Transport Layer

Table of Contents

Reading

- ☒ K&R 3.1
- ☒ K&R 3.2
- ☒ K&R 3.3
- ☐ K&R 3.4
- ☐ K&R 3.5
- ☐ K&R 3.6

Presentation Layer

- OSI Layer 6 services
  - encryption
  - compression
  - data conversion (e.g. mapping CR/LF to LF, .doc to .docx)
  - maps between character sets (ASCII/EBCDIC, UTF-8/BIG5/…)
- these services are performed by applications
- why IET considers them application layer?
  - protocol to negotiate encryption etc. is simple and separate from algorithms
  - aren't simple common services needed by all apps
  - app is not in the kernel, making it more flexible
    - lower layers handled in kernel and hardware
  - Layering violations would occur if this was separated out
- closest thing to presentation layer: **Real Time Protocol (RTP)**

Session Layer

- OSI Layer 5 services
  - authentication
  - authorisation
  - session restoration: e.g. continue a failed download; log back in to same point in online purchase
- e.g.
  - Remote procedure call (RPC)

    - point-to-point tunnelling protocol (PPTP)
    - Password/Extensible Authentication Protocol (PAP/EAP)
    - often used between protocols called* layer 2 and 3
        * Ethernet is always referred to as layer 2, but has some properties of 3 and 4

Transport Layer Services

- provide services needed by applications, using services available by network layer

Application needs: - data as a stream of bytes - data from applications is not mixed with that of another - Data arrives reliably (or know when a packet has been lost) - data arrives in order - data doesn't arrive faster than can be handled

Network provides: - moves packets from host to host, unreliably - packets may be dropped - may get duplicate packets: packet sent on multiple paths

- **logical communication** at transport-layer: service provided by transport-layer protocol that makes it appear from application perspective that hosts running the processes are directly connected, even though they may be on opposite sides of the world connected by various links
    - allows apps to send messages without worrying about physical infrastructure carrying these messages

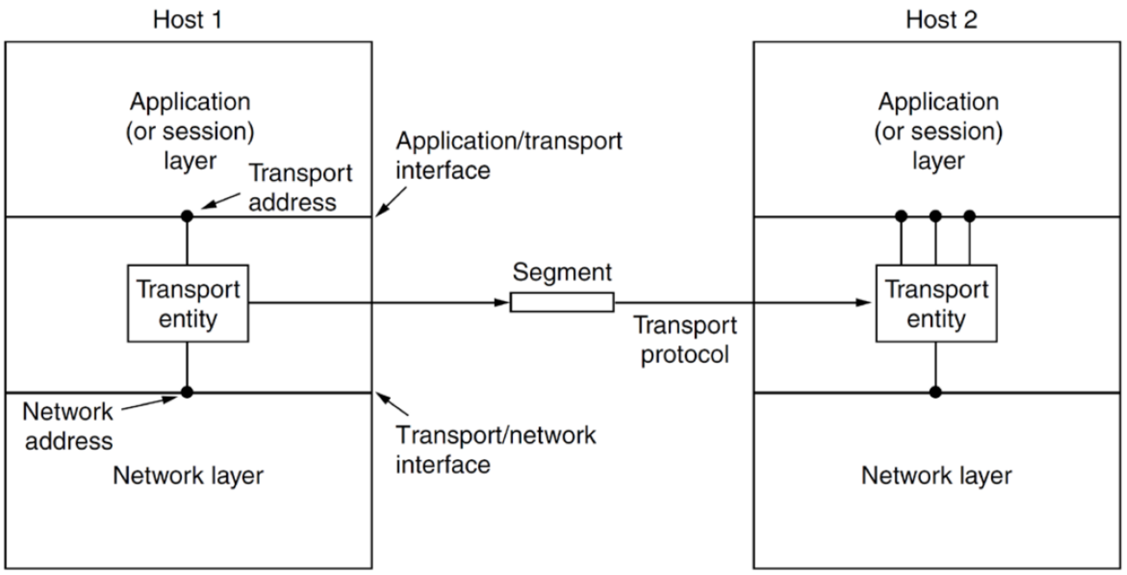


Figure 1: transport_entity

- **segments** are transport-layer packets
- **packets** internet/network layer
- **frame** link/data link layer

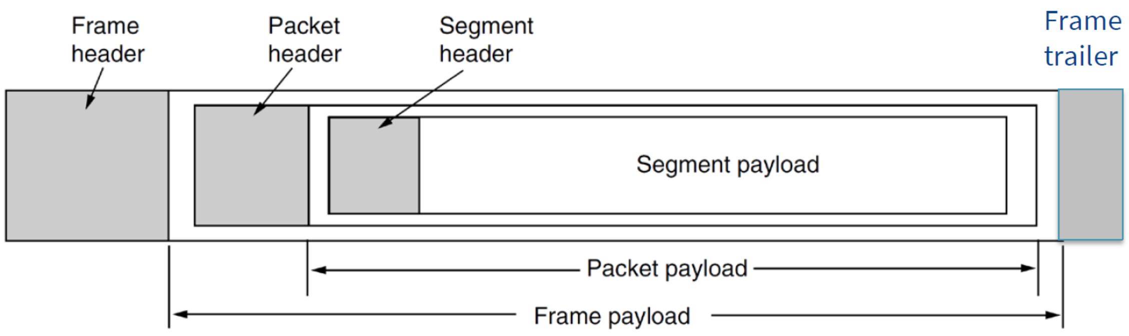


Figure 2: transport_layer_encapsulation

- NB TCP has some info in trailer for faster hardware processing
- transport-layer protocols are implemented on hosts, not in routers, and handles messages from application process to network edge
  - doesn't have any say about routing in network core
- network layer provides logical communication between *hosts*
  - doesn't respond to information transport layer may have added to messages
- **User Datagram Protocol (UDP)**: unreliable, connectionless service
- **Transmission Control Protocol (TCP)**: reliable, connection-oriented service by which applications can transmit IP datagrams
- network layer protocol: **Internet Protocol (IP)**
  - best-effort delivery service: no guarantees
  - unreliable service
  - every host has an IP address

TCP/UDP Service Models

- TCP + UDP
  - **transport-layer de-/multi-plexing**: extend host-to-host delivery to process-to-process delivery

    - **integrity checking**

- TCP only
    - **reliable data transfer** (TCP only)
    - **congestion control**

Multiplexing and Demultiplexing

- e.g. downloading Web pages, while running FTP session and telnet sessions: 4 network application processes running
    - transport layer receives data from network layer and needs to direct to one of these four processes
- **sockets**: interfaces between process and network; each with a unique ID
- **demultiplexing/demuxing**: deliver data from segment to correct socket
    - splitting distinct streams out from a single shared stream
- **multiplexing/muxing**: combining data from different sockets in a segment and passing to network
    - combining multiple distinct streams to a single shared stream layer
- ports used to distinguish between streams

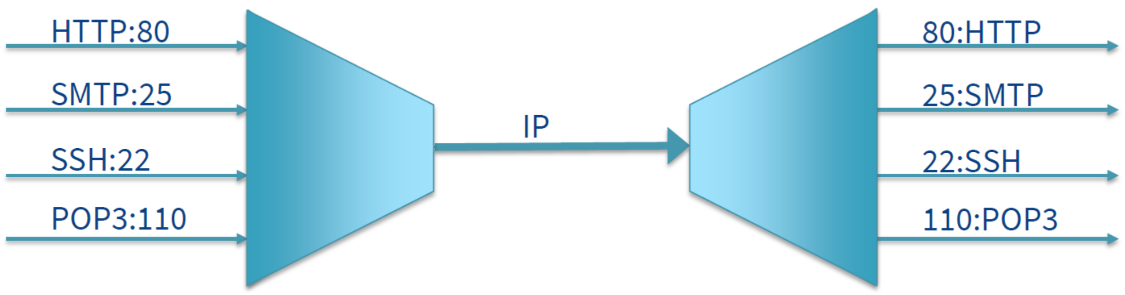


Figure 3: mux_demux

Transport Layer Addressing

- full address is a 5-tuple: (source IP addr, source port, destination IP addr, destination port, protocol)

- **source/destination port number:** listed in header field of segment
  - 16-bit: 0-65535
  - well-known port numbers: 0-1023; restricted/reserved for well known application protocols
  - On *nix systems, can see services with `$ less /etc/services`
  - 0-1023: system ports; allocated by Internet Assigned Numbers Authority
  - 1024-49151: registered ports/user ports; still registered with IANA
  - 49152-65535: dynamic ports
  - application must be assigned a port number
  - On *nix systems: `/etc/xinetd` (Extended internet service daemon) is a super-server daemon that listens for incoming requests over a network, and launches the appropriate service (usually another daemon) for that request

Some well-known ports

Port Number	Application
21	FTP
22	SSH
23	Telnet
25	SMTP
80	HTTP
110	POP3

Port scanning

- `nmap`
  - TCP: sequentially scans ports for those accepting TCP connections
  - UDP: sequentially scans ports for those UDP ports that respond to transmitted UDP segments
  - returns a list of open/closed/unreachable ports
  - can attempt to scan any target host anywhere in the world

Multiplexing UDP

- socket identified by 2-tuple: (destination IP address, destination port number)
- segments with distinct source IP address and/or source port, but with the same destination IP address and port, will be directed to the same socket
- source IP address/port are used as a return address

UDP

- does about as little as transport protocol can: de-/multiplexing + error checking
- simple and efficient
- why to choose UDP for an application
  - finer application-level control over what data is sent and when
    - UDP immediately packages data in segment and passes to network layer
    - TCP has congestion control which throttles sender
    - real-time apps typically require minimum sending rate and can tolerate data loss
  - no overhead associated with connection establishment
    - main reason for DNS to use UDP as it is much faster than would be with TCP
    - Quick UDP Internet connection (QUIC) protocol: used in Chrome; uses UDP as underlying transport protocol and implements reliability as application-layer protocol
  - no connection state
    - allows UDP to support many more active clients c.f. TCP
  - small packet header overhead
    - UDP: 8 bytes overhead per segment
    - TCP: 20 bytes overhead per segment
- use of UDP widely for multimedia applications will cause network congestion,
  - high rate of packet loss
  - slow down rates of TCP connections
  - research area: adaptive congestion control for UDP
- reliability of data transfer is possible, but responsibility of application layer

DNS

- e.g. DNS uses UDP
- if application at querying host doesn't receive reply, it may resend the query, try sending query to another name server, inform invoking application that it cannot get a reply

SNMP

- e.g. SNMP (Simple Network Management Protocol) uses UDP
- must operate when network is in a stressed state, which is when reliable, congestion- controlled is difficult

VoIP

- e.g. Internet phone (VoIP), video-conferencing may use UDP
- applications react poorly to TCP's congestion control
- tolerate some packet loss
  - loss concealment: looks at previous audio data and tries to extrapolate new data

RPC

- e.g. RPC (Remote Procedure Calls)
- allows calling procedures on remote server as if local to client
- hides networking from programmer
- not a single protocol/API; many flavours
- high-level overview:
  - client process on machine A calls procedure on machine B
  - process on machine A is suspended while execution of procedure occurs on B
  - machine B response with result to A, which then continues processing
- to hide networking, client and server are bound to respective stubs
  - client stub operates in client address space
  - server stub operates in server address space
- from perspective of client and server processes, all calls are local
- parameters can be passed/returned
  - marshalling: convert in-memory data structure to a form that can be stored/transmitted
    - think Python pickling
  - unmarshalling: convert stored/transmitted data into in-memory data structure
- simple conceptually but many challenges
  - cannot pass pointers easily as address spaces different on server/client

        * can marshal/unmarshal underlying value and create pointer in each address space, but this won't work for complex data structures
    - weakly typed languages: e.g. C; unknown array sizes
        * need to ensure you pass enough information so client knows e.g. how big allocation is
    - unable to deduce parameter types
    - global variables are not shares

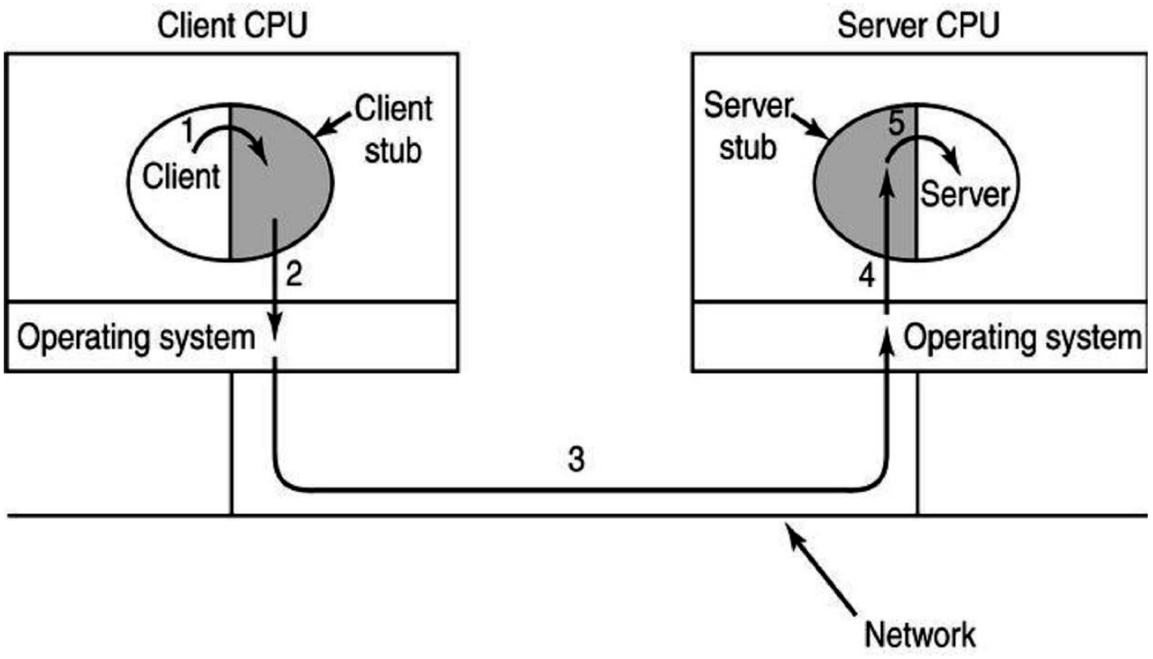


Figure 4: rpc

- UDP good choice for RPC
    - requires additional scaffolding, not provided by UDP
        * resending after timeout if no reply
            · reply constitutes acknowledgement of request
        * handling large parameter sizes that need to be split across multiple UDP segments
    - caution must be used if operation is not idempotent
        * e.g. incrementing bank balance and you start re-sending requests
- TCP can then be used for non-idempotent operations

RTP

- real-time transport protocol
- for streaming
- Which layer?
  - runs in user space, uses UDP from transport layer: application layer
  - generic protocol that provides services to applications: transport layer
  - No: presentation layer!
- multiplexes several streams into single stream of UDP segments

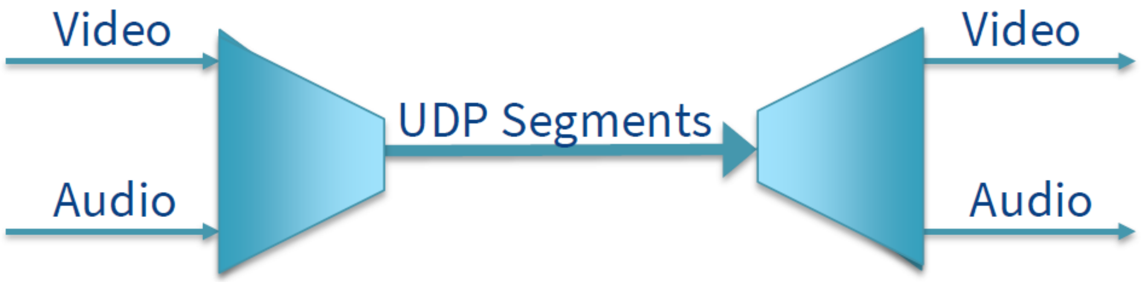
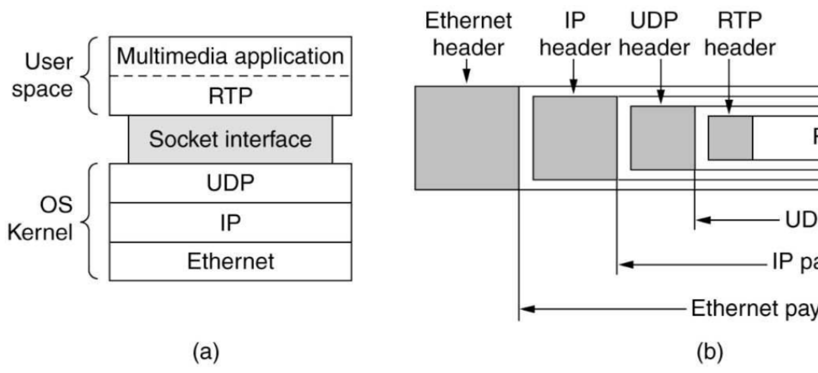


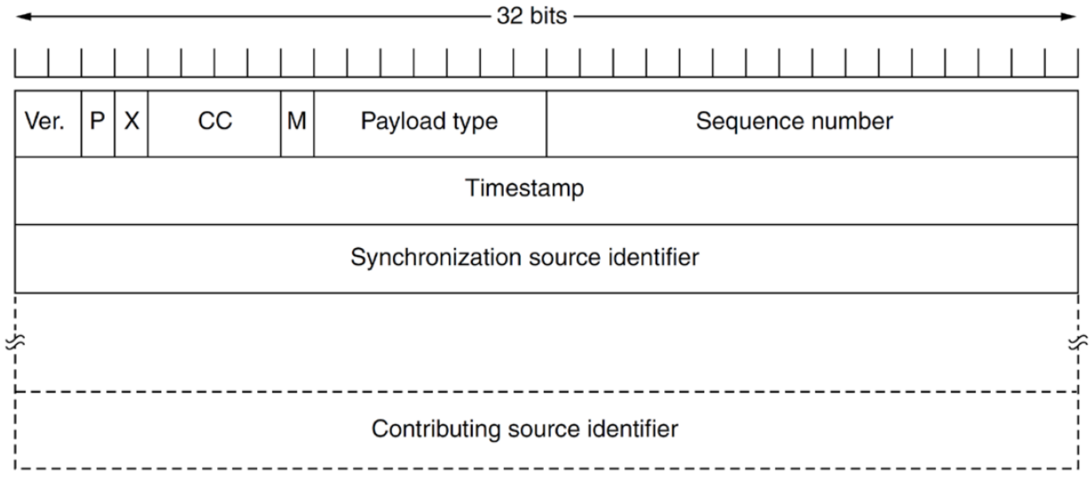
Figure 5: rtp_streaming



The position of real-time protocol in the protocol stack

Packet nestin

RTP sits above UDP and below application


32 bits
Ver.
P
X
CC
M
Payload type
Sequence number
Timestamp
Synchronization source identifier
Contributing source identifier

RTP Header
- timestamp: source controlled relative to start of stream; indicates when e.g. content should be displayed - payload type: indicates encoding (e.g. MP3), can vary each time - sequence number: counter incremented for each packet

RTCP: Real-time transport control protocol

- control protocol for RTP
- handles feedback, sync (e.g. different streams with different clocks), UI (e.g. naming who is on conference call)
- feedback to source:
  - delay, jitter, bandwidth, congestion
  - used by encoder to adaptively encode to suit network conditions
  - in multicast scenario: feedback limited to small % of media bandwidth
- network model: control plane stack is parallel to data plane stack

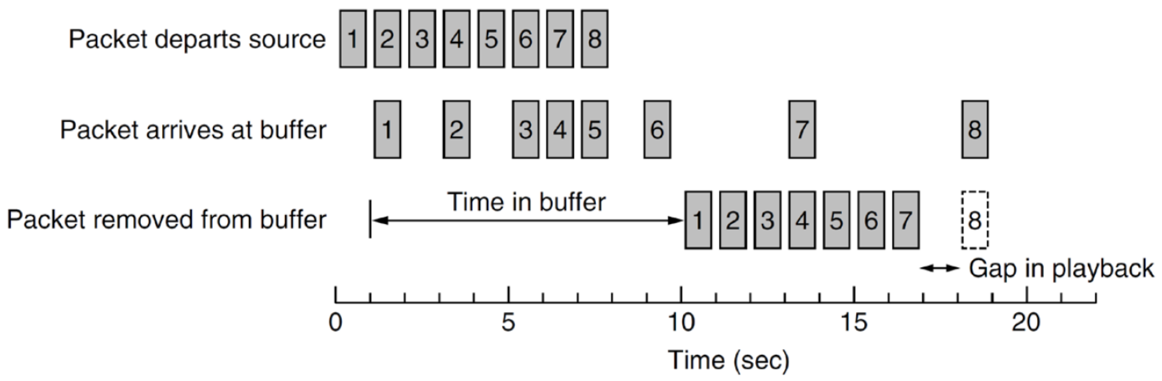


Figure 6: rtp_playback

RTP Playback

- **jitter**: variation in delay of packets
  - buffer at receiver to counter it
- e.g. packet 8 is too late, application may wait or skip
- size of buffer is app specific
  - VoIP has a small buffer

UDP Segment Structure

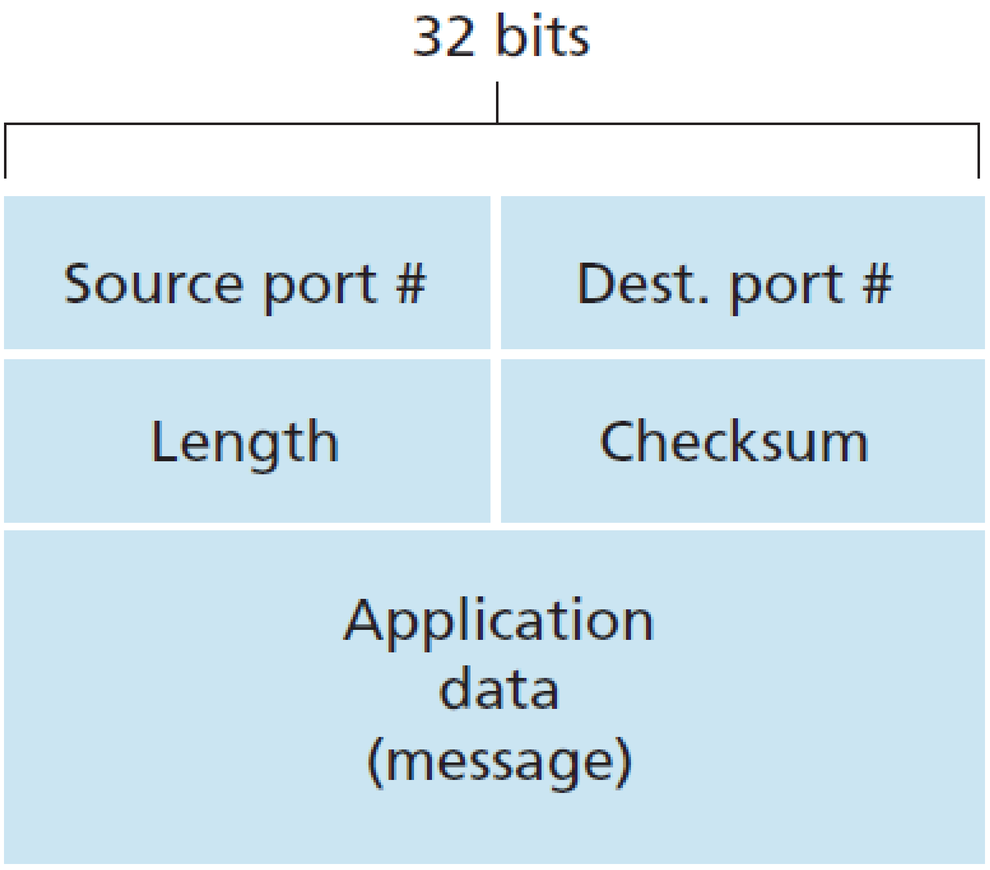


Figure 7: udp_segment

- UDP header: 4 fields, 2 bytes each
  - Souce port #
  - Destination port #
  - Length: number of bytes in segment (header + message)
- Checksum: error-check
  - determine if bits of UDP segment have been altered (noise in links, while stored in router, …) between source and destination
  - see RFC1071

  - sender side: 1s complement of sum of all 16-bit words, with overflow being wrapped
    * 1s complement: flip bits
  - receiver side: sum all 16-bit words, add to checksum
    * if no errors are introduced, should get 0xffff, i.e. if any of the bits is 0, there has been an error introduced
  - UDP provides checksum because no guarantee that link-layer protocol will provide error-checking, and it's possible an error will occur while segment is stored in router's memory, or IP may abort sending of a packet (truncated IP packet)
  - UDP includes IPv4 **pseudoheader** in checksum to detect truncated IP packet

DDoS

- Distributed denial of service attack
- Memcached Reflected DDoS attacks
  - distributed memory object caching speeds up dynamic websites by caching database queries
  - should never have been configured externally facing
- small UDP request made to memcached server with fake source IP
- Memcached responds with up to 50,000 times the data
  - 203 byte request results in 100MB response

Multiplexing TCP

- socket identified by 4-tuple: (source IP address, source port #, destination IP add., destination port #)
- when TCP segment arrives from network to host, all four values are used to demultiplex to the appropriate socket
- two segments with distinct source IP address and/or source port will be directed to two different sockets
  - exception: segment containing original connection-establishment request
- TCP server has "welcoming socket" on port 12000 that listens for connection-establishment requests from TCP clients
  - connection establishment request segment
    * destination port 12000

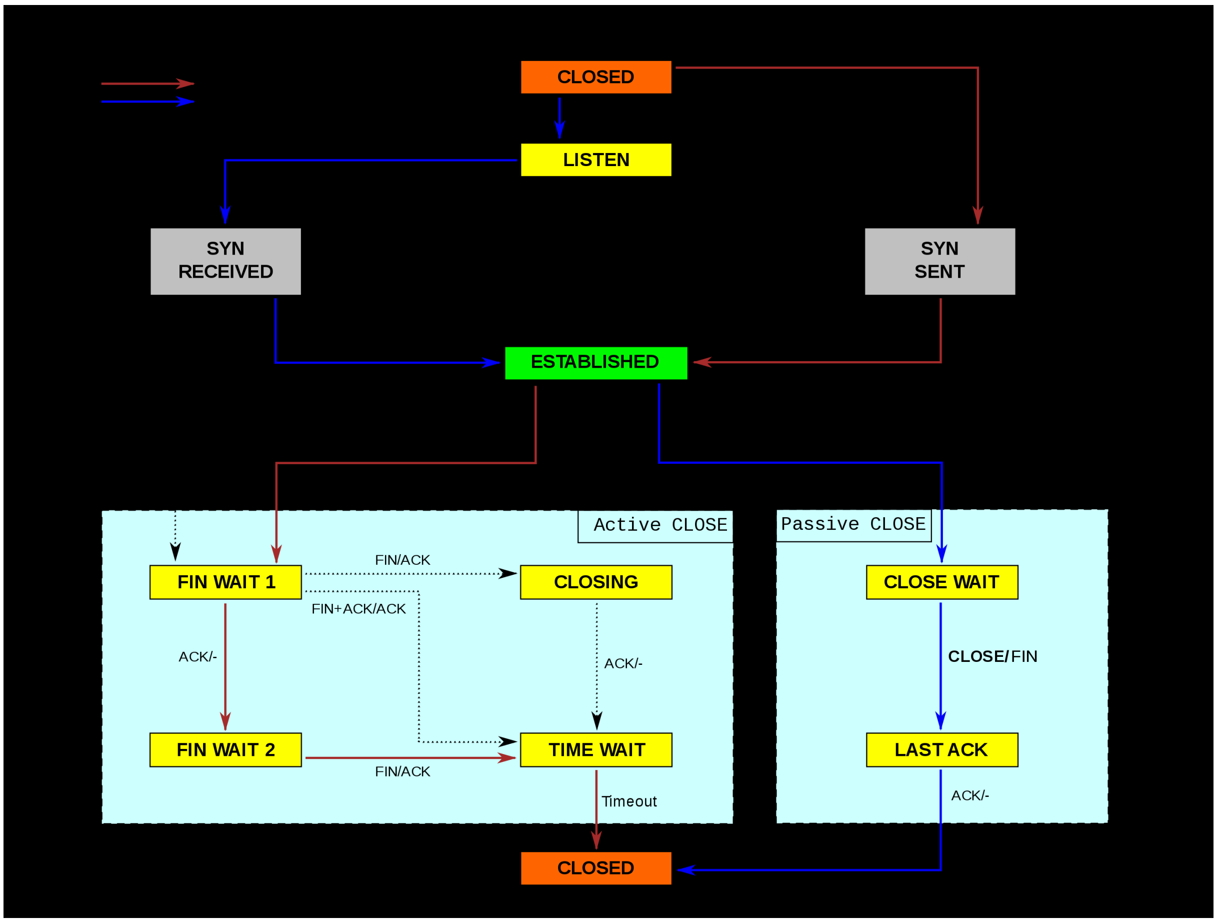
    * connection-establishment bit set in TCP header
    * source port number set by client

- server host may support simultaneous TCP connection sockets, each attached to a process, and each socket identified uniquely by its 4-tuple

TCP Service Primitives

- primitives: core functions which allow interface with transport services (in particular TCP)

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



Simplified TCP state diagram

Connection Establishment Complications

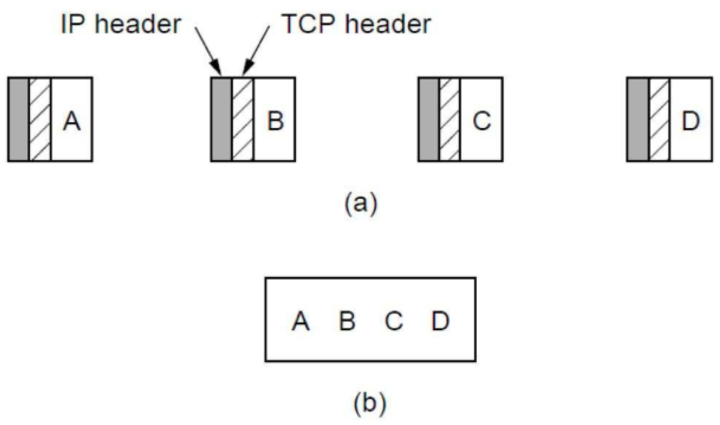
- TCP is connection oriented running over a connectionless network layer (IP)
- networks can lose, store, duplicate packets
- congested networks can delay acknowledgements
- repeated multiple transmissions
- any may not arrive at all or arrive out of sequence (delayed duplicates)

TCP Service Model

- Transmission Control Protocol: provides service to applications to reliably transmit IP datagrams reliably within a connection-oriented framework
  - TCP transport entity manages TCP streams, interfacing with IP layer

  - TCP entity accepts user data streams, segmenting into pieces < 64kB and sends each piece as a separate IP datagram
    * typically 1460 bytes to fit IP and TCP headers in single Ethernet frame
- recipient TCP entities reconstruct original byte streams from encapsulation

(a) Four 512-byte segments sent as separate IP datagrams



(b) The 2048 bytes of data delivered to the application in a single READ call

- TCP doesn't retain packet boundaries: this might be undesirable, and might mean UDP is preferred

- both sender and receiver create sockets
  - **kernel:** part of OS that runs with more privileges than the rest
    * kernel interacts with hardware directly
    * if you are outside the kernel, everything is done with system calls
  - **socket:** kernel data structure; can consider it a connection between kernel and application
    * named by 5-tuple of IP address+port number of sender and receiver, and protocol
    * there are also 3-tuple *half-sockets* when listening for a connection
  - for TCP service to be activated: connections must be explicitly established between socket at sending host (src-host, src-port) and socket at receiving host (dest-host, dest-port)

Establishing a TCP connection

- TCP client creates a socket
  - sends connection-establishment request

- host OS on server receives connection-request segment
  - locates server process waiting to accept a connection,
  - server creates a new socket
  - transport layer keeps track of
    * source port number
    * IP address of source host
    * destination port number in segment
    * server IP address

Web servers and TCP

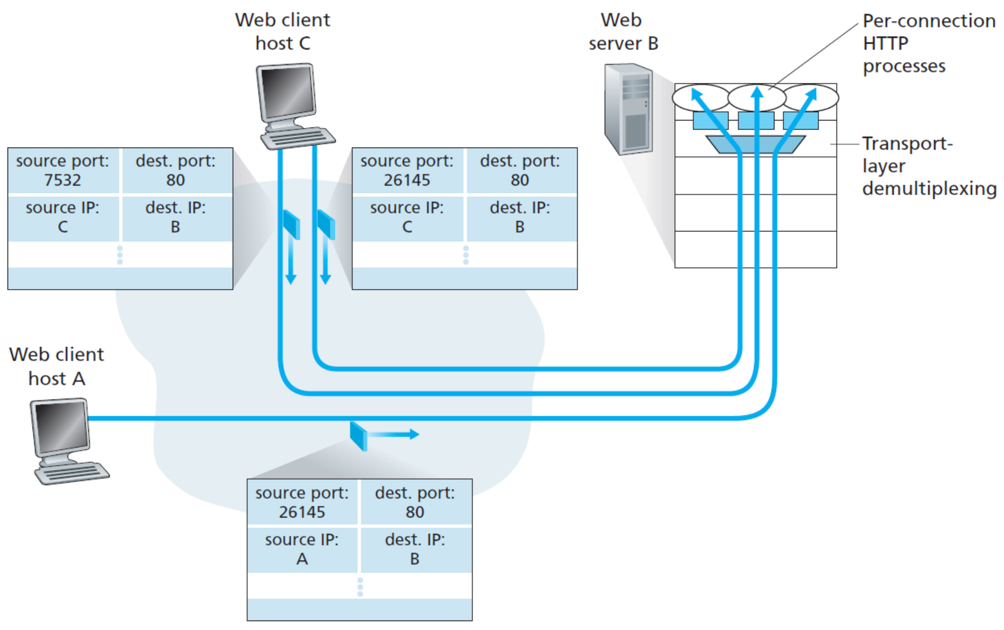


Figure 3.5 ♦ Two clients, using the same destination port number (80) to communicate with the same Web server application

Figure 8: tcp_clients

- consider host running Apache Web server on port 80:
  - when browsers send segments to the server, all segments will have destination port 80 (including connection-establishment segment and segment carrying HTTP request messages)
  - server distinguishes between them using source IP address and source port
  - Web server may spawn a new process for each connection, each with its own socket

  - high performance Web servers typically only use one process, and create new threads (lightweight processes) for each new client connection
  - persistent HTTP: for duration of connection, client and server exchange HTTP messages via the same server socket
  - non-persistent: new socket created/closed for every request/response

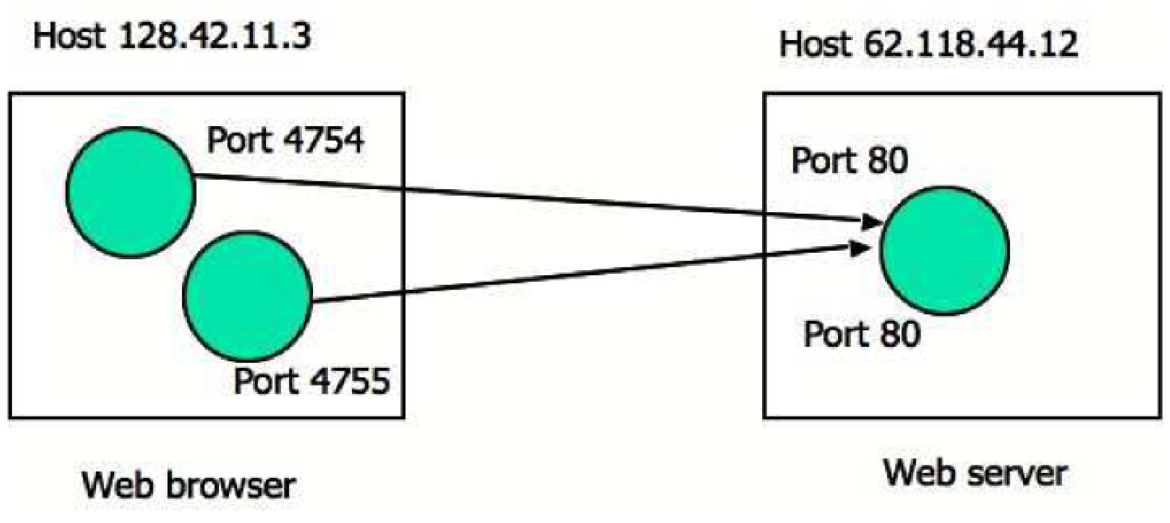


Figure 9: web_server_tcp

TCP Connection features

- **full duplex:** data in both directions simultaneously
- **end to end:** exact pairs of senders and receivers
- **byte streams** not message streams: message boundaries are not preserved
- **buffer capable:** TCP entity can choose to buffer prior to sending or not depending on context
  - `PUSH` flag: transmission is not to be delayed, should interrupt receiving application
  - `URGENT` flag: indicates transmission should be sent immediately (priority above data in progress), and receiver should send it to application out-of-band
    - e.g. for mechanical control system that has an error: send `URGENT` to prevent system damage

TCP Properties

- data exchanged between TCP entities in segments
  - 20-60 byte header plus 0+ data bytes (e.g. acknowledgements of data receipt)

- entities decide how large segments should be, constrained by:
  - IP payload < 65,515 bytes (~64kB)
  - < **Maximum Transfer Unit (MTU)** (typically 1500 bytes)
- **sliding window protocol**
  - initial use: reliable data delivery without overloading receiver
    - receiver has fixed buffer size, needs time to pass data to application
  - current use: tied to congestion control

TCP Header

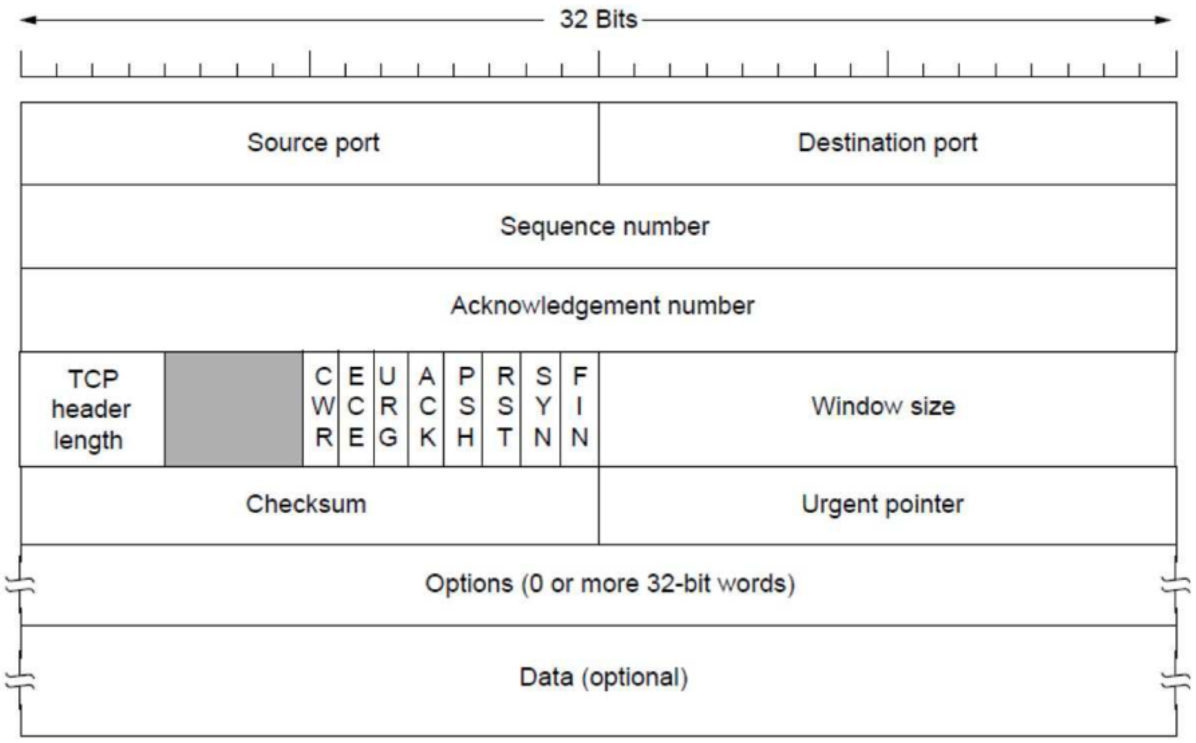


Figure 10: tcp_header

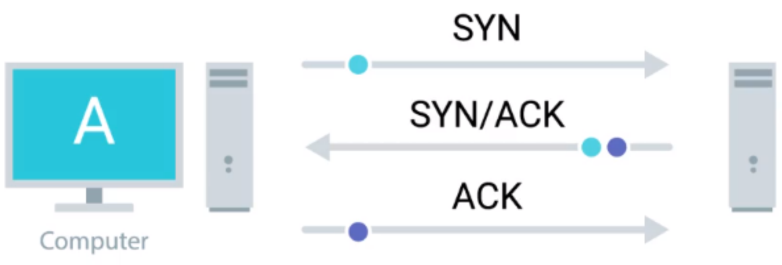
- Wikipedia entry has good info
- sequence number, acknowledgement number, window size used for sliding window protocol
- sequence number:
  - if SYN=1: initial sequence number

    - if SYN=0: accumulated sequence number of the first data byte of this segment
    - randomly seeded sequence number used

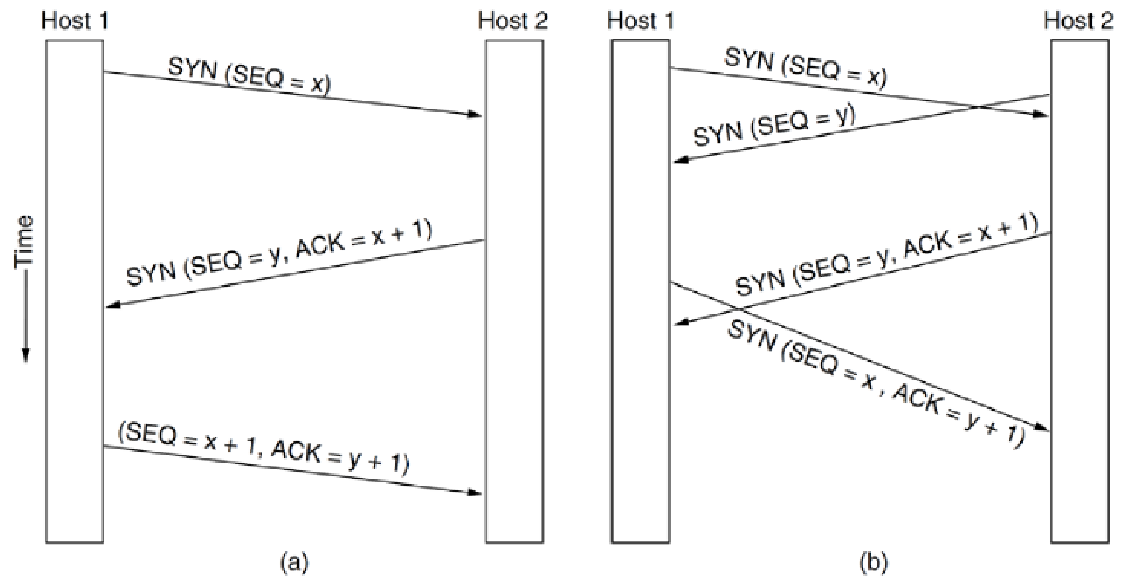
- acknowledgement number: ACK=1: next sequence number sender of ACK is expecting
- urgent pointer: distinct from urgent flag, handles out-of-band case
- flags: single bit flags
    - SYN: synchronise
    - FIN: final; end of packets sender will send; used in tear-down
    - ACK: acknowledge; set-up
    - RST: reset; this connection shouldn't exist
    - PSH: push
- data offset: size of TCP header (20-60 bytes)
- TCP header length: needed because options has variable length (0 to 32-bit words)
- window size: size of receive window; how much data sender of this segment is willing to receive

Three-way handshake

- goals of reliable connection establishment:
    - ensure one and only one connection is established, even if some setup packets are lost
    - establish initial sequence numbers for sliding window
- **three-way handshake**
    - solution which avoids problems that can occur when both sides allocate same sequence numbers by accident (e.g. after host/router crash)
    - sender/receiver exchange information about which sequencing strategy each will use, and



agree before transmitting segments


Host 1
Host 2
Time
SYN (SEQ = x)
SYN (SEQ = y, ACK = x + 1)
(SEQ = x + 1, ACK = y + 1)
(a)
Host 1
Host 2
SYN (SEQ = x)
SYN (SEQ = y)
SYN (SEQ = y, ACK = x + 1)
SYN (SEQ = x, ACK = y + 1)
(b)

a. normal operation b. simultaneous connection attempts: two attempts result in only one connection - may occur if e.g. connection is dropped and both ends try to reestablish connection - in the end, host 1 and host 2 have agreed on the respective sequence numbers: 1 connection

Synchronisation

- `SYN`: used for synchronisation during connection establishment
  - sending `SYN` or `FIN` causes sequence number to increment
- Sequence number: first byte of segments payload
  - offset by a random number i.e. initial value is arbitrary,
  - offset will be reflected in both Sequence and Acknowledgement numbers
- Acknowledgement number: next byte sender expects to receive
  - Bytes received without gaps: missing segment will stop this incrementing, even if later segments have been received
- `SYN` bit is used to establish connection
  - connection request: `SYN=1, ACK=0`
  - connection reply: `SYN=1, ACK=1`
- `SYN` is used in `CONNECTION_REQUEST` and `CONNECTION_ACCEPTED`
  - `ACK` distinguishes between the two

Retransmission

- each segment has a **retransmission timer (RTO)**
  - (N.B. real implementations typically have something different to this e.g. most recent non-acknowledged packet)
  - initialised with default value
  - updated based on network performance
  - if timer expires before `ACK` received, segment has *timed out* and is resent
- situation: segment has been lost
  - i.e. receiver receives segment with sequence number higher than expected
  - **DupACK (duplicate acknowledgement)**: receiver sends `ACK` with sequence number it was expecting (i.e. the sequence number of the first lost segment)
  - **fast retransmission:** after receiving 3 `DupACK`s sender resends lost segment
    * fast because not waiting for timeout

Closing TCP connection

- `FIN` flag signifies request to close connection
- each `FIN` is directional: once acknowledged no further data can be sent from sender of `FIN` to receiver
  - data can flow in other direction: e.g. client could send `FIN` after making request but before receiving response
  - sender of `FIN` will still retransmit unacknowledged segments
- typically requires 4 segments to close: 1 `FIN` and 1 `ACK` for each direction
  - can be optimised:
    * Host A sends `FIN` request
    * Host B responds with `ACK` + `FIN` together
    * Host A sends `ACK`
    * connection is closed

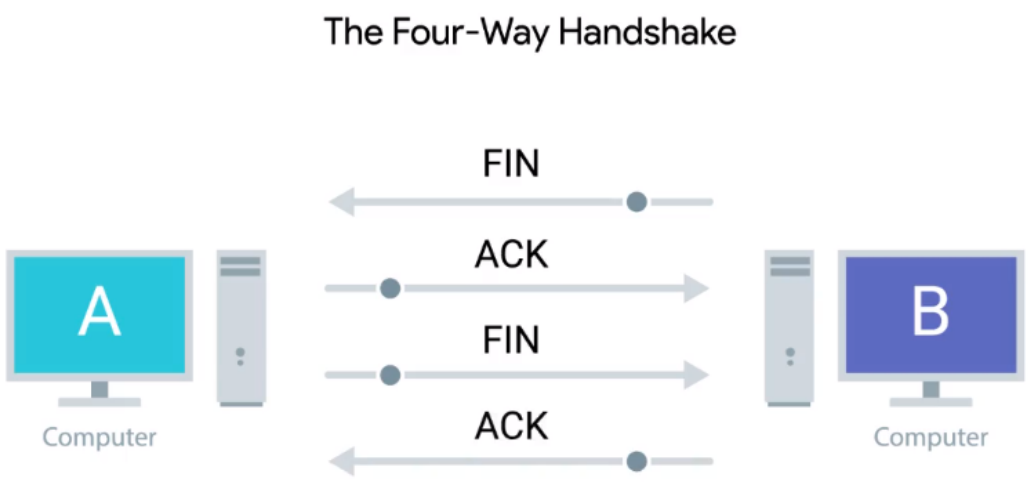


Figure 11: closing_tcp_connection

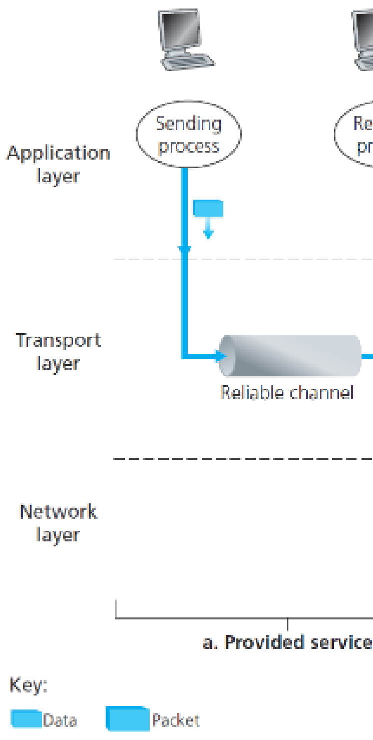
- **RST**: reset, hard close of a connection
  - sender is closing the connection and will not listen for further messages
  - sent in reply to a packet sent to a 5-tuple with no open connection e.g. invalid data being sent; crashed process that left a remote socket open that OS is cleaning up
  - can be used to close a connection but `FIN` is the orderly shutdown

`SYN` flooding

- popular attack in 90s for DoS of a server
- arbitrary initial random sequence number:
  - server needs to remember initial sequence number for each receive `SYN` request
  - attacker would make initial `SYN` requests then not send appropriate `ACK`
  - server gradually fills up queue with sequence numbers for now defunct connections
- one solution: `SYN` cookies
  - rather than store sequence number, derive it from connection information and a timer that creates a stateless `SYN` queue with cryptographic hashing
  - performance cost to validate `SYN` cookies, but preferable to being unresponsive
  - typically only enabled when under attack

Reliable Data Transfer

- reliable data transfer is fundamentally important problem in networking and applies at transport layer, link layer, application layer
- service abstraction provided to upper layers is a reliable channel through which data can be transferred



  - reliable channel: no data bits corrupted, all data delivered in order sent
    rdt: reliable data transfer; udt: unreliable data transfer
- **unidirectional data transfer** is the focus here, but **full duplex**, while conceptually similar, is tediously detailed
-