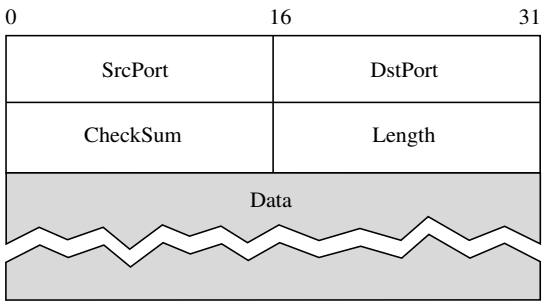
Copyright 1996. All rights reserved. Permission to reproduce this document for not for profit educational purposes is hereby granted. This document may not be reproduced for commercial purposes without the express written consent of the author.

End-to-End (Transport) Protocols

- Underlying best-effort network
  - drops 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 apply flow control to the sender
  - support multiple application processes on each host

Simple Demultiplexor (UDP)

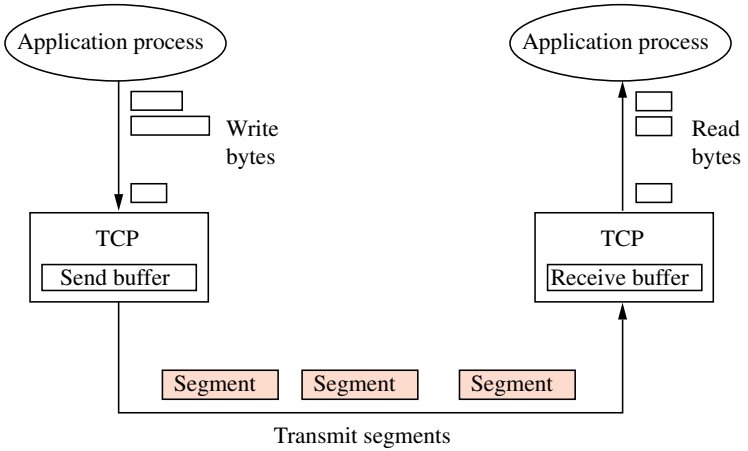
- Unreliable and unordered datagram service
- Adds multiplexing
- No flow control
- Endpoints identified by ports
  - servers have *well-known* ports
  - see **`/etc/services`** on Unix
- Optional checksum
  - pseudo header + udp header + data
- Header format



Reliable Byte-Stream (TCP)

Overview

- Connection-oriented
- Byte-stream
  - sending process writes some number of bytes
  - TCP breaks into *segments* and sends via IP
  - receiving process reads some number of bytes



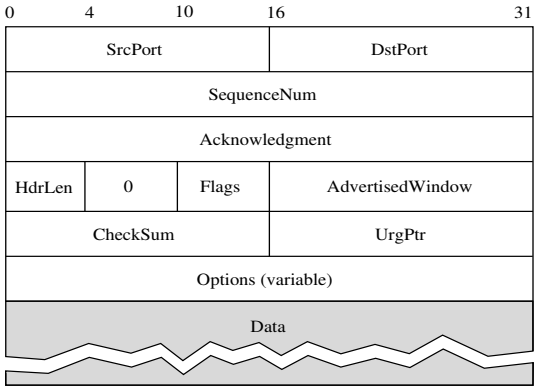
- Full duplex
- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network

End-to-End Issues

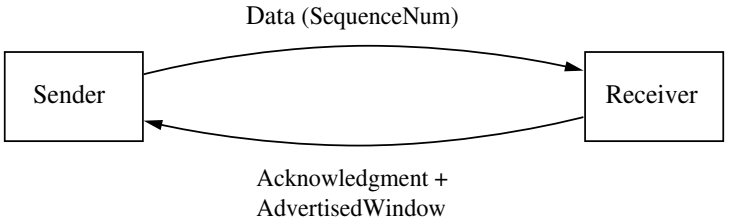
Based on sliding window protocol used at data link level, but the situation is very different.

- Potentially connects many different hosts
  - need explicit connection establishment and termination
- Potentially different RTT
  - need adaptive timeout mechanism
- Potentially long delay in network
  - need to be prepared for arrival of very old packets
- Potentially different capacity at destination
  - need to accommodate different amounts of buffering
- Potentially different network capacity
  - need to be prepared for network congestion

Segment Format



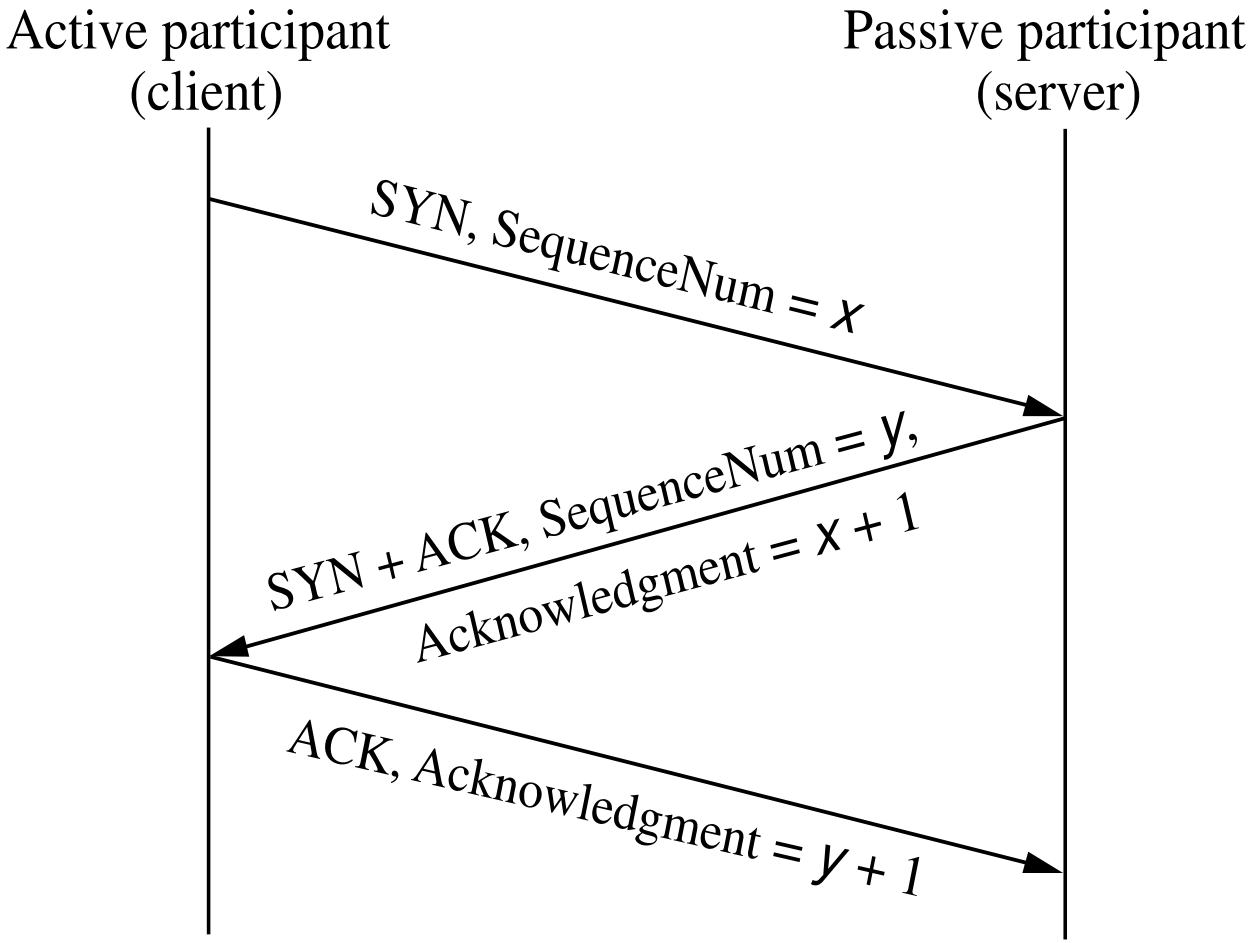
- Each connection identified with 4-tuple:
  - **<SrcPort, SrcIPAddr, DstPort, DstIPAddr>**
- Sliding window + flow control
  - **Acknowledgment, SequenceNum, AdvertisedWindow**



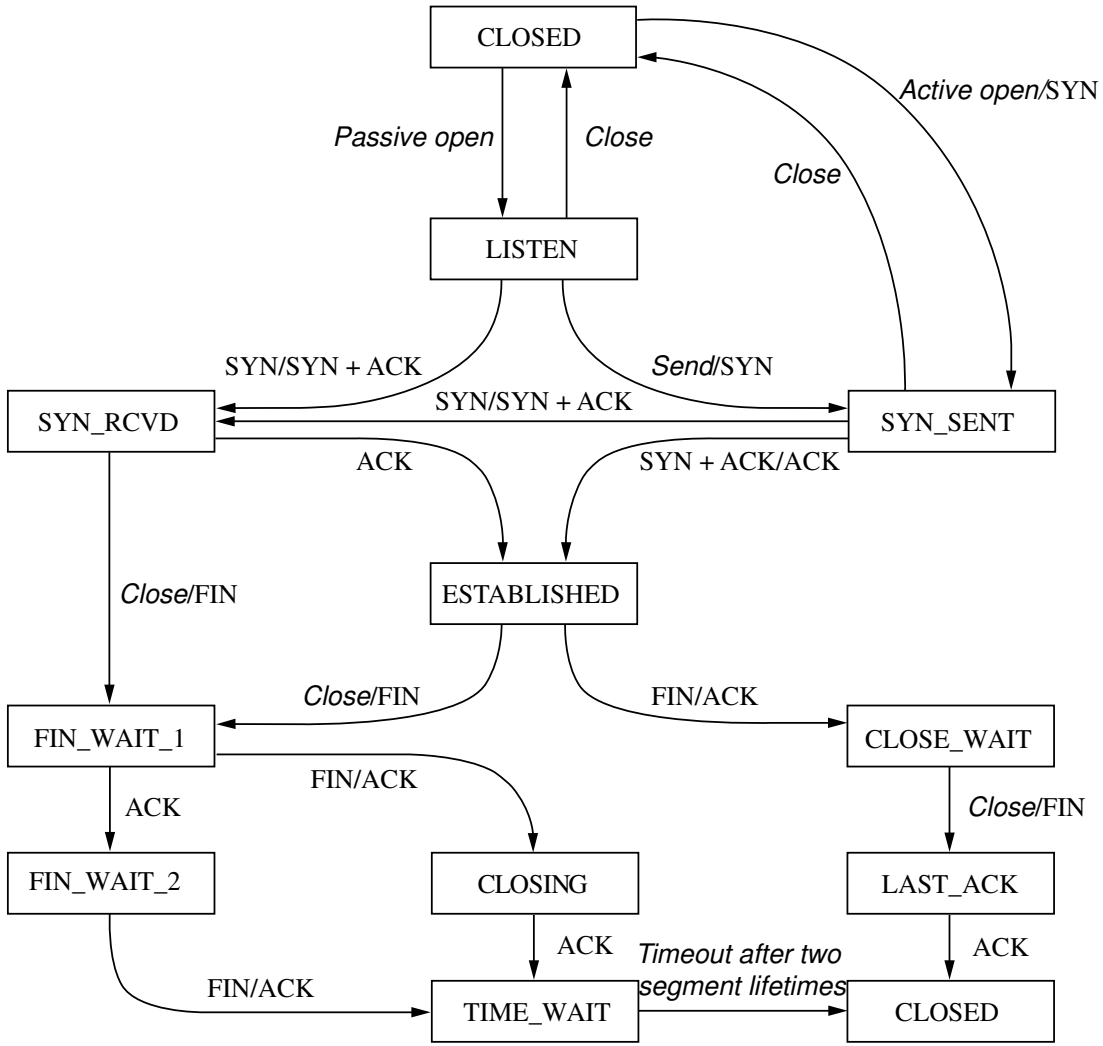
- Flags: **SYN, FIN, RESET, PUSH, URG, ACK**
- Checksum: pseudo header + tcp header + data

Connection Establishment and Termination

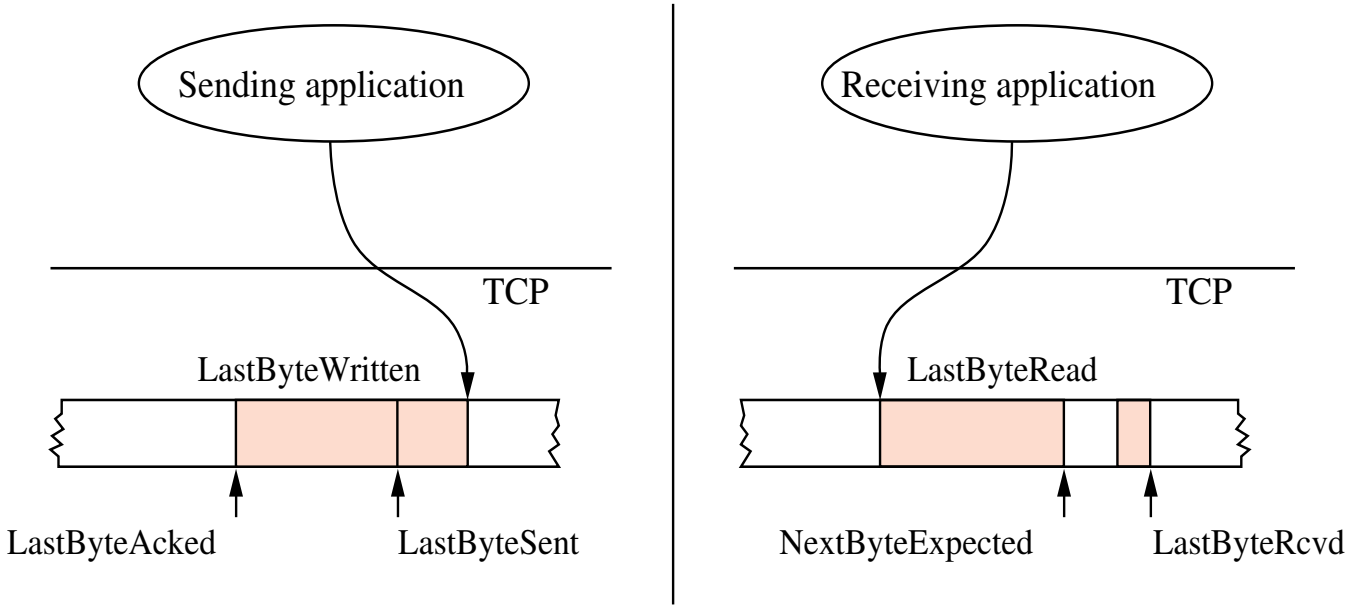
- Three-Way Handshake



State Transition Diagram



Sliding Window Revisited



- Each byte has a sequence number
- ACKs are cumulative

- Sending side
  - **`LastByteAcked Š LastByteSent`**
  - **`LastByteSent Š LastByteWritten`**
  - bytes between **`LastByteAcked`** and **`LastByteWritten`** must be buffered
- Receiving side
  - LastByteRead < NextByteExpected
  - bytes between **`NextByteRead`** and **`LastByteRcvd`** must be buffered

Flow Control

- Sender buffer size: **`MaxSendBuffer`**
- Receive buffer size: **`MaxRcvBuffer`**
- Receiving side
  - **`LastByteRcvd - NextByteRead Š MaxRcvBuffer`**
  - **`AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - NextByteRead)`**

- Sending side
  - **NextByteExpected Š LastByteRcvd + 1**
  - **LastByteSent - LastByteAcked Š AdvertisedWindow**
  - **EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)**
  - **LastByteWritten - LastByteAcked Š MaxSendBuffer**
  - block sender if **(LastByteWritten - LastByteAcked) + *y* > MaxSendBuffer**
- Always send ACK in response to an arriving data segment
- Persist when **AdvertisedWindow=0**

Keeping the Pipe Full

- Wrap Around: 32-bit **`SequenceNum`**
- Bandwidth & Time Until 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

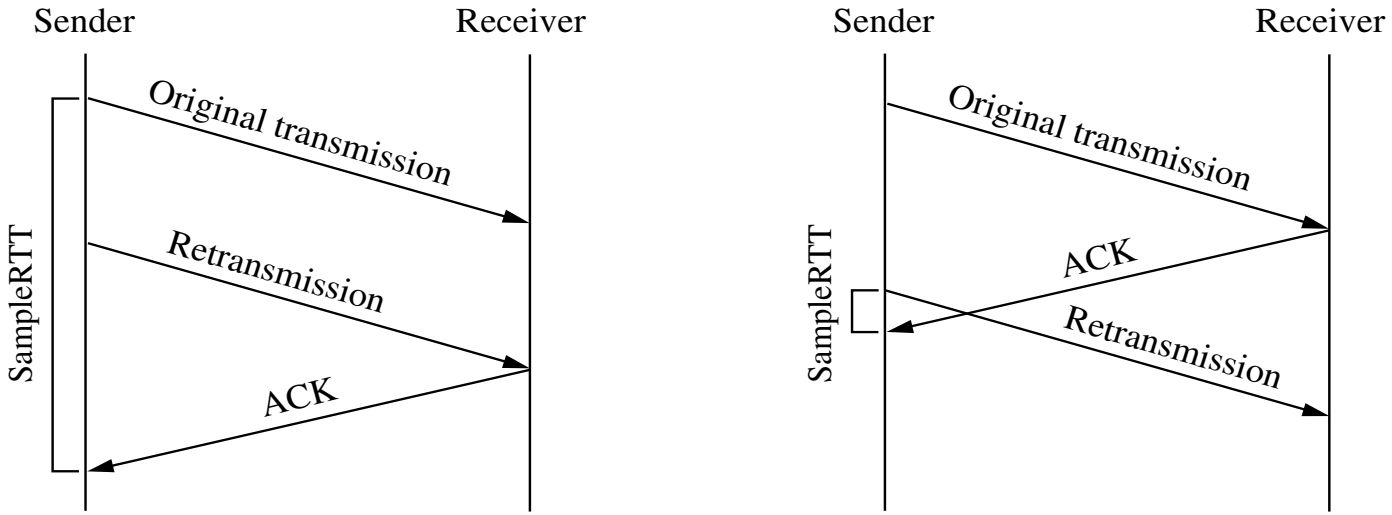
- Bytes in Transit: 16-bit **`AdvertisedWindow`**
- Bandwidth & Delay x Bandwidth Product

Bandwidth	Delay x 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

Adaptive Retransmission

- Original Algorithm
- Measure **`SampleRTT`** for each segment/ACK pair
- Compute weighted average of RTT
  - **`EstimatedRTT =`** α x **`EstimatedRTT +`** β x **`SampleRTT`**
  - where $\alpha + \beta$ **`= 1`**
  - α between 0.8 and 0.9
  - β between 0.1 and 0.2
- Set timeout based on **`EstimatedRTT`**
  - **`TimeOut = 2`** x **`EstimatedRTT`**

Karn/Partridge Algorithm



- Do not sample RTT when retransmitting
- Double timeout after each retransmission

- Jacobson/Karels Algorithm
- New calculation for average RTT

```
Diff = SampleRTT - EstimatedRTT
EstimatedRTT = EstimatedRTT + (δ x
Deviation = Deviation + δ(|Diff|- Deviation)
```

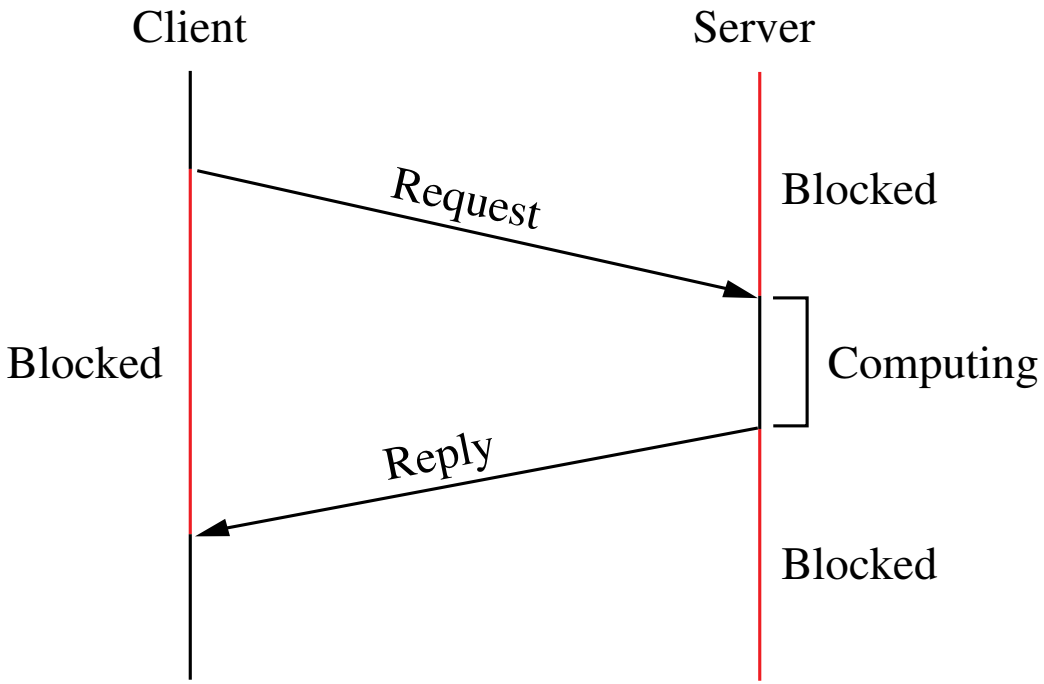
  - where δ is a fraction between 0 and 1
- Consider variance when setting timeout value
  - TimeOut = μ x EstimatedRTT + ϕ x Deviation
  - where $\mu = 1$ and $\phi = 4$
- Notes
  - algorithm only as good as granularity of clock (500ms on Unix)
  - accurate timeout mechanism important to congestion control (later)

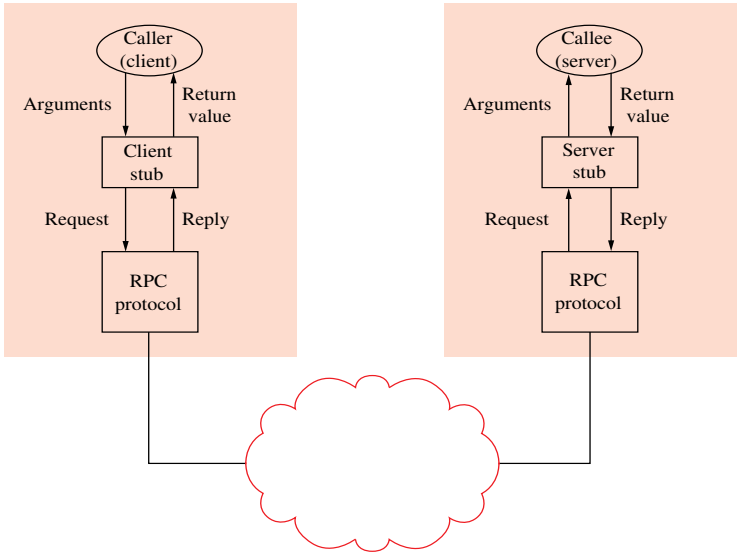
TCP Extensions

- Implemented as header options
- Store timestamp in outgoing segments
- Use 32-bit timestamp to extend sequence space (PAWS)
- Shift (scale) advertised window

Remote Procedure Call

Overview


Client
Server
Request
Blocked
Blocked
Computing
Reply
Blocked

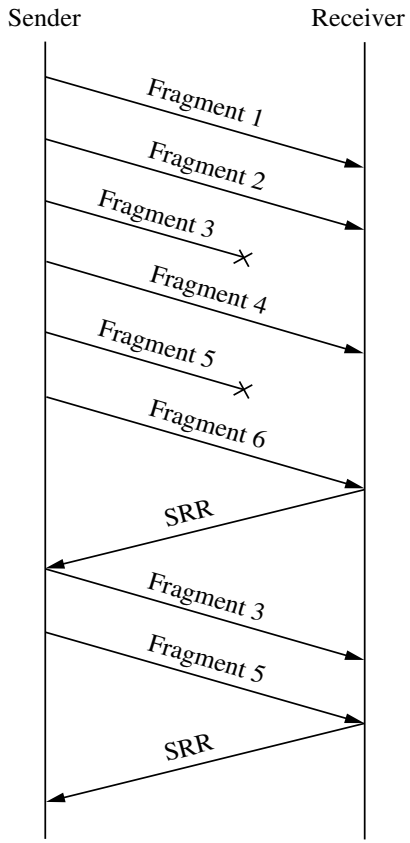


RPC protocol consists of three basic functions

- 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.

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*).



Sender:

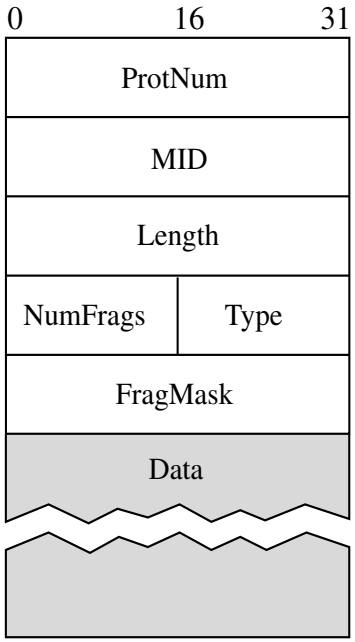
- After sending all fragments, set timer DONE
- If receive SRR, send missing fragments and reset DONE
- If timer DONE expires, free fragments

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

BLAST Header Format

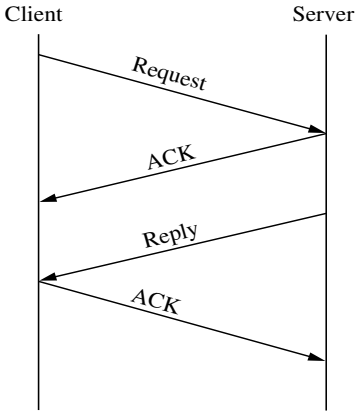


- **`MID`** must protect against wrap around
- **`Type`** = **`DATA`** or **`SRR`**
- **`NumFrags`** indicates number of fragments in message
- **`FragMask`** distinguishes among fragments:
  - if **`Type=DATA`**, identifies this fragment
  - if **`Type=SRR`**, identifies missing fragments

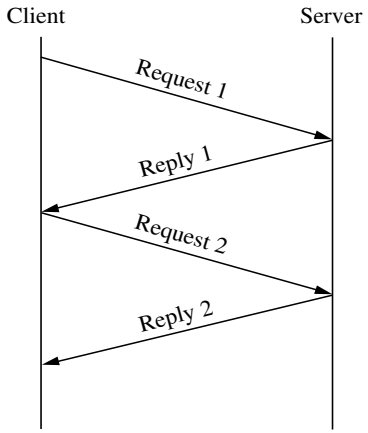
Request/Reply (CHAN)

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

Simple case:



Implicit Acknowledgements:



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

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;
```

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;
```

Synchronous versus Asynchronous Protocols

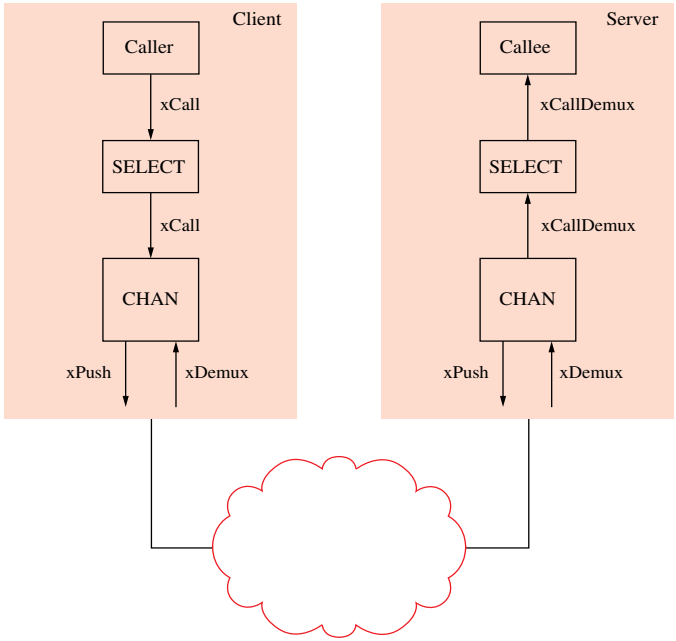
```
Asynchronous Interface
   xPush(Sessn s, Msg *msg)
   xPop(Sessn s, Msg *msg, void *hdr)
   xDemux(Protl hlp, Sessn s, Msg *msg)

Synchronous Interface
   xCall(Sessn s, Msg *req, Msg *rep)
   xCallPop(Sessn s, Msg *req, Msg *rep,
            void *hdr)
   xCallDemux(Protl hlp, Sessn s, Msg *req,
              Msg *rep)
```

CHAN is a Hybrid Protocol

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

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

Example Code

Client Side

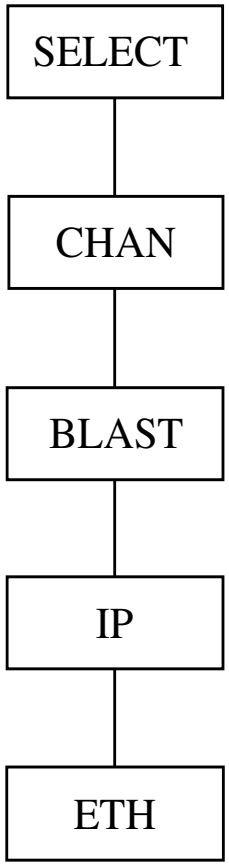
```
static XkReturn
selectCall(Sessn self, Msg *req, Msg *rep)
{
   SelectState *state=(SelectState *)self->state;
   char        *buf;

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

Server side

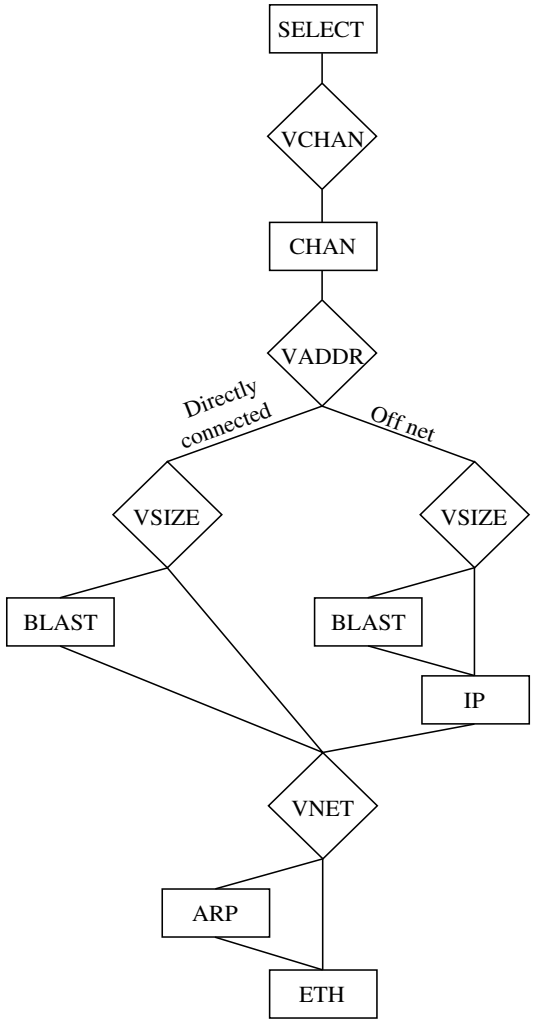
```
static XkReturn
selectCallPop(Sessn s, Sessn lls, Msg *req,
              Msg *rep, void *inHdr)
{
   return xCallDemux(xGetUp(s), s, req, rep);
}
```

Putting it All Together

Simple RPC Stack



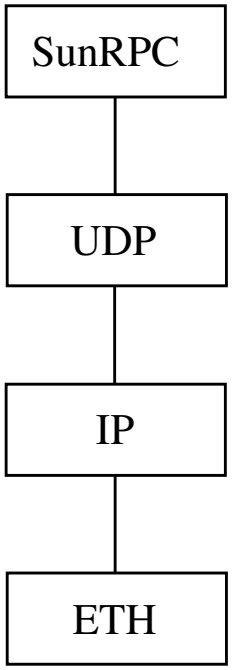
A More Interesting RPC Stack



VCHAN: A Virtual Protocol

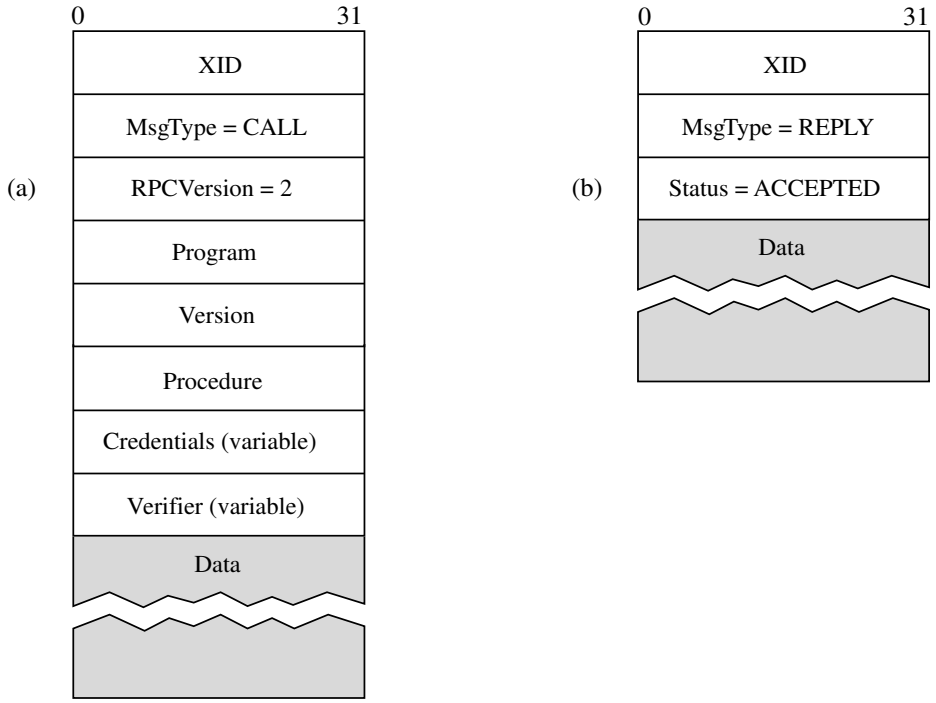
```
static XkReturn
vchanCall(Sessn s, Msg *req, Msg *rep)
{
   Sessn      chan;
   XkReturn   result;
   VchanState *state=(VchanState *)s->state;

   /* 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;
}
```

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

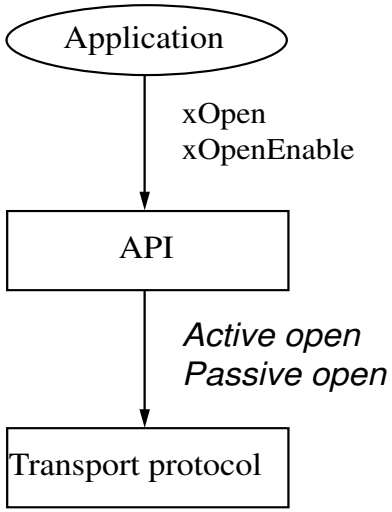
SunRPC Header Format



- **`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

Application Programming Interface

It is important to separate the *implementation* of protocols from the *interface* they export. This is espeically 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.



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

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

- Active open on client

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

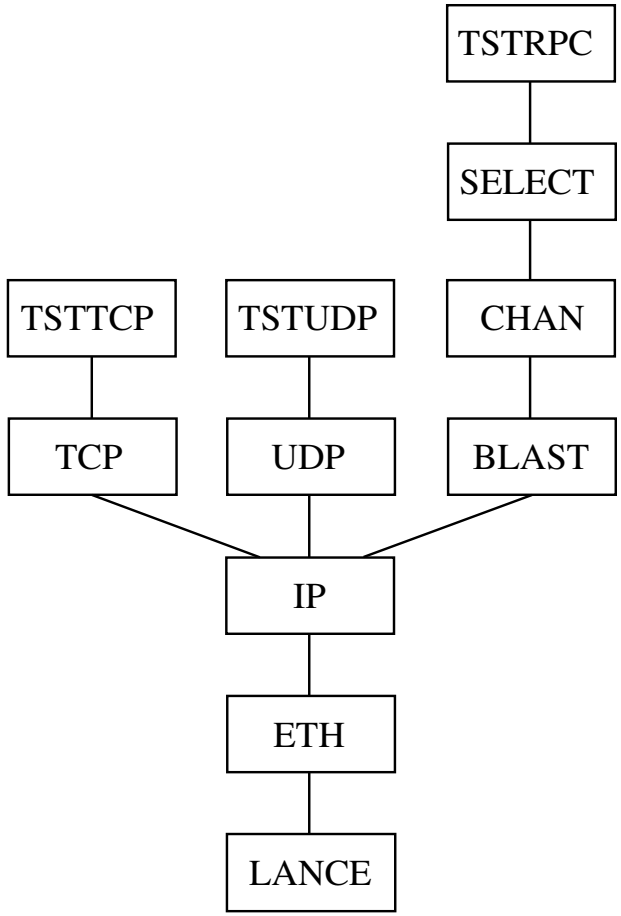
- Sending and receiving messages

```
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


TSTRPC
SELECT
TSTTCP
TSTUDP
CHAN
TCP
UDP
BLAST
IP
ETH
LANCE

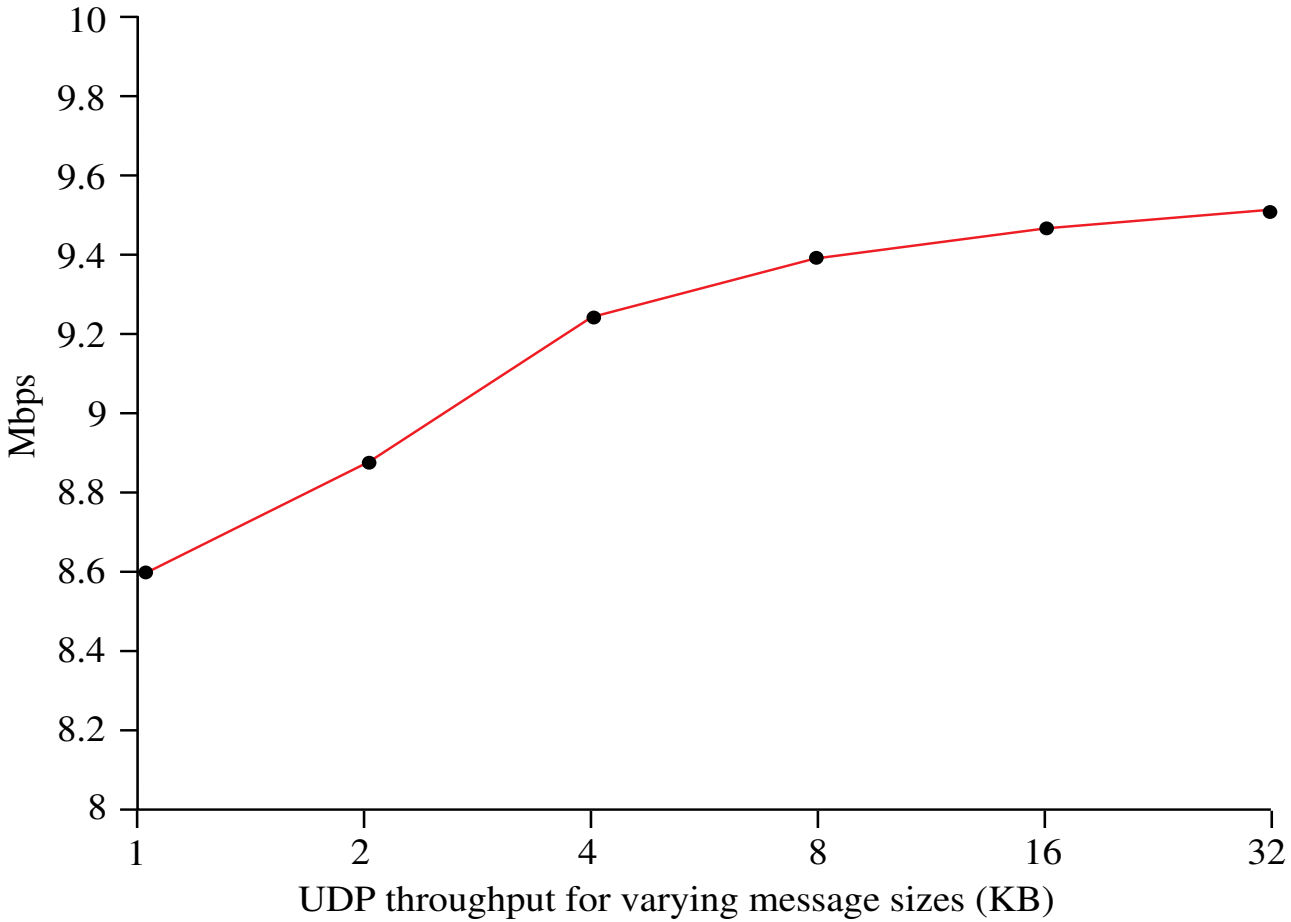
Round-Trip Latency (µs)

Message size (bytes)	UDP	TCP	RPC
1	297	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

Throughput


10
9.8
9.6
9.4
9.2
9
8.8
8.6
8.4
8.2
8
Mbps
1
2
4
8
16
32
UDP throughput for varying message sizes (KB)