

COMPUTER NETWORKS  
CS 45201  
CS 55201

CHAPTER 5  
End-to-End protocols

Paul A. Farrell and H. Peyravi  
Department of Computer Science  
Kent State University  
Kent, Ohio 44242  
*farrell@mcs.kent.edu*  
*<http://www.cs.kent.edu/~farrell>*

Fall 2001

Contents

- End-to-End (Transport) Protocols
- Simple Demultiplexer (UDP)
- Reliable Byte-Stream (TCP)
- Remote Procedure Call

## 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 Demultiplexer (UDP)

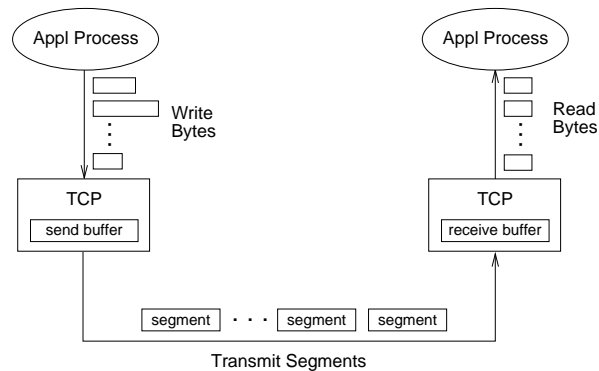
- Unreliable and unordered datagram service
- Adds multiplexing - multiple connections between hosts
- No flow control
- Endpoints identified by ports (16 bits -  $\approx$  64K possible per host)
  - ▶ servers have *well-known* ports
  - ▶ client and server use these to agree on other port for communication
  - ▶ see `/etc/services` on Unix
- Checksum (Optional IPv4, Mandatory IPv6) - same as IP algorithm
  - ▶ pseudo header + udp header + udp data
  - ▶ pseudo header is IP protocol number, source and destination IP addresses, UDP length field
- Header format

Src Port	Dest Port
Check Sum	Length

## 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

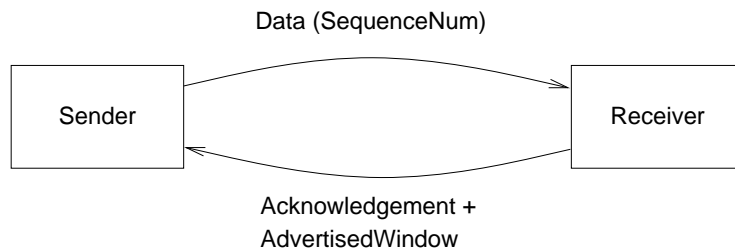
Based on sliding window protocol used at data link level, but the situation is very different.

1. Potentially connects many different hosts
  - need explicit connection establishment and termination
2. Potentially different RTT
  - need adaptive timeout mechanism
3. Potentially long delay in network
  - need to be prepared for arrival of very old packets
  - is limit, discarded after TTL
  - MSL (maximum segment lifetime) - recommended 120 sec
4. Potentially different capacity at destination
  - need to accommodate different amounts of buffering
5. Potentially different network capacity
  - need to be prepared for network congestion

### Segment Format

Src Port		Dest Port	
SequenceNum			
Acknowledgement			
HdrLen (4)	0 (6)	Flags (6)	Advertised Window
Checksum		UrgPtr	
options (variable)			
data			

- Each connection identified with 4-tuple:
  - ▶  $\langle \text{SrcPort}, \text{SrcIPAddr}, \text{DstPort}, \text{DstIPAddr} \rangle$
- Sliding window + flow control
  - ▶ Acknowledgment, SequenceNum, AdvertisedWindow



### ■ Flags

- ▶ SYN: TCP connection establishing
- ▶ FIN: TCP connection terminating
- ▶ RESET: Receiver is confused - abort connection
- ▶ PUSH: Sender invokes push operation
- ▶ URG: segment contains urgent data
- ▶ ACK: Acknowledgment

### ■ Checksum

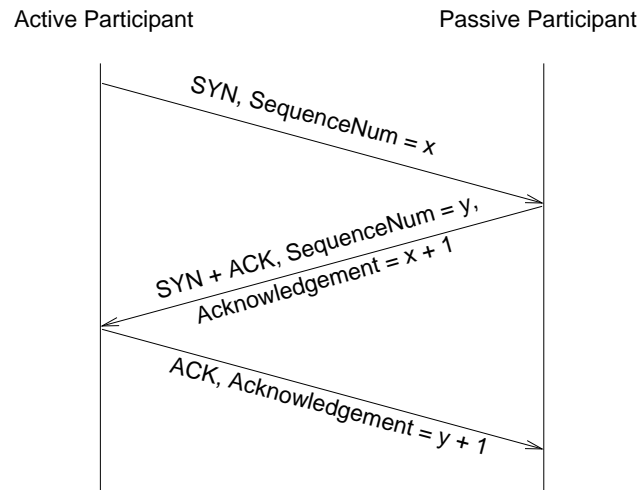
- ▶ pseudo header + tcp header + data

When does TCP send segment?

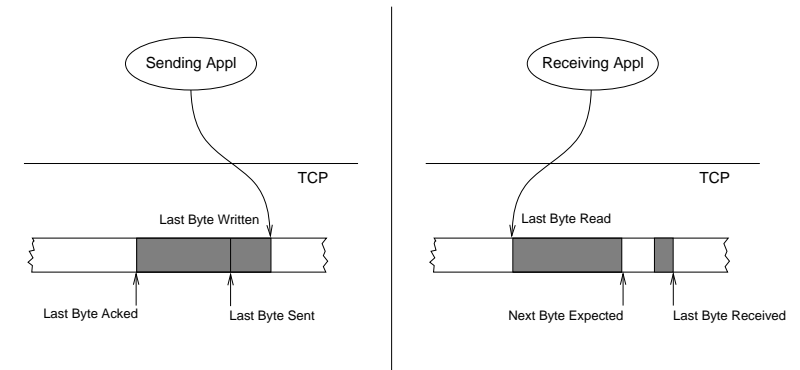
1. After MSS (Maximum Segment Size) bytes are buffered
  - usually largest size that IP will not fragment
  - $\text{MSS} = \text{MTU} - \text{sizeof}(\text{TCP} + \text{IP headers})$
2. if sender flushes buffer with **push** operation
  - Telnet does after each character
3. when timer expires

## Connection Establishment and Termination

### Three-Way Handshake



## Sliding Window Revisited



- Each byte has a sequence number
- ACKs are cumulative
- Sending side
  - ▶  $\text{LastByteAcker} \leq \text{LastByteSent}$
  - ▶  $\text{LastByteSent} \leq \text{LastByteWritten}$
  - ▶ bytes between `LastByteAcker` and `LastByteWritten` must be buffered
- Receiving side
  - ▶  $\text{LastByteRead} < \text{NextByteExpected}$ , Why?
  - ▶  $\text{NextByteExpected} \leq \text{LastByteRcvd} + 1$ , Why?

## Flow Control

- Receiver *advertises* a window size to prevent buffer overflow
- Sender buffer size: `MaxSendBuffer`
- Receive buffer size: `MaxRcvBuffer`
- Receiving side
  - ▶  $\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer}$
  - ▶  $\text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead})$
- Sending side
  - ▶  $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$
  - ▶  $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$
  - ▶  $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$
  - ▶ TCP blocks sender from sending  $y$  bytes if  $(\text{LastByteWritten} - \text{LastByteAcked}) + y > \text{MaxSendBuffer}$
- Always send ACK in response to an arriving data segment, but not otherwise
- Sender persists in sending 1 byte when `AdvertisedWindow=0`
- Eventually ACK will arrive with new `AdvertisedWindow`

## Keeping the Pipe Full

- Wrap Around: 32-bit `SequenceNum` - want no wrap in 120 sec

Bandwidth	Time Until Wrap Around
T1 (1.5Mbps)	6.4 hours $2^{32}/(1.544/8)$ bytes =6.18 hrs
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` allows 64KB of data in pipe

- ▶ Assume  $\text{RTT} = 100$  ms      typical crosscountry delay in US

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

## Adaptive Retransmission

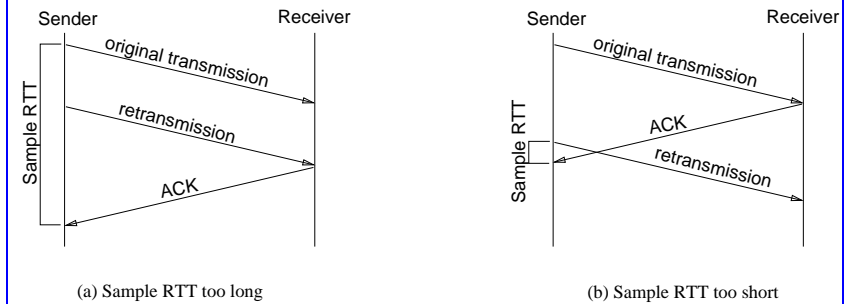
### ⇒ Original Algorithm

- Measure SampleRTT for each segment/ACK pair
- Compute weighted average of RTT
  - ▶  $\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + \beta \times \text{SampleRTT}$
  - ▶ where  $\alpha + \beta = 1$
  - ▶  $\alpha$  between 0.8 and 0.9
  - ▶  $\beta$  between 0.1 and 0.2
- Set timeout based on EstimatedRTT
  - ▶  $\text{TimeOut} = 2 \times \text{EstimatedRTT}$

### ⇒ A flaw

- Does ACK really acknowledges a transmission?
- No, it acknowledges receipt of a segment
- How many retransmissions had taken place before ACK arrived?

### ⇒ Wrong samples



- in (a) sample should be for the second attempt
- in (b) sample should be for the first attempt

### ⇒ Karn/Partridge Algorithm

- Do not sample RTT when retransmitting
- Double timeout after each retransmission
  - ▶ Similar to backoff algorithm

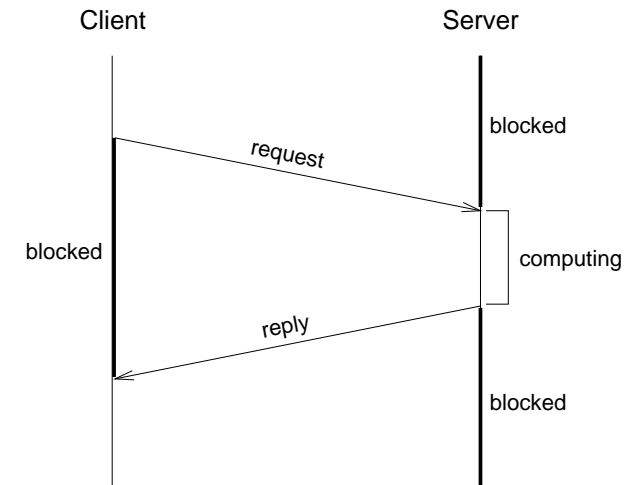
### ⇒ Jacobson/Karels Algorithm

- Karn/Partridge algorithm was introduced when the Internet was not suffering the current congestion
- Consider variance when setting timeout value
- Jacobson/Karels came up with a new calculation for average RTT
  - Difference = SampleRTT - EstimatedRTT
  - EstimatedRTT = EstimatedRTT + ( $\delta \times$  Difference)
  - Deviation = Deviation +  $\delta (|$ Difference $-$  Deviation)
  - ▶ where  $\delta$  is a fraction between 0 and 1
- Timeout =  $\mu \times$  EstimatedRTT +  $\phi \times$  Deviation
  - ▶ where  $\mu = 1$  and  $\phi = 4$

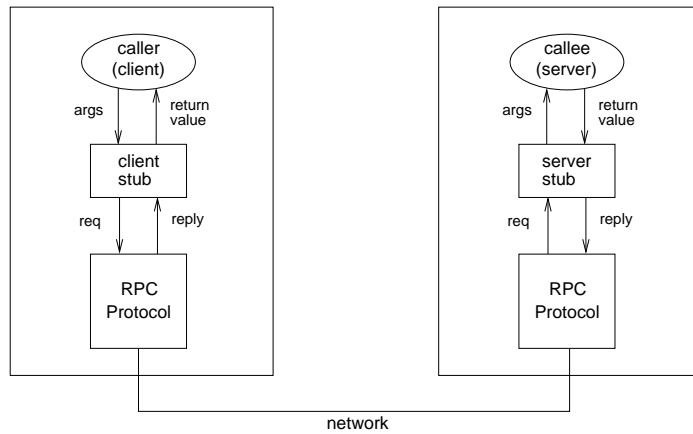
## Remote Procedure Call

### Overview

- Common pattern of communication used by application programs
- Also called message transaction





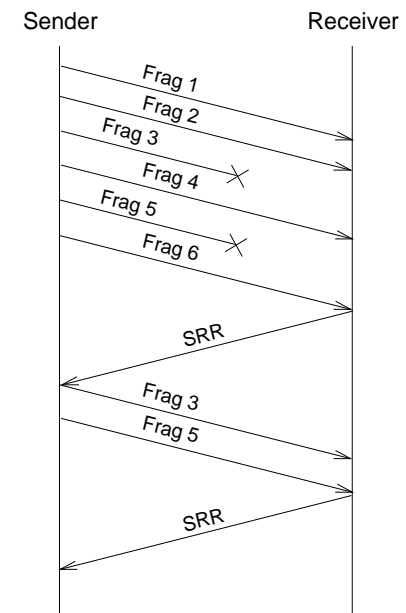


Peterson divides RPC protocol into three basic functions

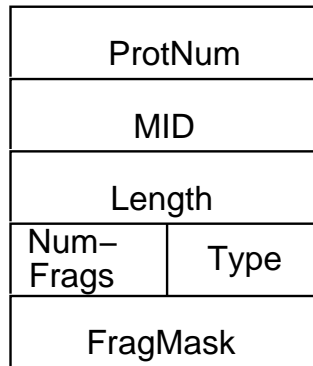
- BLAST: fragments and reassembles large messages
- CHAN: synchronizes request and reply messages
- SELECT: dispatches request messages to the correct process

### 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 (or partial acknowledgments)*.



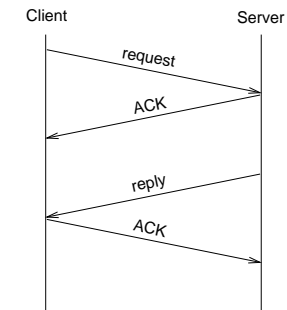
## BLAST Header Format



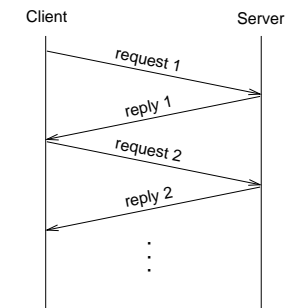
- MID must protect against wrap around
- Type = DATA or SRR
- NumFrag indicates number of fragments in message
- FragMask distinguishes among fragments:
  - ▶ if Type=DATA, identifies this fragment
  - ▶ if Type=SRR, identifies missing fragments

## 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:

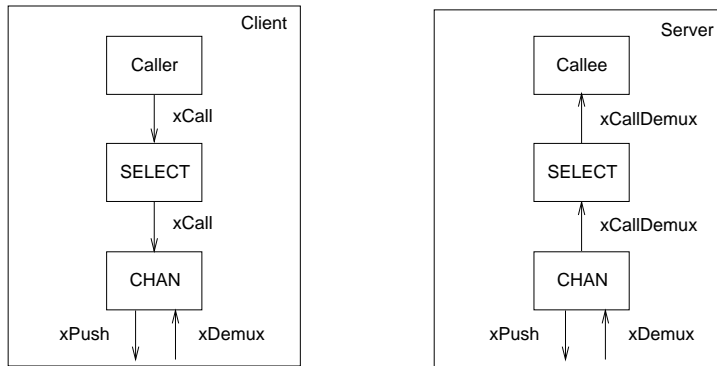


Implicit Acknowledgments:



## 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

## Putting it All Together

### Simple RPC Stack

