Five-Layer Internet Model

- **Application**: the application (e.g., the Web, Email)
- **Transport**: end-to-end connections, reliability
- **Network**: routing
- **Link (data-link)**: framing, error detection
- **Physical**: 1’s and 0’s/bits across a medium (copper, the air, fiber)
Creating a network app

write programs that:

• run on (different) end systems
• communicate over network
• e.g., web server s/w communicates with browser software
Creating a network app

no need to write software for network-core devices!

• network-core devices do not run user applications

• applications on end systems – rapid app development, propagation
HTTP: HyperText Transfer Protocol

Client/Server model

- **client**: browser that uses HTTP to request, and receive Web objects.
- **server**: Web server that uses HTTP to respond with requested object.
What IS A Web Browser?
HTTP and the Web

- web page consists of objects
- object can be: an HTML file (index.html)

demo.cs.swarthmore.edu/index.html

This is the root page of the demo server. The interesting examples live in the /example directory. They are:

- /example/directory/: An example of a directory.
- /example/fiona.jpg: An example image (one of Kevin's cats).
- /example/hello.txt: A simple text file.
- /example/index.html: An HTML file serving as the default page for the /example directory.
- /example/pic.html: An HTML file that links to the cat picture.
- /example/pride_and_prejudice.pdf: A large PDF (binary) file containing Jane Austen's "Pride and Prejudice".
- /example/pride_and_prejudice.txt: A large text file containing Jane Austen's "Pride and Prejudice".
Web objects

- web page consists of objects
- object can be: JPEG image
Web objects

- web page consists of objects
- object can be: audio file
Web objects

• web page consists of objects

• object can be: video, java applets, etc.
HTTP and the Web

- A web page consists of a **base HTML-file** which includes several referenced objects.
- Each object is addressable by a URL, e.g.,

This is the root page of the demo server. The interesting examples live in the `/example` directory. They are:

- `/example/directory/`: An example of a directory.
- `/example/fiona.jpg`: An example image (one of Kevin's cats).
- `/example/hello.txt`: A simple text file.
- `/example/index.html`: An HTML file serving as the default page for the `/example` directory.
- `/example/pic.html`: An HTML file that links to the cat picture.
- `/example/pride_and_prejudice.pdf`: A large PDF (binary) file containing Jane Austen's "Pride and Prejudice".
- `/example/pride_and_prejudice.txt`: A large text file containing Jane Austen's "Pride and Prejudice".

```
demo.cs.swarthmore.edu/example/pic.html
```

**host name**

**path name**
HTTP Overview

1. User types in a URL.

http://some.host.name.tld/directory/name/file.ext

host name

path name
HTTP Overview

2. Browser establishes connection with server using the Sockets API.
   Calls `socket()`  // create a socket
   Looks up “some.host.name.tld” (DNS: `getaddrinfo`)
   Calls `connect()`  // connect to remote server
   Ready to call `send()`  // Can now send HTTP requests
3. Browser requests data the user asked for

GET /directory/name/file.ext HTTP/1.0
Host: some.host.name.tld

[other optional fields, for example:]
User-agent: Mozilla/5.0 (Windows NT 6.1; WOW64)
Accept-language: en
4. Server responds with the requested data.

HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 1299
Date: Sun, 01 Sep 2013 21:26:38 GMT
(Data data data data data...)
HTTP Overview

5. Browser renders the response, fetches any additional objects, and closes the connection.
HTTP Overview

1. User types in a URL.
2. Browser establishes connection with server.
3. Browser requests the corresponding data.
4. Server responds with the requested data.
5. Browser renders the response, fetches other objects, and closes the connection.

It’s a document retrieval system, where documents point to (link to) each other, forming a “web”.

HTTP Overview (Lab 1)

1. User types in a URL.
2. Browser establishes connection with server.
3. Browser requests the corresponding data.
4. Server responds with the requested data.
5. Browser renders the response, fetches other objects, Save the file and close the connection.

It’s a document retrieval system, where documents point to (link to) each other, forming a “web”.
Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:
   
   `telnet demo.cs.swarthmore.edu 80`

   Opens TCP connection to port 80 (default HTTP server port) at example server.

   Anything typed is sent to server on port 80 at demo.cs.swarthmore.edu
Trying out HTTP (client side) for yourself

2. Type in a GET HTTP request:

```
GET / HTTP/1.1
Host: demo.cs.swarthmore.edu
```

(Hit carriage return twice) This is a minimal, but complete, GET request to the HTTP server.

3. Look at response message sent by HTTP server!
Example

$ telnet demo.cs.swarthmore.edu 80
Trying 130.58.68.26...
Connected to demo.cs.swarthmore.edu.
Escape character is '^[].'
GET / HTTP/1.1
Host: demo.cs.swarthmore.edu

HTTP/1.1 200 OK
Vary: Accept-Encoding
Content-Type: text/html
Accept-Ranges: bytes
ETag: "316912886"
Last-Modified: Wed, 04 Jan 2017 17:47:31 GMT
Content-Length: 1062
Date: Wed, 05 Sep 2018 17:27:34 GMT
Server: lighttpd/1.4.35
Example

$ telnet demo.cs.swarthmore.edu 80
Trying 130.58.68.26...
Connected to demo.cs.swarthmore.edu.
Escape character is '^[]'.
GET / HTTP/1.1
Host: demo.cs.swarthmore.edu

Response
headers

<html><head><title>Demo Server</title></head>
<body>

.....
</body>
</html>

Response
body
(This is what
you should be
saving in lab 1.)
HTTP request message

• two types of HTTP messages: request, response

• HTTP request message: ASCII (human-readable format)

```
GET /index.html HTTP/1.0\r\nHost: web.cs.swarthmore.edu\r\nUser-Agent: Firefox/3.6.10\r\nAccept: text/html,application/xhtml+xml\r
Accept-Language: en-us,en;q=0.5\r\nAccept-Encoding: gzip,deflate\r\nAccept-Charset: ISO-8859-1,utf-8;q=0.7\r\nKeep-Alive: 115\r\nConnection: keep-alive\r\n```

request line
(GET, POST, HEAD, etc. commands)

variable #
header lines
two carriage return, line feed characters

 carriage return character
line-feed character
Why do we have these \r\n (CRLF) things all over the place?

GET /index.html HTTP/1.1\r\nHost: web.cs.swarthmore.edu\r\nUser-Agent: Firefox/3.6.10\r\nAccept: text/html, application/xhtml+xml\r\nAccept-Language: en-us, en; q=0.5\r\nAccept-Encoding: gzip, deflate\r\nAccept-Charset: ISO-8859-1, utf-8; q=0.7\r\nKeep-Alive: 115\r\nConnection: keep-alive\r\n\r\nA. They’re generated when the user hits ‘enter’.
B. They signal the end of a field or section.
C. They’re important for some other reason.
D. They’re an unnecessary protocol artifact.
Why do we have these \r\n (CRLF) things all over the place?

GET /index.html HTTP/1.1\r\nHost: web.cs.swarthmore.edu\r\nUser-Agent: Firefox/3.6.10\r\nAccept: text/html,application/xhtml+xml\r\nAccept-Language: en-us,en;q=0.5\r\nAccept-Encoding: gzip, deflate\r\nAccept-Charset: ISO-8859-1,utf-8;q=0.7\r\nKeep-Alive: 115\r\nConnection: keep-alive\r\n\r\nA. They’re generated when the user hits ‘enter’.
B. They signal the end of a field or section.
C. They’re important for some other reason.
D. They’re an unnecessary protocol artifact.
How else might we delineate messages?

A. There’s not much else we can do.

B. Force all messages to be the same size.

C. Send the message size prior to the message.

D. Some other way (discuss).
HTTP is all text...

• Makes the protocol simple
  – Easy to delineate message (\r\n)
  – (Relatively) human-readable
  – No worries about encoding or formatting data
  – Variable length data

• Not the most efficient
  – Many protocols use binary fields
    • Sending “12345678” as a string is 8 bytes
    • As an integer, 12345678 needs only 4 bytes
  – The headers may come in any order
  – Requires string parsing / processing
HTTP response message

HTTP/1.1 200 OK
Date: Sun, 26 Sep 2010 20:09:20 GMT
Server: Apache/2.0.52 (CentOS)
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT
ETag: "17dc6-a5c-bf716880"
Accept-Ranges: bytes
Content-Length: 2652
Keep-Alive: timeout=10, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=ISO-8859-1

data data data data data data ...

data, e.g., requested HTML file: may not be text!
HTTP response status codes

Status code appears in first line of server-to-client response message.

200 OK
• Request succeeded, requested object later in this msg

301 Moved Permanently
• Requested object moved, new location specified later in this msg
  (Location:)

400 Bad Request
  – Request msg not understood by server

403 Forbidden
  – You don’t have permission to read the object

404 Not Found
  – Requested document not found on this server

505 HTTP Version Not Supported
HTTP response status codes

Status code appears in first line of server-to-client response message.
Many others! Search “list of HTTP status codes”

420 Enhance Your Calm (twitter)
    – Slow down, you’re being rate limited
451 Unavailable for Legal Reasons
    – Censorship?
418 I’m a Teapot
    – Response from a teapot requested to brew a beverage
      (announced Apr 1)
Client-Server communication

• Client:
  – initiates communication
  – must know the address and port of the server
  – active socket

• Server:
  – passively waits for and responds to clients
  – passive socket
What is a socket?

An abstraction through which an application may send and receive data,

in the same way as a open-file handle allows an application to read and write data to storage.
TCP Socket Procedures: Client

- **socket()**: create a new communication endpoint
- **connect()**: actively attempt to establish a connection
- **send()**: receive some data over a connection
- **recv()**: send some data over a connection
- **close()**: release the connection
Recall Inter-process Communication (IPC)

• Processes must communicate to cooperate
• Must have two mechanisms:
  – Data transfer
  – Synchronization
• On a single machine:
  – Threads (shared memory)
  – Message passing
Message Passing (local)

- Operating system mechanism for IPC
  - send (destination, message_buffer)
  - receive (source, message_buffer)
- Data transfer: in to and out of kernel message buffers
- Synchronization
Interprocess Communication (non-local)

• Processes must communicate to cooperate

• Must have two mechanisms:
  – Data transfer
  – Synchronization

• Across a network:
  – Threads (shared memory) **NOT AN OPTION**!
  – Message passing
Message Passing (network)

- Same synchronization
- Data transfer
  - Copy to/from OS socket buffer
  - Extra step across network: hidden from applications
- Synchronization?
TCP socket connection

Server

socket() → bind() → listen() → accept() → recv() → send() → close()

Client

socket() → connect() → send() → recv() → close()
If the client sends a GET request to the server using send() but forgets to send the last /r/n which of the following can happen?

A. Server, Client both recv()
B. Server send()s, Client recv()s
C. Server recv()s, Client send()s
D. Some other combination
If the client sends a GET request to the server using `send()` but forgets to send the last `/r/n` which of the following can happen?

A. Server, Client both `recv()`
B. Server `send()`s, Client `recv()`s
C. Server `recv()`s, Client `send()`s
D. Some other combination
If the client sends a GET request to the server using send() but forgets to send the last /r/n, which of the following can happen?

- socket()
- bind()
- listen()
- accept()
- recv()
- send()
- close()
- connect()
- send()
- recv()
- close()

Synchronization locally on one machine:
- relies on synchronization primitives.

Synchronization over the network:
- depends on the order of sends and receives!
Descriptor Table

For each Process

OS stores a table, per process, of descriptors

Kernel
Descriptors

**NAME**
socket -- create an endpoint for communication

**SYNOPSIS**
#include <sys/socket.h>

int
socket(int domain, int type, int protocol);

**DESCRIPTION**
socket() creates an endpoint for communication and returns a descriptor.

---

**DESCRIPTION**

The **open()** system call opens the file specified by pathname. If the specified file does not exist, it may optionally (if O_CREAT is specified in flags) be created by open().

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
Descriptor Table

For each Process

OS stores a table, per process, of descriptors

http://www.learnlinux.org.za/courses/build/shell-scripting/ch01s04.html
socket()

For each Process

int sock = socket(AF_INET, SOCK_STREAM, 0);

- socket() returns a socket descriptor
- Indexes into table

Kernel

stdin  stdout  stderr
socket()

For each Process

```c
int sock = socket(AF_INET,
                  SOCK_STREAM, 0);
```

OS stores details of the socket, connection, and pointers to buffers

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer ☐, Receive buffer ☐

stdin   stdout   stderr

Kernel
socket()

For each Process

```c
int sock = socket(AF_INET,
    SOCK_STREAM, 0);
```

OS stores details of the socket, connection, and pointers to buffers

Buffer:
Temporary data storage location

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer , Receive buffer

Kernel
int sock = socket(AF_INET, SOCK_STREAM, 0);

**Socket Buffers**

For each Process

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

Application buffer / storage space:

```
Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer , Receive buffer
```

Kernel
Socket Buffers

For each Process

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer, Receive buffer

Application buffer / storage space:

Kernel

Internet
Socket Buffers

For each Process

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

Application buffer / storage space:

- Family: AF_INET, Type: SOCK_STREAM
- Local address: NULL, Local port: NULL
- Send buffer, Receive buffer

recv(): Move data from socket buffer to process

For each Process

Kernel

Internet

Slide 50
Socket Buffers

For each Process

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

**Application buffer / storage space:**

**Kernel**

**Internet**

**send(): Move data from process to socket buffer**

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer, Receive buffer
Socket Buffers

For each Process

\[
\text{int sock = socket(AF_INET, SOCK_STREAM, 0);}\
\]

Application buffer / storage space:

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer , Receive buffer

Free space?  Is data here?

Challenge: Your process does NOT know what is stored here!

Kernel
What should we do if the receive socket buffer is empty? If it has 100 bytes?

For each Process

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);  // r_buf (size 200)
(assume we connect()ed here...)
int recv_val = recv(sock, r_buf, 200, 0);
```

Two Scenarios:

- Socket buffer
  - Empty
  - 100 bytes

Kernel
What should we do if the receive socket buffer is empty? If it has 100 bytes?

For each Process

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
(assume we connect()ed here…) 
int recv_val = recv(sock, r_buf, 200, 0);
```

Two Scenarios:

<table>
<thead>
<tr>
<th></th>
<th>Empty</th>
<th>100 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Block</td>
<td>Block</td>
</tr>
<tr>
<td>B</td>
<td>Block</td>
<td>Copy 100 bytes</td>
</tr>
<tr>
<td>C</td>
<td>Copy 0 bytes</td>
<td>Block</td>
</tr>
<tr>
<td>D</td>
<td>Copy 0 bytes</td>
<td>Copy 100 bytes</td>
</tr>
<tr>
<td>E</td>
<td>Something else</td>
<td></td>
</tr>
</tbody>
</table>
What should we do if the receive socket buffer is empty? If it has 100 bytes?

For each Process

```
int sock = socket(AF_INET, SOCK_STREAM, 0);  
(int) sock = socket(AF_INET, SOCK_STREAM, 0);  
(assume we connect()ed here...)
int recv_val = recv(sock, r_buf, 200, 0);
```

Two Scenarios:

<table>
<thead>
<tr>
<th></th>
<th>Empty</th>
<th>100 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Block</td>
<td>Block</td>
</tr>
<tr>
<td>B</td>
<td>Block</td>
<td>Copy 100 bytes</td>
</tr>
<tr>
<td>C</td>
<td>Copy 0 bytes</td>
<td>Block</td>
</tr>
<tr>
<td>D</td>
<td>Copy 0 bytes</td>
<td>Copy 100 bytes</td>
</tr>
<tr>
<td>E</td>
<td>Something else</td>
<td></td>
</tr>
</tbody>
</table>

Socket buffer

- Empty
- 100 bytes

Kernel
What should we do if the send socket buffer is full?
If it has 100 bytes?

For each Process

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
// (assume we connect()ed here...)
int send_val = send(sock, r_buf, 200, 0);
```

Two Scenarios:

- Socket buffer
  - Full
  - 100 bytes

Kernel
What should we do if the send socket buffer is full? If it has 100 bytes?

For each Process

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);  // s_buf (size 200)
(assume we connect()ed here...)
int send_val = send(sock, r_buf, 200, 0);
```

Two Scenarios:

<table>
<thead>
<tr>
<th></th>
<th>Full</th>
<th>100 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Return 0</td>
<td>Copy 100 bytes</td>
</tr>
<tr>
<td>B</td>
<td>Block</td>
<td>Copy 100 bytes</td>
</tr>
<tr>
<td>C</td>
<td>Return 0</td>
<td>Block</td>
</tr>
<tr>
<td>D</td>
<td>Block</td>
<td>Block</td>
</tr>
<tr>
<td>E</td>
<td>Something else</td>
<td></td>
</tr>
</tbody>
</table>

Socket buffer

- Full
- 100 bytes

Kernel
Blocking Implications

recv()

- Do not assume that you will recv() all of the bytes that you ask for.
- Do not assume that you are done receiving.
- Always receive in a loop!*

send()

- Do not assume that you will send() all of the data you ask the kernel to copy.
- Keep track of where you are in the data you want to send.
- Always send in a loop!*

* Unless you’re dealing with a single byte, which is rare.
ALWAYS check send()/recv() return values!

When recv() returns a non-zero number of bytes always call recv() again until:

- the server closes the socket,
- or you’ve received all the bytes you expect.
ALWAYS check send() / recv() return values!

When recv() returns a non-zero number of bytes always call recv() again until:

- In the case of your web client: keep receiving until the server closes the socket.
ALWAYS check send() / recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown
recv(sock, data, 200, 0);
ALWAYS check send()/recv() return values!

• E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown
recv(sock, data, 200, 0);

Data received = 50
Remaining buffer size = 150
ALWAYS check send() / recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

```c
recv(sock, data, 200, 0);
```

Data size to receive = unknown

```
data received = 50
Remaining buffer size = 150
```

// Receive remaining bytes from offset of 50
ALWAYS check send() / recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown
recv(sock, data, 200, 0);

Data received = 50
Remaining buffer size = 150

// Receive remaining bytes from offset of 50
recv(sock, data + 50, 200 – 50, 0)
Data received = ?
ALWAYS check send()/recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown
recv(sock, data, 200, 0);

Data received = 50
Remaining buffer size = 150

// Receive remaining bytes from offset of 50
recv(sock, data + 50, 200 – 50, 0)
Data received = ?

Repeat until server closes the socket. (return value = 0)
ALWAYS check send() and recv()'s return value!

• When send() /recv() return value is less than the data size, **you are responsible for sending/receiving the rest.**

Data sent: 0
Data to send: 130
send(sock, data, 130, 0);

Data sent: 60
Data to send: 130

// what should your next send call look like?
send(...)
ALWAYS check send() return value!

- When send() return value is less than the data size, you are responsible for sending the rest.

Data sent: 0
Data to send: 130
send(sock, data, 130, 0);

Data sent: 60
Data to send: 130
// Copy the 70 bytes starting from offset 60.
send(sock, data + 60, 130 - 60, 0);
ALWAYS check send() return value!

- When send() return value is less than the data size, you are responsible for sending the rest.

```
Data sent: 0
Data to send: 130
send(sock, data, 130, 0);
```

```
Data sent: 60
Data to send: 130
// Copy the 70 bytes starting from offset 60.
send(sock, data + 60, 130 - 60, 0);
```

Repeat until all bytes are sent. (data_sent == data_to_send)
Blocking Summary

send()
• Blocks when socket buffer for sending is full
• Returns less than requested size when buffer cannot hold full size

recv()
• Blocks when socket buffer for receiving is empty
• Returns less than requested size when buffer has less than full size

Always check the return value!