Last week on Communication Networks

- BGP Policies
  - Follow the Money
- Protocol
  - How does it work?
- Problems
  - security, performance, ...

The Internet topology is shaped according to business relationships

There are 2 main business relationships today:
- customer/provider
- peer/peer
  - many less important ones (siblings, backups, ...)

Materials inspired from Scott Shenker & Jennifer Rexford
Customers pay providers to get Internet connectivity

Peers don’t pay each other for connectivity, they do it out of common interest

DT and ATT exchange tons of traffic. They save money by directly connecting to each other.

Business relationships conditions
route selection

For a destination \( p \), prefer routes coming from
- customers over
- peers over
- providers

Route exportation

For a destination, routes
- from customers are propagated to everyone else
- from peers and providers are only propagated to customers

Follow the Money
BGP Policies

Border Gateway Protocol
policies and more

BGP sessions come in two flavors

2
Problems
security, performance, …
external BGP (eBGP) sessions connect border routers in different ASes

iBGP sessions are used to disseminate externally-learned routes internally

BGP UPDATEs carry an IP prefix together with a set of attributes

- **IP prefix**
- **Attributes**

  - **Local-Pref**: outbound traffic control
  - **MED**: inbound traffic control
  - **AS-PATH**: loop avoidance
  - **Next-Hop**: egress point identification

Prefer routes...

- with higher Local-Pref
- with shorter AS-PATH length
- with lower MED
- learned via eBGP instead of iBGP
- with lower IGP metric to the next-hop
- with smaller egress IP address (tie-break)

Border Gateway Protocol (BGP) policies and more

- **BGP Policies**
  - Follow the Money
- **Protocol**
  - How does it work?
- **Problems**
  - Security, performance, ...

Follow the Money

We’re continuing our journey up the layers, now looking at the transport layer

This week on Communication Networks

This week on Communication Networks
What do we need in the Transport layer?

Functionality implemented in network
- Keep minimal (easy to build, broadly applicable)

Functionality implemented in the application
- Keep minimal (easy to write)
- Restricted to application-specific functionality

Functionality implemented in the “network stack”
- The shared networking code on the host
- This relieves burden from both app and network
- The transport layer is a key component here

UDP: Datagram messaging service
- No-frills extension of “best-effort” IP
- UDP provides only two services to the App layer
  - Multiplexing/Demultiplexing among processes
  - Discarding corrupted packets (optional)

TCP: Reliable, in-order delivery
- TCP provides a connection-oriented, reliable, bytestream transport service
- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down

Connections (or sessions)
- Reliability requires keeping state
  - Sender: packets sent but not ACKed, and related timers
  - Receiver: noncontiguous packets
- Each bytestream is called a connection or session
  - Each with their own connection state
  - State is in hosts, not network!

What Problems Should Be Solved Here?

Data delivering, to the correct application
- IP just points towards next protocol
- Transport needs to demultiplex incoming data (ports)

Files or bytestreams abstractions for the applications
- Network deals with packets
- Transport layer needs to translate between them

Reliable transfer (if needed)
- Not overloading the receiver
- Not overloading the network

What Is Needed to Address These?

Demultiplexing: identifier for application process
- Going from host-to-host (IP) to process-to-process

Translating between bytestreams and packets:
- Do segmentation and reassembly
- Reliability: ACKs and all that stuff
- Corruption: Checksum

Not overloading receiver: “Flow Control”
- Limit data in receiver’s buffer

Not overloading network: “Congestion Control”

What transport protocols do not provide

Delay and/or bandwidth guarantees
- This cannot be offered by transport
- Requires support at IP level (and let’s not go there)

Sessions that survive change-of-IP-address
- This is an artifact of current implementations
- As we shall see....
Important Context: Sockets and Ports

Sockets: an operating system abstraction

Ports: a networking abstraction
- This is not a port on a switch (which is an interface)
- Think of it as a logical interface on a host

Sockets

A socket is a software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system
- socketID = socket(…, socket.TYPE)
- socketID.sendto(message, …)
- socketID.recvfrom(…)

Two important types of sockets
- UDP socket: TYPE is SOCK_DGRAM
- TCP socket: TYPE is SOCK_STREAM

Ports

Problem: which app (socket) gets which packets

Solution: port as transport layer identifier (16 bits)
- Packet carries source/destination port numbers in transport header
- OS stores mapping between sockets and ports
- Port: in packets
- Socket: in OS

More on Ports

Separate 16-bit port address space for UDP, TCP

"Well known" ports (0-1023)
- Agreement on which services run on these ports
- e.g., ssh:22, http:80
- Client (app) knows appropriate port on server
- Services can listen on well-known port

Ephemeral ports (most 1024-65535):
- Given to clients (at random)

Multiplexing and Demultiplexing

Host receives IP datagrams
- Each datagram has source and destination IP address,
- Each segment has source and destination port number

Host uses IP addresses and port numbers to direct the segment to appropriate socket

Payload

| 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 6-bit Version | 4-bit Header Length | 8-bit Type of Service (TOS) | 16-bit Total Length (Bytes) |
| 16-bit Identification | 13-bit Fragment Offset |
| 8-bit Protocol | 16-bit Header Checksum |
| 32-bit Source IP Address | 32-bit Destination IP Address |
| Options (if any) | Payload |

| 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 4-bit Version | 5-bit Protocol | 4-bit Type of Service (TOS) | 16-bit Total Length (Bytes) |
| 16-bit Identification | 13-bit Fragment Offset |
| 8-bit Time to Live (TTL) | 16-bit Header Checksum |
| 32-bit Source IP Address | 32-bit Destination IP Address |
| Options (if any) | Payload |
Connection Mappings
For UDP ports (SOCK_DGRAM)
  - OS stores (local port, local IP address) ↔ socket
For TCP ports (SOCK_STREAM)
  - OS stores (local port, local IP; remote port, remote IP) ↔ socket

Why the difference?
Implications for mobility
Why do you need to include local IP?

UDP: User Datagram Protocol
Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Send messages to and receive them from a socket
UDP described in RFC 768 – (1980!)
  - IP plus port numbers to support (de)multiplexing
  - Optional error checking on the packet contents
    - (checksum field = 0 means “don’t verify checksum”)

Why Would Anyone Use UDP?
Finer control over what data is sent and when
  - As soon as an application process writes into the socket
    - ... UDP will package the data and send the packet
No delay for connection establishment
  - UDP just blasts away without any formal preliminaries
  - … which avoids introducing any unnecessary delays
No connection state
  - No allocation of buffers, sequence #s, timers …
  - … making it easier to handle many active clients at once
Small packet header overhead
  - UDP header is only 8 bytes

Popular Applications That Use UDP
Some interactive streaming apps
  - Retransmitting lost/corrupted packets often pointless:
    - by the time the packet is retransmitted, it’s too late
  - telephone calls, video conferencing, gaming…
  - Modern streaming protocols using TCP (and HTTP)
Simple query protocols like Domain Name System (DNS)
  - Connection establishment overhead would double cost
  - Easier to have application retransmit if needed

Transmission Control Protocol (TCP)
Reliable, in-order delivery (previously, but quick review)
  - Ensures byte stream (eventually) arrives intact
  - In the presence of corruption and loss
Connection oriented
  - Explicit set-up and tear-down of TCP session
Full duplex stream-of-bytes service (today)
  - Sends and receives a stream of bytes, not messages
Flow control (previously, but quick review)
  - Ensures that sender doesn’t overwhelm receiver
Congestion control (next week)
  - Dynamic adaptation to network path’s capacity
**Basic Components of Reliability**

- **ACKs**
  - Can’t be reliable without knowing whether data has arrived
  - TCP uses byte sequence numbers to identify payloads

- **Checksums**
  - Can’t be reliable without knowing whether data is corrupted
  - TCP does checksum over TCP and pseudoheader

- **Timeouts and retransmissions**
  - Can’t be reliable without retransmitting lost/corrupted data
  - TCP retransmits based on timeouts and duplicate ACKs
  - Timeout based on estimate of RTT

**Other TCP Design Decisions**

- **Sliding window flow control**
  - Allow W contiguous bytes to be in flight

- **Cumulative acknowledgements**
  - Selective ACKs (full information) also supported (ignore)
  - Single timer set after each payload is ACKed
  - Timer is effectively for the “next expected payload”
  - When timer goes off, resend that payload and wait
  - And double timeout period

- **Various tricks related to “fast retransmit”**
  - Using duplicate ACKs to trigger retransmission

**TCP Header**

- Source port
- Destination port
- Sequence number
- Acknowledgment
- HdrLen
- Flags
- Advertised window
- Checksum
- Urgent pointer
- Options (variable)
- Data

**Segments and Sequence Numbers**

- Application @ Host A
- Host B
- Segment sent when:
  1. Segment full (Max Segment Size),
  2. Not full, but times out

**TCP Segment**

- **IP packet**
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet

- **TCP packet**
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long

- **TCP segment**
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU – (IP header) – (TCP header)
**Sequence Numbers**

Host A

ISN (initial sequence number)

Sequence number

= 1st byte in segment = ISN + k

---

**ACKing and Sequence Numbers**

Sender sends packet

- Data starts with sequence number X
- Packet contains B bytes
  - X, X+1, X+2, ..., X+B-1

Upon receipt of packet, receiver sends an ACK

- If all data prior to X already received:
  - ACK acknowledges X+B (because that is next expected byte)
- If highest contiguous byte received is smaller value Y
  - ACK acknowledges Y+1
- Even if this has been ACKed before

---

**TCP Header**

Starting byte offset of data carried in this segment

Source port | Destination port | Sequence number | Acknowledgment | HdrLen | Flags | Advertised window | Checksum | Options (variable) | Data
---|---|---|---|---|---|---|---|---|---

---

**Sliding Window Flow Control**

Advertised Window: W

- Can send W bytes beyond the next expected byte

Receiver uses W to prevent sender from overflowing buffer

Limits number of bytes sender can have in flight
Filling the Pipe

Simple example:
- $W$ (in bytes), which we assume is constant
- $RTT$ (in sec), which we assume is constant
- $B$ (in bytes/sec)

How fast will data be transferred?

If $W/RTT < B$, the transfer has speed $W/RTT$
If $W/RTT > B$, the transfer has speed $B$

Advertised Window Limits Rate

Sender can send no faster than $W/RTT$ bytes/sec
Receiver only advertises more space when it has consumed old arriving data
In original TCP design, that was the sole protocol mechanism controlling sender’s rate
What’s missing?

Implementing Sliding Window

Both sender & receiver maintain a window
- Sender: not yet ACK’ed
- Receiver: not yet delivered to application

Left edge of window:
- Sender: beginning of unacknowledged data
- Receiver: beginning of undelivered data

For the sender:
- Window size = maximum amount of data in flight
For the receiver:
- Window size = maximum amount of undelivered data

Sliding Window

For the sender, when receives an acknowledgment for new data, window advances (slides forward)

For the receiver, as the receiving process consumes data, the window slides forward

Sliding Window
**Sliding Window Summary**

Sender: window advances when new data ack’d

Receiver: window advances as receiving process consumes data

Receiver advertises to the sender where the receiver window currently ends (“righthand edge”)
  - Sender agrees not to exceed this amount
  - It makes sure by setting its own window size to a value that can’t send beyond the receiver’s righthand edge

**TCP Header: What’s left?**

- **Source port**
- **Destination port**
- **Sequence number**
- **Acknowledgment**
- **Advertised window**
- **HdrLen**
- **Flags**
  - SYN
  - ACK
  - FIN
  - RST
  - PSH
  - URG
- **Checksum**
- **Urgent pointer**
- **Options (variable)**
- **Data**
Step 1: A’s Initial SYN Packet

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>ACK</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
</tr>
</thead>
<tbody>
<tr>
<td>A’s port</td>
<td>B’s port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A’s Initial Sequence Number</td>
<td>(Irrelevant since ACK not set)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advertised window</td>
<td>5=20B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A tells B it wants to open a connection...

Step 2: B’s SYN-ACK Packet

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>ACK</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
</tr>
</thead>
<tbody>
<tr>
<td>B’s port</td>
<td>A’s port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B’s Initial Sequence Number</td>
<td>ACK = A’s ISN plus 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advertised window</td>
<td>20B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B tells A it accepts, and is ready to hear the next byte...
... upon receiving this packet, A can start sending data

Step 3: A’s ACK of the SYN-ACK

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>A’s port</td>
<td>B’s port</td>
<td></td>
</tr>
<tr>
<td>A’s Initial Sequence Number</td>
<td>B’s ISN plus 1</td>
<td></td>
</tr>
<tr>
<td>Advertised window</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A tells B it’s likewise okay to start sending
... upon receiving this packet, B can start sending data

Timing Diagram: 3-Way Handshaking

Client (initiator)
SYN, SeqNum = x
SYN + ACK, SeqNum = y, Ack = x + 1
ACK, Ack = y + 1

Server
Active
Open
Passive
Open
connect()
listen()
accept()

SYN Loss and Web Downloads

User clicks on a hypertext link
- Browser creates a socket and does a “connect”
- The “connect” triggers the OS to transmit a SYN
If the SYN is lost...
- 3-6 seconds of delay: can be very long
- User may become impatient
- ... and click the hyperlink again, or click “reload”
User triggers an “abort” of the “connect”
- Browser creates a new socket and another “connect”
- Essentially, forces a faster send of a new SYN packet!
- Sometimes very effective, and the page comes quickly

Normal Termination, One Side At A Time

Finish (FIN) to close and receive remaining bytes
- FIN occupies one octet in the sequence space
Other host ack’s the octet to confirm
Closes A’s side of the connection, but not B’s
- Until B likewise sends a FIN
- Which A then acks

Tearing Down the Connection

Normal Termination, One Side At A Time

Finish (FIN) to close and receive remaining bytes
- FIN occupies one octet in the sequence space
Other host ack’s the octet to confirm
Closes A’s side of the connection, but not B’s
- Until B likewise sends a FIN
- Which A then acks

What if the SYN Packet Gets Lost?

Suppose the SYN packet gets lost
- Packet is lost inside the network, or:
- Server discards the packet (e.g., listen queue is full)
Eventually, no SYN-ACK arrives
- Sender sets a timer and waits for the SYN-ACK
- ... and retransmits the SYN if needed
How should the TCP sender set the timer?
- Sender has no idea how far away the receiver is
- Hard to guess a reasonable length of time to wait
- SHOULD (RFCs 1122 & 2988) use default of 3 seconds
- Other implementations instead use 6 seconds
Normal Termination, Both Together

Same as before, but B sets FIN with their ack of A’s FIN

SYN
SYN ACK
ACK
Data
FIN
FIN + ACK
ACK
time
A
B

Connection now closed

Timeout:
Avoid reincarnation
Can retransmit FIN ACK if ACK lost

Abrupt Termination

A sends a RESET (RST) to B
- E.g., because app. process on A crashed
- That’s it
- B does not ack the RST
- Thus, RST is not delivered reliably
- But: if B sends anything more, will elicit another RST

TCP State Transitions

Reliability: TCP Retransmission

Timeouts and Retransmissions

Reliability requires retransmitting lost data
Involves setting timer and retransmitting on timeout
TCP resets timer whenever new data is ACKed
- Retx of packet containing “next byte” when timer goes off

Example
Arriving ACK expects 100
- Sender sends packets 100, 200, 300, 400, 500
  - Timer set for 100
Arriving ACK expects 300
- Timer set for 300
Timer goes off
  - Packet 300 is resent
Arriving ACK expects 600
- Packet 600 sent
- Timer set for 600

Setting the Timeout Value

Timeout too long \(\rightarrow\) inefficient
Timeout too short \(\rightarrow\) duplicate packets

RTT Estimation

Use exponential averaging of RTT samples

\[
\text{SampleRTT} = \text{AckRcvdTime} - \text{SendPacketTime} \\
\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT} \\
0 < \alpha \leq 1
\]
Exponential Averaging Example

\[
\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}
\]
Assume RTT is constant → \( \text{SampleRTT} = \text{RTT} \)

Problem: Ambiguous Measurements

How do we differentiate between the real ACK, and ACK of the retransmitted packet?

Karn/Partridge Algorithm

- Measure \( \text{SampleRTT} \) only for original transmissions
- Once a segment has been retransmitted, do not use it for any further measurements
- Computes \( \text{EstimatedRTT} \) using \( \alpha = 0.875 \)

Timeout value (RTO) = \( 2 \times \text{EstimatedRTT} \)

Use exponential backoff for repeated retransmissions
- Every time RTO timer expires, set RTO ← 2 · RTO
  - (Up to maximum 60 sec)
- Every time new measurement comes in (= successful original transmission), collapse RTO back to \( 2 \times \text{EstimatedRTT} \)

Karn/Partridge in action

This is all very interesting, but…..

Implementations often use a coarse-grained timer
- 500 msec is typical

So what?
- Above algorithms are largely irrelevant
- Incurring a timeout is expensive

So we rely on duplicate ACKs

Loss with cumulative ACKs

"Duplicate ACKs" are a sign of an isolated loss
- The lack of ACK progress means 500 hasn’t been delivered
- Stream of ACKs means some packets are being delivered

Therefore, could trigger resend upon receiving \( k \) duplicate ACKs
- TCP uses \( k = 3 \)

We will revisit this in congestion control

Loss with cumulative ACKs

Sender sends packets with 100B and seqnos.: 100, 200, 300, 400, 500, 600, 700, 800, 900, …

Assume the fifth packet (seqno 500) is lost, but no others

Stream of ACKs will be: 200, 300, 400, 500, 500, 500, 500, …