RW354
Principles of Computer Networking

A.E. Krzesinski and B.A. Bagula
Department of Computer Science
University of Stellenbosch

Last updated: 15 September 2004

The material presented in these slides is used with permission from

- *Larry L. Peterson and Bruce S. Davie. Computer Networks: A Systems Approach (Second Edition). Morgan Kaufmann Publishers. ISBN 1-55860-577-0.*
- *William Stallings. Data and Computer Communications (Sixth Edition). Prentice-Hall Inc. ISBN 0-13-571274-2.*
- *Andrew S. Tannenbaum. Computer Networks (Fourth Edition). Prentice Hall Inc. ISBN 0-13-349945-6.*

End-to-End Protocols

The underlying best-effort network

- *drops packets*
- *re-orders packets*
- *delivers duplicate copies of a packet*
- *limits packets to some finite size*
- *delivers packets after an arbitrarily long delay.*

Mechanisms are needed to turn the less-than-desirable properties of the best-effort network into the high level of service required by application programs.

End-to-End Protocols

A transport protocol provides (among others) the following end-to-end services

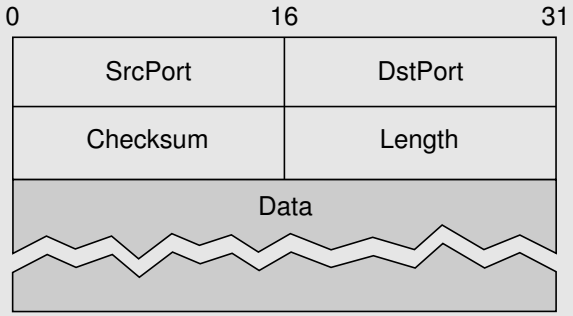
- *guaranteed packet delivery*
- *packets are delivered in the same order they are sent*
- *at most one copy of each packet is delivered*
- *arbitrarily large messages are supported*
- *synchronization is supported*
- *the receiver can apply flow control to the sender*
- *multiple application processes on each host are supported.*

The User Datagram Protocol

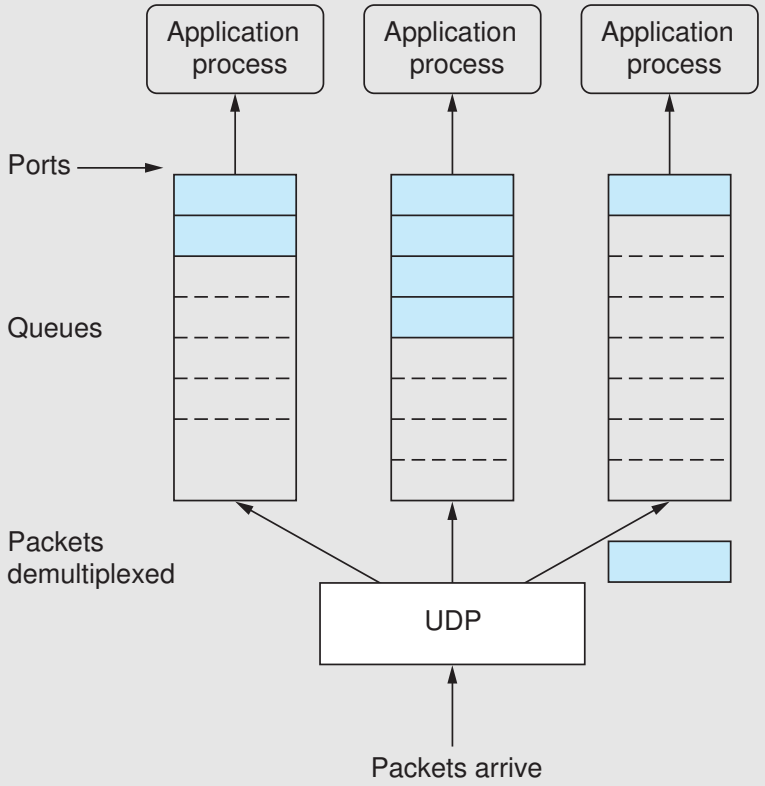
Summary	*UDP provides a low-overhead transport service for application protocols that do not need (or cannot use) the connection-oriented services offered by TCP. UDP is most often used by applications that make heavy use of broadcasts or multicasts, as well as applications that need fast turnaround times on lookups or queries.*	
Protocol ID	*17*	
Relevant STD's	*2*	
	3	*includes RFCs 1122 & 1123*
	6	*RFC 768, re-published*
Relevant RFCs	*768*	*User Datagram Protocol*
	1122	*Host Network Requirements*

UDP: Overview

- *connectionless*
- *lightweight, unreliable, unordered datagram service*
- *no flow control*
- *adds multiplexing*
  - *the endpoints are identified by ports*
  - *servers have well-known ports: see* `/etc/services` *on Unix*
- *optional checksum*
  - *pseudo header + udp header + data*
- *header format*



UDP: Ports are implemented by message queues

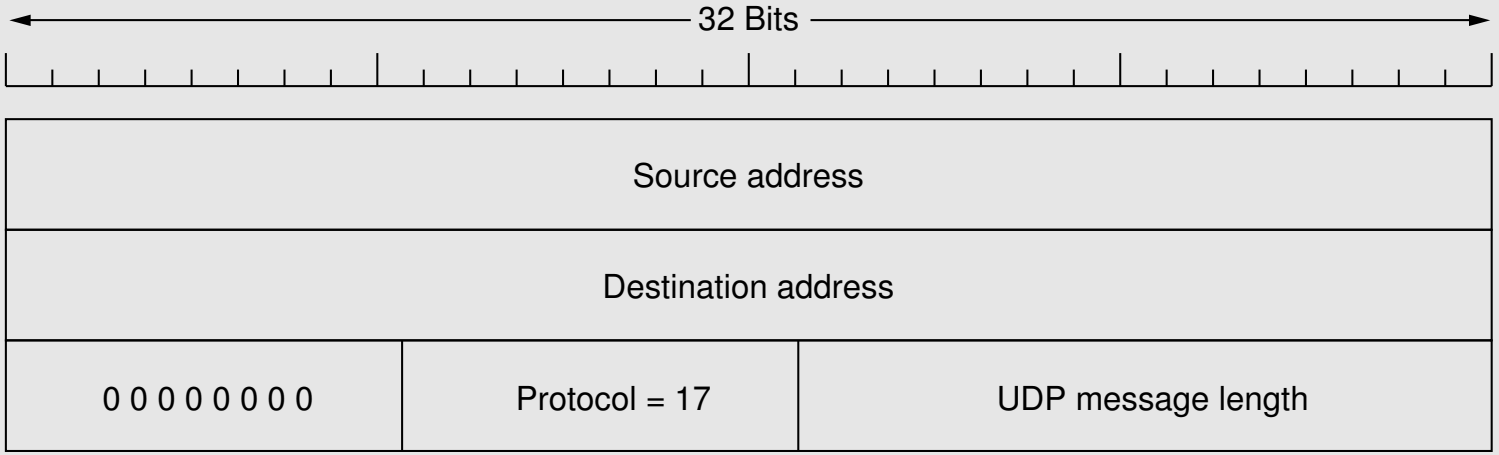


- *the message goes to the end of the appropriate queue*
- *if the queue is full the message is dropped*
- *an application retrieves messages from the front of the appropriate queue: if the queue is empty the process blocks until a message arrives.*

UDP: Psuedo Header

UDP can use a checksum to ensure correct delivery of the message. The UDP checksum is optional in IPv4 and compulsory in IPv6. There is no IP checksum in IPv6.

The UDP checksum is computed over the UDP header, the UDP body and the psuedo header.



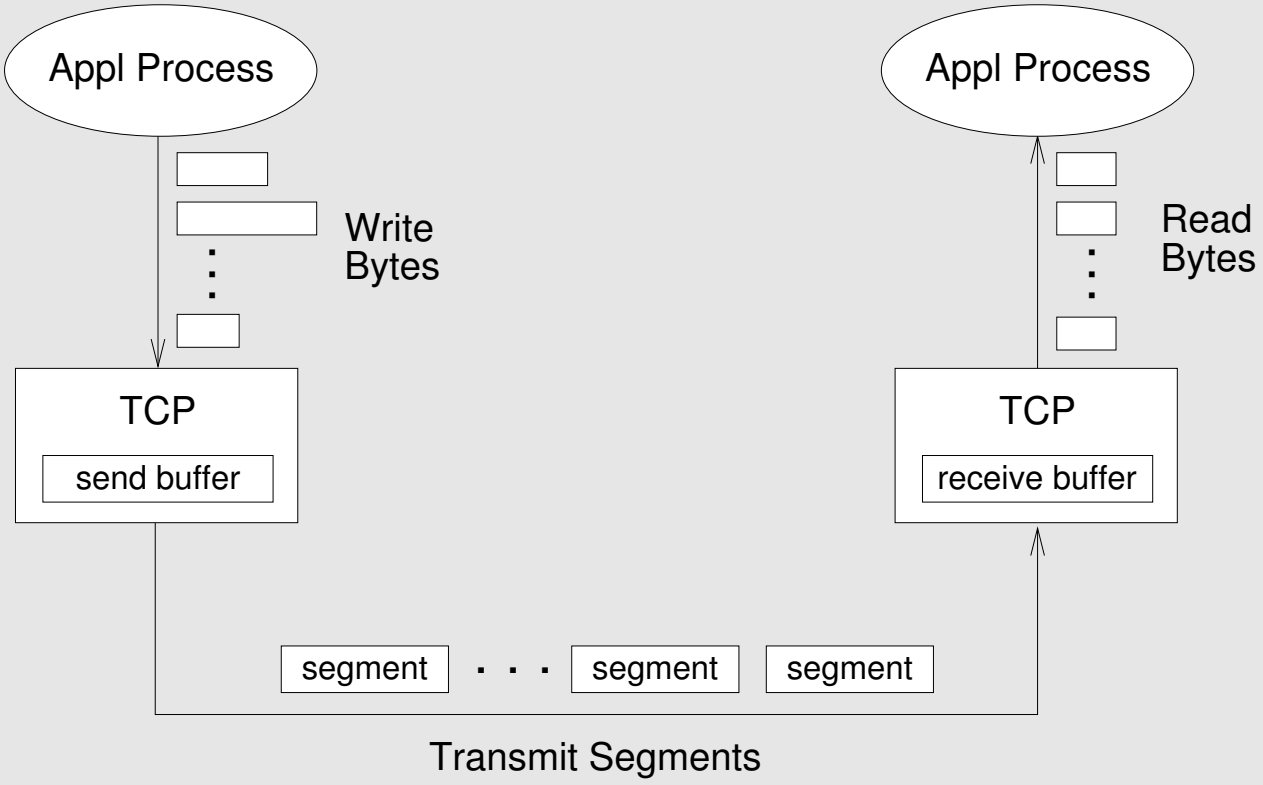
If the IP packet was delivered to the wrong address – the UDP checksum will detect this.

The Transmission Control Protocol

Summary	*TCP provides a reliable, connection-oriented transport protocol for transaction-oriented applications to use.*	
Protocol ID	*6*	
Relevant STD's	*2*	
	3	*includes RFCs 1122 & 1123*
	7	*RFC 793, re-published*
Relevant RFCs	*793*	*TCP*
	896	*The Nagle Algorithm*
	1122	*Host Network Requirements*
	1323	*Window Scale and Timestamp*
	2018	*Selective Acknowledgments*
	2581	*TCP Congestion Control*
	2113	*Router Alert Option*
Related RFCs	*1072*	*Extensions for High Delay*
	1106	*Negative Acknowledgments*
	1146	*Alternate Checksums*
	1337	*Observations on RFC 1323*
	1644	*Transaction Extensions*
	1948	*Defending Against Sequence Number Attacks*
	2414	*Increasing the Initial Window*
	2525	*Known TCP Implementation Problems*
	2582	*Experimental New Reno Modifications to Fast Recovery*

TCP: Overview

- *connection-oriented*
- *byte-stream*
  - *sending process writes some number of bytes*
  - *TCP breaks into segments & sends via IP*
  - *receiving process reads some number of bytes.*



TCP: Overview

The majority of Internet applications use TCP

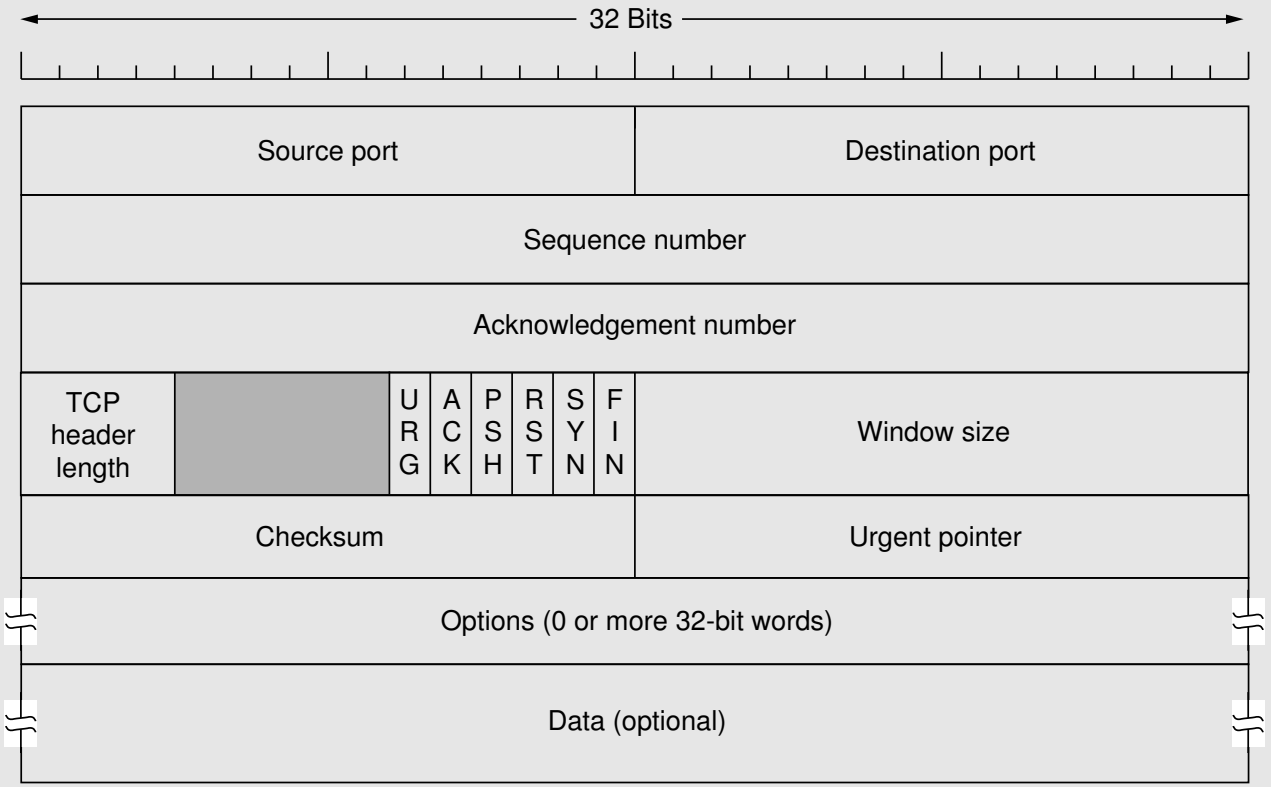
- *reliable: error detection & correction*
- *fully duplex: two byte streams, one flowing in each direction*
- *flow control: keep the sender from overrunning the receiver*
- *congestion control: keep the sender from overrunning the network.*

A TCP connection is NOT a virtual circuit.

TCP: End-to-End Issues

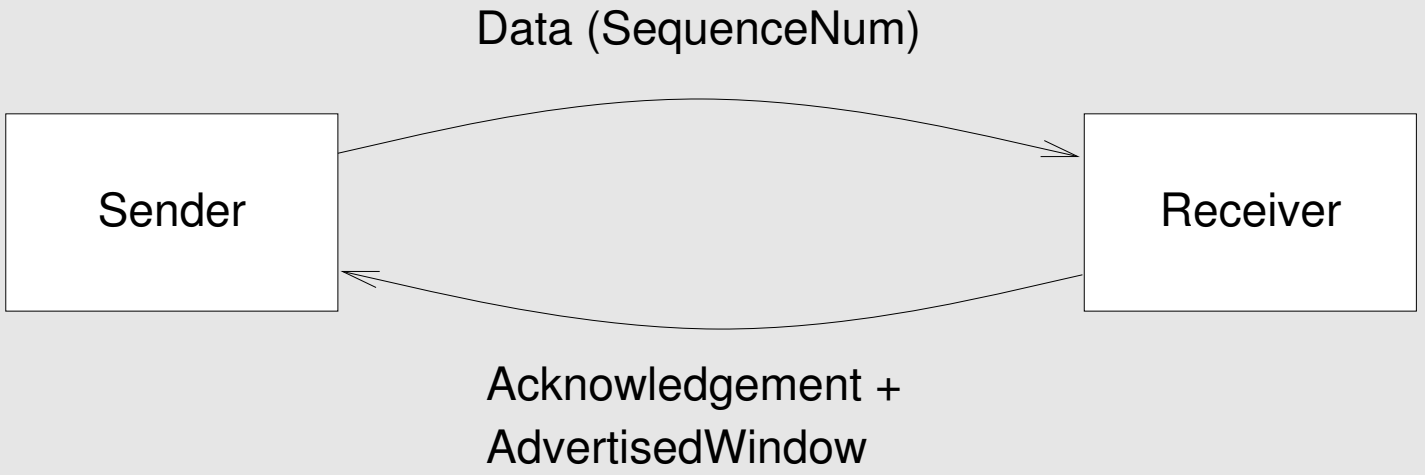
- *Potentially connects many different hosts*
  - *needs explicit connection/termination.*
- *Potentially different RTTs*
  - *needs an adaptive timeout mechanism.*
- *Potentially long delays in network*
  - *needs to re-order packets*
  - *very old packets can arrive.*
- *Potentially different capacity at the destination*
  - *needs to allocate different amounts of buffering.*
- *Potentially different network capacity*
  - *needs to be prepared for network congestion.*

TCP: Segment Format



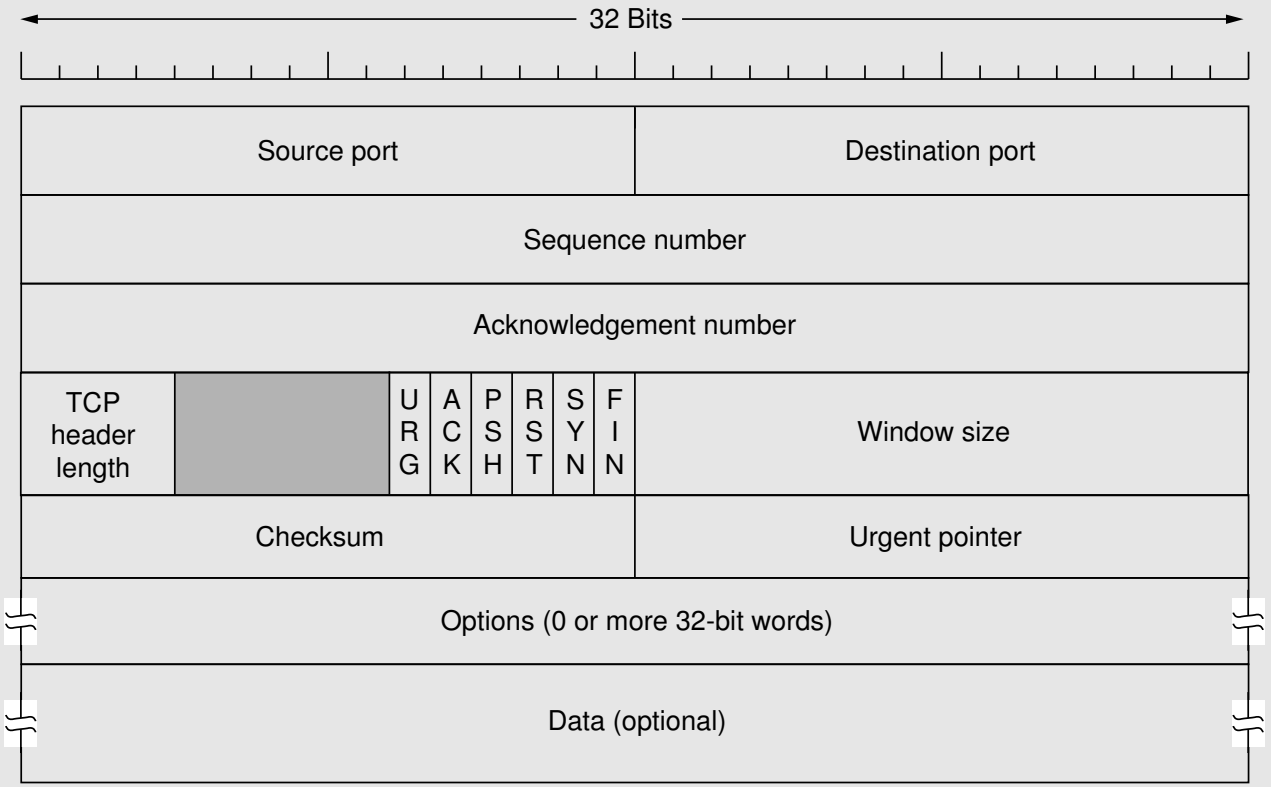
- *Each connection identified by a 4-tuple* (`SrcPort`, `SrcIPAddr`, `DstPort`, `DstIPAddr`)

TCP: Sequence Numbers



- `SequenceNum`
  - *the sequence number of the first byte in this segment*
- `Acknowledgment`
  - *the sequence number of the last byte successfully received*
- `AdvertisedWindow`
  - *the number of bytes in transit, unacknowledged.*

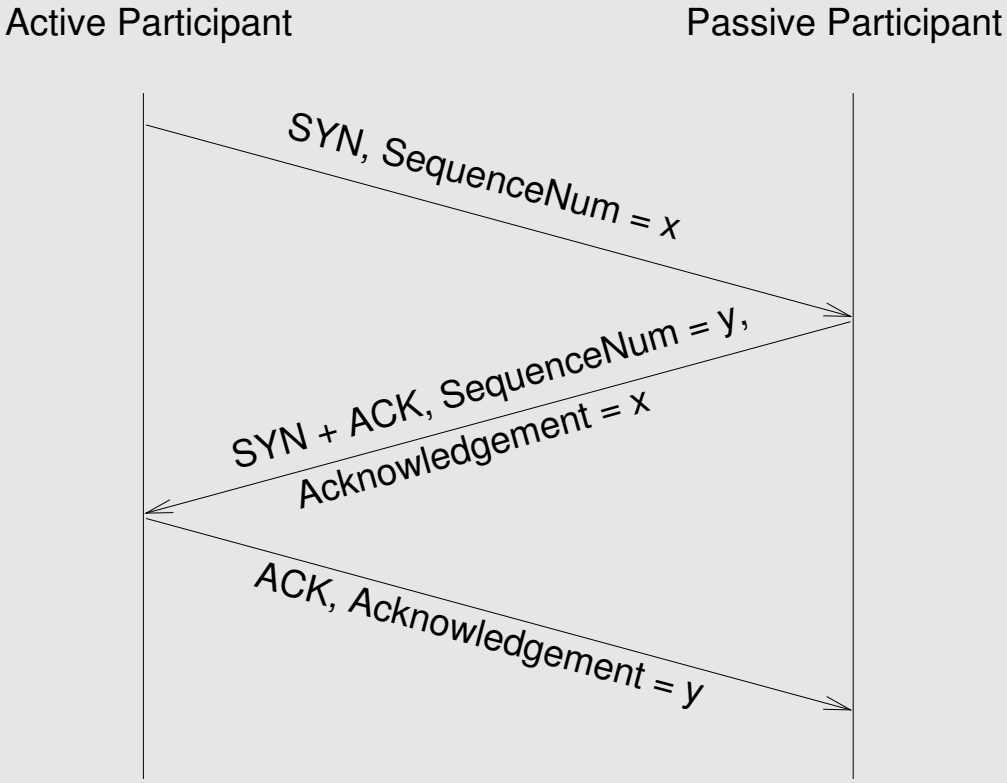
TCP: Segment Format



- *Flags*
  - SYN, FIN, RESET, PUSH, URG, ACK
- *Checksum*
  - *pseudo header + tcp header + data*

TCP: Connection Establishment & Termination

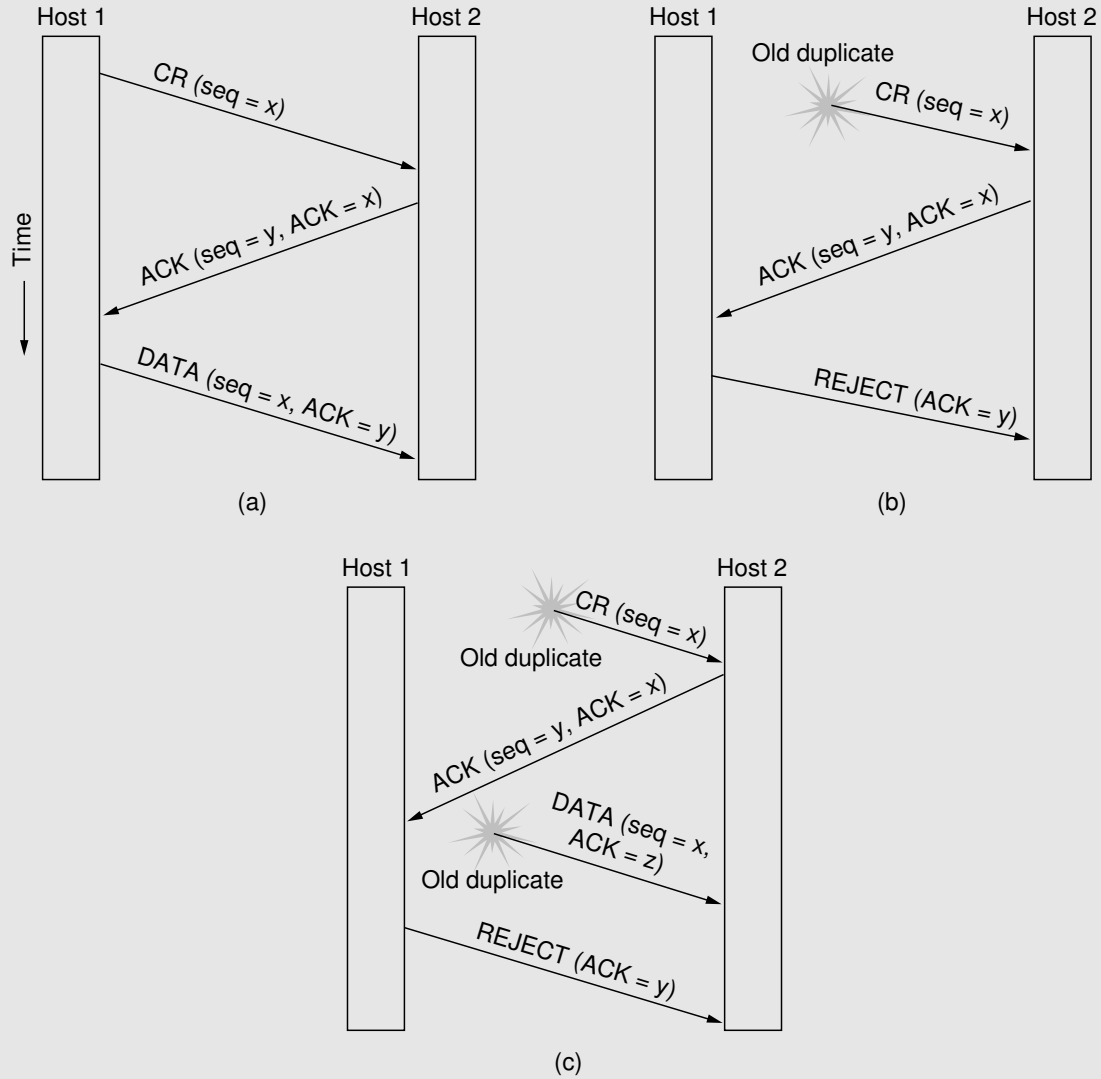
Three-way handshake



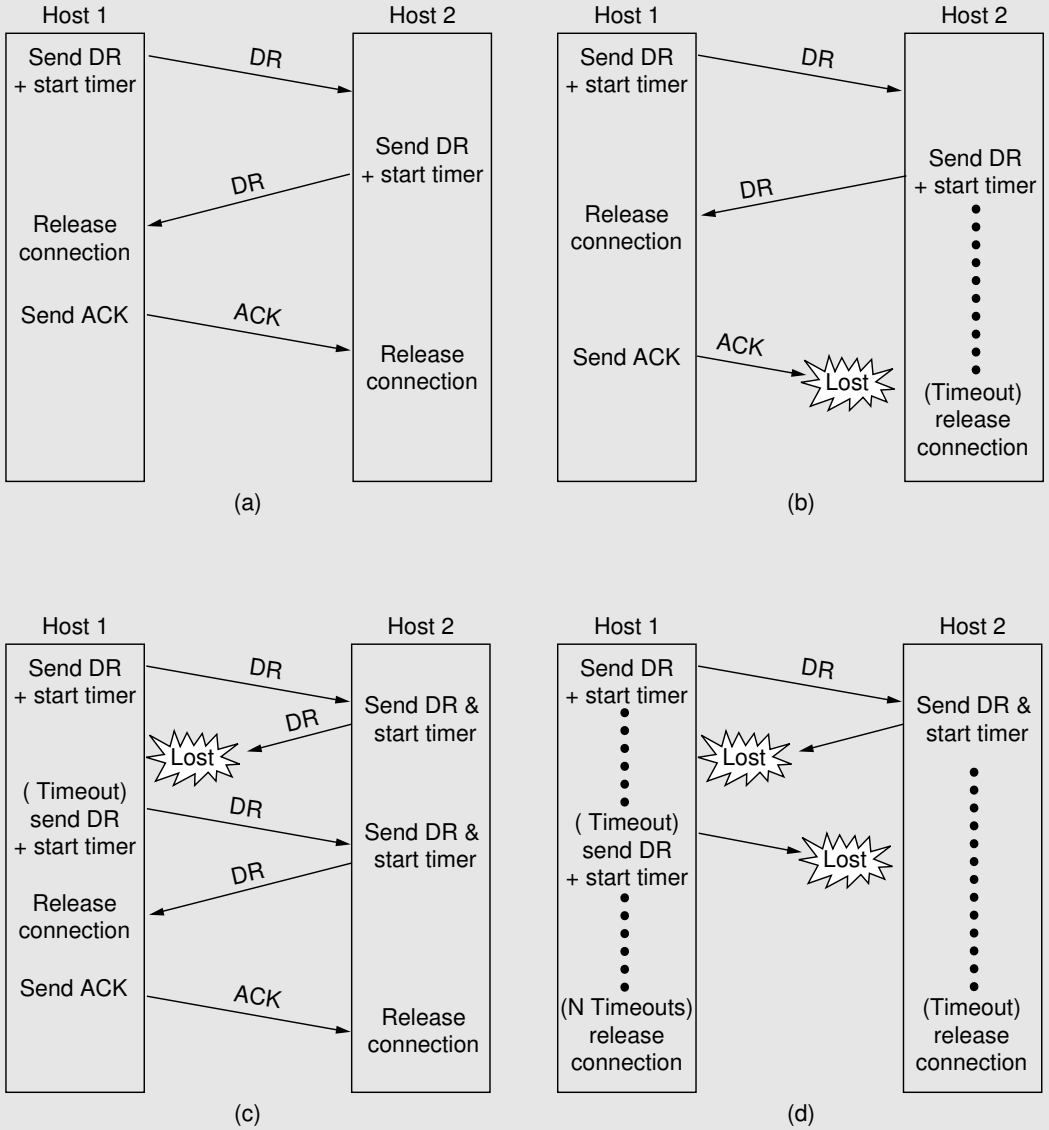
The initial sequence number is chosen at random.

TCP: Connection Establishment

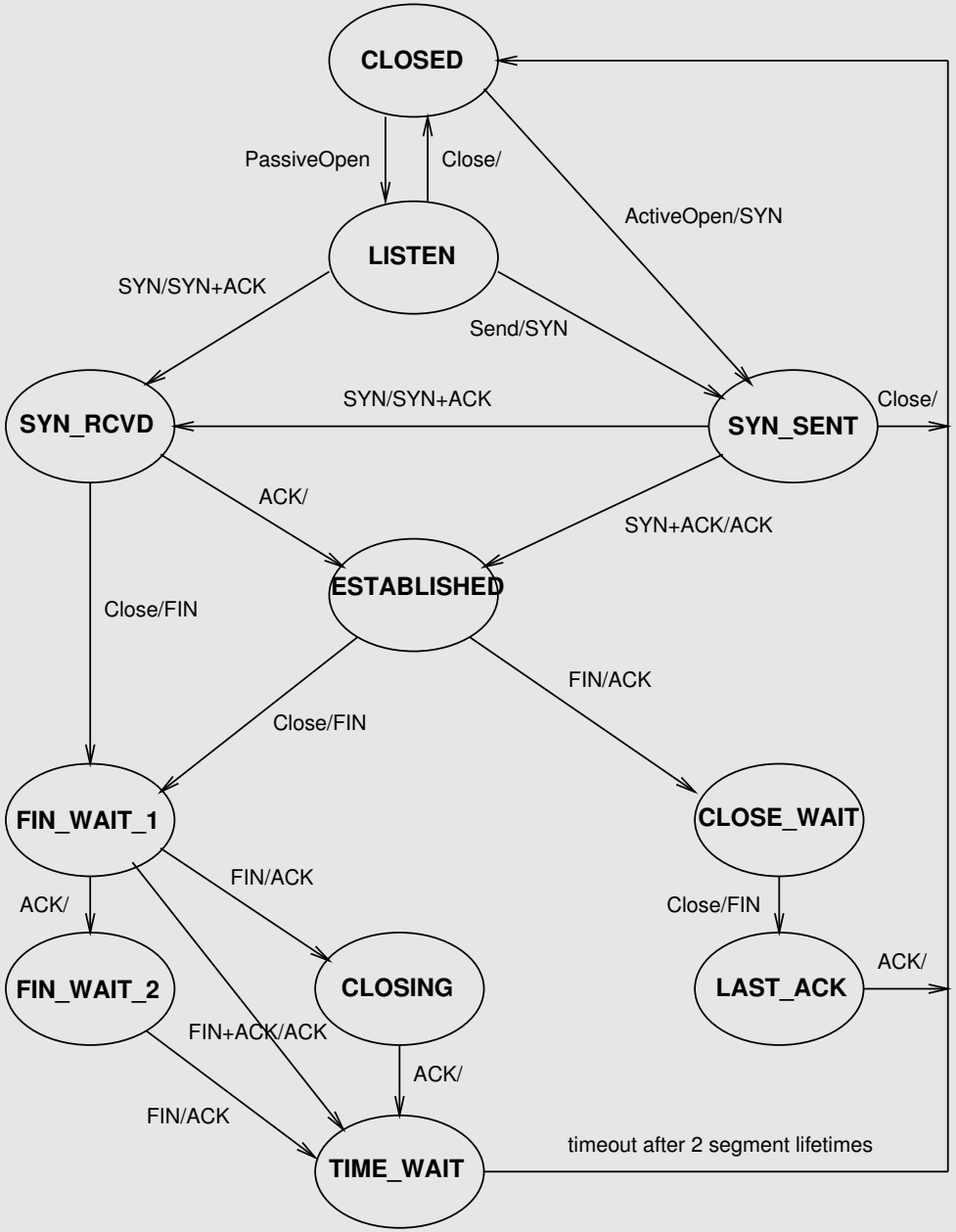
Three-way Handshake



TCP: Connection Termination



TCP: State Transition Diagram



TCP: Path Maximum Transfer Unit Discovery

The maximum size of an IP packet is determined by the MTU of the layer 2 protocol at the source, destination & intermediate networks.

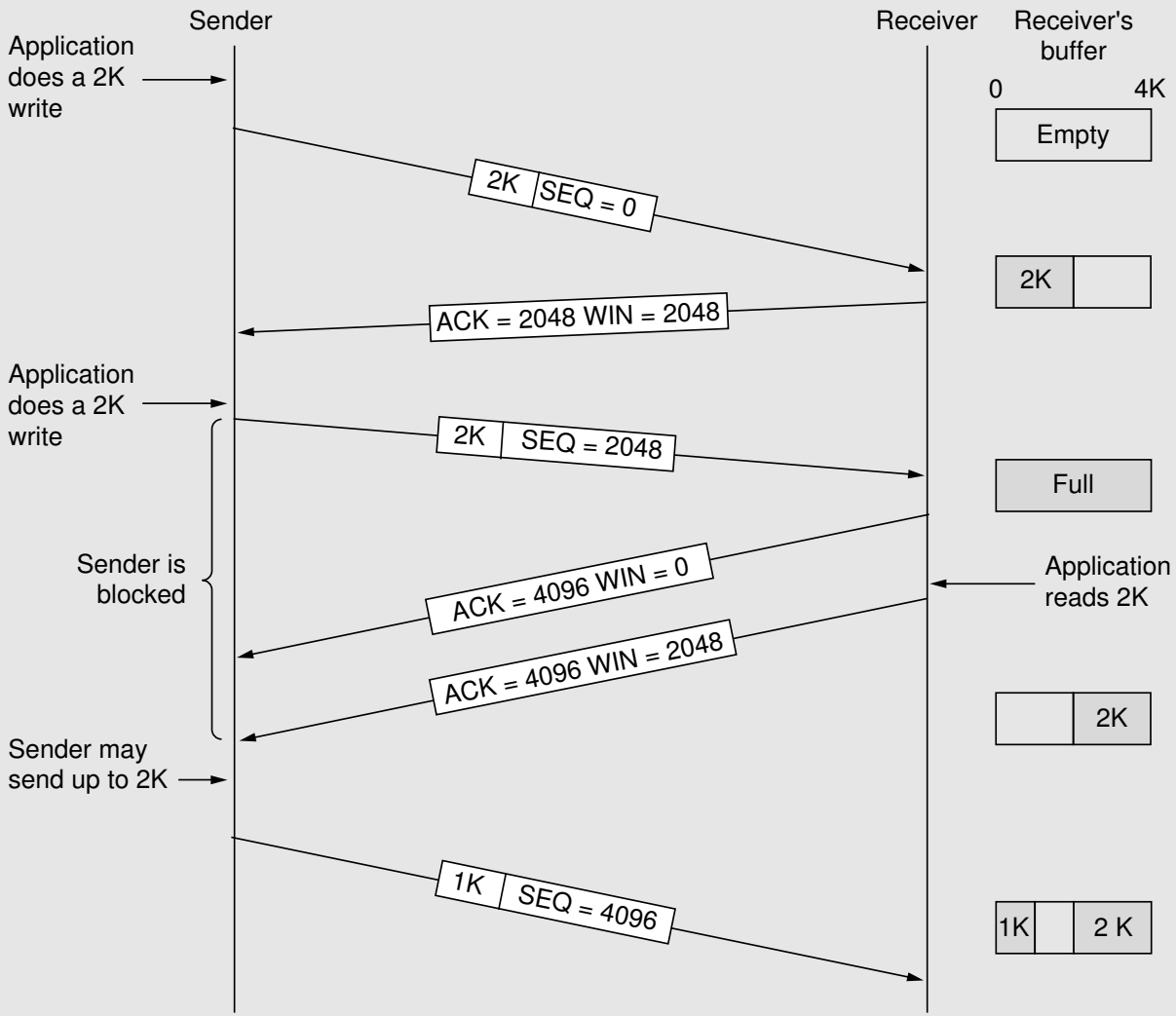
TCP knows the MTU of the endpoint networks, but not of the intermediate networks.

The source sends an IP packet of maximum size (determined by the endpoint MTUs) with the `DontFragment` *flag set*

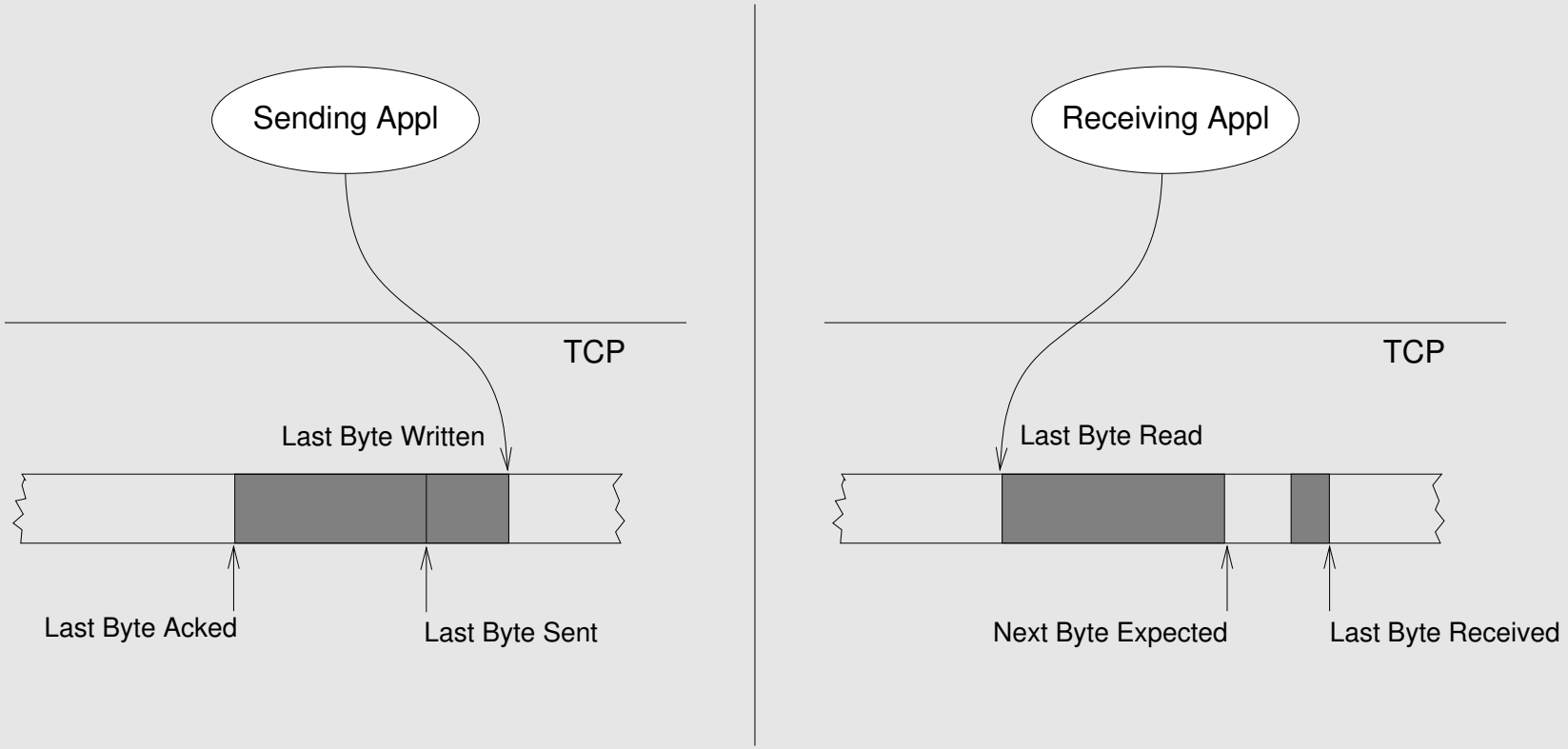
- *if the IP packet is rejected by an intermediate network, a* `CannotFragment` *ICMP error packet is returned to the source*
- *the source sends smaller IP packets until no ICMP packets are returned.*

Problems: some routers & firewalls do not return ICMP packets due to security concerns.

TCP: Window Management

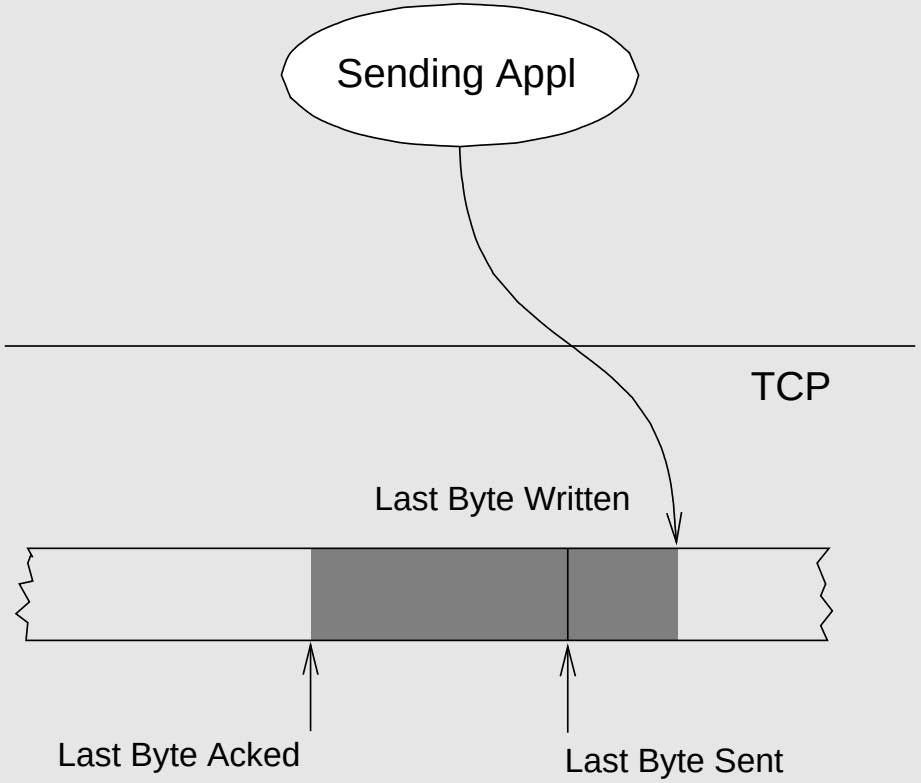


TCP: Sliding Window



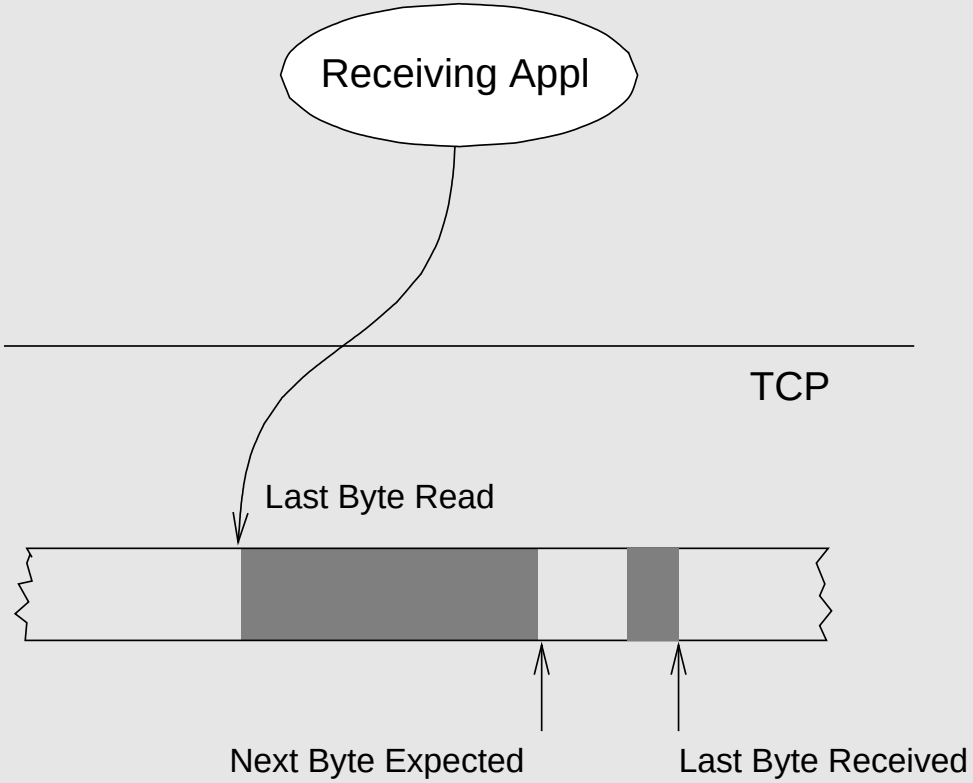
- *each byte has a sequence number*

TCP: Sliding Window Sending Side



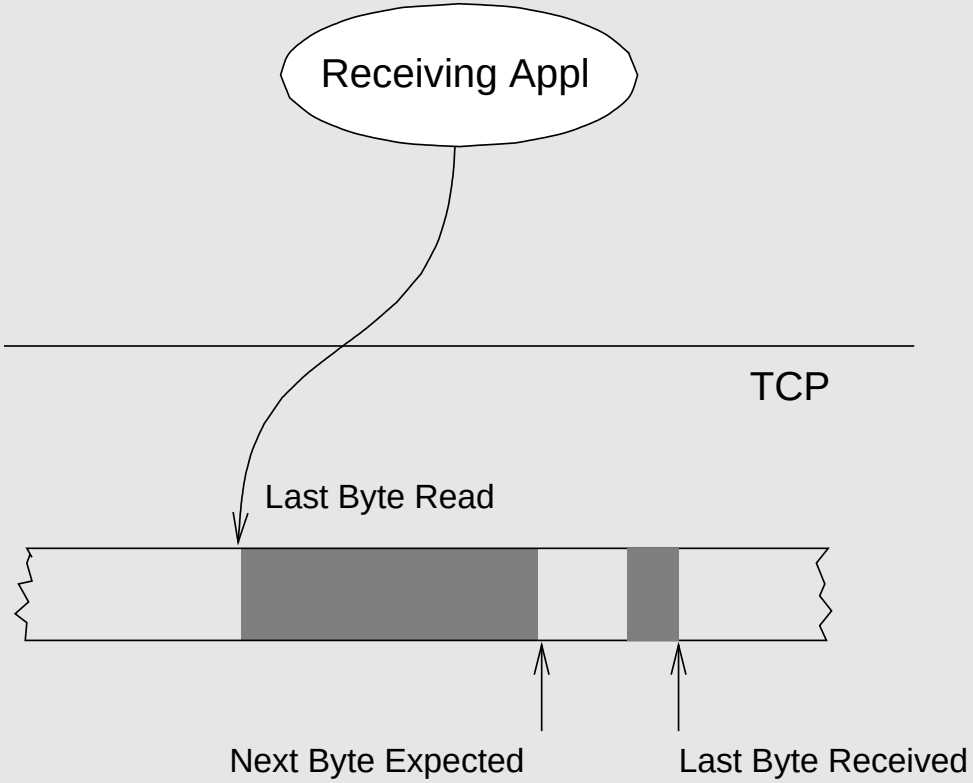
- `LastByteAcked` $\leq$ `LastByteSent`
- `LastByteSent` $\leq$ `LastByteWritten`
- *bytes* $(\texttt{LastByteWritten}, \texttt{LastByteAcked})$ *are buffered.*

TCP: Sliding Window Receiving Side



- $\texttt{LastByteRead} < \texttt{NextByteExpected}$
- $\texttt{NextByteExpected} \leq \texttt{LastByteRcvd} + 1$
- *bytes* $(\texttt{LastByteRcvd}, \texttt{LastByteRead})$ *are buffered.*

TCP: Flow Control Receiving Side

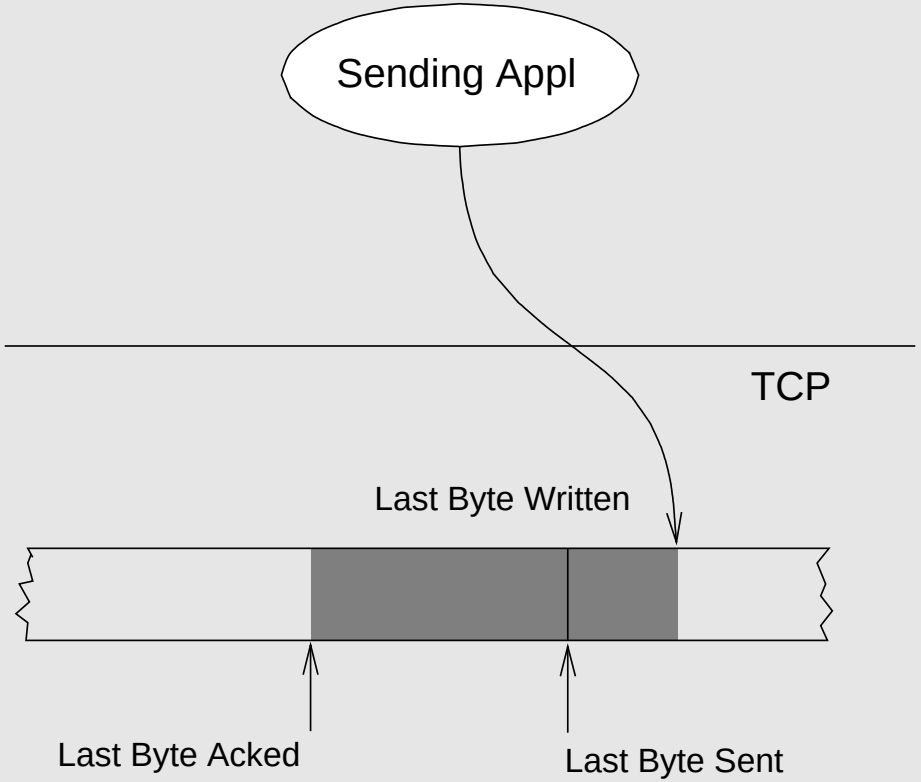


Receive buffer size: `MaxRcvBuffer`

- $(\texttt{LastByteRcvd} - \texttt{LastByteRead}) \leq \texttt{MaxRcvBuffer}$
- $\texttt{AdvertisedWindow} = \texttt{MaxRcvBuffer} - (\texttt{LastByteRcvd} - \texttt{LastByteRead})$

The `AdvertisedWindow` *is communicated to the sender.*

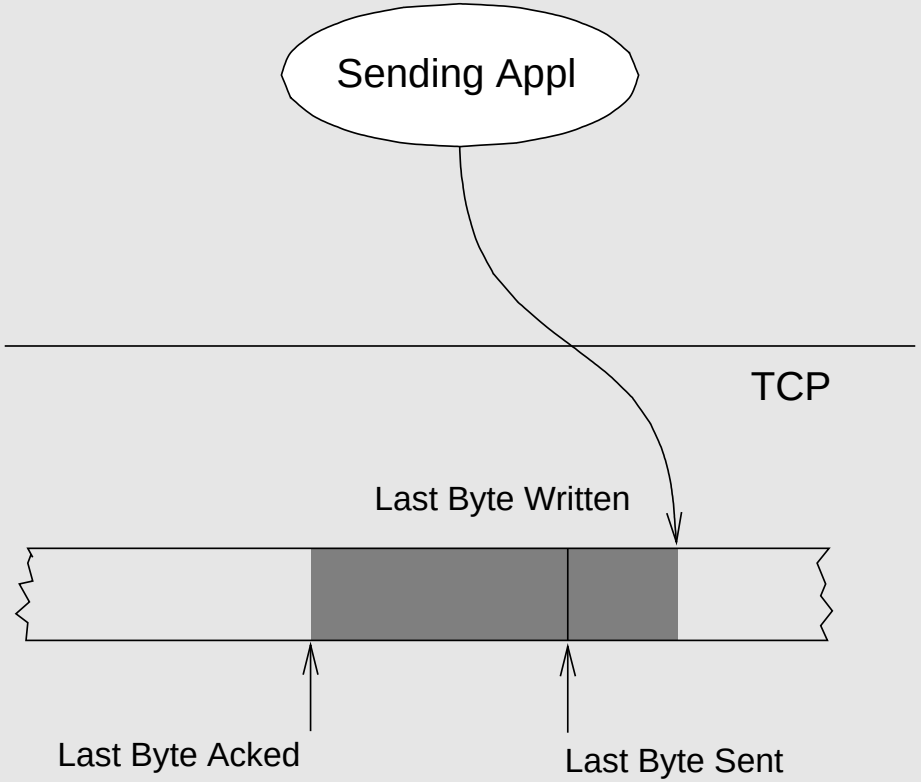
TCP: Flow Control Sending Side



The sender conforms to the `AdvertisedWindow`

- $(\texttt{LastByteSent} - \texttt{LastByteAcked}) \leq \texttt{AdvertisedWindow}$
- $\texttt{EffectiveWindow} = \texttt{AdvertisedWindow} - (\texttt{LastByteSent} - \texttt{LastByteAcked})$

TCP: Flow Control Sending Side



Sender buffer size: `MaxSendBuffer`

- $(\texttt{LastByteWritten} - \texttt{LastByteAcked}) \leq \texttt{MaxSendBuffer}$

Block the sender if

- $(\texttt{LastByteWritten} - \texttt{LastByteAcked}) + \texttt{y} > \texttt{MaxSendBuffer}$

TCP: Flow Control

- *the receiving side always sends an ACK in response to an arriving data segment*
- *the sending side stops sending segments when it receives an ACK with* $\texttt{AdvertisedWindow} = 0$ *except for*
  - *urgent data may be sent*
  - *1-byte probe segments are sent at regular time intervals (exponential backoff, maximum 64 or 128 seconds) until an ACK is received with* $\texttt{AdvertisedWindow} > 0$
  - *the receiver sends an unsolicited ACK as soon as* $\texttt{AdvertisedWindow} > 0$
- *smart sender / dumb receiver.*

TCP: Transmitting Small Segments

Consider a TCP connection to a text editor that reacts to every keystroke

- *the sender sends 1 character: a 41-byte IP packet carries a 21-byte TCP segment*
- *the receiver sends an ACK: a 40-byte IP packet carries a 20-byte TCP segment*
- *the editor reads 1 character, the receiver sends an* `AdvertisedWindow` *window update: a 40-byte IP packet*
- *the editor echoes the character: the receiver sends a 41-byte IP packet.*

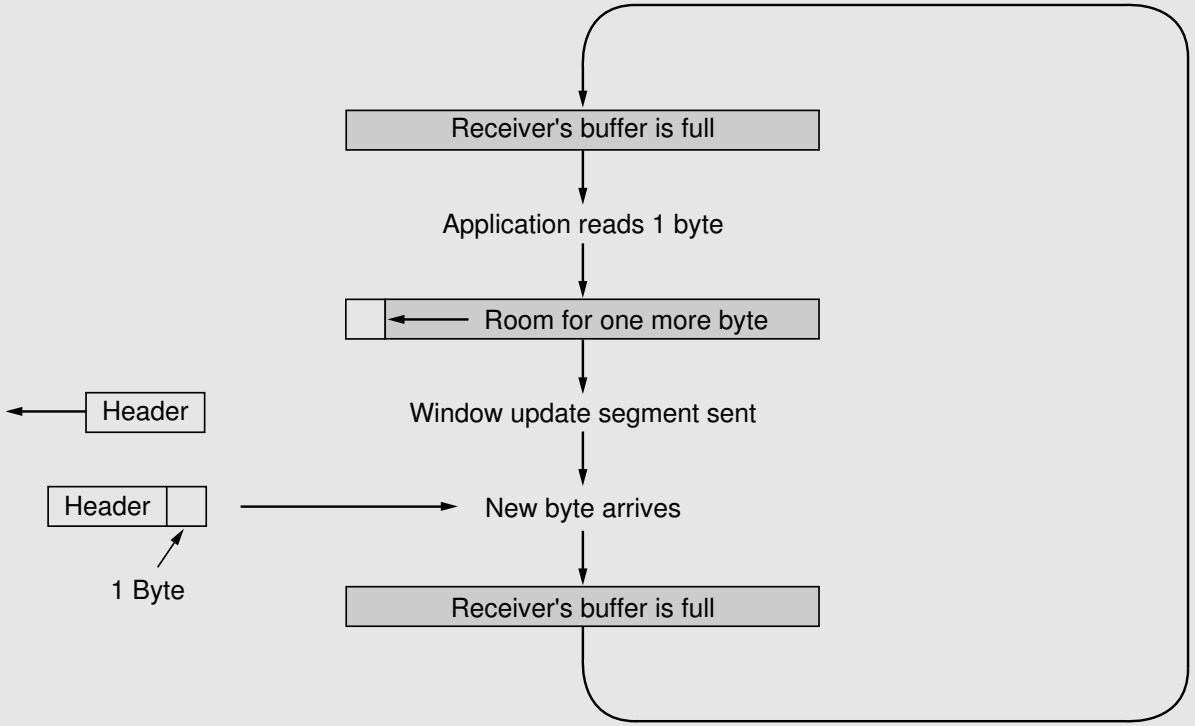
TCP: Nagle's Algorithm RFC 896

Segments smaller than the MSS are delayed until all prior segments have been acknowledged or until a MSS can be sent

- *on a low-latency LAN small segments are sent and ACK'd quickly, allowing another small segment to be sent immediately – effectively eliminating Nagle's algorithm*
- *on a slow LAN ACK's take a long time to be returned – Nagle's algorithm combines multiple writes into a single segment improving network efficiency.*

Disable Nagle's algorithm when TCP sends mouse movements for X-Windows applications.

TCP: Silly Window Syndrome RFC 1122



Clark's solution: the receiver sends a window update when

- *it can accept the advertised MSS, or*
- *its buffer is half empty.*

TCP: Keeping the Pipe Full

The 32-bit `SequenceNum` *can wrap around*

Bandwidth	*Time Until Wrap Around*
T1 (1.5Mbps)	*6.4 hours*
Ethernet (10Mbps)	*57 minutes*
T3 (45Mbps)	*13 minutes*
FDDI (100Mbps)	*6 minutes*
STS-3 (155Mbps)	*4 minutes*
STS-12 (622Mbps)	*55 seconds*
STS-24 (1.2Gbps)	*28 seconds*

TCP options: use a 32-bit timestamp to extend the sequence number space: Protection Against Wrapped Sequence numbers PAWS.

TCP: Keeping the Pipe Full

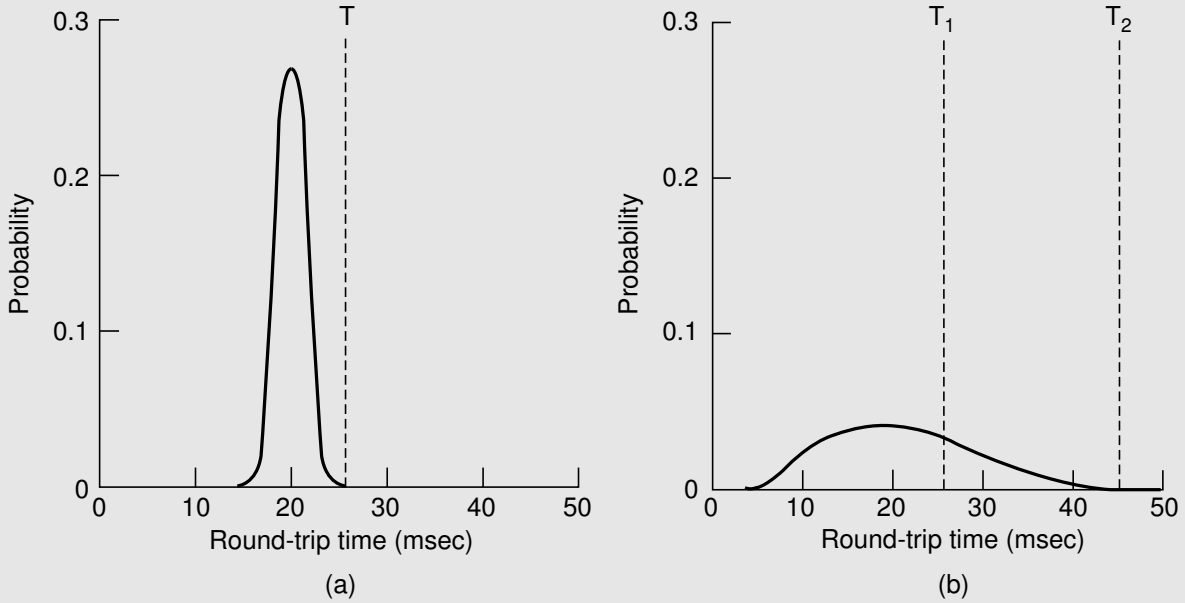
The 16-bit `AdvertisedWindow` *is the number of bytes in transit: 64KB.*

Assume a RTT of 100ms.

Bandwidth	*Delay $\times$ Bandwidth Product*
T1 (1.5Mbps)	*18KB*
Ethernet (10Mbps)	*122KB*
T3 (45Mbps)	*549KB*
FDDI (100Mbps)	*1.2MB*
STS-3 (155Mbps)	*1.8MB*
STS-12 (622Mbps)	*7.4MB*
STS-24 (1.2Gbps)	*14.8MB*

TCP options: scale the advertised window RFC 1323.

TCP: Adaptive Retransmission



Probability density of RTTs for (a) the data link layer (b) TCP.

Estimating the TCP timeout is not easy.

TCP: Adaptive Retransmission

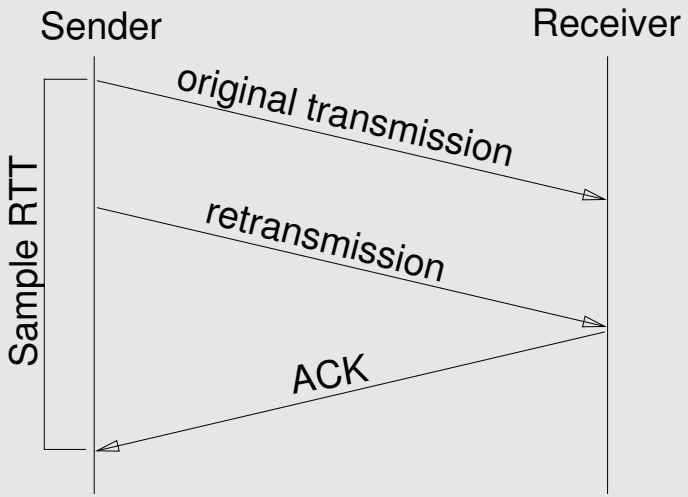
The original timeout calculation

- *measure the* `SampleRTT` *for each segment/ACK pair*
- *compute a weighted average of the RTT*

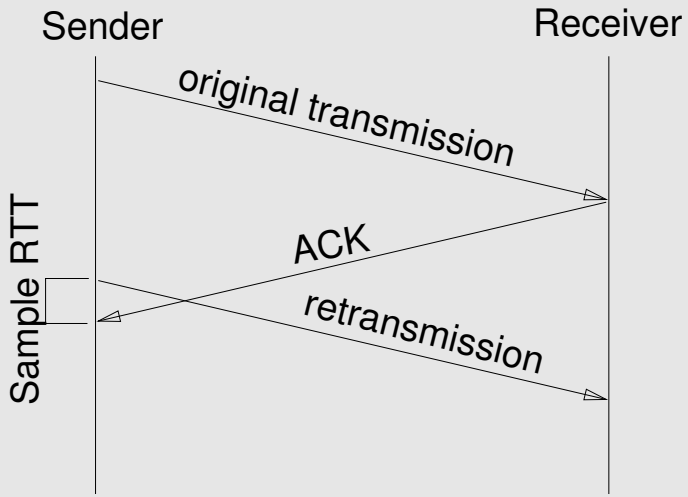
$$\texttt{EstimatedRTT} = \alpha \times \texttt{EstimatedRTT} + \beta \times \texttt{SampleRTT}$$

  - α *is a smoothing factor*
  - $\alpha + \beta = 1$
  - $0.8 \leq \alpha \leq 0.9$
- *set the timeout*
  - $\texttt{TimeOut} = 2 \times \texttt{EstimatedRTT}$

TCP: Karn/Partridge Algorithm



(a) Sample RTT too long



(b) Sample RTT too short

- *do not sample the RTT when re-transmitting*
- *double the timeout after each re-transmission.*

TCP: Karn/Partridge Algorithm

$\texttt{TimeOut} = 2 \times \texttt{EstimatedRTT} \times \texttt{ScaleFactor}$

- *the RTT is measured for each packet that is not re-transmitted*
- *double the timeout after each re-transmission*
  - `ScaleFactor` *is initially set to 1 & is doubled for each re-transmission up to 64 where it remains constant*
  - `Scalefactor` *is reset to 1 when a packet is acknowledged without re-transmission.*

TCP: Karn/Partridge Algorithm

Transmission attempt	Packet number 1	2	3	4	5	6
1	1/1	1/2	2/4	8/1	1/1	1/1
2		2/2	4/8			
3			8/8			

Consider a sequence of six packets where the second & third packets are subject to one & two timeouts respectively.

The first packet is transmitted with scalefactor $C = 1$. The first packet is not re-transmitted: $C = 1$ is passed on to the second packet.

The second packet is first transmitted with $C = 1$ & is re-transmitted with $C = 2$: $C = 2$ is passed on to the third packet.

The third packet is first transmitted with $C = 2$ & then re-transmitted with $C = 4$ & $C = 8$. $C = 2$ is passed on to the fourth packet.

The fourth packet is transmitted with $C = 8$ but, because it is not re-transmitted, $C = 1$ is passed on to the fifth packet.

The fifth & sixth packets are both transmitted with $C = 1$ & are not subject to timeouts hence they pass $C = 1$ on to their successors.

TCP: Calculating running averages

$$\texttt{EstimatedRTT} = \alpha \times \texttt{EstimatedRTT} + \beta \times \texttt{SampleRTT}$$

$$\begin{aligned} E_n &= \frac{\sum_{i=1}^{n} t_i}{n} \\ &= \frac{\sum_{i=1}^{n-1} t_i}{n-1}\frac{n-1}{n} + \frac{t_n}{n} \\ &= \frac{n-1}{n} E_{n-1} + \frac{t_n}{n} \\ &= \alpha E_{n-1} + \beta t_n \end{aligned}$$

where $\alpha = (n-1)/n$, $\beta = 1/n$, $\alpha + \beta = 1$.

TCP: Calculating running averages

Also

$$E_n = \frac{n-1}{n} E_{n-1} + \frac{t_n}{n} = E_{n-1} + \delta(t_n - E_{n-1})$$

where $\delta = 1/n$. *The mean deviation is given by*

$$\begin{aligned}
\sigma_n &= \frac{1}{n-1} \sum_{i=1}^{n} |E_n - t_i| \\
&= \frac{1}{n-2} \sum_{i=1}^{n-1} |E_n - t_i| \, \frac{n-2}{n-1} + \frac{|E_n - t_n|}{n-1} \\
&= \sigma_{n-1} + \Delta(|E_n - t_n| - \sigma_{n-1})
\end{aligned}$$

where $\Delta = 1/(n-1)$. *The mean deviation is a cheap estimator of the standard deviation.*

TCP: Jacobson/Karels Algorithm

- *Compute the average RTT & the mean deviation*

$$\begin{aligned} \texttt{Difference} &= \texttt{SampleRTT} - \texttt{EstimatedRTT} \\ \texttt{EstimatedRTT} &+= \delta \times \texttt{Difference} \\ \texttt{Deviation} &+= \delta(|\texttt{Difference}| - \texttt{Deviation}) \end{aligned}$$

where $0 < \delta \leq 1$.

- *Compute the timeout*

$$\texttt{TimeOut} = \mu \times \texttt{EstimatedRTT} + \phi \times \texttt{Deviation}$$

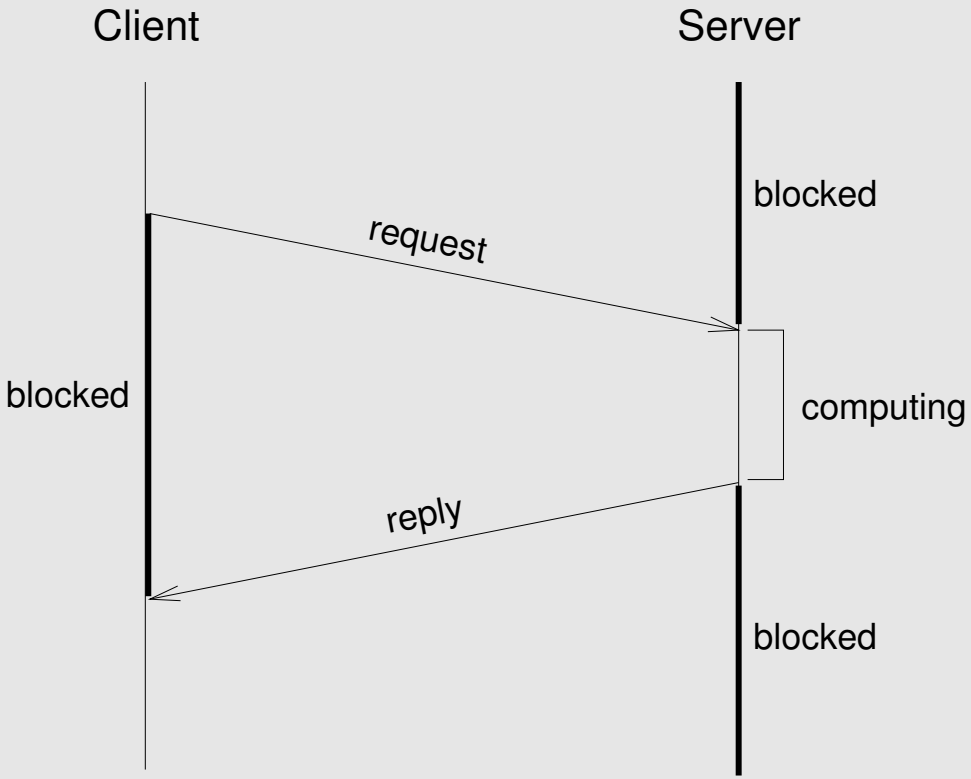
where $\mu = 1$, $\phi = 4$.

Note: an accurate timeout mechanism is important for congestion control (later).

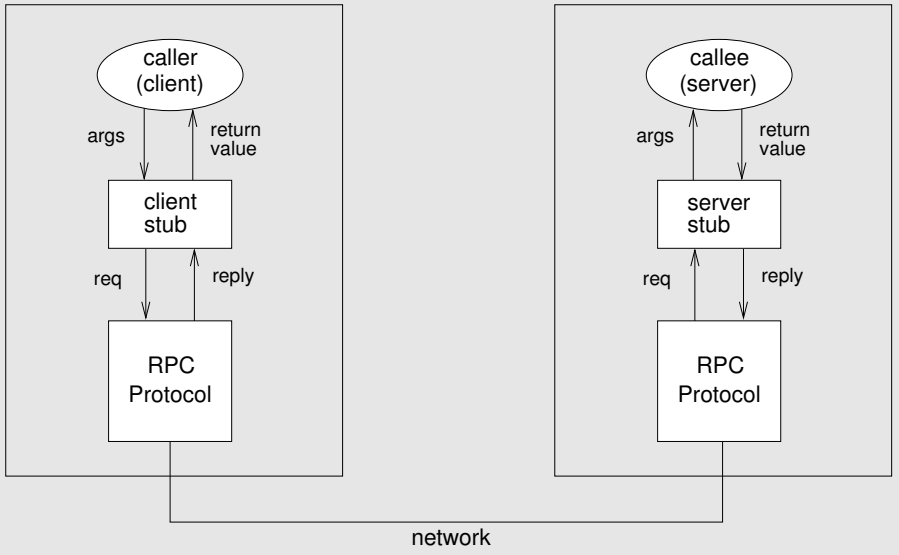
TCP: Extensions

- *Implemented as header options*
- *Store a timestamp in the outgoing segments*
- *Use a 32-bit timestamp to extend the sequence space (PAWS)*
- *Scale the advertised window*

RPC: Overview



RPC: Overview



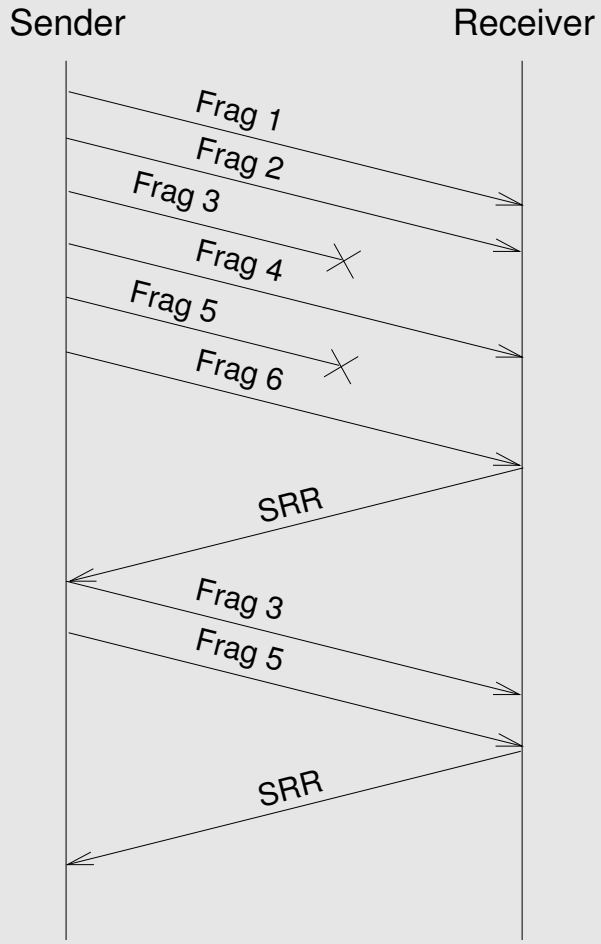
RPC protocol consists of three basic functions

- *BLAST: fragments and reassembles large messages*
- *CHAN: synchronizes request and reply messages*
- *SELECT: dispatches request messages to the correct process*

We consider RPC stubs later.

RPC: Bulk Transfer (BLAST)

Unlike AAL and IP in that it tries to recover from lost fragments; persistent, but does not guarantee delivery. Strategy is to use selective retransmission (partial acknowledgements).



RPC: Bulk Transfer (BLAST)

Sender:

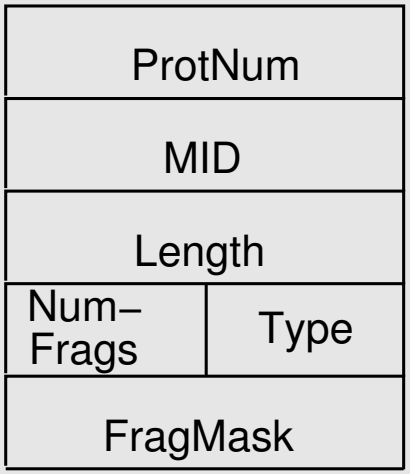
- *After sending all fragments, set timer DONE*
- *If receive SRR, send missing fragments and reset DONE*
- *If timer DONE expires, free fragments*

RPC: Bulk Transfer (BLAST)

Receiver:

- *When first fragment arrives, set timer LAST_FRAG*
- *When all fragments present, reassemble and pass up*
- *Four exceptional conditions*
  - *if last fragment arrives but message not complete*
    - *send SRR and set timer RETRY*
  - *if timer LAST_FRAG expires*
    - *send SRR and set timer RETRY*
  - *if timer RETRY expires for first or second time*
    - *send SRR and set timer RETRY*
  - *if timer RETRY expires for third time*
    - *give up and free partial message*

RPC: BLAST Header Format

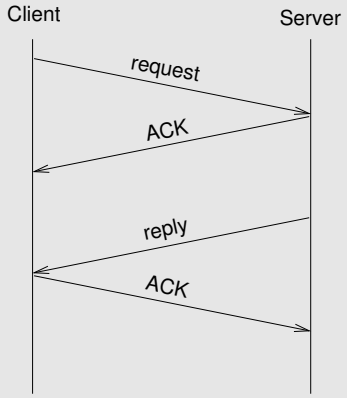


- `MID` *must protect against wrap around*
- `Type` = `DATA` *or* `SRR`
- `NumFrags` *indicates number of fragments in message*
- `FragMask` *distinguishes among fragments:*
  - *if* `Type` = `DATA`, *identifies this fragment*
  - *if* `Type` = `SRR`, *identifies missing fragments*

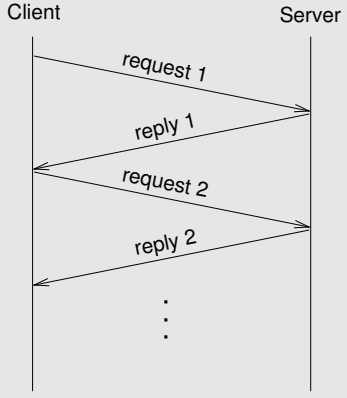
RPC: Request/Reply (CHAN)

Guarantees message delivery, and synchronizes client with server; i.e., blocks client until reply received. Supports at-most-once semantics.

Simple case:



Implicit Acknowledgements:



RPC: Request/Reply (CHAN)

Complications:

- *Lost message (request, reply, or ACK)*
  - *set RETRANSMIT timer*
  - *use message id (MID) field to distinguish*
- *Slow (long running) server*
  - *client periodically sends “are you alive” probe, or*
  - *server periodically sends “I’m alive” notice*
- *Want to support multiple outstanding calls*
  - *use channel id (CID) field to distinguish*
- *Machines crash and reboot*
  - *use boot id (BID) field to distinguish*

RPC: CHAN Header Format

```
typedef struct {
u_short  Type;    /* REQ, REP, ACK, PROBE */
u_short  CID;     /* unique channel id */
int      MID;     /* unique message id */
int      BID;     /* unique boot id */
int      Length;  /* length of message */
int      ProtNum; /* high-level protocol */
} ChanHdr;
```

RPC: CHAN Session State

```
typedef struct {
u_char    type;          /* CLIENT or SERVER */
u_char    status;        /* BUSY or IDLE */
int       retries;       /* number of retries */
int       timeout;       /* timeout value */
XkReturn  ret_val;       /* return value */
Msg       *request;      /* request message */
Msg       *reply;        /* reply message */
Semaphore reply_sem;     /* client semaphore */
int       mid;           /* message id */
int       bid;           /* boot id */
} ChanState;
```

RPC: Synchronous versus Asynchronous Protocols

Asynchronous Interface

`xPush(Sessns, Msg * msg)`

`xPop(Sessns, Msg * msg, void * hdr)`

`xDemux(Protlhlp, Sessns, Msg * msg)`

Synchronous Interface

`xCall(Sessns, Msg * req, Msg * rep)`

`xCallPop(Sessns, Msg * req, Msg * rep, void * hdr)`

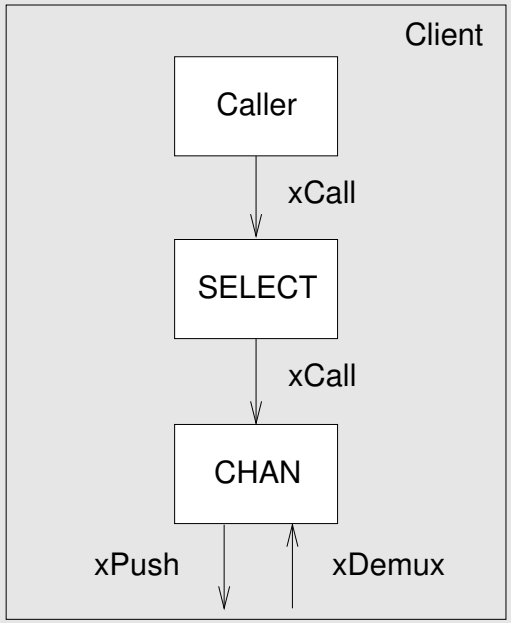
`xCallDemux(Protlhlp, Sessns, Msg * req, Msg * rep)`

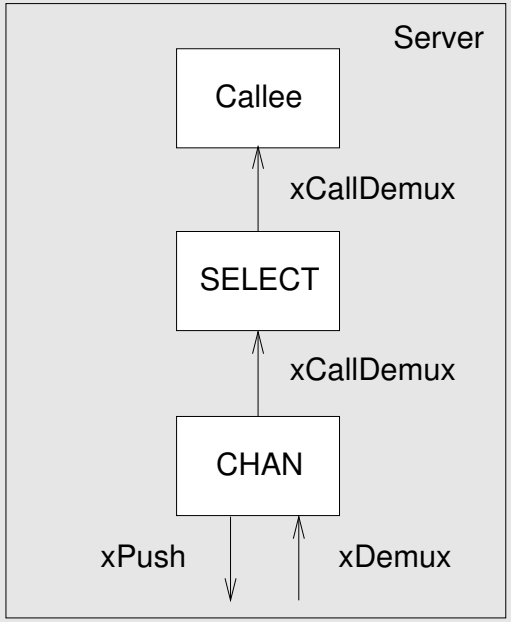
CHAN is a Hybrid Protocol

- *Synchronous from above:* `xCall`
- *Asynchronous from below:* `xPop`/`xDemux`

RPC: Dispatcher (SELECT)

Dispatches request messages to the appropriate procedure; fully synchronous counterpart to UDP.





Address Space for Procedures

- *Flat: unique id for each possible procedure*
- *Hierarchical: program + procedure within program*

RPC: Example Code

Client Side

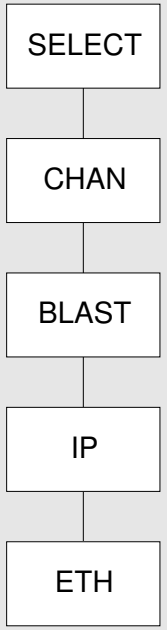
```
static XkReturn
selectCall(Sessn self, Msg *req, Msg *rep) {
SelectState *state=(SelectState *)self->state;
char        *buf;

buf = msgPush(req, HLEN);
select_hdr_store(state->hdr, buf, HLEN);
return xCall(xGetDown(self, 0), req, rep);
}
```

Server side

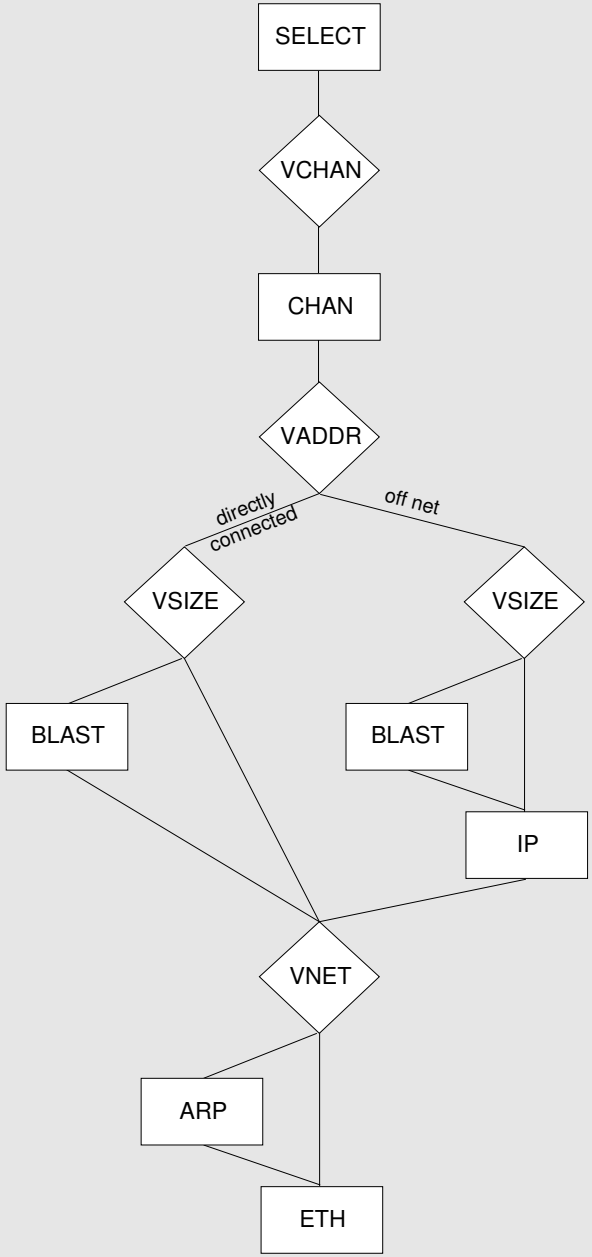
```
static XkReturn
selectCallPop(Sessn s, Sessn lls, Msg *req,
     Msg *rep, void *inHdr) {
return xCallDemux(xGetUp(s), s, req, rep);
}
```

RPC: Putting it All Together

Simple RPC Stack



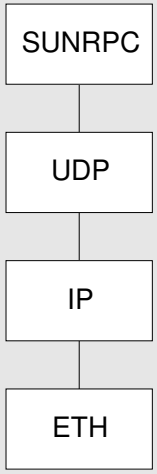
RPC: A More Interesting RPC Stack



RPC: VCHAN: A Virtual Protocol

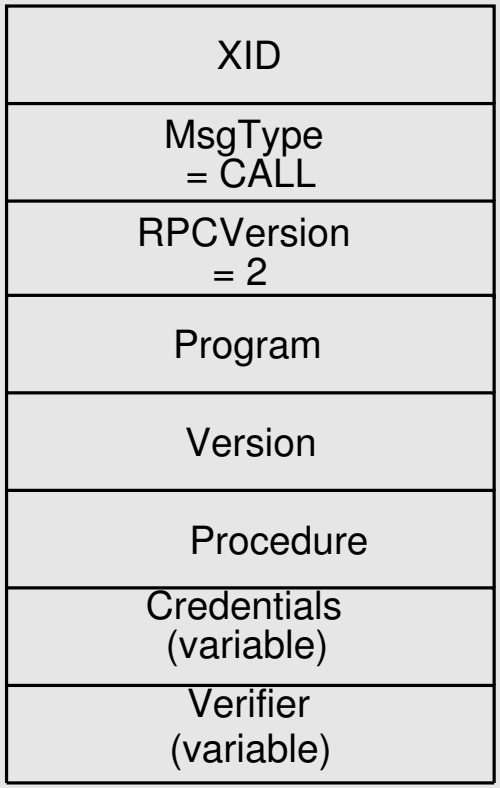
```
static XkReturn
vchanCall(Sessn s, Msg *req, Msg *rep) {
Sessn      chan;
XkReturn   result;
VchanState *state=(VchanState *)s->state;

/* wait for an idle channel */
semWait(&state->available);
chan = state->stack[--state->tos];

/* use the channel */
result = xCall(chan, req, rep);

/* free the channel */
state->stack[state->tos++] = chan;
semSignal(&state->available);

return result;
}
```

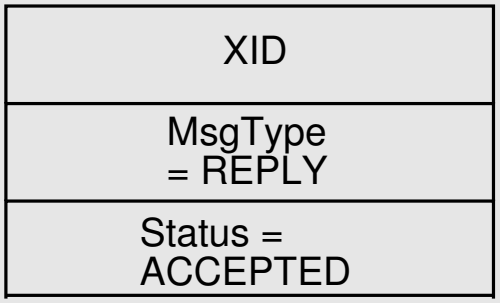
RPC: SunRPC



- *IP implements BLAST-equivalent*
  - *except no selective retransmit*
- *SunRPC implements CHAN-equivalent*
  - *except not at-most-once*
- *UDP + SunRPC implement SELECT-equivalent*
  - *UDP dispatches to program (ports bound to programs)*
  - *SunRPC dispatches to procedure w/in program*

RPC: SunRPC Header Format

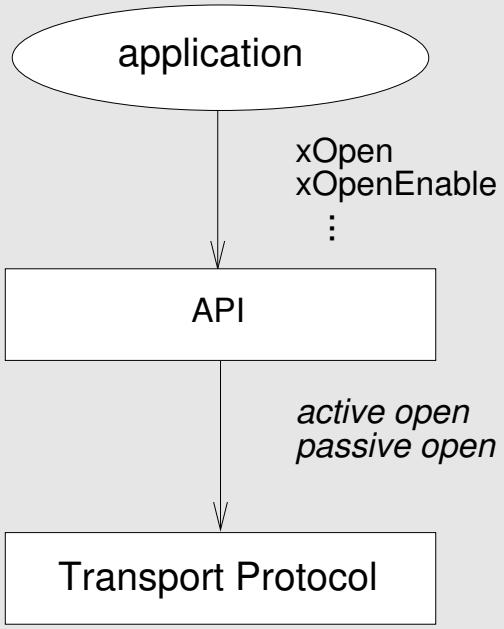




- `XID` *(transaction id) is similar to CHAN's* `MID`
- *Server does not remember last* `XID` *it serviced*
- *Problem if client retransmits request while reply is in*

API: Application Programming Interface

It is important to separate the implementation of protocols from the interface they export. This is especially important at the transport layer since this defines the point where application programs typically access the network. This interface is often called the application programming interface, or API.



API: Application Programming Interface

Notes

- *The API is usually defined by the OS.*
- *We now focus on one specific API: sockets*
- *Defined by BSD Unix, but ported to other systems*

API: Socket Operations

- *Creating a socket*

```
int socket(int domain, int type, int protocol)
```

  - `domain=PF_INET, PF_UNIX`
  - `type=SOCK_STREAM, SOCK_DGRAM`

- *Passive open on server*

```
int   bind(int socket, struct sockaddr *address, int  addr_len)
int accept(int socket, struct sockaddr *address, int *addr_len)
int listen(int socket, int backlog)
```

- *Active open on client*

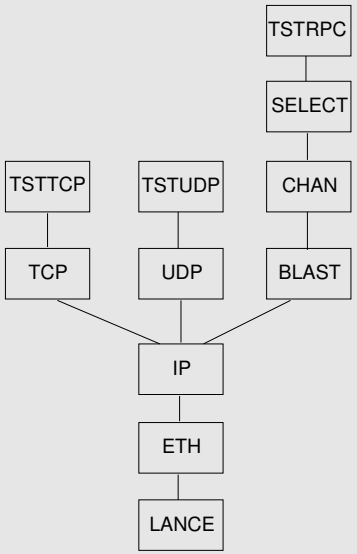
```
int connect(int socket, struct sockaddr *address, int addr_len)
```

- *Sending and receiving messages*

```
int write(int socket, char *message, int msg_len, int flags)
int  read(int socket, char *buffer,  int buf_len, int flags)
```

Performance: Experimental Method

- *DEC 3000/600 workstations (Alpha 21064 at 175MHz)*
- *10Mbps Ethernet (Lance controller)*
- *Ping-pong tests; 10,000 round trips*
- *Each test repeated five times*
- *Latency: 1-byte, 100-byte, 200-byte,... 1000-byte messages*
- *Throughput: 1KB, 2KB, 4KB,... 32KB*
- *Protocol Graphs*



Performance: Round-Trip Latency (μs)

Message size (bytes)	UDP	TCP	RPC
1	279	365	428
100	413	519	593
200	572	691	753
300	732	853	913
400	898	1016	1079
500	1067	1185	1247
600	1226	1354	1406
700	1386	1514	1566
800	1551	1676	1733
900	1719	1845	1901
1000	1878	2015	2062

Per-Layer Latency

- *ETH + wire: 216μs*
- *UDP/IP: 58μs*

Performance: Throughput