Course on

Computer networks

Transport layer

Henk Corporaal

Leiden University
2000

Transport layer

Tasks:

- Provide reliable and cost effective end-to-end communication service to the application layer
- Independent of used network (shielding)
  - boundary between network and applications

Note: transport layer would not be needed if

1. networks are reliable
2. all networks use the same protocol

Transport layer

Overview:

- Transport service
- Elements of transport layer protocols
- A simple protocol
- Internet protocols
  - TCP and UDP
- ATM layer protocols
  - ATM Adaption Layer (AAL)
- Performance issues

Transport Service

Transport entity provides reliable service to application layer (see fig 6-1)

TPDU (transport protocol data unit) denotes message sent from entity to entity

- Both connection oriented and connectionless service

- Quality of Service (QoS) parameters:
  - throughput and transit delay
  - residual error ratio: fraction of lost/garbled messages
  - protection (against tapping)
  - priority
  - resilience: probability that transport layer suddenly terminates

Transport Service

Transport Service looks similar to Data Link Service

- Both provide point-to-point connection
- Both have to deal with error control, sequencing, flow control, retransmission, etc.

Differences:

- Transport connection is indirect
- Network has memory: packets may be stored (for many seconds) and suddenly show up
- Many connections have to be managed (instead of a fixed number of links)

Elements of transport protocols

Primitives for a simple transport service:

- LISTEN : block until some process connects
- CONNECT
- SEND
- RECEIVE : block until data TPDU arrives
- DISCONNECT

See fig. 6-5 for simple connection/disconnection state diagram

Elements of transport protocols

Berkeley Unix socket primitives:

- SOCKET : create new communication end point
  - returns file descriptor
- BIND : attach local address to socket
  - some socket addresses are universally known and need no bind
- LISTEN : accept connections; give queue size
  - non-blocking
- ACCEPT : block until connection attempt arrives
- CONNECT : attempt to establish connection
- SEND : send data
- RECEIVE : receive data
- CLOSE : release connection

Note: client does not need BIND; its socket does not need an address (the server does not need it)

Client and Server

Client:

```
s = socket(domain, type, protocol);
connect (s, socket-address, addr-length);          //blocking
while (...) { send (s, buffer, len, flags); }
```

Server:

```
s = socket(domain, type, protocol);
bind (s, socket-address, address-length);
listen (s, max-pending);
while (1)
{ new_socket = accept(s, sockete-address, address-length);   //blocking
    while (len = recv(new_socket, buf, sizeof(buf), flag) )
          fputs (buf, stdout);
}
```


Elements of transport protocols

Addressing

- TSAP (transport service access point); fig 6-8

How do I know the TSAP of the destination?

- Using well known addresses
  - works only for (stable) key services
- Use name server (or directory server)
  - connect to this server
  - send message and ask for TSAP address of needed server
  - set up a connection with needed server

  Note: new services have to register with name server

Elements of transport protocols

TCP addressing:

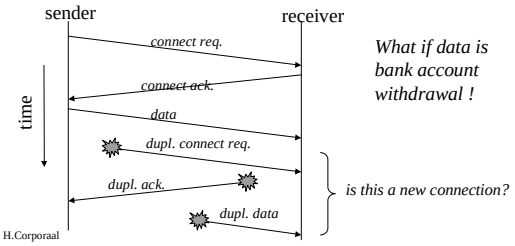
- (IP address, port number)

- Example ports of well knows services:
  - port 7: Echo
  - port 23: Telnet
  - port 25: SMTP (email)
  - port 80: HTTP (www)
  - port 110: POP (reading remote email)

Elements of Transport Protocols

Establishing a connection

Problem : packets can be retransmitted (duplicated) within the network => in principle whole transaction can be repeated



Elements of Transport Protocols

Solution:

- give packets unique sequence numbers
- restrict the lifetime of packets
  - offer a certain time after which we know that packets and its acknowledgements are ‘dead’
  - sequence numbers can be reused

Rule:

two equally numbered TPDUs need never be life at the same time

if they are, it’s a retransmission!

Establishing a Connection

- Each host has local clock running sufficiently fast
  - clock even runs when host is down!
  - use low order k bits as sequence # during connection set up.
  - following packets get successive sequence #
- Sequence # should be sufficiently large
  - after wrap around, TPDUs with old numbers are dead

Establishing a Connection

Crash scenario

- after crash host does not know seq. # of last sent packets

Where to start (with what seq #) ?

- wait T seconds (T = max. lifetime packets) and thereafter use previous procedure
- if T is large the host may wish to start earlier
  - it has to obey the forbidden zone (see fig. 6-10)
  - if not, after restart duplicate seq# may be still alive

Elements of Transport Protocols

Using sequence numbers we can set up a save protocol for making a connection

- Three-way handshake (fig 6-11[a])
  - 1. send connection request : CR (seq = x)
  - 2. acknowledge sequence # x : ACK (seq = y, ack = x)
  - 3. acknowledge sequence # y (by sender) : DATA (seq = x, ACK = y)

This protocol works even if delayed duplicates are around (fig 6-11[b,c])

- There is no combination of old packets which are not recognized

Elements of Transport Protocols

Connection release options:

- asymmetric (like telephone system)
- symmetric
  - treat connection as two separate unidirectional links which have to be closed separately

Asymmetric release may loose data : fig 6-12

Symmetric release works fine if amount of data to be transmitted is fixed and known

Otherwise: 2-army problem (fig 6-13): how does the blue army know it's save to attack?

2-army problem

No real solution exist

Proof:

- Assume an n-way protocol does the job
- the final n^{th} message is either
  - not essential $\Rightarrow$ remove it and use n-1 way protocol
  - essential
- In the latter case:
  - we are not sure if the n^{th} message arrives (there is no $n+1^{th}$ acknowledge) $\Rightarrow$ the protocol does not work since this message is essential ! (i.e. the army will not attack)

Connection Release

The 2-army problem makes clear that both sides are never sure if they both agree in release

Solution (partly):

- use three-way handshake protocol with time outs (see fig 6-14[a])
  - host 1 sends disconnect request + starts timer
  - host 2 responds with disconnected request + starts timer
  - host 1 acknowledges the request
- Most failures are covered (see fig 6-14[b,c,d])

Connection Release

Host 1 releases connection after N time outs

- if we don't do this the host would continue for ever when host 2 has broken the connection (e.g. due to time out, fig 6-14[d])

Failure if initial DR and all N retransmissions are all lost => half open connection (from host 2 to host 1)

- auto disconnect
  - disconnect if no TPDU arrives after certain time
  - requires to send dummy TPDUs now and then

Elements of Transport protocols

Flow control and buffering

Unreliable network:

- sender must buffer TPDUs as long as they are not acknowledged
- receiver does not need buffering; however, for high throughput with long delays a window can be used
- for efficient communication sender needs to know how many buffers are (still) available => dynamic buffer management using sliding window
  - e.g. receiver can send message: I have reserved X buffers for this connection

Buffer management

Example of sliding window protocol is shown in fig. 6-16

- deadlock may occur if allocation TPDU is lost

Solution:

- periodically request buffer status (of the receiver)

Sliding window of sender should be large enough to cover network delay (roundtrip time T)

- window size = network bandwidth * T
  - i.e. the number of 'outstanding' bytes

Elements of Transport Protocols

How to recover from host (server) crashes ?

Consider client server transaction

1. client sends a transaction TPDU
2. server receives and acknowledges transaction
3. server handles transaction data to application process

What if server crash between 2 and 3 ?

- client will not retransmit TPDU
- if we reverse steps 2 and 3, the client retransmits, and application process will handle transaction twice

Recover from Host Crash

8 options (2 x 4):

- 2 server options:
  - acknowledges first,
  - or it writes (to application) first
- 4 client options:
  - always retransmit (after server crash)
  - never retransmit
  - retransmit only when client has **no** unacknowledged TPDU
  - retransmit only when client has an unacknowledged TPDU

All 8 options may result in losses or duplicates (fig 6-18)

Conclusion:

Recovery from layer N crash can only be handled by layer N+1 enough status information has to be retained

Simple Transport Protocol

In section 6.3 an example is detailed assuming the following:

- asymmetric connection model using blocked CONNECT and LISTEN primitives
- symmetric release: both sides have to disconnect
- reliable network service: connection oriented
- use same header for transport and network layer
  - VC# is also transport connection #

Simple Transport Protocol

5 service primitives (see functions in fig 6-20)

con # = LISTEN (local)	blocking
con # = CONNECT (local, remote)	blocking, time out
status = SEND (con#, buffer, # bytes)	non-blocking
status = RECEIVE (con#, buffer, #bytes)	blocking
status = DISCONNECT (con#)	non-blocking

If status < 0 then failure

Failure reasons: transport address in use / illegal, host down, illegal buffer address, etc.

Simple Transport Protocol

Network interface: to-net & from-net functions

6 parameters:

- connection id (= VC #)
- Q : 1 if control message
- M : 1 if more data follows
- packet type (6 possibilities ; fig 6-19)
- pointer to data
- # bytes of data

Simple Transport Protocol

Each connection is in one of 7 states (pg. 513):

- IDLE: connection net yet established
- WAITING : outstanding Call Request
- QUEUED : received Call Request; no LISTEN yet
- ESTABLISHED : connection ready
- SENDING : wait for permission to sent a packet
- RECEIVING : waiting for data
- DISCONNECTING : local disconnect performed

State transition upon one of 12 events:

- one of 5 primitives executed
- one of 6 packets arrived
- timer expires (connection not accepted)

Simple Transport Protocol

FSM description:

fig 6-21 : 12 events x 7 states transition matrix

fig 6-22 : equivalent transition graph

Why transition matrix ?

- check every entry (= combination of event & state) for:
  - impossible events
  - illegal states
- use as implementation: using event handler A[event, state], being a pointer to appropriate procedure
- specification purpose

Internet Transport Protocols

TCP : transmission control protocol

- reliable; handle retransmissions
- connection oriented
- splits data stream into ~ 1500 byte pieces to form IP datagram
- re-assembly at destination (put in right order)

UDP : user datagram protocol

- unreliable
- connectionless
- little more than bare IP

TCP

TCP Overview

- Service model
- Protocol
- Header structure
- Connection management
- Transmission policy
- Congestion control
- Timer management
- Wireless TCP

TCP Service Model

Both sender and receiver have to create sockets

- socket # = IP address + Port# (16-bit, = TSAP)
- connection is identified by: (socket 1, socket 2)
- Port# < 1024 : well-known ports
  - FTP: 21, Telnet:23, SMTP: 25, HTTP: 80
- TCP connection is :
  - full-duplex, point-to-point
  - byte stream : message boundaries are not preserved (just like within all UNIX files and devices)

TCP Service Model

TCP may buffer at both sides; consequently

- Transmission may be delayed
- PUSH flag : force messages out (e.g. for telnet)
- URGENT flag
  - used when user hits Ctrl-C or DEL
  - receiving application is interrupted (using a signal in UNIX)

TCP Protocol

- Every byte has 32-bits sequence#
  - on 10 Mbps it takes about an hour to wrap around
- Data exchange in segments
  - segment contains 20 byte header, options, + data
  - must fit into MTU : maximum transfer unit of a network
- Sliding window protocol with timeout
  - receiver sends back ack# equal to next expected segment#
  - receiver uses piggyback
  - sender retransmits if timeout occurs

TCP Segment Header

Header is 20 bytes (+ possible options; see fig 6-24) with fields :

- Source and destination port # (16-bit)
- Sequence # (32-bit)
- Acknowledgement # (32-bit) : gives next byte expected
- Header length (4-bit)
  - how many 32-bit words in header (note variable sized header because of 'options' field)

TCP Segment Header

- Six 1-bit flags
  - Urgent : if 1, urgent pointer indicates where urgent data is found
  - ACK : if 1, ack field is valid
  - PUSH: request to receiver not to buffer data, but send directly to host application
  - RST : reset connection (e.g. if connection is refused)
  - SYN : establish connection (syn = 1, ack = 0)
    acknowledge connection (syn = 1, ack = 1)
  - FIN release connection (in 1-direction)
    after sending a FIN you may still receive data

TCP Segment Header

- Window size (16-bit)
  - used for flow control
  - gives # bytes of available receiver bufferspace
  - 16 bits is somewhat short !
    => support scaling (up to 2^{14})
  - Zero window field is allowed
- Checksum
  - checksums the whole segment including pseudo header (fig 6-25)
  - sum all 16-bit words in 1's complement (and take its complement)
    - if receiver sums these words incl. checksum it gets zero
    - including pseudo header helps detect packet misdelivery but violates protocol stack

TCP Header Segment

- Options
  - Specify max. payload of receiver
    - All Internet hosts are expected to handle at least 556 byte segments
  - Scaling window unit (up to 14 bits)
  - Selective repeat (using NAK: negative ack. TPDUs)

TCP Connection Management

TCP used three-way handshake to make connection (fig 6-26[a])

- server application executes LISTEN and ACCEPT primitives
  - with or without specifying a specific source (IP address and port#)
- client executes CONNECT with server address ⇒
  - TCP sends SYN segment (1 byte)
  - server TCP checks if some application is listening, if so, this process gets the SYN segment
- server application accepts connection and an *ack* is sent back, otherwise a *reject* is sent

TCP Connection Management

If 2 parties try to make connection (fig 6-26[b]) only one connection is made

Connection release:

- Full duplex connection = two simplex connections
  - a simplex connection is released by sending FIN flag
  - release connection if ack received or when time out occurs (= 2 x life time)
  - these two segments have to be sent for each simplex connection (so in total 4 segments)

TCP Connection Management

Lifetime IP packets 2 minutes

- Initial seq# not zero
  - clocking scheme used
  - clock tick every 4 µsec.
- After crash reboot may not finish before packet lifetime (= 120 sec)

Fig 6-27 shows TCP states; fig 6-28 its FSM

TCP Transmission Policy

Sender has window on the state of receiver buffer

- receiver controls this window (fig 6-29)
  - if window = 0 only urgent messages can be sent
- both sides (sender and receiver) may buffer segments before sending them / delivering them
  - this may increase throughput because communicating larger chunks of data is often benificial
  - do not use this for a telnet connection !

TCP Transmission Policy

Two problems:

1. ***Too much overhead***:
   Sender sends 1 byte at a time (eg. telnet connection)

- each character generates 162 bytes of segments (exclusive frame overhead)
  - 41 byte segment (TCP + IP headers + character)
  - 40 byte acknowledged
  - (+ window update + character echo)
- Solution : Nagle's algorithm
  - send first byte directly
  - wait (and buffer next bytes) till acknowledge comes in
  - Note: do not use this to transmit mouse movements

TCP Transmission Policy

2. Receiver application consumes 1 byte at a time (but sender wants to send large chunks)

   => ***silly window syndrome*** (fig 6-30)

   – for each byte it sends a window update (of 1 byte)

- Solution: wait with sending window updates
  – perform window updates in large sizes only
  – sender may also decide to send larger chunks only (up to the agreed segment size with the receiver)

TCP Congestion Control

Real solution to congestion control is to slow down the data rate of the sender

Steps:

- Detection: *assume that time outs are caused by congestion only* (works for 'clean' channels only)
- Two potential problems (see fig 6-31):
  – ***Network congestion***: keep congestion window
  – ***Receiver congestion***: use standard receiver window
- Sender keeps 2 windows: receiver window (RW) and congestion window (CW)

TCP Congestion Control

Sender is allowed to send minimum of two windows (in # of bytes)

send window = min (RW, CW)

Determine CW size (see fig 6-32) :

- TCP uses an additional threshold (initially 64 kbytes) and a 'slow start' algorithm
- If segment is acknowledged before 'congestion timer' goes off, increase CW

TCP Timer Management

TCP uses (conceptually) multiple timers:

- Retransmission timer
  – started when segment is sent
  – dynamically adjusted because of large variation in round trip time (fig 6-33)
- Persistence timer
  – used to remove deadlock caused by lost window update message of receiver
  – send probe if timer goes off
- Keep alive timer (optional)
  – check (when timer goes off) if other side is still alive
  – if not, finish the connection

TCP Timer Management

Let us look how *retransmission timeout* is tuned

Estimate round trip time: RTT

$RTT = \alpha\ RTT + (1-\alpha)\ M$
where M = **measured** round trip delay
$a \sim 7/8$

Estimate *mean deviation D* in RTT (as cheap replacement for the standard deviation):

TCP Timer Management

$D = \alpha\ D + (1 - \alpha)\ |RTT - M|$

Now choose:

Timeout = RTT + 4*D

- 4 times standard deviation misses 1 % ! (see fig. 6-33b)

TCP Timer Management

Problem: if after retransmission ack comes in, it is unclear if this ack belongs to original or retransmitted segment: so what is M?

Solution: *Karn's algorithm*:

- do not include retransmission, however,
- double timeout on each retransmission

UDP: User Data Protocol

UDP is just a way to encapsulate raw IP datagrams and send them *without having to establish a connection*

- used for client-server applications that need only one send and receive segment
- header: 2 words (8 bytes; fig. 6-34)
  - source and destination port numbers
  - length
  - checksum

TCP and wireless connections

Problems occur if part of the trajectory is wireless IP

When packets are lost:

- sender should *slow down* on wired network (congestion is a likely cause)
- sender should *try harder* on wireless network (corrupted packets, retransmit them)

=>Timers and parameters have to be tuned differently

TCP and wireless connections

Solutions:

1. Make separate TCP connections (fig 6-35)
   - not transparent
   - receipt of ACK does not mean that receiver got the message

or

2. Make (small) modifications to Network Layer within the intermediate base station
   - e.g. base station may retransmit a TCP segment to mobile without involving source

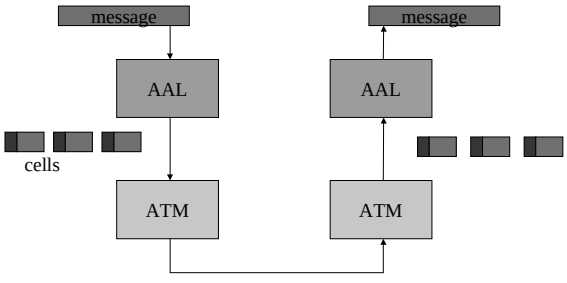
ATM

It is not clear whether ATM has a transport layer

- ATM layer is clearly a network layer
- AAL (ATM adaption layer) on top, could be interpreted as transport layer

- However, unlike TCP
  - not reliable
  - no error, flow and other control
- Main goal of AAL: transmit voice and video streams:
  - shield application from assembly / reassembly of cells

Segmentation and reassembly in ATM



ATM Adaption Layer protocols

ATM Adaption Layer (AAL) recognizes different applications

- real-time versus non real-time
- constant versus variable bit rate
- connection-oriented versus connectionless service

==> 8 services

Not all of them are supported

AAL defines 5 standards: AAL1 - AAL5

- Only AAL5 likely survives
- For comparison see fig. 6-45; read criticism on pg. 555

Exercises

Make from chapter 6 the following exercises:

- 1, 4, 5, 8, 9, 14, 15, 16, 20, 23