CprE 450/550x
Distributed Systems and Middleware

Basics of Computer Networks (cont.)

Yong Guan
3216 Coover
Tel: (515) 294-8378
Email: guan@ee.iastate.edu
January 27, 2004

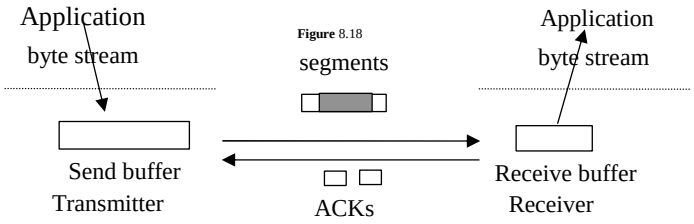
Transport Layer

Message Stream (UDP)
Byte-Stream (TCP)

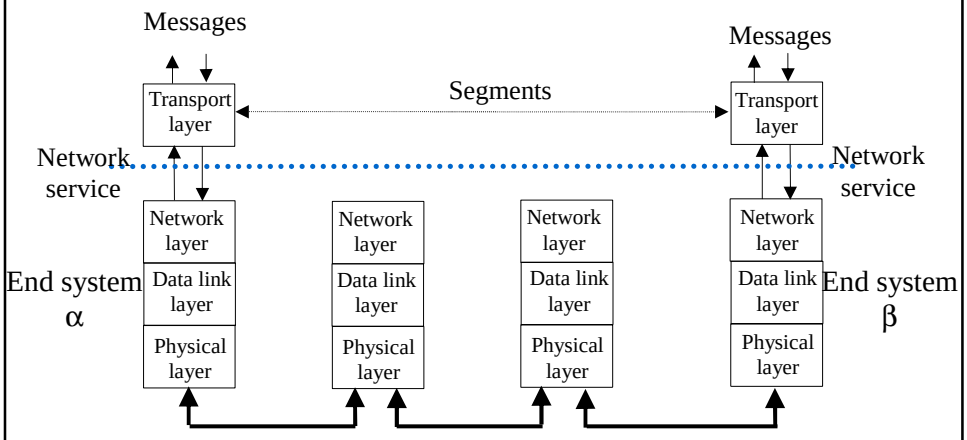
Transport Layer

Outline

- End-to-End Protocols
- Underlying Network Service Model
- Message *vs*. Byte Streams
- Connection Establishment/Termination
- Sliding Window Revisited
- Flow Control
- Adaptive Timeout



End-to-End Protocols over A Network Layer



Transport Services and Protocols

5

- Provide *logical communication* capability between application processes running on different hosts
- Transport protocols run in end systems
- Transport *vs*. Network Layer services:
  - *network layer: d*ata transfer between end systems
  - *transport layer: d*ata transfer between processes
    - relies on, enhances, network layer services

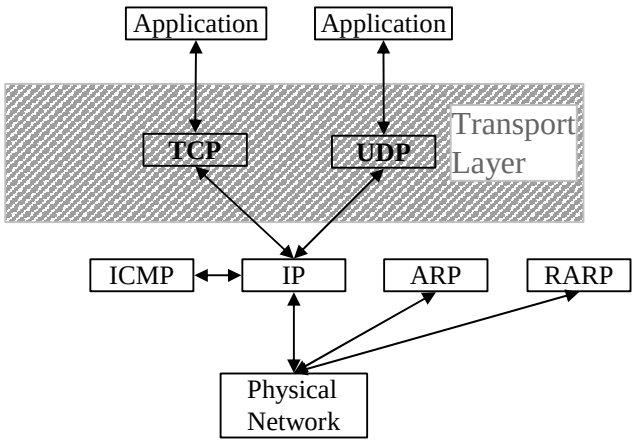
End-to-End Protocols

6

- Underlying Best-Effort network (IP)
  - drop messages
  - re-orders messages
  - delivers duplicate copies of a given message
  - limits messages to some finite size
  - delivers messages after an arbitrarily long delay
- Common End-to-End services
  - guarantee message delivery
  - deliver messages in the same order they are sent
  - deliver at most one copy of each message
  - support arbitrarily large messages
  - support synchronization
  - allow the receiver to flow control the sender
  - support multiple application processes on each host

IP Related Protocols

7



Transport-Layer Protocols

8

Internet Transport Services:

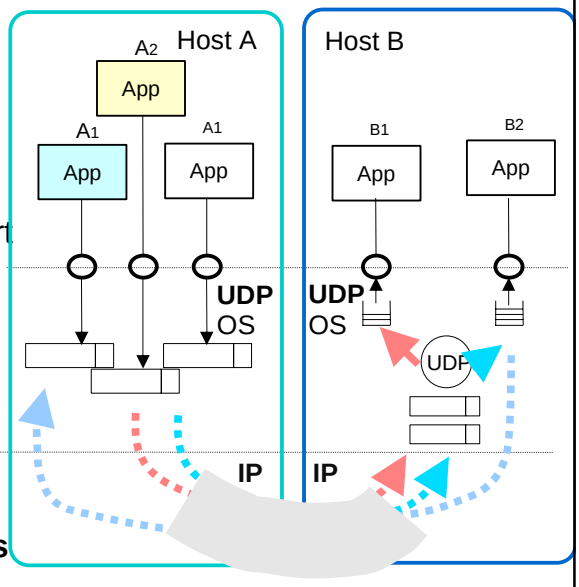
- TCP: Reliable, in-order unicast delivery
  - congestion
  - flow control
  - connection setup
- UDP: Unreliable (“best-effort”), unordered unicast or multicast delivery
- Services not supported:
  - real-time
  - bandwidth guarantees
  - reliable multicast.

UDP Overview [RFC 768]

1. UDP is a *connectionless datagram service*.
   - No connection establishment: packets may show up at any time.
2. UDP packets are *self-contained*.
   - Every UDP packet is a single message ;
   - “Message-oriented” communication ;
   - Message “boundaries” are maintained by UDP ;
   - UDP reassenbled at destintation and delivered as single message
3. UDP is *unreliable*:
   1. No acknowledgements to indicate delivery of data.
   2. Checksums cover the header, and only optionally cover the data.
   3. Contains no mechanism to detect missing or mis-sequenced packets.
   4. No mechanism for automatic retransmission.
   5. No mechanism for flow control, and so can over-run the receiver.

User Datagram Protocol (UDP)

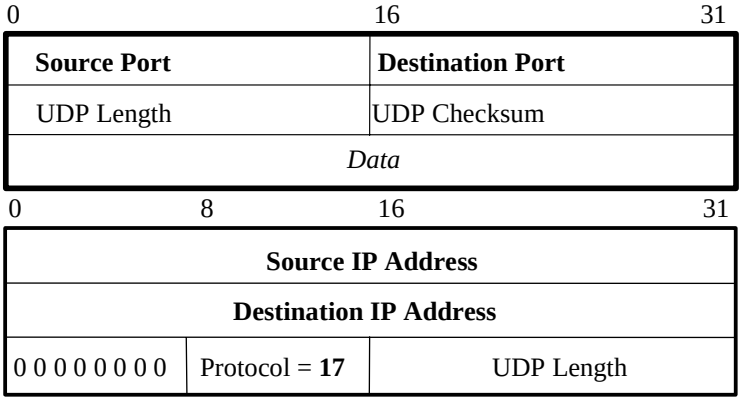
- Unreliable and unordered datagram service [RFC 768]
- No flow control
- Simple Packet Multiplexor for Transport Users
  - Endpoints identified by *port numbers: ports are the de-multiplexing keys*
  - servers have *well-known* ports
  - see **`/etc/services`** on Unix



Simple Packet Multiplexor (UDP)

11

- Header Format [RFC 768]
  - Source / Destination Ports: 16 bit integers of source and sink;
  - UDP Length: 16 bit unsigned integer for entire datagram.
    - Size ≤ 2^{16} (64KB); Fragmentation by IP
  - Optional IP checksum: pseudo-header + UDP-header + data



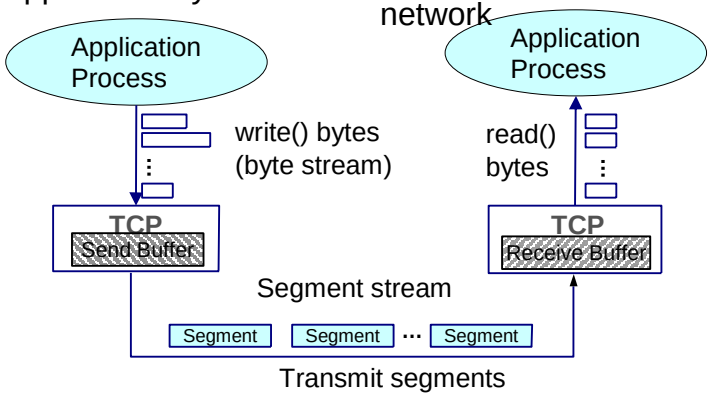
User-Datagram Protocol (UDP)

12

- Applications may prefer using UDP:
  - *don't need reliable delivery ;*
  - *do not need connection oriented service or cannot afford the connection setup overhead ;*
  - *have their own special needs, such as streaming of real-time audio or video ;*
  - `tftp`*, RIP, DNS, SNMP, RTP, etc.*

TCP Overview [RFC793, 1122, 1323, 2018, 2581]

- Connection-oriented
- Byte-stream
  - Application writes bytes
  - TCP sends *segments*
  - appl. reads bytes
- Full duplex
- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network



TCP Characteristics

1. TCP is *connection-oriented*.
   - 3-way handshake used for connection setup/teardown.
2. TCP provides a *stream-of-bytes* service.
3. TCP is reliable:
   - Acknowledgements indicate delivery of data.
   - Checksums are used to detect corrupted data.
   - Sequence numbers detect missing, or mis-sequenced data.
   - Corrupted data is retransmitted after a timeout.
   - Mis-sequenced data is re-sequenced.
   - (Window-based) Flow control prevents over-run of receiver.
4. TCP uses *congestion control* to share network capacity among users.

Encapsulation of User Data

15

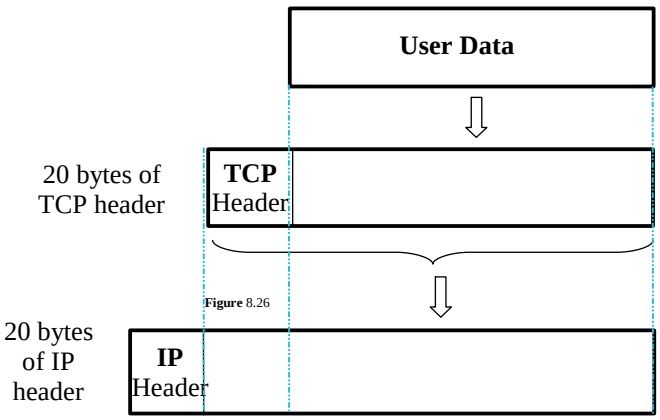


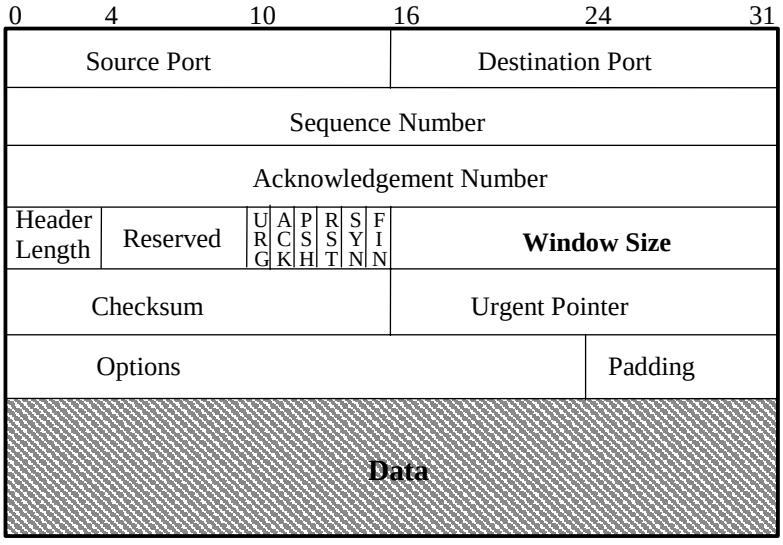
Figure 8.26

- User data are encapsulated in TCP Segments.
- A TCP Segment is the unit of data for user messages.
- Maximum Segment Size (MSS) ≤ Path MTU

TCP Segment Format

16

- TCP Segment Format



TCP Segment Format – 1

17

- Source / Destination Ports (16b unsinged int):
  - the source and sink port numbers of transport user
- Sequence (Acknowledgement) Numbers (32b unsigned int):
  - number of first byte sent (expected from other side) in the segment to other side
  - Initial Sequence Number (ISN) by sender (ISN + 1) (**SYN**)
    - ISN is (must be) chosen at random.
  - Acknowledgement Number is byte expected next (**ACK**)
- Header Length: in 32b words
- Reserved (0)

TCP Segment Format – 2

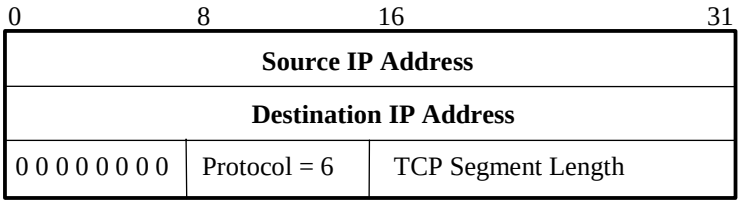
18

- Flags:
  - URG: *urgent pointer* is valid
  - ACK: Acknowledgement number is valid
  - PSH: deliver data received by receiving TCP immediately
  - RST: Receving TCP must abort connection
  - SYN: Connection Request with ISN = SN;
  - FIN: Sender has no more data to send to receiving TCP;
    - `shutdown(sd, 1) /* C */;`
- (“Advertised”) Window Size: amount of data receiver is willing to accept.
  - Credit of data
- Urgent Pointer (if URG flag is set)
  - points to last byte of “urgent” data;
  - any data from beginning of segment to UP are “urgent”

TCP Header Fields (*Cont'd)*

19

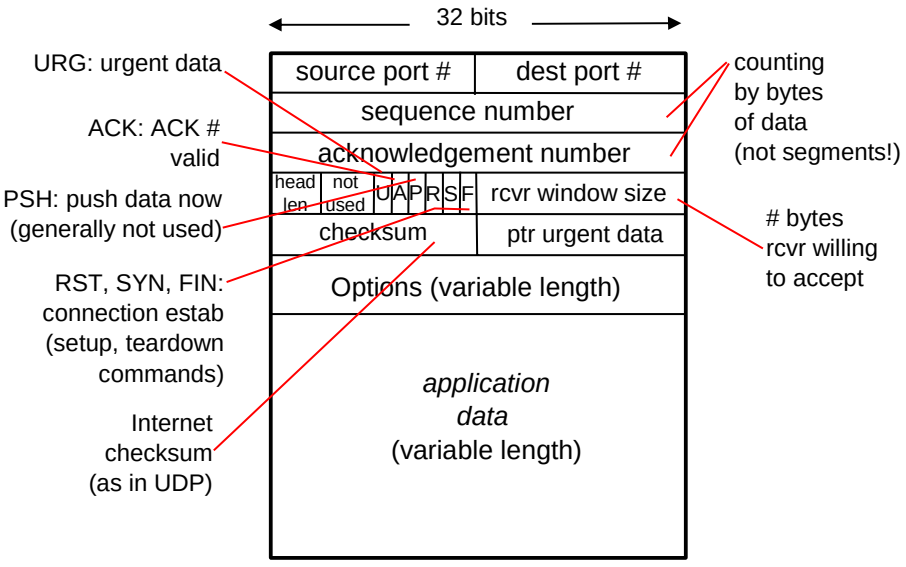
- TCP Checksum is computed over
  - TCP header + TCP Pseudo-header + TCP segment data



- Options Field includes optional settings

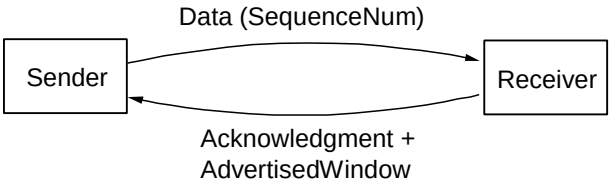
TCP Segment Structure (Summary)

20



Segment Format (*cont'd*)

- Each TCP connection is identified with 4-tuple:
  - **`(TCP, SrcPort, SrcIPAddr, DsrPort, DstIPAddr)`**
- TCP Sliding Window + Flow Control:
  - **`( acknowledgment, SequenceNum, AdvertisedWindow )`**



- Flags
  - `FIN, RESET, PUSH, URG, ACK`

TCP Connection Management

TCP sender, receiver establish "connection" before exchanging data segments

- Initialize TCP variables:
  - sequence numbers
  - buffers, flow control info (*e.g.,* **`RcvWindow`**)
- *client :* connection initiator
  - **`socket(); sockaddress;`**
  - **`connect();`**
- *server :* contacted by client
  - **`socket(); sockaddress; bind(); listen(); accept();`**

Three way handshake:

Step 1: client end system sends TCP SYN control segment to server
- specifies initial sequence number
- sides exchange initialization parameters, such as MSS.

Step 2: server end system receives SYN, replies with SYNACK control segment
- ACKs received SYN
- allocates buffers
- specifies server→ receiver initial sequence number.

TCP Three-way Handshake for Connection Establishment

- **Upon Connection Establishment, both sides exchange the Maximum Segment Sizes (MSS) which are willing to accept:**
  - **If client is non-local, MSS = 536 (default)**
  - **536 = 576 – 20+20**
  - **MSS only if SYN is set**

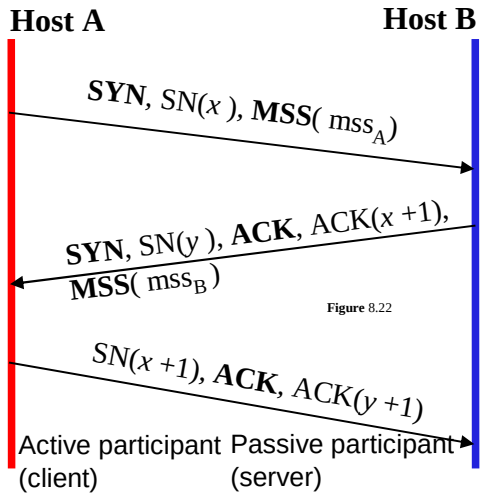
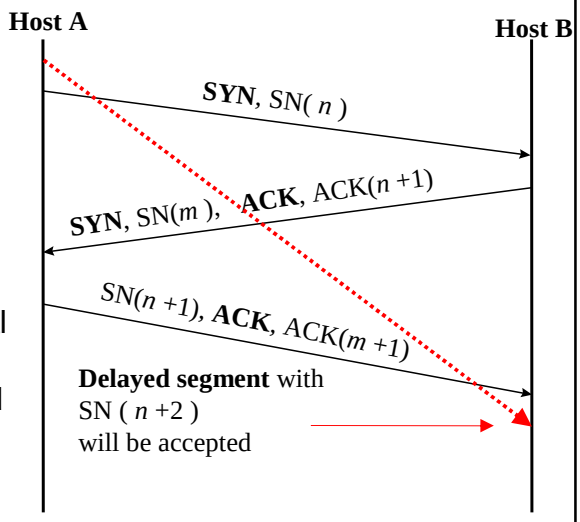


Figure 8.22

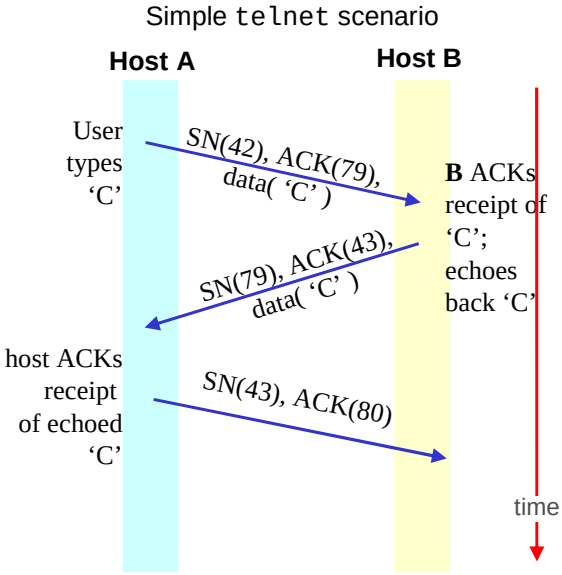
TCP: Initial Sequence Number

- The ISN should change over time [RFC 793].
  - ISN should be viewed as a counter which increments every 4μ–seconds
- When the same initial sequence number is chosen by a host, old segments cannot be distinguished from new current ones.



Data Transfer: Sequence Numbers and ACKs

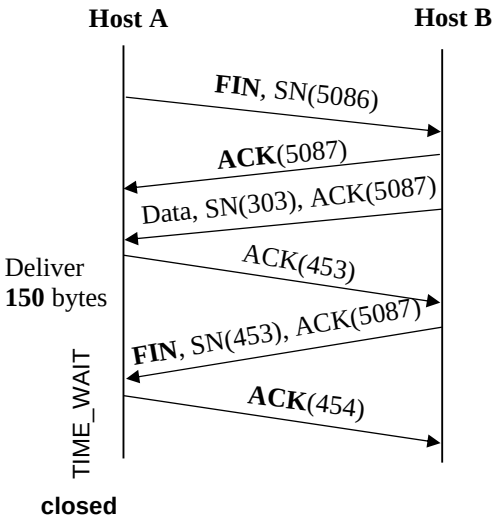
- Sequence Numbers:
  - byte stream "number" of first byte in segment's data
- ACKs:
  - seq. # of next byte expected from other side
  - cumulative ACK
- TCP spec doesn't say *how receiver handles out-of-order segments*
  - up to implementor
- Example next figure:
  - `telnet` interaction;



TCP Connections: Graceful Close

Closing a TCP connection:

- TCP close is symmetric
- client **`close();`** socket
- Step 1: client end system sends TCP FIN control segment to server (client knows when input has finished
- Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.
- Note: Connections stay for 2MSL in TIME_WAIT state;
  - MSL = 30, 60, 120 secs



State Transition Diagram

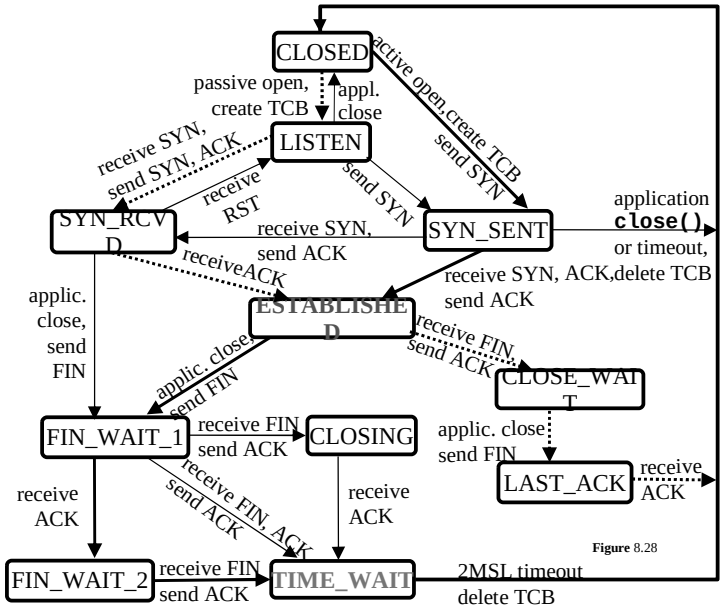
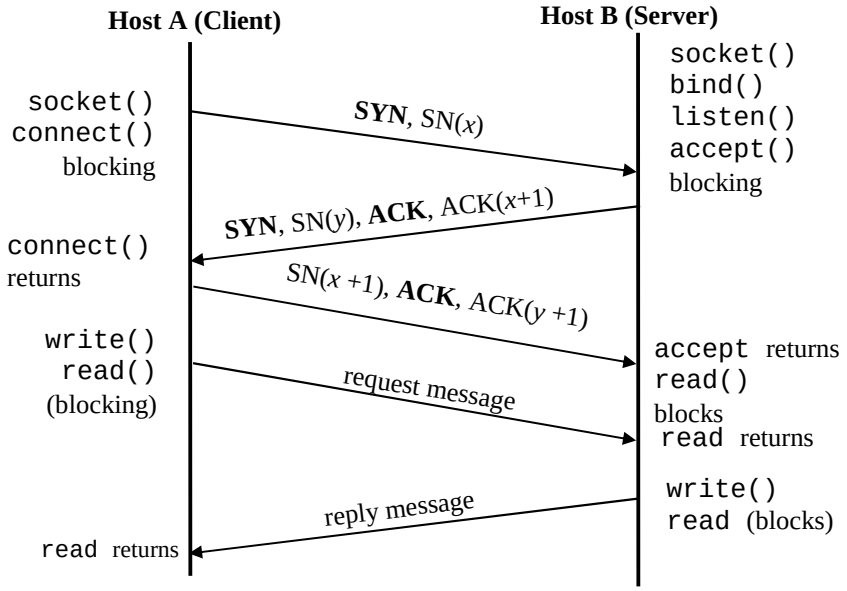


Figure 8.28

TCP and POSIX Socket API

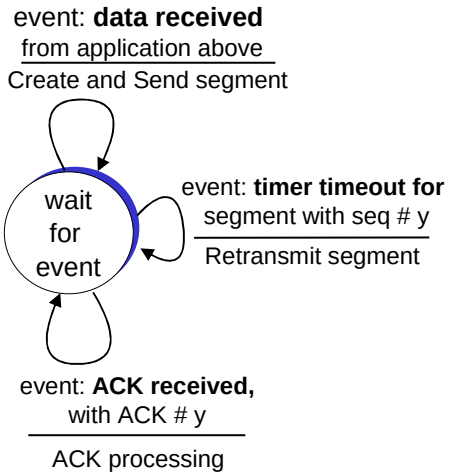


TCP: Reliable Data Transfer

Simplified Sender

Assume:

1. one way data transfer ;
2. no flow control ;
3. no congestion control .



TCP ACK Generation [RFC 1122, RFC 2581]

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	*delayed* ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next *expected* byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

TCP: Reliable Data Transfer (SSender)

Simplified TCP Sender (SSender)

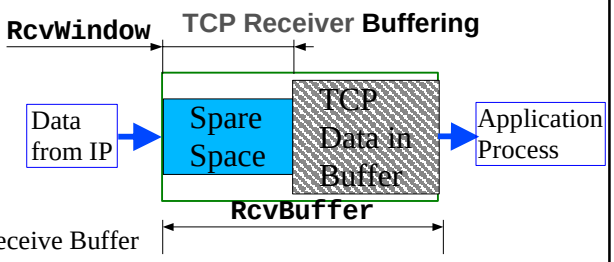
```
00  sendbase = initial_sequence number ;
01  nextseqnum = initial_sequence number ;
03  while ( TRUE ) {
04    switch ( event )
05        event Data received from application above :
06                create TCP segment with sequence number nextseqnum ;
07                start timer for segment nextseqnum ;
08                pass segment to IP;
09                nextseqnum = nextseqnum + length(data) ;
10        event Timer Timeout for segment with sequence number y :
11                Retransmit segment with sequence number y ;
12                compue new timeout interval for segment y ;
13                restart timer for sequence number y ;
14        event  ACK received, with ACK field value of y :
15                if (y > sendbase) { /* cumulative ACK of all data up to y */
16                   cancel all timers for segments with sequence numbers < y ;
17                   sendbase = y ;
19                } else { /* a duplicate ACK for already ACKed segment */
20                   increment number of duplicate ACKs received for y ;
21                   if ((number of duplicate ACKS received for y) == 3) {
22                            /* TCP fast retransmit */
23                      resend segment with sequence number y ;
24                      restart timer for segment y ;
25                }
26    } /* end of while ( TRUE ) */
```

TCP Basic Problem: Flow Control

- How much traffic can sender introduce in a connection and the network?
  - How much data can a TCP sender have outstanding in the network?
  - How much data should TCP retransmit when an error occurs? Just selectively repeat the missing data?
  - How does the TCP sender avoid over-running the receiver's buffers?
- Two components
  1. Flow Control: Make sure that the receiver can receive as fast as you send them out ;
  2. Congestion Control: Make sure that the network can deliver the packets to the receiver as fast as you send them out.

TCP Window Flow Control

TCP Flow Control
sender shouldn't overrun receiver's buffers by transmitting data too fast
Receiver must determine the data **rate**

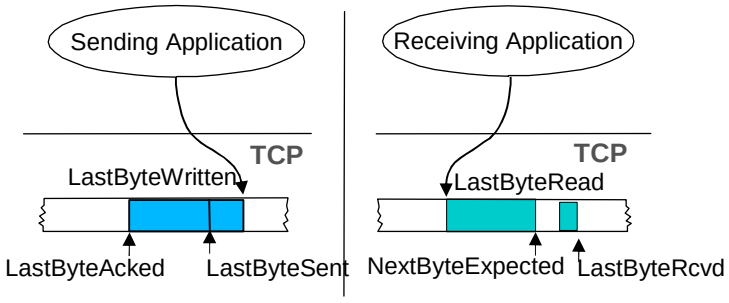


RcvBuffer = size or TCP Receive Buffer
RcvWindow = amount of spare room in Buffer
RcvWindow = Advertised Window

- TCP Receiver: explicitly informs sender of (dynamically changing) amount of free buffer space in **RcvBuffer**
  - **RcvWindow field** in TCP Header contains the free buffer space
- TCP Sender: keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**

Sliding Window Flow-Control Revisited

- Sending TCP side
  - **LastByteAcked < = LastByteSent**
  - **LastByteSent < = LastByteWritten**
  - buffer bytes between **LastByteAcked** and **LastByteWritten**
- Receiving TCP side
  - LastByteRead < NextByteExpected
  - NextByteExpected < = LastByteRcvd +1
  - buffer bytes between NextByteRead and LastByteRcvd



TCP Flow Control

35

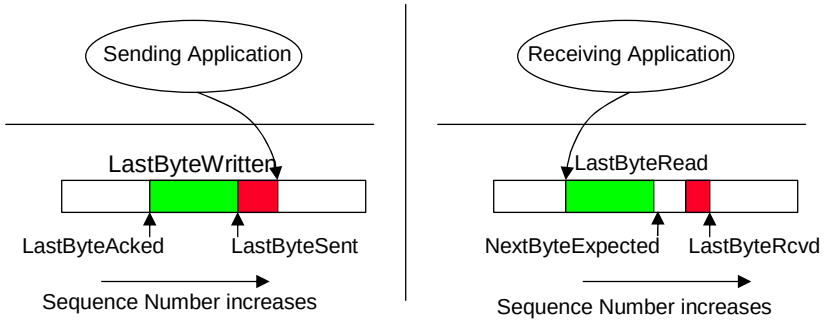
- TCP Send buffer size: **`MaxSndBuff`**
- TCP Receive buffer size: **`MaxRcvBuff`**
- Receiving side:
  - **`LastByteRcvd`** – **`LastByteRead`** ≤ **`MaxRcvBuff`**
  - **`AdvertisedWind`** = **`MaxRcvBuff`** - (**`LastByteRcvd`** - **`NextByteRead`**)
- Sending side
  - **`LastByteSent`** - **`LastByteAcked`** ≤ **`AdvertisedWindow`**
  - **`EffectiveWind`** = **`AdvertisedWind`** - (**`LastByteSent`** - **`LastByteAcked`**)
  - **`LastByteWritten`** - **`LastByteAcked`** ≤ **`MaxSendBuff`**
  - if ((**`LastByteWritten`** - **`LastByteAcked`**) + *y* > **`MaxSenderBuff`**)
    `block sender; /* application write() blocks */`
- Always send ACK in response to arriving data segment
  - See later TCP ACK transmisstion policies
- Persist when **`AdvertisedWind = 0`**

TCP Flow Control

36

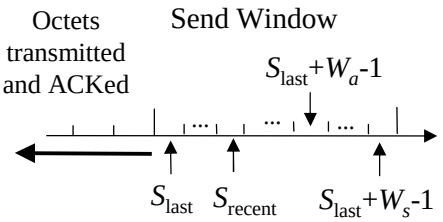
- Receiver window (**`MaxRcvBuff`** – maximum buffer size at receiver)
- Sender window (**`MaxSendBuff`** – maximum size at sender)

AdvertisedWind = MaxRcvBuff – (LastByteRcvd – LastByteRead)
EffectiveWind = AdvertisedWind – (LastByteSent – LastByteAcked)
MaxSendBuff ≥ LastByteWritten – LastByteAcked



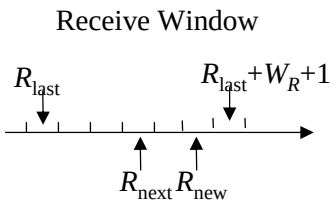
TCP Sender and Receiver State [Gavi]

Transmitter



S_{last} oldest unacknowledged octet
S_{recent} highest-numbered transmitted octet
$S_{last}+W_a-1$ highest-numbered octet that can be transmitted
$S_{last}+W_s-1$ highest-numbered octet that can be accepted from the application

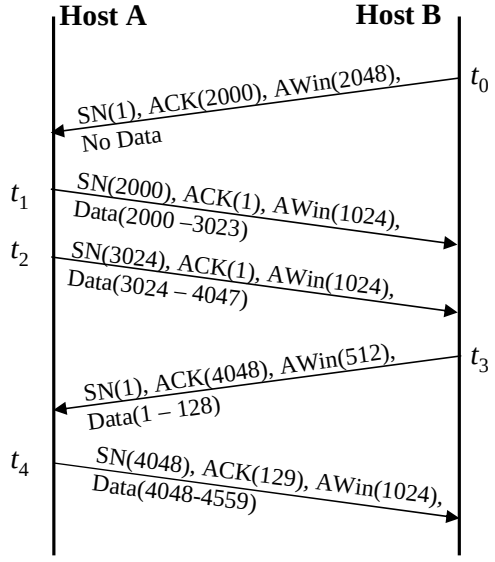
Receiver



R_{last} highest-numbered octet not yet read by the application
R_{next} next expected octet
R_{new} highest numbered octet received correctly
$R_{last}+W_R-1$ highest-numbered octet that can be accommodated in receive buffer

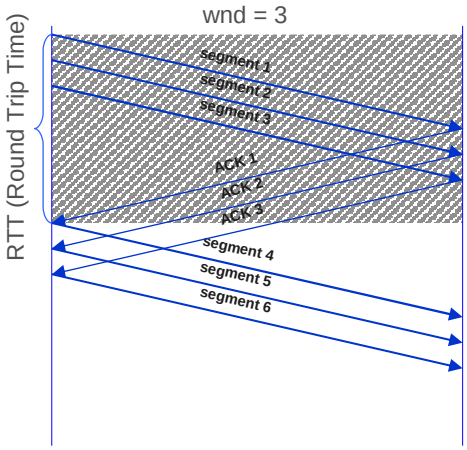
TCP Window Flow Control

- TCP exchange of segments and ACKs
- TCP: Window Flow-Control receiver sends explicit amount of free space in RcvBuffer (credit amount)
  - SN(*n*): sequence number *n*,
  - ACK(*m*): acknowledge *m*-1
  - AWin(*k*): the amount of credit at TCP receiver buffer (advertised window or **`AdvertisedWind`**)

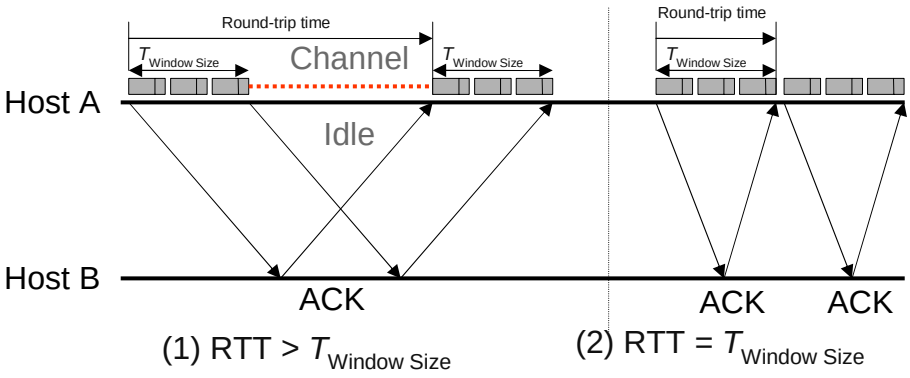


Flow Control: Window Size and Throughput

- Sliding-window based flow control:
  - Higher window → higher throughput
  - Throughput = wnd/RTT
  - Need to worry about sequence number wrapping
- Window size controls throughput:
  - $W_S \geq 2\alpha + 1$
  - *Familiar ?*



Flow Control: Window Size and Throughput



TCP Round Trip Time and Timeout

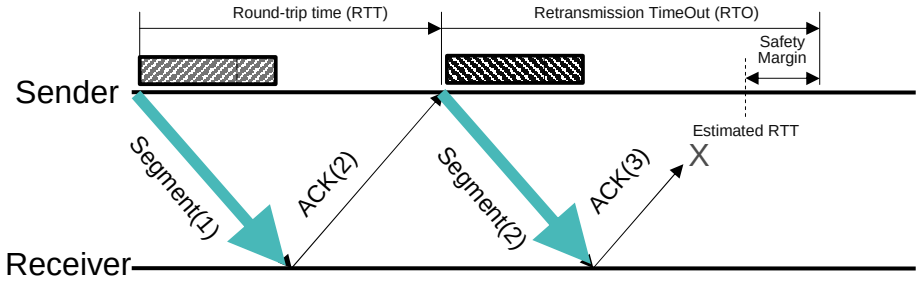
TCP Retransmission Timeout Value

- Longer than RTT
  - RTT is varying with time
- Too short: premature timeout
  - unnecessary retransmissions
- Too long: slow reaction to segment loss

RTT Estimation

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

TCP: Retransmission and Timeouts



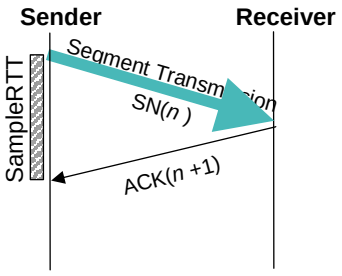
TCP has to use adaptive retransmission timeout values:

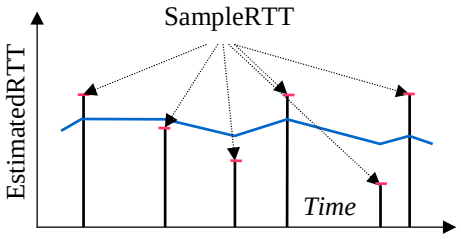
Congestion
Changes in Routing
} RTT changes frequently

Adaptive Retransmission (Original Algorithm)

- Measure **`SampleRTT`** for each TCP segment / ACK pair
- Compute smoothed estimator **`EstimatedRTT`** of RTT with a low-pass filter (*)
- Set timeout based on **`EstimatedRTT [RFC 793]`**
  - **`TimeOut = 2 ×EstimatedRTT`**

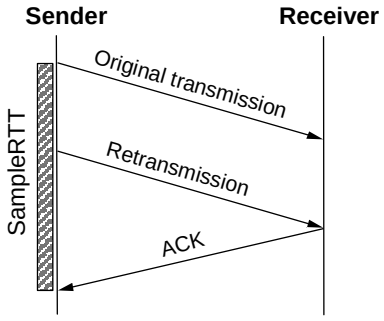
SampleRTT = AckRcvdTime – SendSegmentTime
EstimatedRTT = $\alpha \times$ EstimatedRTT + (1 - α) $\times$ SampleRTT
TimeOut = $\beta \times$ EstimatedRTT,
where $0 < \alpha \leq 1$, (in practice α between 0.8 and 0.9, $\beta = 2$)

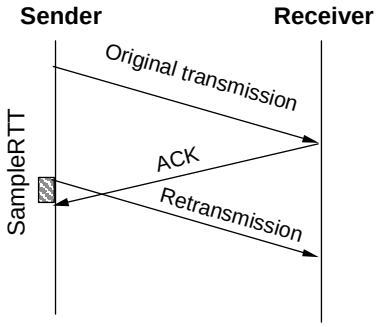




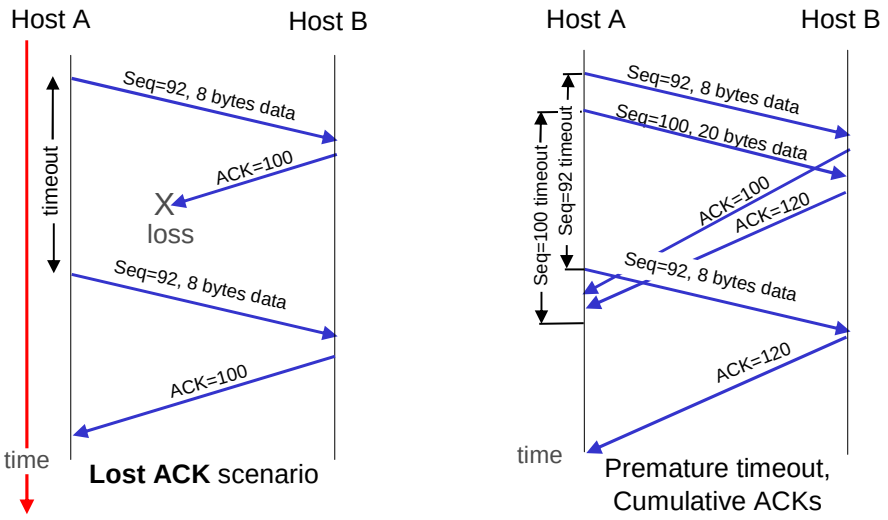
Karn/Partridge Algorithm [1987]

- Do not sample RTT when retransmitting
  - Measure *SampleRTT* only for original transmissions
- Double timeout after each retransmission
  - Exponential backoff → for each retransmission, double *EstimatedRTT*





TCP: Retransmission Scenarios



Jacobson/Karels Algorithm

- Smoothed estimator **`EstimatedRTT`** of RTT cannot follow dynamic fluctuations of in the RTT [Jacobson 1988, 1990]
- Need to take into account the variance in ***SampleRTT***
- Note
  - accurate timeout mechanism important to congestion control
  - algorithm only as good as granularity of clock (500ms on OLD UNIX systems; modern UNIX 10ms is common)

$$
\begin{aligned}
&\text{diff} = \text{SampleRTT} - \text{EstimatedRTT} ;\\
&\text{EstimatedRTT} = \text{EstimatedRTT} + \delta \times \text{diff} ;\\
&\text{Deviation} = \text{Deviation} + \delta \times (|\, \text{diff} \,| - \text{Deviation}) ;\\
&\text{TimeOut} = \mu \times \text{EstimatedRTT} + \varphi \times \text{Deviation} ;\\
&0 < \delta \leq 1,\\
&\mu = 1,\\
&\phi = 4
\end{aligned}
$$

TCP: Implementation Policy Options – 1

47

- Several policy implementation posibilities are left open by the TCP standard.
  - Send Policy : if PUSH flag is unset and window closed, TCP buffers application data in the TCP_Transmit_Buffer. TCP may create one segment per `write()` or it may concatenate all written data into one message.
  - Delivery Policy : If PUSH flag is unset, receiving TCP may deliver to application received data, either in units as each segment arrives, or it may buffer them until the total amount of data exceeds a threshold.
  - Segment Accept Policy : Segment data are accumulated in the TCP_Receive_Buffer as they arrive. When segments arrive out of order TCP can accept segments:
    - *In-Order* : out of order segments are discarded; this operates similar to Go-Back-*N* window flow control ;
    - *In-Window* : accept segments falling within the receive window ;

TCP: Implementation Policy Options – 2

48

- TCP policy implementation (*cond't*)
  - Retransmit Policy : TCP has to retransmit a segment SN(*n)* if ACK(*n*+1) is not received in due time. There are three re-transmission policies:
    - *First-Only* : One re-tranmission timer is maintained for the entire send queue. ACKs reset timer and allow TCP to remove the ACKed segments from the queue. On timer expiration TCP retransmits segment at front of queue and resets the timer ;
    - *Batch* : One re-tranmission timer is maintained for the entire send queue. ACKs reset timer and allow TCP to remove the ACKed segments from the queue. On timer expiration TCP retransmits all segment in the queue and resets the timer ;
    - *Individual* : One re-tranmission timer is maintained for each segment in the send queue. ACKs reset timers and allow TCP to remove the ACKed segments from the queue. On timer expiration TCP retransmits the corresponding segment and resets the timer ;

TCP: Implementation Policy Options – 3

49

- TCP policy implementation (*cond't*)
  - Acknowledgement Policy : when TCP receives a segment that is in sequence SN(*n*), has two options:
    - *Immediate Acknowledgement* : After data is accepted by application TCP imediately sends ACK(*n*+1) frame ;
    - *Cummulative Acknowledgement* : TCP may continue accumulating data waiting until an outbound segment is ready within which ACK(*n*+1) is piggybacked. A timer for the ACK(*n*+1) is set to expire in case that no outbound segments need to be transmitted in the near future.

TCP Error Control

50

- TCP implements a variation of Go-Back-*N* ARQ retransmission error-control scheme.
- It couples error and congestion control:
  - errors are caused by congestion
- TCP maintains four timers per connection.

TCP Implementations: Timers

51

- TCP maintains four timers per connection.
  - Retransmission Timer: started after the transmission of a segment; expirations causes segment retransmission;
  - Persist Timer: used to ensure that window credit information is replied to, even if the other side (receiver) has advertised window size = 0;
  - Keepalive Timer : used to periodically probe the other side and detect crashes ;
  - 2MSL Timer : measures the time a connection has been in TIME_WAIT state.

TCP Implementations: Retransmission Timer

52

- Retransmission Timer (RT): it is started after the transmission of a segment SN(*n*); expirations causes retransmission of this segment ;
  - The RT doubles its timer period when consecutive time outs take place ;
  - RT $\leq$ 64 seconds, always ;
  - TCP gives up after a fixed number of retransmissions (~14).

TCP Implementations: Persist Timer

53

- Persist Timer (PT): is used to let sender probe periodically the receiver with a one byte segment, of its flow control window size (advertised window) in order to detect when it increases above zero.
  - When a receiver's advertised window goes down to zero and the ACK that notifies the sender that it has been opened is lost, is lost the sender wont be notified of the possibility of restarting transmission.
  - The PT forces the sender to query the receiver about its advertised window size ; the PT is started by the sender ;
  - PT is also exponentially backed-off, rounded to 5sec ;
  - PT = { 5, 5, 6, 12, 24, 48, 60, 60, ... }, $5 \leq RT \leq 64$ seconds, always ;
  - TCP continues to probe until advertised window > 0.

TCP Implementations: Keep-alive Timer

54

- Keep-alive Timer (AT): is used to let a TCP probe periodically the other side to make sure that it is still alive.
  - A probe packet is sent after the connection has been idle for 2 hours ;
  - After A has probed B:
    - B responds with an ACK — B is up and running;
    - No response from B — B has crashed ; A will send 10 more probes, each 75sec apart. If still no response, A closes the connection.
    - B responds with a RST — B was rebooted; it resets ;
    - B responds with a RST — B is up but unreachable; it resets ;

Transport Layer and Congestion Control

55

Congestion:

- Informally: “too many sources sending too much data too fast for *network* to handle”
- Different from flow control
- Manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- A very important (and often very hard to solve) problem!

TCP Congestion Control

56

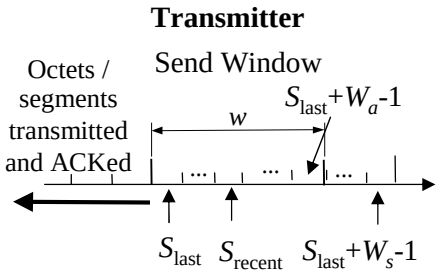
- Two “phases”
  - slow start
  - congestion avoidance
- Variables:
  - **`Congwin`**
  - **`threshold:`** defines threshold between two slow start phase, congestion control phase

- 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

TCP Congestion Control

- End-to-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:

$$\text{Throughput} = \frac{w \times \text{MSS}}{\text{RTT}} \quad \text{Bytes/sec}$$



S_{last} oldest unacknowledged octet
S_{recent} highest-numbered transmitted octet
$S_{last}+W_a-1$ highest-numbered octet that can be transmitted
$S_{last}+W_s-1$ highest-numbered octet that can be accepted from the application

Transport Layer: Summary

- Principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Implementation in the Internet
  - UDP
  - TCP

Application Layer

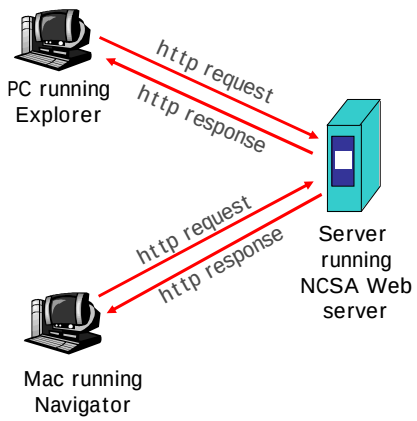
Internet apps: application, transport protocols

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	NSF	TCP or UDP
Internet telephony	proprietary (e.g., Vocaltec)	typically UDP

The Web: the http protocol

http: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests
- http1.0: RFC 1945
- http1.1: RFC 2068



The http protocol: more

http: TCP transport service:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- http messages (application-layer protocol messages) exchanged between browser (http client) and Web server (http server)
- TCP connection closed

http is "stateless"

- server maintains no information about past client requests

aside

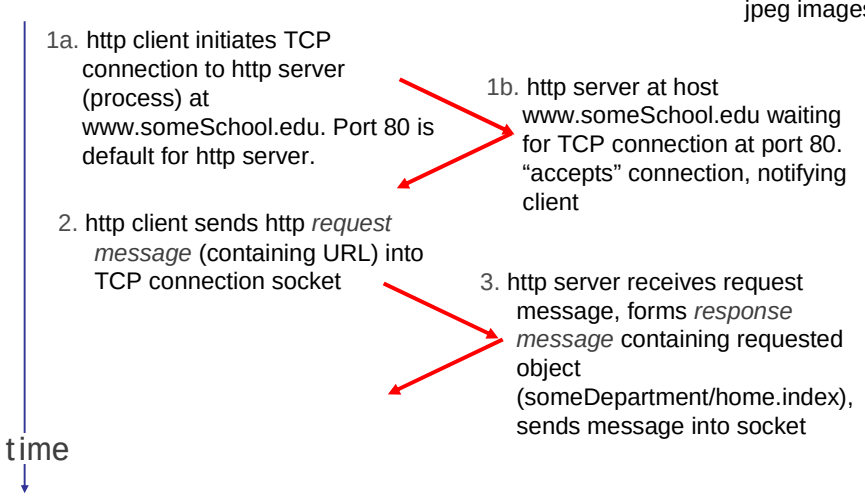
Protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

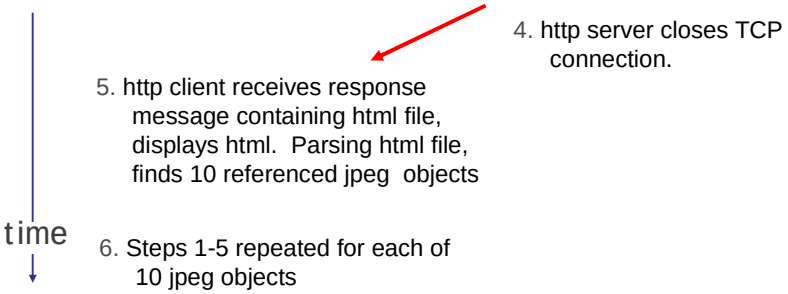
http example

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

(contains text, references to 10 jpeg images)



http example (cont.)



Non-persistent, persistent connections

Non-persistent

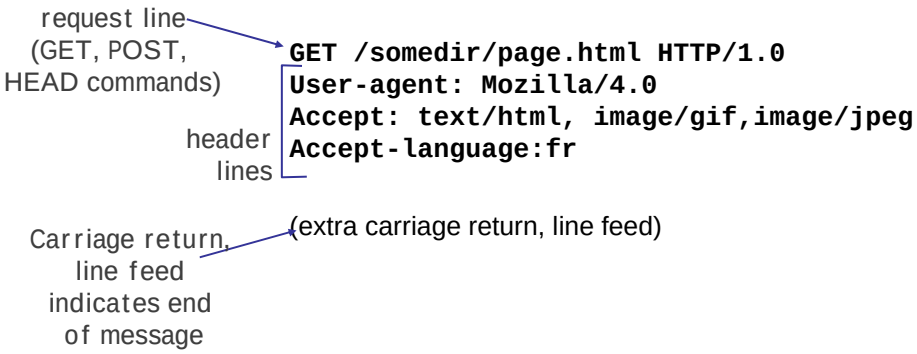
- http/1.0: server parses request, responds, closes TCP connection
- 2 RTTs to fetch object
  - TCP connection
  - object request/transfer
- each transfer suffers from TCP's initially slow sending rate
- many browsers open multiple parallel connections

Persistent

- default for htp/1.1
- on same TCP connection: server, parses request, responds, parses new request,..
- client sends requests for all referenced objects as soon as it receives base HTML.
- fewer RTTs, less slow start.

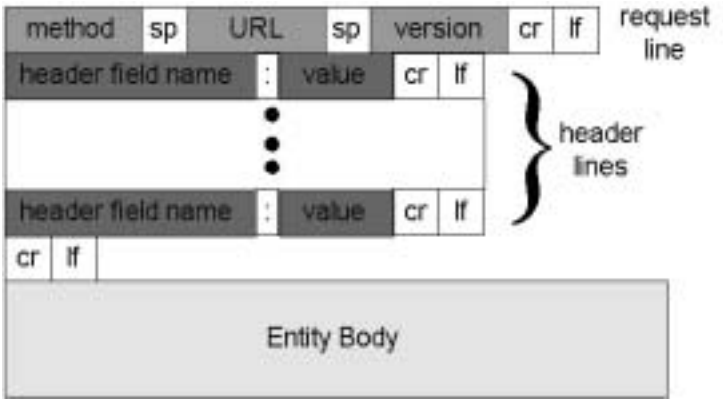
http message format: request

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



67

http request message: general format



68

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 codes

In first line in server->client response message.

A few sample codes:

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

Trying out http (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet www.eurecom.fr 80
```

Opens TCP connection to port 80 (default http server port) at www.eurecom.fr. Anything typed in sent to port 80 at www.eurecom.fr

2. Type in a GET http request:

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

By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to http server

3. Look at response message sent by http server!

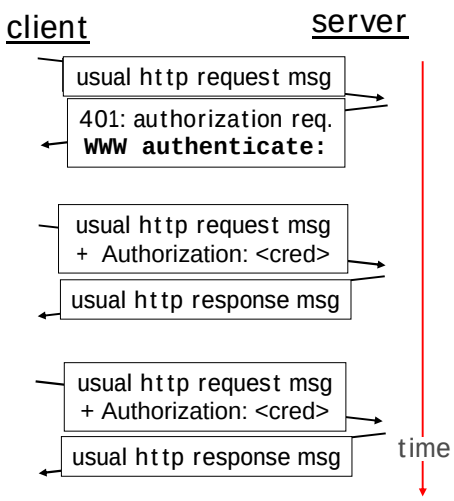
User-server interaction: authentication

Authentication : control access to server content

- authorization credentials: typically name, password
- stateless: client must present authorization in *each* request
  - authorization: header line in each request
  - if no authorization: header, server refuses access, sends

    `WWW authenticate:`

    header line in response

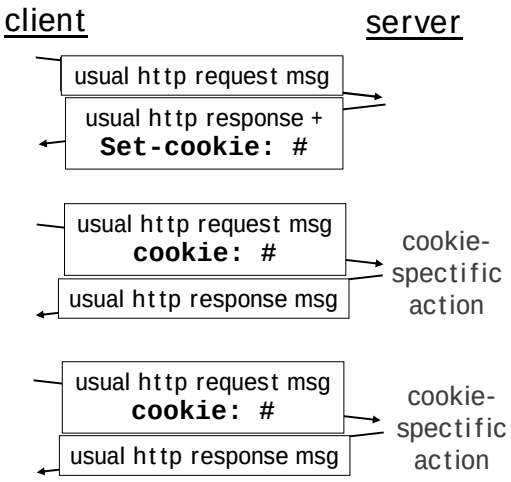


Cookies: keeping "state"

- server-generated # , server-remembered #, later used for:
  - authentication
  - remembering user preferences, previous choices
- server sends "cookie" to client in response msg

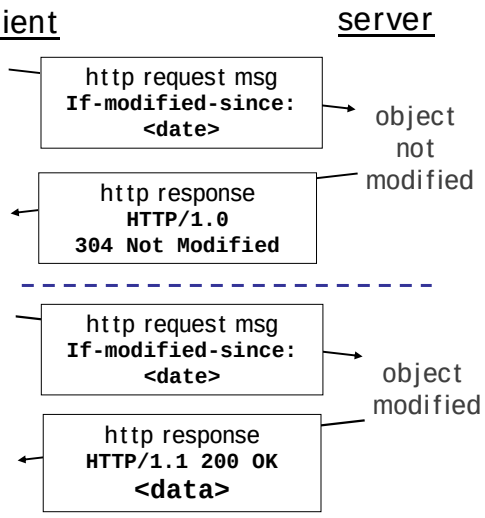
  `Set-cookie: 1678453`
- client presents cookie in later requests

  `cookie: 1678453`



Conditional GET: client-side caching

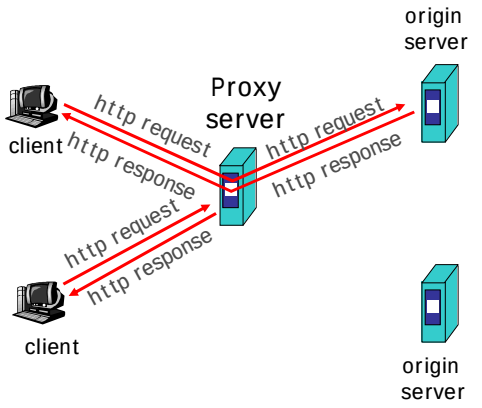
- Goal: don't send object if client has up-to-date cached version
- client: specify date of cached copy in http request
  `If-modified-since: <date>`
- server: response contains no object if cached copy is up-to-date:
  `HTTP/1.0 304 Not Modified`



Web Caches (proxy server)

Goal: satisfy client request without involving origin server

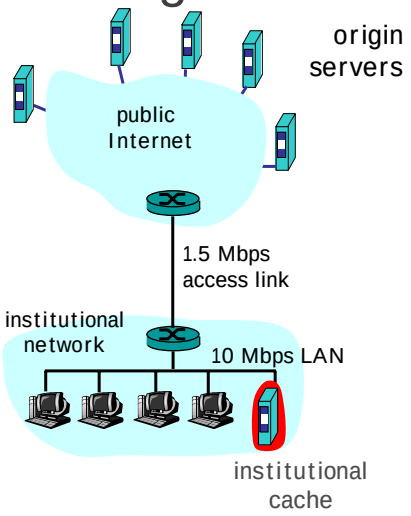
- user sets browser: Web accesses via web cache
- client sends all http requests to web cache
  - object in web cache: web cache returns object
  - else web cache requests object from origin server, then returns object to client



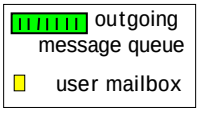
Why Web Caching?

Assume: cache is "close" to client (e.g., in same network)

- smaller response time: cache "closer" to client
- decrease traffic to distant servers
  - link out of institutional/local ISP network often bottleneck



Electronic Mail

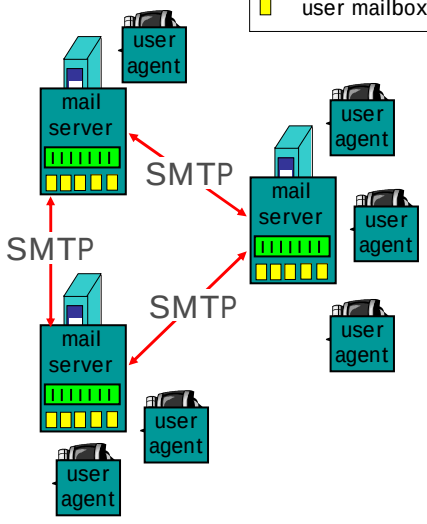


Three major components:

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

User Agent

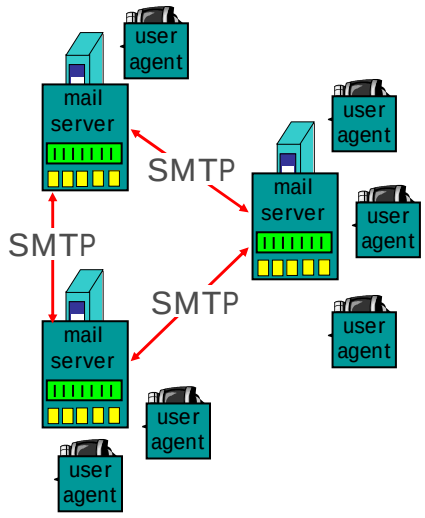
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Netscape Messenger
- outgoing, incoming messages stored on server



Electronic Mail: mail servers

Mail Servers

- mailbox contains incoming messages (yet to be read) for user
- message queue of outgoing (to be sent) mail messages
- smtp protocol between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server



Electronic Mail: smtp [RFC 821]

- uses tcp to reliably transfer email msg from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCII

79

Sample smtp interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C:   How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

80

Try smtp interaction for yourself:

- **`telnet servername 25`**
- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

above lets you send email without using email client (reader)

smtp: final words

- smtp uses persistent connections
- smtp requires message (header & body) to be in 7-bit ASCII
- certain character strings not permitted in msg (e.g., `CRLF.CRLF`). Thus msg has to be encoded (usually into either base-64 or quoted printable)
- smtp server uses `CRLF.CRLF` to determine end of message

Comparison with http:

- http: pull
- email: push
- both have ASCII command/response interaction, status codes
- http: each object encapsulated in its own response msg
- smtp: multiple objects sent in multipart msg

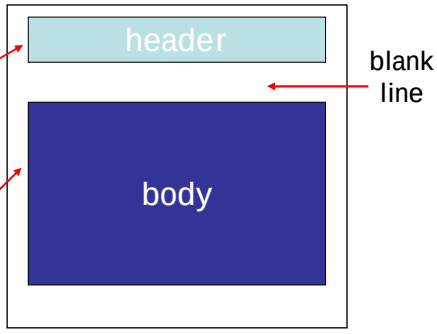
Mail message format

smtp: protocol for exchanging email msgs

RFC 822: standard for text message format:

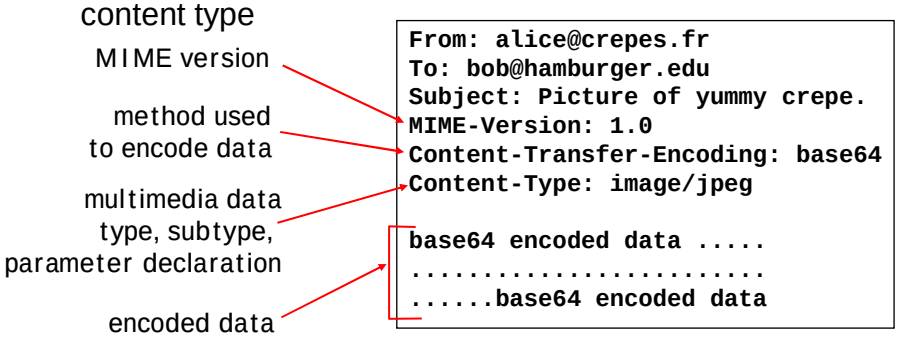
- header lines, e.g.,
  - To:
  - From:
  - Subject:

  different from smtp commands!
- body
  - the "message", ASCII characters only



Message format: multimedia extensions

- MIME: multimedia mail extension, RFC 2045, 2056
- additional lines in msg header declare MIME content type



MIME types

`Content-Type: type/subtype; parameters`

Text

- example subtypes: **`plain, html`**

Image

- example subtypes: **`jpeg, gif`**

Audio

- exampe subtypes: **`basic`** (8-bit mu-law encoded), **`32kadpcm`** (32 kbps

Video

- example subtypes: **`mpeg, quicktime`**

Application

- other data that must be processed by reader before “viewable”
- example subtypes: **`msword, octet-stream`**

Multipart Type

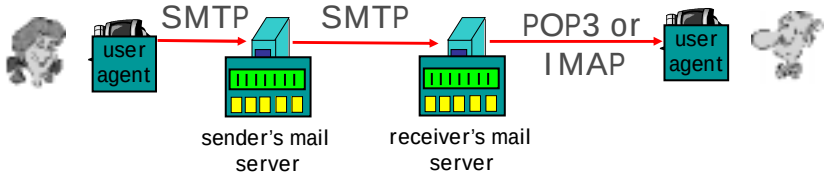
```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=98766789

--98766789
Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain

Dear Bob,
Please find a picture of a crepe.
--98766789
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .....
.........................
......base64 encoded data
--98766789--
```


Mail access protocols



- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
    - authorization (agent <-->server) and download
  - IMAP: Internet Mail Access Protocol [RFC 1730]
    - more features (more complex)
    - manipulation of stored msgs on server
  - HTTP: Hotmail , Yahoo! Mail, etc.

POP3 protocol

authorization phase

- client commands:
  - **`user:`** declare username
  - **`pass:`** password
- server responses
  - **`+OK`**
  - **`-ERR`**

transaction phase, client:

- **`list:`** list message numbers
- **`retr:`** retrieve message by number
- **`dele:`** delete
- **`quit`**

```
S: +OK POP3 server ready
C: user alice
S: +OK
C: pass hungry
S: +OK user successfully logged on
```

```
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```


DNS: Domain Name System

People: many identifiers:

- SSN, name, passport #

Internet hosts, routers:

- IP address (32 bit) - used for addressing datagrams
- "name", e.g., gaia.cs.umass.edu - used by humans

Q: map between IP addresses and name ?

Domain Name System:

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol* host, routers, name servers to communicate to *resolve* names (address/name translation)
  - note: core Internet function, implemented as application-layer protocol
  - complexity at network's "edge"

DNS name servers

Why not centralize DNS?

- single point of failure
- traffic volume
- distant centralized database
- maintenance

doesn't *scale!*

- no server has all name-to-IP address mappings

local name servers:

- each ISP, company has *local (default) name server*
- host DNS query first goes to local name server

authoritative name server:

- for a host: stores that host's IP address, name
- can perform name/address translation for that host's name

DNS: Root name servers

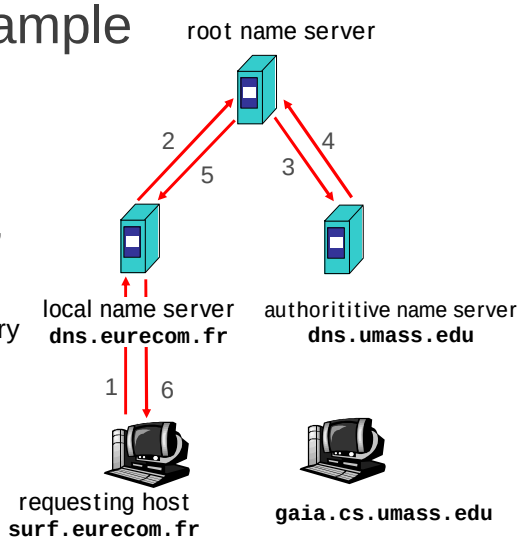
- contacted by local name server that can not resolve name
- root name server:
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server

13 root name servers worldwide

Simple DNS example

host **`surf.eurecom.fr`** wants IP address of **`gaia.cs.umass.edu`**

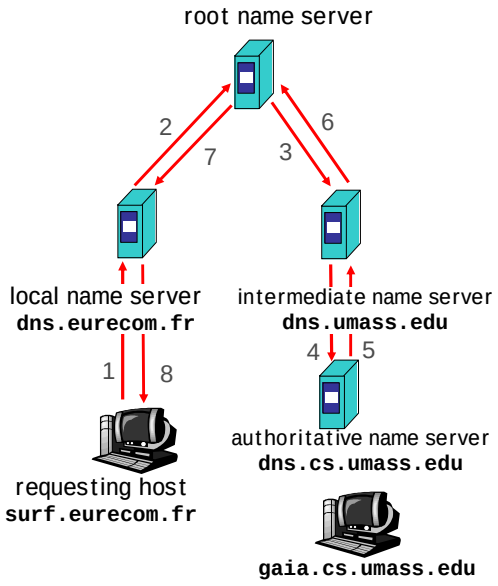
1. contacts its local DNS server, **`dns.eurecom.fr`**
2. **`dns.eurecom.fr`** contacts root name server, if necessary
3. root name server contacts authoritative name server, **`dns.umass.edu`**, if necessary



DNS example

Root name server:

- may not know authoritative name server
- may know *intermediate name server:* who to contact to find authoritative name server



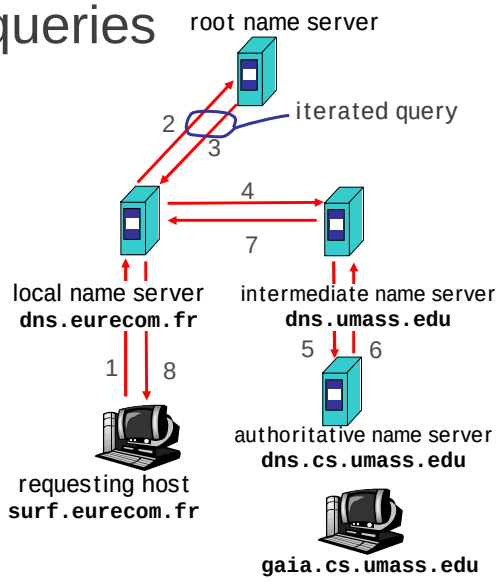
DNS: iterated queries

recursive query:

- puts burden of name resolution on contacted name server
- heavy load?

iterated query:

- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS: caching and updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time
- update/notify mechanisms under design by IETF
  - RFC 2136
  - http://www.ietf.org/html.charters/dnsind-charter.html

DNS records

DNS: distributed db storing resource records (RR)

RR format: **(name, value, type,ttl)**

- Type=A
  - **name** is hostname
  - **value** is IP address
- Type=NS
  - **name** is domain (e.g. foo.com)
  - **value** is IP address of authoritative name server for this domain
- Type=CNAME
  - **name** is alias name for some "cannonical" (the real) name

    `www.ibm.com` is really

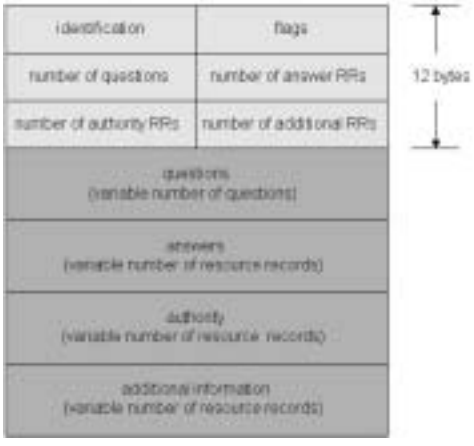
    `servereast.backup2.ibm.com`
  - **value** is cannonical name
- Type=MX
  - **value** is name of mailserver associated with **name**

DNS protocol, messages

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

msg header

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



97

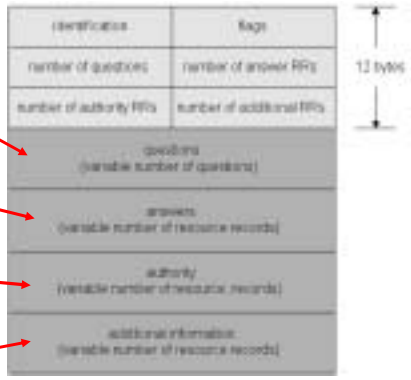
DNS protocol, messages

Name, type fields for a query

RRs in reponse to query

records for authoritative servers

additional “helpful” info that may be used



98

Any Questions?

See you next time.