Networked Applications: Sockets

Topics
- Programmer’s view of the Internet
- Sockets interface

End System: Computer on the ‘Net

Internet

Also known as a “host”...
Clients and Servers

Client program
- Running on end host
- Requests service
- E.g., Web browser

Server program
- Running on end host
- Provides service
- E.g., Web server

GET /index.html

"Site under construction"

Client-Server Communication

Client “sometimes on”
- Initiates a request to the server when interested
- E.g., Web browser on your laptop or cell phone
- Doesn’t communicate directly with other clients
- Needs to know the server’s address

Server is “always on”
- Services requests from many client hosts
- E.g., Web server for the www.cnn.com web site
- Doesn’t initiate contact with the clients
- Needs a fixed, well-known address
Client and Server Processes

Program vs. process
- Program: collection of code
- Process: a running program on a host

Communication between processes
- Same end host: inter-process communication
  - Governed by the operating system on the end host
- Different end hosts: exchanging messages
  - Governed by the network protocols

Client and server processes
- Client process: process that initiates communication
- Server process: process that waits to be contacted

Delivering the Data: Division of Labor

Network
- Deliver data packet to the destination host
- Based on the destination IP address

Operating system
- Deliver data to the destination socket
- Based on the destination port number

Application
- Read data from and write data to the socket
- Interpret the data (e.g., render a Web page)
A Programmer’s View of the Internet

1. Hosts are mapped to a set of 32-bit IP addresses.
   - 128.2.203.179

2. The set of IP addresses is mapped to a set of identifiers called Internet domain names.
   - 128.2.203.179 is mapped to www.cs.cmu.edu

3. Internet sockets are communication endpoints.

4. A process on one Internet host can communicate with a process on another Internet host over a connection.

Internet Sockets

Sending message from one process to another
- Message must traverse the underlying network

Process sends and receives through a “socket”
- In essence, the doorway leading in/out of the house

Socket as an Application Programming Interface
- Supports the creation of network applications
Using Ports to Identify Services

Client host 128.2.194.242

Service request for 128.2.194.242:80 (i.e., the Web server)

Web server (port 80)

Echo server (port 7)

Service request for 128.2.194.242:7 (i.e., the echo server)

Web server (port 80)

Echo server (port 7)

Internet Connections

Clients and servers communicate by sending streams of bytes over connections.

Connections are point-to-point, full-duplex (2-way communication), and reliable.

Client socket address 128.2.194.242:51213

Server socket address 208.216.181.15:80

Client host address 128.2.194.242

Server host address 208.216.181.15

Note: 51213 is an ephemeral port allocated by the kernel

Note: 80 is a well-known port associated with Web servers
Knowing What Port Number To Use

Popular applications have well-known ports
- E.g., port 80 for Web and port 25 for e-mail
- See [http://www.iana.org/assignments/port-numbers](http://www.iana.org/assignments/port-numbers)

Well-known vs. ephemeral ports
- Server has a well-known port (e.g., port 80)
  - Between 0 and 1023
- Client picks an unused ephemeral (i.e., temporary) port
  - Between 1024 and 65535

Uniquely identifying the traffic between the hosts
- Two IP addresses and two port numbers
- Underlying transport protocol (e.g., TCP or UDP)

Port Numbers are Unique on Each Host

Port number uniquely identifies the socket
- Cannot use same port number twice with same address
- Otherwise, the OS can’t demultiplex packets correctly

Operating system enforces uniqueness
- OS keeps track of which port numbers are in use
- Doesn’t let the second program use the port number

Example: two Web servers running on a machine
- They cannot both use port “80”, the standard port #
- So, the second one might use a non-standard port #
UNIX Socket API

Socket interface
- Originally provided in Berkeley UNIX
- Later adopted by all popular operating systems
- Simplifies porting applications to different OSes

In UNIX, everything is like a file
- All input is like reading a file
- All output is like writing a file
- File is represented by an integer file descriptor

API implemented as system calls
- E.g., connect, read, write, close, …

Typical Client Program

Prepare to communicate
- Create a socket
- Determine server address and port number
- Initiate the connection to the server

Exchange data with the server
- Write data to the socket
- Read data from the socket
- Do stuff with the data (e.g., render a Web page)

Close the socket
Servers Differ From Clients

Passive open
- Prepare to accept connections
- … but don’t actually establish
- … until hearing from a client

Hearing from multiple clients
- Allowing a backlog of waiting clients
- ... in case several try to communicate at once

Create a socket for each client
- Upon accepting a new client
- … create a new socket for the communication

Typical Server Program

Prepare to communicate
- Create a socket
- Associate local address and port with the socket

Wait to hear from a client (passive open)
- Indicate how many clients-in-waiting to permit
- Accept an incoming connection from a client

Exchange data with the client over new socket
- Receive data from the socket
- Do stuff to handle the request (e.g., get a file)
- Send data to the socket
- Close the socket

Repeat with the next connection request
Client Creating a Socket: socket()

Operation to create a socket
- `int socket(int domain, int type, int protocol)`
- Returns a descriptor (or handle) for the socket
- Originally designed to support any protocol suite

Domain: protocol family
- PF_INET for the Internet

Type: semantics of the communication
- SOCK_STREAM: reliable byte stream
- SOCK_DGRAM: message-oriented service

Protocol: specific protocol
- UNSPEC: unspecified
- (PF_INET and SOCK_STREAM already implies TCP)

Client: Learning Server Address/Port

Server typically known by name and service
- E.g., “www.cnn.com” and “http”

Need to translate into IP address and port #
- E.g., “64.236.16.20” and “80”

Translating the server’s name to an address
- `struct hostent *gethostbyname(char *name)`
- Argument: host name (e.g., “www.cnn.com”)
- Returns a structure that includes the host address

Identifying the service’s port number
- `struct servent *getservbyname(char *name, char *proto)`
- Arguments: service (e.g., “ftp”) and protocol (e.g., “tcp”)
Client: Connecting Socket to the Server

Client contacts the server to establish connection

- Associate the socket with the server address/port
- Acquire a local port number (assigned by the OS)
- Request connection to server, who will hopefully accept

Establishing the connection

- `int connect(int sockfd, struct sockaddr *server_address, socklen_t addrlen)`
- Arguments: socket descriptor, server address, and address size
- Returns 0 on success, and -1 if an error occurs

Client: Sending and Receiving Data

Sending data

- `ssize_t write(int sockfd, void *buf, size_t len)`
- Arguments: socket descriptor, pointer to buffer of data to send, and length of the buffer
- Returns the number of characters written, and -1 on error

Receiving data

- `ssize_t read(int sockfd, void *buf, size_t len)`
- Arguments: socket descriptor, pointer to buffer to place the data, size of the buffer
- Returns the number of characters read (where 0 implies “end of file”), and -1 on error

Closing the socket

- `int close(int sockfd)`
Server: Server Preparing its Socket

Server creates a socket and binds address/port
- Server creates a socket, just like the client does
- Server associates the socket with the port number
  (and hopefully no other process is already using it!)

Create a socket
- `int socket(int domain, int type, int protocol)`

Bind socket to the local address and port number
- `int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)`
- Arguments: socket descriptor, server address, address length
- Returns 0 on success, and -1 if an error occurs

Server: Allowing Clients to Wait

Many client requests may arrive
- Server cannot handle them all at the same time
- Server could reject the requests, or let them wait

Define how many connections can be pending
- `int listen(int sockfd, int backlog)`
- Arguments: socket descriptor and acceptable backlog
- Returns a 0 on success, and -1 on error

What if too many clients arrive?
- Some requests don’t get through
- The Internet makes no promises…
- And the client can always try again
Server: Accepting Client Connection

Now all the server can do is wait…
- Waits for connection request to arrive
- Blocking until the request arrives
- And then accepting the new request

Accept a new connection from a client
- `int accept(int sockfd, struct sockaddr *addr, socketlen_t *addrlen)`
- Arguments: socket descriptor, structure that will provide client address and port, and length of the structure
- Returns descriptor for a new socket for this connection

Server: One Request at a Time?

Serializing requests is inefficient
- Server can process just one request at a time
- All other clients must wait until previous one is done

May need to time share the server machine
- Alternate between servicing different requests
  - Do a little work on one request, then switch to another
  - Small tasks, like reading HTTP request, locating the associated file, reading the disk, transmitting parts of the response, etc.
- Or, start a new process to handle each request
  - Allow the operating system to share the CPU across processes
- Or, some hybrid of these two approaches
Client and Server: Cleaning House

Once the connection is open
- Both sides and read and write
- Two unidirectional streams of data
- In practice, client writes first, and server reads
- … then server writes, and client reads, and so on

Closing down the connection
- Either side can close the connection
- … using the close() system call

What about the data still “in flight”
- Data in flight still reaches the other end
- So, server can close() before client finishing reading

One Annoying Thing: Byte Order

Hosts differ in how they store data
- E.g., four-byte number (byte3, byte2, byte1, byte0)

Little endian (“little end comes first”) ← Intel PCs!!!
- Low-order byte stored at the lowest memory location
- Byte0, byte1, byte2, byte3

Big endian (“big end comes first”)
- High-order byte stored at lowest memory location
- Byte3, byte2, byte1, byte 0

Makes it more difficult to write portable code
- Client may be big or little endian machine
- Server may be big or little endian machine
**Endian Example: Where is the Byte?**

<table>
<thead>
<tr>
<th>8 bits memory</th>
<th>16 bits Memory</th>
<th>32 bits Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+1</td>
<td>+3</td>
</tr>
<tr>
<td></td>
<td>+0</td>
<td>+2</td>
</tr>
<tr>
<td></td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td>+0</td>
<td>+2</td>
</tr>
<tr>
<td></td>
<td>+0</td>
<td>+3</td>
</tr>
<tr>
<td></td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td>+0</td>
<td>+2</td>
</tr>
<tr>
<td></td>
<td>+0</td>
<td>+3</td>
</tr>
</tbody>
</table>

**Big-Endian**

- 1000: 78
- 1001: 
- 1002: 
- 1003: 

**Little-Endian**

- 1000: 78
- 1001: 
- 1002: 
- 1003: 

**IP is Big Endian**

But, what byte order is used “on the wire”

- That is, what do the network protocol use?

The Internet Protocols picked one convention

- IP is big endian (aka “network byte order”)

Writing portable code require conversion

- Use htons() and htonl() to convert to network byte order
- Use ntohs() and ntohl() to convert to host order

Hides details of what kind of machine you’re on

- Use the system calls when sending/receiving data structures longer than one byte
Why Can’t Sockets Hide These Details?

Dealing with endian differences is tedious
- Couldn’t the socket implementation deal with this
- … by swapping the bytes as needed?

No, swapping depends on the data type
- Two-byte short int: (byte 1, byte 0) vs. (byte 0, byte 1)
- Four-byte long int: (byte 3, byte 2, byte 1, byte 0) vs. (byte 0, byte 1, byte 2, byte 3)
- String of one-byte charters: (char 0, char 1, char 2, …) in both cases

Socket layer doesn’t know the data types
- Sees the data as simply a buffer pointer and a length
- Doesn’t have enough information to do the swapping

The Web as an Example
Client/Server Application
**The Web: URL, HTML, and HTTP**

**Uniform Resource Locator (URL)**
- A pointer to a "black box" that accepts request methods
- Formatted string with protocol (e.g., `http`), server name (e.g., `www.cnn.com`), and resource name (e.g., `coolpic.jpg`)

**HyperText Markup Language (HTML)**
- Representation of hypertext documents in ASCII format
- Format text, reference images, embed hyperlinks
- Interpreted by Web browsers when rendering a page

**HyperText Transfer Protocol (HTTP)**
- Client-server protocol for transferring resources
- Client sends request and server sends response

---

**Example: HyperText Transfer Protocol**

```
GET /~mdamian/csc2405/ HTTP/1.1
Host: www.csc.villanova.edu
<CRLF>
```

**Request**

```
HTTP/1.1 200 OK
Date: Mon, 16 Feb 2009 08:09:03 GMT
Server: Apache/1.3.27 (Unix)
Last-Modified: Sun, 26 Aug 2007 15:45:05 GMT
Content-Type: text/plain
Content-Length: 259
<CRLF>
...
```

**Response**
Components: Clients, Proxies, Servers

Clients
- Send requests and receive responses
- Browsers, spiders, and agents

Servers
- Receive requests and send responses
- Store or generate the responses

Proxies (see “HTTP Proxy” assignment!)
- Act as a server for the client, and a client to the server
- Perform extra functions such as anonymization, logging, blocking of access, caching, etc.

Example Client: Web Browser

Generating HTTP requests
- User types URL, clicks a hyperlink, or selects bookmark
- User clicks “reload”, or “submit” on a Web page
- Automatic downloading of embedded images

Layout of response
- Parsing HTML and rendering the Web page
- Invoking helper applications (e.g., Acrobat, PowerPoint)

Maintaining a cache
- Storing recently-viewed objects
- Checking that cached objects are fresh
Client: Typical Web Transaction

User clicks on a hyperlink

Browser learns the IP address
- Invokes gethostbyname(www.cnn.com)
- And gets a return value of 64.236.16.20

Browser creates socket and connects to server
- OS selects an ephemeral port for client side
- Contacts 64.236.16.20 on port 80

Browser writes the HTTP request into the socket
- "GET /index.html HTTP/1.1
  Host: www.cnn.com
  <CRLF>"

In Fact, Try This at a UNIX Prompt...

telnet www.cnn.com 80
GET /index.html HTTP/1.1
Host: www.cnn.com
<CRLF>

And you'll see the response...
Client: Typical Web Transaction (Cont)

Browser parses the HTTP response message
- Extract the URL for each embedded image
- Create new sockets and send new requests
- Render the Web page, including the images

Opportunities for caching in the browser
- HTML file
- Each embedded image
- IP address of the Web site

Web Server

Web site vs. Web server
- **Web site**: collections of Web pages associated with a particular host name
- **Web server**: program that satisfies client requests for Web resources

Handling a client request
- Accept the socket
- Read and parse the HTTP request message
- Translate the URL to a filename
- Determine whether the request is authorized
- Generate and transmit the response
Web Proxy

See assignment.