

## **Problem: getting processes to communicate**

- Simple demultiplexer (UDP)
- Reliable byte-stream (TCP)
- Remote procedure call (RPC)
- transport for real-time apps (RTP)

# End-to-End Protocols

---

Hongwei Zhang

<http://www.cs.wayne.edu/~hzhang>



Victory is the beautiful, bright coloured flowers. Transport is the stem without which it could never have blossomed.

--- Winston Churchill

Acknowledgement: this lecture is partially based on the slides of Dr. Larry Peterson

# Factors affecting transport protocol design

---

- From above: application requirements
  - Reliably deliver message
  - Deliver message in-order
  - Deliver at most once copy of each message
  - Support arbitrarily sized message (large or small)
  - Synchronize sender and receiver
  - Flow control
  - Support multiple application processes on each host

# Factors (contd.)

---

- From below: network properties
  - Can drop messages (how?)
  - Can reorder messages (how?)
  - Can deliver duplicate copies of a message (how?)
  - Can delay message delivery
  
  - Limit messages to some finite size  
i.e., *best-effort* service
- Challenge: how to turn the less-than-desirable networks into the high level services required by application programs (processes)

# Outline

---

- Simple demultiplexer (UDP)
- Reliable byte-stream (TCP)
- Remote procedure call (RPC)
- Transport for real-time applications (RTP)

# Outline

---

- Simple demultiplexer (UDP)
- Reliable byte-stream (TCP)
- Remote procedure call (RPC)
- Transport for real-time applications (RTP)

# Simple Demultiplexor: User Datagram Protocol (UDP)

---

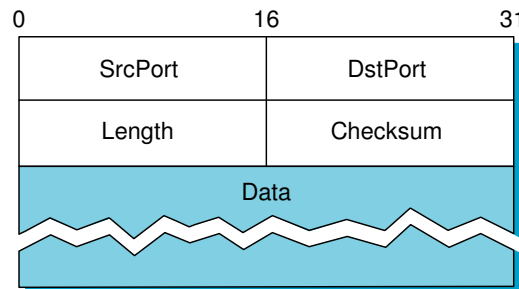
- Unreliable and unordered datagram service
- No flow control
- But, it enables multiple application processes on each host to share the network

# UDP (contd.)

---

- Multiplexing/demultiplexing via port number, i.e., endpoints identified by ports (more precisely, <host, port>)
  - servers have *well-known* ports
  - see `/etc/services` on Unix

- Header format



- Optional checksum
  - Over “UDP header + data + **pseudo header** (*protocol number, src. IP, dst. IP* of IP header, and *length* of UPD header)”
  - Pseudo header is to make sure the this message has been delivered between the correct two endpoints

# Outline

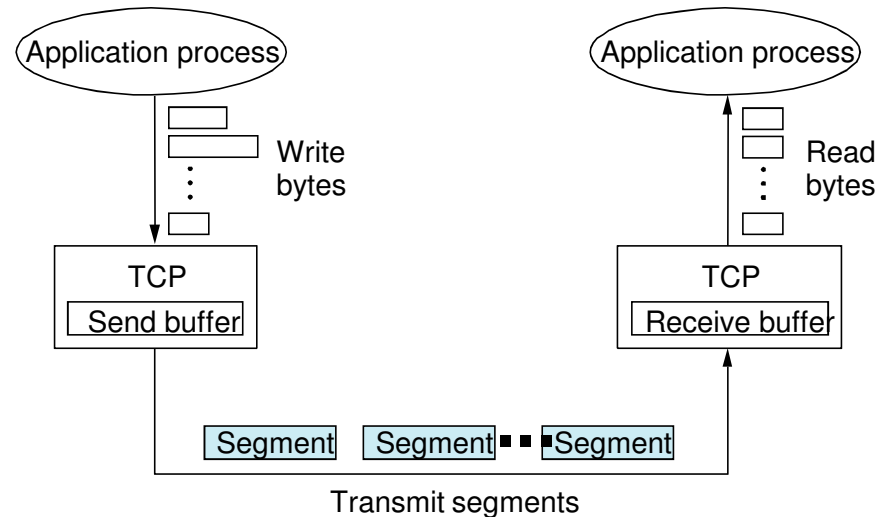
---

- Simple demultiplexer (UDP)
- **Reliable byte-stream (TCP)**
- Remote procedure call (RPC)
- Transport for real-time applications (RTP)

# TCP (Transmission Control Protocol) Overview

---

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



# End-to-end issues: Transport vs. Data Link

---

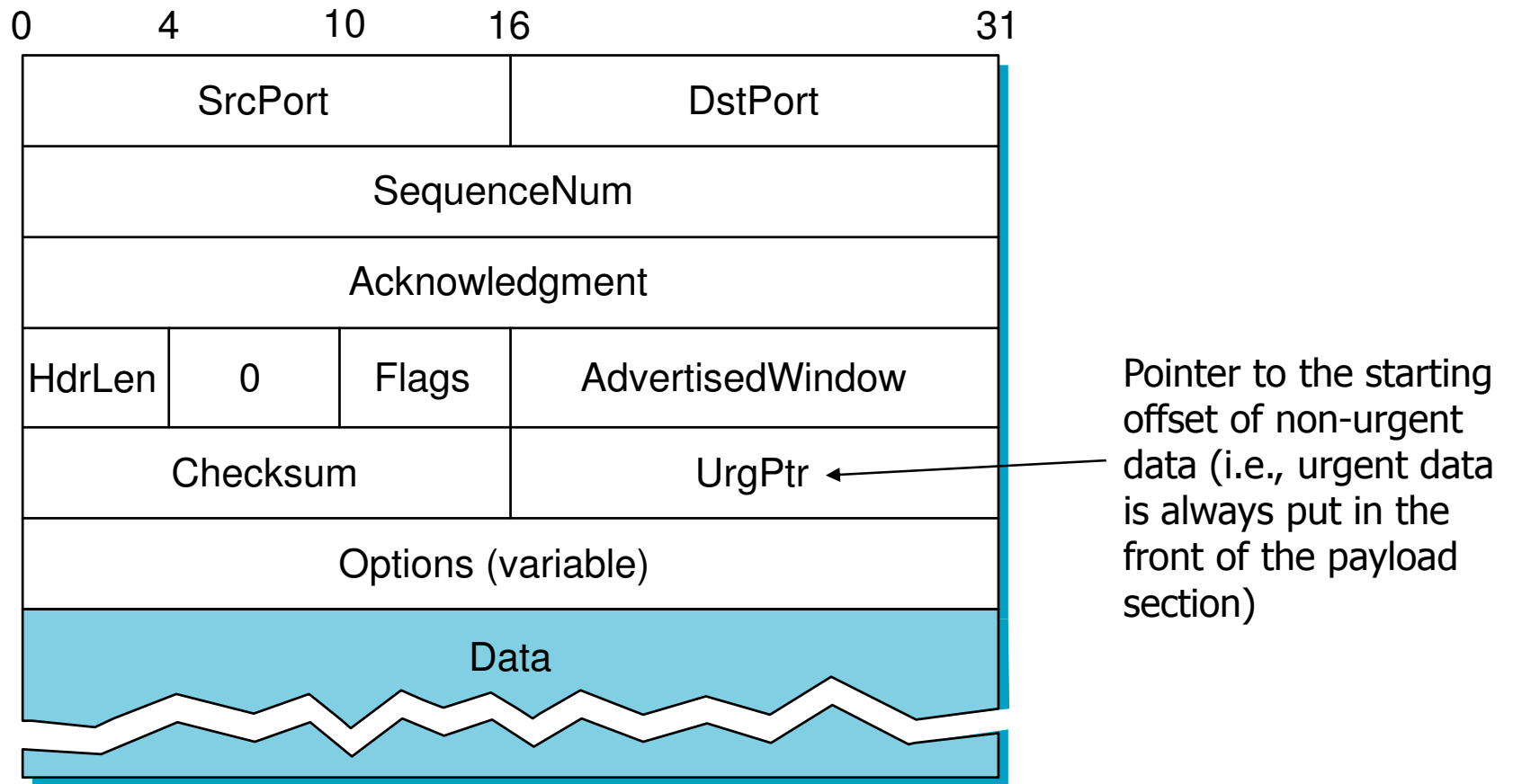
- A network potentially connects many different hosts
  - need explicit connection establishment (to agree on communication parameters etc.) and termination
- Different networks can have different RTT
  - need adaptive timeout mechanism
- Potentially long delay in network
  - need to be prepared for arrival of very old packets
- Potentially different capacity at destination
  - need to accommodate different node capacity (flow control)
- Potentially different network capacity
  - need to be prepared for network congestion (congestion control)

# End-to-end issues (contd.)

---

- Hop-by-hop guarantee does not imply end-to-end guarantee
  - Heterogeneity: different part of a connection may provide different QoS
  - Hop-reliability does not guarantee end-to-end reliability: packet drop due to queue overflow
- This leads to “end-to-end argument”
  - Idea: a function should not be provided at a lower level unless it can be completely and correctly implemented at that level
  - Practice: this rule may not be observed for performance optimization
    - e.g., CRC check and error correction at link layer helps avoid wasting resources in delivering a corrupted frame all the way to the ultimate receiver

# Segment Format

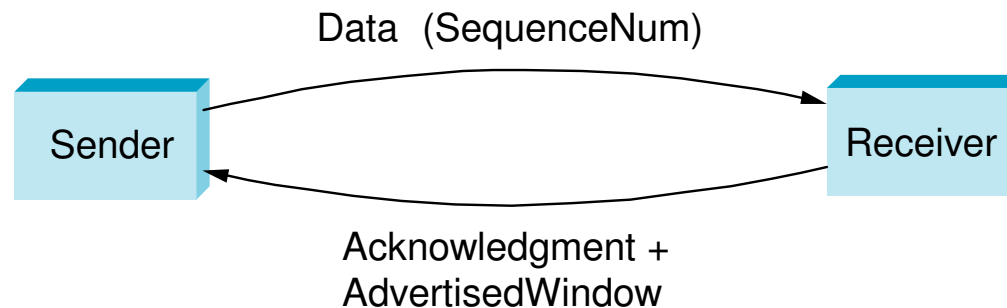


Q: why doesn't the format include a field identifying the "payload length"?

# Segment Format (contd.)

---

- Each connection is identified with 4-tuple:
  - `(SrcPort, SrcIPAddr, DsrPort, DstIPAddr)`
- Sliding window + flow control
  - `SequenceNum, acknowledgment, AdvertisedWinow`



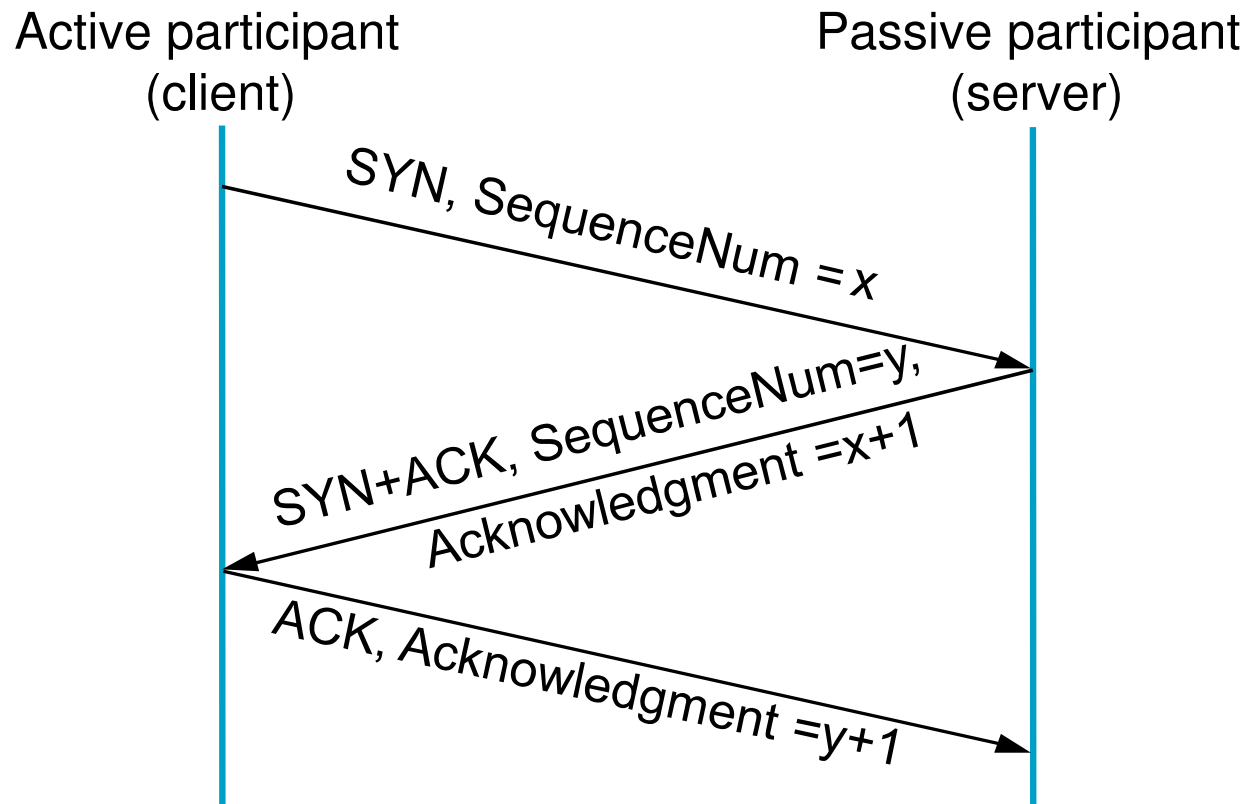
- Flags
  - `SYN, FIN, RESET, ACK, URG (for urgent data), PUSH`
- Checksum
  - TCP header + data + pseudo header (as in UDP)

# Connection Establishment and Termination

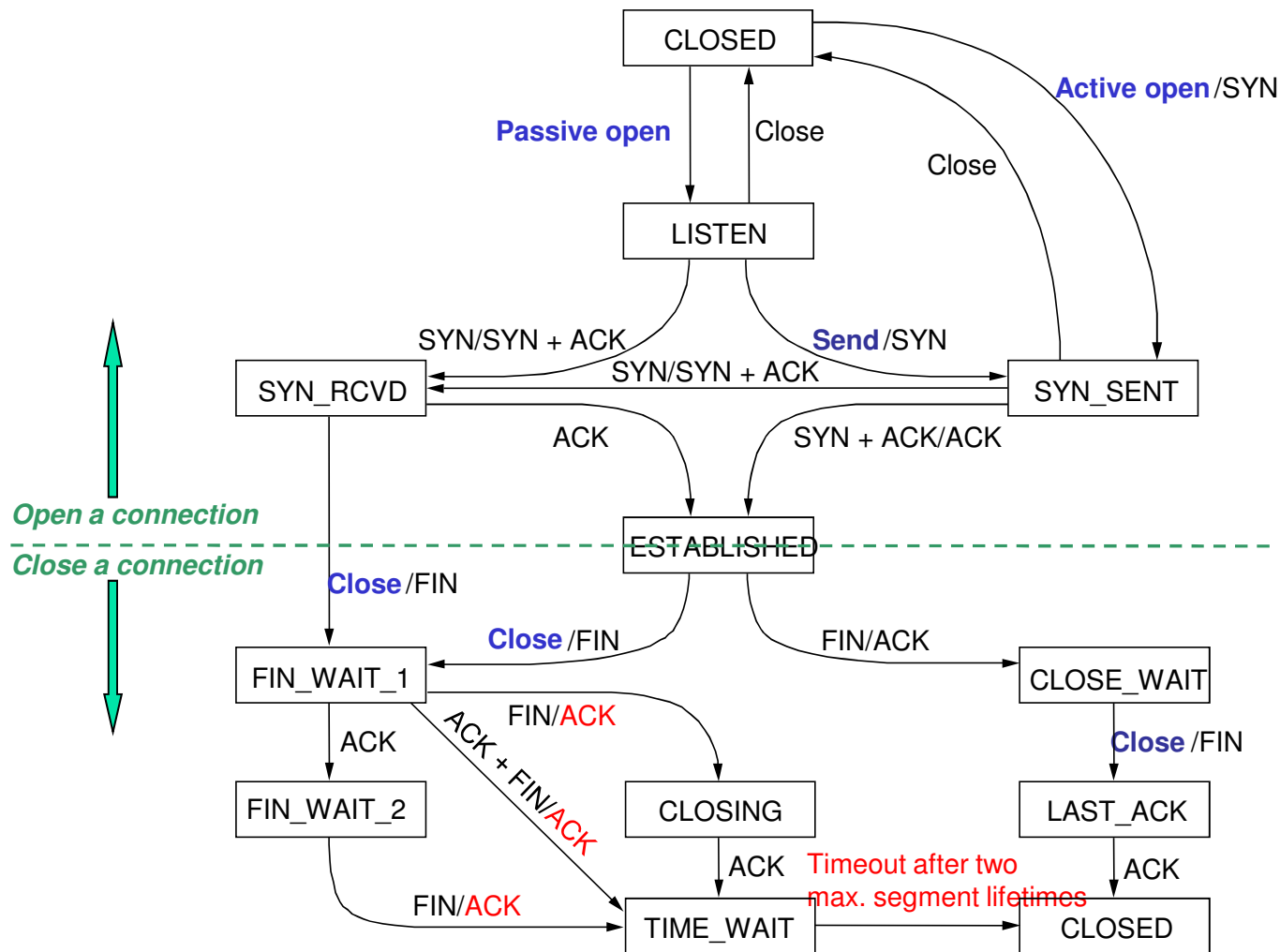
---

*Three-way handshake:*

to setup and terminate a bidirectional connection



# State Transition Diagram *event{/action}*



Note for "closing a connection":

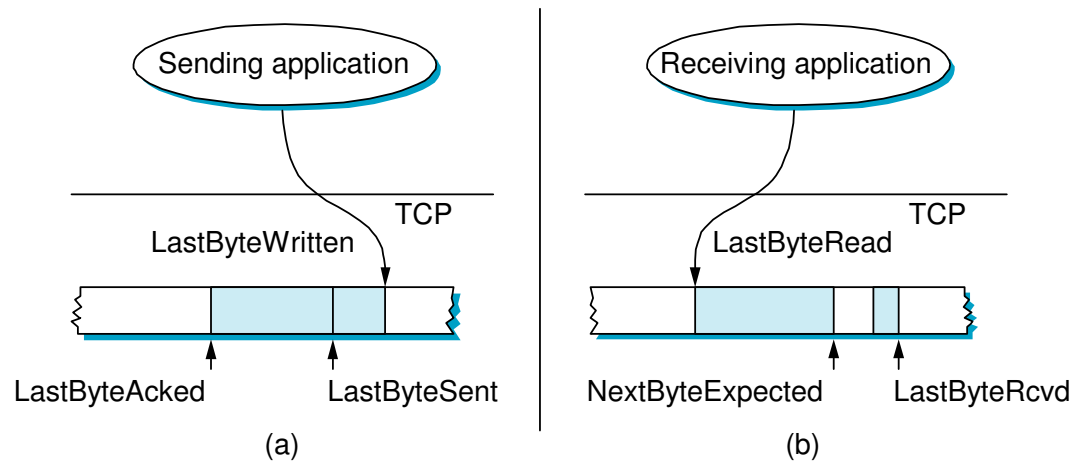
- Has to close the connection in both directions
- A connection in TIME-WAIT state must wait for  $2 * \text{max. segment lifetime}$  before moving to CLOSED state, to *reduce* the prob. of the the following undesirable event:
  - ACK is lost
  - another incarnation of the connection is established, but
  - receives a retransmitted FIN from the other end because it does not receive ACK
  - the new incarnation of the connection has to be closed

# Sliding Window Revisited

---

- Objectives
  - Reliable, in-order data delivery
  - Flow control
- Major difference from the link-layer sliding window (where sender and receiver tend to be homogeneous and have same buffer space) discussed before:
  - Size of the sliding window is not fixed; instead
  - The receiver advertises a window size to the sender
    - to deal with heterogeneity in wide area network

# Reliable and ordered delivery: similar to the link-layer sliding window algorithm



## ■ Sending side

- buffer bytes between **LastByteAcked** and **LastByteWritten**
- $\text{LastByteAcked} \leq \text{LastByteSent}$
- $\text{LastByteSent} \leq \text{LastByteWritten}$

## ■ Receiving side

- buffer bytes between **LastByteRead** and **LastByteRcvd**
- $\text{LastByteRead} < \text{NextByteExpected}$
- $\text{NextByteExpected} \leq \text{LastByteRcvd} + 1$

# Timeout value in timer-based retransmission

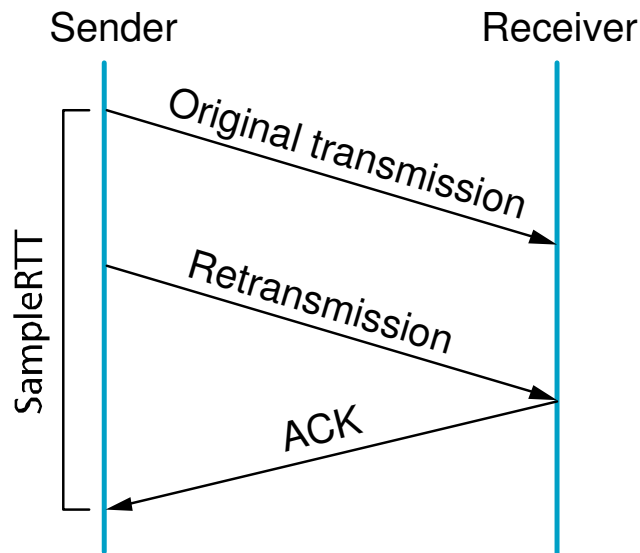
---

- Original Algorithm
  - Measure **SampleRTT** for each segment/ACK pair
    - Timestamp segment-transmission and ACK-reception, and calculate the difference
  - Compute weighted average of RTT
    - **EstimateRTT** =  $\alpha \times \text{EstimateRTT} + (1 - \alpha) \times \text{SampleRTT}$   
where  $\alpha$  between 0.8 and 0.9
  - Set timeout based on **EstRTT**
    - **TimeOut** =  $2 \times \text{EstRTT}$

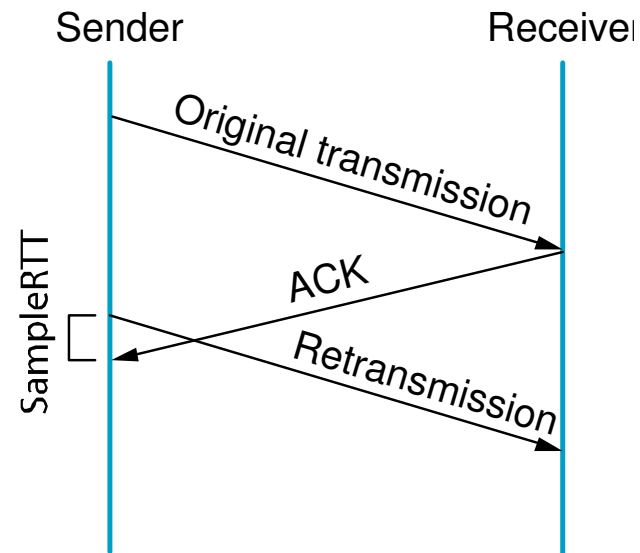
# Problem with original algorithm

---

- Ambiguity/difficulty in sampling
  - If a segment is retransmitted, it is hard to tell if the ACK is for the first transmission or another



(a)



(b)

# Karn/Partridge Algorithm

---

- Do not sample RTT when retransmitting
- Double timeout after each retransmission to avoid congestion
  - Assumption: packet loss are due to queue overflow (since wired links are highly reliable)
- (-) ad hoc; does not take the variance of RTT into account

# Jacobson/Karels Algorithm

---

- Estimate both mean and variance of RTT
  - $\text{Diff} = \text{SampleRTT} - \text{EstRTT}$
  - $\text{EstimateRTT} = (1 - \delta) \times \text{EstimateRTT} + \delta \times \text{SampleRTT}$
  - $\text{Dev} = (1 - \delta) \times \text{Dev} + \delta \times |\text{Diff}|$   
where  $\delta$  is a factor between 0 and 1
- Consider variance when setting timeout value
  - $\text{TimeOut} = \mu \times \text{EstRTT} + \phi \times \text{Dev}$   
where, typically,  $\mu = 1$  and  $\phi = 4$
- Notes
  - algorithm only as good as granularity of clock (500ms on Unix)
  - accurate timeout mechanism important to congestion control (later)

# Flow Control

---

- Recall: to keep sender from overrunning receiver
- How?
  - the receiver throttles the sender by advertising a window that is no larger than the amount of data that it can buffer
- Notation:
  - Send buffer size: **MaxSendBuffer**
  - Receive buffer size: **MaxRcvBuffer**

# Flow control (contd.)

---

- Receiving side: dynamically calculate AdvertisedWindow
  - $\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer}$
  - $\text{AdvertisedWindow} = \text{MaxRcvBuffer} - (\text{NextByteExpected} - \text{NextByteRead})$
  
- Sending side
  - $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$
  - $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{LastByteSent} - \text{LastByteAcked})$
  
  - $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$ 
    - If sending process tries to write  $\gamma$  bytes to TCP, but  $(\text{LastByteWritten} - \text{LastByteAcked}) + \gamma > \text{MaxSenderBuffer}$ , then TCP blocks the sending process

## Flow control (contd.)

---

- Q: how does the sender know the AdvertisedWindow is no longer 0 after receiving a 0-sized AdvertisedWindow?
  - In TCP, a receiver always send ACK in response to arriving data segment (but does not send ACK proactively)
  - To stick to the “smart sender/dumb receiver” rule, the sender persists in sending a segment with 1 byte of data every so often (instead of letting receiver send ACK periodically)
    - Another reason for using sender-based approach is that the sender needs to know the changed/increased AdvertisedWindow only if it has data to transmit (like “on-demand service”)

# Triggering transmission at sender side

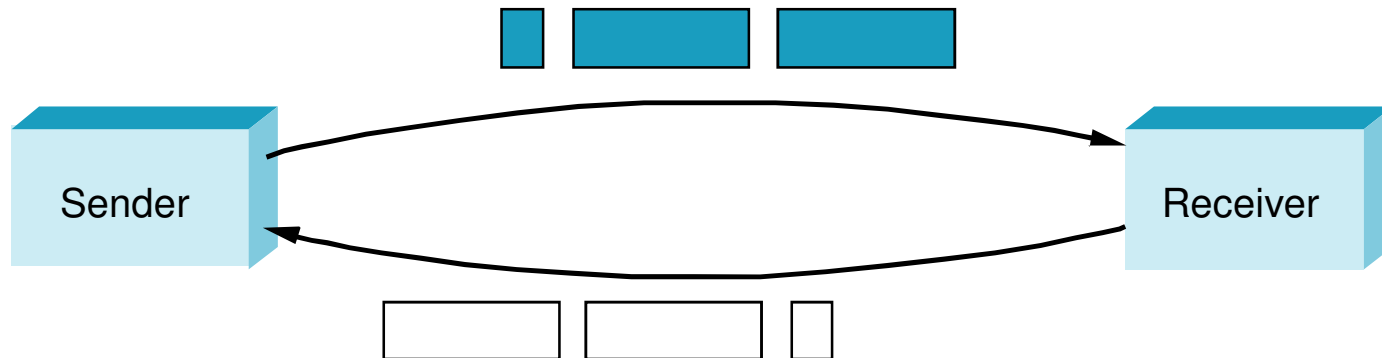
---

- Three cases (ignoring flow & congestion control for now)
  - Sender has a max. segment
  - Sending process explicitly asked for immediate transmission via the *push* operation
  - “timer (via self-clocking)” based triggering
    - To address the “silly window syndrome” issue

# Silly Window Syndrome

---

- What if sender aggressively exploits open window (i.e., immediately transmits data irrespectively of its size) ?



- Small segments introduced into the system remains in the system indefinitely (i.e., will never be coalesced)

# Receiver-side solutions

---

- After advertising zero window, wait for space equal to a maximum segment size (MSS)
- Small segments may still be generated
  - Delay acknowledgements to coalesce “small available spaces”
    - Challenge: how long to wait so as not to cause unnecessary retransmission?

# Sender-side algorithm: Nagle's Algorithm

---

- How long does sender delay sending data?
  - too long: hurts interactive applications
  - too short: poor network utilization
  - strategies: timer-based vs. self-clocking
- When application generates additional data
  - if *fills a max segment (and window open)*  
send it
  - else if *there is unack'ed data in transit*  
buffer it until ACK arrives
  - else  
send it
- Note: *receiver-driven self-clocking*: let application consumption speed guide network speed (cool!)

# Protection Against Wrap Around

---

- 32-bit SequenceNum, and 16-bit AdvertisedWindow
  - No problem with the correct operation of sliding window algorithm
- The issue is:
  - TCP assumes a Max. Segment Lifetime (MSL), e.g., 120 seconds
  - Thus, would not like the sequence number wrap around within MSL time
  - Q: whether this is the case in existing Internet?

## Time till 32-bit sequence number wraps around

---

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

- May be okay for today's network (where session speed may be < 155Mbps), but would need larger sequence number space in the near future as network BW increases
- Is addressed by TCP extension (to be discussed soon)

# Keeping the Pipe Full

---

- AdvertisedWindow should be big enough to keep the pipe full
  
- Q: is the 16-bit AdvertisedWindow large enough for today's network?

## Required window size for 100-ms RTT

---

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

- 16-bit AdvertisedWindow supports max. window size of 64KB
- not big enough to handle even a T3 connection across the continental US
- Is addressed by TCP extension (to be discussed soon)

# TCP Extensions

---

- Implemented as header options (rather than required)
  - For flexibility, backward-compatibility (and thus incremental deployability)
- Three extensions
  - Store timestamp in outgoing segments
    - Enables per-packet based RTT sampling (rather than per-ACK based RTT sampling)
  - Extend sequence number space with 32-bit timestamp
    - Protection against wrapped sequence numbers (PAWS)
    - Timestamp is used only to protect against wraparound, and NOT treated as a part of the sequence number
  - Scale 16-bit AdvertisedWindow
    - Left-shift up to 14 bits

# Alternative design choices for transport control

---

- TCP: reliable byte-stream
- A0: UNRELIABLE byte-stream
  - good for multimedia where delay and delay jitter are more serious than loss of a small percentage of packets

## Alternative design choices (contd.)

---

- A1: why do we use byte-stream instead of *message-stream*?
  - would require certain upper bound on message size if using message-stream; yet there is no perfect upper bound since different applications would have different message size requirement
  - can enforce record boundaries in TCP (e.g., use *UrgPtr* field or *push* operation)

## Alternative design choices (contd.)

---

- A2: why use explicit connection setup and teardown?
  - Connection setup so that the receiver can reject if it is unable to accept the connection request
  - Explicit teardown so that we do not need to use timer-based connection teardown
    - Timer-based connection teardown would make Telnet infeasible since Telnet may want to keep a connection alive for a long time (e.g., weeks)

## Alternative design choices (contd.)

---

- A3: why not use *rate-based* flow control (and congestion control)?
  - Complexity: how often should the rate be relayed back to sender if we were to use rate-based control?
  - Window vs. rate based flow and congestion control?
    - Still an active research question in different domains (e.g., wireless networks, sensor networks, etc.)

## Alternative design choices (contd.)

---

- A4: Why not use TCP for “request/reply inter-process communication”?
  - Request/reply applications always deal with *messages*, yet TCP is *byte* oriented; would need additional layer of transformation if we used TCP
  - High overhead as a result of connection setup & teardown in TCP for transmitting small request/reply messages
  - => RPC

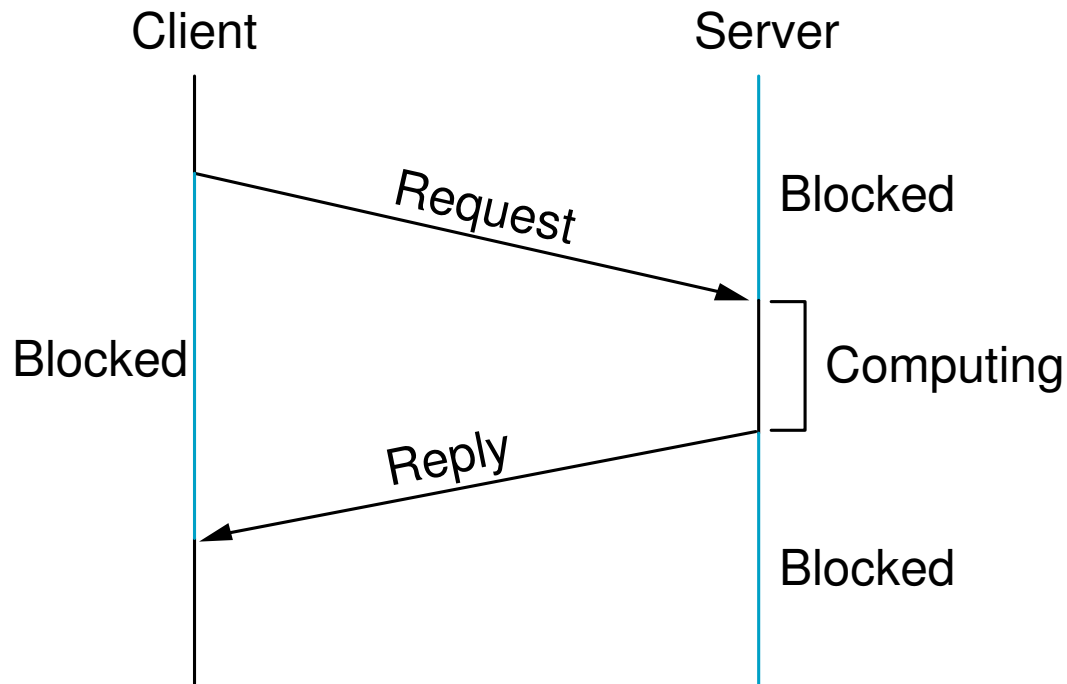
# Outline

---

- Simple demultiplexer (UDP)
- Reliable byte-stream (TCP)
- Remote procedure call (RPC)
- Transport for real-time applications (RTP)

# RPC

---



- Serves as the basis of many distributed systems

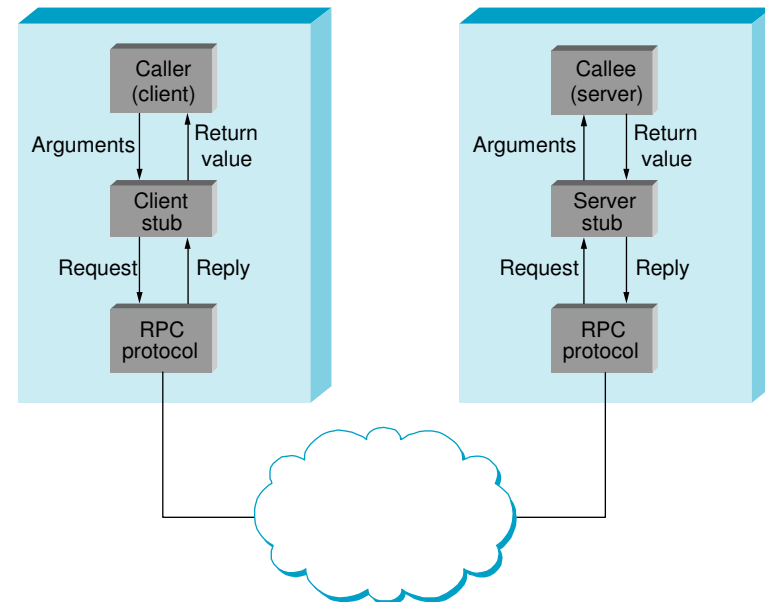
# Why another transport protocol (suite) for RPC?

---

- UDP does not guarantee any reliability
- TCP incurs high overhead (e.g., setting up and tearing down the connection) simply to delivery a pair of request/reply messages

# RPC Components

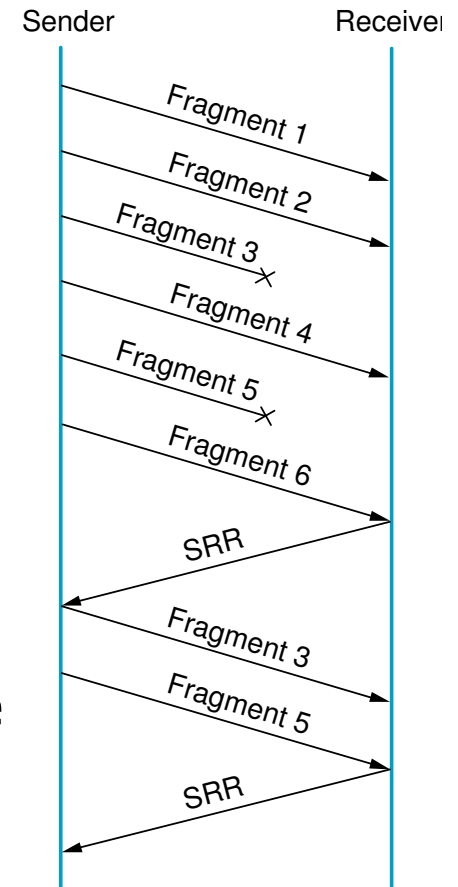
- Complete RPC mechanism



- We focus on “RPC Protocol (stack)”
  - fragments and reassembles large messages (by BLAST)
  - synchronizes request and reply messages (by CHAN)
  - dispatches request to the correct process/procedure (by SELECT)

# Bulk Transfer (BLAST)

- Fragmentation & reassembly as in ATM-AAL and IP
- Unlike AAL and IP, tries to recover from lost fragments
  - So as not to retransmit the whole large packet (for higher efficiency)
  - Strategy: selective retransmission via negative acknowledgment
- But does not go so far to guarantee 100% reliable delivery
  - Does not wait for any of the fragment to be acked before sending the next (hence the name *Blast*)
    - Why not flow and congestion control?



# BLAST Details

---

- Sender: temporarily keeps a fragment for potential retransmission
  - after sending all fragments, set timer DONE
  - if receive Selective Retransmission Request (SRR), send missing fragments and reset DONE
  - if timer DONE expires, free fragments;
    - Give up if there is lost fragments
  - SRR acts as negative acknowledgment
  - *Interprets lost negative-ack as "fragments have been received"*
    - *Thus, does not guarantee reliable fragment delivery*

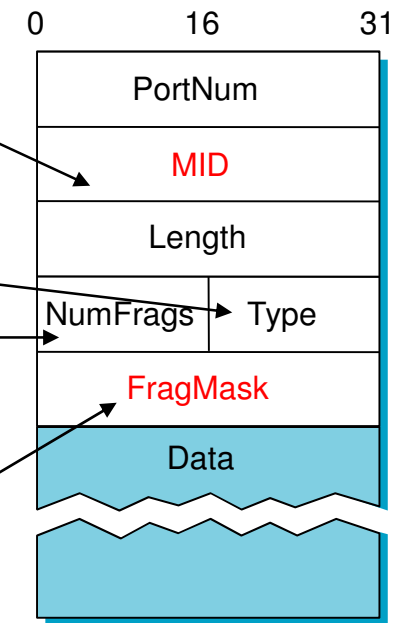
## BLAST Details (contd.)

---

- Receiver: in the presence of fragment loss, sends limited number of retransmission requests
  - when the first fragment arrives, set timer LAST\_FRAG
    - LAST\_FRAG is reset whenever receiving a new fragment
  - when all fragments are present, reassemble and pass up
  - four exceptional conditions:
    - if the 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 a third time
      - give up and free partial message

# BLAST Header Format

- MID (message ID): must protect against wrap around
- TYPE = DATA or SRR
- NumFrag: indicates total number of fragments in the message
- FragMask distinguishes among fragments
  - if Type=DATA, identifies fragments carried in this packet
  - if Type=SRR, identifies missing fragments



# Summary of BLAST

---

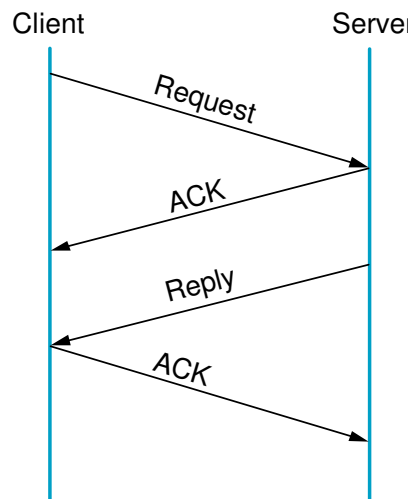
- For fragmentation and reassembly, but tries to recover lost fragments (to improve efficiency)
- No sliding-window flow control (unlike sliding window for reliable transmission)
  - Only need to deliver a single “large” message each time; no need for flow and congestion control
- No guarantee on reliable message delivery, instead efficiency-oriented fragment retransmission
  - Reliability is guaranteed by the protocol next layer up, i.e., CHAN

# Request/Reply (CHAN)

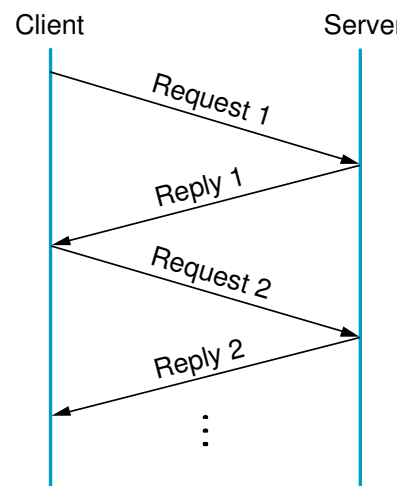
---

- Guarantees message delivery
- Synchronizes client with server
  - Implements a logical request/reply *channel* between client and server (thus the name CHAN)
  - At most one message is active on a given channel at any time
- Supports *at-most-once* semantics

Simple case



Implicit Acks



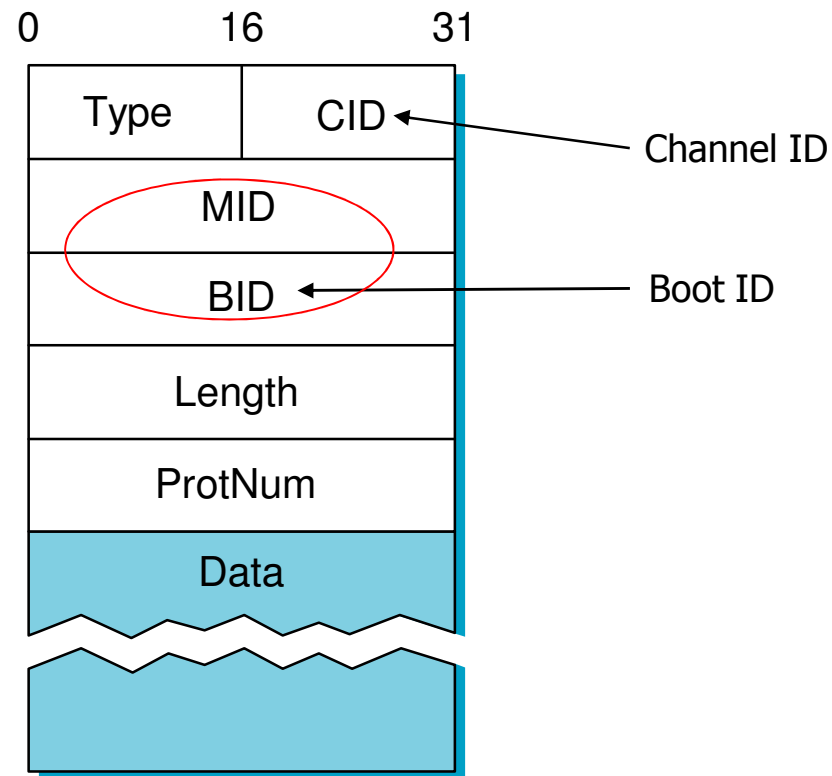
# CHAN Details

---

- Lost message (request, reply, or ACK)
  - set RETRANSMIT timer
  - use message id (MID) field to distinguish
- Machines crash and reboot
  - use boot id (BID) 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

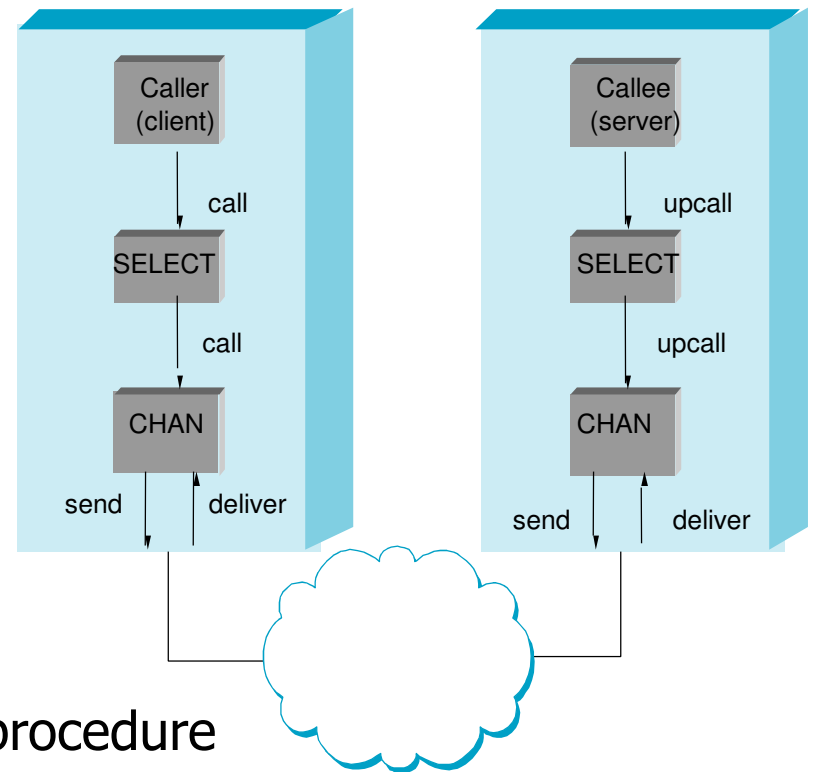
# CHAN Header Format

---



# Dispatcher (SELECT)

- Dispatch to appropriate procedure
- Synchronous counterpart to UDP
- Implement concurrency (open multiple CHANs)
- Address Space for Procedures
  - flat: unique id for each possible procedure
  - hierarchical: program + procedure number (commonly used)



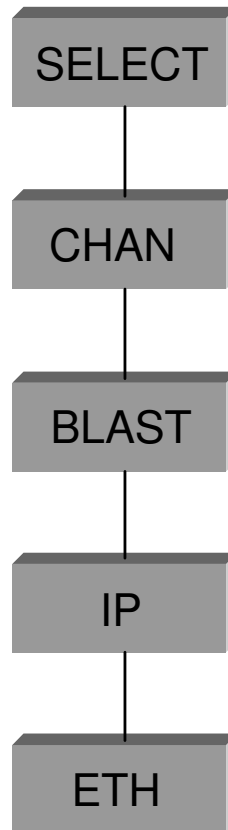
# Putting it all together

---

- Simple RPC stack
- SunRPC
- DCE-RPC

# Simple RPC Stack

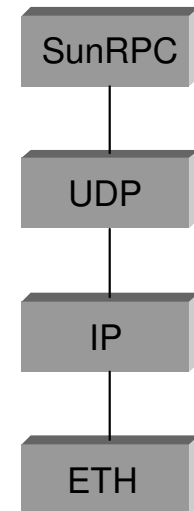
---



# SunRPC

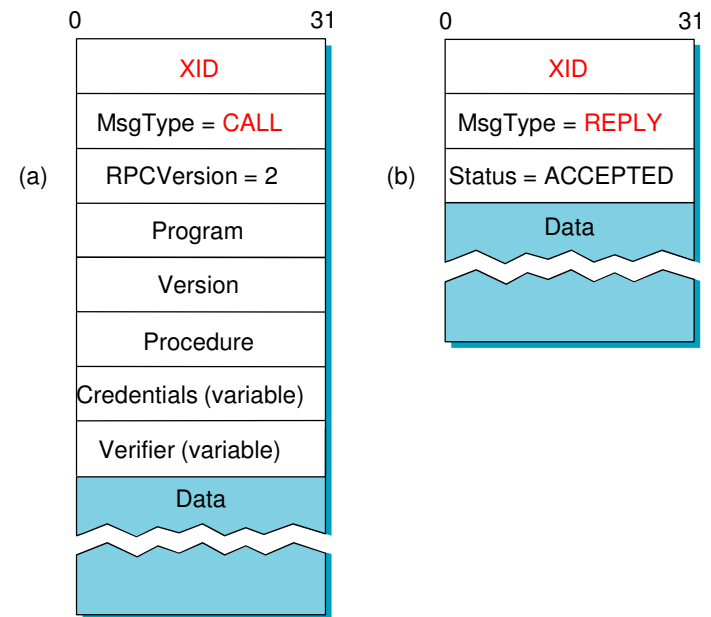
---

- Plays a central role in Sun's popular Network File System (NFS)
- Has become a de facto standard; IETF is considering making it a standard Internet protocol
- IP implements BLAST-equivalent
  - except no selective retransmit
- SunRPC implements CHAN-equivalent
  - except not at-most-once (i.e., duplicate is possible)
- UDP + SunRPC implement SELECT-equivalent
  - UDP dispatches to program (ports bound to programs)
  - SunRPC dispatches to procedure within 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 transit
  - Duplicate packet reception, which will be a problem if the operations executed is not idempotent



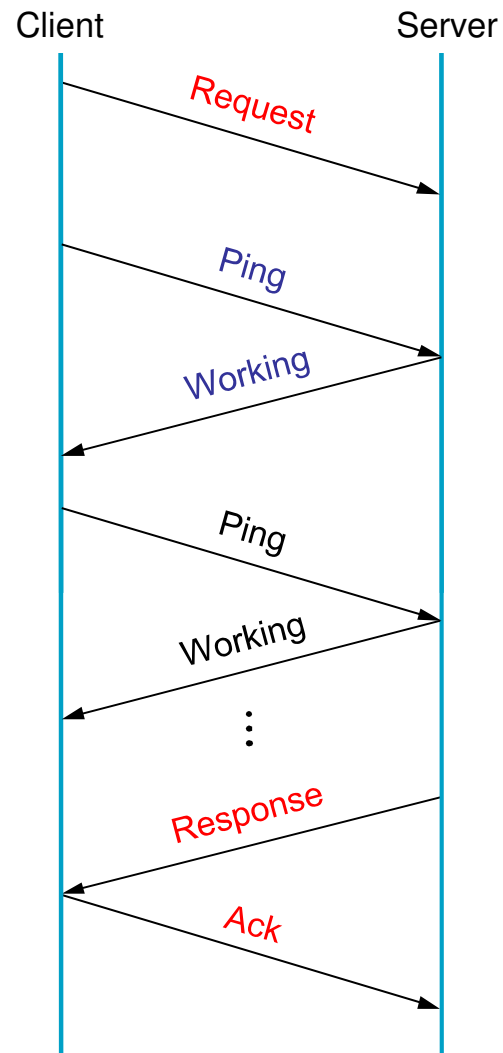
# DCE-RPC

---

- It is the RPC protocol at the core of DCE (Distributed Computing Environment) systems
- It serves as the underlying protocol for CORBA (Common Object Request Broker Architecture) proposed by OMG
- Similar to SunRPC, DCE-RPC defines a *two-level addressing* scheme: UDP demultiplexes to server/program, DCE-RPC dispatches to a particular procedure
- Unlike SunRPC, DCE-RPC implements at-most-once call semantics

# Typical DCE-RPC message exchange

---

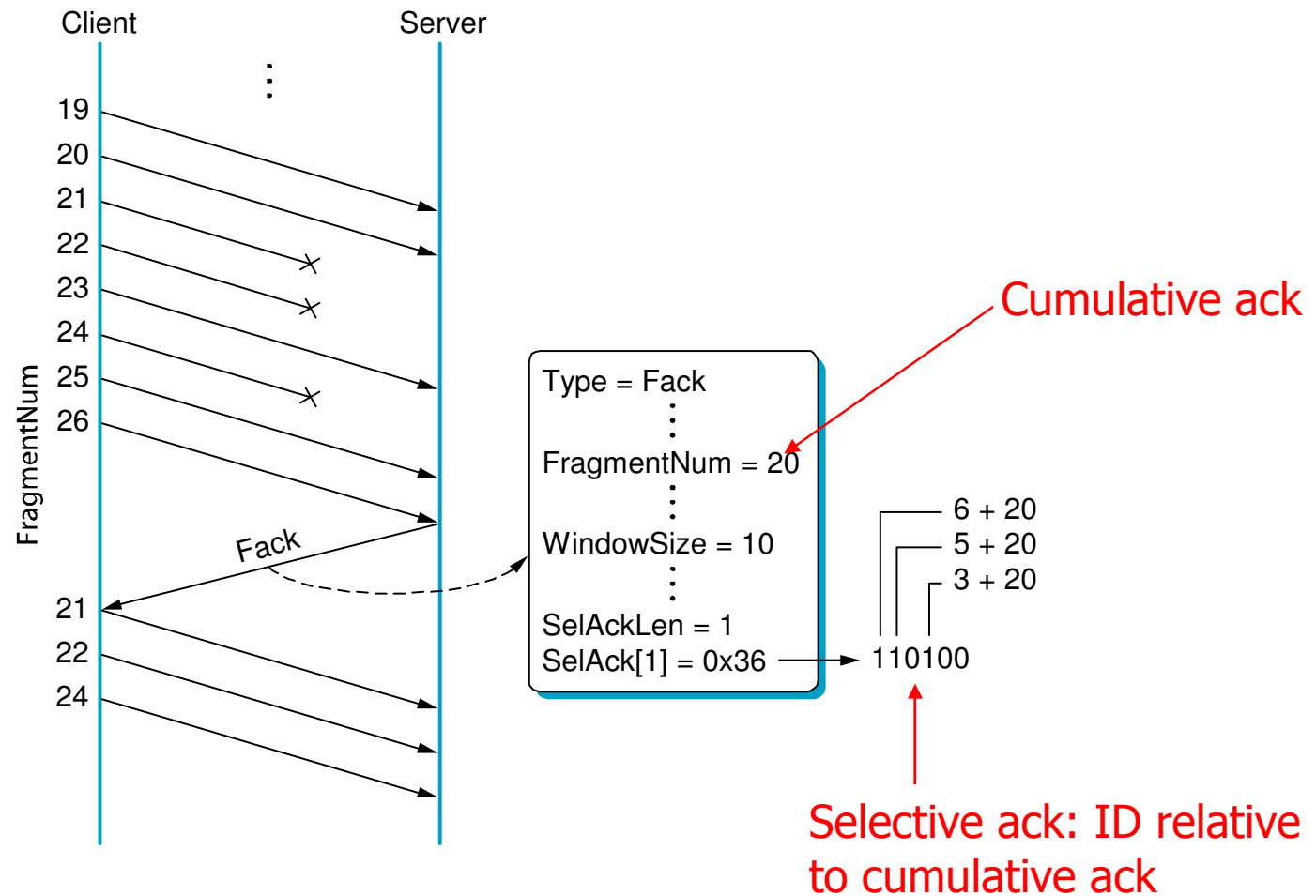


# Transport control in DCE-RPC

---

- Unlike BLAST (which uses a bit-vector to identify fragments), DCE-RPC fragment is assigned a unique fragment number
  - So as to support larger message size
- DCE-RPC could support very large messages (e.g., containing up to 64K fragments)
  - Receiver-window-size based flow control
  - Congestion control (similar to that in TCP)
- Use both cumulative ACK and selective ACK

# Fragmentation with cumulative & selective acks in DCE-RPC



Q: how is DCE-RPC different from TCP?

- Both have cumulative/selective ack, flow control, and congestion control

# Outline

---

- Simple demultiplexer (UDP)
- Reliable byte-stream (TCP)
- Remote procedure call (RPC)
- Transport for real-time applications (RTP)

# RTP

---

- Real-time traffic: digitized voice, video, etc.
- Experiments with real-time traffic since 1981
- Why not existing transport protocols?
  - UDP: best effort, no guarantee on delay and delay jitter
  - TCP: long delay and large delay jitter due to retransmission
  - RPC: designed for interactive exchange of (mostly short) messages

## Requirements for real-time traffic transport

---

- To be generic and to support different applications (e.g., w/ diff. encoding schemes)
- To identify timing relationship among received data;  
To synchronize related media streams (e.g., audio & video data streams)
- To detect and report packet loss (even though no need for 100% reliability)

# RTP: Real-time Transport Protocol

---

- Runs over UDP
- Application-Level Framing
  - leave application specific details to applications through “profile” and “formats”
    - Profile: specifies how to interpret the RTP header information
    - Format: specifies how to interpret the data following the RTP header
- 1) Data packets: specified by RTP
  - Timestamp: for timing and synchronization
    - At application-specific granularity (app defines “tick”)
  - Sequence number: for detecting lost or misordered packets
  - ...
- 2) Periodic control packets: specified by RTCP (Real-time Transport Control Protocol)
  - loss rate (fraction of packets received since last report)
  - delay jitter
  - ...

# Summary

---

- Simple demultiplexer (UDP)
  - Unreliable datagram
- Reliable byte-stream (TCP)
- Remote procedure call (RPC)
  - Reliable datagram + synchronization + request/reply procedure-call
- Transport for real-time applications (RTP)

# Discussion

---

- Different application requirements in sensor networks
  - Event-detection applications
    - Reliable (but no need for 100% reliability) and real-time data transport
    - Hop-based vs. end-to-end error and flow control
    - Hongwei Zhang, Anish Arora, Young-ri Choi, Mohamed Gouda, Reliable Bursty Convergecast in Wireless Sensor Networks, *ACM MobiHoc'05*
  - Data-collection applications
    - Reliable transport
    - Most of the time, no need for real-time data transport

# Further readings

---

- TCP

- USC-ISI, *Transmission Control Protocol*, RFC793, Sept. 1981

- RPC

- A. Birrell, B. Nelson, *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, Feb. 1984

# Assignment – Chapter 5

---

- Exercise#4
  - Chapter 5: Exercises 2, 9(a), 12, 39, and 46