Transport Layer

Arnav Gupta

December 1, 2024

Contents

1 Transport Layer Services

Provides logical communication between application processes running on different hosts.

Transport protocols actions in end systems are:

- sender: breaks application messages into segments and passes to network layer
- receiver: reassembles segments into messages and passes to application layer

Two main transport protocols are TCP and UDP.

Network layer is logical communication between hosts, transport layer is logical communication between processes (relies on and enhances network layer services).

Sender:

1. passed an application layer message
2. determines segment header fields values
3. creates segment
4. passes segment to IP

Receiver:

1. receives segment from IP
2. checks header values
3. extracts application layer message
4. demultiplexes message up to application via socket

TCP (Transmission Control Protocol):

- reliable, in-order unicast delivery between 2 processes
- congestion control (throttle sender when network overloaded)
- flow control (sender won't overwhelm receiver)

- connection-oriented (connection set-up, stateful)
- does not provide timing, minimum throughput guarantee, or security

UDP (User Datagram Protocol):

- unreliable, unordered delivery between 2 processes
- transaction-oriented, stateless
- no-frills extension of best-effort IP
- does not provide reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup

Sometimes UDP is better if sped is more important than reliability.

2 Sockets, Multiplexing, and Demultiplexing

Process: program running within a host

With the same host, two processes communicate using IPC. In different hosts, they exchange messages.

Client process initiates communication while **server process** waits to be contacted.

Application process sends/receives messages to/from its **socket**, which is where messages are sent and received from.

There are two sockets involved in communication, one on each side.

UDP sockets are for unreliable datagrams, TCP sockets are reliable and byte stream oriented.

To receive messages, processes must have IDs that include both IP address and port numbers associated with process on host. (ex. HTTP server: 80, mail server: 25)

Host device interface has unique 32-bit IP address. Since many application processes can be running on the same host, more than the IP address is needed to ID the process.

Server usually has a well-known port and the client picks an unused temporary port.

Port number uniquely IDs the socket, so cannot use the same port number twice with the same address. The OS enforces uniqueness by keeping track of which port numbers are in use (stops 2nd program from using same port number).

Multiplexing at sender: transport layer handles data from multiple sockets and adds transport header (used for demultiplexing)

Demultiplexing at receiver: transport layer uses header info to deliver received messages to correct socket

- host receives IP datagrams with source and destination IP addresses
- host uses IP addresses and port number to direct segment to appropriate socket (differs for UDP and TCP)

2.1 UDP Sockets

Connectionless: each UDP segment handled independently and identified by a pair (IP address, port number)

Process using UDP in a client will create a socket with its IP address and a temporary port number (for return) and send segment to UDP socket in server using server IP address and permanent port number.

When host receives UDP segment, check destination port number in segment and direct UDP segment to socket with that port number.

IP/UDP datagrams/segments with the same destination port number but different source IP address or port numbers are directed to the same socket at receiving host.

2.2 TCP Sockets

Client must first contact server (requires server process running and server socket created to welcome client contact).

Client contacts server by creating TCP socket, specifying IP address, and port number of server process. With this, client TCP establishes connection to server TCP.

When contacted by the client, the server TCP creates a new socket for the server process to communicate with that particular client, which allows the server to talk to multiple clients.

From an application viewpoint, TCP provides reliable, in-order byte stream transfer between client and server.

A TCP socket is identified by the source and destination IP addresses and the source and destination port numbers.

To demux, the receiver uses all 4 values to direct the segment to the appropriate socket.

The server may support many simultaneous TCP sockets, with each socket ID'd by its own tuple (associated with a different connecting client).

3 Connectionless Transport: UDP

No-frills, bare bones Internet transport protocol that is best-effort, so segments may be lost or delivered out of order.

No handshaking between UDP sender and receiver, and each segment is handled independently.

With UDP, no connection establishment delay. UDP is also simpler (no connection state), has small header size, and no congestion control (as fast as desired).

UDP is used for streaming multimedia apps (loss tolerant, rate sensitive), DNS, SNMP (management protocol), and HTTP/3.

If reliable transfer needed over UDP, add needed reliability and congestion control at application layer.

UDP segment has source port number, destination port number, length (including header), checksum, and payload (application data).

With UDP checksum, detect errors in transmitted segment.

Sender treats contents of UDP segment (including header and IP addresses) as a sequence of 16-bit integers. The **checksum** is a one's complement sum of segment content (2 sequences of 16 bits at a time). Checksum is put into UDP checksum field.

Receiver computes the sum of all 16-bit integers including checksum.

4 Connection-oriented Transport: TCP

TCP provides a completely reliable, connection-oriented, full-duplex byte stream transport service that allows two application processes to form a connection, send data in either direction, and then terminate the connection.

Non-trivial since IP can lose, re-order, corrupt, delay, fragment, or duplicate packets.

TCP provides:

- reliable data transfer
- congestion control
- flow control

TCP is:

- point-to-point (1 sender, 1 receiver)
- byte stream (no message boundaries)
- full duplex (has maximum segment size)
- cumulative ACKs
- pipelining with congestion and flow control setting window size
- connection-oriented with handshaking to initialize states
- flow controlled so sender does not overwhelm receiver

4.1 Segment Structure

TCP segment consists of:

- source and destination port numbers
- sequence number for retransmission
- ACK number: sequence number of next expected byte
- length of TCP header
- congestion notification bits
- ACK bit
- connection management bits

- receive window for flow control (number of bytes receiver willing to accept)
- header checksum
- TCP options: variable length
- application data: variable length data sent by application into TCP socket

4.2 Reliable Data Transfer

TCP creates reliable data transfer service on top of IP's unreliable service through pipelined segments, cumulative ACKs, and single retransmission timer.

Retransmissions are triggered by timeout event, duplicate ACKs, and no NACK.

Sequence number is byte stream number of first byte in segment's data.

The ACK has the sequence number of the next byte expected from the other side (cumulative ACK).

How the receiver handles out-of-order segments is up to the implementer.

Set TCP timeout value longer than RTT, but RTT can vary. Too short causes premature timeout or unnecessary retransmissions and too long causes slow reaction to segment loss. To estimate RTT, sample measured time from segment transmission until ACK receipt. Since sample can vary, better to average several measurements.

In many TCP implementations, minimum timeout value is 500ms due to clock granularity.

Exponential weighted moving average (EWMA) has influence of past sample decreasing exponentially fast:

$$\text{EstimatedRTT} = (1 - \alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}$$

typically using $\alpha = 0.125$.

Timeout interval used is EstimatedRTT plus some safety margin like $4 \times$ DevRTT, where

$$\text{DevRTT} = (1 - \beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|$$

typically using $\beta = 0.25$.

4.2.1 Sender

Sender gets data from application, creates a segment with a sequence number (byte stream number of first data byte in segment) and then starts the timer if not already running (timer for oldest unACKed segment). Sender can have a timeout after some interval after which it retransmits the segment that caused the timeout and restarts the timer. Sender can receive and ACK, which can acknowledge previously unACKed segments (update what is known to be ACKed and start timer if there are still unACKed segments).

4.2.2 Receiver

The receiver can get an in-order data segment with expected sequence number. The TCP receiver waits for next segment, and if none, then send ACK.

The receiver can get an in-order segment with the expected sequence number but one other segment has an ACK pending. The receiver immediately sends a single cumulative ACK, ACKing both in-order segments.

The receiver can get an out-of-order segment with a higher than expected sequence number. The receiver immediately sends a duplicate ACK indicating the sequence number of the next expected byte.

The receiver can get a segment that partially or completely fills a gap. The receiver immediately sends an ACK provided that the segment starts at the lower end of the gap.

4.2.3 Fast Retransmit

Timeout period can be relatively long, so long delay before resending lost packet.

Can detect lost segments via duplicate ACKs, where the sender often sends many segments back-to-back and so if a segment is lost, there will likely be many duplicate ACKs. So always resend unACKed segment with smallest sequence number, since it is likely that the unACKed segment is lost.

4.3 Flow Control

Can be issues if network layer delivers data faster than application layer removes data from socket buffers.

Flow control: receiver controls sender so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP receiver advertises free buffer space in `rwnd` field in TCP header:

- `RcvBuffer` size set via socket options (typical default is 4096 bytes), but autoadjusted by many OSs

Sender limits amount of unACKed data to received `rwnd`. Under normal conditions, receive buffer will not overflow.

4.4 Connection Management

Before exchanging data, sender and receiver handshake (agree to establish connection and on connection params).

Client creates socket and server accepts socket connection (both sides establish connection).

2-way handshake does not always work in network due to :variable delays, retransmitted messages due to message loss, message reordering, and lack of visibility of other side.

Can have issue with half-open connection (no client).

4.4.1 Startup/Shutdown Solution

Uses three-message exchange, known as 3-way handshake.

Necessary and sufficient for unambiguous, reliable startup and unambiguous, graceful shutdown. `SYN` for startup and `FIN` for shutdown.

1. client chooses initial sequence number and sends TCP SYN message
2. server receives this, chooses initial sequence number, and sends TCP SYNACK message to ACK client SYN
3. client receives SYNACK which indicates server is live, and sends ACK for SYNACK which may contain client-to-server data (connection established on client)
4. server receives ACK which indicates client is live (connection established on server)

`SYN` has flag set to 1. `SYNACK` has both `SYN` and `ACK` flags set (with sequence number and ACK number).

Client and server both each close their side of the connection by sending TCP segment with `FIN` bit of 1. On receiving `FIN`, `ACK` can be combined

with own `FIN` in response. Simultaneous `FIN` exchanges can be handled.

5 Principles of Congestion Control

Too many sources sending too much data too fast for the network to handle. Can manifest in long delays or packet loss.

Long delays can occur when the maximum per-connection throughput is reached and so delay increases greatly as arrival rate approaches capacity (infinite buffer).

Packet loss can occur with finite buffers. Sender then retransmits the lost, timed-out packet. Application layer input and output will be the same, but transport layer input includes retransmissions.

Packet loss can also happen due to the sender timing out prematurely and sending two copies, both of which are delivered (unneeded retransmissions).

Retransmissions cause more work for given receiver throughput. Unneeded retransmissions mean link carries multiple copies of a packet, decreasing maximum achievable throughput.

When a packet is dropped, any upstream transmission capacity and buffering used for that packet was wasted.

Congestion insights:

- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- unneeded duplicates further decrease effective throughput
- upstream transmission capacity/buffering wasted for packets lost downstream

5.1 Approaches

5.1.1 End-to-End

No explicit feedback from network. Congestion inferred from observed loss and delay.

This is the approach taken by classic TCP.

5.1.2 Network-Assisted

Routers provide direct feedback to sending/receiving hosts with flows passing through congested router. This may indicate congestion level or explicitly set sending rate.

Used in TCP ECN, ATM, DECbit protocols

6 TCP Congestion Control

TCP uses its window size to perform end-to-end congestion control where `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control).

Window size is the number of bytes allowed to be in flight simultaneously:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

Basic idea of congestion control is for the sender to control transmission by dynamically changing the value of `cwnd` based on its perception of congestion.

Due to flow control, effective window size is

$$\text{EW} = \min(\text{cwnd}, \text{RcvWindow})$$

6.1 Approach

Sender increases transmission rate probing for usable bandwidth until loss event occurs, then decrease `cwnd`. By controlling the window size, TCP effectively controls rate.

Transmission rate when using window control is at best equal to one window of bytes every round trip time:

$$\text{rate} = \approx \frac{\text{cwnd}}{\text{RTT}} \text{bytes/sec}$$

Some schemes include: Tahoe, Reno, Vegas, Cubic, NewReno, Sack, SCTP, TFRC, and BBR.

Phases of congestion control: slow start and congestion avoidance.

`ssthresh` defines threshold between a slow start phase and a congestion avoidance phase.

6.1.1 Slow Start (SS)

TCP starts with `cwnd` = 1 segment and increases the window size as ACKs are received. TCP increases the window fast (one segment for every ACK received).

When `cwnd` = `ssthresh`, TCP goes into Congestion Avoidance.

Typically, during slow start, `cwnd` doubles every RTT (exponential increase).

On loss event, `ssthresh` is set to $\frac{1}{2}$ of `cwnd` just before loss event.

6.1.2 Congestion Avoidance

In this mode most of the time.

TCP increases `cwnd` slower (how depends on the version).

In Tahoe and Reno, `cwnd` is increased by one segment every RTT (linear increase).

TCP continues to increase `cwnd` until congestion occurs.

To indicate congestion, the types of loss events to check for are:

- 3 duplicate ACKs (temporary congestion)
- timeout (significant congestion)

When congestion occurs, decrease the window size `cwnd`.

In TCP Reno:

- if timeout occurs, `sshthresh` = $\frac{1}{2}$ `cwnd`, `cwnd` = 1, and go to Slow Start
- if 3 duplicate ACKs occur, `cwnd` = `ssthresh` = $\frac{1}{2}$ `cwnd` and stay in the phase it is in (fast recovery)
  - in Tahoe, sender goes back to Slow Start

Additive Increase: increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease: cut sending rate in half at each loss event

AIMD sawtooth behaviour is probing for bandwidth. AIMD optimizes congested flow rates network-wide and has desirable stability properties.

With TCP CUBIC, ramp to W_{max} (sendng rate at which congestion loss was detected) faster but approach W_{max} more slowly. This gives higher throughput for longer.

- K is when TCP window size reaches W_{max}
- goal is to increase W as a function of the cube of the distance between current time and K
- default in Linux, most popular on web

TCP increases sending rate until packet lost occurs at some router's output, which is the **bottleneck link**. Best to focus on congested bottleneck link.

6.1.3 Delay-Based

Keeping sender-to-receiver pipe just full enough, by keeping bottleneck link busy transmitting but avoiding high delays or buffering.

Let RTT_{min} be the minimum observed RTT (uncongested path). Uncongested throughput with congestion window `cwnd` is $\frac{\text{cwnd}}{RTT_{min}}$.

- if measured throughput is close to congested throughput, increase `cwnd` linearly
- if measured throughput far below uncongested throughput, decrease `cwnd` linearly

This is congestion control without inducting/forcing loss. Maximize throughput while keeping delay low.

Some deployed TCPs take this approach.

TCP deployments often implement network-assisted congestion control:

- two bits in IP header marked by network router to indicate congestion
  - policy to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets the ECE bit on ACK segment to notify sender of congestion (E bit)
- involved both IP and TCP

If K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K.

TCP is fair under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance