Lecture Outline

- Introduction to end-to-end protocols
- UDP
- RTP
- TCP
- Programming details
End-To-End Protocols

- Enable communication between 2 or more processes (which may be on different hosts in different networks)

- The Transport Layer is the lowest Layer in the network stack that is an end-to-end protocol
Transport Layer Protocols

- Connectionless protocols considered here

- Basic Function:
  - Enable process-to-process communication via virtual process-hooks called *ports*.

  4-Tuple Connection Identifier:
  
  \(<\text{SrcPort}, \text{SrcIPAddr}, \text{DestPort}, \text{DestIPAddr}>\)

- A transport protocol may provide several features in addition.
Most popular transport protocols

- **User Datagram Protocol (UDP):**
  - Provides the process identification functionality via ports
  - Option to check messages for correctness with CRC check

- **Transmission Control Protocol (TCP):**
  - Ensures reliable delivery of packets between source and destination processes
  - Ensures in-order delivery of packets to destination process
  - Other options

- **Real Time Protocol (RTP):**
  - Serves real-time multimedia applications
  - Header contains sequence number, timestamp, marker bit etc
  - Runs over UDP
User Datagram Protocol (UDP)

### Header Fields:

- **Src Port**: Unique identification number assigned to the source process by the kernel in the source node.
- **Dest Port**: The Unique Identification number assigned to the destination process by the kernel in the destination node.
- **Checksum**: Filled on source side. Checked on receiver side to ensure message correctness. Calculated over <Data, UDP hdr, portion of IP hdr>
- **Length**: Total number of bytes in (UDP header + data bytes)

---

**UDP Header**

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src Port</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Dest Port</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Checksum</td>
<td>2 bytes</td>
</tr>
<tr>
<td>Length</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>
Example of an application using UDP

- My application called Network Performance Monitor (NPM) needs to measure the pattern of packet losses in a network.
- Application needs sequence numbers and timestamps in each packet
- UDP does not provide this facility; So NPM adds its own header to each packet
Application requirements

- Like NPM, most applications need much more from a transport protocol than the basic functionality.
- Multimedia applications require tracking of packet loss, delay and jitter.
- Most other applications such as HTTP, Database Management, FTP etc, require reliable data transport.

- TCP, UDP and RTP satisfy needs of the most common applications.
- Applications requiring other functionality usually use UDP for transport protocol, and implement additional features as part of the application.
Introduction to TCP

- First proposed by Vinton Cerf and Robert Kahn, 1974 (They were awarded the ACM Turing award 2004)

- The TCP/IP protocol suite has enabled computers of all sizes, from different vendors, different OSs, to communicate with each other.

- Used by 80% of all traffic on the Internet
A simple File Transfer Application

- Receiver process waits for connection and data from sender
- Sender process requests receiver for a connection
- Once the two processes are “connected”, the sender process transfers the file and closes.

- Receiver should be started first
- Receiver port Id should be known to the sender
Programming Viewpoint

Data Sending Application

Send bytestream to connected Socket of type SOCK_STREAM

TCP

User Space
Handles Application details

Kernel Space
Handles communication details

- Sender application process only needs to provide a bytestream to the kernel
- Kernels on sending and receiving hosts operate TCP processes
- Receiver application process only needs to read received bytes from the assigned TCP buffers
A top-level view of TCP operation

4-Tuple Connection Identifier:
\(<\text{SrcPort, SrcIPAddr, DestPort, DestIPAddr}>\)
Summary of TCP’s Operation Sequence

- All Operations are sender driven; TCP protocol completely implemented at the ends
- Sequence numbers maintained in bytes (remember, TCP serves a byte stream)

Start of operation:
- Connection Establishment by a Three-Way Handshake algorithm
- Consensus on Initial Sequence Number (ISN)

Data Transfer:
- Sends the data in packets, reliably and as fast as the network/receiver permits

Finish:
- Connection tear-down by a Three-Way Handshake algorithm
- Both sides independently close their half of the connection
TCP Header Format

Flags: SYN FIN RESET PUSH URG ACK

At least 20 bytes

TCP segment

TCP Header

<table>
<thead>
<tr>
<th>0</th>
<th>15</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Destination port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledgement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hdr len</td>
<td>Flags</td>
<td>Advertised window</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data

Application

Byte Stream

TCP
Three-Way Handshake Algorithm

- SYN and ACK flags in the header used
- Initial Sequence numbers x and y selected at random
- Required to avoid same number for previous incarnation on the same connection

Note: in NS2 simulator, sequence numbers are simplified to packet-based numbering. The Ack indicates the last packet in the fully received sequence (x in the first Ack above)
Connection Establishment

---

Connection Establishment

---
Connection Tear-down

- Each side closes its half of the connection independently.

Sender

FIN

FIN-ACK

Receiver

Data write

Data ack

FIN

FIN-ACK
Observe that a connection in the TIME_WAIT state can move to CLOSED state only after waiting for $2^{\text{Max-TTL}}$
TCP data transfer broadly explained

Goal of TCP: Deliver data reliably and in order as fast as possible
(Throughput = bytes delivered/ time taken)

- **Flow Control:**
  - Increase the sending rate to use the network (the route) to full capacity or receiver capability
  - But scale back if congestion occurs or if receiver is flooded

- **Error Control or Congestion Control:**
  - When packets are lost, it implies that the one or more queues in intermediate routers have overflowed.
    - Retransmit lost packets
    - Scale back flow rate to reduce congestion

- Flow control and error control are intertwined using the congestion window (cwnd)
The Congestion Window ($cwnd$) is TCP’s main tool of operation.

- A sliding window used for both Flow control and Error Control.

Error Control:
- $cwnd$ maintains all packets not-yet acknowledged.

Flow control:
- Size of $cwnd$ is the highest burst size that can be sent at one time.
- Higher the $cwnd$, more the number of packets “in the pipe”.
**Cwnd (sliding window) Operation**

Receive (ACK n+k) => Receiver has received all bytes upto (not incl.) n+k

NEW congestion window

- All bytes ACKed
- Sent but not ACKed
- New bytes sent
- Not yet sent

depends on mode of operation
Conceptual sender-side TCP

TCP operation is paced by its ACKs

Wait for ACK

Duplicate ACK arrives

ACKs indicate possible losses

If 1st or 2nd duplicate ACK, Do limited_retransmit()
If 3rd duplicate ACK, Do congestion_control()

New ACK arrives

ACKs indicate lossless operation

(operate in slow-start or congestion avoidance)
Measure RTT if applicable
Set Cwnd_start = purge_acked_pkts()
Set Cwnd = cwnd + increment_value()
Send all the new packets
restart timer if applicable

No ACKs

Timer triggers;

No ACK for retransmission;
Timer triggers

set ssthresh = cwnd/2 ; Set cwnd = 1;
Retransmit cwnd_start packet

Timeout

New ACK arrives

No ACKS

TCP operation is paced by its ACKs
Sender’s modes of operation (when no losses)

- **Slow-start mode:**
  - $cwnd$ growth in this mode when
    - $cwnd$ size < slow-start-threshold AND
    - No losses are detected
  - $cwnd$ increases by a segment with every incoming ACK
  - Exponential increase
  - $cwnd$ incremented by the number of ACKs received in one round-trip-time

- **congestion-avoidance mode**
  - $cwnd$ growth in this mode in all other cases
  - $cwnd$ incremented by $(1/cwnd) \times \text{number of bytes acked}$ with each incoming ACK
  - Additive increase
  - $cwnd$ incremented by at most one segment in each round-trip-time
Visualization of slow-start and congestion avoidance

Assumes:
• ssthresh = 16
• All segments are ACKed and there are no packet losses

Courtesy: TCP/IP Illustrated, Vol 1 by W.R.Stevens
Receiver-side flow control

- Avoid flooding receiver with data
  - Notifies sender of number of bytes it can accept in *advertisedWindow* field in ACK header.
- Sender bytes sent = $\min(cwnd, \text{advertisedWindow})$
- Receiver delivers bytes in correct order to application process by maintaining a receive window

![Diagram of receive buffer with acknowledged and unacknowledged segments]
TCP’s Error Control Mechanism

- Data segments and ACKs may get lost in transit; Losses interpreted as due to network congestion (i.e. buffer overflow in an intermediate router)
- TCP sender sets deadlines for ACK arrival using timers;
  - Deadlines a function of estimated RTT
- If deadlines not met:
  - cwnd scaled down
  - Segment(s) retransmitted
- Accurate Round-trip Time estimation critical for efficient TCP operation
- Premature timeouts and retransmissions place huge toll on the net throughput
The Reliability Mechanism

- Receiver generates ACKs each time a segment is received
- ACKs are cumulative
Round-trip time (RTT) Estimation

- Round-trip time is variable. Smoothed RTT estimator:
  - \[ R = \alpha R + (1- \alpha)M \]
    - \( R \): smoothed RTT
    - \( M \): new RTT measurement
    - \( \alpha \): smoothing factor (typically = 0.9)

- A single RTT estimator active at a time. Cumulative ACKs also considered

- Karn’s Algorithm: Retransmitted segments not considered for RTT estimation because of retransmission ambiguity problem
Timeout and Retransmission

- RTO Timer Expiry:
  - No ACKs arrive at all before RTO timer expires
  - TCP interprets this as heavily congested network
  - *Exponential Backoff* triggered when no segment is transmitted and network is given time to recover from congestion
  - After the backoff duration, the first unacknowledged segment is retransmitted subsequent resumption of data flow only after Backoff duration

- Expiration of either timer sets $cwnd=1$, and *slow-start* mode
Timeout impact on Throughput

- A timeout reduces $cwnd$ size to 1 => Just 1 segment transmitted in 1 RTT.
- $cwnd$ subsequently grows very cautiously in slow-start mode
- Bad for lossy high bandwidth-delay paths
- All segments following the lost segment are also retransmitted: even if they have been successfully received (out of order) at the receiver
- A possible bulk retransmission of a large portion, may further contribute to network congestion
Ideal behavior of TCP Tahoe

At t2, t3, t4:
• Duplicate ACKs arrive
• Retransmission timer expires
• Single packet retransmitted in slow-start mode at cwnd=1
• Next segments sent based on cumulative ACKs received
Optimizations in various TCP flavors

- Several optimizations for better TCP throughput in the past 30 years.
- Most important among them:
  - Fast Retransmit
  - Fast Recovery
  - Selective Acknowledgement (SACK)
  - Delayed ACKs
Fast Retransmit

- Don’t wait for retransmission timer to expire that causes $cwnd$ to drop to 1
- React to duplicate ACKs instead
  - Don’t know if duplicate ACKs are because of packet loss or reordering. Threshold set to 3 duplicate ACKs
  - On receiving 3 duplicate ACKs:
    - Requested segment retransmitted $cwnd$ growth continues
    - Set $ssthresh = (\frac{1}{2} \times \text{MIN}(cwnd, \text{rcvr\_adv\_window}))$ bytes
    - Set $cwnd = (ssthresh + \text{num-dup-ACKs} \times \text{segment\_size})$ bytes
  - $cwnd$ continues to grow with arriving dup-ACKs
    - Each duplicate ACK implies that a segment has left the network and reached the receiver
  - New segment transmitted if $cwnd$ size permits
Fast Recovery

- When ACK for retransmission received \( cwnd \) growth resumes in congestion avoidance mode with \( cwnd = ssthresh \) rather than starting in slow-start mode with \( cwnd = 1 \)

TCP Reno

- Most popular TCP flavor; implemented in most operating systems
- Includes Fast Retransmit and Fast Recovery

From paper: S. Gopal and S. Paul “TCP Dynamics in 802.11 wireless local area networks”, ICC 2007
TCP NewReno

- Enhancement to Reno’s Fast recovery; so as to handle multiple losses in a congestion window

From paper: S. Gopal and S. Paul “TCP Dynamics in 802.11 wireless local area networks”, ICC 2007
Programming viewpoint

Sender side

Open a TCP flow

assign socket

socket(SOCK_STREAM,...)

int TCPsock = returned socket id

details.set_local_port

details.set_local_IPAddr

details.set_dest_port

details.set_dest_IPAddr

connect(TCPsock, .......)

connected socket

bind(TCPsock, details,...)

bound

Fill Buffer with bytes to send

send(bytes, Buffer)

Kernel handles all data transfer procedures of TCP

Fill Buffer with more bytes to send

send(bytes, Buffer)

return success/failure

Done sending all data; indicate finish to kernel

close(s)

Kernel initiates connection teardown

Establish connection with destination

Kernel processes

Sumathi Gopal, Comm Nets II, ECE Dept
Rutgers University
Programming viewpoint

Receiver side

Open TCP Socket

int TCPsock = returned socket id
TCPsock.set_local_port
TCPsock.set_local_IPAddr

Indicate to kernel you're expecting a connection on this socket

Now wait for a connection request

accept(TCPsock, ...)

A connection request received

Now wait and read bytes when available

recv(TCPsock, Buffer)

Kernel maintains TCP recv process; Receive data for connected socket
Store in SOCKBUF; fill Buffer and notify user process

Save received bytes; read more data

recv(TCPsock, Buffer)

Close connection request by sender

Interpret as data finished from sender

return numBytes

close(s)

Kernel handles connection teardown

RECVING

USER PROCESS

KERNEL

bind(TCPsock, details, ...)

bound

listen(TCPsock, ...)

OK

return -1

Sumathi Gopal, Comm Nets II, ECE Dept
Rutgers University
TCP is not the ideal for all applications

- TCP optimized for wired networks
- Performance is poor in wireless networks
- Applications with stringent delay requirements do not use TCP, because of possible unbounded delays
Real-Time Protocol

- Quality of Service (QoS) factors: Reliability, Delay and Jitter
- Because of possibly unbounded retransmissions in TCP, large delay and jitter may ensue.
- Applications prefer UDP instead.
- RTP protocol operates over UDP, and with header containing
  - timestamp
  - sequence number
  - A marker bit
  - Packet concatenation etc
- RTP provides no other correction strategies like in TCP; Applications handle all aspects themselves.
- RTP modules run in user-space. RTP libraries included in the application.
Summary

• Numerous transport protocols proposed
• TCP sustained because of its distributed nature and because of the TCP/IP protocol suite that enabled computer systems to connect across boundaries

• Ample scope exists for new transport protocols given proliferation of heterogeneous networks and devices
Homework

- 5.16
- 5.13
- 5.28
- 5.34