Transport Layer, UDP, and TCP

CSC 343·643

Wake Forest University
Department of Computer Science

Fall 2013

Transport Layer

- Reliable, cost-effective data transport from source to destination
  - End-to-end protocol
  - Only implemented at hosts (**not** routers)
- *What is the difference between transport and network layers?*
  - Transport provides logical communication between processes
  - Network provides logical communication between hosts

    What does this difference imply? What is an example?
- *Transport entity* provides services to upper layers
  - Type of service, data transfer, connection management, and flow control

Type of Service

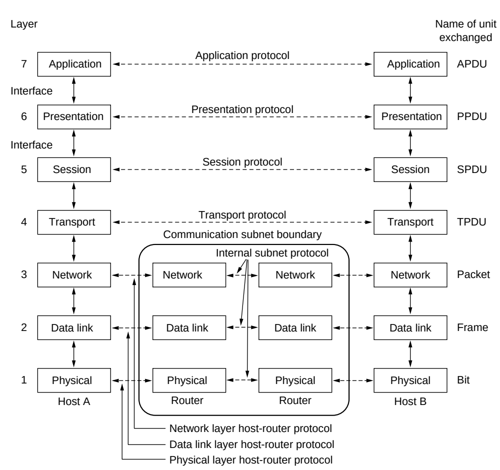
Two types of service available *connection-oriented* and *connectionless*

- Connection-oriented
  - Establishment, maintenance, and termination of connection
  - Typically implies reliable service
- Connectionless
  - Unreliable service (delivery not guaranteed)
  - Reduces the overhead associated with transport layer

  Network layer has connection-oriented and connectionless, why is it specified here? Is it redundant?

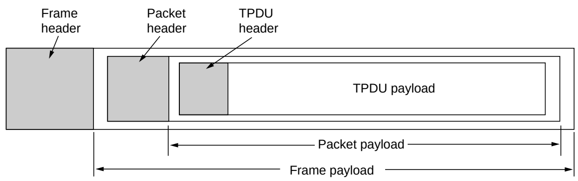
- Might want to provide a service not provided by lower layer
  - What if the network-layer is connection-oriented but unreliable?
  - What happens if a router crashes?
  - *Provide a reliable service over an unreliable network*

  Can you provide a reliable service using an unreliable network?

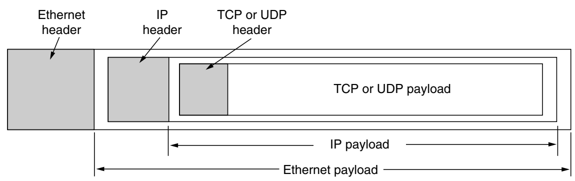


Transport Protocol Data Unit

- TPDU are messages sent from transport entity to transport entity
- Message encapsulated as it passes through other layers
  - TPDU → network layer datagram → link-layer frame



- For example TCP or UDP using IP using Ethernet

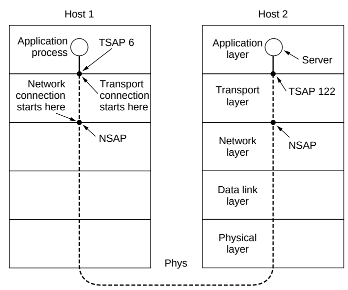


Transport Addresses

- When an application **process** wishes to set-up a connection to a remote application **process**, it must specify which one
  - `152.17.140.92` provides `http` and `ssh` services
  - Therefore network layer address is **not** sufficient
- In the Internet, transport services identified using **port numbers**
  - Some port numbers are *well-known*, 80 is for `http`
  - DOS command `netstat -a` shows ports in use

    So what is a port scan?
- The generic term for transport addresses is **Transport Service Access Points** (TSAP)
  - The generic term for network addresses is NSAP

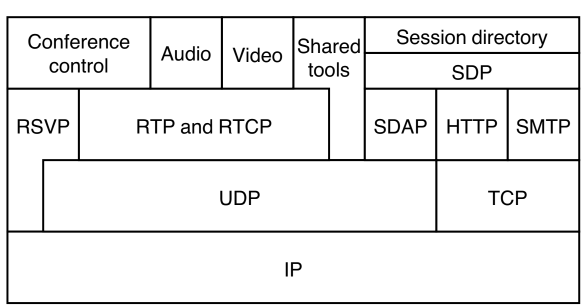
- Assume a server provides a *time-of-day* service
  1. Let time-of-day server use TSAP 122 and await connection
  2. Client wants the time-of-day, issues a *connect* request, specifying TSAP 6 as source and TSAP 122 destination
  3. Must also identify NSAP addresses
  4. After connection established, time-of-day sent back to client



How does the client know the TSAP?

Internet Transport Layer

- There are two distinct Internet transport-layer protocols
  - User Datagram Protocol (UDP)
  - Transmission Control Protocol (TCP)



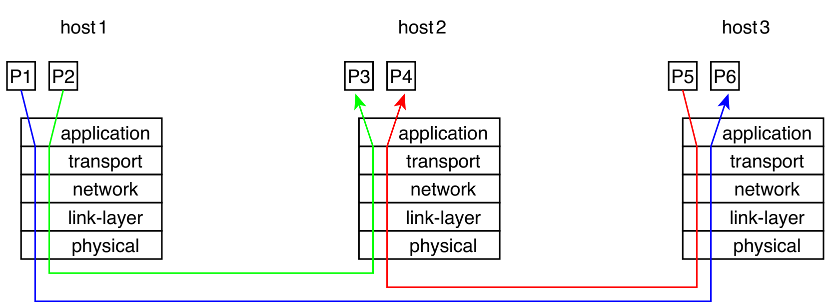
- UDP provides unreliable connectionless service
- TCP provides reliable connection-oriented service

Multiplexing and Demultiplexing Applications

Different interpretation of multiplexing and demultiplexing...

- Remember IP delivers data between two end systems
  - Each identified with a unique IP address
  - IP does **not** deliver data between two applications
- Transport layer receives data from network layer
  - Must deliver data to the *appropriate* application (process), this is called **demultiplexing**
- Consider an end user running `telnet` and `http`
  - Transport layer protocol (TCP) will receive data from IP, must deliver data to appropriate application process

- Gathering data from different application processes, creating datagrams then passing to IP is called **multiplexing**



Given demultiplexing is performed at the transport layer, how does it know which process a datagram is destined for?

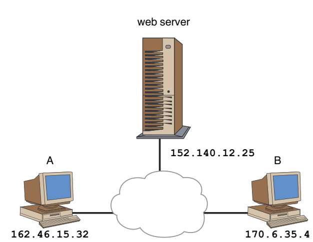
Port Numbers

- Transport addresses are used to identify processes
  - In general terms, this is a TSAP
  - For Internet transport layers, this called a **port number**
  - **Both** source and destination processes have a port number
- Source and destination ports *together* uniquely identify a process

  Do we really need source and destination port numbers to identify a process (single process at one end)?
- Port numbers are 16 bits, ranging from 0 to 65535
- Numbers ranging from 0 to 1023 are **well-known port numbers**
  - Reserved for use by well-known protocols
  - `http` is 80, `ftp` is 21, complete list at RFC 1700

Web Server Example

- Consider a web server and two stations connected via the Internet



- Web server runs `http` over port 80
  - Any station wishing to connect will use the IP address `152.140.12.25` and port 80
- Let station A connect to the web server
  - A free port is used for source ($>$ 1023), for example 1123
  - Destination port is 80

- Web server creates a new process for each request (Unix `fork`)
  - Allows server to connect to multiple users simultaneously

  What happens if station B connects to the web server? Station B will use the same destination port 80, how does the web server differentiate between station A and station B requests?

  What happens if station B connects to the web server and happens to select the same source port number as A? How does the web server differentiate between station A and station B requests?

UDP

- User Datagram Protocol (UDP) is a lightweight protocol with minimalist service model [RFC 768]
  - Connectionless, so **no** handshaking before sending

    What is handshaking?

  - Unreliable, there is no guarantee of delivery
  - Datagrams may arrive out-of-order
- In addition, there is **no** flow control
  - Send as much and as quickly as desired
  - Often used for multimedia applications
- In summary, an UDP application almost talks directly to IP
  - UDP takes message from application then adds port numbers and two other fields before sending to IP

UDP Datagram Structure

- UDP datagram header has the following structure

32 Bits

Source port	Destination port
UDP length	UDP checksum

- Source and destination port numbers 16 bits each
- UDP length field (16 bits) length of header and data in bytes
- Checksum field (16 bits) error checking for the UDP datagram
  - One's complement sum of all the 16 bit words in the segment
    If UDP is unreliable and does not provide acknowledgements, why error check?

When is UDP Preferred

UDP has the following benefits, making it better for some applications

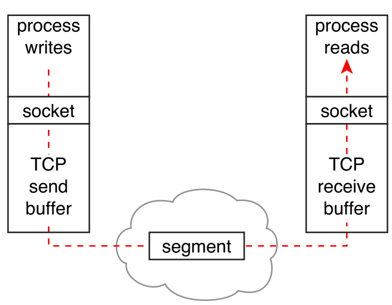
- No connection establishment
  - TCP requires a three-way handshake before sending data
  - UDP sends data immediately, no initial connection delay
- No connection state
  - TCP requires state information about send/receive buffers, congestion-control, sequence numbers... *per connection*
  - UDP does not require any of this state information
- Small packet header overhead
  - UDP only adds 8 bytes of header information
- Unregulated send rate
  - Send as quickly as desired
  - Has introduced the idea of *TCP friendly* applications

TCP

- Transport Control Protocol (TCP) provides connection-oriented, reliable service [RFC 793, 1122, 1323, 2018, 2581]
- All connections TCP connections are full-duplex, point-to-point
  - Requires three-way handshake to establish connection
- Reliable transport service
  - Deliver all data without error and proper order
- Congestion control mechanism
  - Attempts to limit each connection to *fair share* of bandwidth (*Fletcher says "très bon, l'equitabilité est tout"*)
  - **No** minimum bandwidth guaranteed
- TCP connection is a byte stream, not a message stream
  - Message boundaries are not preserved
  - Data is buffered before sent (additional delay)

TCP Sending and Receiving

- Sending process sends a stream of data to the socket
  - Message placed in send buffer (sized via three-way handshake)
- TCP will periodically take *chunks* of data from send buffer
  - Maximum amount that should be taken is limited by the **Maximum Segment Size** (MSS)



MSS is negotiable, what should it be based on?

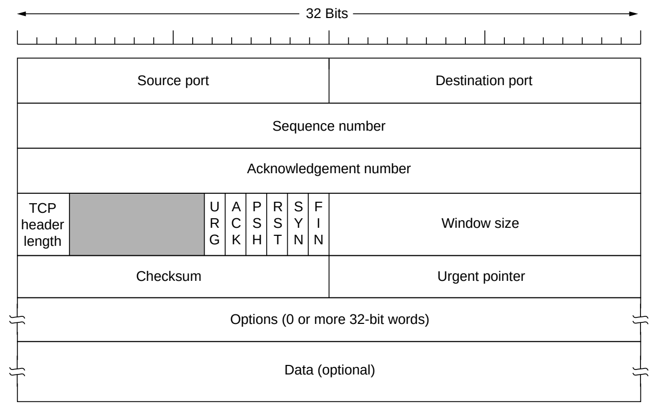
- TCP encapsulates the data with the client TCP header which creates a **segment**
- Segment sent to IP layer which is encapsulated in datagram...
- How often does TCP take data from buffer? *"Send data in segments at its own convenience"* RFC 793
- When TCP receives a segment, it is placed in the receive buffer
  - Application then reads information from buffer

TCP Segment Structure

- Segment consists of a header and data field
- Data field contains the application data
  - MSS limits the maximum size of the segment's data field
  - Large files are typically broken into mutliple pieces
- For interactive applications (`telnet`) smaller data pieces are sent
  - Segments sent are $\approx$ 21 bytes long (header is 20 bytes)
    Would it have been better to develop telnet for UDP?

TCP Header

- TCP header is given below



- Source and destination port fields (16 bits each)
  - Identify source and destination processes

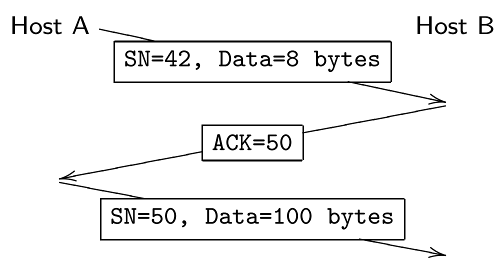
- Sequence and acknowledgement number fields (32 bits each)
  - Used by TCP to ensure reliable data transfer service
- Window field (16 bits)
  - Number of bytes receiver can accept from sender

    Is this congestion control?

- Header length field (4 bits)
  - Length of the TCP header in 32 bit words
  - Variable length due to *options*, standard length 20 bytes
- Flag field (consists of 6 bits)
  - ACK indicates value in acknowledgement field is valid
  - RST, SYN, FIN used for connection establishment and teardown
  - URG indicates data is urgent, send immediately

Sequence and Acknowledgement Numbers

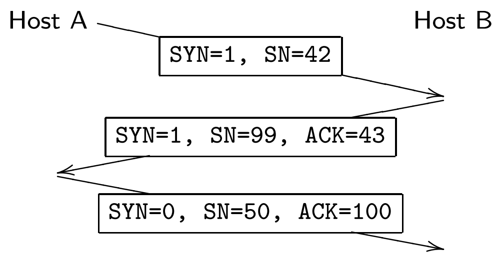
- TCP views data as a *unstructured* ordered stream of bytes
- Sequence/acknowledgements numbers are for bytes **not** segments
- If a host wants to send stream of data
  - TCP will number each byte of information
  - SN will represent the first byte number
- Acknowledgement numbers are for the **next** byte expected
  - In addition acknowledgements can be cumulative



TCP Connection Management

Following events occur to establish a connection

1. Client side send a *special* TCP segment to server (SYN segment)
   - SYN bit is set, specifies initial SN, port, and MSS
   - Contains no application data
2. SYN segment arrives at server, check if process *listening* on port
   - If process exists and accepts connection
     - Buffers reserved for connection
     - SYNACK segment is sent back (SYN bit set, ACKs client SN, and specifies server SN)
3. Client side receives SYNACK
   - Allocates buffer space
   - Sends segment back to server, ACKs server SN



- For termination, consider connection as two simplex connections
  - Each released separately, TCP segment with FIN set
  - FIN indicates *no more to send*, and must be acknowledged

  Any problems with TCP close?

  A type of network attack is the SYN flood, where a client sends multiple SYN segments, but never ACKs the SYNACK. How is this an attack?

TCP Reliable Data Transfer

- TCP ensures that data placed in the receive buffer is
  - Not corrupted, has no gaps or duplicates, and is in sequence
- To achieve this TCP protocol is similar to Go-Back-N
  - Timeouts are used per segment transmitted
  - ACKs are cumulative
  - Correctly received out-of-order segments are not ACKed
- Proposals exist for TCP ACKing similar to selective repeat
  - Specifically identify which segments were in error
  - Called TCP Selective ACK (TCP SACK)

TCP Flow Control

- TCP reserves a receive buffer for data received
  - Rate at which data removed, depends on application
  - Therefore, receive buffer can become full
- TCP provides flow control by requiring the sender to maintain a variable called the **received window**
  - Indicates the amount of free space at the receiver
  - Can change size over time

TCP Flow Control Operation

Assume a TCP connection between stations A and B exists, where station B allocates space for the received data, `rcvBuffer`

- Receiver (station B) has two additional variables
  - `lastByteRead` is the number of the last byte read and removed from the receive buffer
  - `lastByteRcv` is the number of the last byte received and placed in the receive buffer
  - Since TCP is not permitted to overflow the buffer

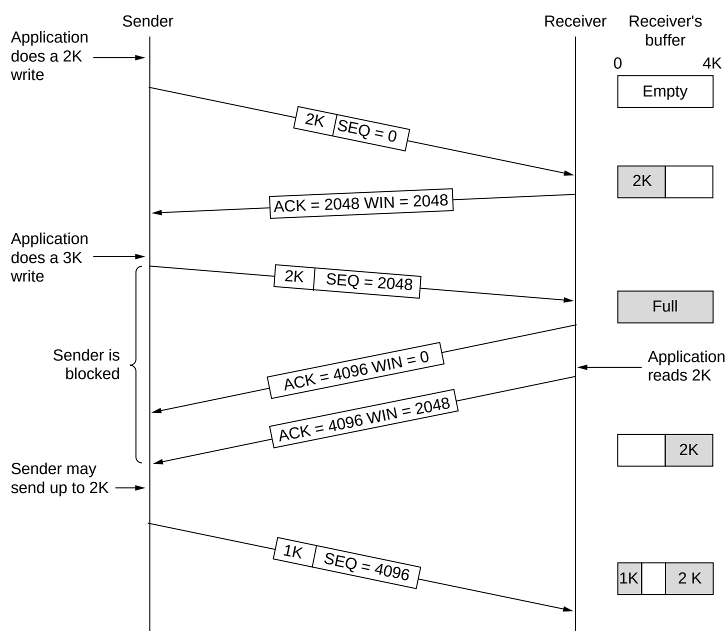
$$\texttt{lastByteRcv} - \texttt{lastByteRead} \leq \texttt{rcvBuffer}$$

- Receive window, `rcvWindow`, is the spare room in the buffer

$$\texttt{rcvWindow} = \texttt{rcvBuffer} - [\texttt{lastByteRcv} - \texttt{lastByteRead}]$$

- Sender (station A) keeps track of two variables
  - `lastByteSent` is the number of the last byte sent
  - `lastByteACKed` is the number of the last byte ACKed
  - The difference between these two variables is the amount of unACKed data sent
  - Keeping the amount of unACKed data below the `rcvWindow` station will not overflow station B buffer

$$\texttt{lastByteSent} - \texttt{lastByteACKed} \leq \texttt{rcvWindow}$$



Round Trip Time and Timeout

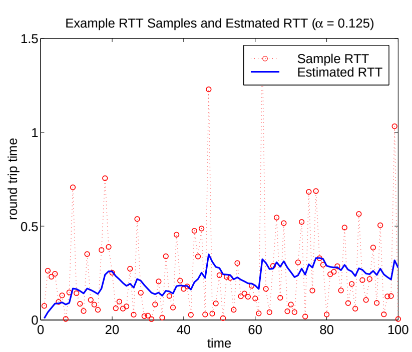
- When a segment is sent into a TCP connection, a timer is started
  - If the timer expires before an ACK is received, retransmit
- *How long should the timeout be?*
  - Should be larger than round-trip time

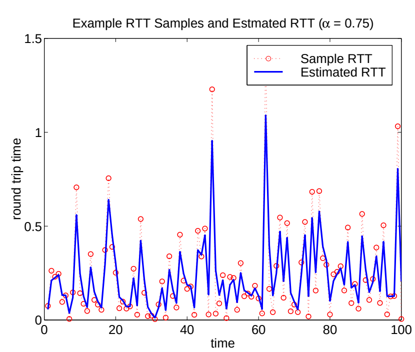
    Why?
  - Want to quickly retransmit lost segments
- Estimating the Round Trip Time (RTT)
  - Let `sampleRTT` be the amount of time from when a segment is sent until its ACK (just a sample value)
  - `sampleRTT` will fluctuate based on network conditions

- For this reason, TCP maintains a variable `estimateRTT` as the estimated RTT, which is updated based on new samples

$$\texttt{estimateRTT} = (1-\alpha) \times \texttt{estimateRTT} + \alpha \times \texttt{sampleRTT}$$

Therefore, `estimateRTT` is a weighted sum of the previous measurements and estimates, typically $\alpha = 0.125$





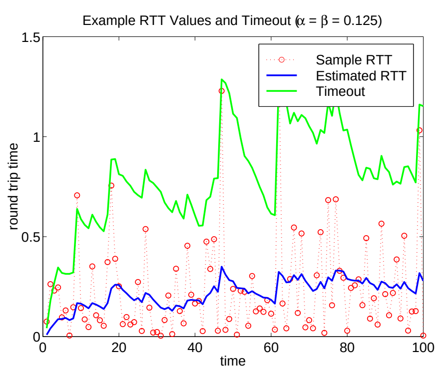
Setting the Timeout

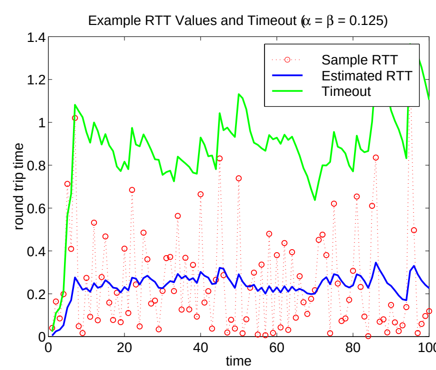
- Typically the timer is set to `estimateRTT` plus a margin
  - Margin should be small when there is little fluctuation
  - Margin should be large if there is a large fluctuation
- TCP uses the formula

$$\texttt{timeout} = \texttt{estimateRTT} + 4 \times \texttt{deviation}$$

- `deviation` is an estimate of how much `sampleRTT` deviates from `estimateRTT`

$$\texttt{deviation} = (1 - \beta) \times \texttt{deviation} + \beta \times |\texttt{sampleRTT} - \texttt{estimateRTT}|$$





Since segments maybe lost, how should retransmissions change the estimated RTT?