Transport Protocols

Class 10 Overview

- Transport Protocols
- Project TRN1: Reliable Transport

The ISO 7-layer model
Transport Layer

- Transport Layer:
  - end-to-end error and flow control
  - in Internet: also congestion control
  - protocol resides in source and destination nodes
  - provides consistent service interface to network
- must provide reliable connection despite:
  - connectionless network services
  - virtual circuit resets on connection-oriented network services
  - packet re-ordering that can occur when a Transport Connection is split across multiple VCs

Transport Protocol: end-to-end operation

Transport Layer vs. DLC layer

- TL and DLC bear resemblance:
  - both perform error control, re-sequencing and flow control
  - data link layer operates on a link basis
  - TL operates end-to-end across a network
Elements of Transport Protocols

- **Connection Management:**
  - establishment, refusal and release
  - manage mapping between transport and network connections or build connection-oriented service from connectionless network service
  - link applications across a network
- **Segmentation and reassembly of application data:**
  - fragment data stream into network suitable packet sizes
  - correctly associate packets with each of multiple transport connections (where these exist)
  - receiver must re-sequence and re-assemble to correct msg

Elements of Transport Protocols, cont’d

- **Recovery from network failures:**
  - re-assignment after network disconnects & re-synchronization after network resets
- **Error Control and Flow Control (esp. for LANs)**
- **Multiplexing:**
  - mapping multiple transport connections to one network connection

Elements of Transport Protocols, cont’d

- **Splitting & Recombining:**
  - mapping one transport connection to multiple network connections
Internet Transport Protocols

- **UDP: User Datagram Protocol**
  - a.k.a.: “unreliable data protocol”
  - used for ‘best-effort’ service

- **TCP: Transmission Control Protocol**
  - reliable transport

### UDP Message Format

```
0 4 8 16 24 31
```

- **UDP Source Port**
- **UDP Destination Port**
- **UDP Message Length**
- **UDP Checksum**
- **Zero, or header+data checksum**

### UDP Well-Known Ports

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Keyword</th>
<th>UNIX Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ECHO</td>
<td>echo</td>
<td>echo</td>
</tr>
<tr>
<td>7</td>
<td>DISCARD</td>
<td>discard</td>
<td>discard</td>
</tr>
<tr>
<td>11</td>
<td>USERS</td>
<td>sysstat</td>
<td>active users</td>
</tr>
<tr>
<td>13</td>
<td>DAYTIME</td>
<td>daytime</td>
<td>daytime</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>netstat</td>
<td>who is up or netstat</td>
</tr>
<tr>
<td>17</td>
<td>QUOTE</td>
<td>qotd</td>
<td>quote of the day</td>
</tr>
<tr>
<td>19</td>
<td>CHARGEN</td>
<td>chargen</td>
<td>random character generator</td>
</tr>
<tr>
<td>37</td>
<td>TIME</td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>42</td>
<td>NAME SERVER</td>
<td>nameserver</td>
<td>domain name server</td>
</tr>
<tr>
<td>43</td>
<td>NIC NAME</td>
<td>whois</td>
<td>who is</td>
</tr>
<tr>
<td>67</td>
<td>BOOTPS</td>
<td>bootps</td>
<td>bootstrap protocol server</td>
</tr>
<tr>
<td>68</td>
<td>BOOTP</td>
<td>bootpc</td>
<td>bootstrap protocol client</td>
</tr>
<tr>
<td>69</td>
<td>TFTP</td>
<td>ftp</td>
<td>trivial file transfer protocol</td>
</tr>
</tbody>
</table>
TCP

- TCP provides connection for connectionless IP
  - reliable data transfer (selective repeat or go-back-N)
  - stream model of data
  - maintains order in delivered stream
  - flow control via sliding window
  - multiplexing multiple sessions via "ports"
- Berkeley implementation:
  - socket = IP address + port number
  - FDX transmission (ACK piggy-backing in reverse stream)

TCP, cont’d

- a TCP connection is identified completely by:
  - <sending IP addr/port, receiving IP addr/port>
- precedence and security options
  - not widely used
- disconnect

TCP Sliding Window

Window of data to be transported

Very long stream of data

Segments filled in as received
Connection Setup

Problem: Delayed Duplicate Sequence Numbers

- packets may be delayed in the network and arrive out of order at destination
- this presents possibility that old packets from earlier transport connection may arrive and appear to be in sequence for some current transport connection

Connection Setup

Problem: Delayed Duplicate Sequence Numbers

- two mechanisms commonly used to guard against this:
  1. discard packets within the network after a certain time (or number of hops)
  2. prohibit sequence number re-use for a certain time
     (in this case, sequence number is a combination of a connection identifier and a packet identifier within the connection; thus, really the connection identifiers must be recycled in a time-sensitive manner)

TL Connection Establishment: 3-way Handshake

- 3-way handshake used to avoid problems that can arise from delayed-duplicate packets in connection setup:
  SOURCE
  ConReq(to_cid=X)
  ConInd(to_cid=X, from_cid=Y)
  Data(to_cid=X, from_cid=Y)
  (Could be Ack if no data to send)

  DESTINATION
  ConReq(from_cid=X)
  ConInd(from_cid=Y)
  ConRes(to_cid=X, from_cid=Y)
  Data(from_cid=X, to_cid=Y)
**TL Connection Release**

3-way handshake

- To prevent problems with old duplicates of DISC messages, a 3-way handshake is also used for connection release:

```
DisReq (from cid= X, to cid= Y) & Start Timer
DisConf (to cid= X, from cid= Y)
DisAck (from cid= X, to cid= Y) & Delete Connection
```

**TCP Connection State Machine**
(from Comer)

```
Closed
passive open
anything / reset
Listen
SYN Recvd
close
Close
Wait
SYN Sent
Established
ClosingFIN WAIT-1
FIN WAIT-2 Timed
WAIT Last
close / FIN
ACK / timeout after 2 segment lifetimes
FIN / ACK
FIN+ACK / ACK
FIN / ACK
close / FIN
ACK / timeout / reset
SYN / SYN+ACK
reset
begin
SYN / SYN+ACK
```

**TCP Segment Format**

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4</td>
<td>Source Port</td>
</tr>
<tr>
<td>4-10</td>
<td>Sequence Number</td>
</tr>
<tr>
<td>10-16</td>
<td>Acknowledgment Number</td>
</tr>
<tr>
<td>16-24</td>
<td>Data (if any)</td>
</tr>
<tr>
<td>24-31</td>
<td>Padding</td>
</tr>
</tbody>
</table>
```

©J Mark Pullen, 20017
TCP Well-Known Ports

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Keyword</th>
<th>UNIX keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>RJE</td>
<td></td>
<td>remote job entry</td>
</tr>
<tr>
<td>11</td>
<td>USR</td>
<td>systat</td>
<td>active users</td>
</tr>
<tr>
<td>13</td>
<td>DAYTIME</td>
<td>daytime</td>
<td>daytime</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>netstat</td>
<td>network status program</td>
</tr>
<tr>
<td>17</td>
<td>QUOTE</td>
<td>qotd</td>
<td>quote of the day</td>
</tr>
<tr>
<td>20</td>
<td>FTP DATA</td>
<td>ftp-data</td>
<td>file transfer protocol (data)</td>
</tr>
<tr>
<td>21</td>
<td>FTP</td>
<td>ftp</td>
<td>file transfer protocol</td>
</tr>
<tr>
<td>23</td>
<td>TELNET</td>
<td>telnet</td>
<td>network terminal emulator</td>
</tr>
<tr>
<td>25</td>
<td>SMTP</td>
<td>smtp</td>
<td>simple mail transfer protocol</td>
</tr>
<tr>
<td>37</td>
<td>TIME</td>
<td></td>
<td>time</td>
</tr>
<tr>
<td>42</td>
<td>NAMENSERV</td>
<td>nameserver</td>
<td>host name server</td>
</tr>
<tr>
<td>43</td>
<td>NICNAME</td>
<td>whois</td>
<td>who is</td>
</tr>
<tr>
<td>53</td>
<td>DOMAIN</td>
<td>nameserver</td>
<td>domain name server</td>
</tr>
<tr>
<td>80</td>
<td>HTTP</td>
<td>http</td>
<td>world wide web http</td>
</tr>
</tbody>
</table>

TCP Operation

- Code bits: used to originate connection
- Window: size varied for congestion control (reduce multiplicatively, increase additively)
- Round-Trip time (RTT): TCP keeps track of time from SEND to ACK
- Timeout: if some multiple of RTT passes without ACK, SEND again
- establish connection:
  - requires 3-way handshake (one end must perform "passive open")
- push: issued by sending app to cause TCP to send a segment (important for interactive apps)

TCP Startup

- port selection:
  - app has pre-established "meeting" port
  - if multiple sessions are possible for the app, can use initial connection to agree on new "working" port
- slow-start:
  - to avoid congestion
  - start with minimum window
  - double window size each round-trip
  - if no congestion seen (increase in RTT), continue doubling until configuration max reached
TCP Timer Management

- multiple timers are needed to manage TCP's complexity:
  1. retransmission timer
  2. persistence timer
     - used to prevent deadlock if window size is 0 and receiver update is lost
  3. keep-alive timer
  4. timer for TIMED WAIT state

TCP Retransmission Timer

- used to decide when to re-transmit segment
- problem: hard to determine correct time-out value for WAN:
  - if timeout is too short, unnecessary re-xmits occur, clogging network
  - if timeout is too long, performance degraded due to long re-xmit delays
- solution: use highly dynamic algorithm that constantly adjusts timeout interval

TCP Timeout Algorithm

- First step: maintain a variable, rtt, representing best current RTT estimate.
  
  Suppose an ACK returns in time M, M < timeout; then set:
  
  \[ rtt = \alpha rtt + (1 - \alpha)M, \alpha < 1 \]
TCP Timeout Algorithm, cont’d

- Next, establish deviation estimator $D$ to account for large variability:
  $$D = \alpha D + (1 - \alpha) |rtt - M|$$
- Finally, set timeout as:
  $$timeout = rtt + 4D$$

TCP Timeout Algorithm, cont’d

- Problem: when an ACK arrives, unclear whether is for the first transmission or the later one
- Solution (Karn’s algorithm): do not update $rtt$ on any segments that have been re-transmitted; instead, double the timeout

TCP for Interactive Applications

- interactive apps, e.g., telnet, may need to send as little as one byte of data in a segment
- push feature allows app to tell TCP to send data without waiting to accumulate full segment
- when network response is slow, sending one byte per segment would be inefficient; use Nagle’s algorithm to avoid this:
  - while waiting for ACK, accumulate send-data in a buffer
  - when ACK arrives, send contents of buffer
TCP/IP: Some Observations

- X.25 deals with reliability at all layers; TCP/IP deals with it only at transport layer.
- OSI has strict boundary between layers 3 and 4; some TCP/IP implementations get better results by ‘cheating’ and sharing data between layers 3 and 4.
- TCP/IP software normally resides in host operating system (e.g., UNIX kernel).

OSI Transport Layer Design Issues

- Main question for transport layer protocol designer: “what can be assumed about the network service?”

<table>
<thead>
<tr>
<th>Network Service Class</th>
<th>Description From OSI Transport Layer Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Error free service without network resets.</td>
</tr>
<tr>
<td>B</td>
<td>Perfect packet delivery with network resets.</td>
</tr>
<tr>
<td>C</td>
<td>Unreliable service (lost and duplicated packets possible), and network resets.</td>
</tr>
</tbody>
</table>

OSI Transport Layer Answer: 5 Transport Classes

- Class 0 & 2:
  - Assume Type A network service, but class 2 adds ability to multiplex transport connections onto single network connection.
- Class 1 & 3:
  - Assume Type B network service, but class 3 adds ability to multiplex transport connections onto single network connection.
OSI Transport Layer Answer:
5 Transport Classes

- Class 4:
  - assume Type C network service
  - includes ability to multiplex multiple connections onto single network connection
  - also includes ability to split single transport connection across multiple network connections
- TCP:
  - similar to class 4 OSI Transport protocol, but has differences in details

OSI Transport Class 4 vs. TCP

- TCP checksums computed with 16-bit words; TP4 checksum computed with 8-bit words
- TCP acknowledges bytes; class 4 acknowledges transport packets
- TCP operates only over connectionless network service
- TCP includes functions for session synchronization, and graceful connection close; OSI reserves these functions to Session Layer

Footnote: IP/OSI Interoperation
Network Address Translation

- As presented in last class, IPv4 is running out of addresses and ISPs are slow to make major change to IPv6
- A work-around in common use today is **network address translation (NAT)**:
  - Use one IPv4 address to ‘front’ for many hosts
  - Assign hosts addresses from ‘private’ address space
    - E.g., class A net 10 (ARPANET) set aside for testing
  - Run a process that uses TCP port multiplexing to identify ‘back side’ host address

NAT Operation

- PortMappingTable:
  - Port:real_src:real_port:dest_addr:dest_port

Outside world, i.e., the Internet, only sees IP addr’s of NAT gateway.

NAT Operation, cont’d

- Port map table entries are allocated on-demand for requests originating ‘inside’ NAT space
- Table entries have time-outs
  - Reset each time they are used
  - Typically 2 – 3 minutes
- Packets arriving with dest_port not found in table are dropped (firewall)
Why NAT is ‘evil’…

- involves re-writing of IP headers
  - hence also re-computation of checksums
- some applications are ‘NAT sensitive’
  - use of NAT breaks them, e.g., ftp
  - propose use of ‘application level gateways’ to rewrite parts of messages as well as headers
- threatens:
  - end-to-end security
  - end-to-end functions


Class 10 Overview

- Transport Protocols
- Project TRN1: Reliable Transport

Project TRN1: Reliable Transport

- simplified from TCP
  - only one connection open at a time
  - hdx xmit
  - very short segments (20 bytes)
  - no DLC ARQ
- 4 send states: closed, syn_sent, sending, close_wait
- study tl.cpp pseudo-code comments, then program send_rtl_segments in stub tl_send.cpp
Project TRN1: Reliable Transport

- might want to change profile for debugging:
  - print_at[] to see details of network operation
  - interactive=TRUE so output comes a few lines at a time
- successful run will pass all email messages