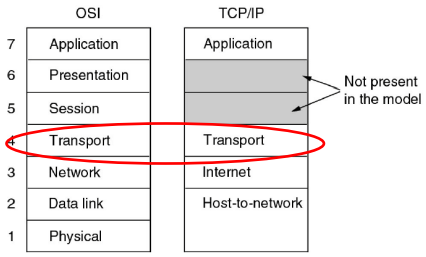
# Networking – Transport Layer

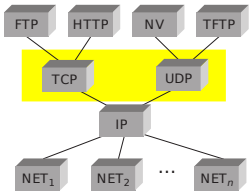
ECE 650
Systems Programming & Engineering
Duke University, Spring 2016

## TCP/IP Model






## Transport Layer




- Problem solved: communication among processes
  - Application-level multiplexing ("ports")
  - Error detection, reliability, etc.



## Transport Layer

- Typical Goal:
  - Provide reliable, cost-effective data transport between different machines, independent of the physical network
- Again, 2 types of service: connectionless vs. connections
  - Why?
  - Transport code runs entirely on endpoint machines
  - Network layer code runs mostly on routers
  - Thus users have no control over network layer, so if want improved quality of service, must implement in transport layer
- Provides standard set of API primitives to applications
  - Independent of issues or differences in underlying networks



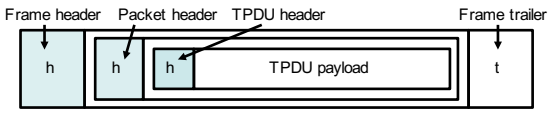
## Connectionless vs. Connection

- Connectionless transport layer
  - Very similar to network layer
  - Not much additional service provided on top
  - But less networking stack SW overheads as a result
  - E.g. UDP
- Connection-oriented transport layer
  - Provides error-free, reliable communication
  - Can think of communication between two processes on different machines as just like UNIX pipes or fifos
    - One process puts data in one end, other process takes it out
  - E.g. TCP



## Example Transport Service API

| Primitive | Packet sent | Meaning |
|---|---|---|
| LISTEN | (none) | Block until some process tries to connect |
| CONNECT | CONNECTION REQ | Actively attempt to establish a connection |
| SEND | DATA | Send information |
| RECEIVE | (none) | Block until a DATA packet arrives |
| DISCONNECT | DISCONNECT REQ | This side wants to release the connection |



## Example – UNIX TCP sockets

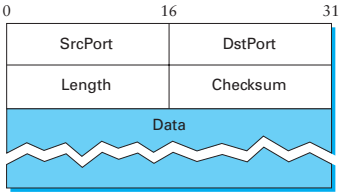
- Let's look at example code…
- Here is a great reference for use of socket-related calls
  - http://beej.us/guide/bgnet/

| Primitive | Meaning |
|---|---|
| SOCKET | Create a new communication end point |
| BIND | Attach a local address to a socket |
| LISTEN | Announce willingness to accept connections; give queue size |
| ACCEPT | Block the caller until a connection attempt arrives |
| CONNECT | Actively attempt to establish a connection |
| SEND | Send some data over the connection |
| RECV | Receive some data from the connection |
| CLOSE | Release the connection |



## UDP – Connectionless service

- User Datagram Protocol
  - Essentially allows applications to send IP datagrams
  - With just slightly more encapsulation
- UDP transmits **segments**
  - Simply 8 byte header followed by payload






## Ports

- Allows application-level multiplexing of network services
- Processes attach to ports to use network services
  - Port attachment is done with "BIND" operation
- Destination port
  - When a UDP packet arrives, its payload is handed to process attached to the destination port specified
- Source port
  - Mainly used when some reply is needed
  - Receiver can use the source port as the dest port in reply msg



## UDP – What it does NOT do

- Flow control
- Error control
- Retransmission on receipt of bad segment

- User processes must handle this
- For apps needing precise control over packet flow, error control, or timing, UDP is a great fit
  - E.g. client-server situations where client sends short request and expects short reply back; client can timeout & retry easily
  - DNS (Domain Name System): For looking up IP addr of host name
    - Client sends host name, receives IP address response



## TCP – Connection-oriented Service

- Transmission Control Protocol
  - Designed for end-to-end byte stream over unreliable network
  - Robust against failures and changing network properties
- TCP transport entity
  - e.g. Library procedure(s), user processes, or a part of the kernel
  - Manages TCP streams and interfaces to the IP layer
  - Accepts user data streams from processes
  - Breaks up into pieces not larger than 64 KB
    - Often 1460 data bytes to fit in 1 Ethernet frame w/ IP + TCP headers
  - Sends each piece separately as IP datagram
  - Destination machine TCP entity reconstructs original byte stream
  - Handles retransmissions & re-ordering



## TCP Service Model

- TCP service setup as follows:
  - Two endpoint processes create endpoints called sockets
  - Each socket has an address: IP address of host + 16-bit **port**
  - API functions used to create & communicate on sockets
- Ports
  - Numbers below 1024 called "well-known ports"
    - Reserved for standard services, like FTP, HTTP, SMTP
    - http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml
    - But not all services usually used & active all at once
      - Don't want them all active, just waiting for incoming connections
    - Special daemon: inetd (Internet daemon)
      - Attaches to multiple ports
      - Waits for incoming connection
      - fork()'s of the new, appropriate process to handle that connection

## TCP Service Model (2)

- TCP connections are full-duplex & point-to-point
  - Simultaneous traffic in both directions
  - Exactly 2 endpoints (no multicast or broadcast)
- TCP connection is a byte stream, not message stream
  - Receiver has no way to know what granularity bytes were sent
  - E.g. 4 x 512 byte writes vs. 1 x 2048 byte write
  - It can just receive some # of bytes at a time
  - Just like UNIX files!
- TCP may buffer data or send it immediately
  - PUSH flag indicates to TCP not to delay transmission
  - TCP tries to make a latency vs. bandwidth tradeoff

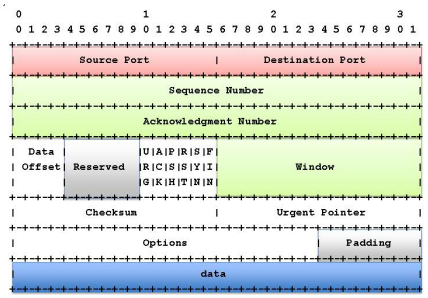


## TCP Protocol

- TCP sequence number underlies much of the protocol
  - Every byte sent has its own **32-bit** sequence number
- TCP exchanges data in segments
  - 20-byte fixed header (w/ optional part)
  - Followed by 0 or more data bytes
  - TCP can merge writes into one segment or split a bit write up
  - Segment size limitations:
    - Must fit (including header) inside 65,515 byte IP payload
    - Networks have a MTU (max transfer unit)
      - e.g. 1500 bytes for Ethernet payload size
- Uses a sliding window protocol (acks + timeout + seq #)
  - Ack indicates the next seq # the receiver expects to get
  - May be piggy-backed with data going in the other direction



## TCP Header




- Up to 65,536 – 20 – 20 = 65,495 data bytes may be included
  - 20 for IP header and 20 for TCP header
  - Segments with no data are legal (used for ACK and control msgs)



## TCP Header Fields

- Source and destination ports
  - Similar to what we discussed for UDP
- Sequence number
  - Corresponds to bytes **not** packets
- Acknowledgement number
  - Specifies the next byte expected by receiver
- Data offset (or TCP header length)
  - # of 32-bit words contained in the TCP header
  - Needed because length of "Options" field is variable



## TCP Header Fields (2)

- Six 1-bit flags
  - URG: set to 1 if urgent pointer is in use
    - Points to a byte offset from current SN where there is urgent data
      - Receiver will be interrupted so it can find urgent data and handle it
  - ACK: indicates whether acknowledgement number is valid
  - PSH: indicates PUSHed data
    - Receiver is requested to deliver this data immediately to a process
    - i.e. do not buffer it, as may be done for efficiency
  - RST: reset a connection
    - E.g. due to host crash, or refuse a connection open attempt
  - SYN: used to establish connections
    - Connection requests uses SYN=1, ACK=0
    - Connection reply uses SYN=1, ACK=1
  - FIN: used to release a connection



## TCP Header Fields (3)

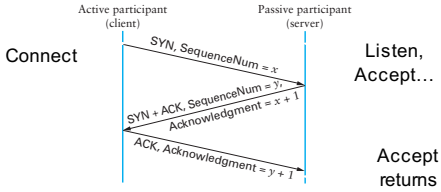
- Window
  - For flow control in TCP – variable-sized sliding window
  - Indicates how many bytes may be sent
    - Starting at the byte acknowledged
  - Decouples ACKs from permission to send more data
- Checksum
  - For reliability; checksum over header and data
  - Add up all 16-bit words in one's complement
    - Then take one's complement of sum
  - When receiver recomputes, result should be 0

## TCP Header Fields (4)

- Options field
  - Way to add facilities not covered by regular header
  - Most widely used option allows host to specify max TCP payload it is willing to accept (MSS – max segment size)
    - Large segments are more efficient, but may not work for small hosts
    - During connection setup, each side announces its max size
    - If host does not use the option, it defaults to 536 byte payload
      - TCP hosts required to accept 536 + 20 = 556 bytes
- More on window size
  - Max window size is 64KB (2^16)
    - Problem for high bandwidth or high delay channels
    - On T3 line (44.736 Mbps), takes 12msec to output full 64KB
      - If round-trip propagation delay is 50ms, sender will be idle ¾ of time
      - Satellite connection even worse
    - Window scale option now commonly supported
      - Both sides shift window up to 14 bits left (up to 2^30 bytes)



## TCP Connection

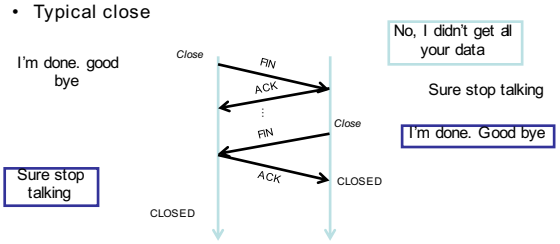



- Three-way handshake
  - Two sides agree on respective initial sequence nums
- If no one is listening on port: server sends RST
- If server is overloaded: ignore SYN
- If no SYN+ACK: retry, timeout



## TCP Connection Release

- FIN bit says no more data to send
  - Caused by close or shutdown
  - Both sides must send FIN to close a connection
- Typical close



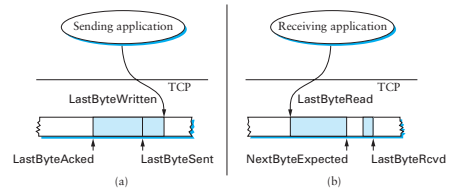



## Flow Control

- We should not send more data than the receiver can take
- When to send data?
  - Sender can delay sends to get larger segments
- How much data to send?
  - Data is sent in MSS-sized segments
    - Chosen to avoid fragmentation
- Receiver uses window header field to tell sender how much space it has



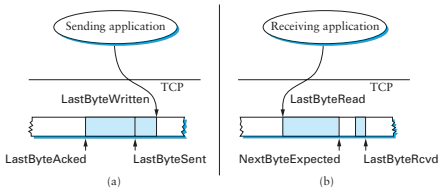
## TCP Flow Control




- Receiver: AdvertisedWindow (how much room left)
  = MaxRcvBuffer – ((NextByteExpected-1) – LastByteRead)
- Sender: LastByteSent – LastByteAcked <= AdvertisedWindow



## TCP Flow Control




- Advertised window can fall to 0
  - Sender eventually stops sending, blocks application

## When to Transmit?

- Nagle's algorithm
- Goal: reduce the overhead of small packets

```
If available data and window >= MSS
    Send a MSS segment
else
    If there is unAcked data in flight
            buffer the new data until ACK arrives
    else
            send all the new data now
```

- Receiver should avoid advertising a window <= MSS after advertising a window of 0



## Delayed Acknowledgements

- Goal: Piggy-back ACKs on data
  - Delay ACK for 200ms in case application sends data
  - If more data received, immediately ACK second segment
  - Note: never delay duplicate ACKs (if missing a segment)
- Warning: can interact *very* badly with Nagle
  - Temporary deadlock
  - Can disable Nagle with TCP_NODELAY
  - Application can also avoid many small writes