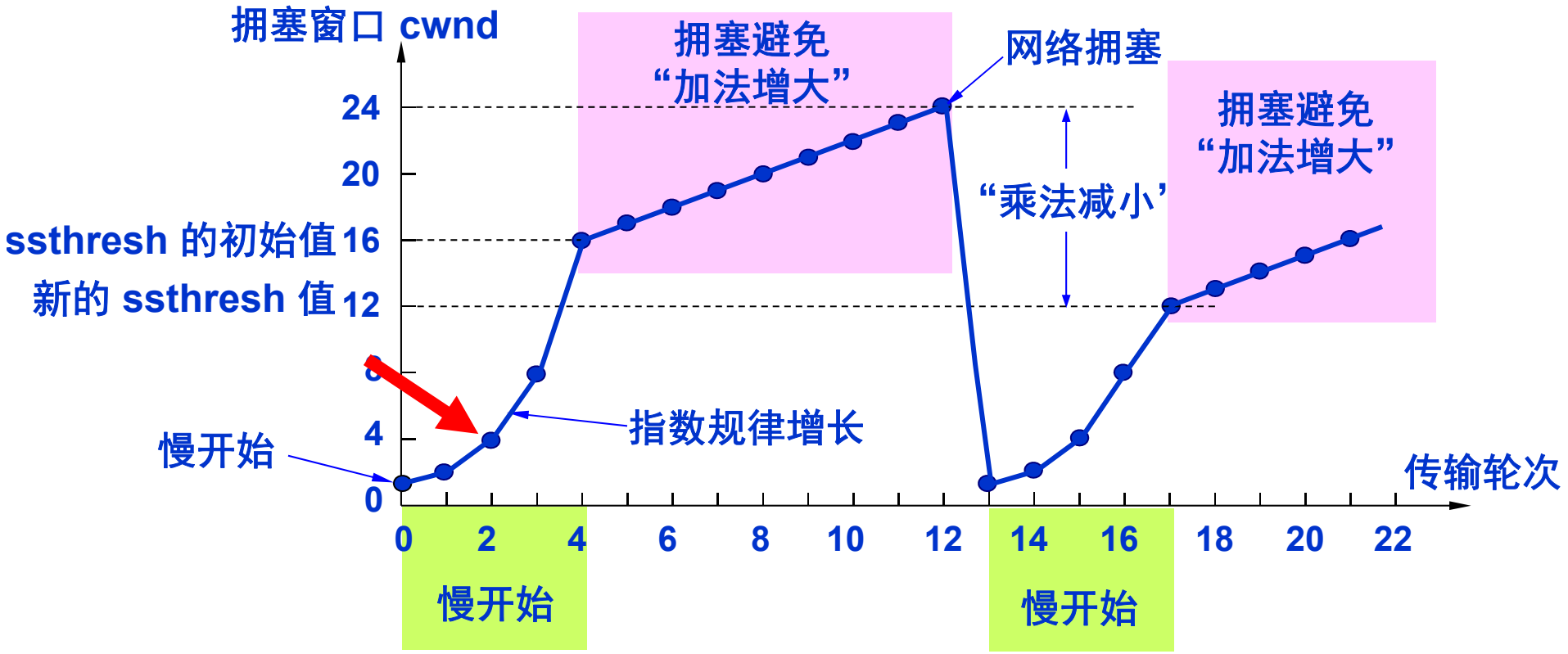
在执行慢开始算法时，拥塞窗口 cwnd=1，发送第一个报文段 M0。

慢开始和拥塞避免算法的实现举例



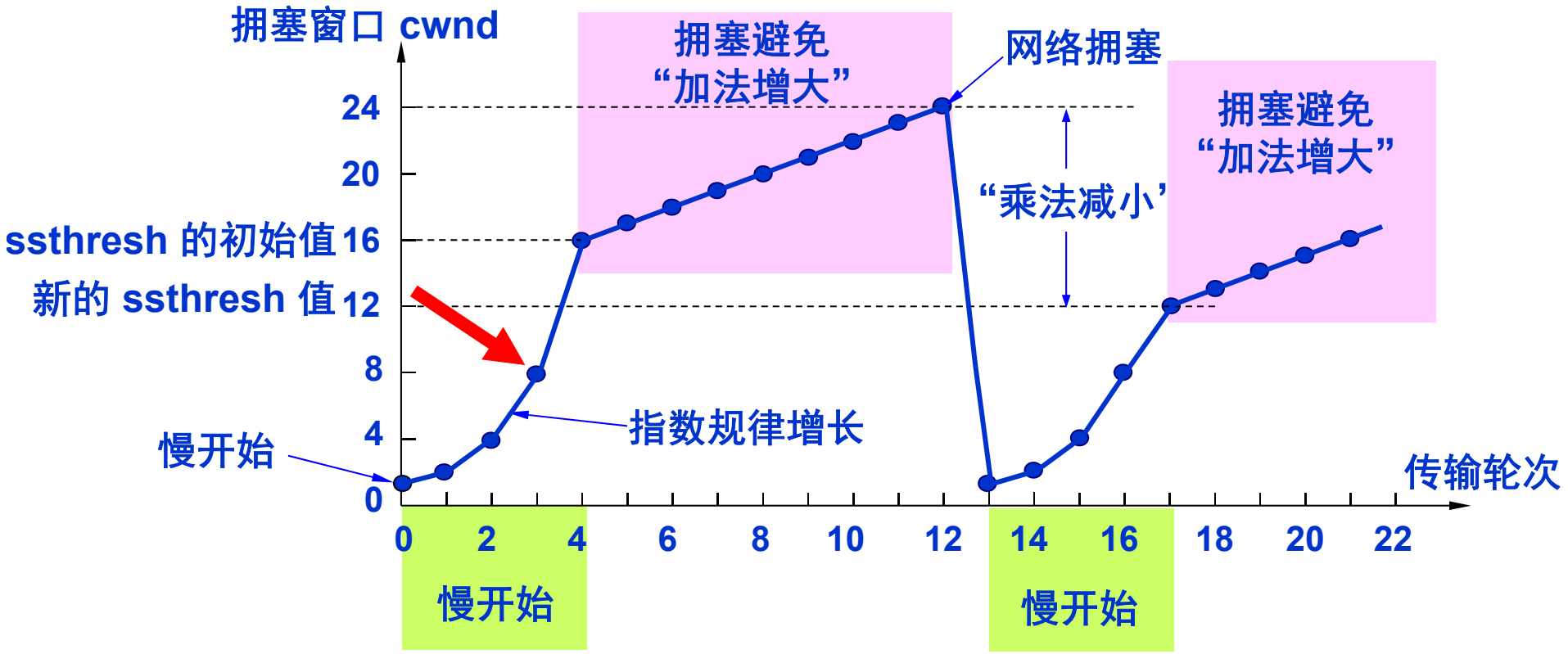
发送端每收到一个确认，就把 cwnd 加 1。于是发送端可以接着发送 M1 和 M2 两个报文段。

慢开始和拥塞避免算法的实现举例



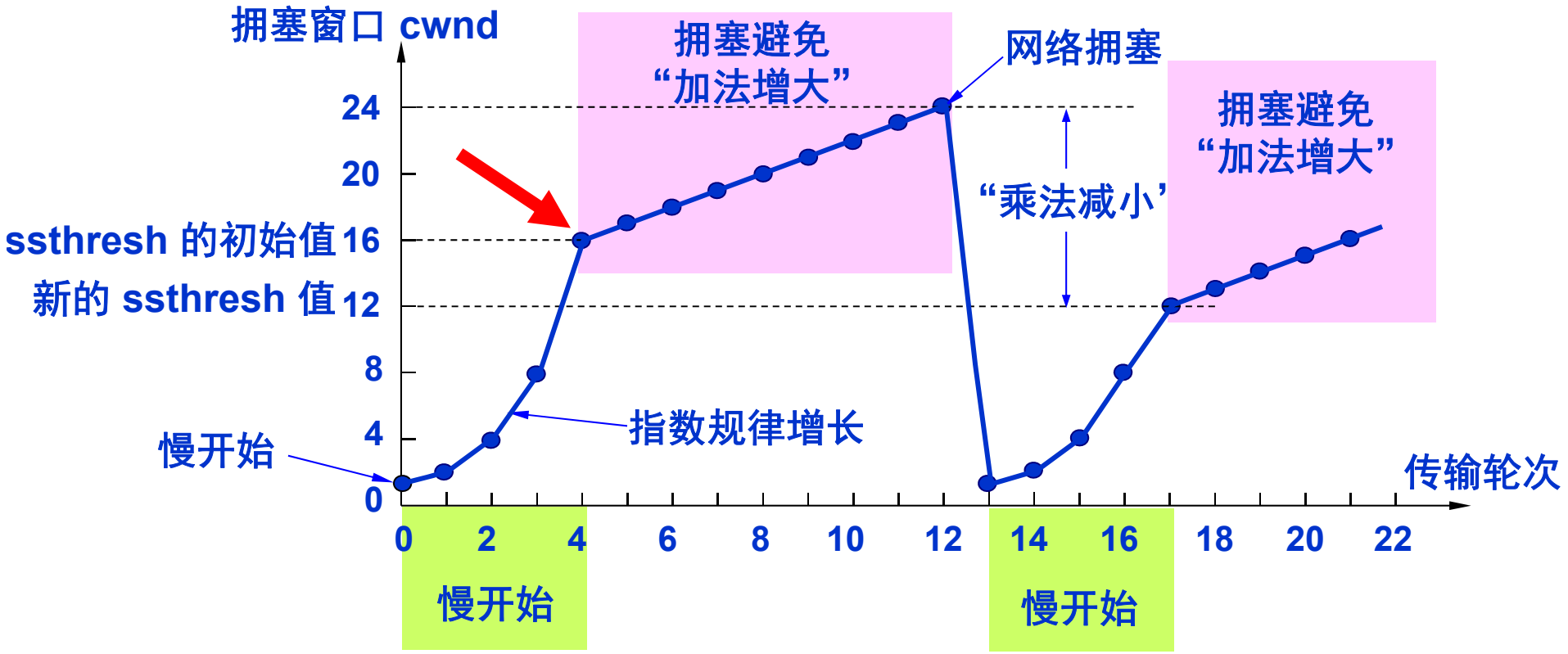
接收端共发回两个确认。发送端每收到一个对新报文段的确认，就把发送端的 cwnd 加 1。现在 cwnd 从 2 增大到 4，并可接着发送后面的 4 个报文段。

慢开始和拥塞避免算法的实现举例



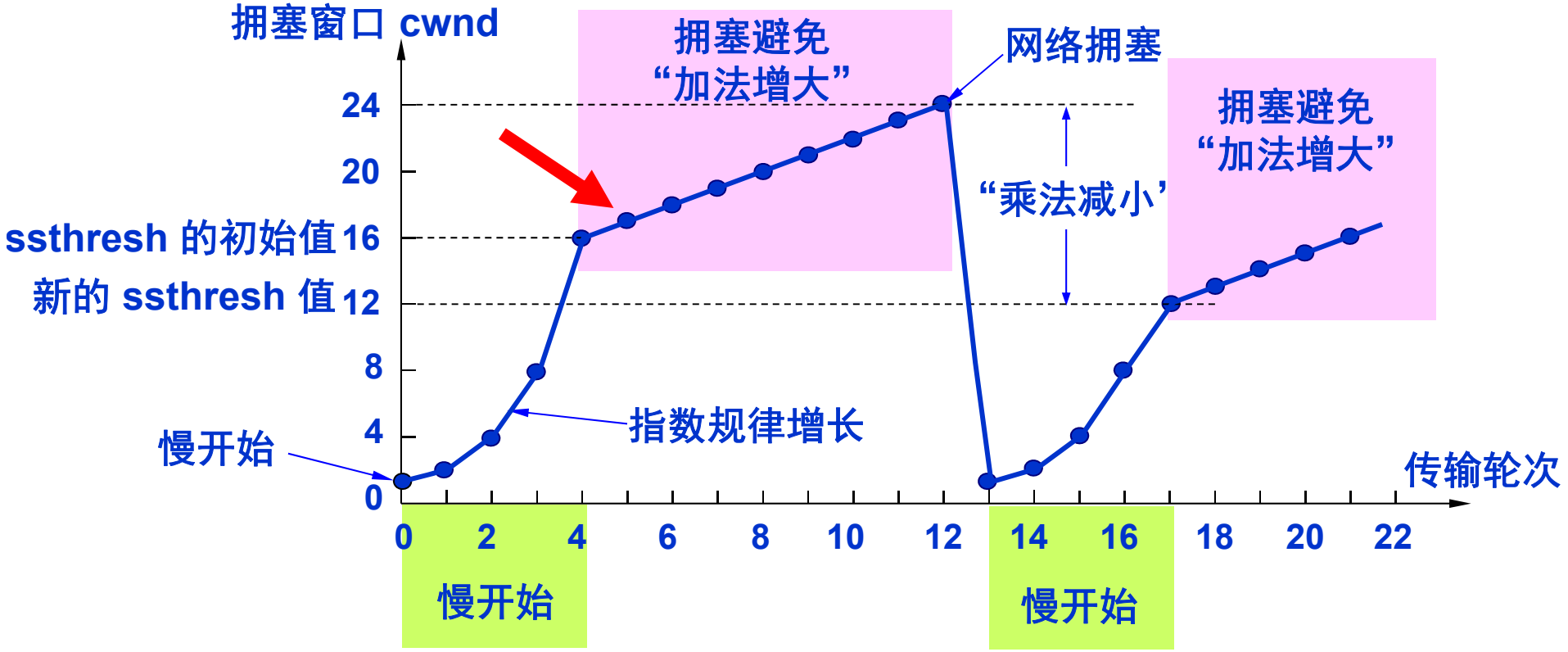
发送端每收到一个对新报文段的确认，就把发送端的拥塞窗口加 1，因此拥塞窗口 cwnd 随着传输轮次按指数规律增长。

慢开始和拥塞避免算法的实现举例



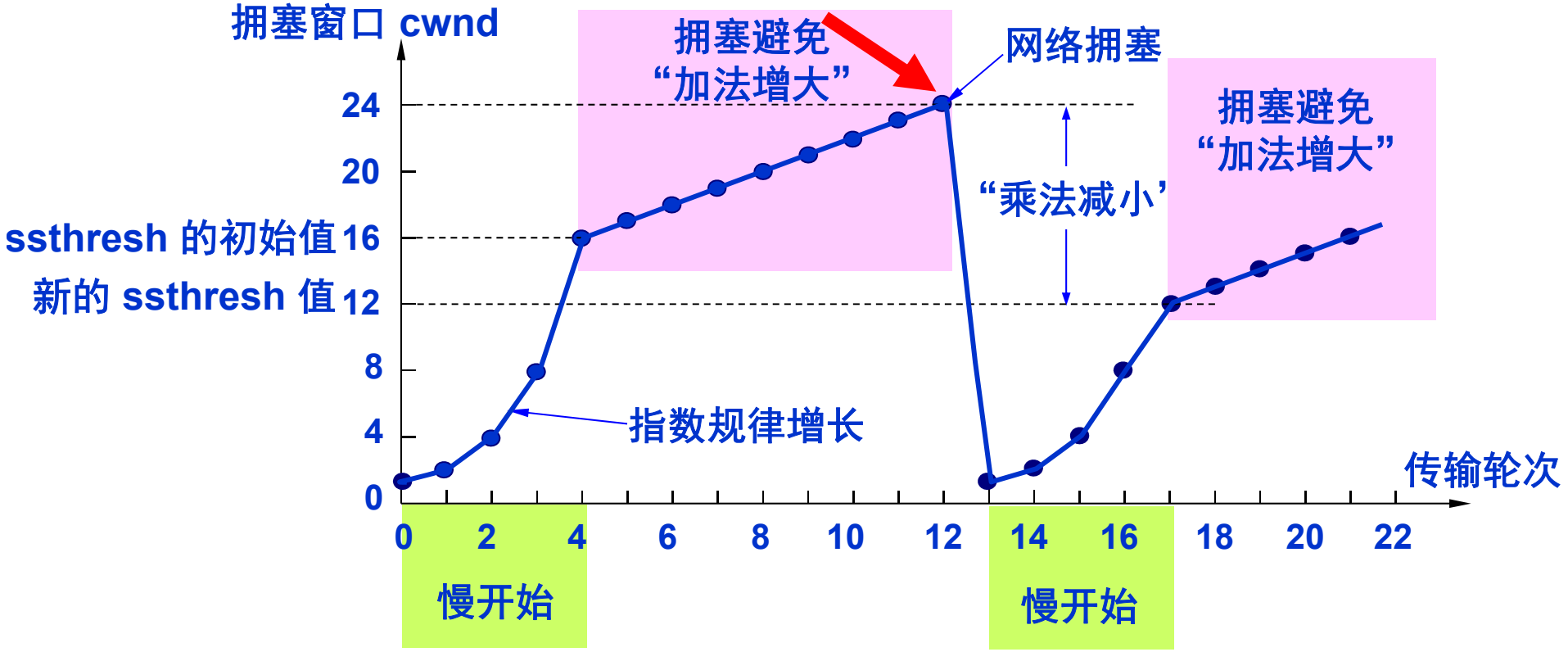
发送端每收到一个对新报文段的确认，就把发送端的拥塞窗口加 1，cwnd < 16，cwnd=cwnd+8=16。

慢开始和拥塞避免算法的实现举例



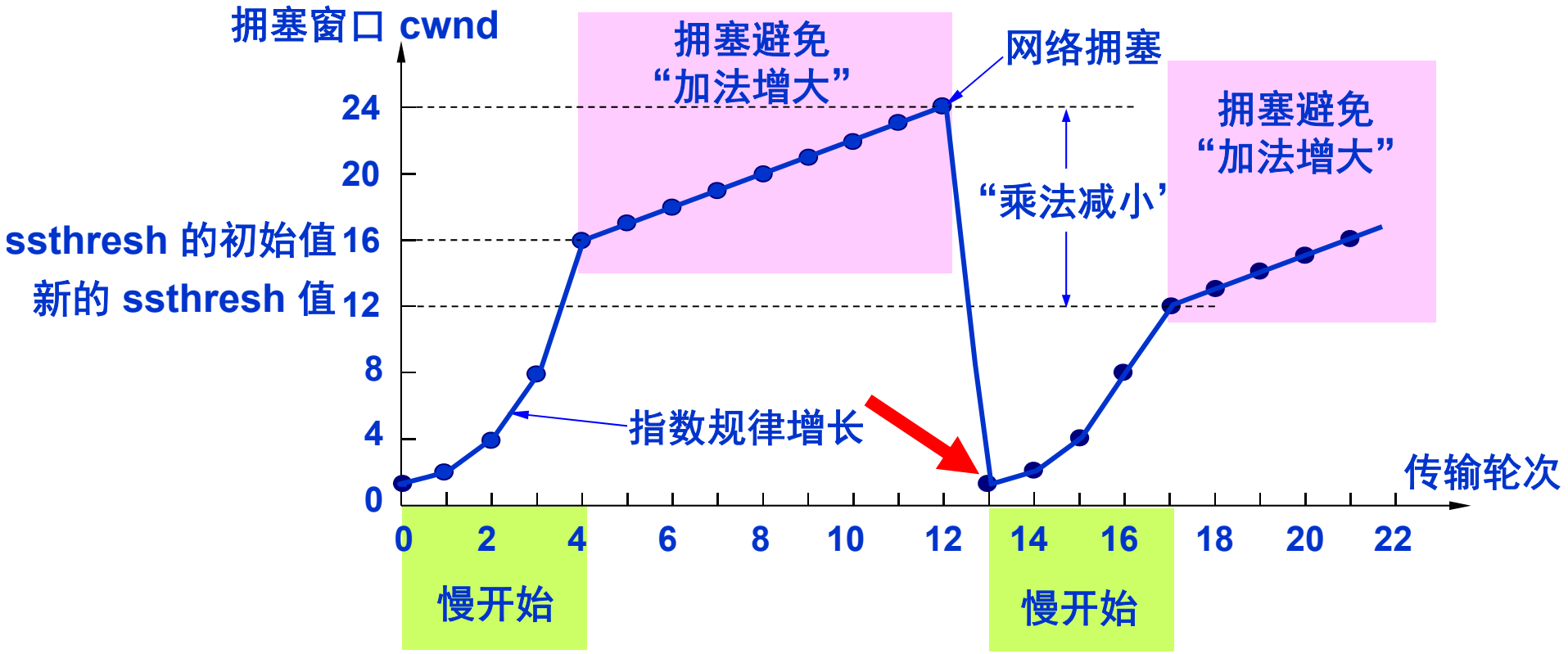
发送端每收到一个对新报文段的确认，就把发送端的拥塞窗口加 1，CWND>16，因此拥塞窗口 cwnd = cwnd + 16*MSS(MSS/CWND)=17。

慢开始和拥塞避免算法的实现举例



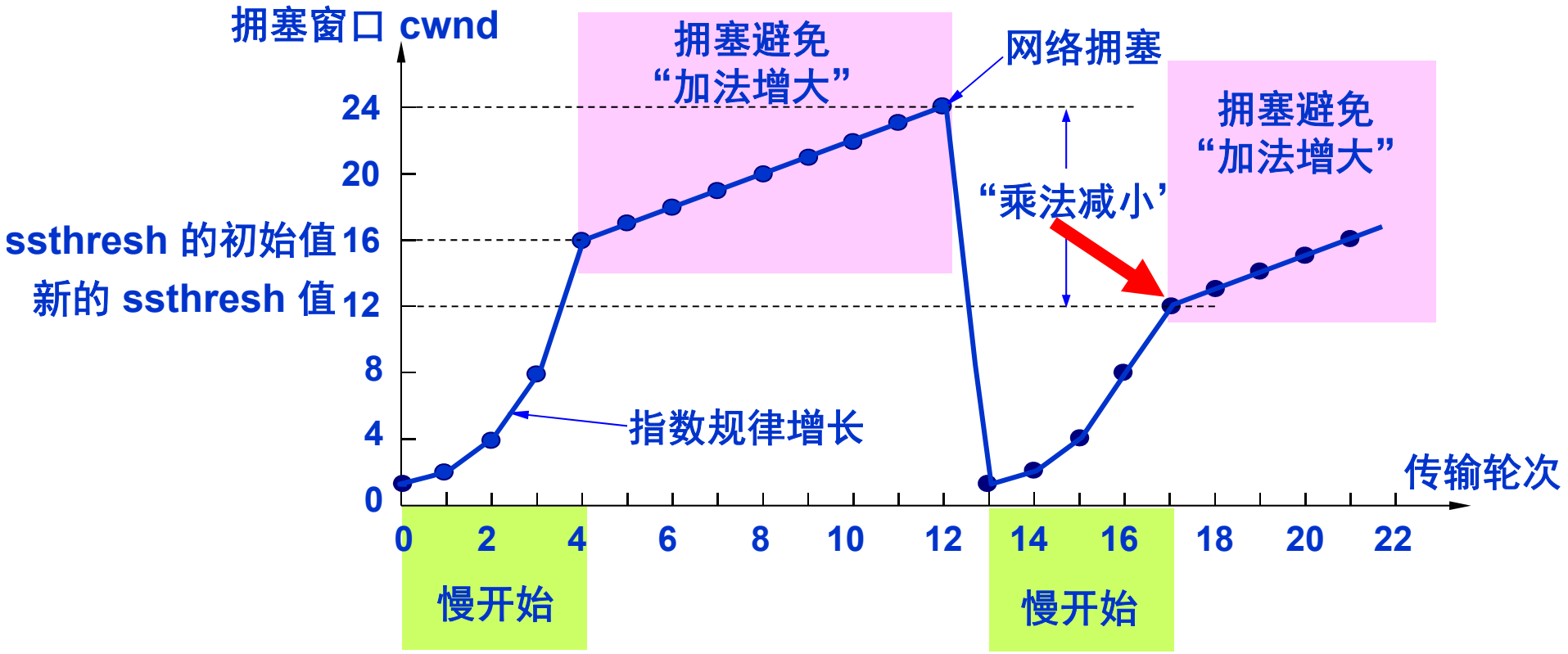
假定拥塞窗口的数值增长到 24 时，网络出现超时，表明网络拥塞了。

慢开始和拥塞避免算法的实现举例



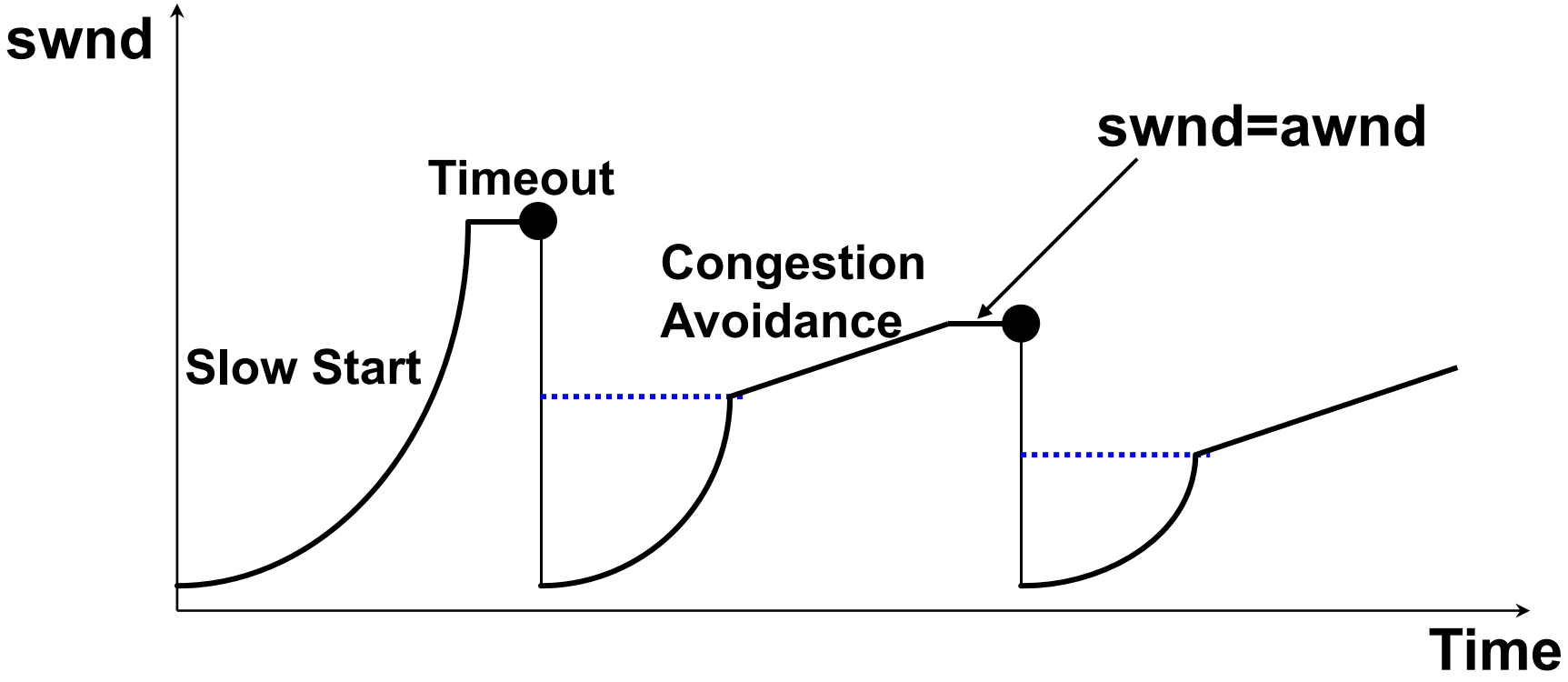
更新后的 ssthresh 值变为 12（即发送窗口数值 24 的一半），拥塞窗口再重新设置为 1，并执行慢开始算法。

慢开始和拥塞避免算法的实现举例



当 cwnd = 12 时改为执行拥塞避免算法，拥塞窗口按按线性规律增长，每经过一个往返时延就增加一个 MSS 的大小。

The big picture



必须强调指出

- “拥塞避免”并非指完全能够避免了拥塞。利用以上的措施要完全避免网络拥塞还是不可能的。
- “拥塞避免”是说在拥塞避免阶段把拥塞窗口控制为按线性规律增长，使网络比较不容易出现拥塞。

Fast Retransmit (快重传)

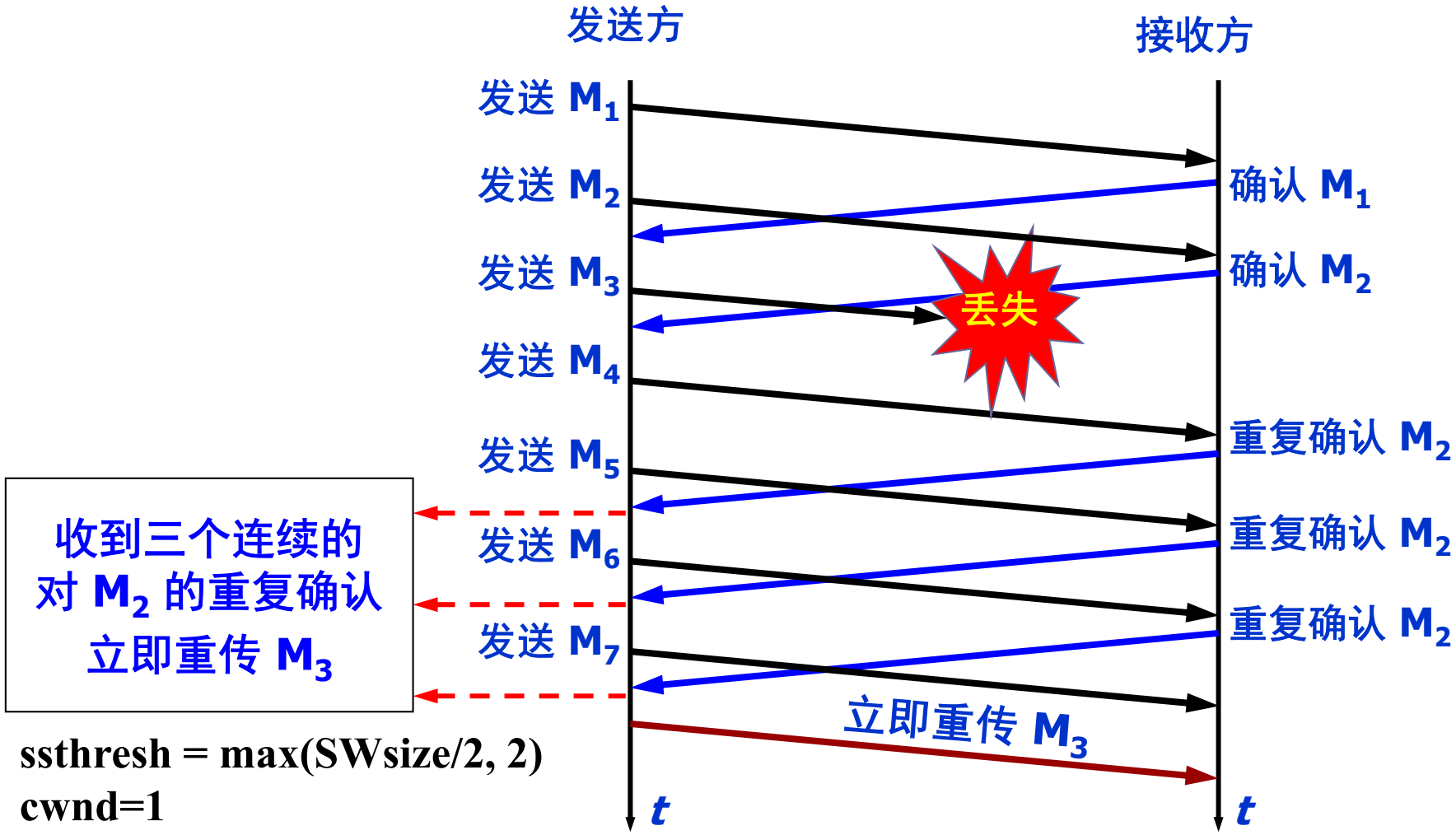
- **Fast Retransmit :** immediately retransmits after 3 dupACKs without waiting for timeout
- Adjusts

  SWsize = min(awnd, cwnd)

  ssthresh ← max(SWsize /2, 2)

  cwnd=1
- Enter Slow Start

Fast Retransmit



Fast Retransmit

Note

Fast Retransmit
Upon receipt of three duplicate ACKs, the TCP Sender retransmits the lost segment.

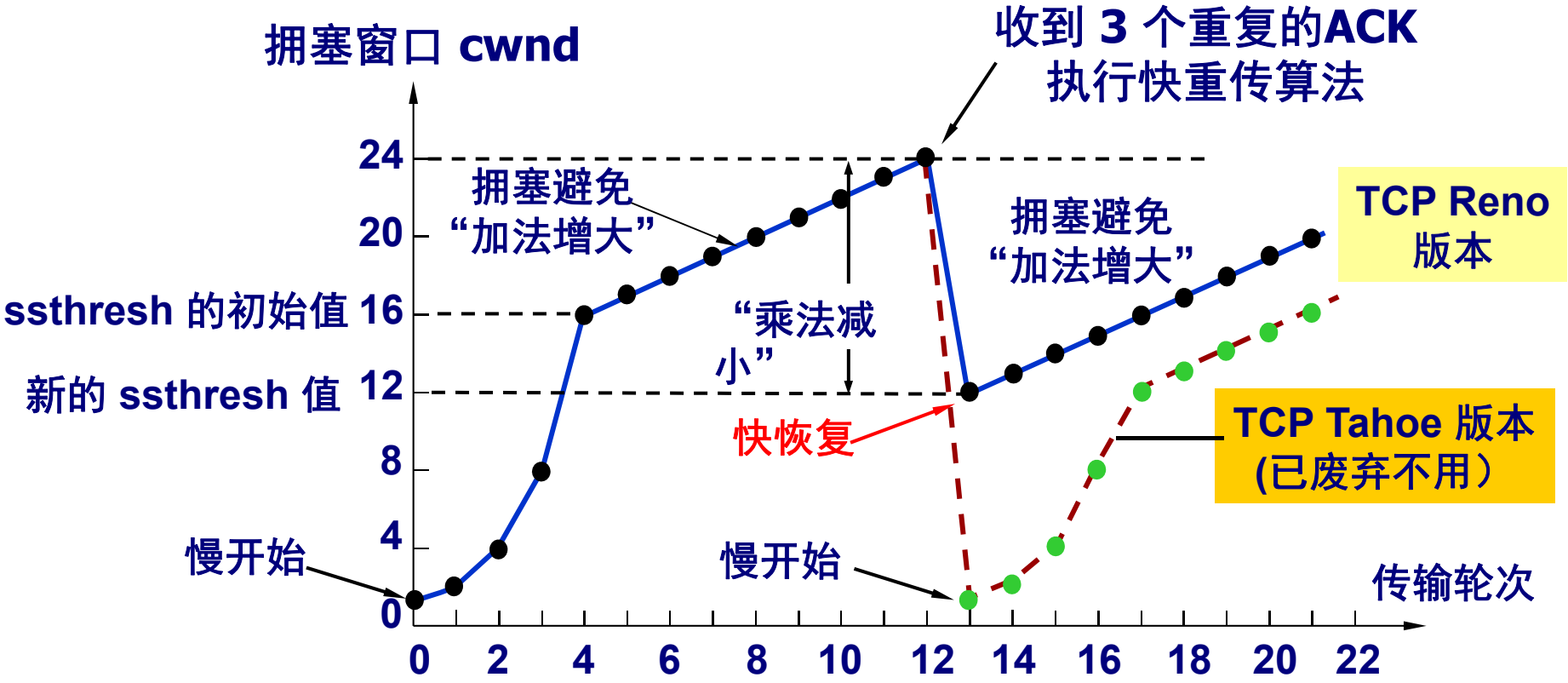
Fast Recovery

- **Fast recovery: added with TCP Reno.**
- **Basic idea:**
  - **When fast retransmit detects three duplicate ACKs, start the recovery process from congestion avoidance (not slow start) region and use ACKs in the pipe to pace the sending of packets.**

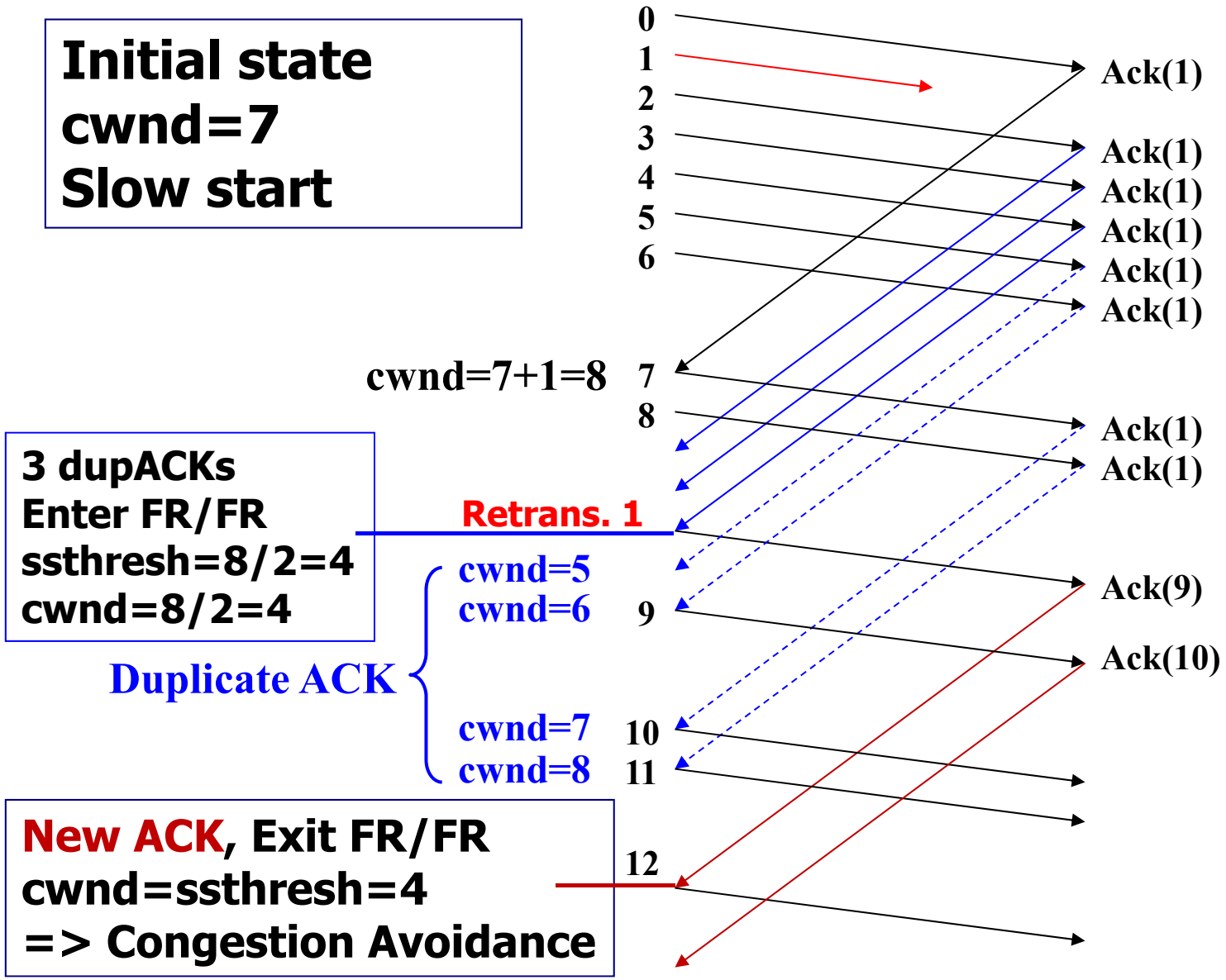
Fast Retransmit & Fast Recovery

- **After receiving *3 dupACKS:* Enter FR/FR**
  - Retransmit the lost segment;
  - Set ssthresh = max(SWsize/2, 2) ;
  - Set cwnd = ssthresh ; /* **window inflation** */
- **If dupACK arrives:**
  - Set cwnd = cwnd + 1;
  - Transmit new segment, if allowed.
- **If new ACK arrives:**
  - Set cwnd = ssthresh ; /* **window deflation** */
  - Exit FR/FR, **enter CA.**
- **If RTO expires: Perform slow-start**
  - ssthresh = SWsize/2 ;
  - cwnd = 1 ;

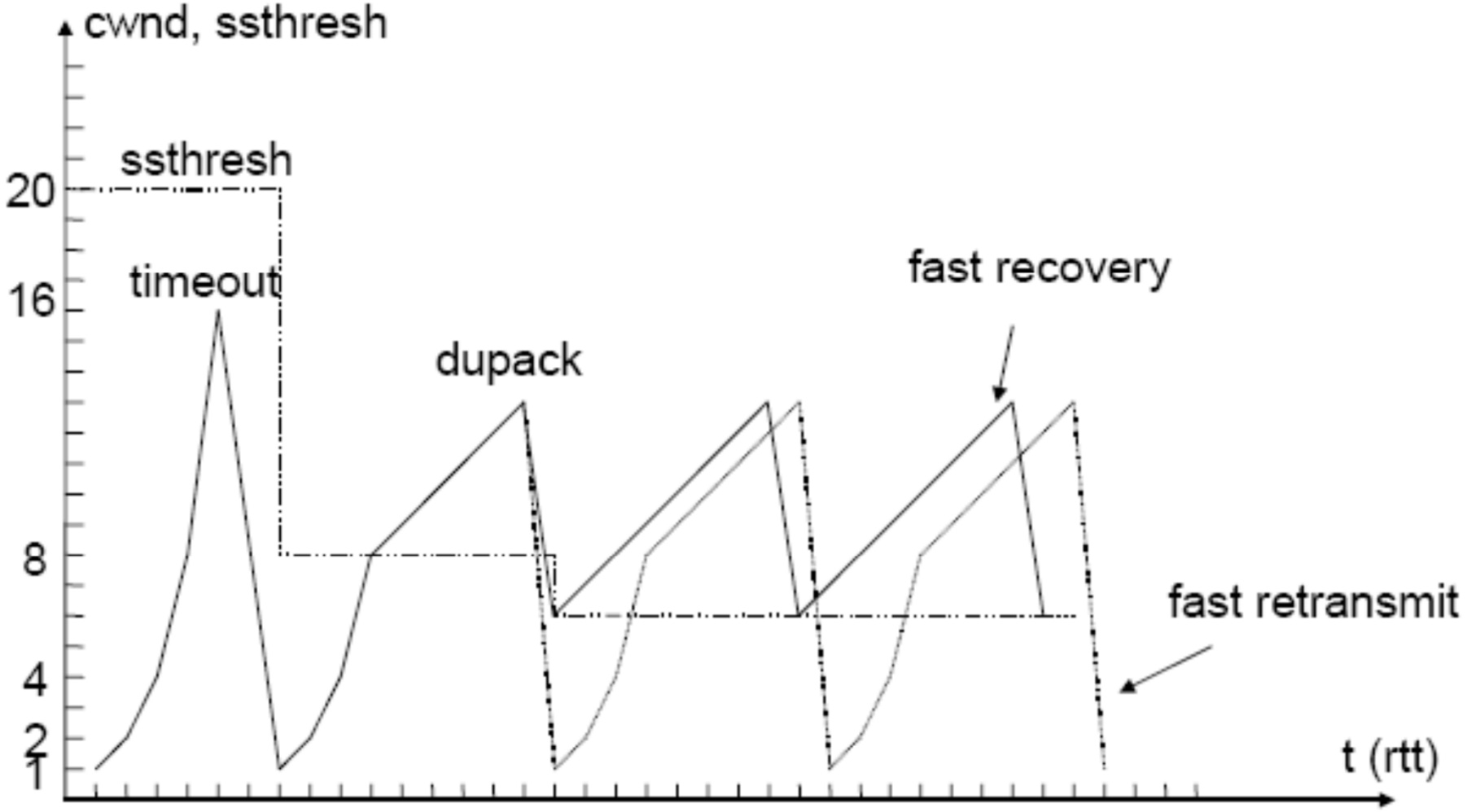
Fast Retransmit & Fast Recovery



Fast Retransmit & Fast Recovery



The big picture



Summary: TCP Congestion Control

- When `CongWin` is below `Threshold`, sender in slow-start phase, window grows exponentially. (Slow Start)
- When `CongWin` is above `Threshold`, sender is in congestion-avoidance phase, window grows linearly. (Congestion Avoidance)
- When a 3 duplicate ACK occurs, `Threshold` set to `CongWin/2` and `CongWin` set to `Threshold`. (Fast Retrans./Fast Recovery)
- When timeout occurs, `Threshold` set to `CongWin/2` and `CongWin` is set to 1 MSS. (Slow Start)

Summary: TCP/Reno Congestion Control

Initially:

```
cwnd = 1;
ssthresh = infinite (64K);
```

For each newly ACKed segment:

```
cwnd = cwnd + MSS; /* slow start */
if (cwnd >= ssthresh) /* congestion avoidance */
    cwnd = cwnd + MSS*MSS/cwnd;
```

Triple-duplicate ACKs:

```
/* Fast Retransmit ; Fast Recovery */
cwnd = ssthresh = cwnd/2;
```

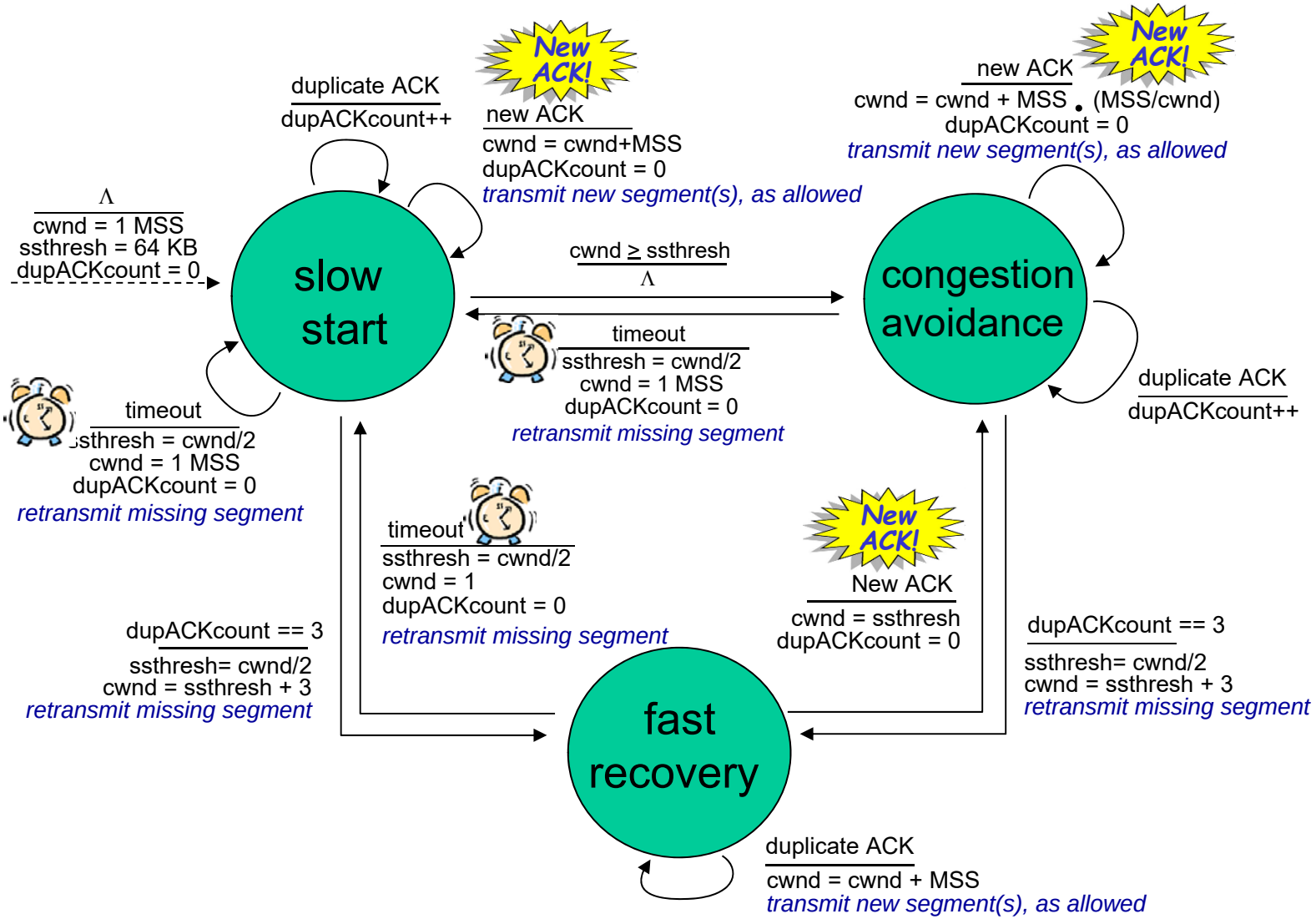
Timeout: `/* slow start */`

```
ssthresh = cwnd/2;
cwnd = 1;
```

TCP sender congestion control

Event	State	TCP Sender Action	Commentary
ACK receipt for previously unacked data	Slow Start (SS)	CongWin = CongWin + MSS, If (CongWin >= Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
ACK receipt for previously unacked data	Congestion Avoidance (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
Loss event detected by triple duplicate ACK	SS or CA	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
Timeout	SS or CA	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
Duplicate ACK	SS or CA	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

Summary: TCP/Reno Congestion Control



例1

- 在一条往返时间为 10ms 的无拥塞线路上使用TCP传输文件。接收窗口为 24KB ， 最大数据段长度为 2KB ， 超时后阈值被设置到16KB。请问需要多长时间才发送满窗口的数据？
- 解：

  慢启动阶段呈指数增长：2KB，4KB，8KB，16KB，从这个点开始，按线性增长：18KB，20KB，22KB，24KB，此时拥塞窗口达到接收方窗口的大小，共用时间70ms。

Summary

- **Transport layer**
  - End-end communication
- **Transport Services**
  - connection-oriented
  - unreliable connectionless
- **Transport address = TSAP address**
  - Well-known TSAP addresses
  - Name servers
  - Daemon (process server)

Summary

- **Transport Connection Establishment**
  - 3-way handshake
- **Transport Connection Release**
  - Asymmetric
  - Symmetric
  - Three-way-handshake + timers
  - Half close
- **Flow control**
  - Sliding-window (varying window size)
- **Multiplexing**
  - Upward
  - Downward

Summary

- The Internet Transport Layer
  - Service
    - Connectionless, unreliable (UDP)
    - Connection-oriented, reliable (TCP)
  - Address
    - Port number
    - Multiplexing
  - Socket
    - endpoint identifier = IP addr. + port number

Summary

- UDP
  - "best effort" transport protocol
  - Connectionless, unreliable
- TCP
  - Connection-Oriented
  - Byte Stream Delivery
  - Reliable, in-order Delivery
  - Full-Duplex
  - Flow control
  - Congestion control

Summary

- TCP
  - Connection
    - setup: 3-way handshaking
    - Terminate: half-close
  - Reliability
    - Checksum
    - Sequence numbers
    - Acknowledgement
    - Timeouts
    - Retransmission

Summary

- TCP
  - Flow control & Congestion control
    - Sliding-window
  - Congestion control
    - By sender
    - Window-based, sliding window
    - Detection: loss, 3-dup ACKs
    - Policy: AIMD
    - 2 parameters: congestion window, threshold
    - Slow Start, Congestion Avoidance, Fast retransmission, Fast recovery