Lecture 25: End-to-End Architecture and TCP
Recall: Two-Phase Commit

• Consensus Goal: Everyone agrees on the state of the distributed system
  – Doesn’t depend who you ask. Doesn’t matter if nodes go down

• Distributed Transactions
  – Atomic, can’t revert once agreement is reached

• Voting protocol requires unanimity

• Transaction committed if and only if: all workers and coordinator vote to commit
  – Nodes never take back their vote
  – Nodes work in lock step (for an item)
    » Don’t perform new transactions until old one is resolved
    » Stall until transaction is resolved

• Majority-based protocols make progress despite failed nodes
  – Add considerable complexity
Bottom line: in a network . . .

- All distributed systems boil down to message passing
  - Across a network, the internet
  - Spread across the world

- Things fail
  - Nodes die, software crashes, routers too, messages get corrupted in flight, ...
  - Dealing with reliability / failure is inherent
  - Many techniques to utilize

- Limited knowledge of what happened is fundamental

- Question is what steps are taken at what level?
  - What is the “contract” with the infrastructure?
  - What can we reason from and what can we not?
The E2E Concept

• Traditional Engineering Goal: design the infrastructure to meet application requirements
  – Optimizing for Cost, Reliability, Performance, …

• Challenge: infrastructure is most costly & difficult to create and evolves most slowly
  – Applications evolve rapidly, as does technology

• End-to-end Design Concept
  – Utilize intelligence at the point of application
  – Infrastructure need not meet all application requirements directly
  – Only what the end-points cannot reasonably do themselves
    » Avoid redundancy, semantic mismatch, …
  – Enable applications and incorporate technological advance

• Design for Change! - and specialization
  – Layers & protocols
Recall: Sockets over TCP/IP

• Special kind of socket: server socket
  – Has file descriptor
  – Can’t read or write

• Two operations:
  1. `listen()`: Start allowing clients to connect
  2. `accept()`: Create a *new socket* for a *particular* client connection
Socket Setup over TCP/IP

- **5-Tuple identifies each connection:**
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (TCP here)

- Often, Client Port “randomly” assigned
  - Done by OS during client socket setup

- Server Port often “well known”
  - 80 (web), 443 (secure web), 25 (sendmail), etc
  - Well-known ports from 0—1023
Recall: Sockets in Schematic

Client

Create Client Socket

Connect it to server (host:port)

Connection Socket

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept syscall()

Connection Socket

read request

write response

Close Connection Socket

Close Server Socket
Linux Network Architecture
Networking Challenge

• Many different applications
  – Email, Web, Online Games, etc.

• Many different network types and technologies
  – Wireless, Wired, Optical, etc.

• Inherently distributed, parts can fail, no overall management authority, …

• How do we manage this complexity?
• Such that the whole is more “reliable” than its parts
  – Available, despite underlying unreliability
Networking Challenge

• Re-implement every application for every technology?

• No
Networking Challenge

- Re-implement sockets for every technology?
  - No
Layering

• Complex services from simpler ones
  – Physical and Link Layers (Wi-Fi, Ethernet, …)
    » Unreliable, local exchange of limited-size frames
  – Network (IP) – routing between local networks
    » Unreliable, global exchange of limited-size packets
  – Transport (e.g., TCP or UDP) – Glue
    » Reliable (with retries), ordering, stream of bytes
    » Not-neccesarily reliable datagrams (UDP)
  – Application – Everything on top of sockets
The Internet Hourglass

- Applications
- Transport
- Data Link
- Physical

- Sockets – the OS “Hourglass”

- The Hourglass Model

- Waist

- IP

- TCP
- UDP

- SMTP
- HTTP
- DNS
- NTP

- Ethernet
- SONET
- 802.11

- Copper
- Fiber
- Radio
Internet Protocol (IP)

• Internet Protocol: Internet’s network layer
• Service it provides: “Best-Effort” Packet Delivery
  – Tries it’s “best” to deliver packet to its destination
  – Packets may be lost
  – Packets may be corrupted
  – Packets may be delivered out of order
Internet: Network of Networks

Hierarchy of Networks: Scales to billions of hosts

subnet1
subnet2
Router
Other subnets
Router
subnet3
Transcontinental Link

11/26/19
UCB CS162 Fa19 L25
Internet Protocol Features

• Routing – an IP packet goes anywhere
  – Just need the destination IP address

• Fragmentation – split big messages into smaller pieces
  – Think about downloading a file
  – Maximum size 64K
  – Reassemble at destination
  – Hides differences in link/physical layers

• Multiple protocols running on top
  – ICMP, TCP, UDP, …
Internet Protocol "Non-Features"

- **Unreliable Delivery ("Best Effort")**
  - IP packet delivery not guaranteed
  - May be lost by underlying physical layer (e.g., radio noise)
  - May be dropped in transit

- **Out-of-order/duplicate delivery**
  - Tolerance of physical layer retrying transmission
  - Tolerance of multiple paths
Internet Architecture: The Five Layers

- Lower three layers implemented everywhere
- Top two layers implemented only at hosts
- Logically, layers interacts with peer’s corresponding layer
Physical Communication

- Communication goes down to physical network
- Then from network peer to peer
- Then up to relevant layer
Implications of Hourglass

- There is only one Network-Layer Protocol: IP
- Allows networks to interoperate
- Above IP: Applications function on all networks
- Below IP: Change network’s construction without disturbing applications
- One drawback: Changing IP itself (e.g. transitioning to IPv6) very involved
End-to-End Principal

• Seen as a guiding principle of the Internet
• Some types of network functionality can only be correctly implemented end-to-end
  – Reliability, security, etc.
• Implementing complex functionality in the network:
  – Doesn’t necessarily reduce complexity on end hosts
  – Does increase network complexity
  – Imposes a cost on all applications, even if they don’t need the functionality
Example: Reliable File Transfer

- Solution 1: make each step reliable, and then concatenate them

- Solution 2: end-to-end check and try again if necessary
Summary: Network Layers

- **Link Layer (local network)**
  - Send *frames* addressed to neighboring machines
  - Ethernet, Wi-Fi

- **Network layer (connecting local networks)**
  - Forwarding between local networks
  - Send *packets* addressed to machines anywhere
  - IP

- **Transport Layer (making streams)**
  - Turn sequence of packets into reliable byte stream
  - TCP

- **OS: Sockets abstraction : Application**
  - File Handle, Read/Write like files
  - But setup/teardown very different from open/close
Layering: Packets in Envelopes

- Application Layer
- Transport Layer
- Network Layer
- Datalink Layer
- Physical Layer

Data

Trans. Hdr.

Net. Hdr.

Frame Hdr.

101010100110101110

101010100110101110

UCB CS162 Fa19 L25
Ordered Messages: Problem

• Want to divide a message into packets / frames

• Think about downloading a file over IP
  – 64K max packet size

• IP might reorder these packets
  – Imagine receiving the end of a file before the beginning!
Ordered Messages: Solution

• Simulate ordered messages on top of unordered messages

• Assign each packet a sequence number: 0, 1, 2, 3, ...
  – If packets arrive out of order, hold on to them
  – Deliver them *in order* to user (through socket interface)

• Example: Hold on to #3 until #2 arrives, etc.
Reliable Message Delivery: Problem

• All physical networks can garble or drop packets
  – Physical hardware problems (bad wire, bad signal)

• Therefore, IP can garble or drop packets
  – It doesn't repair this itself (end-to-end principle!)

• Building reliable message delivery
  – Confirm that packets aren't garbled
  – Confirm that packets arrive exactly once
Using Acknowledgements

- Checksum: Detect garbled packets
- Receiver sends a packet to acknowledge when a packet received and ungarbled
  - No acknowledgement? Resend after timeout
- What if acknowledgement dropped?
  - Packet is resent (wasteful), second chance to acknowledge
What about duplicates?

• Add a Sequence Number
• Wait for its Ack before sending the next
  – “stop and wait”
• Latency limits bandwidth
  – Remember Little’s Law
  – $N = L \times B$
• With sequence #s can detect duplicates and maintain order
Window-Based Acknowledgements

- Windowing protocol (not quite TCP)
- Send up to $N$ packets without ack
  - Allows pipelining of packets
  - $N$ limits queue size
- Both source and destination need to store $N$ packets (why?)
- Each packet has sequence number
  - ACK says "Received all packets up to number $X$"
  - Advances window
Transport Layer (4)

• Service:
  – Provide end-to-end communication between processes
  – Demultiplexing of communication between hosts
  – Possible other services:
    » Reliability in the presence of errors
    » Timing properties
    » Rate adaption (flow-control, congestion control)

• Interface: send message to “specific process” at given destination; local process receives messages sent to it
  – How are they named?

• Protocol: port numbers, perhaps implement reliability, flow control, packetization of large messages, framing

• Prime Examples: TCP and UDP
Internet Transport Protocols

• Datagram service (UDP)
  – No-frills extension of “best-effort” IP
  – Multiplexing/Demultiplexing among processes

• Reliable, in-order delivery (TCP)
  – Connection set-up & tear-down
  – Discarding corrupted packets (segments)
  – Retransmission of lost packets (segments)
  – Flow control
  – Congestion control

• Services not available
  – Delay and/or bandwidth guarantees
  – Sessions that survive change-of-IP-address
Application Layer (7 - not 5!)

- Service: any service provided to the end user
- Interface: depends on the application
- Protocol: depends on the application

- Examples: Skype, SMTP (email), HTTP (Web), Halo, BitTorrent …

- What happened to layers 5 & 6?
  - “Session” and “Presentation” layers
  - Part of OSI architecture, but not Internet architecture
  - Their functionality is provided by application layer
Socket API

• Base level Network programming interface

Application

Transport

Network

Socket API

TCP

UDP

IP
BSD Socket API

- Created at UC Berkeley (1980s)

- Most popular network API

- Ported to various OSes, various languages
  - Windows Winsock, BSD, OS X, Linux, Solaris, …
  - Socket modules in Java, Python, Perl, …

- Similar to Unix file I/O API
  - In the form of file descriptor (sort of handle).
  - Can share same read()/write()/close() system calls
TCP: Transport Control Protocol

• Reliable, in-order, and at most once delivery

• Stream oriented: messages can be of arbitrary length

• Provides multiplexing/demultiplexing to IP

• Provides congestion and flow control

• Application examples: file transfer, chat, http
TCP Service

1) Open connection: 3-way handshaking

2) Reliable byte stream transfer from (IPa, TCP_Port1) to (IPb, TCP_Port2)
   • Indication if connection fails: Reset

3) Close (tear-down) connection
Recall: Socket creation and connection

- File systems provide a collection of permanent objects in structured name space
  - Processes open, read/write/close them
  - Files exist independent of the processes
- Sockets provide a means for processes to communicate (transfer data) to other processes.
- Creation and connection is more complex
- Form 2-way pipes between processes
  - Possibly worlds away
Recall: Client Protocol

```c
char *hostname;
int sockfd, portno;
struct sockaddr_in serv_addr;
struct hostent *server;

server = buildServerAddr(&serv_addr, hostname, portno);

/* Create a TCP socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0)

/* Connect to server on port */
connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
printf("Connected to %s:%d\n", server->h_name, portno);
```

**Protocols:**
- **PF_LOCAL**: Host-internal protocols, formerly called PF_UNIX,
- **PF_UNIX**: Host-internal protocols, deprecated, use PF_LOCAL,
- **PF_INET**: Internet version 4 protocols,
- **PF_ROUTE**: Internal Routing protocol,
- **PF_KEY**: Internal key-management function,
- **PF_INET6**: Internet version 6 protocols,
- **PF_SYSTEM**: System domain,
- **PF_NDRV**: Raw access to network device
Recall: Sockets in Schematic

Client

Create Client Socket

Connect it to server (host:port)

Connection Socket

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

Accept syscall()

Connection Socket

write response

read request

Close Connection Socket

Close Server Socket

Close Client Socket
Open Connection: 3-Way Handshaking

• Goal: agree on a set of parameters, i.e., the start sequence number for each side
  – Starting sequence number: sequence of first byte in stream
  – Starting sequence numbers are random
Open Connection: 3-Way Handshaking

- Server waits for new connection calling `listen()`
- Sender call `connect()` passing socket which contains server’s IP address and port number
  - OS sends a special packet (SYN) containing a proposal for first sequence number, \( x \)
Open Connection: 3-Way Handshaking

- If it has enough resources, server calls `accept()` to accept connection, and sends back a SYN ACK packet containing
  - Client’s sequence number incremented by one, \((x + 1)\)
    » Why is this needed?
  - A sequence number proposal, \(y\), for first byte server will send

```
Client (initiator)                      Server

Active Open
connect()                               listen()

SYN, SeqNum = x                         accept()

SYN and ACK, SeqNum = y and Ack = x + 1

ACK, Ack = y + 1

allocate buffer space
```

UCB CS162 Fa19 L25
3-Way Handshaking (cont’d)

• Three-way handshake adds 1 RTT delay

• Why?
  – Congestion control: SYN (40 byte) acts as cheap probe
  – Protects against delayed packets from other connection (would confuse receiver)
Close Connection

- **Goal:** both sides agree to close the connection
- **4-way connection tear down**

![Diagram]

- Can retransmit FIN ACK if it is lost
- Timeout
- Closed
Recall: Sockets in Schematic

Client

1. Create Client Socket
2. Connect it to server (host:port)
3. Close Client Socket
4. write request
5. read response
6. Close Connection Socket

Server

1. Create Server Socket
2. Bind it to an Address (host:port)
3. Listen for Connection
4. Listen for Connection
5. Accept syscall()
6. read request
7. write response
8. Close Connection Socket
9. Close Server Socket
The Transfer Part - Reliability

Request Response Protocol

Client (issues requests)  Server (performs operations)

write(rqfd, rqbuf, buflen);

n = read(rfd, rbuf, rmax);

write(wfd, respbuf, len);

n = read(resfd, resbuf, resmax);

Client (issues requests)

Server (performs operations)
TCP Sliding Window

- Window represents packets:
  - That might need to be re-sent (dropped, garbled)
  - That receiver needs to buffer (in-order delivery to user)
Transmission Control Protocol (TCP)

- Reliable byte stream between two processes on different machines, over the Internet
- Bi-directional: two streams for every connection
- Segments byte streams into “packets”, hands those to IP
  - IP may “fragment” into link frames
- Window-based acknowledgement protocol
- Automatically retransmits lost packets
- Adjusts rate of transmission to avoid congestion
  - How? Window Size
Reliable Transfer

• Retransmit missing packets
  – Numbering of packets and ACKs

• Do this efficiently
  – Keep transmitting whenever possible
  – Detect missing packets and retransmit quickly

• Two schemes
  – Stop & Wait
  – Sliding Window (Go-back-n and Selective Repeat)
Window-Based Acknowledgement

• Packet lost?
  – Resent after timeout (no ACK received)

• Acknowledgement lost?
  – Packet resent, causing ACK to be resent, too

• Discard out-of-order packets?
  – If no, need some way to indicate holes in window
Detecting Packet Loss?

• **Timeouts**
  – Sender timeouts on not receiving ACK

• **Missing ACKs**
  – Receiver ACKs each packet
  – Sender detects a missing packet when seeing a gap in the sequence of ACKs
  – Need to be careful! Packets and ACKs might be reordered

• **NACK: Negative ACK**
  – Receiver sends a NACK specifying a packet it is missing
Stop & Wait w/o Errors

- Send; wait for ack; repeat
- **RTT**: Round Trip Time (RTT): time it takes a packet to travel from sender to receiver and back
  - One-way latency (d): one way delay from sender and receiver

\[
\text{RTT} = 2 \times d \quad \text{(if latency is symmetric)}
\]
Stop & Wait w/o Errors

- How many packets can you send?
- 1 packet / RTT
- Throughput: number of bits delivered to receiver per sec

![Diagram of Stop & Wait protocol]

Sender

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTT</td>
</tr>
</tbody>
</table>

Receiver

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTT</td>
</tr>
</tbody>
</table>

packets

ACK 1

ACK 2

ACK 3
Stop & Wait w/o Errors

- Say, RTT = 100ms
- 1 packet = 1500 bytes
- Throughput = $1500 \times 8\text{bits} / 0.1\text{s} = 120 \text{ Kbps}$

\begin{center}
\begin{tabular}{c|c|c}
\textbf{Sender} & \textbf{Receiver} \\
\hline
\text{RTT} & \text{RTT} \\
\text{RTT} & \text{RTT} \\
\text{Time} & \\
\end{tabular}
\end{center}

1

ACK 1

ACK 2

2

3

11/26/19 UCB CS162 Fa19 L25
Stop & Wait w/o Errors

- Can be very inefficient for high capacity links
- Throughput doesn’t depend on the network capacity → even if capacity is 1Gbps, we can only send 120 Kbps!

---

**Diagram:**

- Sender
- Receiver
- RTT
- Time
- ACK 1
- ACK 2
Stop & Wait with Errors

- If a loss wait for a retransmission timeout and retransmit
- How do you pick the timeout?

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTT</td>
<td></td>
</tr>
<tr>
<td>timeout</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

ACK 1
Sliding Window

- *window* = set of adjacent sequence numbers

- The size of the set is the *window size*

- Assume window size is n

- Let A be the last ACK’d packet of sender without gap; then window of sender = \{A+1, A+2, ..., A+n\}

- Sender can send packets in its window

- Let B be the last received packet without gap by receiver, then window of receiver = \{B+1, ..., B+n\}

- Receiver can accept out of sequence, if in window
Sliding Window w/o Errors

- Throughput = $W \cdot \frac{\text{packet\_size}}{\text{RTT}}$

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>{}</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>{}</td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td>{}</td>
</tr>
<tr>
<td>{2, 3, 4}</td>
<td>{}</td>
</tr>
<tr>
<td>{3, 4, 5}</td>
<td>{}</td>
</tr>
<tr>
<td>{4, 5, 6}</td>
<td>{}</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Unacked packets in sender’s window

Out-o-seq packets in receiver’s window

Window size (W) = 3 packets

11/26/19

UCB CS162 Fa19 L25
Example: Sliding Window w/o Errors

• Assume
  – Link capacity, C = 1Gbps
  – Latency between end-hosts, RTT = 80ms
  – packet_length = 1000 bytes

• What is the window size W to match link’s capacity, C?

• Solution
  We want Throughput = C
  Throughput = W*packet_size/RTT
  C = W*packet_size/RTT
  W = C*RTT/packet_size = 10^9bps*80*10^-3s/(8000b) = 10^4 packets

Window size ~ Bandwidth (Capacity), delay (RTT/2)
Sliding Window with Errors

• Two approaches
  – Go-Back-n (GBN)
  – Selective Repeat (SR)

• In the absence of errors they behave identically

• Go-Back-n (GBN)
  – Transmit up to $n$ unacknowledged packets
  – If timeout for ACK($k$), retransmit $k$, $k+1$, ...
  – Typically uses NACKs instead of ACKs
    » Recall, NACK specifies first in-sequence packet missed by receiver
GBN Example with Errors

Window size (W) = 3 packets

Timeout Packet 4

Assume packet 4 lost!

Why doesn’t sender retransmit packet 4 here?

Out-of-seq packets in receiver’s window

4 is missing

Packet 4

NACK 4

NACK 4

Out of sequence packets in the receiver's window.
Selective Repeat (SR)

- Sender: transmit up to $n$ unacknowledged packets

- Assume packet $k$ is lost

- Receiver: indicate packet $k$ is missing (use ACKs)

- Sender: retransmit packet $k$
SR Example with Errors

Window size \((W) = 3\) packets

Unacked packets in sender’s window

- \(\{1\}\)
- \(\{1, 2\}\)
- \(\{1, 2, 3\}\)
- \(\{2, 3, 4\}\)
- \(\{3, 4, 5\}\)
- \(\{4, 5, 6\}\)
- \(\{4, 5, 6\}\)
- \(\{7\}\)

ACK 5

ACK 6

Time

Sender

Receiver
TCP Windows and Seq. Numbers

Sender has three regions:

1. Sent and acknowledged
2. Sent and not acknowledged
3. Not yet sent

Sequence Numbers

Sender

Not yet sent

Sent

Received

Sender

Sent

Not yet received

Received

Received

Buffered

Receiver

Given to app

Not yet received

Not yet sent

Sender

UCB CS162 Fa19 L25
TCP Windows and Seq. Numbers

Receiver has three regions:
1. Received and acknowledged
2. Received and buffered
3. Not yet received
Window-Based Acknowledgements

100 140 190 230 260 300 340 380 400

Seq:100 Size:40
Seq:140 Size:50
Seq:190 Size:40
Seq:230 Size:30
Seq:260 Size:40
Seq:300 Size:40
Seq:340 Size:40
Seq:380 Size:20

Retransmit!
Congestion

- Too much data trying to flow through some part of the network

- IP's solution: Drop packets

- What happens to TCP connection?
  - Lots of retransmission – wasted work
  - Lots of waiting for timeouts – underutilized connection
Congestion Avoidance

• Solution: Adjust Window Size
• AIMD: Additive Increase, Multiplicative Decrease
  – When packet dropped (missed ack), cut window size in half
  – If no timeouts, increase window size by C for each acknowledgement received
    » Until half the old window then slower
Congestion Avoidance: Changing Window

Congestion Avoidance

• Congestion
  – How long should timeout be for re-sending messages?
    » Too long → wastes time if message lost
    » Too short → retransmit even though ACK will arrive shortly
  – Stability problem: more congestion ⇒ ACK is delayed ⇒ unnecessary timeout ⇒ more traffic ⇒ more congestion
    » Closely related to window size at sender: too big means putting too much data into network

• How does the sender’s window size get chosen?
  – Must be less than receiver’s advertised buffer size
  – Try to match the rate of sending packets with the rate that the slowest link can accommodate
  – Sender uses an adaptive algorithm to decide size of N
    » Goal: fill network between sender and receiver
    » Basic technique: slowly increase size of window until acknowledgements start being delayed/lost

• TCP solution: “slow start” (start sending slowly)
  – If no timeout, slowly increase window size (throughput) by 1 for each ACK received
  – Timeout ⇒ congestion, so cut window size in half
  – “Additive Increase, Multiplicative Decrease”
Summary: Network Layers

• Link Layer (local network)
  – Send *frames* addressed to neighboring machines
  – Ethernet, Wi-Fi

• Network layer (connecting local networks)
  – Forwarding between local networks
  – Send packets addressed to machines anywhere
  – IP

• Transport Layer (making streams)
  – Turn sequence of packets into reliable byte stream
  – TCP
# Glue: Adding Functionality

<table>
<thead>
<tr>
<th>Physical Reality: Frames</th>
<th>Abstraction: Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited Size</td>
<td>Arbitrary Size</td>
</tr>
<tr>
<td>Unordered (sometimes)</td>
<td>Ordered</td>
</tr>
<tr>
<td>Unreliable</td>
<td>Reliable</td>
</tr>
<tr>
<td>Machine-to-Machine</td>
<td>Process-to-Process</td>
</tr>
<tr>
<td>Only on Local Area Net</td>
<td>Routed Anywhere</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
</tbody>
</table>
## Glue: Adding Functionality

### Network Layer: IP

<table>
<thead>
<tr>
<th>Physical Reality: Frames</th>
<th>Abstraction: Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited Size</td>
<td>Arbitrary Size</td>
</tr>
<tr>
<td>Unordered (sometimes)</td>
<td>Ordered</td>
</tr>
<tr>
<td>Unreliable</td>
<td>Reliable</td>
</tr>
<tr>
<td>Machine-to-Machine</td>
<td>Process-to-Process</td>
</tr>
<tr>
<td>Only on Local Area Net</td>
<td>Routed Anywhere</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
</tbody>
</table>
# Glue: Adding Functionality

## Transport Layer: TCP

<table>
<thead>
<tr>
<th>Physical Reality: Frames</th>
<th>Abstraction: Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited Size</td>
<td>Arbitrary Size</td>
</tr>
<tr>
<td>Unordered (sometimes)</td>
<td>Ordered</td>
</tr>
<tr>
<td>Unreliable</td>
<td>Reliable</td>
</tr>
<tr>
<td>Machine-to-Machine</td>
<td>Process-to-Process</td>
</tr>
<tr>
<td>Only on Local Area Net</td>
<td>Routed Anywhere</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Synchronous</td>
</tr>
</tbody>
</table>

- TCP is the Transport Control Protocol.
- It is used to provide reliable and connection-oriented services.
- TCP packets are transmitted as segments.
Summary: TCP

• Syn | Syn-Ack | Ack to realize Connect/Listen Accept
  – vs Open for a file
  – Demultiplex to processes based on Port

• Use sequence numbers to solve out-of-order delivery problem

• Use acknowledgements to solve reliable delivery problem
  – For better utilization, allow a window of unacknowledged packets
  – Adjust window size in response to perceived congestion events

• FIN | FIN ACK in both direction to tear down
  – Upon close
Summary: TCP

• Use sequence numbers to solve out-of-order delivery problem
• Use acknowledgements to solve reliable delivery problem
• For better utilization, allow a window of unacknowledged packets
• Adjust window size in response to perceived congestion events
TCP State Machine

• From Comer, Internetworking with TCP/IP

Figure 12.13 The TCP finite state machine. Each endpoint begins in the closed state. Labels on transitions show the input that caused the transition followed by the output if any.