Networked Applications: Refresh on Sockets

Topics

- Programmer’s view of the Internet
- Sockets interface

End System: Computer on the ‘Net

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
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

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

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

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

Connection socket pair (128.2.194.242:51213, 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

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 #
- E.g., http://www.cnn.com:8080
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

Java Sockets Interface
C Sockets Interface

More complex

Diagram:
- Server
  - socket()
  - bind()
  - listen()
  - accept()
  - read()
  - process request
  - write()

- Client
  - socket()
  - connect()
  - write()
  - send request
  - establish connection
  - send response
  - read()
C Sockets Interface

socket
- First step in making a network connection: create a socket

bind
- The server binds the socket so it can listen

listen
- The server listens on the socket for connection request from client

connect
- The client tries to connect to the server (connection request sent over the socket)

accept
- The accepts the connection request: connection established

Help with C Sockets Interface

See(nethelp.c, nethelp.h)
(available on class website)
**Echo Client: Main Routine**

```c
#include "nethelp.h"

/* usage: ./echoclient host port */
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];

    host = argv[1];
    port = atoi(argv[2]);

    clientfd = open_clientfd(host, port);
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        write(clientfd, buf, strlen(buf));
        readline(clientfd, buf, MAXLINE);
        write(1, buf, n);
    }
    close(clientfd);
    exit(0);
}
```

**Echo Server: Main Routine**

```c
int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr clientaddr;

    port = atoi(argv[1]); /* the server listens on a port passed
                           * on the command line */
    clientlen = sizeof(clientaddr);
    listenfd = open_listenfd(port);

    while (1) {
        connfd = accept(listenfd, &clientaddr, &clientlen);
        echo(connfd); /* Service client */
        close(connfd); /* Close connection with client */
    }
}
Echo Server: Main Loop

The server loops endlessly, waiting for connection requests, then reading input from the client, and echoing the input back to the client.

```c
main() {
    /* create and configure the listening socket */
    while(1) {
        /* accept(): wait for a connection request */
        /* echo(): read and echo input lines from client til EOF */
        /* close(): close the connection */
    }
}
```

Echo Server: accept

`accept()` blocks waiting for a connection request.

```c
int listenfd; /* listening descriptor */
int connfd; /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
```

`accept` returns a connected descriptor (`connfd`) with the same properties as the listening descriptor (`listenfd`):

- Returns when the connection between client and server is created and ready for I/O transfers.
- All I/O with the client will be done via the connected socket.

`accept` also fills in client’s IP address.
Echo Server: `accept` Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`.

2. Client makes connection request by calling and blocking in `connect`.

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`.

Connected vs. Listening Descriptors

Listening descriptor
- End point for client connection requests.
- Created once and exists for lifetime of the server.

Connected descriptor
- End point of the connection between client and server.
- A new descriptor is created each time the server accepts a connection request from a client.
- Exists only as long as it takes to service client.

Why the distinction?
- Allows for concurrent servers that can communicate over many client connections simultaneously.
  - E.g., Each time we receive a new request, we create a new thread to handle the request.
Echo Server: echo

The server reads and echoes text lines until EOF (end-of-file) is encountered.

- EOF notification caused by client calling `close(clientfd)`.
- IMPORTANT: EOF is a condition, not a particular data byte.

```c
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];

    while((n = readline(connfd, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", n);
        write(connfd, buf, n);
    }
}
```

Testing Servers Using `telnet`

The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections

- Our simple echo server
- Web servers
- Mail servers

Usage:

- `bash$ telnet <host> <portnumber>`
  - Creates a connection with a server running on `<host>` and listening on port `<portnumber>`.
Testing the Echo Server With `telnet`

bash$ echoserver 5000

In a separate terminal window:

bash$ telnet tanner 5000
Trying 128.2.222.85...
Connected to tanner.csc.villanova.edu
Escape character is '^]'.
123
123
Connection closed by foreign host.

bash$

---

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

Implement multi-threaded servers

- Start a new thread to handle each request
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
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 characters: (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 (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/CSC8560 / HTTP/1.1
Host: www.csc.villanova.edu
<CRLF>

Request

HTTP/1.1 200 OK
Date: Mon, 27 Aug 2012 08:09:03 GMT
Server: Apache/1.3.27 (Unix)
Last-Modified: Sun, 26 Aug 2012 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
<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.