Transport Layer and UDP

To do ...

- Transport Layer – The Basics
- The Simplest Transport Layer Protocol - UDP
A transport-layer protocol provides for *logical* communication between app processes running in different hosts

- From the app’s perspective, as if the nodes were directly connected

Implemented at end hosts, not in routers

- Converts app-layer packets into segments by
  - Possible breaking messages into smaller chunks
  - Adding some headers
- Segments are given to the network layer ...
Transport and Network Layer

- Transport-layer protocols – logical communication between processes running in different hosts

- Network-layer … between hosts
  - Subtle but important difference

- Clearly, the services a transport-layer protocol can provide is constrained by what the underlying network-layer service offers
  - No delay or bandwidth guarantees from network? …

- Constrained is not the same as completely determined …
  - E.g., A reliable transport on an unreliable network
Transport Layer in the Internet

- Internet transport-layer protocols – UDP and TCP
- What do they get from IP, the network-layer protocol?
  - An unreliable, best-effort service
    - i.e., best effort = no guaranteed segment delivery
    - No guaranteed integrity of data in the segments
- What do they offer
  - Extending host-to-host delivery to process-to-process – transport-layer multiplexing/demultiplexing
  - Integrity checking with error-detection fields in the header
  - Reliable data transfer, using flow control, seq #, acks, timers
  - Congestion control (for the general good) ensuring every connection in a congested link gets an equal share of bw
Transport Layer in the Internet

- **UDP – User Datagram Protocol**
  - Unreliable, connectionless service
  - Extending IP service “between hosts” to “between processes” – transport-layer multiplexing
  - Basic error checking
  - No setup costs, no transmission delays above IP

- **TCP – Transmission Control Protocol**
  - Reliable, connection-oriented service
  - Transport-layer multiplexing + basic error checking
  - Besides API provides abstraction of a stream of bytes, hiding
    - Message sizes, lost messages, duplication and ordering, flow control and congestion avoidance
Multiplexing/Demultiplexing

- Multiple processes, one connection (let’s imagine)
  - How can you tell which process each segment belongs to?
- Transport layer in the receiving host delivers data to a socket
  - Each socket has a unique identifier
  - Each segment has fields to identify the receiving socket – a destination port number → demultiplexing
- On the other end, gather data chunks from different sockets, encapsulate them into segments with header info, and pass them to the network layer → multiplexing
Connectionless Multiplexing

- To create a UDP socket
  
  ```python
  clientSocket = socket(AF_INET, SOCK_DGRAM)
  ```
  
  - Get back a socket # (between 1024 and 65535, currently not in use)
    - or we can associate it with a specific port # with bind
      
      ```python
      clientSocket.bind(('', 19157))
      ```

- On the client side, let the transport layer assign a port #

- A UDP port is fully identified by a two-tuple consisting of a destination IP address and a destination port #
  
  - The source address acts as a return address
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(), clientAddress)

from socket import *
serverName = 'localhost'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('Input lowercase sentence: ')
clientSocket.sendto(message.encode(), (serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print("From Server:", modifiedMessage.decode())
clientSocket.close()
Connection-oriented Multiplexing

- A TCP socket is identified by a four-tuple `<src IP, src port, dst IP, dst port>`
  - Demultiplexing happens based on the four values
- Server has a ‘welcoming socket’ to wait for connection-establishment requests from clients

```python
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
```

- When it receives a request, it creates a new socket for that client

```python
while True:
    connectionSocket, addr = serverSocket.accept()
```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
print("The server is ready to receive")
while True:
    connectionSocket, addr = serverSocket.accept()
    message = connectionSocket.recv(1024).decode()
    modifiedMessage = message.upper()
    connectionSocket.send(modifiedMessage.encode())
    connectionSocket.close()

from socket import *
serverName = 'localhost'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
message = input('Input lowercase sentence: ')
clientSocket.connect((serverName, serverPort))
clientSocket.send(message.encode())
modifiedMessage = clientSocket.recv(1024)
print('From server: ', modifiedMessage.decode())
clientSocket.close()
from socket import *
sservername = 'localhost'
sserverPort = 12000
cclientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('Input lowercase sentence: ')
cclientSocket.sendto(message.encode(), (servername, sserverPort))
modifiedMessage, serverAddress = cclientSocket.recvfrom(2048)
print("From Server: ", modifiedMessage.decode())
cclientSocket.close()

from socket import *
sservername = 'localhost'
sserverPort = 12000
cclientSocket = socket(AF_INET, SOCK_STREAM)
message = input('Input lowercase sentence: ')
cclientSocket.connect((servername, sserverPort))
cclientSocket.send(message.encode())
modifiedMessage = cclientSocket.recv(1024)
print('From server: ', modifiedMessage.decode())
cclientSocket.close()
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(), clientAddress)

from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
print("The server is ready to receive")
while True:
    connectionSocket, addr = serverSocket.accept()
    message = connectionSocket.recv(1024).decode()
    modifiedMessage = message.upper()
    connectionSocket.send(modifiedMessage.encode())
    connectionSocket.close()
Demultiplexing – Two Clients Connected to a Web Server

Web client host A

<table>
<thead>
<tr>
<th>Src port: 7532</th>
<th>Dst port: 80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src IP: C</td>
<td>Dst IP: B</td>
</tr>
</tbody>
</table>

... 

<table>
<thead>
<tr>
<th>Src port: 26145</th>
<th>Dst port: 80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src IP: C</td>
<td>Dst IP: B</td>
</tr>
</tbody>
</table>

... 

Web server host B

Per-connection HTTP process

Transport layer demultiplexing

Web client host C

<table>
<thead>
<tr>
<th>Src port: 26145</th>
<th>Dst port: 80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src IP: A</td>
<td>Dst IP: B</td>
</tr>
</tbody>
</table>

...
Connectionless Transport – UDP

• User Datagram Protocol [RFC 768] – As little as possible from a transport-level protocol
  – Nearly the same as IP
  – Just allowing multiple applications on a host to share one network
  – Plus some error checking

• Multiplexing/demultiplexing …
  – Takes application msgs, attaches src/dst port numbers for multiplexing
  – Pass the resulting segment to the network layer
  – No handshaking between src/dst – connectionless
  – On arrival, use dst port to deliver the segment to the correct app

• One example application – DNS
Why Would You Want to Use UDP? *Why not TCP?*!

- **Finer app-level control over what data is sent and when** – UDP just passes on whatever the app gives it; TCP has a congestion control mechanism that throttles the sender, a potential problem for real-time applications.

- **No connection establishment** – TCP three-way handshake introduces delay in setting a connection.

- **No connection state** – TCP keeps connection state that includes receive and send buffers, congestion control parameters and sequence and ack number parameters → constrains scalability.

- **Small packet header overhead** – TCP segment has 20B of header in every segment, vs 8B in UDP.
How do Processes Learn Each Others’ Ports?

- Client initiates the exchange so, just include that and the server will learn the client’s port

- How does the client learn of the server’s port?
  - A common approach – a well-known port, e.g., DNS in port 53 (in a Unix machine, look at /etc/services)
  - Another option is a “port mapper”, with a well-known port answering questions like ‘what port should I use to reach x?’
UDP Segment Structure

- Not much needed, four fields each of two bytes
- **Source** and **destination ports** as discussed
- **Length** – number of bytes in header+data
  Needed since the size of data field may be different from one segment to the next
- **Checksum** to check if the segment has been altered in transfer
Checksum is a simple way to detect data corruption

- Break the data into sequence of 16-bit integers
- Do the 1s complement of the sum
  - Add the integers
  - Wrap the carry-out bits to the least-significant position
  - Finally, invert the result (0 to 1, 1 to 0)

\[
\begin{align*}
0110011001100000 &\quad A \\
0101010101010101 &\quad B \\
1000111100001100 &\quad C
\end{align*}
\]

\[
\begin{align*}
0110011001100000 &\quad A \\
0101010101010101 &\quad B \\
1011101110110101 &\quad A+B
\end{align*}
\]

\[
\begin{align*}
0110011001100000 &\quad A \\
0101010101010101 &\quad B \\
1011101110110101 &\quad A+B
\end{align*}
\]
Checksum is a simple way to detect data corruption

- Why do you need error detection here?
  - Many link-layer protocols have it already
  - Yeah, but not all
  - And the error could be introduced when the segment is stored in a router’s memory

- An example of the end-to-end principle*

- Notice there’s no “recovery” by checksum, just detection
  - Discard the bad segment or pass it on with a warning

---

* J. Saltzer, D. Reed and D. Clark, End-to-End Arguments In System Design, ACM TOCS, 2(4), 1984
Recap

- Providing comm. services to applications – the transport layer
- At least, multiplexing/demultiplexing for communication processes – this + some checking = UDP
- A good basis for reliable data transfer and TCP …