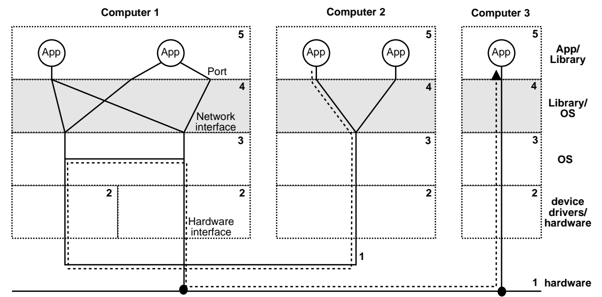
Lecture 8

Transport Layer

References:

- Textbook chapter 6 up to sections 6.5.10, seciton 5.3.4.
- RFCs: 768 (UDP), 793 (TCP), 1122 (Clarifications of TCP), 1323 (Extensions of TCP).

Review: what transport layer does



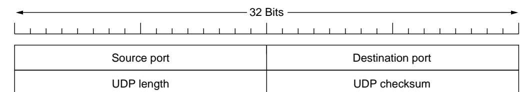
- It serves for **multiplexing** the network layer interface.
- It serves to **provide additional functions** other than those supported directly by the network layer.

Addressing

- A computer typically has one or a few network interface, which **must serve all users**. This is done by allowing users to create **connections**.
  The situation should be familiar to you: E.g., computer has only one CPU, one big trunk of memory, etc., and must serve all users.
- End-points of connections are called **TSAP** (Transport Service Access Point). Each program can allocate some TSAP for communication.
- Each TSAP in use has a **unique** identifier to distinguish among them.
- Application layer produces data and sends them to the TSAP. The transport layer **tag it with that TSAP identifier** of both end-points to form **TPDU** (Transport Protocol Data Unit) and send it to the network layer.
- In Internet, TSAP are **ports**, TSAP identifiers are **port numbers**, and TPDUs are **segments**.

UDP packets

- The TCP/IP protocol suite has two primary transport protocol. **UDP** (User Datagram Protocol) is the simpler one.
- The idea is simple: **expose the capability of IP packets** to the user, by allowing multiplexing.
- With such a simple aim, each UDP segments are encapsulated in a **single IP packet** (protocol 17), with a very simple format…

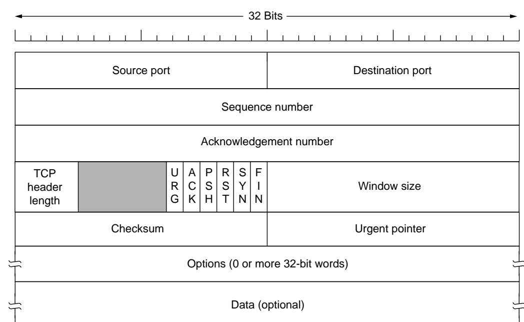


- No IP address is specified. The IP packet will contain it anyway, so the network layer can tell it about the source and destination IP addresses.

From datagram to virtual circuit

- UDP is useful when (1) **reliable** transfer is really **not needed**, and (2) the communication model **does not allow connection** (i.e., multicast).
- UDP provides not provide any mechanism for flow control, duplicate handling, lost frame handling, etc.
- **Why they happen?** (1) Underlying data link is **unreliable**, (2) different packets are **routed** differently and thus arrives out-of-order, or (3) some network layer is forced to **drop packets** when buffers run out.
- If applications want **reliability**, they have to **add** flow control to it, using something similar to **sliding window**.
- Or alternatively, one can use **TCP** (Transmission Control protocol), which does everything to provide reliable stream-like service.

TCP packets



Many fields have obvious usage: port numbers, checksum, header length, sequence and acknowledgement numbers (for sliding window).
Acknowledgement number is for piggybacking, valid when ACK=1.

TCP packets, continued

Meaning of other fields and flags…

- **URG** is for **urgent** data. Applications receiving urgent data will receive a notification (e.g., signal), and thus read the stream immediately. The **ending location** of the urgent data is stored in **Urgent pointer**, and applications can check that all urgent data are read.
- **PSH** allows interactive applications to ask the receiver **not** to **buffer** the data. This is useful for interactive data like keystrokes.
- **RST** allows a connection to be **reset**, usually for **rejecting** requests.
- **Options** allow **extensions** to the protocol. E.g., the selective repeat algorithm need NAKs, and this is implemented by an option.
- **SYN** and **FIN** are used to **establish and tear down connections**. **Window size** is for flow control. We'll talk more about them.

Addressing

- TCP connections are identified by **addresses and port numbers** of **both** ends of the connections.
- So at the same time, a port of a computer can be **listening**, and a connection of the same port may be **connected** to multiple other ends.
- And since the other end is identified with both an address and a port number, the **same computer** can make **multiple connections** to the **same service** of the same server (by using different port).
- This is in contrast with UDP. UDP has no "connection", so only **one process** can be **binded** to a port at a time.

Windowing algorithm: a recap

- Data sent has a **sequence number** associated with it. Both the sender and the receiver has a **window** of sequence number.
- The sender window specifies **what data can be sent** before **waiting.** When acknowledgement correctly arrives, the window is advanced.
- The receiver window specifies **what data it expects**. Data outside the receiver window will be dropped without further consideration. The window advances when data arrives correctly.
- Acknowledgement may be sent by a dedicated packet, or piggybacked via a packet sent over **traffic of the reverse direction**.
- After data is sent, a timer is set up. If it **timeouts** before it is **acknowledged**, it will be **resent**.
- NAKs can be used to **force** resent before timeout, when receiver notice some data is missing.

Differences to data link

There is a big difference: the **assumption of the link**.

- Physical layer: there is no **reordering** or **duplicated data**, and delay is caused only by **queueing** and is usually quick.
- Network layer: packets **reorders** and **duplicates** arbitrarily, and delay caused by congestion and change of route is **long** and **varies** a lot for the duration of the same link.

At a sudden we don't know how to set the **windowing parameters**…

- How many **buffers** to allocate? Too large will waste memory and delay resends, too small will slow down communication and waste bandwidth.
- How to set the **timeout**? It shouldn't timeout unless a packet is really lost, but the **large variance** means that delays real timeouts.

The worse news: all these changes even during the same connection!

TCP's version of sliding window

- **Every byte** of data transferred has a sequence number (**seq**), as well as some event like establishing and closing of a connection.
- Seq's have **32 bits**, which is thought to wrap around very slowly (although nowadays it is not as slow as desired).
- The actual sequence number used is **the next seq expected**. Upon receiving 123–456, ack will be 457, and the sender knows everything before 457 arrives correctly.
- Buffers are allocated **dynamically**. If a connection seems to be active, more buffers can be allocated so that transfer can speed up.
  But… the network might not be happy with the speed up. More about it later.
- Segments sent include the **remaining buffer size** in the *Window size* field. The partner then adjusts how much data to send at a time.

Problem of Dynamic Connections

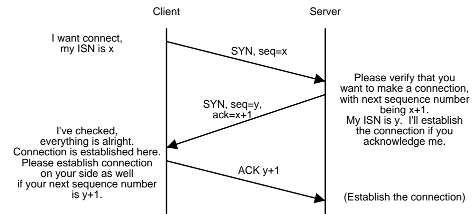
- Duplication and reordering causes major problems on **establishment of connections**. Let's see a problematic situation…
  - Say connections starts with seq=0 in both directions.
  - Suppose a server starts, **listens** to a well-known port, and a client **connects** to it. The network is congested, so every packet is resent a few times. The whole connections completes.
  - Unluckily, the server **crashes** and **restarts** immediately afterwards. Even more unluckily, all duplicated packets arrive after the restart.
  - The server has no option but to **treat them as creating a new connection**, repeating whatever the client asks the server to do.
- Things requested once might be done twice. So much for reliability.
- Problem: server can't tell whether a segment is old or new.

The use of clock

- With delayed duplicated packets coming out at any time, we really don't have a better solution. But in most networks the **delay is bounded**.
- E.g., IP packets have TTL set to at most 255, so after 255 hops it is guaranteed to be dead. Most implementations set this much smaller.
  It is usually assumed that no packet arrives after a delay of a few minutes.
- With this assumption, the problem can be solved by **choosing the initial sequence numbers (ISN)** carefully.
- Idea: **old** packets with the **same seq** should be **dead**.
  So if seq of a segment falls in the receiver window, it's new. Different connections use independent segment numbers, so at any time there should be only 1actively in use.
- How to choose? TCP assumes that each computer has a **clock**, which increments every 4 µs even during the computer crashes. It is used as the ISN for new connections.

Three-way handshake

During a connection setup, ISNs for both directions are **exchanged**. We call the process **three-way handshake**, named by the number of segments involved:



Note that three-way handshake serves to exchange ISNs, and has **nothing to do with whether delayed duplicates can get accepted**.
That can only be guaranteed if sequence numbers are used wisely.

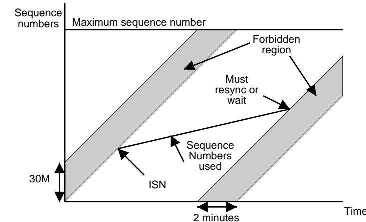
Using sequence numbers wisely

How to make sure old packets with same seq are dead?

- Assumption: **nobody wants more than 1 seq per 4 µs.**
  If every byte need a seq, it means 250KB per second. We are now at 100Mbps, so things has to be done about it. To fix it, one can negotiate a seq per some bytes by using some TCP options.
- With this assumption, **old packets** have seq **smaller** than even ISN of new connections.
- But… 32-bit seq can **wrap around**!
  It wraps around every 4.77 hours.
- If a segment has seq=x is used, where delayed duplicates might arrives when a new connection will use x as ISN, we have a problem.
- Solution: Don't use x now. Either **delay** sending a segment, or **re-synchronize** the sequence numbers so that the current ISN is used.

Forbidden region

- Suppose we assume a packet and its acks of the network vanish in 2 minutes. In this amount of time, ISN increases by 30M.
- If current ISN is x, then using seq in the range from x+30M-1 to x will be **dangerous**, since a delay can cause confusion **if the computer reboots now**. We call the region of seqs the **forbidden region**.



Graceful connection release

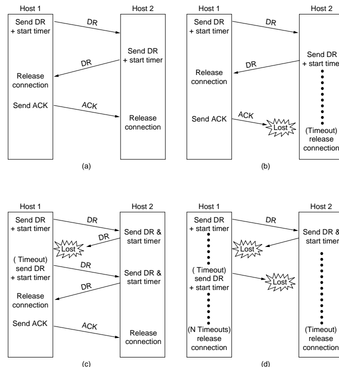
Now let's turn to closing, or **releasing**, connections.

- Two release styles:
  - **Assymmetric release**: if one side release, the connection is broken. Like telephone. **Can lose data.**
    How many times did you want to talk when the other party has hanged up?
  - **Symmetric release**: the connection is treated as **two** unidirectional links. Only sender can terminate a connection (by saying "I've got nothing more to tell"). Won't lose data.
- We want connection release to be **graceful**: after closing connections, **both** sides should be able to free all resources.
- Unluckily, that is **difficult** when packets can be **lost**.

The two-army problem

- Consider the protocol that the one wanting to terminate, say A, **sends "I'm done"**, and the receiver, say B, then **sends "Okay"**.
- After B sends the ACK, can it release the resources? **No!**
  B has no way to know whether A receives the ACK.
- Can we add one "Ack-to-ack" from A to B? Then A cannot know whether B receive it, and can't stop!
- The problem is called the two army problem: if two armies each wants to attack only when the other attack as well, and the only media between them can loss message, there can be provably no attack.
- The problem: suppose there is a protocol will allow an attack in finite number of message exchange, there is a case where the least messages are sent. Then how one know the last message is not lost?

Solution: timeouts



What if even the first DR (in TCP, FIN) is lost? The other end notices nobody is sending for a long time, timeout, and close the connection.

TCP connection states

These just show how TCP actually do it…

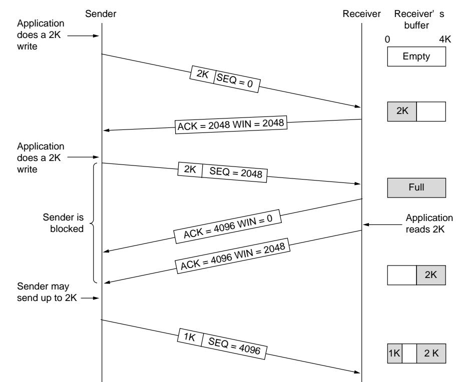
- **LISTEN**: listen has been called (although no connection yet).
- **SYN SENT**: connect has been called, SYN sent. Waiting for ACK.
- **SYN RCVD**: received a SYN. Sent SYN+ACK, waiting for ACK.
- **ESTABLISHED**: both sides can send and receive.
- **CLOSE WAIT**: the other side has sent a FIN. Our side can continue to send, but reading will return EOF.
  This is called passive close.
- **LAST ACK**: after the remote is done, our side has closed. Might need to send the remaining data in buffer, and then send a FIN. Wait for all the acks.

TCP connection states (cont'd)

- **FIN WAIT 1**: our side has closed. May still need to clear the buffer, and send the FIN. Then wait for ACK for the FIN.
  This is active close: we close without the other party telling it first.
- **FIN WAIT 2**: Got the ACK, but the other side is still operating. Continue receive its data.
- **CLOSING**: after FIN WAIT 1, received FIN from the other side. Can stop once everything gets through.
- **TIME WAIT**: Everything is done. But the other side might resend, since our ACK might get lost. Wait here to process those resends.
  Note that this is not needed in passive close: in that case we are the one to wait for an ACK (and if not received, resend).
- **CLOSED**: now the connection can be reused.

States are shown when you type "`netstat -t`" or use other tools, so they are more than "just implementation".

Buffer management



Other subtleties of windows management

- It seems straight-forward: the receiver announce the availability of window in ack, and the sender comply by sending more.
- But what if **window size change** get **lost**? Both sender and receiver would be waiting…
- Answer: The sender periodically sends a **query** for the window size when the receiver indicates that it has no buffer, to get out of the loop.
- Other problems about efficiency . What happen if the sender application **write many 1 byte messages** (typical for telnet etc)?
- A lot of 1-byte message will be sent… but 1-byte message means a 21-byte segment, or 41-byte packet, or 67-byte Ethernet frame… and there's a 66-byte frame for the reverse traffic on ack!
  Not usually a problem for LAN, but on WAN this will add fire to congestion.

Nagle and Clark

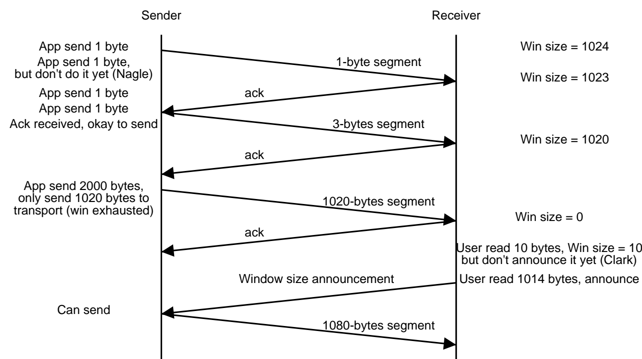
TCP is modified by a few rules to fix the problems.

- **Nagle**: If upper layer push 1-byte, suppress sending of further bytes until it is acknowledged. (At that time we probably have more bytes.)
  The cost: some delay. Some applications don't like that, and can set the TCP_NODELAY socket option to disable it. See `tcp(7)`.

There's a reverse problem: receiver gets 1-byte at a time, causes many window size changes each asking for 1 more byte (**silly window syndrome**)…

- **Clark**: don't send a window size update until it reaches either the receiver's maximum transfer unit or half the allocated buffer.
  This is less problematic than Nagle, as this occurs only when the sender has a lot of data to saturate the buffer quickly.

Events in Nagle and Clark



Setting timeouts

- The sliding window algorithm needs **timeout** to resend segments.
- Suppose we send a segment, wait for t seconds, and can't see its ack. Should we timeout? Yes if **the packet is most likely lost**.
- The probability depends on the current **average round-trip-time** (RTT) and its **variance**, but they changes **during the connection**.
- We use a trick that we know from the OS course: a **weighted average**.
- When a packet is acknowledged, its **elapsed time** is measured, and is used to **update an estimate** of the average *RTT* by a rule $RTT = \alpha RTT + (1 - \alpha)M$, where $0 < \alpha < 1$, e.g., $7/8$.
- **Karn's rule**: don't update *RTT* on retransmitted segments. When retransmit, double the current timeout value to slow down retries.
  For retransmitted segments, we don't know which transmit an ack refers to.

Setting timeouts (cont'd)

- It is quite troublesome to estimate the standard deviation of the mean RTT. So instead we estimate the **mean variance** *D* (i.e., average deviation from mean) of *RTT*.
- The update rule: $D = \alpha D + (1 - \alpha)\,|RTT - M|$.
- With an update to *RTT* and *D*, the **retranmission timeout** is updated to $RTT + kD$ for some k.
  In practice, k is chosen to be 4 to make it easy to calculate.
- **Other timers** needed:
  - **Persistence timer**: how long the sender waits when it is blocked due to the exhaustion of receiver's buffer, before sending a query?
  - **Keepalive timer**: after how long the sender should send some data to indicate it is still there even if it has no data to send?

Congestion control

- Network layer of intermediate routers has **limited capacity**, due to limited memory, bandwidth and CPU cycles.
- If **many transport level connections** (perhaps between different pairs of computers) must go through the same router, the **queue** of routers builds up, so everybody waits longer.
- If that persists, routers memory will get used up, so packets must be **discarded**. The transport layer segments are thus lost.
- Note the difference between flow control and congestion control:
  - Flow control solves a **local** problem that **either end** has no capacity to process the data.
  - Congestion control solves a **global** problem that **intermediate** router runs out of capacity.

Two involved layers

Both the network layer and transport layer are involved in any congestion control scheme.

- The end-points don't know whether a connection or new segment will cause congestion **in the middle** of the network. Only the **network layer** can **detect** them.
- The network layer cannot **stop traffic** by itself, since transport layer must **reduce rate of sending segments** or **connecting** before a congestion get resolved.

In general, congestion are **detected** in the network layer, and the transport layer is **notified** by the network layer in some way.

Congestion prevention

One way to solve it: **don't allow congestion** to occur at the first place.

- **Admission control**: When a connection is built, each intermediate router attempts to **allocate enough resources** for the connection.
- If that succeed, **congestion is unlikely** since all the needed resources is guaranteed to be there already.
- If the allocation cannot be done, **don't allow the connection** to be made. The transport can then try again later or on another route.
- The resources needed by each connection is carefully calculated using the **average data rate and its variance**.
  If variance is too large, the traffic can be "shaped": delay packets when too much data arrives.

We call such scheme **congestion prevention schemes**, since congestion never occurs. The cost: unused resources for bursty traffic.

Open-loop vs. close-loop solution

- Congestion prevention: do everything when **making connections**.
  And do nothing afterwards: all resources are there.
- **Poor-match** to **datagram subnet**: no connection is needed!
  Probably alright for TCP, but what for UDP users?
- **Alternative**: allow connection to be made at any time, but monitor the network to notice congestion and ask transports to slow down.
- People familiar with dynamic systems consider it as a **negative feedback**: sending data rate increase the utilization, and too high utilization reduce data rate, so congestion is controlled.
- We thus call such schemes **close-loop** schemes, in contrast to **open-loop**, congestion prevention schemes which never watch the network.
- This improve utilization, but provides no **performance guarantee**.

Load shedding

- **Load shedding** is a simple way for network layer to tell the transport ends about the congestion: **do nothing!**
- When buffer in a router runs out, just **throw away some packet**.
  Which to throw? Some choices: the earliest, the latest, at random. They work similarly.
- The transport will eventually notice that **data is missing**. It should assume that a **congestion** is occurring.
  This assumes a data link which rarely loss frames, which is usually the case.
- Then the transport should **reduce** the data rate.
  Very unnatural thing to do: when you send and receive nothing, don't try harder. Instead, be more gentle and send less!
- Once enough connection has rate reduced, the connection there is no more data loss, and the data rate can increase again.

Detecting congestion and RED

- If exhausted buffer means a congestion, we can use **buffer utilization** as an indication about the **degree** of congestion.
- An even **more aggressive** way to know congestion is to watch the **line utilization** of an output line. This allows congestion to be detected even before resources is used up.
- Routers would keep a **weighted average** of the quantity being measured. If it goes above some threshold, a congestion control strategy is triggered.
- E.g., TCP allows **throwing away random packets** when the buffers of a router is highly utilized: **RED** (Random Early Detection).
  In Linux, this is available only if the kernel is reconfigured to support RED. See the comments in `net/sched/sch_red.c`.
- This tells the transport layer to slow down (by missing segments) **before** the line becomes completely unusable.

Choke packets and its problem

Another way to tell the transport layer to stop: **ask explicitly**.

- There is a ICMP packet type **Source quench**. Such packets are usually called **choke packets**.
- When a host receives such a packet, it should **slow down** sending packets to that host.
- It is possible for a protocol to specify that **other intermediate routers** between the source and the choking router to slow down as well.
  Although IP doesn't specify that.
- But there's a problem: choke packets are also packets, and thus **adds to the congestion**!
- Nowadays choke packets are not sent; transport users should **treat lack of acknowledgement** or response as indication of congestions.

Congestion handling in TCP

When a TCP connection **timeouts waiting for acks**, we assume there is congestion, so the connection must **slow down** sending. But **how**?

- Answer: **reduce the size of sending window**.
- Until now we assume that the sending window depend only on the **window size** announced by the other end. This is not the full picture.
  That deals with flow control, not congestion control.
- There is another window, called **congestion window**. It **estimates** how much data can be sent without problem.
- Size of sending window is the **minimum** of the *window size* announced by the *receiver* and the *size of the congestion window*.
- When a timeout occurs, the size of **congestion** window is **reduced**.
- At after a while, segments start to get acknowledged normally (congestion is over), it is enlarged again to speed up the connection.

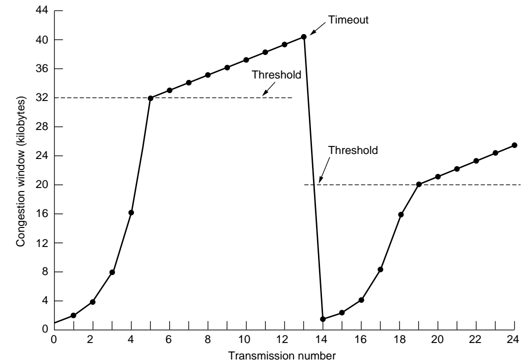
Congestion control algorithm: AIMD

- 2 variables per connection: **threshold** t, **congestion window size** w.
- Most of the time, the algorithm is in **congestion avoidance mode**, when it **seeks a good size** for t. In this mode, $t = w$.
  In the other mode, $w < t$.
- **Increase**: It keeps counting the number of bytes that gets acknowledged. When it reaches t, t is incremented by the **maximum segment size** (MSS), and the count is reset.
  MSS depends on OS, which might be bounded by MTU, but might also be bounded by other factors like page size.
- **Reduction**: on timeout, t is set to $w/2$ (and w is set to 1 MSS).
- In effect, increment (by 1 MSS) occurs only after one **complete window**, while each reduction cuts the congestion windows by **half**.
- We call it **AIMD**: Additive Increase, Multiplicative Decrease.
  We will see why this is a good idea.

Slow start

- How should we set the initial t? If it is set too low, it takes a **real long time to fully utilize** a link with the linear increment algorithm!
- Answer of TCP: **set it large** (64KB), but don't use it as size of congestion window yet. Instead, set it to **1 MSS**.
- When $w < t$, the connection is in the **slow start** mode, when **acknowledgements** increase w by the size of segment acknowledged.
  If w would exceed t, w is set to t instead to start congestion avoidance.
- E.g., if initially $w = 1024$, then 1024 bytes can be sent. If it does, and all acks arrives, then w becomes 2048. If again the whole window is sent and acknowledged, w becomes 4096.
- So w increases **exponentially** fast, trying to match t. Once timeout occurs, we get a sensible value of t, when linear increment makes sense.
  Recall that t is set to $w/2$ on transmission timeout.

Example



Behaviour

- One consequence of AIMD: if two connections both loss a segment, the one with a **larger congestion window** gets **penalized** more heavily.
  Because it needs longer time for the additive increase to recover the lost congestion window size.
- If there are **multiple TCP connections with heavy traffic** through the **same congested router**, this has the tendency to **equalize** the congestion window of all these connections.
- This is conceived as being **fair**. There are problems, though…
  - Not all network packets are TCP. In particular, **UDP programs** has no obligation to use AIMD, and **can be unfair**.
  - Not all users of that router uses the **same number of connections**. One using more connections can use unfair amount of bandwidth.
- **New transport protocols will mandate** the use of AIMD, though.