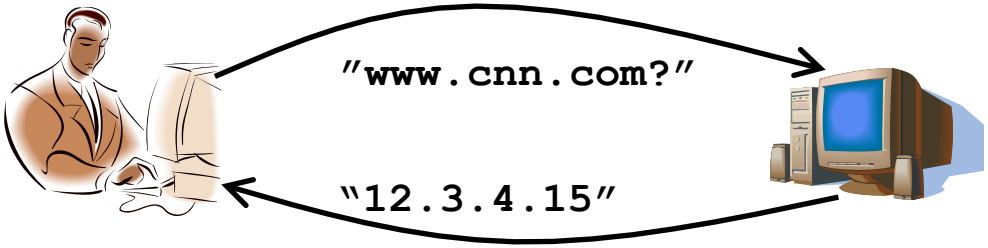
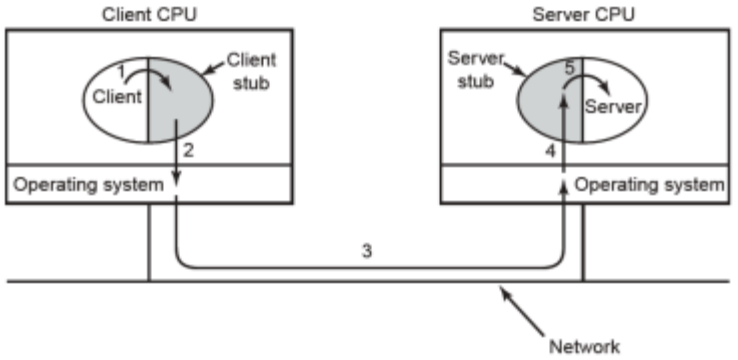
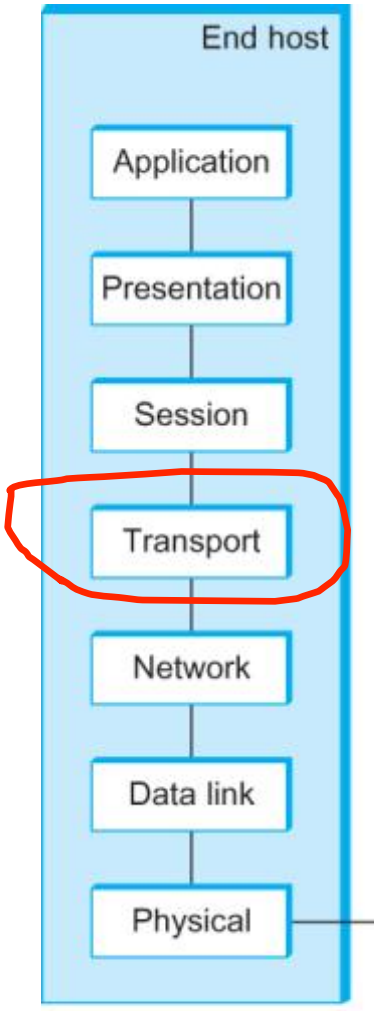
Transport layer and UDP





Overview

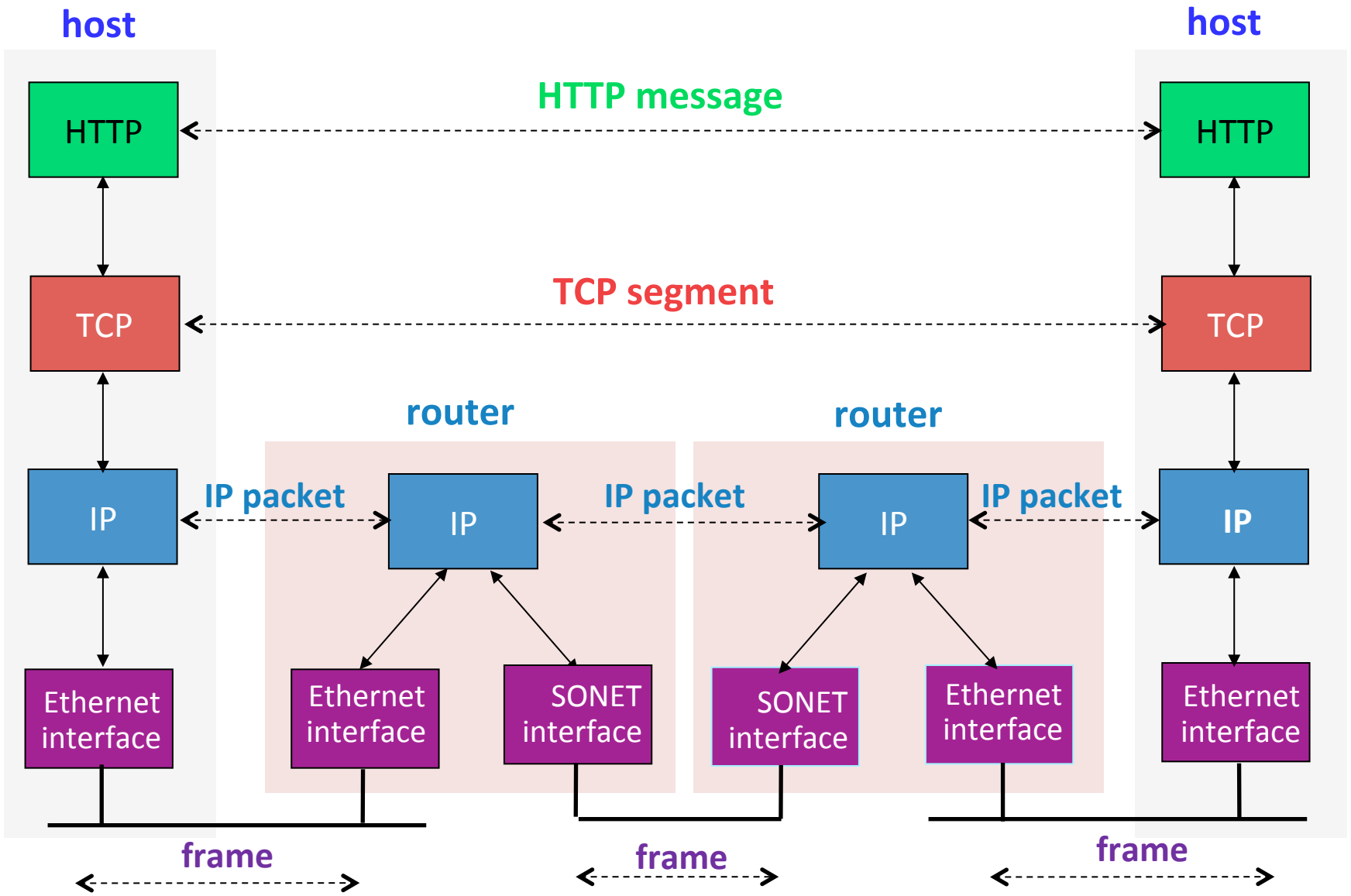
- Principles underlying transport layer
  - Multiplexing/demultiplexing
  - Detecting errors
  - Reliable delivery
  - Flow control
- Major transport layer protocols:
  - User Datagram Protocol (UDP)
    - Simple unreliable message delivery
  - Transmission Control Protocol (TCP)
    - Reliable bidirectional stream of bytes



Transport layer challenges

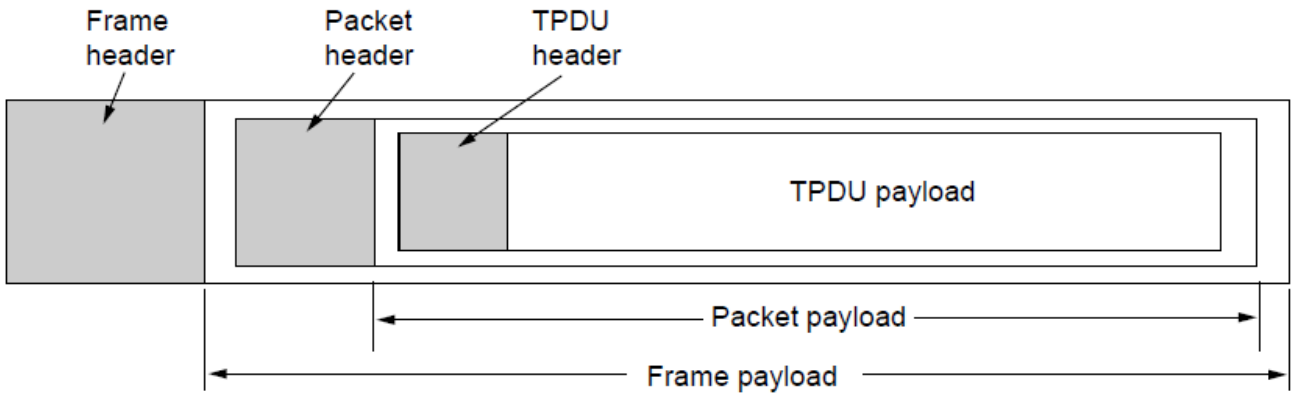
- Running on best-effort network:
  - Messages may be dropped
  - Messages may be reordered
  - Duplicate messages may be delivered
  - Messages have some finite size
  - Messages may arrive after long delay
- Sender must not overrun receiver
- Network may be congested
- Hosts must support multiple applications

Internet layering model



Segments

- Segment
  - Message sent from one transport entity to another transport entity
  - Term used by TCP, UDP, other Internet protocols
  - AKA TPDU (Transport Protocol Data Unit)

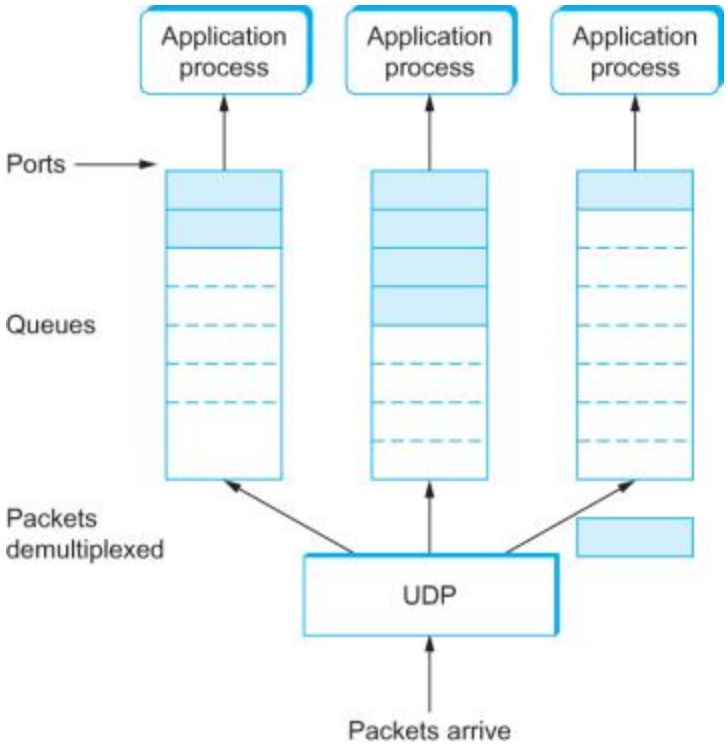


Transport layer

- Goal: Provide end-to-end data transfer
  - Just getting to host machine isn't enough
  - Deliver data from process on sending host to correct process on receiving host
- Solution: OS demultiplexes to correct process
  - Port number, an abstract locater
  - OS demuxes combining with other info
    - UDP <port, host>
    - TCP <source port, source IP, dest port, dest IP>

Simple demultiplexer

- User Datagram Protocol (UDP)
  - Mapping to process using 16-bit port number
  - Detecting errors: (optional) checksum



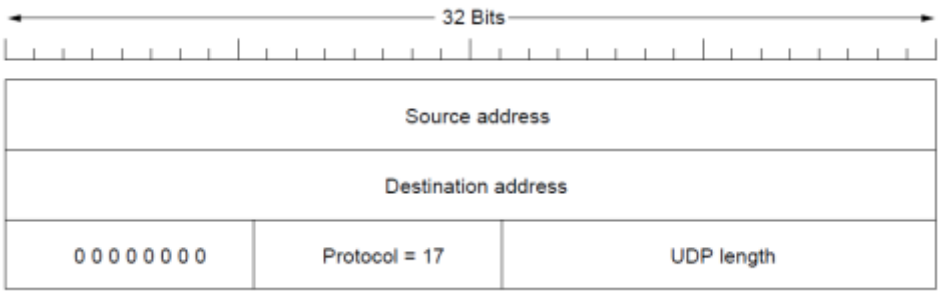
Why use UDP?

- Provides:
  - Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Precise control of when data is sent
    - As soon as app writes to socket, UDP packages and sends
  - No delay establishing a connection
  - No connection state, scales to more clients
  - Small packet overhead, header only 8 bytes long
- Does not provide:
  - Flow control, congestion control, or retransmission on error

UDP checksums

- UDP checksum
  - Add up 16-bit words in one's complement
  - Take one's complement of the sum
  - Done on UDP header, data, IP pseudoheader
    - Helps detect misdelivered packets
    - Violates layers, looking into network layers

UDP header



IP pseudoheader

Port numbers

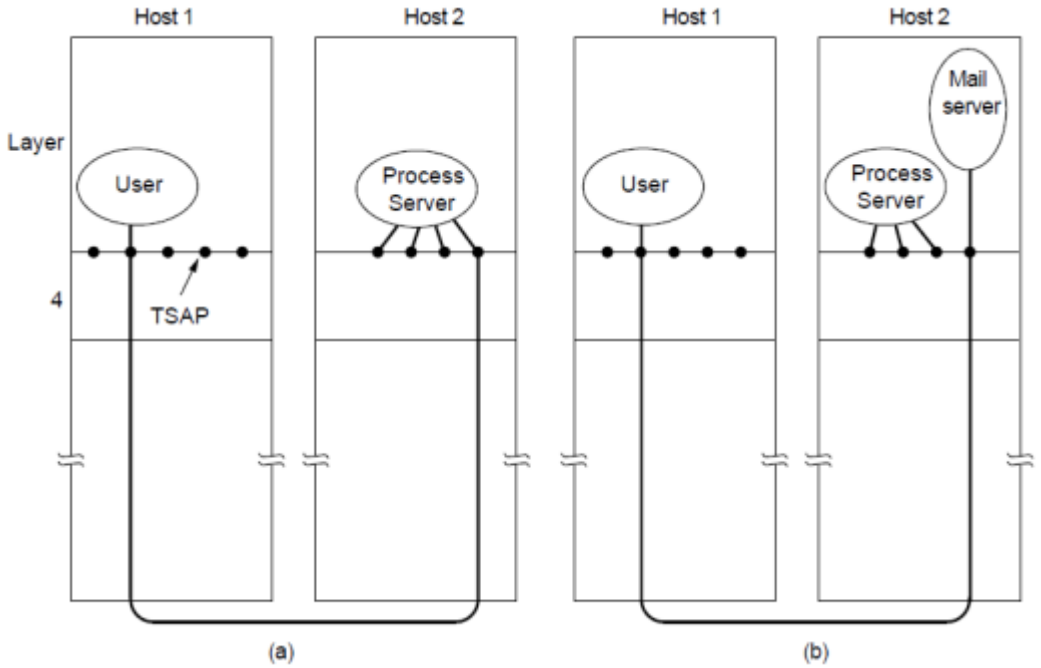
- How do clients know the port number?
  - Well known port number
  - Port mapper service
    - Listens for request on known port number
    - Maps service name such as "BitTorrent" to port number

Port	Description
20/21	FTP
22	SSH
23	Telnet
25	Simple Mail Transfer Protocol (SMTP)
53	Domain Name System (DNS)

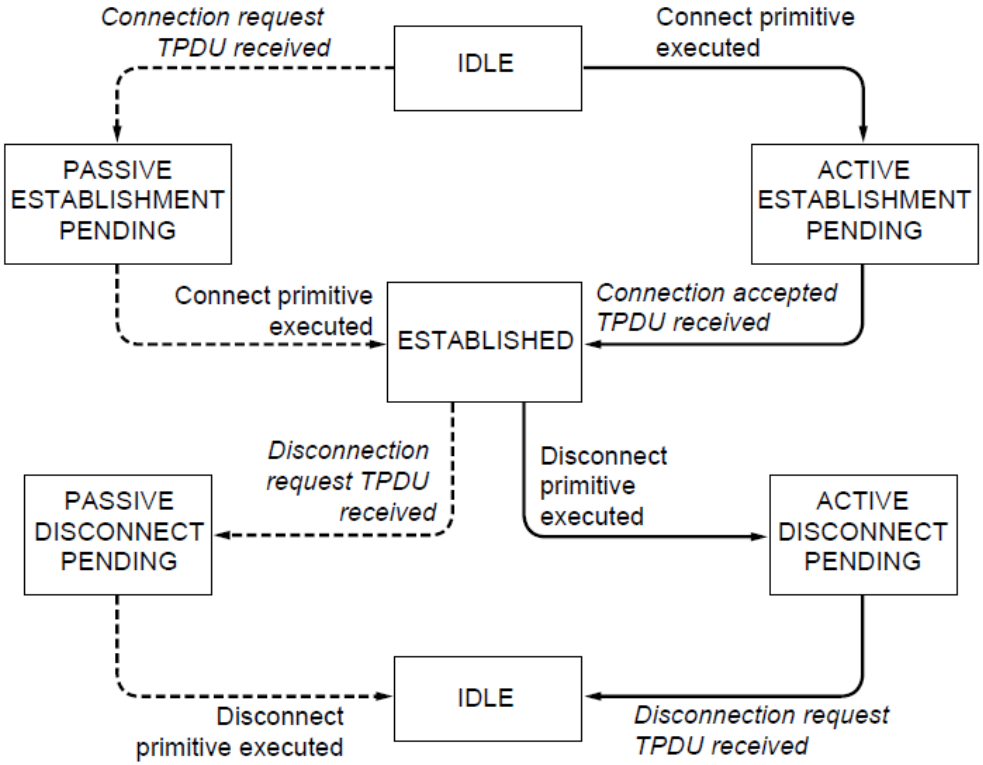
Port	Description
80	HTTP
110	Post Office Protocol (POP3)
143	Internet Message Access Protocol (IMAP)
443	HTTPS
546/7	DHCP

Port numbers

- How do clients know the port number?
  - Initial connection protocol (e.g. inetd, xinetd, launchd)
    - Process server acts a proxy for less used services
    - Listens on set of ports at same time
    - Clients specify service in connection request



Berkeley Sockets



A state diagram for a simple connection management scheme.

Transitions labeled in italics are caused by packet arrivals.

The solid lines show the client's state sequence.

The dashed lines show the server's state sequence.

Berkeley Sockets

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Server	Client
SOCKET	SOCKET
BIND	**CONNECT**
LISTEN	SEND/RECEIVE
ACCEPT	CLOSE
(SEND/RECEIVE)*	
CLOSE	

Sequence for connected sockets.

Server	Client
SOCKET	SOCKET
BIND	SENDTO/RECVFROM
(SENDTO/RECVFROM)*	CLOSE
CLOSE	

Sequence for unconnected sockets.

Sending and receiving

- Connected sockets

```
int send(int sockfd, const void *msg, int len, int flags);
int recv(int sockfd, void *buf, int len, int flags);

sockfd - socket descriptor
msg    - pointer to buffer to be sent/received
len    - length of buffer
flag   - normally 0

Returns bytes sent or received.

NOTE: underlying protocol could be TCP or UDP
```

Sending and receiving

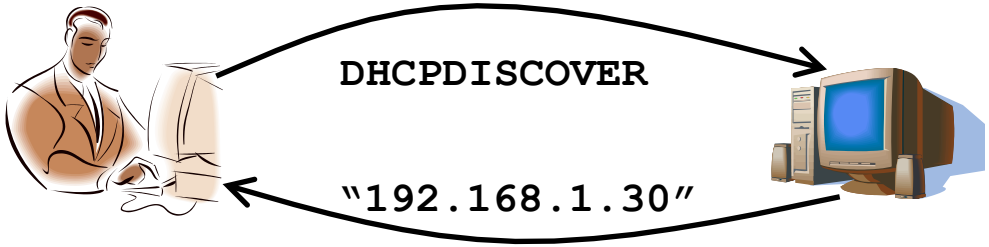
- Unconnected datagram sockets

```
int sendto(int sockfd, const void *msg, int len, int flags,
           const stuct sockaddr *to, socklen_t tolen);

int recvfrom(int sockfd, void *buf, int len, int flags,
             struct sockaddr *from, int *fromlen);

sockfd - socket descriptor
msg    - pointer to buffer to be sent/received
len    - length of buffer
flag   - normally 0
to     - address to send the datagram to
from   - address of who sent the datagram

Returns bytes sent or received.
```

Type of UDP apps, part 1/3

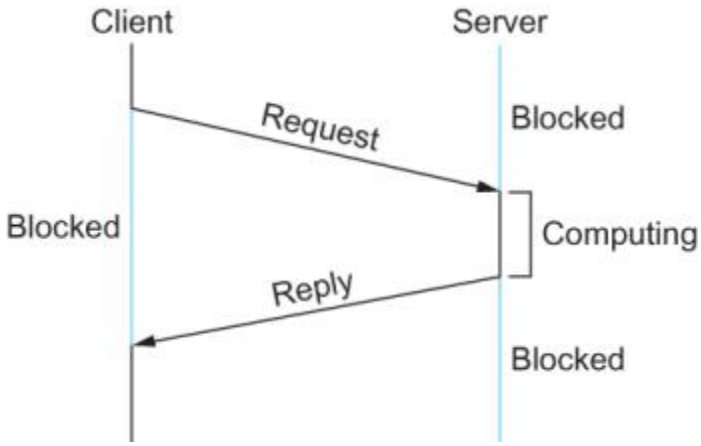
- Simple query protocols
  - Overhead of connection establishment is overkill
  - Easier to have application retransmit if needed
  - e.g. DNS, UDP port 53

  - e.g. DHCP, UDP port 67/68



Type of UDP apps, part 2/3

- Request/reply style interaction
  - Client sends request to server
    - Blocks while waiting for reply
  - Server responds with reply
  - Must deal with:
    - Identify process that can handle request
    - Possible loss of request or reply
    - Correlate request with reply

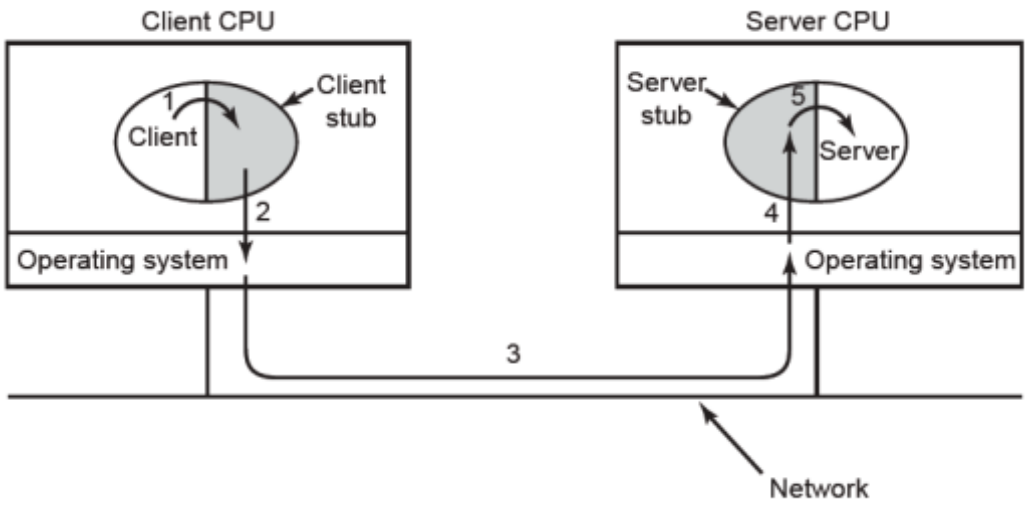


Request/reply example

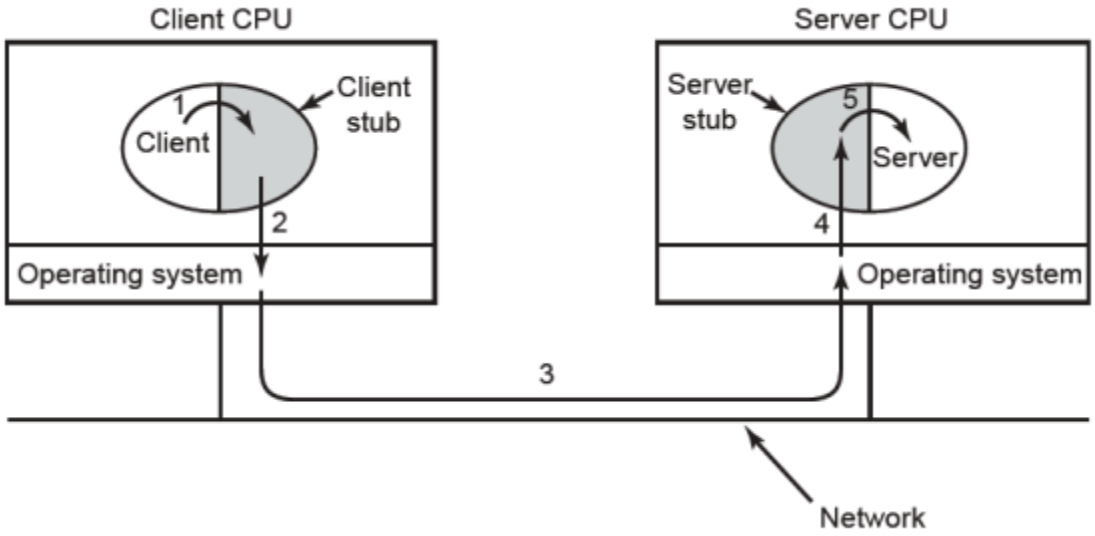
- Remote Procedure Call (RPC)
  - Request/reply paradigm over UDP
  - Allow programs to call procedures located on a remote host
  - Invisible to the application programmer
    - Client code blocks while request made and response waited for from remote host
  - Object-oriented languages:
    - Remote Method Invocation(RMI), e.g. Java RMI

RPC mechanism

- Client stub
  - Represents server procedure in client's address space
- Server stub
  - Hides fact that procedure call from client is not local to the server
- Usually create by a stub compiler



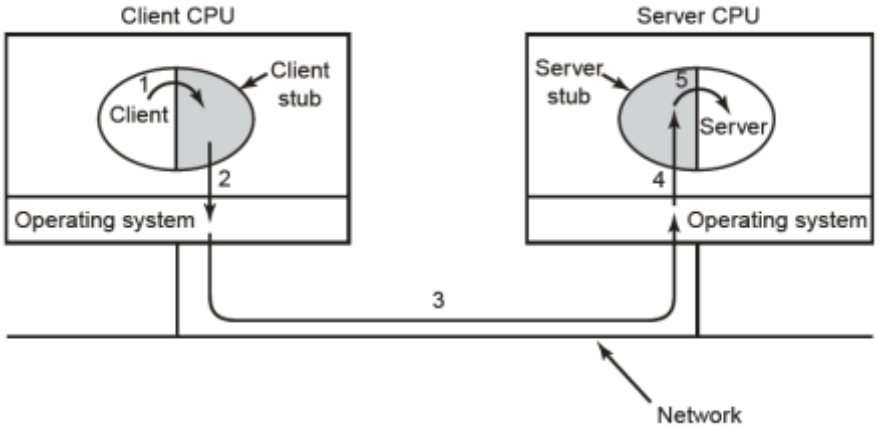
RPC steps



- Step 1 - Client calls the client stub, a local procedure call, parameters pushed on to stack in normal way
- Step 2 - Client packages parameters into a message, "marshaling"
- Step 3 - Client OS sends the message to server machine
- Step 4 - Server OS passes message to server stub
- Step 5 - Server OS calls server procedure with unmarshaled parameters

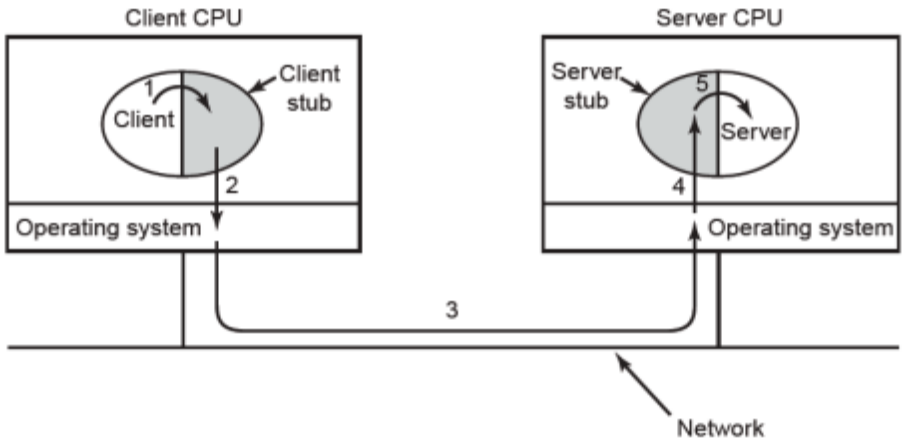
RPC challenges

- RPC challenges
  - Marshaling pointers
    - Call by reference
      - e.g. Pointer to integer
    - Passing pointer to a complex structure, e.g. graph
  - Parameter size may be uncertain
    - e.g. Vector dot product using special termination flag
  - Global variables
  - Operations may not be idempotent
    - Not safe to repeat, e.g. incrementing a counter



RPC problems

- RPC protocol must:
  1. Name space to uniquely ID each procedure
  2. Match each reply with corresponding request
  3. Work on a best-effort network
     - Messages may be lost, corrupted, duplicated, delayed, may exceed size limit, etc.



Specifying RPC names

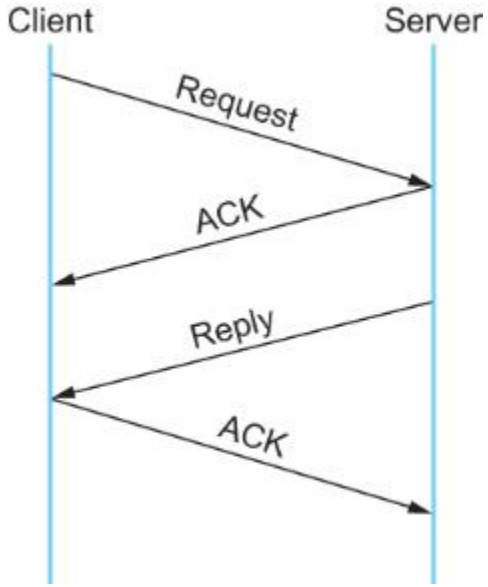
- Problem 1: Identifying a procedure
  - Flat scheme
    - Single ID field carried in RPC request
    - Requires central authority to assign ID
    - e.g. Simple integer ID
  - Hierarchical
    - Two or more level name space
    - e.g. SunRPC, 32-bit program #, 32-bit procedure #
      - Program number 0x00100003 = NFS
      - Procedure number 1 = getattr, 2 = setattr, 6 = read, …

Matching RPC replies

- Problem 2: Matching requests to replies
  - Message ID on request
  - Reply contains same message ID
  - Simple scheme, client starts at 0 and counts up
    - But what if client reboots?
      - Client sends message with ID 0
      - Client crashes, reboots, sends new request ID 0
      - Servers already replied to first one
      - Server discards as duplicate
      - Client never gets response to new request
    - Add boot ID from nonvolatile storage to message ID

Reliable RPC

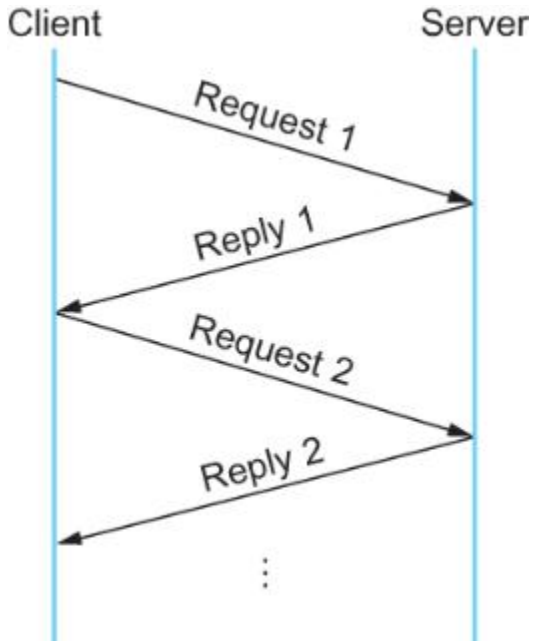
- Problem 3: Work on best-effort network
  - UDP/IP does not provide reliable transport
  - Messages may get corrupted, duplicated, dropped, delayed, etc.
  - Use timeouts and ACKs



Timeline for reliable RPC protocol

Reliable RPC

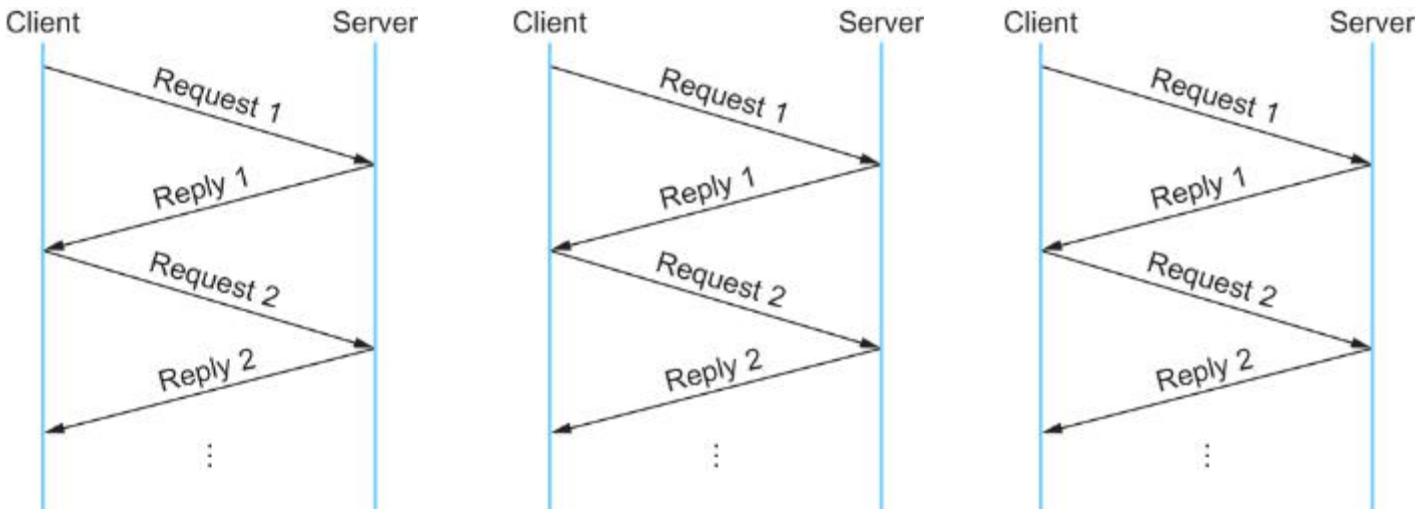
- Explicit ACKs seen unnecessary
  - Reply is implicit ACK of the request
  - Next request is implicit ACK of the previous reply
  - But we can't send a new request until previous one is done
    - Server may block on I/O for a long time
    - Severely limits RPC performance



Timeline for reliable RPC protocol using implicit acknowledgements.

Reliable RPC

- Concurrent logical channels
  - Only one request per channel
  - Multiple channels with different channel IDs



Channel 1 Channel 2 Channel 3

Reliable RPC

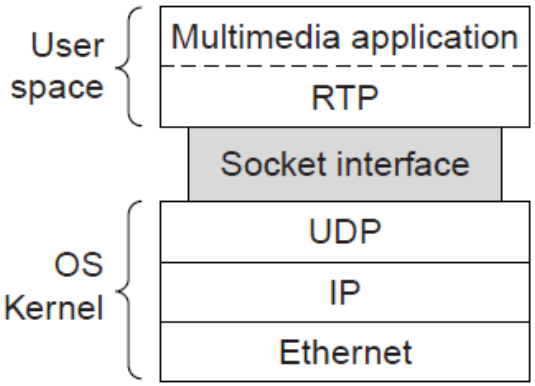
- What if server crashes serving request?
  - Client would be waiting forever
  - Client could send "Are you alive?" messages
  - Server could send "I'm alive" messages
- What if duplicate requests are made?
  - Servers may implement at-most-once semantics
    - Server remembers current sequence number on a particular channel
    - Only service next expected request

Type of UDP apps, part 3/3

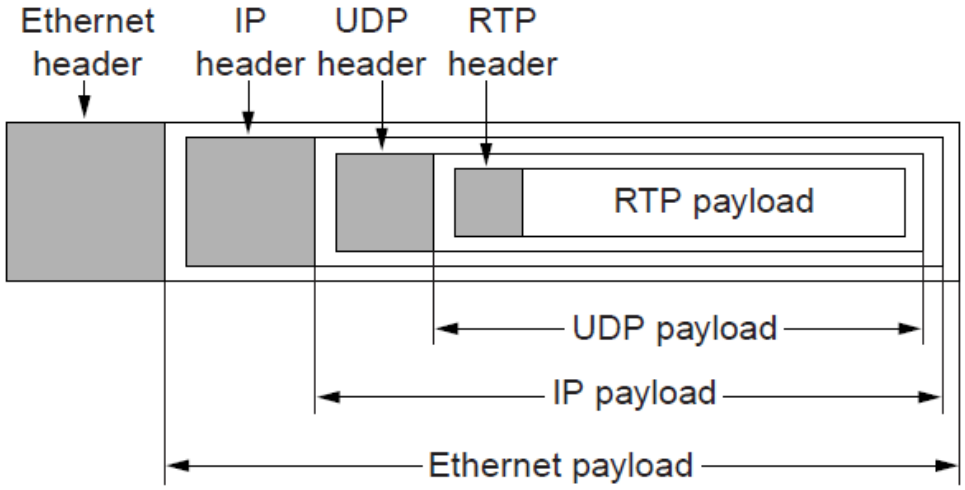
- Multimedia streaming
  - e.g. Voice over IP, video conferencing
  - Time is of the essence
    - By time packet is retransmitted, it's too late!
    - Interactive applications:
      - Human-to-human interaction
      - e.g. conference, first-person shooters
    - Streaming applications:
      - Computer-to-human interaction
      - e.g. Netflix, Spotify

RTP

- Real-time Transport Protocol (RTP)
  - Generic real-time transport protocol
  - How to transport audio/video data in packets
  - Processing to play at right time



(a)

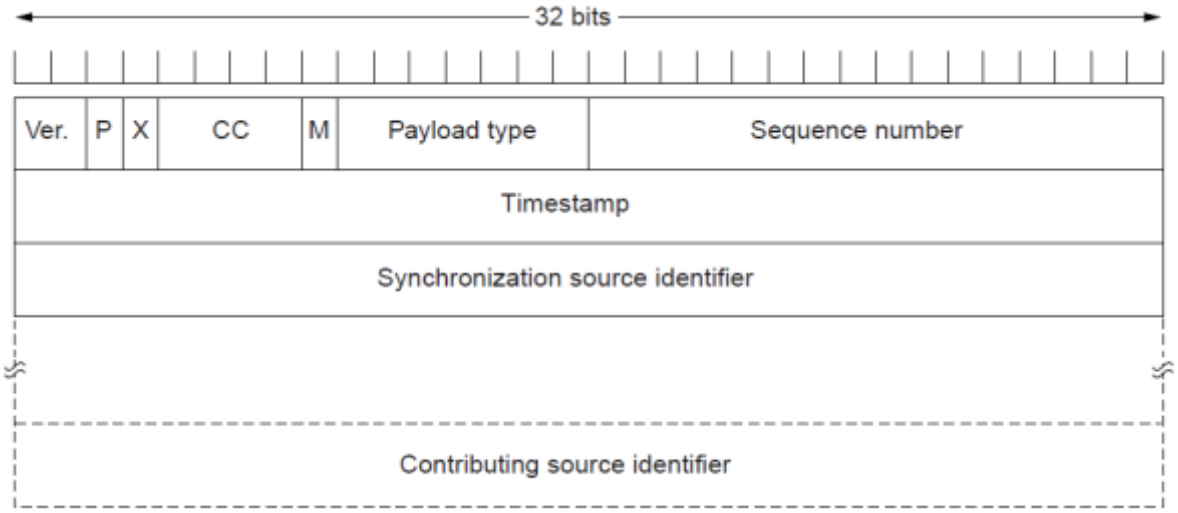


(b)

RTP

- Real-time Transport Protocol (RTP)
  - Multiplexes several real-time streams into single UDP stream
  - Sent to single destination (unicast) or multiple destinations (multicast)
  - Provides:
    - Number of packets
    - Header specifying type of encoding (e.g. MP3,
    - Time stamping of samples

RTP header



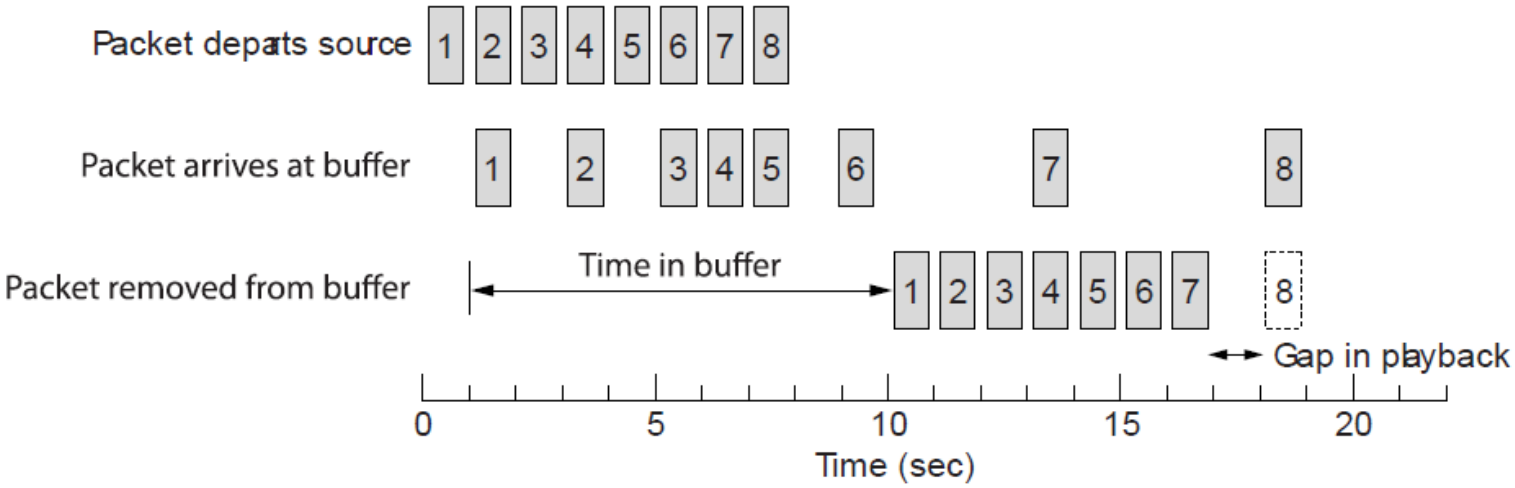
- Payload type - encoding algorithm used
- Sequence number - counter incremented on each packet sent
- Timestamp - produce by stream's source, when first sample in packet was made. Decouples playback from packet arrival.
- Synch. source ID - which steam this packet belongs to
- Contributing source ID - used when mixes present

RTCP

- Real-time Transport Control Protocol (RTCP)
  - Provides feedback on network
    - Delay, variation in delay, jitter, bandwidth, congestion
    - Allows RTP to adjust to different encoding schemes
  - Interstream synchronization
    - Different streams may use different clocks
    - Helps keep them in synch
  - Naming of stream sources
    - e.g. ASCII text for whoever is speaking right now

RTP playback

- Playback using RTP
  - Packets may arrive out of order
  - Buffer a certain amount so receiver can reorder
    - Long buffer possible for streaming apps
    - Short buffer needed for interactive apps



Summary

- Transport layer
  - Providing end-to-end process communication
    - Port numbers allow multiple processes per host
  - Provide reliable transport on best-effort network
- User Datagram Protocol (UDP)
  - Lightweight protocol running on top of IP
  - Three typical classes of applications:
    - Simple queries (DNS, DHCP)
    - Request/reply semantics (RPC)
    - Real-time data (RTP)