COMPUTER NETWORKS- (20MCA23C)
UNIT-V
'TRANSPORT LAYER'

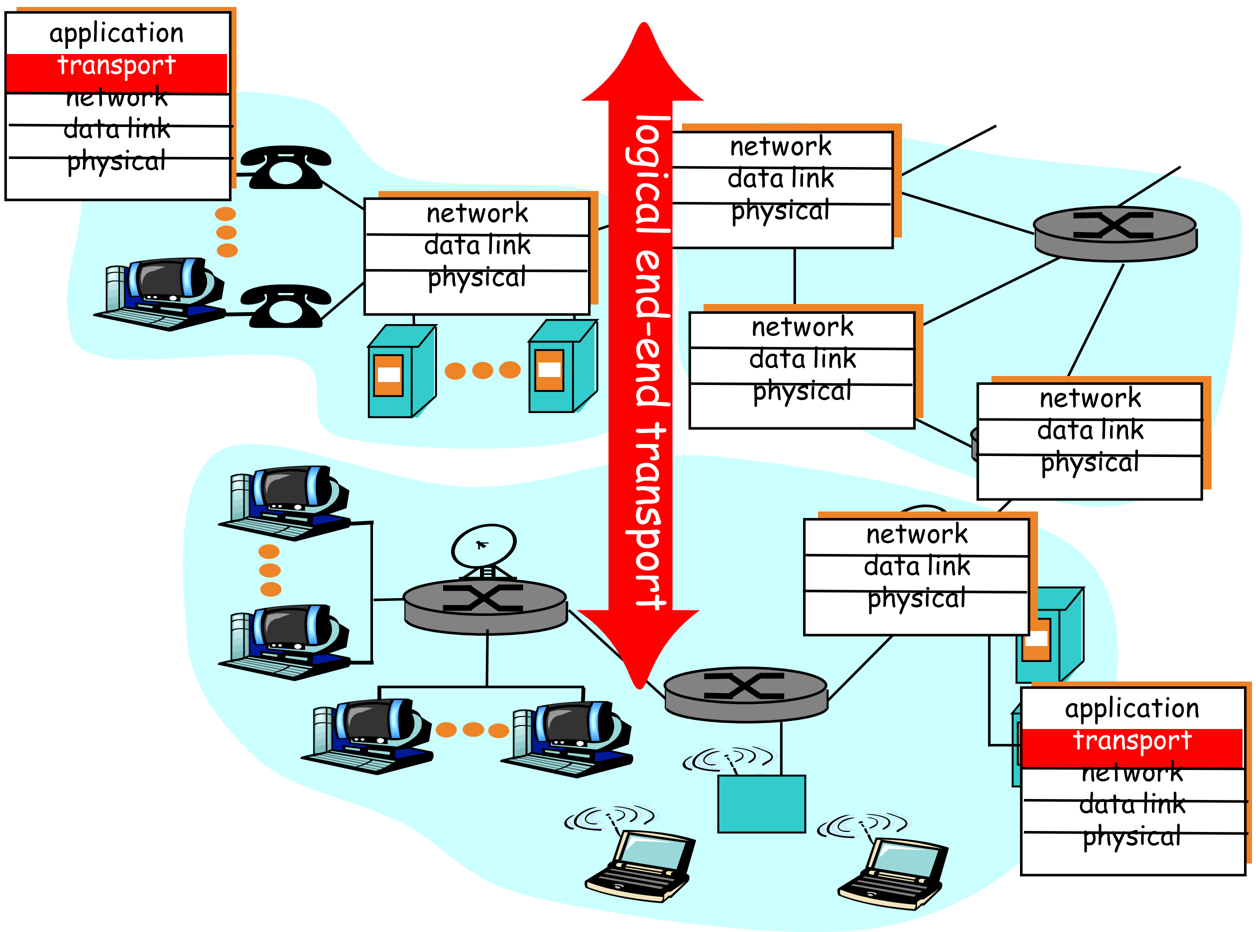
FACULTY:

DR. R. A. ROSELINE, M.SC., M.PHIL., PH.D.,
ASSOCIATE PROFESSOR AND HEAD,
POST GRADUATE AND RESEARCH DEPARTMENT OF COMPUTER APPLICATIONS,
GOVERNMENT ARTS COLLEGE (AUTONOMOUS), COIMBATORE – 641 018.

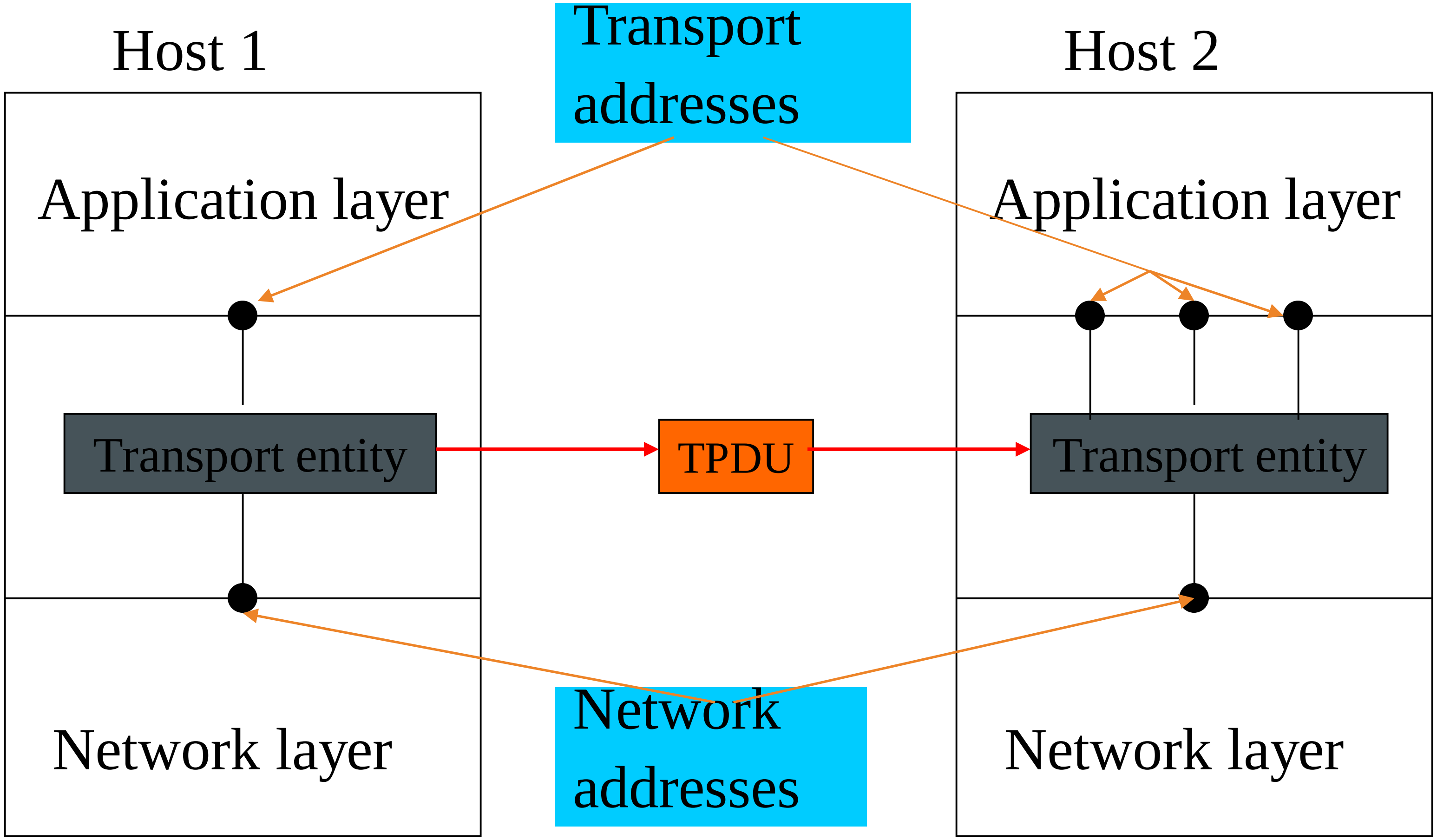
TRANSPORT LAYER

- Services
- Elements of transport protocol
- Simple transport protocol
- UDP
- Remote Procedure Call (see Distributed Systems)
- TCP

LAYER OVERVIEW



LAYER OVERVIEW



SERVICES

- To upper layer
  - efficient, reliable, cost-effective service
  - 2 kinds
    - Connection oriented
    - Connectionless

SERVICES

- needed from network layer
  - packet transport between hosts
  - relationship network <> transport
    - Hosts <> processes
    - Transport service
      - independent network
      - more reliable
    - Network
      - run by carrier
      - part of communication subnet for WANs

SIMPLE SERVICE: PRIMITIVES

- Simple primitives:
  - connect
  - send
  - receive
  - disconnect
- 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

No TPDU

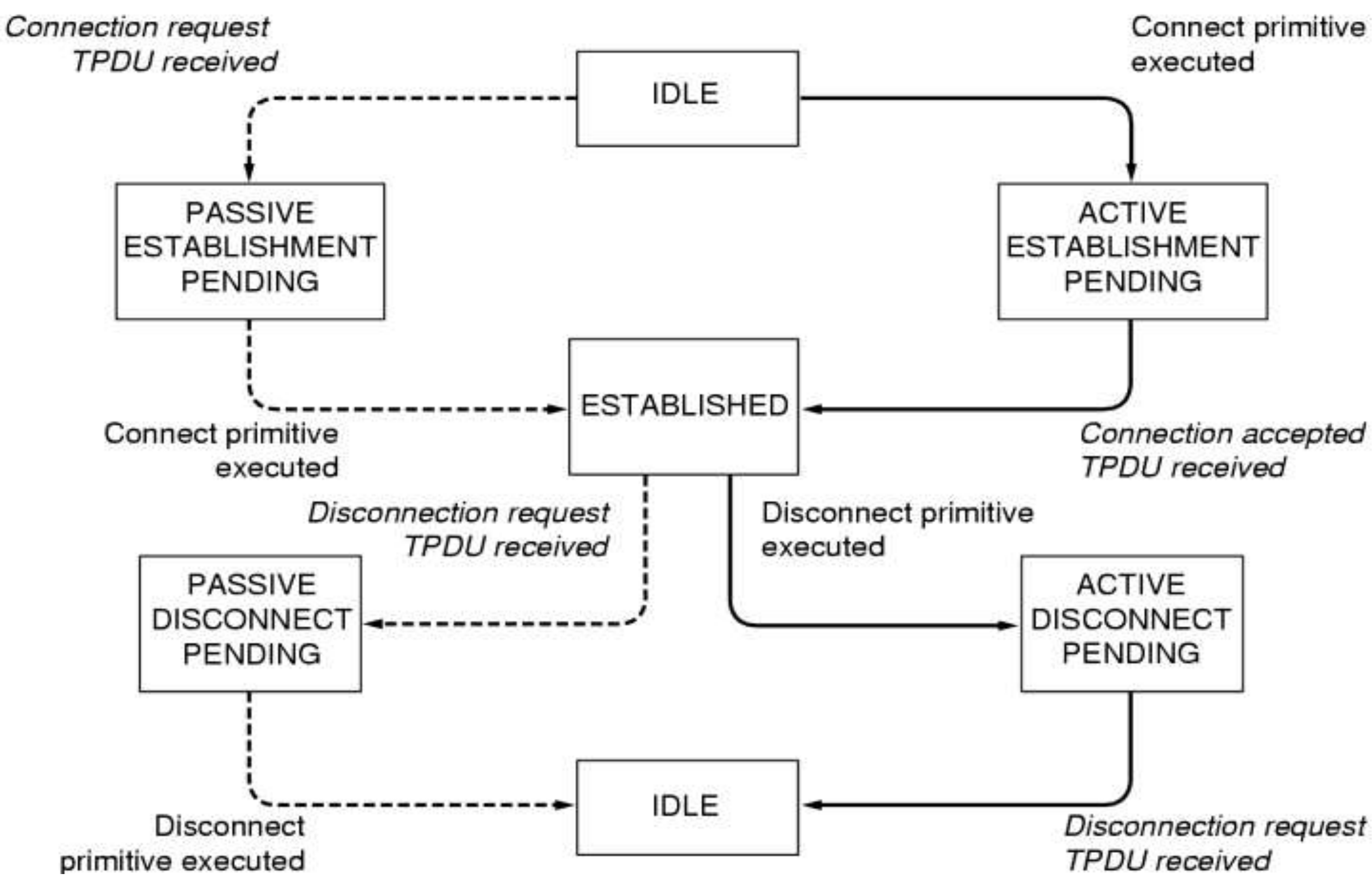
Connection Request TPDU

Data TPDU

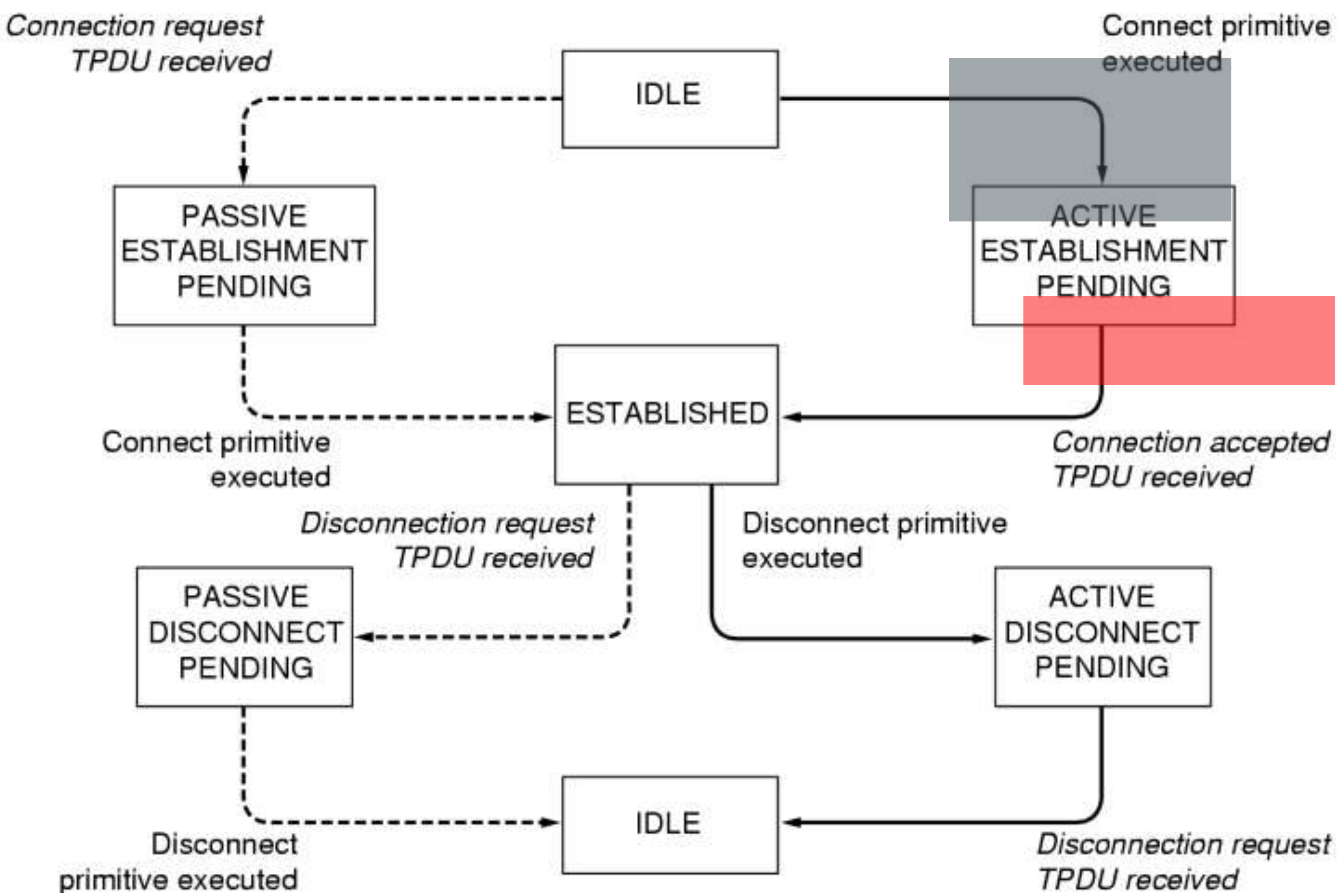
No TPDU

Disconnect TPDU

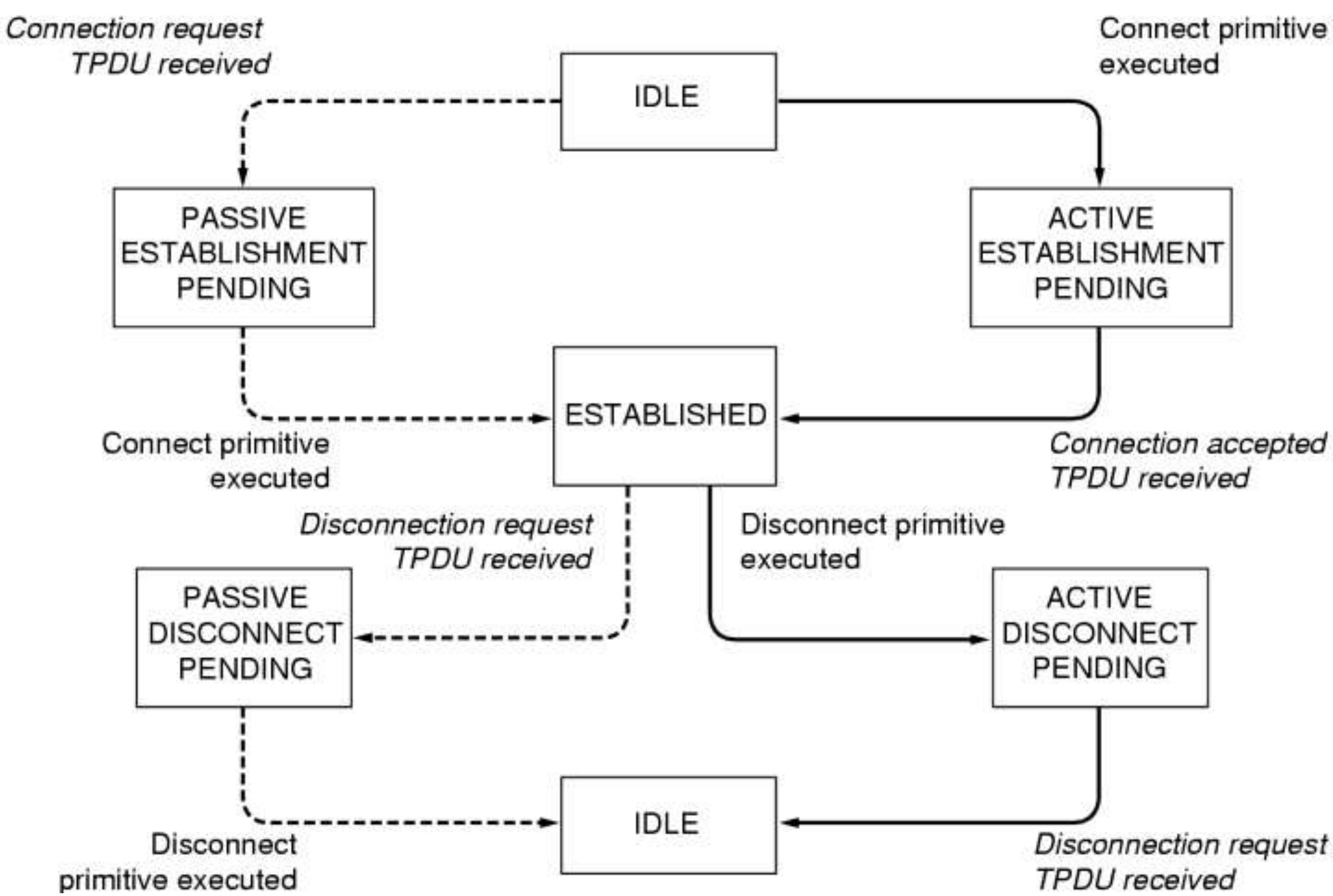
SIMPLE SERVICE: STATE DIAGRAM



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

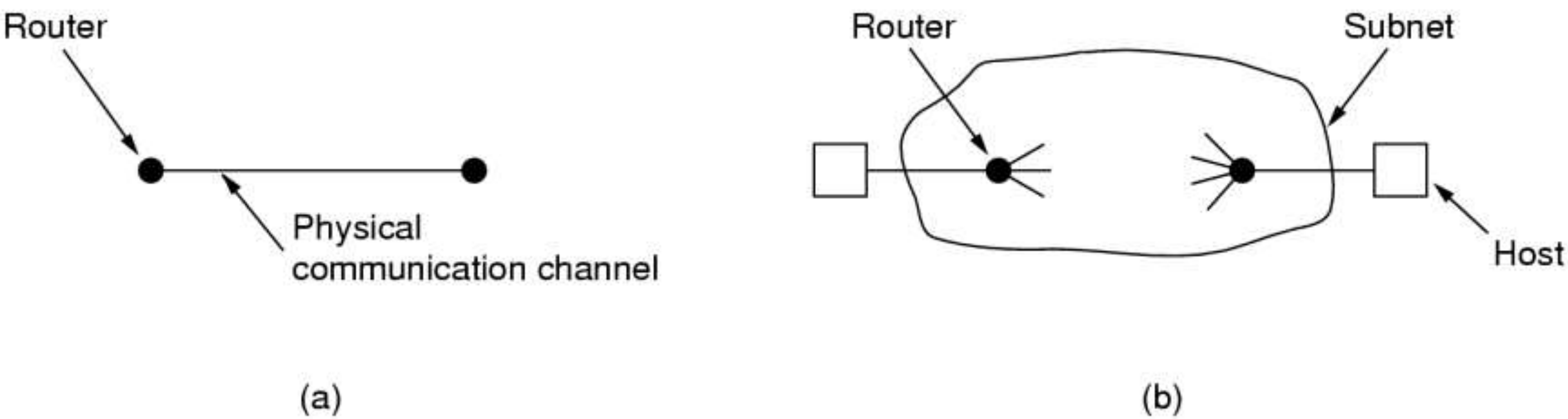
TRANSPORT LAYER

- Services
- Elements of transport protocol
- Simple transport protocol
- UDP
- Remote Procedure Call (see Distributed Systems)
- TCP

ELEMENTS OF TRANSPORT PROTOCOLS (ETP)

- Transport <> Data Link
- Addressing
- Establishing a connection
- Releasing a connection
- Flow control and buffering
- Multiplexing
- Crash recovery

ETP: TRANSPORT <> DATA LINK



Explicit addressing

Connection establishment

Potential existence of storage capacity in subnet

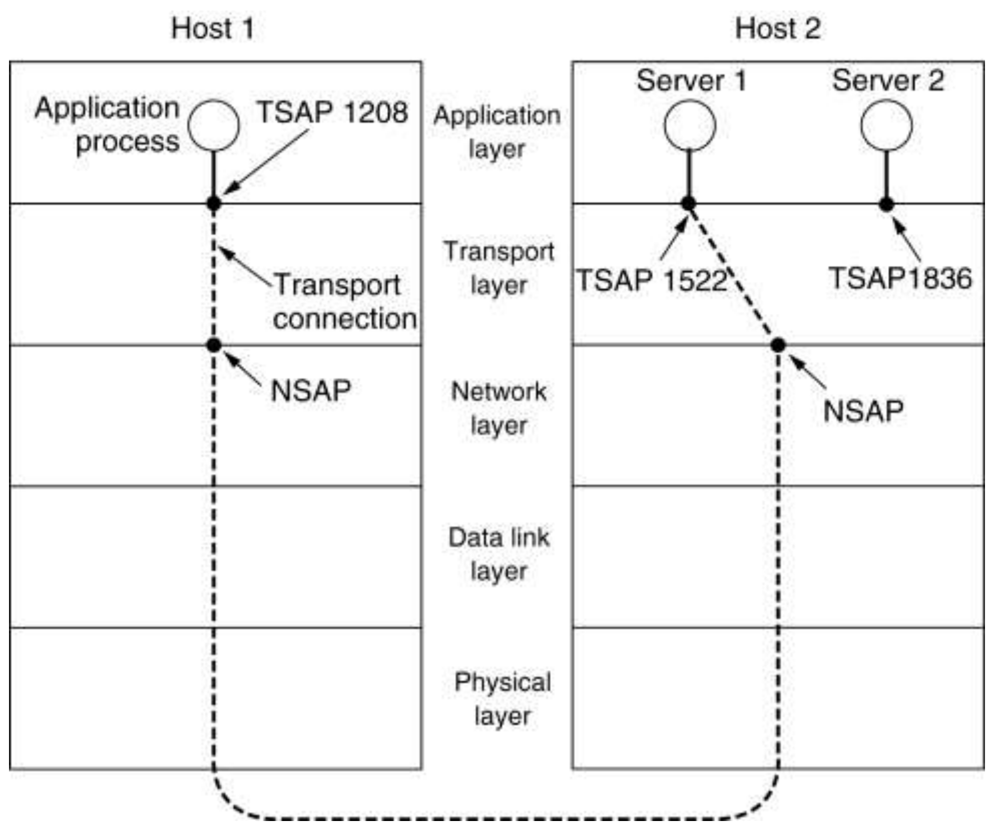
Dynamically varying number of connections

ETP: ADDRESSING

- TSAP = transport service access point
  - Internet: IP address + local port
  - ATM: AAL-SAPs
- Connection scenario
- Getting TSAP addresses?
- From TSAP address to NSAP address?

ETP: ADDRESSING

- Connection scenario



ETP: ADDRESSING

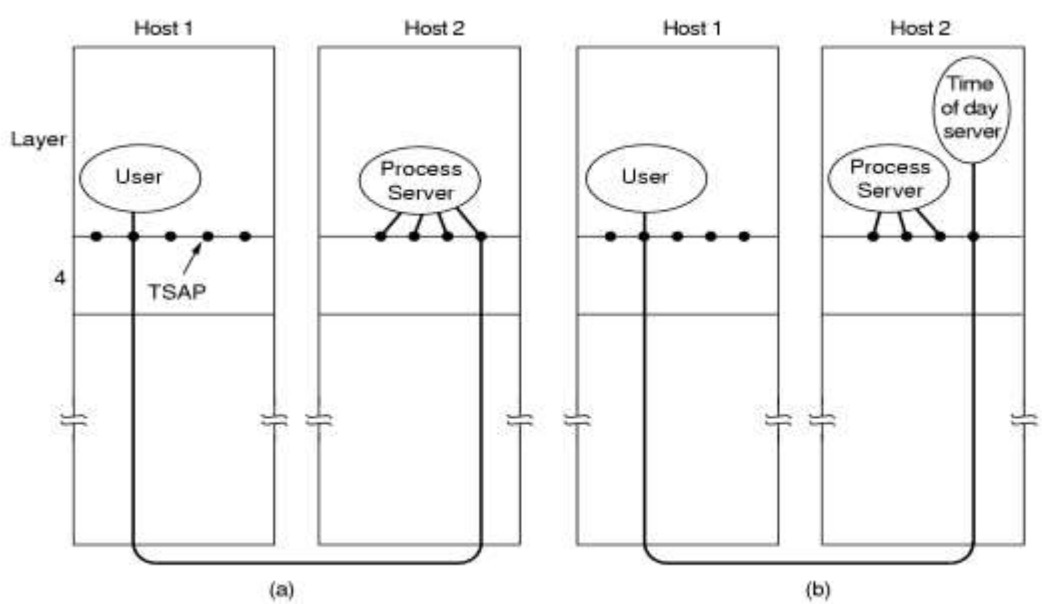
- Connection scenario
  - Host 2 (server)
    - Time-of-day server attaches itself to TSAP 1522
  - Host 1 (client)
    - Connect from TSAP 1208 to TSAP 1522
    - Setup network connection to host 2
    - Send transport connection request
  - Host 2
    - Accept connection request

ETP: 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

ETP: ADDRESSING

- Getting TSAP addresses?
  - Initial connection protocol
    - **to avoid many waiting servers → one process server**
      - **waits on many TSAPs**
      - **creates requested server**



ETP: 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

ETP: ESTABLISHING A CONNECTION

- Problem: delayed duplicates!
- Scenario:
  - Correct bank transaction
    - connect
    - data transfer
    - disconnect
  - Problem: same packets are received in same order a second time!

Recognized?

ETP: ESTABLISHING A CONNECTION

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

- Satisfactory solutions

ETP: ESTABLISHING A CONNECTION

- Satisfactory solutions
  - Ensure limited packet lifetime (incl. Acks)
  - Mechanisms
    - prevent packets from looping + bound congestion delay
    - hopcounter 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

ETP: ESTABLISHING A CONNECTION

- Tomlinson's method
  - requires: clock in each host
    - Number of bits > number of bits in sequence number
    - Clock keeps running, even when a hosts crashes
  - Basic idea:

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

ETP: ESTABLISHING A CONNECTION

- Tomlinson's method

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

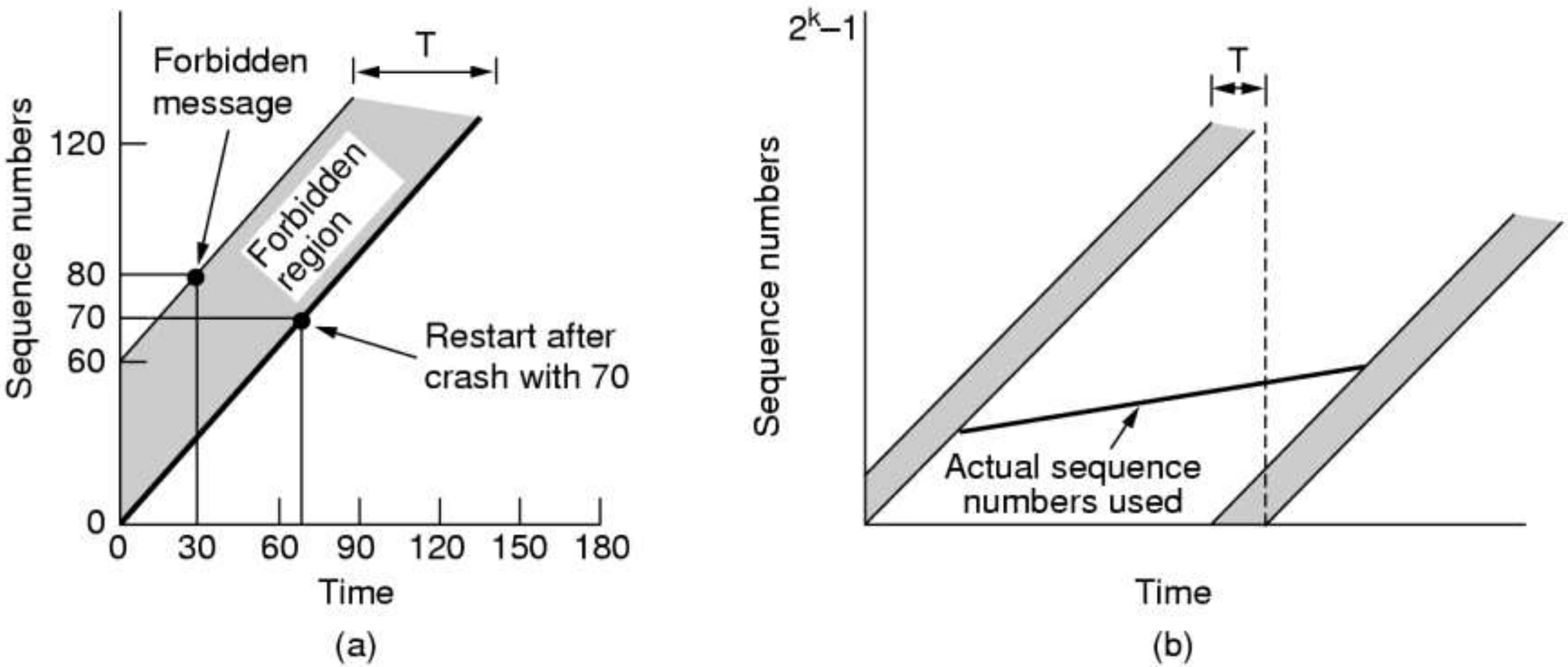
- Problems to solve
  - Selection of the initial sequence number for a new connection
  - Wrap around of sequence numbers for an active connection
  - Handle host crashes

➔ Forbidden region

ETP: ESTABLISHING A CONNECTION

- 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

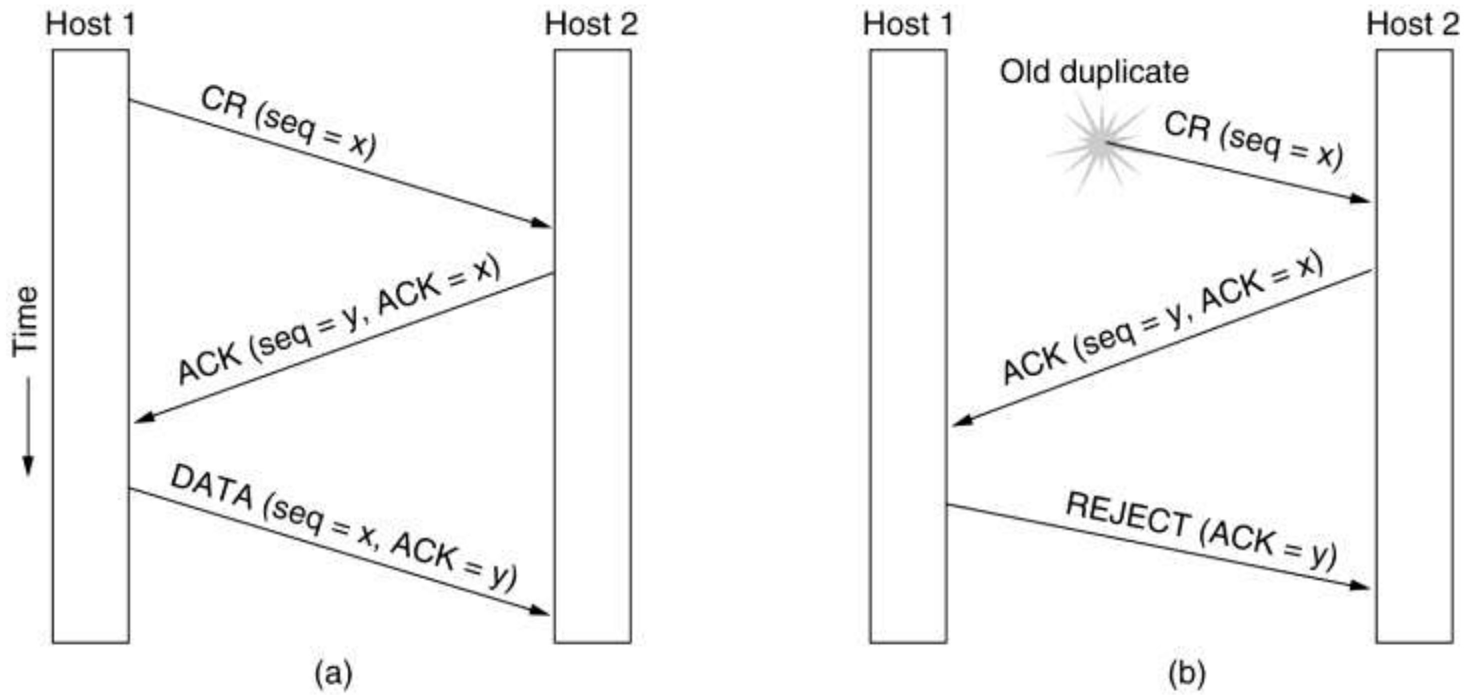
ETP: ESTABLISHING A CONNECTION



ETP: ESTABLISHING A CONNECTION

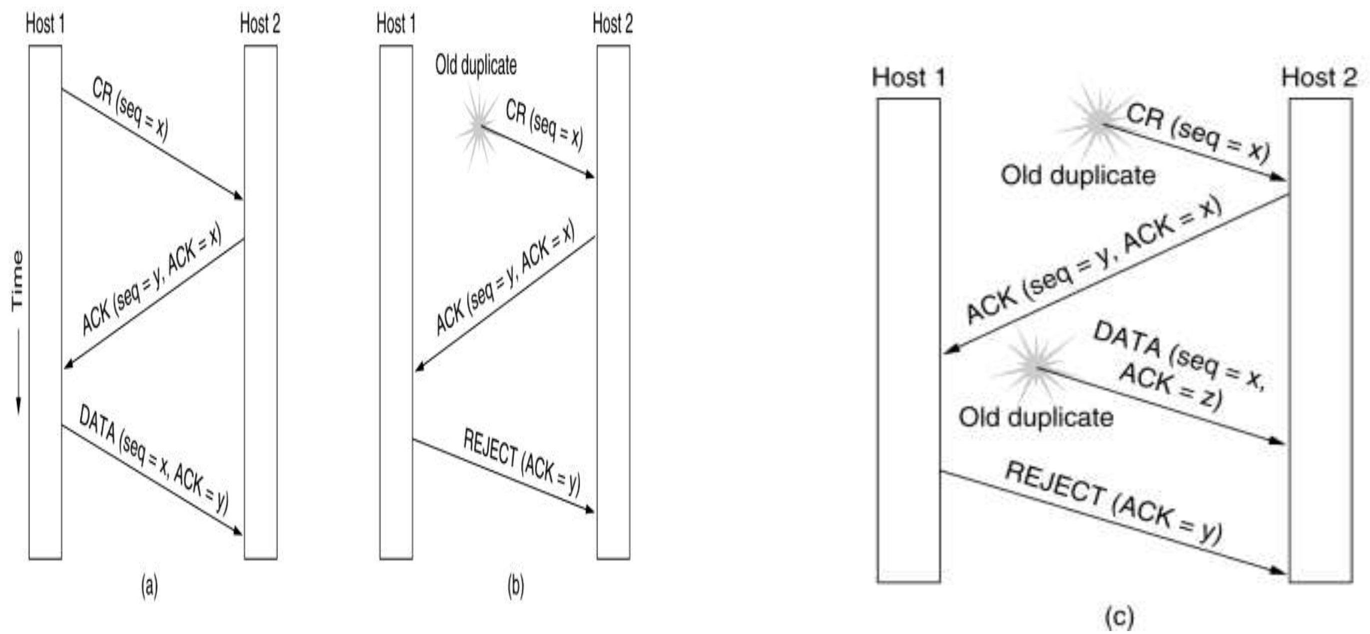
No combination of delayed packets can cause the protocol to fail

- Tomlinson – three-way-handshake



ETP: ESTABLISHING A CONNECTION

- Tomlinson – three-way-handshake

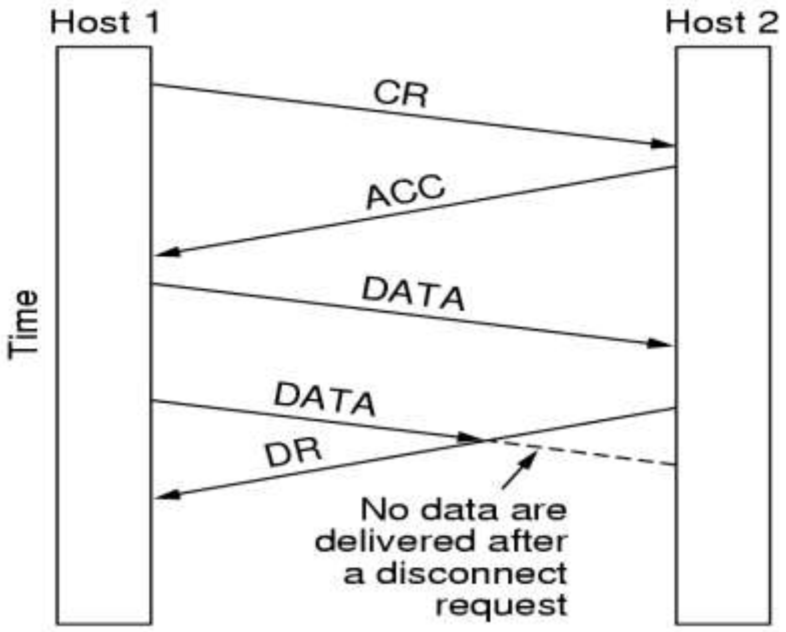


ETP: RELEASING A CONNECTION

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

ETP: RELEASING A CONNECTION

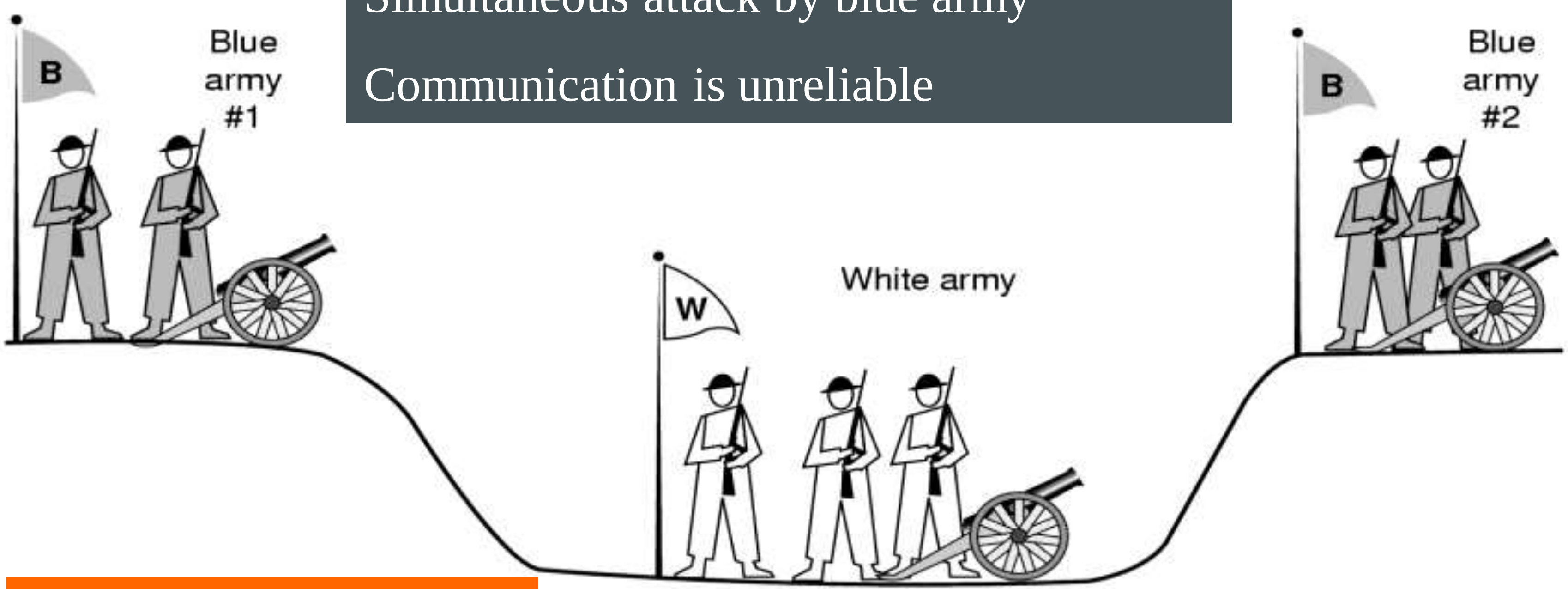
- Asymmetric: data loss



ETP: RELEASING A CONNECTION

Simultaneous attack by blue army

Communication is unreliable

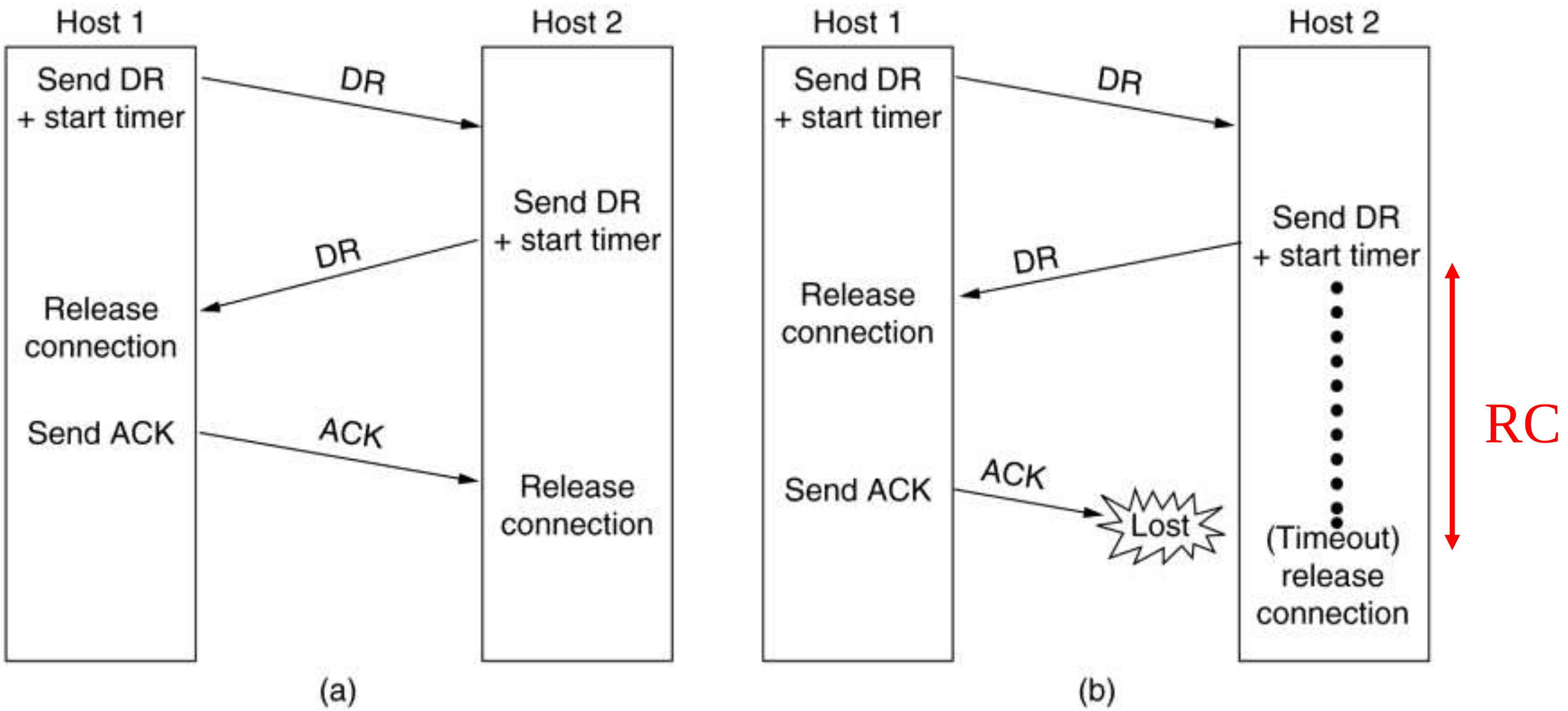


No protocol exists!!

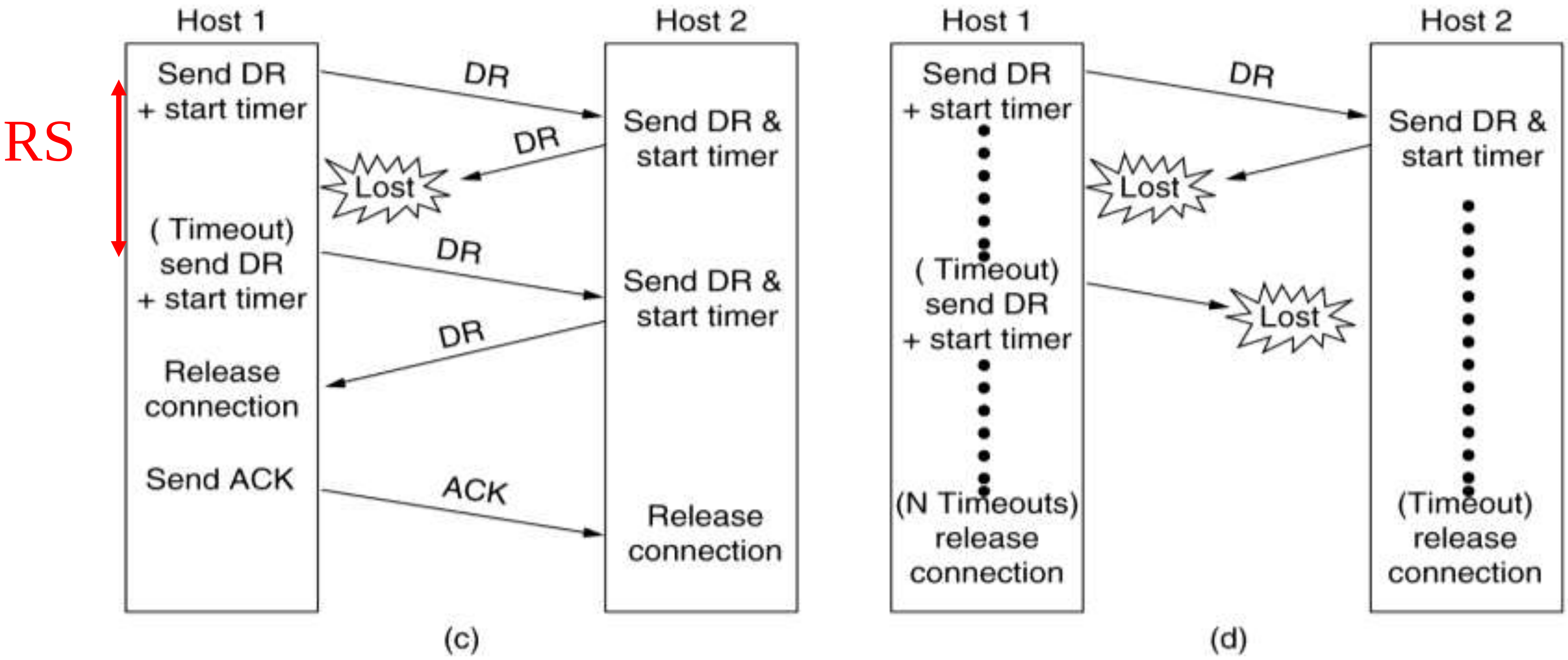
ETP: RELEASING A CONNECTION

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

ETP: RELEASING A CONNECTION



ETP: RELEASING A CONNECTION

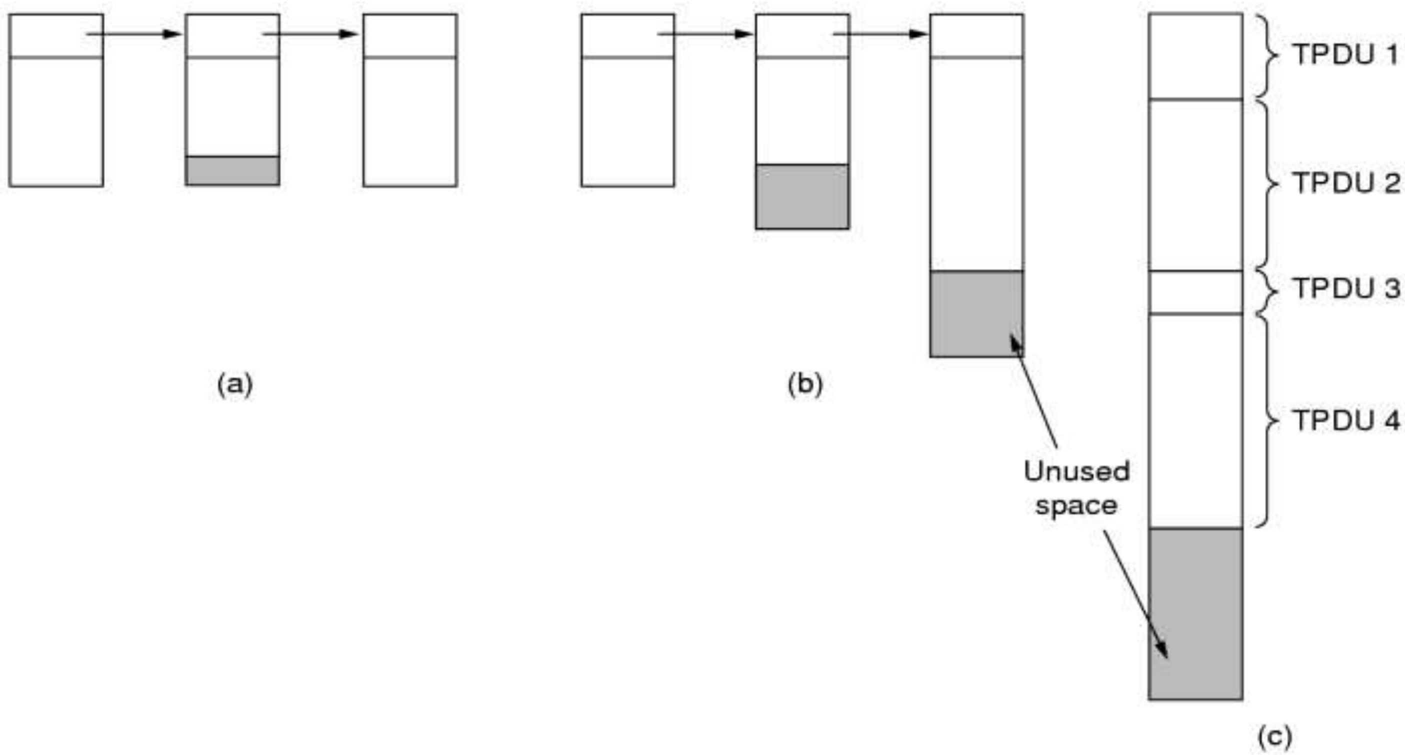


ETP: FLOW CONTROL AND BUFFERING

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

ETP: FLOW CONTROL AND BUFFERING

- Buffer organization



ETP: FLOW CONTROL AND BUFFERING

- Buffer management: 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

ETP: FLOW CONTROL AND BUFFERING

- Where to buffer?
  - datagram network → @ sender
  - reliable network

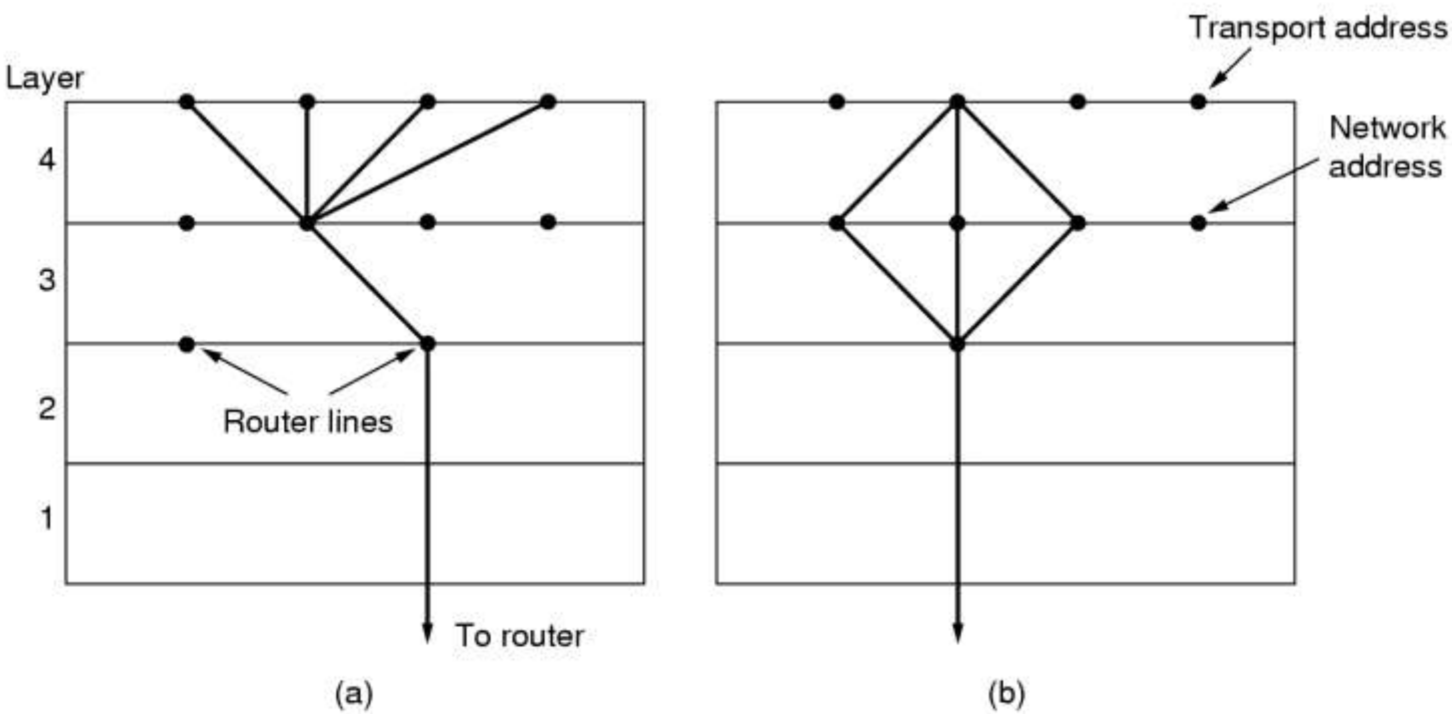
+ Receiver process guarantees free buffers?
  - No: for low-bandwidth bursty traffic
    → @ sender
  - Yes: for high-bandwidth smooth traffic
    → @ receiver

ETP: 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

ETP: MULTIPLEXING

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



ETP: 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?
  - assumptions:
    - no state saved at crashed server
    - no simultaneous events
  - NOT POSSIBLE

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

ETP: CRASH RECOVERY

- Illustration of problem: File transfer:
  - Sender: 1 bit window protocol: states S0, S1
    - packet with seq number 0 transmitted; wait for ack
  - Receiver: actions
    - Ack packet
    - Write data to disk
    - Order?

ETP: CRASH RECOVERY

- Illustration of problem: File transfer

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

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

TRANSPORT LAYER

- Services
- Elements of transport protocol
- Simple transport protocol
- UDP
- Remote Procedure Call (see Distributed Systems)
- 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

SIMPLE TRANSPORT PROTOCOL

- Transport entity: code
  - See fig 6-20, p. 514 – 517
  - 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();
  }
}
```

EXAMPLE TRANSPORT ENTITY (8)

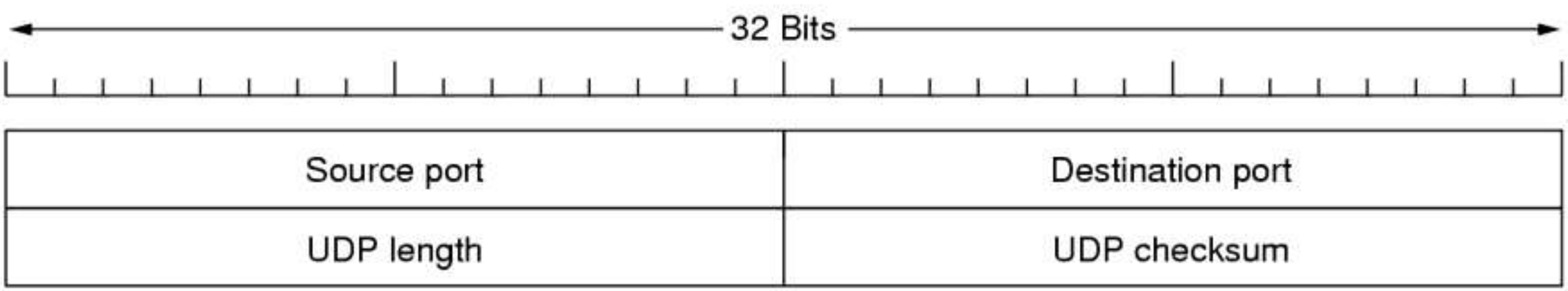
```
}

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);
             }
       }
  }
}
```

TRANSPORT LAYER

- Services
- Elements of transport protocol
- Simple transport protocol
- UDP
- Remote Procedure Call (see Distributed Systems)
- TCP

UDP

- User Data Protocol
  - Datagram service between processes
    - No connection overhead
  - UDP header:
    - Ports = identification of end points



UDP

- Some characteristics
  - Supports broadcasting, multicasting (not in TCP)
  - Packet oriented (TCP gives byte stream)
  - Simple protocol
  - Why needed above IP?

TRANSPORT LAYER

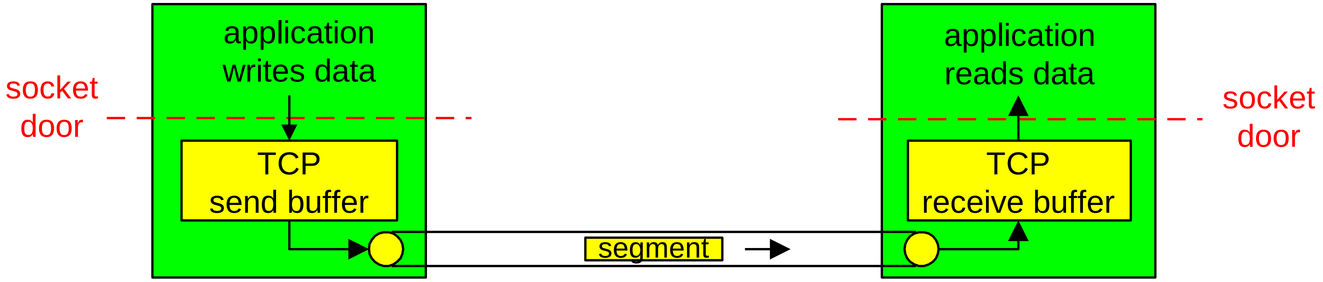
- Services
- Elements of transport protocol
- Simple transport protocol
- UDP
- Remote Procedure Call (see Distributed Systems)
- TCP

TCP SERVICE MODEL

- point-to-point
  - one sender, one receiver
- reliable, in-order byte stream
  - no *message/packet boundaries*
- pipelined & flow controlled
  - window size set by TCP congestion and flow control algorithms
- connection-oriented
  - handshaking to get at initial state
- full duplex data
  - bi-directional data flow in same connection

TCP SERVICE MODEL

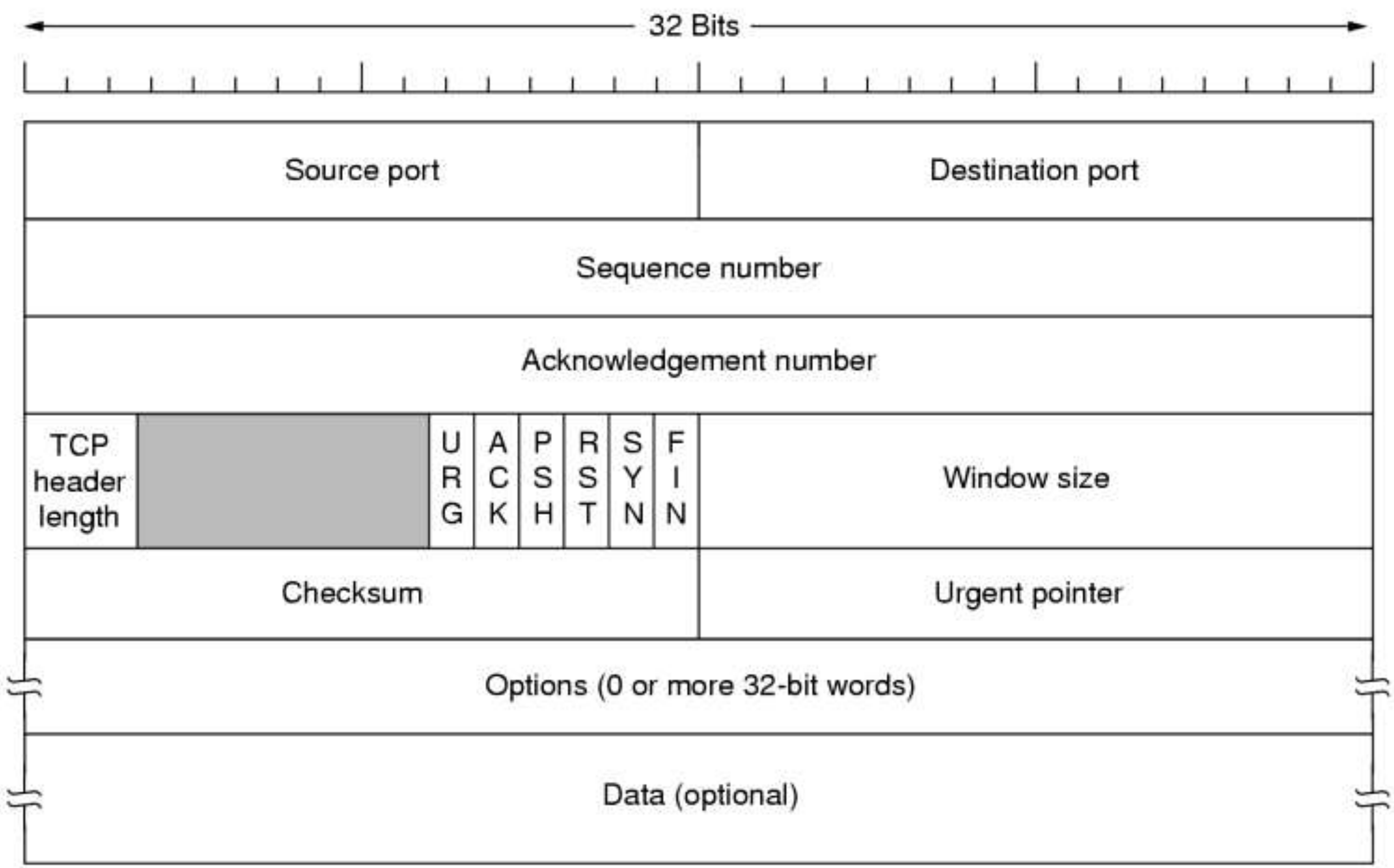
- …
- send & receive buffers



TCP PROTOCOL

- Three-way handshake to set up connections
- Every byte has its own 32-bit sequence number
  - Wrap around
  - 32-bit Acks; window size in bytes
- Segment = unit of data exchange
  - 20-byte header + options + data
  - Limits for size
    - 64Kbyte
    - MTU, agreed upon for each direction
  - Data from consecutive writes may be accumulated in a single segment
  - Fragmentation possible
- Sliding window protocol

TCP HEADER


32 Bits
Source port
Destination port
Sequence number
Acknowledgement number
TCP header length
URG
ACK
PSH
RST
SYN
FIN
Window size
Checksum
Urgent pointer
Options (0 or more 32-bit words)
Data (optional)

TCP HEADER

- source & destination ports (16 bit)
- sequence number (32 bit)
- Acknowledgement number (32 bit)
- Header length (4 bits) in 32-bit words
- 6 flags (1 bit)
- window size (16 bit): number of bytes the sender is allowed to send starting at byte acknowledged
- checksum (16 bit)
- urgent pointer (16 bit) : byte position of urgent data

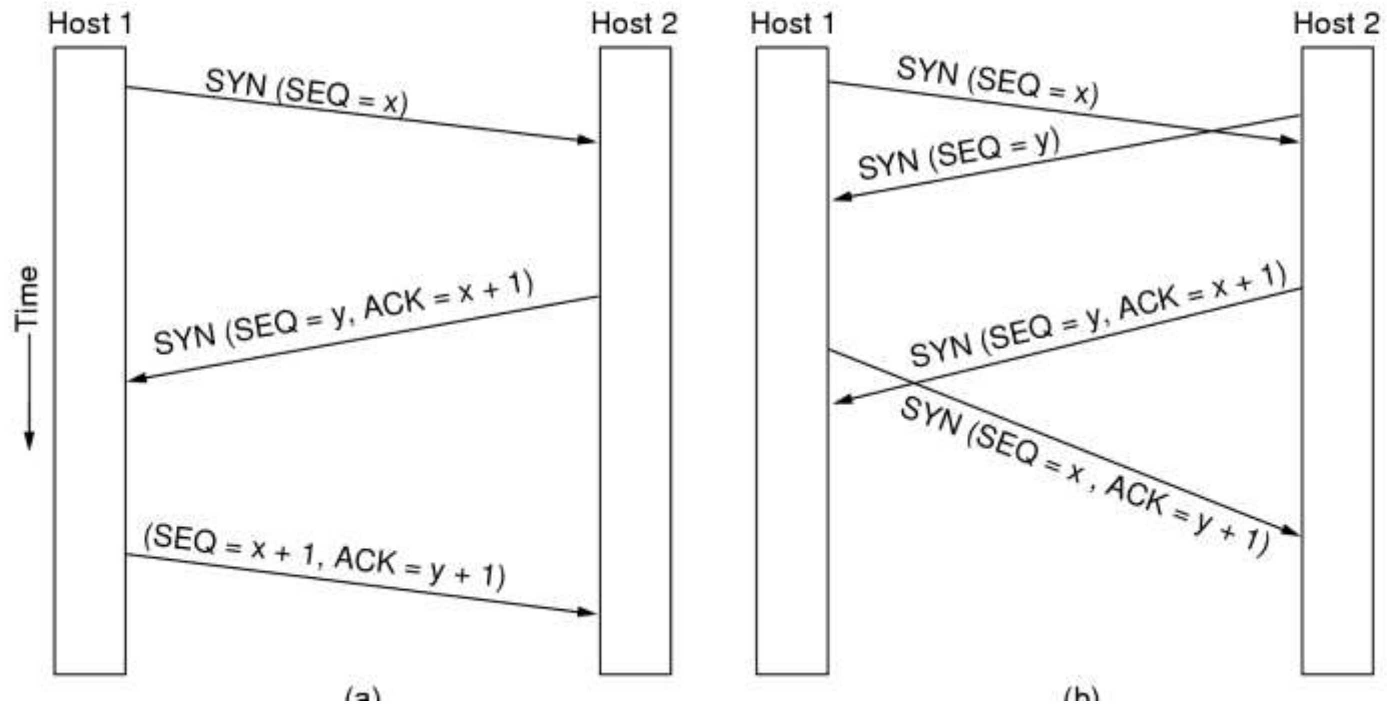
TCP HEADER

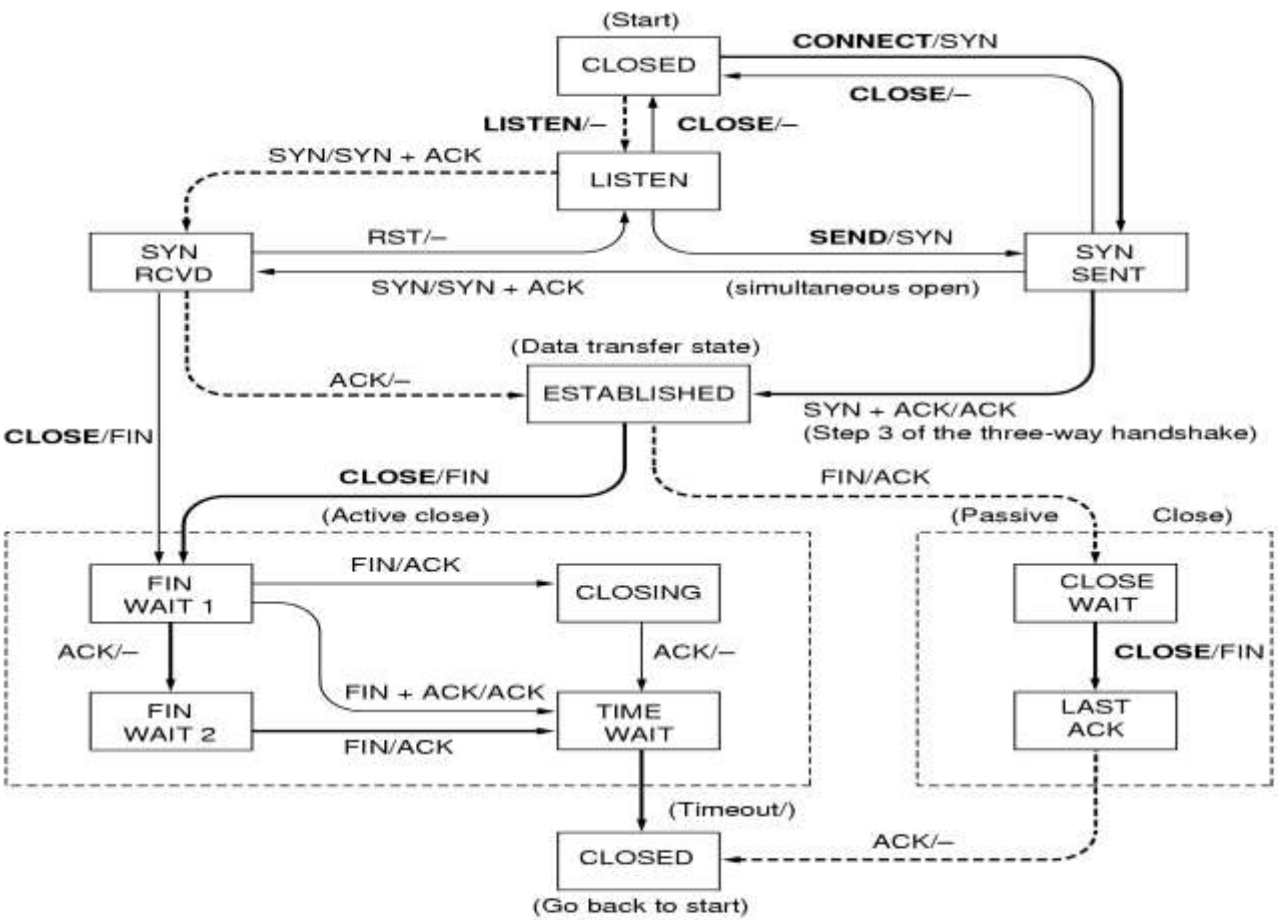
- Flags:
  - URG: urgent pointer in use
  - ACK: valid Acknowledgement number
  - PSH: receiver should deliver data without delay to user
  - RST: reset connection
  - SYN: used when establishing connections
  - FIN: used to release connection
- Options:
  - Maximum payload a host is willing to receive
  - Scale factor window size
  - Use selective repeat instead of go back n

TCP CONNECTION MANAGEMENT

- Three-way handshake
  - Initial sequence number: clock based
  - No reboot after crash for T (maximum packet lifetime=120 sec)
  - Wrap around?
- Connection identification
  - Pair of ports of end points
- Connection release
  - Both sides are closed separately
  - No response to FIN: release after 2*T
  - Both sides closed: wait for time 2 * T

TCP CONNECTION MANAGEMENT




(Start)
CLOSED
CONNECT/SYN
CLOSE/–
LISTEN/–
CLOSE/–
SYN/SYN + ACK
LISTEN
SYN RCVD
RST/–
SEND/SYN
SYN SENT
SYN/SYN + ACK
(simultaneous open)
(Data transfer state)
ACK/–
ESTABLISHED
SYN + ACK/ACK
(Step 3 of the three-way handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
(Active close)
(Passive
Close)
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
ACK/–
ACK/–
CLOSE/FIN
FIN + ACK/ACK
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
(Timeout/)
ACK/–
CLOSED
(Go back to start)

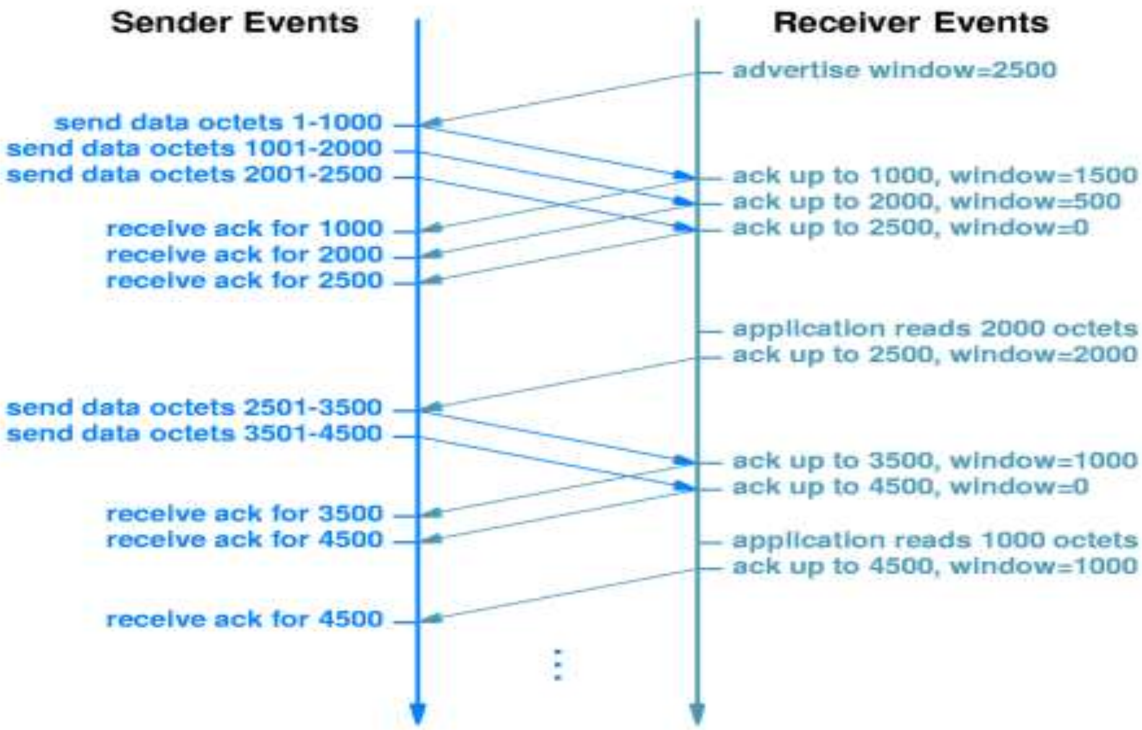
TCP CONNECTION MANAGEMENT

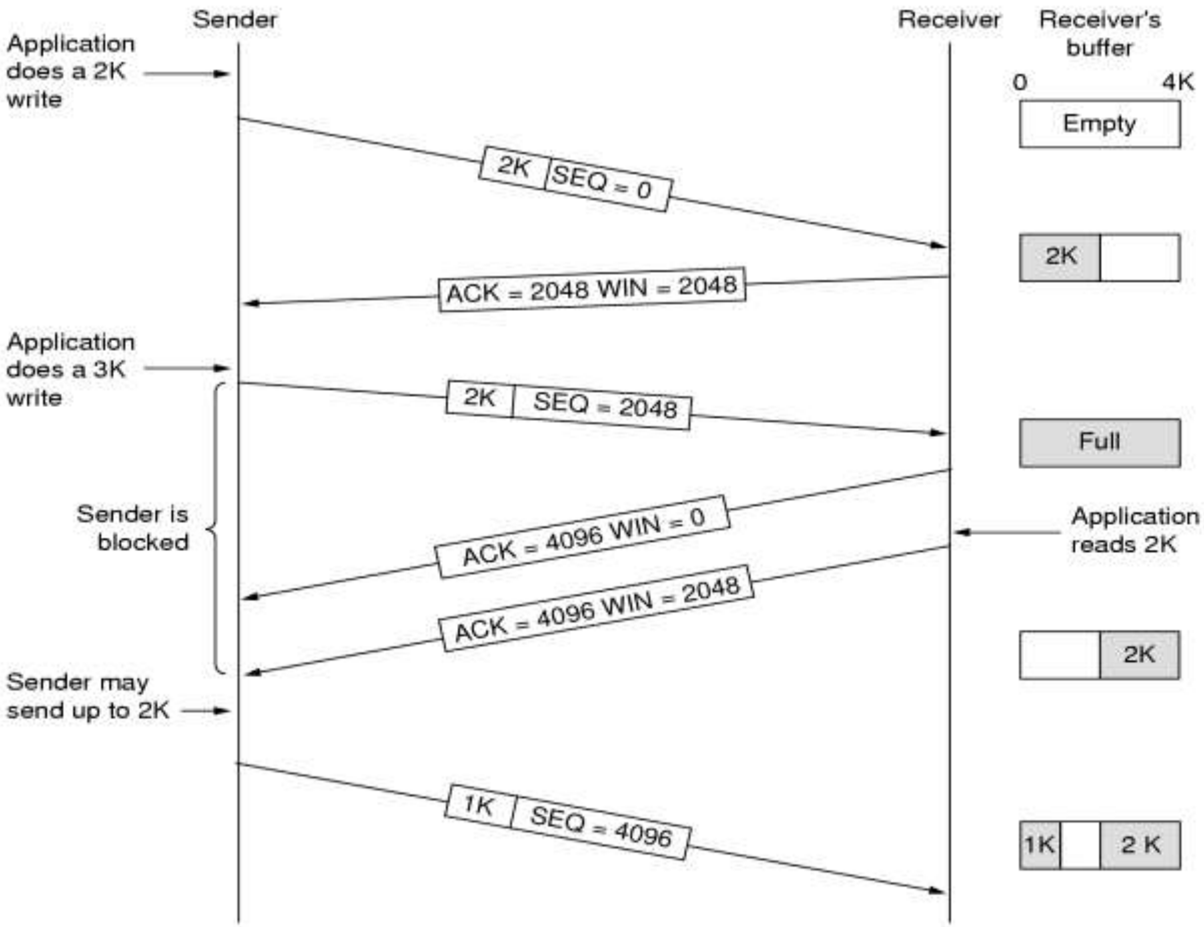
State	Description
Closed	No connection is active or pending
Listen	The server is waiting for an incoming call
SYN rcvd	A connection request has arrived; wait for ACK
SYN sent	The application has started to open a connection
Established	The normal data transfer state
FIN wait 1	The application has said it is finished
FIN wait 2	The other side has agreed to release
Timed wait	Wait for all packets to die off
Closing	Both sides have tried to close simultaneously
Close wait	The other side has initiated a release
Last Ack	Wait for all packets to die off

TCP TRANSMISSION POLICY

- Window size decoupled from Acks (ex. next slides)
- Window = 0 ➔ no packets except for
  - Urgent data
  - 1 byte segment to send Ack & window size
- Incoming user data may be buffered
  - May improve performance: less segments to send
- Ways to improve performance:
  - Delay acks and window updates for 500 msec
  - Nagle's algorithm
  - Silly window syndrome

TCP TRANSMISSION POLICY




Sender
Receiver
Receiver's buffer
0
4K
Application does a 2K write
Empty
2K
SEQ = 0
2K
ACK = 2048 WIN = 2048
Application does a 3K write
2K
SEQ = 2048
Full
Sender is blocked
ACK = 4096 WIN = 0
Application reads 2K
ACK = 4096 WIN = 2048
2K
Sender may send up to 2K
1K
SEQ = 4096
1K
2 K

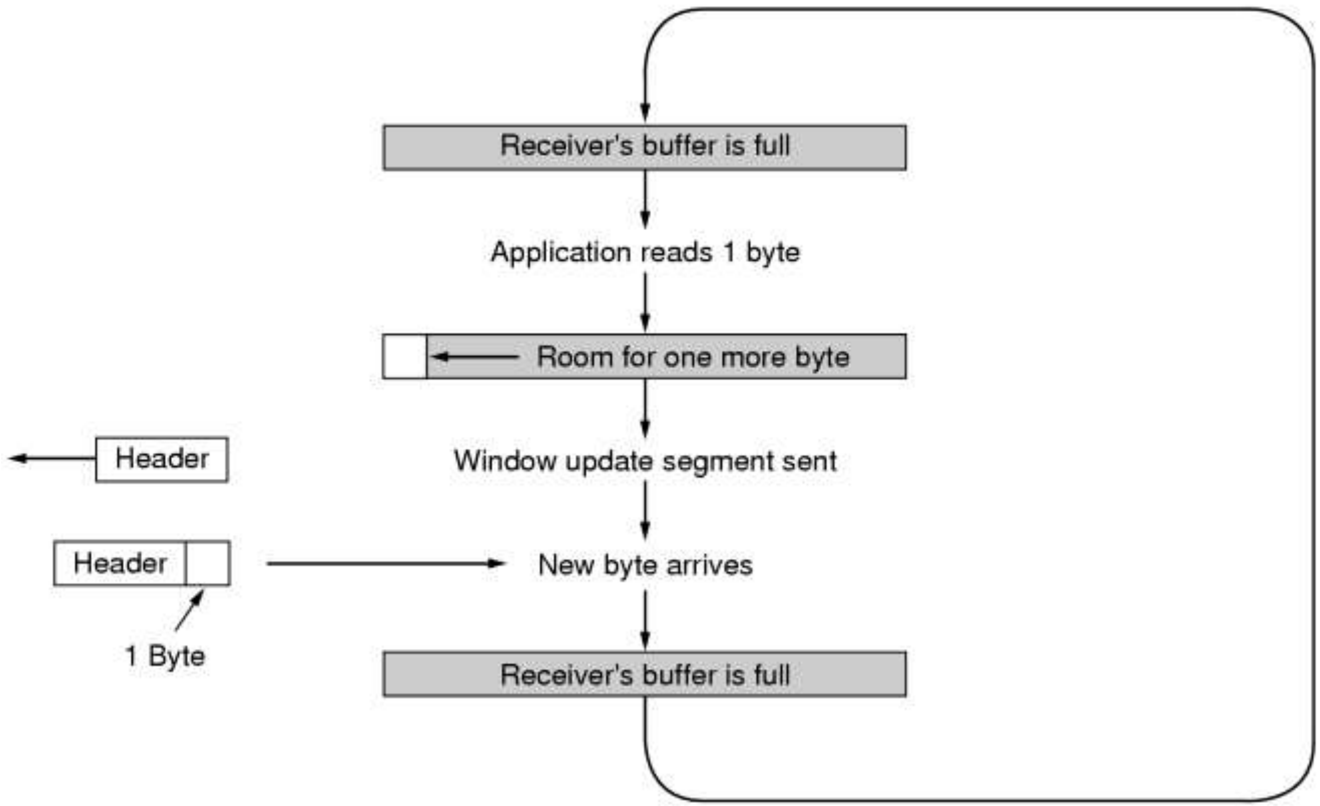
TCP TRANSMISSION POLICY

- Telnet scenario: interactive editor reacting on each keystroke: One character typed
  - ➔ 21 byte segment or 41 byte IP packet
  - 🡐 (packet received) 20 byte segment with Ack
  - 🡐 (editor has read byte) 20 byte segment with window update
  - 🡐 (editor has processed byte; sends echo) 21 byte segment
  - ➔ (client gets echo) 20 byte segment with Ack
- Solutions:
  - delay acks + window updates for 500 msec
  - Nagle's algorithm:
    - Receive one byte from user; send it in segment
    - Buffer all other chars till Ack for first char arrives
    - Send other chars in a single segment
    - Disable algorithm for X-windows applications (do not delay sending of mouse movements)

TCP TRANSMISSION POLICY

- Silly window syndrome
  - Problem:
    - Sender transmits data in large blocks
    - Receiver reads data 1 byte at a time
  - Scenario: next slide
  - Solution:
    - Do not send window update for 1 byte
    - Wait for window update till
      - Receiver can accept MTU
      - Buffer is half empty

TCP TRANSMISSION POLICY



TCP TRANSMISSION POLICY

- General approach:
  - Sender should not send small segments
    - Nagle: buffer data in TCP send buffer
  - Receiver should not ask for small segments
    - Silly window: do window updates in large units

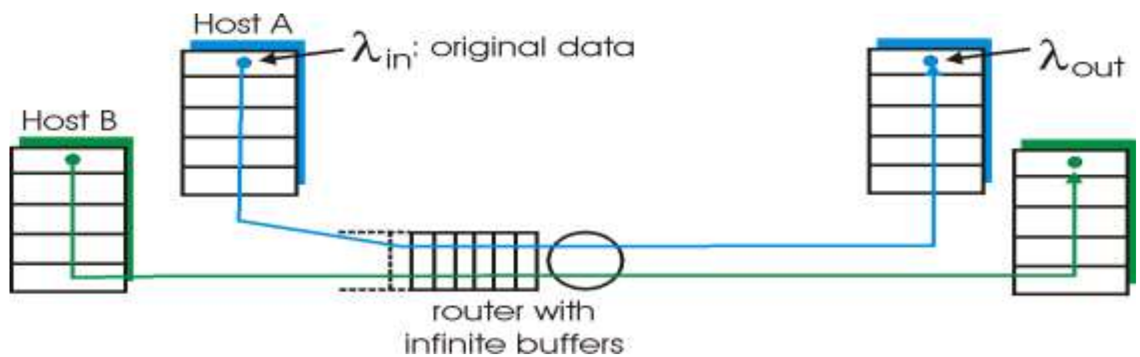
PRINCIPLES OF CONGESTION CONTROL

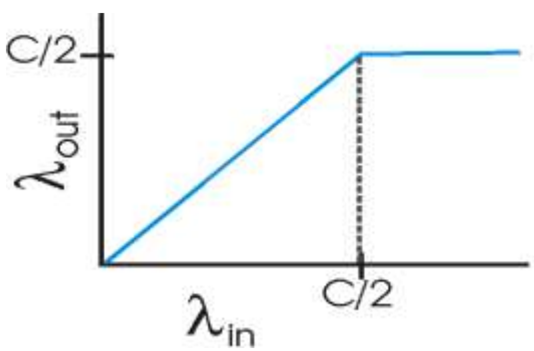
Congestion:

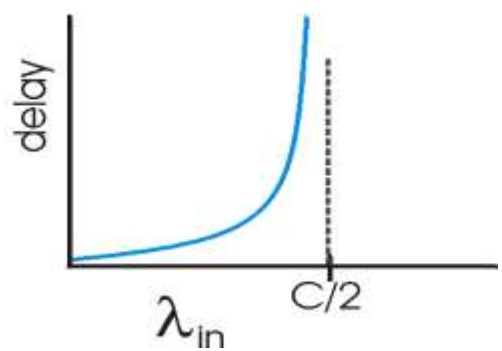
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
  = end-to-end issue!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queue-ing in router buffers)
- a top-10 problem!

CAUSES/COSTS OF CONGESTION: SCENARIO

- two senders, two receivers
- one router, infinite buffers
- no retransmission







- large delays when congested
- maximum achievable throughput

APPROACHES TOWARDS CONGESTION CONTROL

Two broad approaches towards congestion control:

- end-to-end congestion control:
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

- Network-assisted congestion control:
- routers provide feedback to end systems
  - single bit indicating congestion (SNA, ATM)
  - explicit rate sender should send at

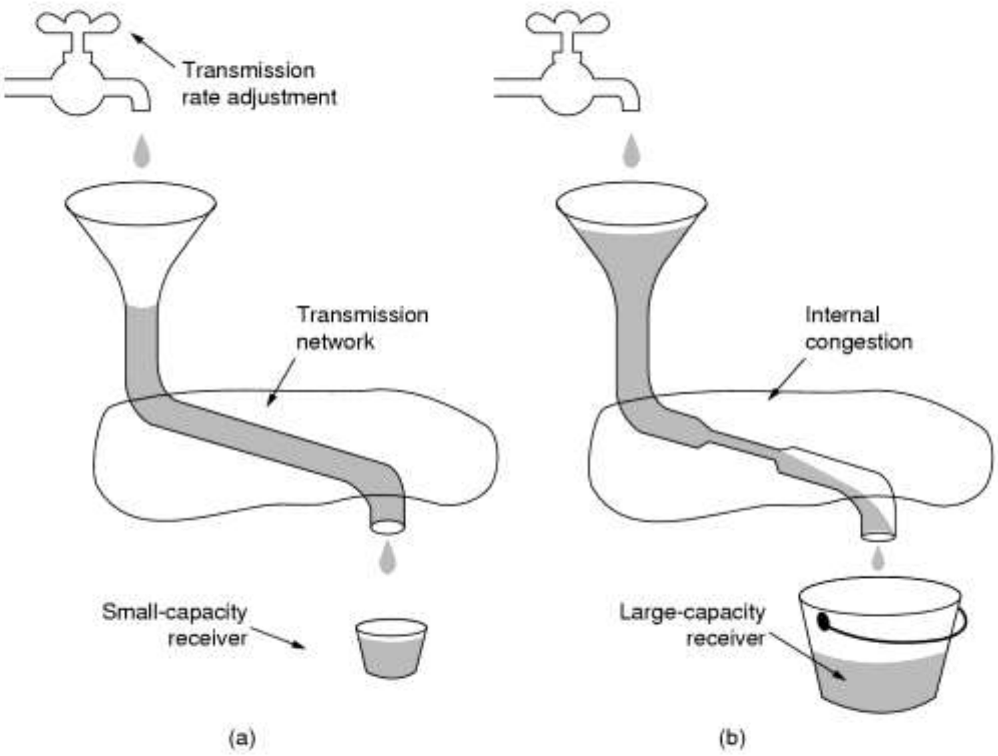
TCP CONGESTION CONTROL

: Rare for wired networks

- How to detect congestion? Packet loss
- Timeout caused by packet loss: reasons
  - Transmission errors
  - Packed discarded at congested router

Hydraulic example illustrating two limitations for sender!

TCP CONGESTION CONTROL



TCP CONGESTION CONTROL

: Rare

- How to detect congestion? Packet loss ➔ congestion
- Timeout caused by packet loss: reasons
  - Transmission errors
  - Packed discarded at congested router

Approach: 2 windows for sender

Minimum of {
- Receiver window
- Congestion window

TCP CONGESTION CONTROL

- end-end control (no network assistance)
- transmission rate limited by congestion window size, **`Congwin`**, over segments:

- **w segments, each with MSS bytes sent in one RTT:**

$$\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

TCP CONGESTION CONTROL:

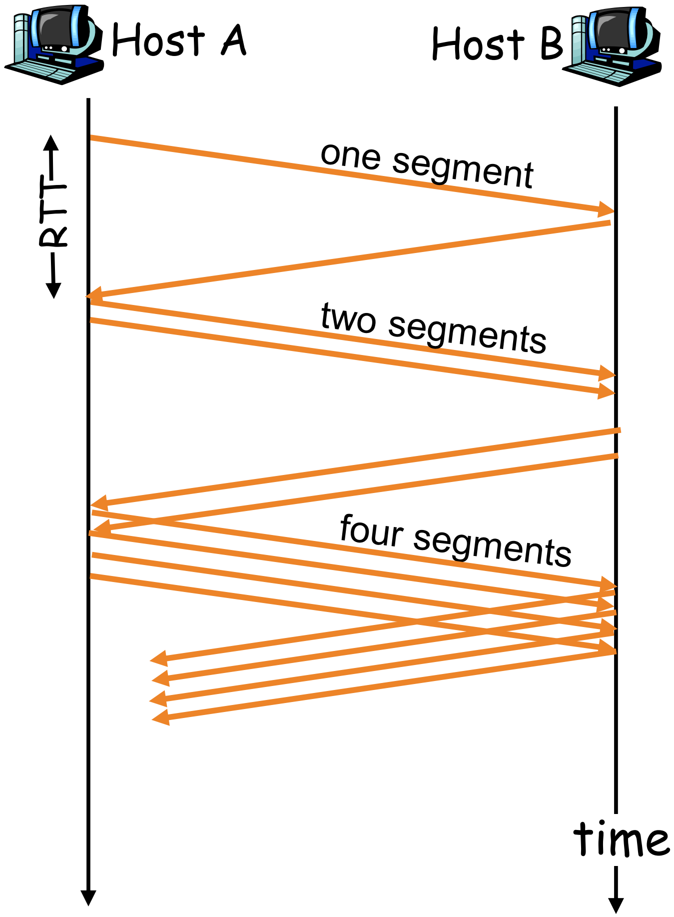
- "probing" for usable bandwidth:
  - ideally: transmit as fast as possible (**`Congwin`** as large as possible) without loss
  - *increase* **`Congwin`** until loss (congestion)
  - loss: *decrease* **`Congwin`**, then begin probing (increasing) again

- two "phases"
  - slow start
  - congestion avoidance
- important variables:
  - **`Congwin`**
  - **`threshold:`** defines threshold between two phases:
    - slow start phase
    - congestion control phase

TCP SLOW START

Slow start algorithm

initialize: Congwin = 1
for (each segment ACKed)
    Congwin++
until (loss event OR
    CongWin > threshold)



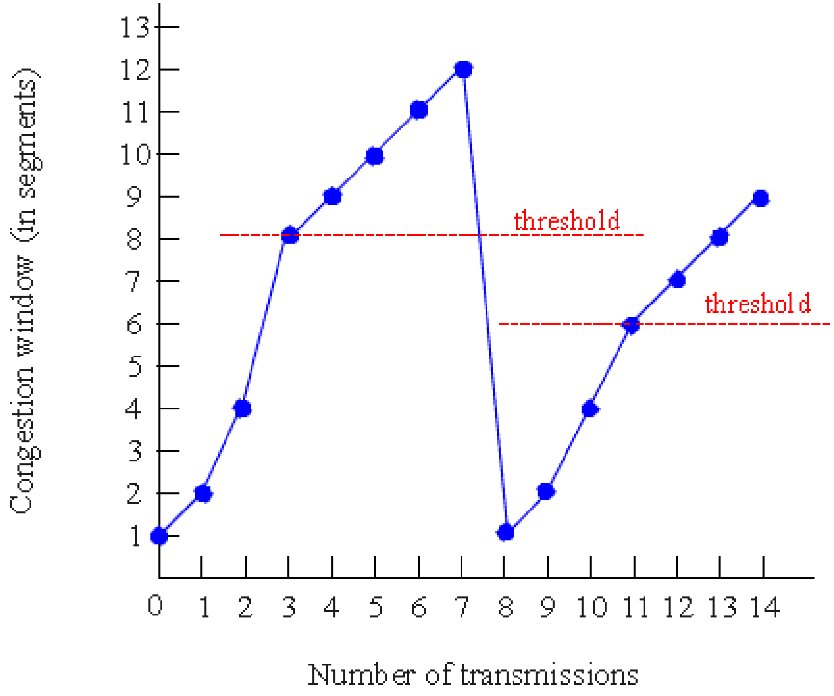
- exponential increase (per RTT) in window size (not so slow!)
- loss event: timeout (Tahoe TCP) and/or three duplicate ACKs (Reno TCP)

TCP CONGESTION AVOIDANCE

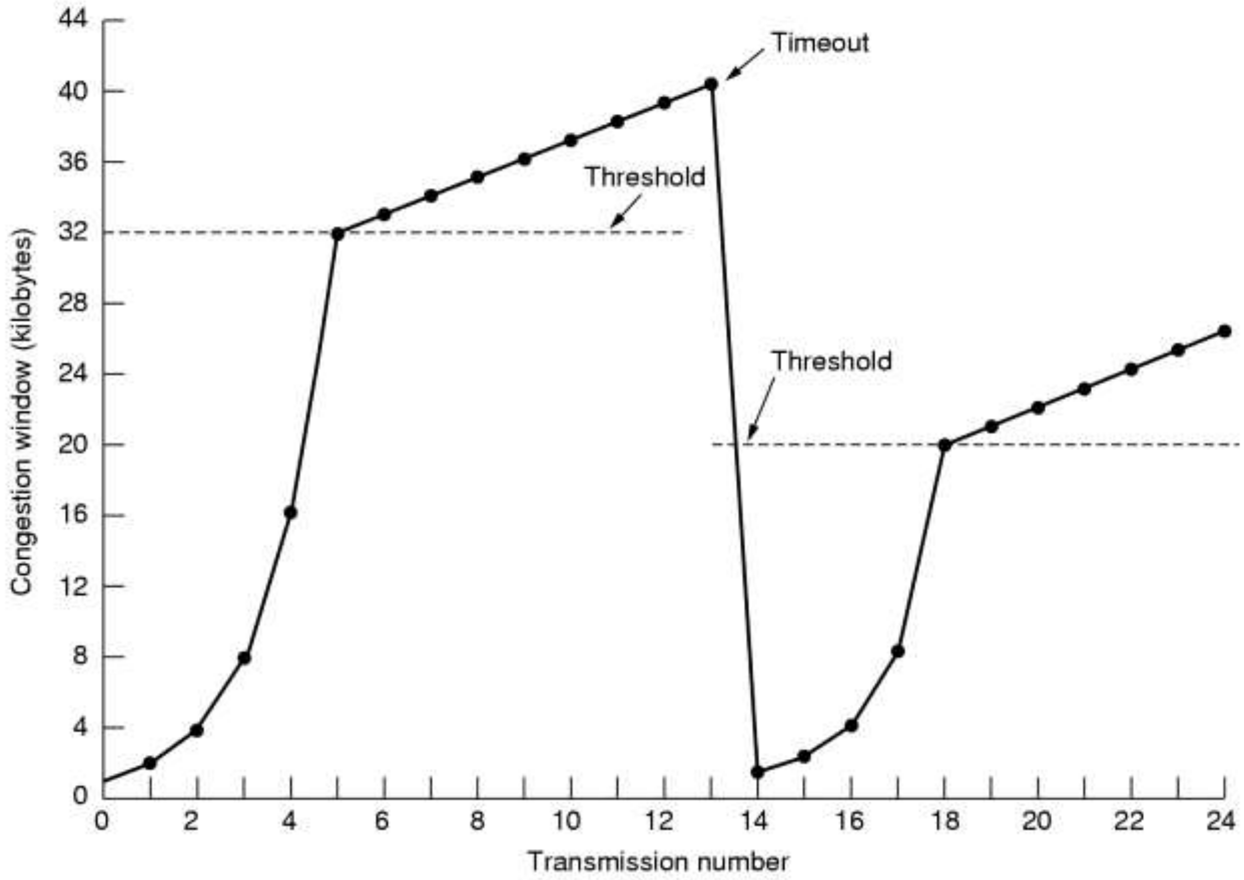
Congestion avoidance

```
/* slowstart is over         */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
      Congwin++
  }
threshold = Congwin/2
Congwin = 1
perform slowstart
```
1

1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs



TCP CONGESTION CONTROL

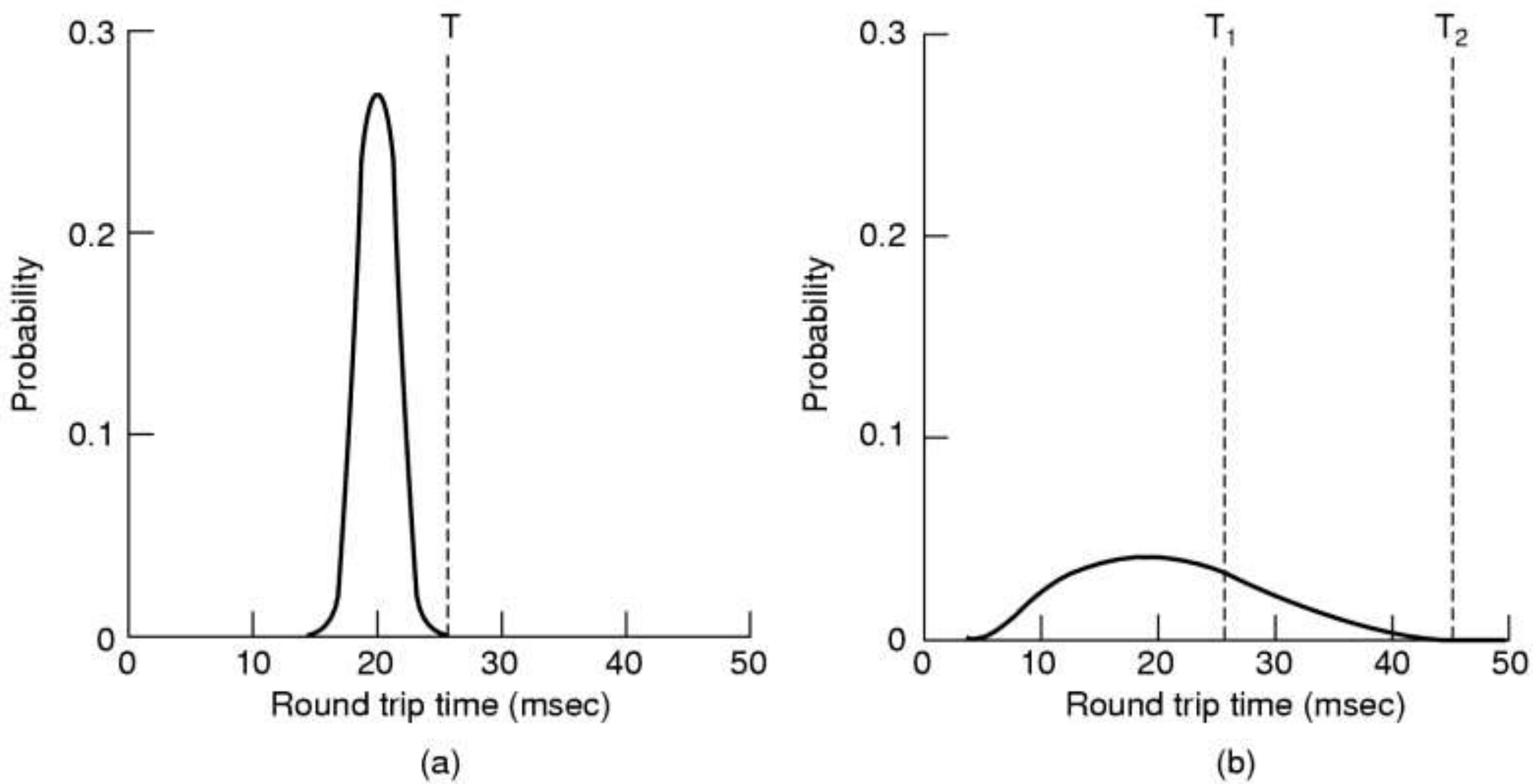


TCP TIMER MANAGEMENT

- How long should the timeout interval be?
  - Data link: expected delay predictable
  - Transport: different environment; impact of
    - Host
    - Network (routers, lines)

  unpredictable
- Consequences
  - Too small: unnecessary retransmissions
  - Too large: poor performance
- Solution: adjust timeout interval based on continuous measurements of network performance

TCP TIMER MANAGEMENT



Data link layer

Transport layer

TCP TIMER MANAGEMENT

Timeout = RTT + 4 * D

- Algorithm of Jacobson:
  - RTT = best current estimate of the round-trip time
  - D = mean deviation (cheap estimator of the standard variance)
  - 4?
    - Less than 1% of all packets come in more than 4 standard deviations late
    - Easy to compute

TCP TIMER MANAGEMENT

Timeout = RTT + 4 * D

- Algorithm of Jacobson:
  - RTT $= \alpha$ RTT + (1 $-\alpha$) M $\alpha = 7/8$

    M = last measurement of round-trip time
  - D $= \alpha$ D + (1 - α) $|$RTT - M$|$
- Karn's algorithm: how handle retransmitted segments?
  - Do not update RTT for retransmitted segments
  - Double timeout

TCP TIMER MANAGEMENT

- Other timers:
  - Persistence timer
    - Problem: lost window update packet when window is 0
    - Sender transmits probe; receivers replies with window size
  - Keep alive timer
    - Check whether other side is still alive if connection is idle for a long time
    - No response: close connection
  - Timed wait
    - Make sure all packets are died off when connection is closed
    - = 2 T

WIRELESS TCP & UDP

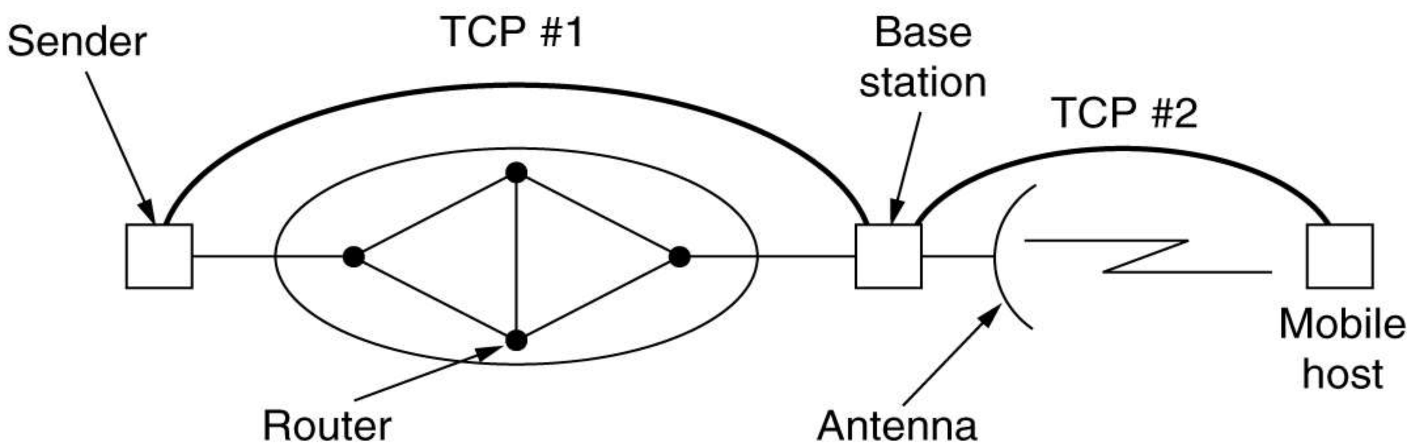
- Transport protocols
  - Independent of underlying network layer
  - BUT: carefully optimized for wired networks
  - Assumption:
    - Packet loss caused by congestion
    - Invalid for wireless networks: always loss of packets
- Congestion algorithm:
  - Timeout (= congestion) ➔ slowdown

`Wireless: Lower throughput – same loss` ➔ **NO** `solution`

- Solution for wireless networks:
  - Retransmit asap

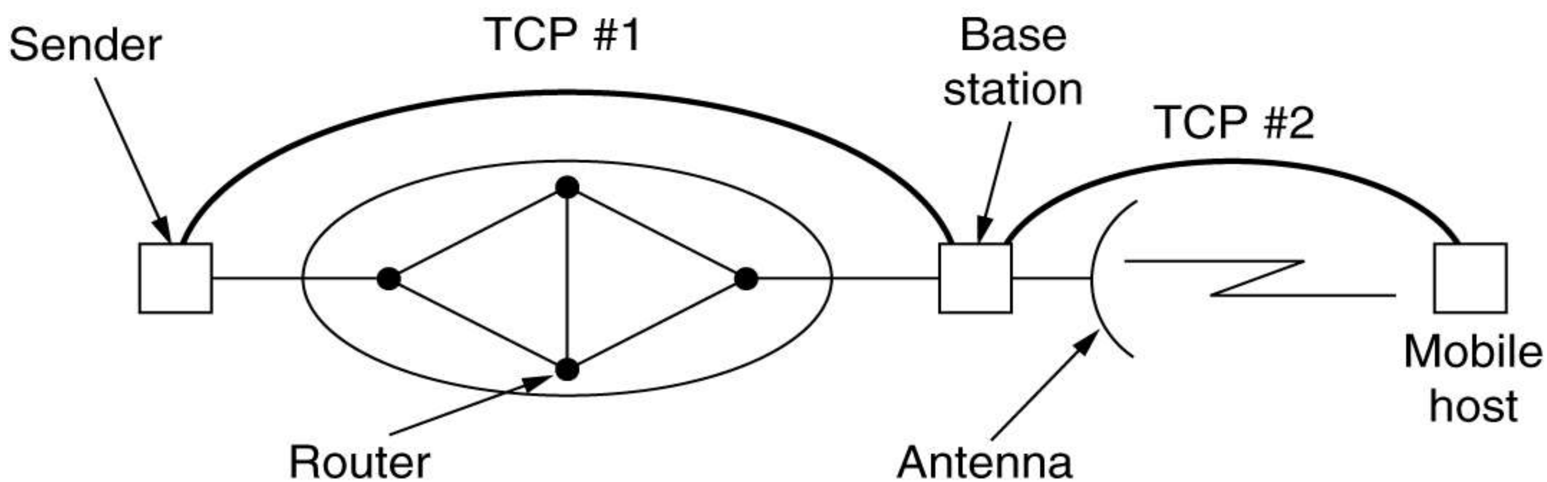
WIRELESS TCP

- Heterogeneous networks



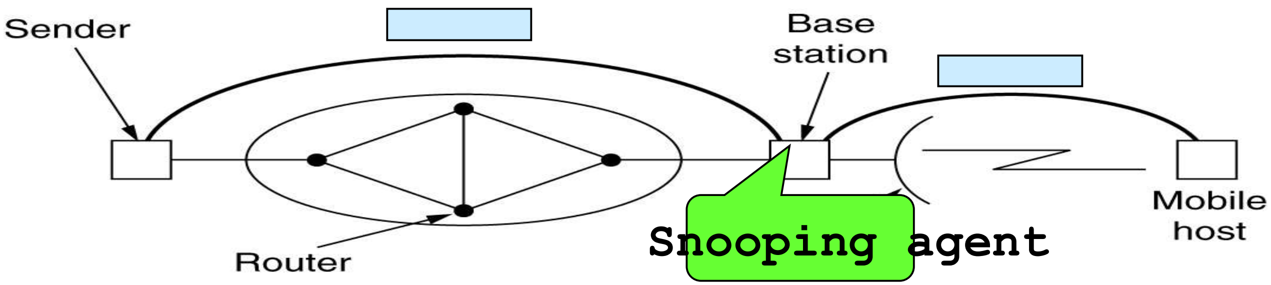
- Solutions?
  - Retransmissions can cause congestion in wired network

WIRELESS TCP



WIRELESS TCP

- Solutions for heterogeneous networks
  - Snooping agent at base station



Congestion algorithm may be invoked

- Cashes segments for mobile host
- Retransmits segment if ack is missing
- Removes duplicate acks
- Generates selective repeat requests for segments originating at mobile host

WIRELESS UDP

- UDP = datagram service → loss permitted
  no problems?

- Programs using UDP expect it to be
  highly reliable
- Wireless UDP: far from perfect!!!

→ Implications for programs recovering from lost UDP messages

TRANSACTIONAL TCP

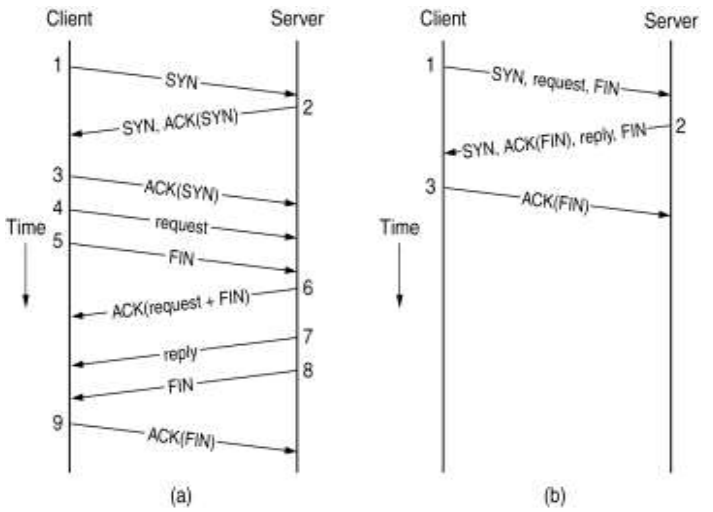
- How to implement RPC?
  - On top of UDP?
  - Yes if
    - Request and reply fit in a single packet
    - Operations are idempotent
  - Otherwise
    - Reimplementation of reliability

  - On top of TCP?

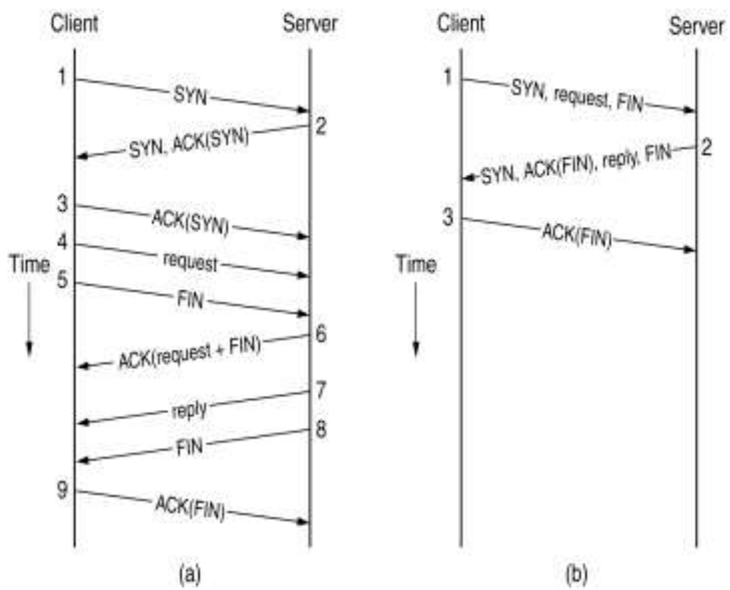
TRANSACTIONAL TCP

How to implement RPC?

- On top of UDP?
  - Yes if
    - Request and reply fit in a single packet
    - Operations are idempotent
  - Otherwise
    - Reimplementation of reliability
- On top of TCP?
  - Unattractive because of connection set up
- Solution: transactional TCP



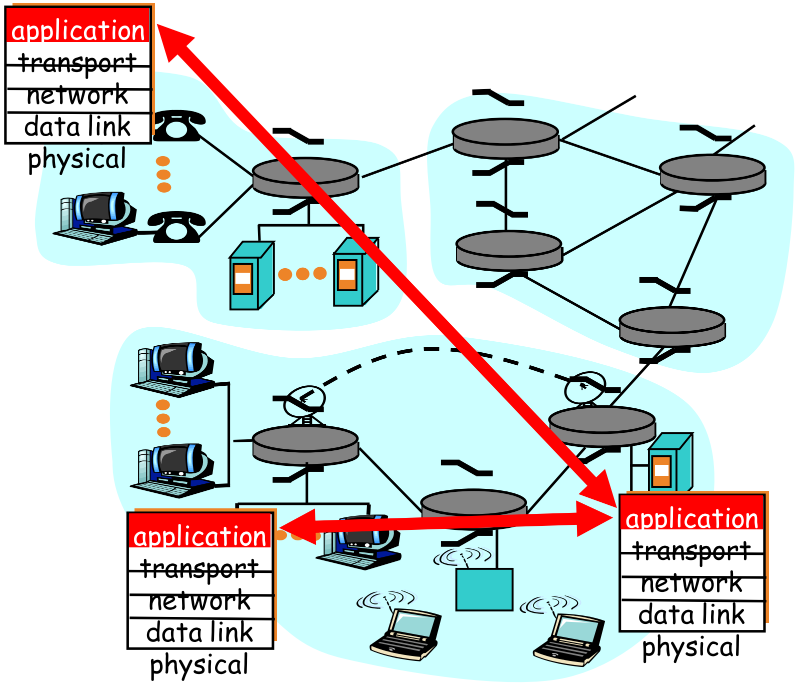
TRANSACTIONAL TCP



How to implement RPC?

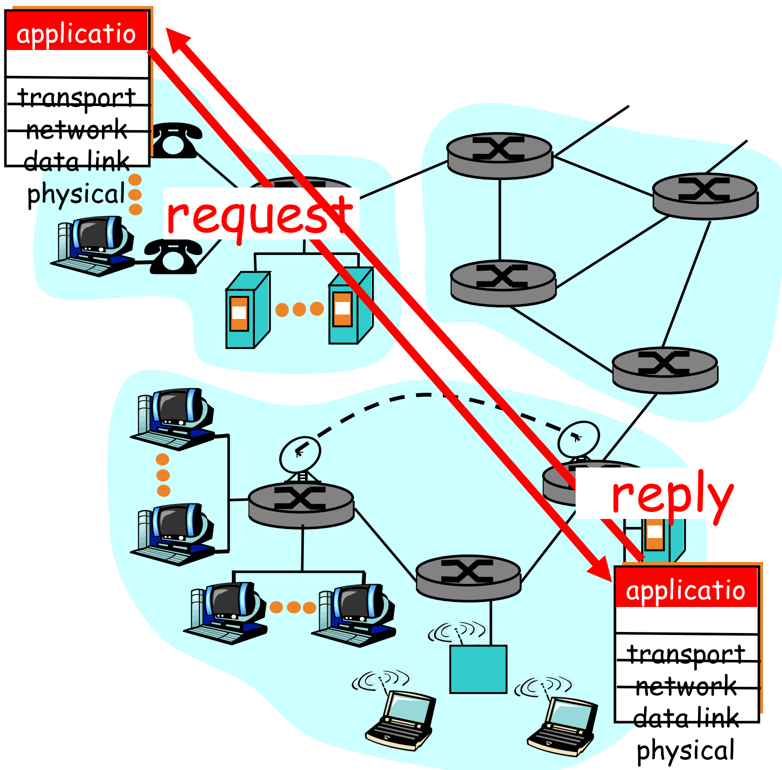
- On top of UDP?
  - Problems withreliability
- On top of TCP?
  - Overhead of connection set up
- Solution: transactional TCP
  - Allow data transfer during setup
  - Immediate close of stream

APPLICATION LAYER



- The three concepts
  - Service model
  - Protocol
  - Interface
- Network application is more than application level protocols
  - Client site
  - Server site
  - Application level protocol

CLIENT/SERVER PARADIGM



- Client
  - Initiates contact with server (speak first)
  - Typically request service from server
  - Question: identify who is/implements client in
    - Web?
    - Email?
- Server
  - Provides requested service to clients
  - Question: identify who is/implements the server counterpart in
    - Web?
    - Email?

WHICH TRANSPORT SERVICE DOES APPLICATION NEED? - PARAMETERS

- Data Loss
  - Loss-tolerant applications, e.g. audio/video
  - other app such as file transfer, telnet requires 100% reliable transmission
- Bandwidth
  - Bandwidth-sensitive applications, such as multimedia, require a maximum amount of bandwidth
  - Elastic applications: can use whatever bandwidth available
- Timing
  - Some apps such as internet telephone requires “low delay” to be effective

TRANSPORT SERVICE REQUIRED BY COMMON APPLICATIONS

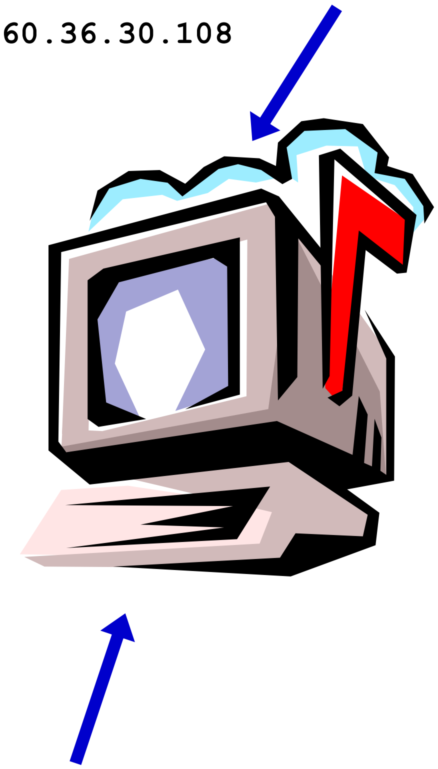
Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	loss-tolerant	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kb-1Mb	yes, 100's msec
		video:10Kb-5Mb	
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few Kbps up	yes, 100's msec
financial apps	no loss	elastic	yes and no

INTERNET APPS AND THEIR TRANSPORT LAYER PROTOCOLS

Application	Application layer protocol	Underlying transport protocol
e-mail	smtp [RFC 821]	TCP
remote terminal access	telnet [RFC 854]	TCP
Web	http [RFC 2068]	TCP
file transfer	ftp [RFC 959]	TCP
streaming multimedia	proprietary (e.g. RealNetworks)	TCP or UDP
remote file server	NFS	TCP or UDP
Internet telephony	Proprietary (private) (e.g., Vocaltec)	typically UDP

DNS – DOMAIN NAME SYSTEM

IP: 160.36.30.108

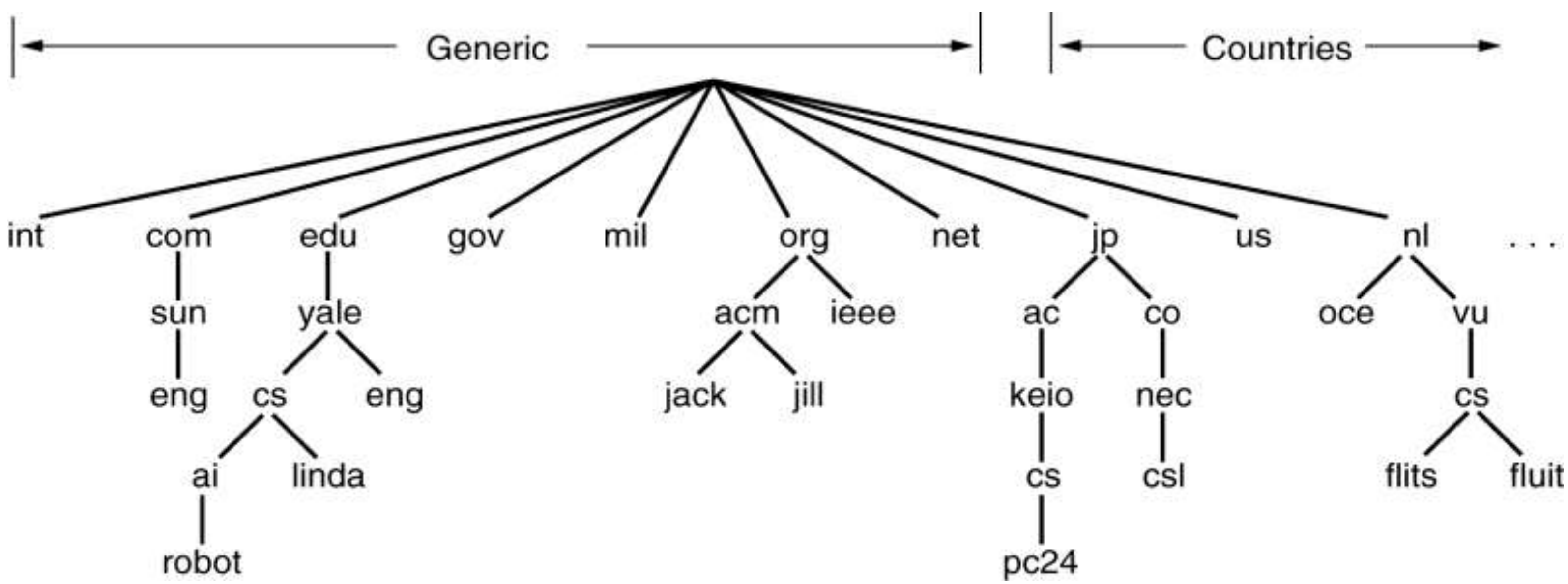


Name: panda.ece.utk.edu

DNS: MAPPING NAME TO ADDRESS

- Name: panda.ece.utk.edu is used by human
- IP address: a 32-bit numerical value used by machine
- DNS
  - A distributed database, implemented by a hierarchy of name servers
  - Application level protocols used by hosts, routers and name servers
  - Internet intelligence is on the edge

DNS NAME SPACE



biz, info, name, pro
aero, coop, museum

ICANN
http://www.icann.org/

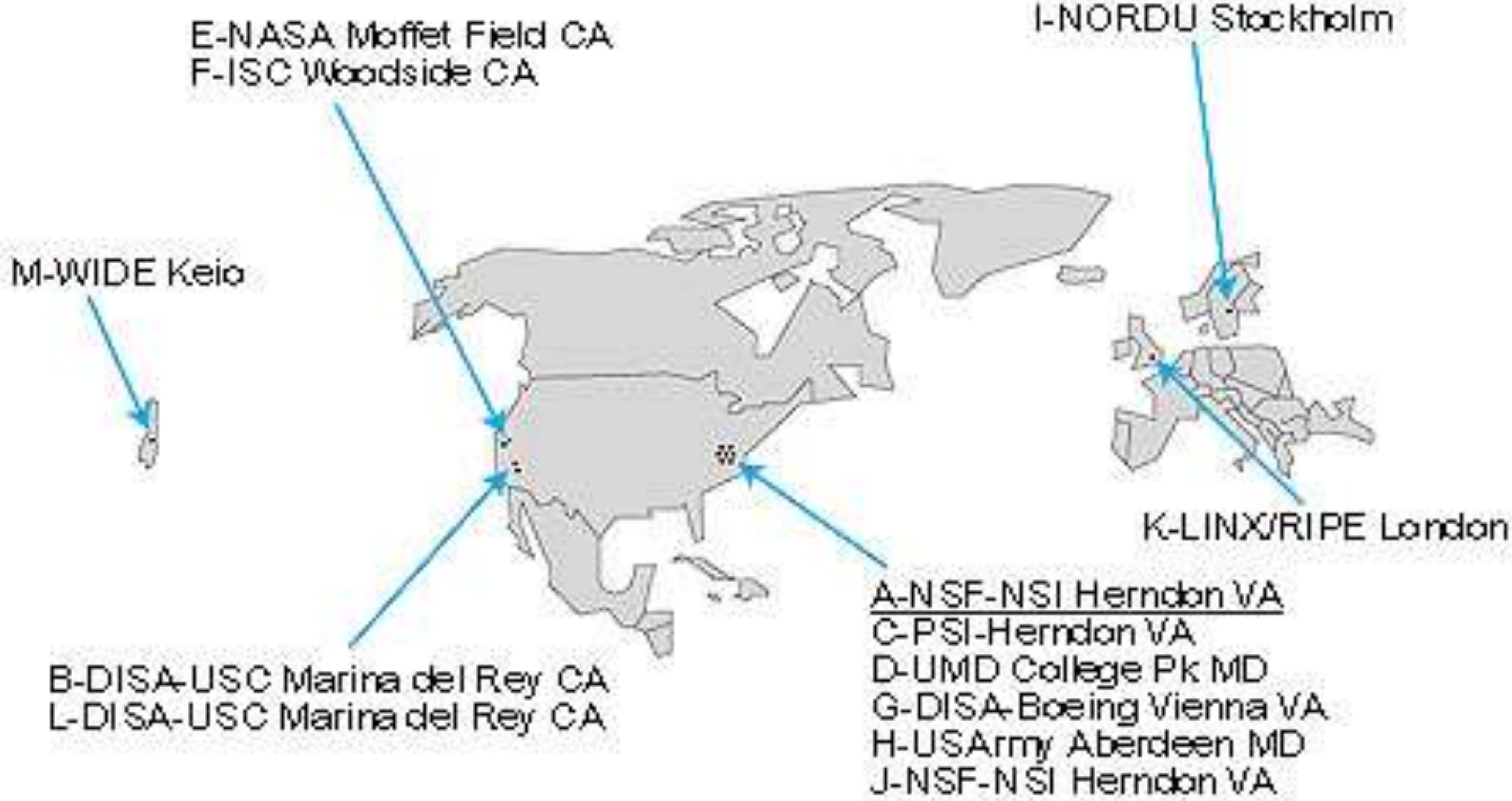
DNS – WHY NOT CENTRIC?

- Single point of failure
- Traffic volume
- Distant name server means slow response
- Scalability
- History: ARPANET begins with a single `hosts.txt`.

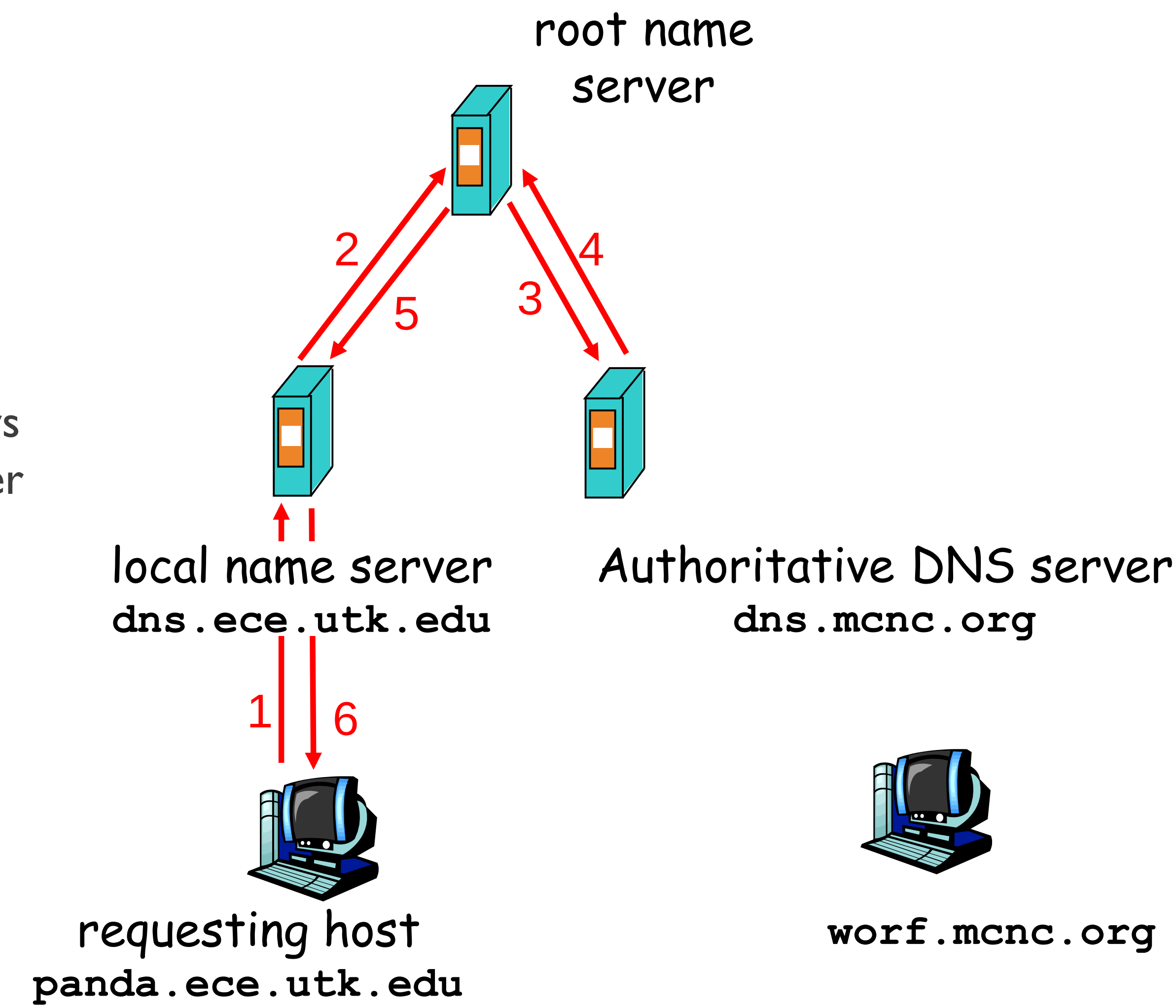
DNS – HIERARCHICAL VIEW

- Local DNS server
- Authoritative DNS server
- Root DNS server

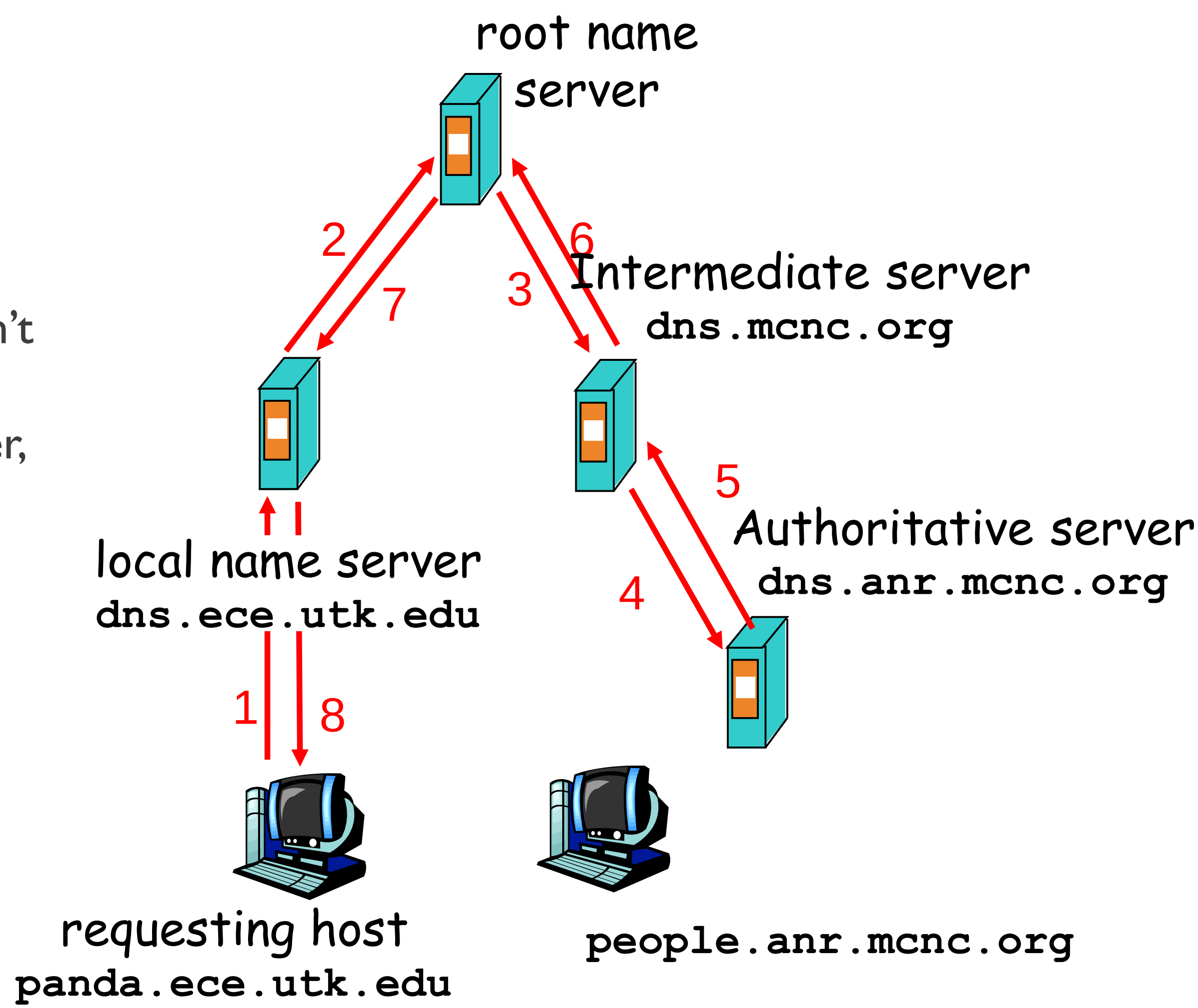
DNS: WHERE ARE ROOT SERVERS?



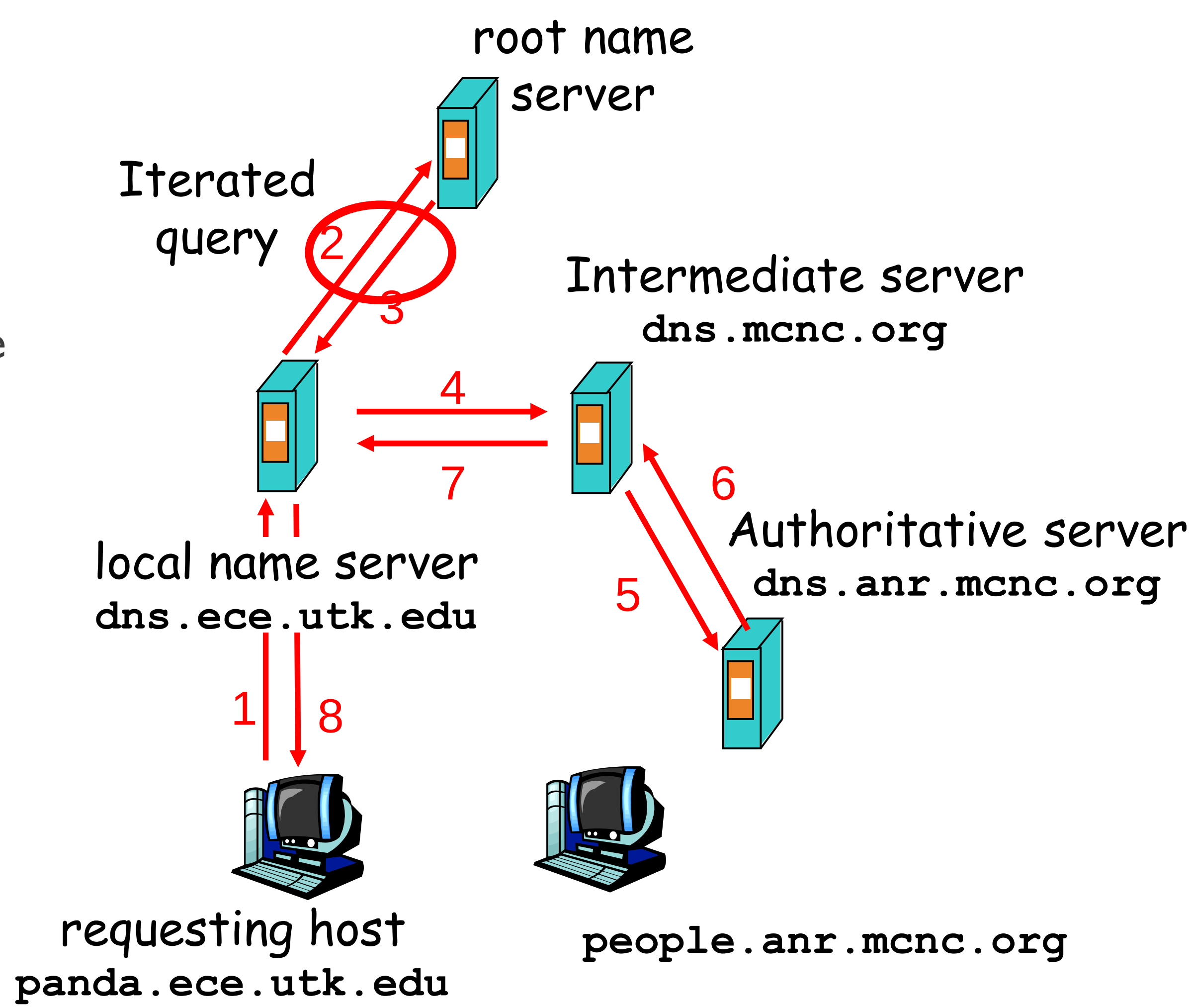
- Case: Root server knows authoritative DNS server



- Case: Root server doesn't know immediate authoritative DNS server, but know the intermediate one



- Root server replies with the name of intermediate server
- Iterated query vs. recursive query



DNS CACHING

- Once (any) server learns new mapping, it caches it
- The cache will expire after some time
- Update/notify mechanism is defined by IETF RFC 2136

DNS SERVER AND SERVICE

- Running on top of UDP
- Port number: 53
- Frequently used by other applications such as SMTP, FTP, HTTP
- Important services
  - Host aliasing
  - Mail server aliasing
  - Load distribution (DNS rotation)
- User utilities: **`dig`**, [**http://www.netliner.com/dig.html**](http://www.netliner.com/dig.html)
- More DNS information: see DNS NET `http://www.dns.net/dnsrd/docs/`

*DNS RESOURCE RECORD

DNS: distributed database storing resource records (RR)

RR format: `(domain_name, ttl, class, type, value)`

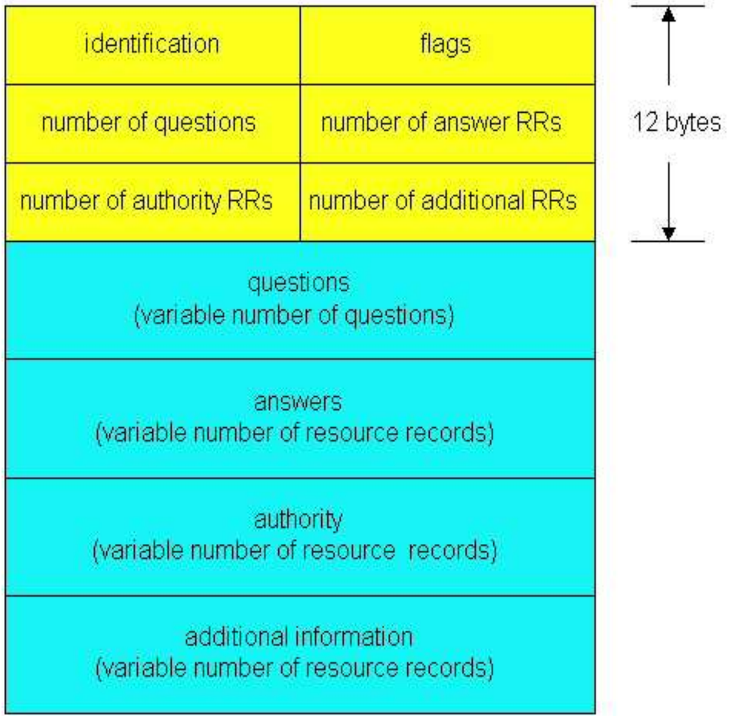
Type	Meaning	Value
SOA	Start of Authority	Parameters for this zone
A	IP address of a host	32-Bit integer
MX	Mail exchange	Priority, domain willing to accept e-mail
NS	Name Server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
HINFO	Host description	CPU and OS in ASCII
TXT	Text	Uninterpreted ASCII text

*DNS: MESSAGE FORMAT

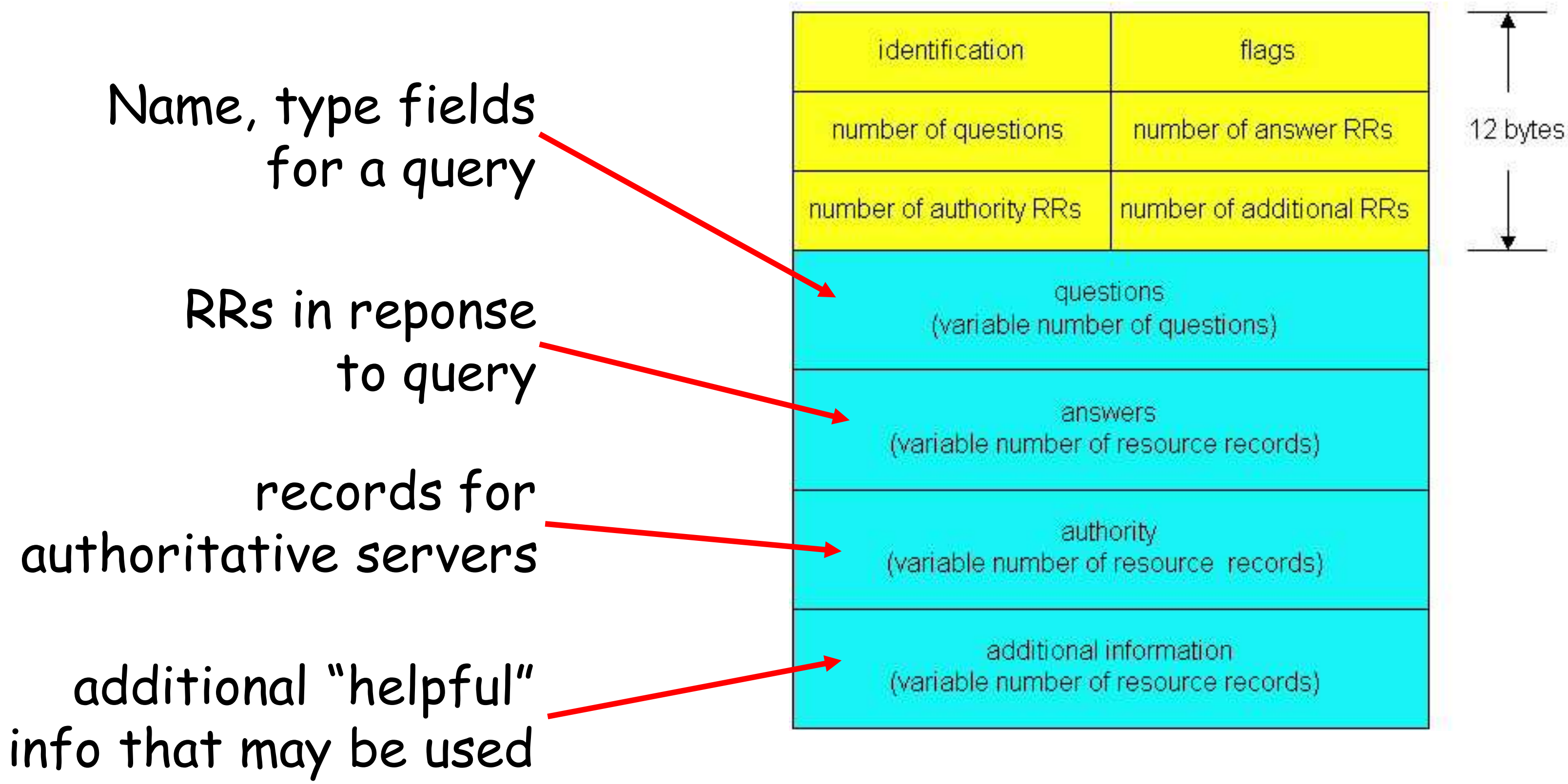
DNS protocol : ***query*** **and** ***reply*** **messages, both with same** ***message format***

message header

- identification: 16 bit # for query, reply to query uses same #
- flags:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative



*DNS PROTOCOL MESSAGE



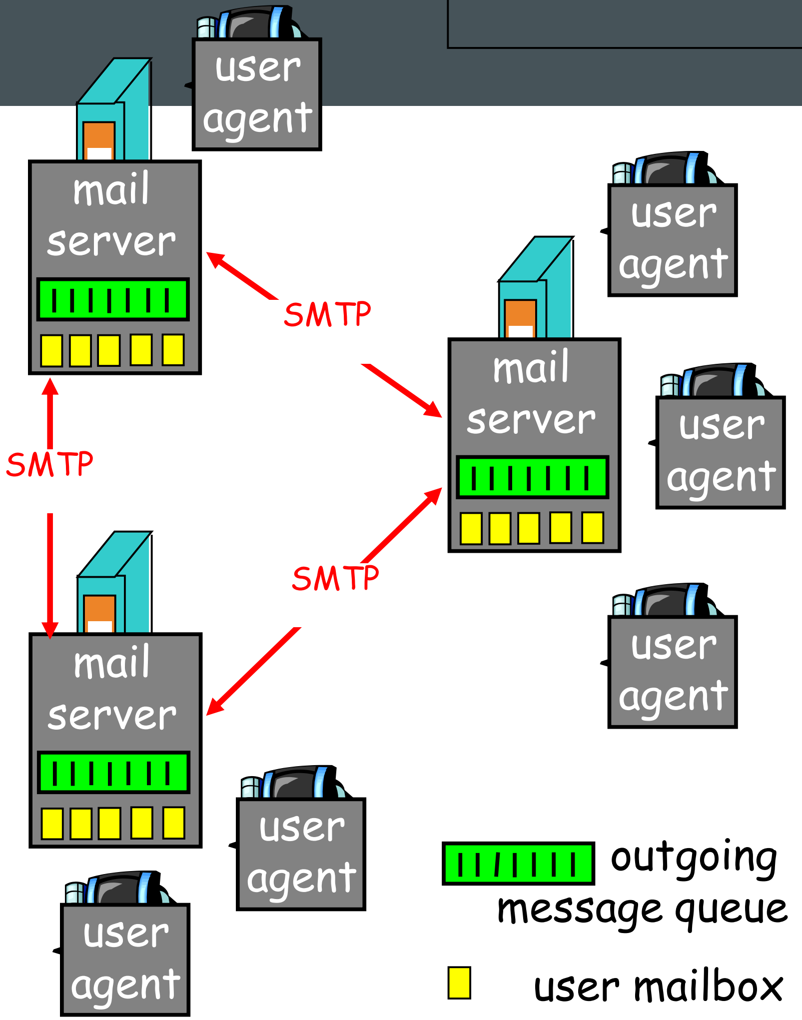
ELECTRONIC MAIL

Three major components:

- user agents
- mail servers
- simple mail transfer protocol: smtp

User Agent

- "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Netscape Messenger
- outgoing, incoming messages stored on server



ELECTRONIC MAIL: MAIL SERVER

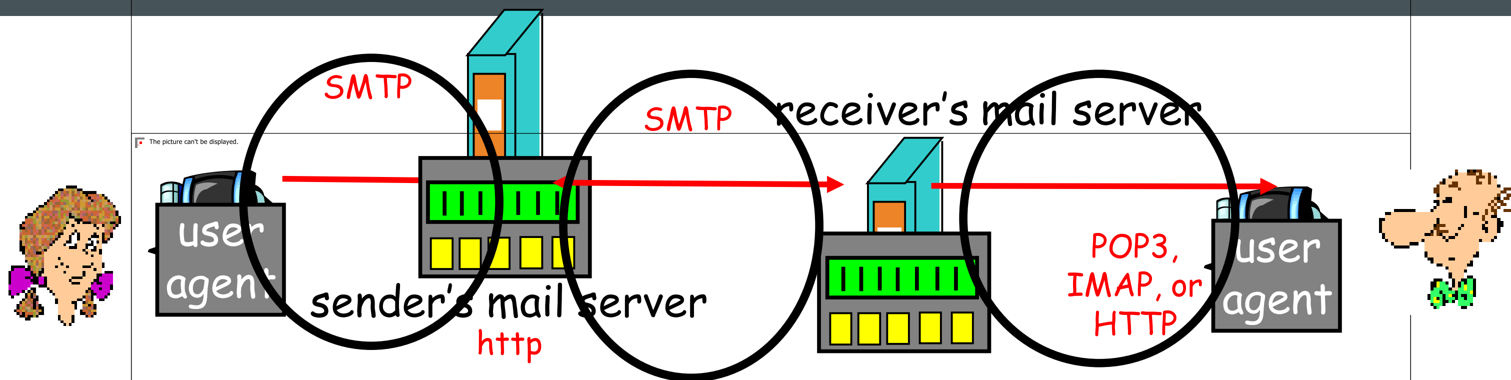
- mailbox contains incoming messages (yet to be read) for user
- message queue of outgoing (to be sent) mail messages
- Common Mail Server:
  - Sendmail
  - MS Exchange

SMTP: RFC 821

- Use TCP for reliable transfer, use port number 25
- Message must be 7-bit ASCII

```
[hqi@aicip hqi]$ telnet panda.ece.utk.edu 25
Trying 160.36.30.108...
Connected to panda.ece.utk.edu.
Escape character is '^]'.
220 panda.ece.utk.edu ESMTP Sendmail 8.11.6/8.11.6; Thu, 21 Nov 2002
09:54:04 -0500
HELO panda.ece.utk.edu
250 panda.ece.utk.edu Hello pegasus.ece.utk.edu [160.36.30.110], pleased
to meet you
MAIL FROM: <hqi@aicip.ece.utk.edu>
250 2.1.0 <hqi@aicip.ece.utk.edu>... Sender ok
RCPT TO: <hqi@panda.ece.utk.edu>
250 2.1.5 <hqi@panda.ece.utk.edu>... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
do you like ketchup?
how about pickles?
.
250 2.0.0 gALEt5U25932 Message accepted for delivery
QUIT
221 2.0.0 panda.ece.utk.edu closing connection
Connection closed by foreign host.
```

MAIL ACCESS PROTOCOL – FINAL DELIVERY



- **SMTP: delivery/storage to receiver's server**
- **Mail access protocol: retrieval from server**
  - **POP: Post Office Protocol [RFC 1939] (port 110)**
    - **authorization (agent <-->server) and download**
    - **Does not maintain state across POP sessions**
    - **Cannot manipulate emails at the server side**
  - **IMAP: Internet Mail Access Protocol [RFC 1730]**
    - **more features (more complex)**
    - **manipulation of stored msgs on server**
    - **Maintain state for the user**
  - **HTTP: Hotmail , Yahoo! Mail, etc.**
    - **Slow**

SUMMARY

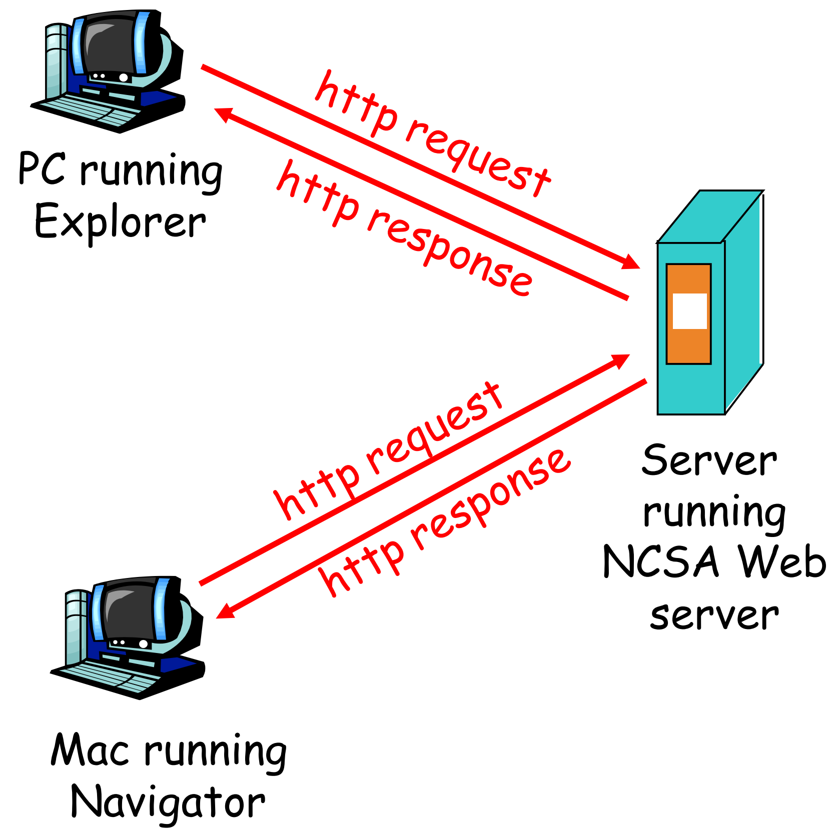
- Application
  - Client
  - Server
  - Protocol
    - What type of service
    - Through what interface
    - Which port
- DNS
  - Aliasing vs. load distribution
  - “nslookup” and “dig”
- Email
  - SMTP
  - Mail access protocol
    - POP3
    - IMAP
    - HTTP

WEB: TERMINOLOGY

- Web page
  - Consists of "objects"
  - Addressed by "url" (universal resource locator)
- Most of web page
  - One base web page
  - Several referenced "objects"
- URL has two components
  - A host name and a path
  - http://panda.ece.utk.edu/~hqi/teaching.html
- Web client
  - Netscape communicator
  - Mozilla
  - Microsoft IE browser
- Web server
  - Apache
  - Microsoft Internet Information Server (IIS)

HTTP

- Web application layer protocol: **a hyper text transfer protocol, http**
- Defined by
  - HTTP 1.0, RFC 1945
  - HTTP 1.1, RFC 2068
- Client/Server Mode
  - Client: browser asks for objects, and display it
    - Request
    - Display
  - Server: provide objects in response to requests



WEB: HTTP OPERATION FLOW

- HTTP utilizes TCP transport services
- HTTP client initiates TCP connection (create socket) to server, at port 80
- Server accepts this connection from client
- HTTP messages (defined by HTTP protocol) are exchanged between http client and http server
- TCP connection closed

- HTTP is stateless
  - Server doesn't maintain the state of past requests
    - 'back'?

HTTP EXAMPLE

Suppose user enters URL www.someSchool.edu/someDepartment/home.index

(conta ins text, references to 10 jpeg images)

1a. http client initiates TCP connection to http server (process) at www.someSchool.edu. Port 80 is default for http server.

1b. http server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

2. http client sends http *request message* (containing URL) into TCP connection socket
(plus another acknowledge message)

3. http server receives request message, forms *response message* containing requested object (someDepartment/home.index), sends message into socket

time

HTTP EXAMPLE (CONT'D)

4. **http server closes TCP connection.**

```
5. http client receives
   response message
   containing html file,
   displays html.  Parsing
   html file, finds 10
   referenced jpeg  objects
```

6. **Steps 1-5 repeated for each of 10 jpeg objects**

time

- **Two RTT (Round-trip time)**
- **Slow start**
- **Place burden on the Web server**

HTTP: PERSISTENT AND NON-PERSISTENT CONNECTION

- Non-persistent
  - HTTP 1.0
  - Server parses request, responds, then closes TCP connection
  - Each object requires 2 RTT
  - Each object suffers slow start
- Persistent
  - HTTP 1.1
  - On the same TCP connection, server parses request, responds, and parses new requests

SMTP VS. HTTP

- HTTP: Direct connection, no intermediate mail servers
- Both use persistent connection
- HTTP is a pull protocol, while SMTP is a push protocol
- SMTP: 7-bit ASCII format, message ended with a line consisting of only a period

*HTTP MESSAGE FORMAT

- two types of http messages: *request, response*
- http request message:
  - ASCII (human-readable format)

request line (GET, POST, HEAD commands)

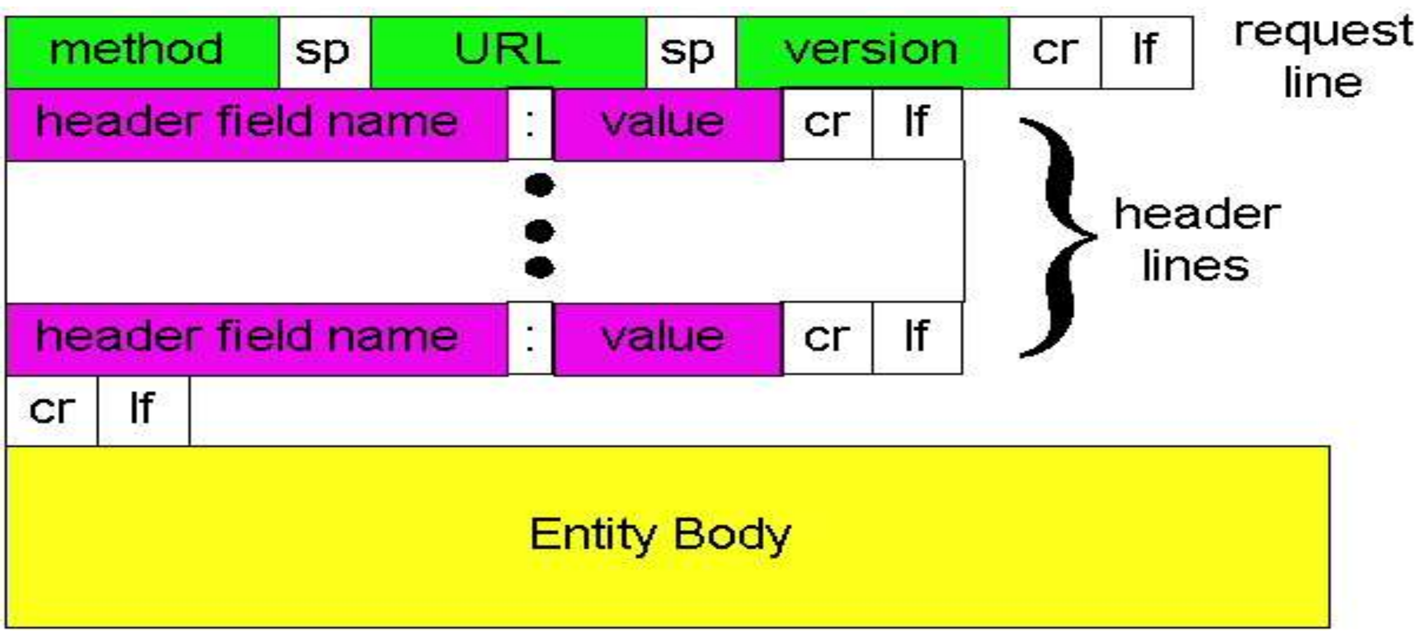
```
GET /somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html,image/gif,image/jpeg
Accept-language:fr

```

header lines

(extra carriage return, line feed)

Carriage return, line feed indicates end of message

*HTTP REQUEST: GENERAL FORMAT



*HTTP MESSAGE FORMAT: RESPONSE

status line (protocol status code status phrase)

header lines

data, e.g., requested html file

```
HTTP/1.0 200 OK
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 …...
Content-Length: 6821
Content-Type: text/html

data data data data data ...
```

*HTTP RESPONSE: STATUS CODE

`200 OK`

- request succeeded, requested object later in this message

`301 Moved Permanently`

- requested object moved, new location specified later in this message (Location:)

`400 Bad Request`

- request message not understood by server

`404 Not Found`

- requested document not found on this server

`505 HTTP Version Not Supported`

TRY OUT HTTP (CLIENT SIDE) FOR YOURSELF

- Telnet to your favorite web site

```
telnet
panda.ece.utk.edu 80
```

open TCP connection to panda port 80, anything you type be sent to panda port 80 socket

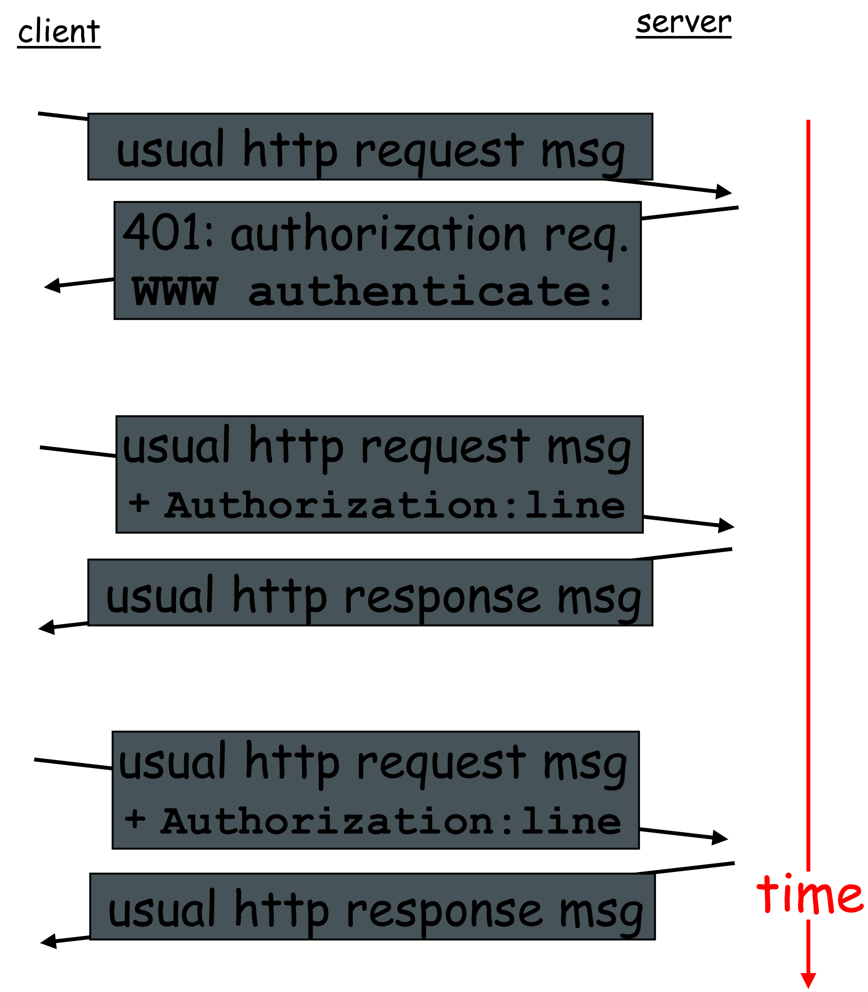
- Type in request, and look at the response

```
GET /~hqi/index.html HTTP/1.0
```

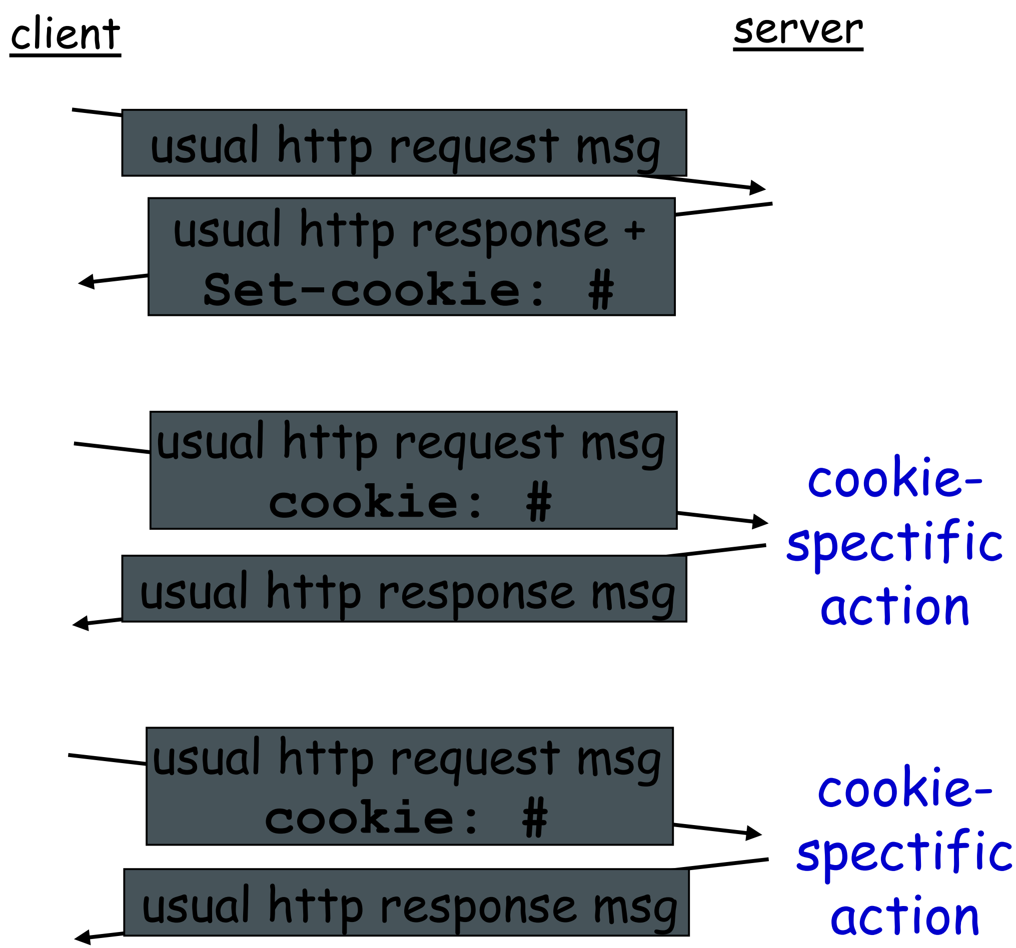
```
[hqi@com779 hqi]$ telnet panda.ece.utk.edu 80

Trying 160.36.30.108...

Connected to panda.ece.utk.edu.

Escape character is '^]'.


get /~hqi/index.html http/1.0


HTTP/1.1 501 Method Not Implemented

Date: Sun, 02 Sep 2001 21:03:28 GMT

Server: Apache/1.3.19 (Unix)  (Red-Hat/Linux) PHP/4.0.4pl1

Allow: GET, HEAD, POST, PUT, DELETE, CONNECT,
OPTIONS, PATCH, PROPFIND, PROPPATCH, MKCOL, COPY,
MOVE, LOCK, UNLOCK, TRACE

Connection: close

Content-Type: text/html; charset=iso-8859-1
```

```
[hqi@com779 hqi]$ telnet panda.ece.utk.edu 80
Trying 160.36.30.108...
Connected to panda.ece.utk.edu.
Escape character is '^]'.

GET /~hqi/index.html http/1.0

HTTP/1.1 200 OK
Date: Sun, 02 Sep 2001 21:05:43 GMT
Server: Apache/1.3.19 (Unix)  (Red-Hat/Linux) PHP/4.0.4pl1
Last-Modified: Sat, 07 Jul 2001 15:42:14 GMT
ETag: "1f222e-df9-3b472dd6"
Accept-Ranges: bytes
Content-Length: 3577
Connection: close
Content-Type: text/html

<HTML>
……
</HTML>
Connection closed by foreign host.
```

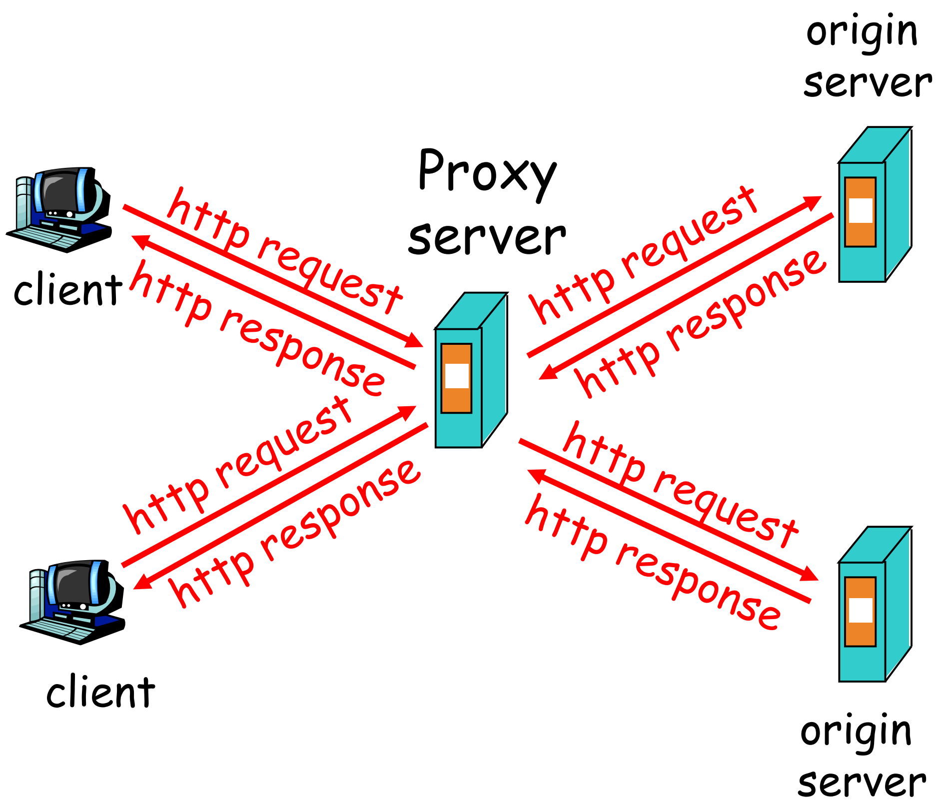
- Purpose of authentication: control access to document
- Means: user name and password
- User must present password on each request, Authorization:line
- Server asks for it by giving it the response with WWW authenticate:



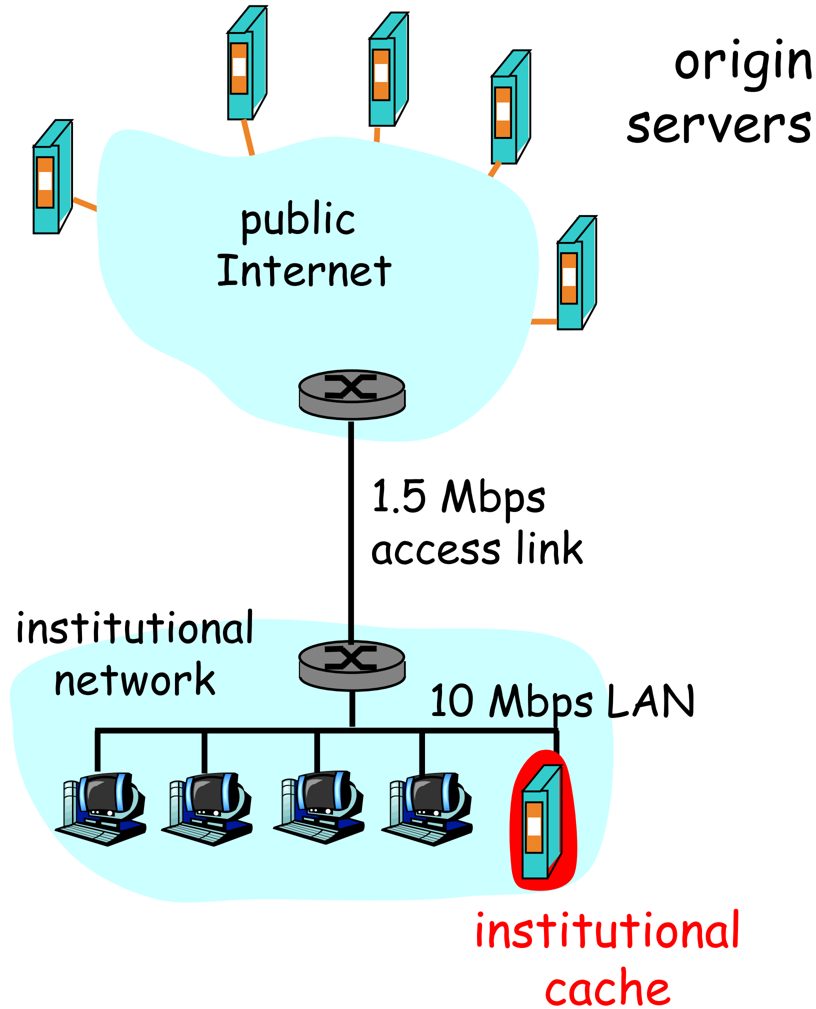
- Server sends user cookie in response message:
  - **`Set-cookie: 1678453`**
- Client presents cookie in later request
  - **`cookie: 1678453`**
- Server matches cookies with stored information: such as user preference, password etc



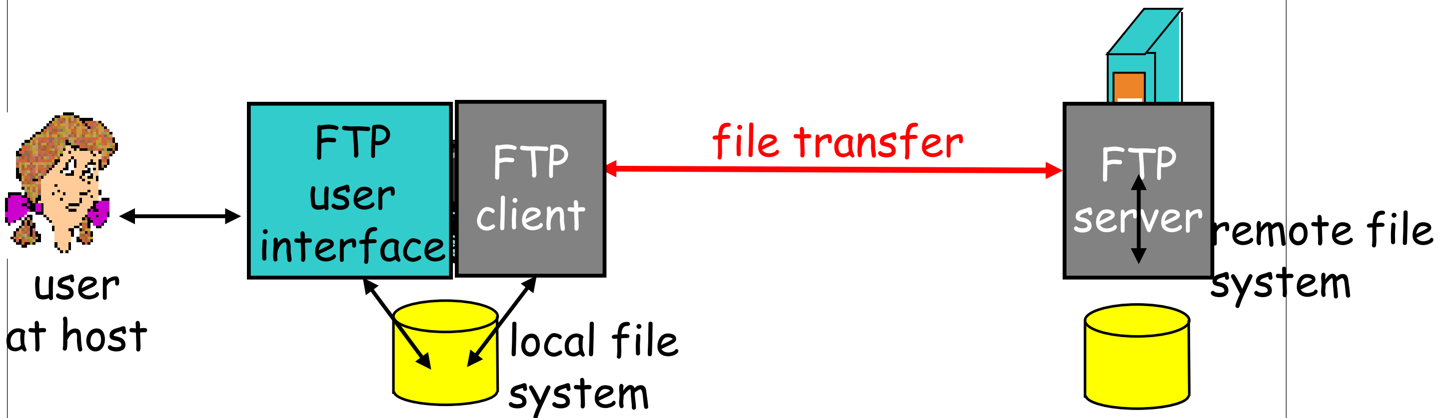
- Goal: to satisfy user request without invoking origin server
- User makes request, the object requested has been cached, then proxy server will reply, else proxy server request the object for client and then response



- Cache should be closer to the clients
- Faster response
- Reduce traffic (pay less money)
- Web cache:
  - Cost is low



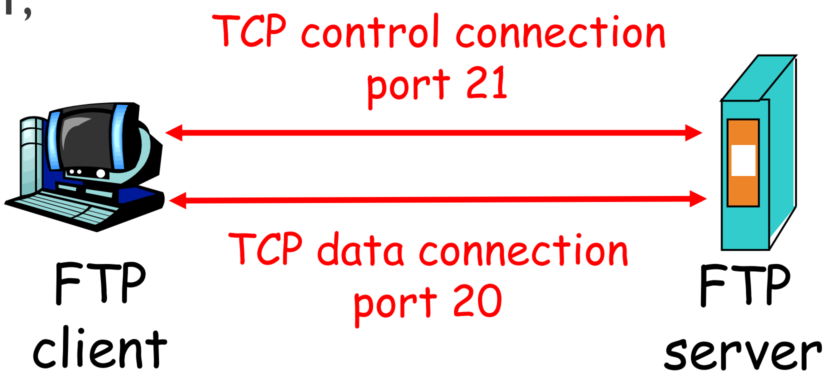
FTP: FILE TRANSFER PROTOCOL



- transfer file to/from remote host
- client/server model
  - *client:* initiates transfer (either to/from remote)
  - *server:* remote host
- ftp: RFC 959

- ftp client contacts ftp server at port 21, specifying TCP as transport protocol
- two parallel TCP connections opened:
  - control: exchange commands, responses between client, server

    "out of band control"
  - data: file data to/from server
- ftp server maintains "state": current directory, earlier authentication



Data connection is closed whenever it finished transferring one file.

FTP: COMMAND AND RESPONSE

Sample commands:

- sent as ASCII text over control channel
- **USER *username***
- **PASS *password***
- **LIST** return list of file in current directory
- **RETR filename** retrieves (gets) file
- **STOR filename** stores (puts) file onto remote host

Sample return codes

- status code and phrase (as in http)
- **331 Username OK, password required**
- **125 data connection already open; transfer starting**
- **425 Can't open data connection**
- **452 Error writing file**

Thank you

The content in this material are from the textbooks and reference books given in the syllabus.