End2End Design – The Internet Architecture

David E. Culler
CS162 – Operating Systems and Systems Programming
http://cs162.eecs.berkeley.edu/
Lecture 32
Nov 12, 2014

Read: end-2-end
HW 5: Due today
Mid 2: 11/14
Proj 3: due 12/8
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
Review: Protocols

• Q1: True _ False _ Protocols specify the syntax and semantics of communication

• Q2: True _ False _ Protocols specify the implementation

• Q3: True _ False _ Layering helps to improve application performance

• Q4: True _ False _ “Best Effort” packet delivery ensures that packets are delivered in order

• Q5: True _ False _ In p2p systems a node is both a client and a server

• Q6: True _ False _ TCP ensures that each packet is delivered within a predefined amount of time
Review: Protocols

• Q1: True  False  Protocols specify the syntax and semantics of communication
• Q2: True  False  Protocols specify the implementation
• Q3: True  False  Layering helps to improve application performance
• Q4: True  False  “Best Effort” packet delivery ensures that packets are delivered in order
• Q5: True  False  In p2p systems a node is both a client and a server
• Q6: True  False  TCP ensures that each packet is delivered within a predefined amount of time
There is just one network-layer protocol, **IP**

The “narrow waist” facilitates **interoperability**
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 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

![Diagram of network layers and communication flow]
Implications of Hourglass

Single Internet-layer module (IP):

• **Allows arbitrary networks to interoperate**
  – Any network technology that supports IP can exchange packets

• **Allows applications to function on all networks**
  – Applications that can run on IP can use any network

• **Supports simultaneous innovations above and below IP**
  – But changing IP itself, i.e., IPv6 is very complicated and slow
Layering: Packets in Envelopes

11/12/14
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 adaptation (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
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
Connecting Communication to Processes
Recall: Sockets

Request Response Protocol

Client (issues requests)  Server (performs operations)

```
write(rqfd, rqbuf, buflen);
```

```
n = read(rfd, rbuf, rmax);
```

```
wait
```

```
write(wfd, respbuf, len);
```

```
n = read(resfd, resbuf, resmax);
```

<table>
<thead>
<tr>
<th>9/10/14</th>
<th>cs162 fa14 L5</th>
</tr>
</thead>
</table>
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: Sockets in concept

Client

1. Create Client Socket
2. Connect it to server (host:port)
3. Write request
4. Read response
5. Close Client Socket

Server

1. Create Server Socket
2. Bind it to an Address (host:port)
3. Listen for Connection
4. Accept connection
5. Read request
6. Write request
7. Close Connection Socket
8. Close Server Socket
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);

/* Carry out Client-Server protocol */
client(sockfd);

/* Clean up on termination */
close(sockfd);
Server Protocol (v1)

/* Create Socket to receive requests*/
lstnsockfd = socket(AF_INET, SOCK_STREAM, 0);

/* Bind socket to port */
bind(lstnsockfd, (struct sockaddr *)&serv_addr,sizeof(serv_addr));

while (1) {
  /* Listen for incoming connections */
  listen(lstnsockfd, MAXQUEUE);

  /* Accept incoming connection, obtaining a new socket for it */
  consockfd = accept(lstnsockfd, (struct sockaddr *) &cli_addr, &clilen);

  server(consockfd);

  close(consockfd);
}
close(lstnsockfd);
Sockets in concept: fork

Client

Create Client Socket
Connect it to server (host:port)
write request
read response
Close Client Socket

Server

Create Server Socket
Bind it to an Address (host:port)
Listen for Connection
Accept connection
Connection Socket
read request
write response
Close Connection Socket
Wait for child

Connection Socket
Close Listen Socket
Close Connection Socket
Close Server Socket
Close Listen Socket
Server Protocol (v2)

while (1) {
    listen(lstnsockfd, MAXQUEUE);
    consockfd = accept(lstnsockfd, (struct sockaddr *) &cli_addr, &clilen);

    cpid = fork(); /* new process for connection */
    if (cpid > 0) { /* parent process */
        close(consockfd);
        tcpid = wait(&cstatus);
    } else if (cpid == 0) { /* child process */
        close(lstnsockfd); /* let go of listen socket */

        server(consockfd);

        close(consockfd);
        exit(EXIT_SUCCESS); /* exit child normally */
    }
}

close(lstnsockfd);
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 \texttt{listen()}
- Sender call \texttt{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

```
Active
Open
connect()
```

Client (initiator) - Server

```
listen()
accept()
allocate buffer space
```

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

11/12/14

UCB CS162 Fa14 L32
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

Can retransmit FIN ACK if it is lost

Timeout

closed

data

FIN ACK

FIN

FIN

FIN ACK

Host 1

Host 2

close

closed
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)
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

![Diagram](image_url)

- ACK 1
- ACK 2
- RTT = 2*d (if latency is symmetric)

<table>
<thead>
<tr>
<th>Time</th>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UCB CS162 Fa14 L32
Stop & Wait w/o Errors

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

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ACK 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ACK 2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

RTT

Time
Stop & Wait w/o Errors

- Say, RTT = 100ms
- 1 packet = 1500 bytes
- Throughput = 1500*8bits/0.1s = 120 Kbps
Stop & Wait w/o Errors

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

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

---

Diagram:
- Sender
- Receiver
- RTT
- Timeout
- Time

Diagram shows a timeline with an ACK marked as 1. The ACK arrives after the timeout period, indicating a retransmission is needed.
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 = $W \times \text{packet\_size}/\text{RTT}$

Unacked packets in sender’s window

- {1} 1
- {1, 2} 2
- {1, 2, 3} 3
- {2, 3, 4} 4
- {3, 4, 5} 5
- {4, 5, 6} 6
- ...
- ...
- ...

Out-o-seq packets in receiver’s window

- {}
- {}
- ...
- ...
- ...

Window size $(W) = 3$ packets
Example: Sliding Window w/o Errors

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

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

• Solution
  We want $Throughput = C$
  $Throughput = W \times packet\_size / RTT$
  $C = W \times packet\_size / RTT$
  $W = \frac{C \times RTT}{packet\_size} = \frac{10^9 \text{bps} \times 80 \times 10^{-3} \text{s}}{8000\text{b}} = 10^4$ packets

Window size $\sim$ 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

Sender

1 2 3 4 5 6

Receiver

{ } { } { } {5} {5,6} { }

Out-o-seq packets in receiver’s window

Timeout Packet 4

Why doesn’t sender retransmit packet 4 here?

Assume packet 4 lost!

4 is missing

NACK 4

NACK 4

Assume packet 4 lost!

Why doesn’t sender retransmit packet 4 here?
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

Unacked packets in sender’s window

Window size ($W$) = 3 packets

Sender

1

2

3

4

5

6

7

Receiver

Time

\{1\}

\{1, 2\}

\{1, 2, 3\}

\{2, 3, 4\}

\{3, 4, 5\}

\{4, 5, 6\}

\{4, 5, 6\}

\{7\}
Summary

• **TCP: Reliable Byte Stream**
  – Open connection (3-way handshaking)
  – Close connection: no perfect solution; no way for two parties to agree in the presence of arbitrary message losses (General’s Paradox)

• **Reliable transmission**
  – S&W not efficient for links with large capacity (bandwidth) delay product
  – Sliding window more efficient but more complex

• **Flow Control**
  – OS on sender and receiver manage buffers
  – Sending rate adjusted according to acks and losses
  – Receiver drops to slow sender on over-run