CS434/534: Topics in Network Systems

Network Server Design: TCP Socket API, Basic Client/Server Structure; HTTP Server; Thread Server

Sept. 16, 2021

Y. Richard Yang
Computer Science Department, Yale University
Email: yry@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/
Admin

- Homepage updated
- Programming project 1 (part 1) will be posted later today
- Schedule a time so that I can meet with each student (or a group)
Recap: HTTP

- The HTTP protocol is designed to support a generic interface protocol
  - Stateless
    - Using cookies to support stateful applications
  - Universal resource
  - State transfer as universal operators to manipulate resources
    - Content-length, Transfer encoding
    - Conditional transfer
    - Intermediates transfer
  - Resource can have multiple representations
    - Self-descriptive messages allowing flexible, extensive representation
    - Content negotiation to select representation
Recap: The Basic Socket API

- Two commonly used transport services exposed through the socket API:
  - connectionless (UDP) socket
  - connection-oriented (TCP) socket

- Application processes use the socket API to bind to port numbers:
  - email (SMTP) server port number 25, web server port number 80

- Outgoing packets sent by the socket are labelled with src/dst port numbers.

- Incoming packets are multiplexed to processes according to port binding.
  - Multiplexing is almost always based on best binding match.
Recap: UDP Socket Mux

Common network systems mechanism: multiplexing based on best flow-rule matching.
Outline

- Admin and recap
- Networked systems (network applications, network-based systems)
  - Overview
  - HyperText Transfer Protocol (HTTP/1.x)
  - High-performance network server design & imple
    - Overview
      - Transport services, protocols
      - Basic Socket API and client/server examples
        » UDP socket and multiplexing basic concept
        » TCP socket and connection management
TCP Socket Design: Starting w/ UDP

Issue: If a single socket, data can be mixed, but TCP is designed to provide a connection abstraction: server reads an ordered sequence of bytes from each individual client.

Issue 2: How to notify server that a new client is connected?
TCP Socket Big Picture

- Welcome socket: the waiting room
- connSocket: the operation room
Client/server Socket Workflow: TCP

Server (running on hostid)  Client

create socket, port=x, for incoming request:
welcomeSocket = ServerSocket(x)

wait for incoming connection request
connectionSocket = welcomeSocket.accept()

read request from connectionSocket
write reply to connectionSocket
close connectionSocket

create socket, connect to hostid, port=x
clientSocket = Socket()

send request using clientSocket
read reply from clientSocket
close clientSocket

TCP connection setup

Client process
Server process
Client/server Socket Workflow: TCP (C version)

**Server**

1. `welcomeSocket = socket()`: create socket
2. `bind(welcomeSocket, ...)`: specify socket address/port
3. `listen(welcomeSocket, ...)`: specify that socket `welcomeSocket` is a listening socket
4. `connectionSocket = accept(welcomeSocket, ...)`: get a connected connection from the queue for socket `welcomeSocket`; create a new socket identified by `connectionSocket`
5. `read() / write()`: do IO on socket `connectionSocket`
6. `close(connectionSocket)`: done

**Client**

1. `clientSocket = socket()`: create socket
2. `bind(clientSocket)`: specify socket address
   
   (optional)
3. `connect(clientSocket, serverAddr, serverPort)`: initialize TCP handshake to server; return until TCP handshake is done
4. `read() / write()`: do IO on socket `clientSocket`
5. `close(clientSocket)`: done

See [https://man7.org/linux/man-pages/man2/socket.2.html](https://man7.org/linux/man-pages/man2/socket.2.html) for Linux Socket API (C)
Basic TCP Socket API:
ServerSocket (java)

- `ServerSocket()` creates an unbound server socket.
- `ServerSocket(int port)` creates a server socket, bound to the specified port.
- `ServerSocket(int port, int backlog)` creates a server socket and binds it to the specified local port number, with the specified backlog.
- `ServerSocket(int port, int backlog, InetAddress bindAddr)` creates a server with the specified port, listen backlog, and local IP address to bind to.

- `bind(SocketAddress endpoint)` binds the `ServerSocket` to a specific address (IP address and port number).
- `bind(SocketAddress endpoint, int backlog)` binds the `ServerSocket` to a specific address (IP address and port number).

- `Socket accept()` listens for a connection to be made to this socket and accepts it.

- `InputStream getInputStream()` returns an input stream for this socket.
- `OutputStream getOutputStream()` returns an output stream for this socket.

- `close()` closes this socket.
Basic TCP Socket API:
(Client)Socket

- `Socket(InetAddress address, int port)`
  creates a stream socket and connects it to the specified port number at the specified IP address.
- `Socket(InetAddress address, int port, InetAddress localAddr, int localPort)`
  creates a socket and connects it to the specified remote address on the specified remote port.
- `Socket(String host, int port)`
  creates a stream socket and connects it to the specified port number on the named host.

- `bind(SocketAddress bindpoint)`
  binds the socket to a local address.

- `connect(SocketAddress endpoint)`
  connects this socket to the server.
- `connect(SocketAddress endpoint, int timeout)`
  connects this socket to the server with a specified timeout value.

- `InputStream getInputStream()`
  returns an input stream for this socket.
- `OutputStream getOutputStream()`
  returns an output stream for this socket.

- `close()`
  closes this socket.
Example client-server app:

1) client reads line from standard input (inFromUser stream), sends to server via socket (outToServer stream)
2) server reads line from socket
3) server converts line to uppercase, sends back to client
4) client reads, prints modified line from socket (inFromServer stream)
Exercise

- %netstat -a -n -p tcp
- %java TCPServer
- %java TCPClient localhost 6789
- %netstat -a -n -p tcp
- Q: How many sockets are created?
**Exercise: TCP Socket Multiplexing**

**Q:** How to decide where to put an incoming TCP packet?

**A:** Packet demultiplexing in TCP is based on **best-match four tuples:**
\[(dst \, addr, \, dst \, port, \, src \, addr, \, src \, port)\]
Typical TCP Server Program Flow

1. `Create ServerSocket(6789)`
2. `connSocket = accept()`
3. `read request from connSocket`
4. `Serve the request`
5. `close connSocket`

-Welcome socket: the waiting room
-connSocket: the operation room
Summary: Basic Socket Programming

- They are relatively straightforward
  - UDP: DatagramSocket
  - TCP: ServerSocket, Socket

- The basic functions of socket are
  - multiplexing/demultiplexing to application processes
    - UDP uses (dst IP, port)
    - TCP uses (src IP, src port, dst IP, dst port)
  - providing i/o streams to read/write data
    - more discussions later
Outline

- Admin and recap
- Networked systems (network applications, network-based systems)
  - Overview
  - HyperText Transfer Protocol (HTTP/1.x)
  - High-performance network server design & imple
    - Overview
      - Transport services, protocols
      - Basic Socket API, client/server programming
      - Basic HTTP server structure
Design Exercise: HTTP Server Design

```
GET /somedir/page.html HTTP/1.0
Host: www.somechool.edu
Connection: close
User-Agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Accept-Language: en

HTTP/1.0 200 OK
Date: Wed, 23 Jan 2008 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 ...
Content-Length: 6821
Content-Type: text/html

data data data data data ...
```

POST /path/script.cgi HTTP/1.0
User-Agent: MyAgent
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

item1=A&item2=B
Example Code

- See BasicHTTPServer/HTTPServer.java

- Try using three methods to fetch index.html
  - `telnet localhost 6789`
  - `curl`
  - Using a browser
URL as Content Abstraction

Create ServerSocket(6789)

connSocket = accept()

read request from connSocket

Map URL to file

Read from file/write to connSocket

close connSocket

It does not have to be a static file
There are multiple approaches to implement dynamic resources

- **Embedded interpreters**
  - http server includes an interpreter for the type of pages
  - interpreter runs inside the server

- **Invoke external programs (http server is agnostic to the external program execution)**
  - Benefits?

http://www.cs.yale.edu/index.shtml
http://www.cs.yale.edu/cgi-bin/ureserve.pl
http://www.google.com/search?q=Yale&sourceid=chrome
CGI: Invoking External Programs

- **Two issues**
  - **Input:** send request related info to external program
    - HTTP request info (e.g., HTTP req header, client input such as form input) from server to the external program
  - **Output:** Get external program output to server (to client)
Example: The CGI Standard

- **CGI standard:** [http://www.ietf.org/rfc/rfc3875](http://www.ietf.org/rfc/rfc3875)
  - Starts the executable as a child process
    - [https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html](https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html)
  - Input: Passes HTTP request info in environment variables and stdin (request body, if any)
    - Example
      - GET /search?q=Yale&sourceid=chrome HTTP/1.0
      - setup environment variables, in particular
        - $QUERY_STRING=q=Yale&sourceid=chrome
      - See RFC3875 for complete list of required variables
  - Output: Redirects output of the child process to the socket
# CGI Example

- **http://localhost/BasicHTTPServer/cgi/price.cgi?appl**

```perl
#!/usr/bin/perl -w

$company = $ENV{'QUERY_STRING'};
print "Content-Type: text/html\n"
print "\n"
print "<html>"
print "<h1>Hello! The price is ";
if ($company =~ /appl/) {
    my $var_rand = rand();
    print 450 + 10 * $var_rand;
} else {
    print "150";
}
print "</h1>";
print "</html>";
```

https://httpd.apache.org/docs/2.2/howto/htaccess.html

(CGI Example)
Discussions

- Major performance problem of standard CGI?
- How may you address it?
Example: fastcgi

- Make the external program a server, so that the http server does not start a new external program each time
- HTTP server becomes the client of the server
- Design a client/server protocol between the HTTP server and the external server
Fastcgi example

Nginx example

```
location ~ .php$ {
    include /etc/nginx/fcgi_params; #or whatever you named it
    fastcgi_pass 127.0.0.1:9000;
}
```

Fastcgi protocol

typedef struct {
    unsigned char version;
    unsigned char type;
    unsigned char requestIdB1;
    unsigned char requestIdB0;
    unsigned char contentLengthB1;
    unsigned char contentLengthB0;
    unsigned char paddingLength;
    unsigned char reserved;
    unsigned char contentData[contentLength];
    unsigned char paddingData[paddingLength];
} FCGI_Record;

Summary

- Basic HTTP server structure follows basic server structure
- General pattern: Many network servers introduce internal protocols to realize the complete system, and the internal protocols are often client server as well
Outline

- Admin and recap
- Networked systems (network applications, network-based systems)
  - Overview
  - HyperText Transfer Protocol (HTTP/1.x)
  - High-performance network server design & imple
    - Overview
      - Transport services, protocols
      - Basic Socket API, client/server programming
      - Basic HTTP server structure
      - Key issue of basic server structure
Question: HTTP Server Execution

Create ServerSocket(6789)

connSocket = accept()

read request from connSocket

Map URL to file content

Read from file/ write to connSocket

close connSocket

Discussion: what does each step do and how long does it take?
Key Issue of Server Design

Accept Client Connection

Read Request

Find File

Send Response Header

Read File

Send Data

may block waiting on network

may block waiting on disk I/O
Exercise/Demo

- Start BasicHTTPServer/HTTPServer
- Start client 1 (TCPClient or telnet) but give partial input
- Start client 2 (e.g., browser)
Outline

- Admin and recap
- Networked systems (network applications, network-based systems)
  - Overview
  - HyperText Transfer Protocol (HTTP/1.x)
  - High-performance network server design & imple
    - Overview
    - Thread-based server design
Multi-Thread/Process Servers

- Idea: introduce independent execution streams, e.g.,
  - A thread is a sequence of instructions which may execute in parallel with other threads
  - When a blocking operation happens, only the flow (thread) performing the operation is blocked
Background: Thread vs Process

A computer

Processes do not share data

int x;
foo() {
  ...
  x...
}

Threads share data within a process

int x;
foo() {
  ...
  x...
}

Processes (e.g., JVM’s)

Threads

int x;
foo() {
  ...
  x...
}
Every Java application has at least one thread

- The “main” thread, started by the JVM to run the application’s main() method
- Most JVMs use POSIX threads to implement Java threads

```java
class RequestHandler extends Thread {
    RequestHandler(Socket connSocket) {
        ...
    }
    public void run() {
        // process request
    }
    ...
}
Thread t = new RequestHandler(connSocket);
t.start();
```

```java
class RequestHandler implements Runnable {
    RequestHandler(Socket connSocket) {
        ...
    }
    public void run() {
        // process request
    }
    ...
}
RequestHandler rh = new RequestHandler(connSocket);
Thread t = new Thread(rh);
t.start();
```
Exercise: a Multi-threaded HTTPServer

- Turn HTTPServer into a multithreaded HTTPServer by creating a thread for each accepted request

Offline: Examples using pthread in C:
https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
Exercise/Demo (Offline)

- Start BasicHTTPServer/HTTPServer
- Start client 1 (TCPClient or telnet) but give partial input
- Start client 2 (e.g., browser)
Problem of Per-Request Thread (Instantiation)

- At any time, #threads (and resource usage) need to be equal to the number of concurrent requests pending in the system
  - Long running (but idle) connections => many concurrent requests [example app with many long running but mostly idle connections?]
- High thread creation/deletion overhead → throughput meltdown → response time explosion

Q: given avg response time (R) and connection arrival rate (λ), how many threads are active on avg?
Handy Tool: Little’s Law (1961)

- For any system with no or (low) loss.
- Assume
  - mean arrival rate $\lambda$, mean time $R$ at system, and mean number $Q$ of requests at system
- Then relationship between $Q$, $\lambda$, and $R$:

$$Q = \lambda R$$

Example: Yale College admits 1500 students each year, and mean time a student stays is 4 years, how many students are enrolled?
Little’s Law

\[ Q = \lambda R \]

\[ \lambda = \frac{A}{t} \quad R = \frac{\text{Area}^t}{A} \quad Q = \frac{\text{time}}{\text{Area}} \]
Next Class: ThreadPool and Non-blocking io servers

- ThreadPool server
- select (non-blocking) IO based server
- Generic server control structure
Background: TCP Server/Client Example
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        sentence = inFromUser.readLine();

        Socket clientSocket = new Socket("server.name", 6789);
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());

        Create input stream
        Create client socket, connect to server
        Create output stream, attached to socket

        System.out.println("Hello! The server replied: 
            " + sentence + " Hi there! How are you?");
    }
}
outToServer.writeBytes(sentence + '\n');

BufferedReader inFromServer =
new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();
}
OutputStream

- public abstract class OutputStream
  - public abstract void write(int b) throws IOException
  - public void write(byte[] data) throws IOException
  - public void write(byte[] data, int offset, int length) throws IOException
  - public void flush() throws IOException
  - public void close() throws IOException
**InputStream**

- public abstract class `InputStream`
  - public abstract int `read()` throws IOException
  - public int `read(byte[] input)` throws IOException
  - public int `read(byte[] input, int offset, int length)` throws IOException
  - public long `skip(long n)` throws IOException
  - public int `available()` throws IOException
  - public void `close()` throws IOException
import java.io.*;
import java.net.*;

class TCPServer {
    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;
        ServerSocket welcomeSocket = new ServerSocket(6789, 2);

        while(true) {
            Socket connectionSocket = welcomeSocket.accept();

            BufferedReader inFromClient =
                new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
        }
    }
}
**Example: Java server (TCP)**

```java
import java.io.*;
import java.net.*;

class TCPServer {
    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(6789, 2);

        while(true) {
            Socket connectionSocket = welcomeSocket.accept();
            // Wait, on welcoming socket for contact by client
        }
    }
}
```
Example: Java server (TCP): Processing

Create input stream, attached to socket

BufferedReader inFromClient =
    new BufferedReader(new
        InputStreamReader(connectionSocket.getInputStream()));

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '
';

DataOutputStream outToClient =
    new DataOutputStream(connectionSocket.getOutputStream());

outToClient.writeBytes(capitalizedSentence);
}
}
Example: Java server (TCP): Output

```java
BufferedReader inFromClient = 
    new BufferedReader(
        new InputStreamReader(connectionSocket.getInputStream()));

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';

DataOutputStream outToClient =
    new DataOutputStream(connectionSocket.getOutputStream());

outToClient.writeBytes(capitalizedSentence);
```

Create output stream, attached to socket

Write out line to socket

End of while loop, loop back and wait for another client connection
Dynamic URL; More Details
Example SSI

- See programming/examples-java-socket/BasicWebServer/ssi/index.shtml, header.shtml, …
Example SSI

- See programming/examples-java-socket/BasicWebServer/ssi/index.shtml, header.shtml, ...

- To enable ssi, need configuration to tell the web server (see conf/apache-htaccess)
  - [https://httpd.apache.org/docs/2.2/howto/htaccess.html](https://httpd.apache.org/docs/2.2/howto/htaccess.html) (Server Side Includes example)
**HTTP: POST**

- If an HTML page contains forms or parameters too large, they are sent using POST and encoded in the message body.

```
method sp URL sp version cr lf
header field name : value cr lf
...                 
header field name : value cr lf
...                 
```

Entity Body
HTTP: POST Example

POST /path/script.cgi HTTP/1.0
User-Agent: MyAgent
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

item1=A&item2=B

Example using nc:
programming/examples-java-socket/BasicWebServer/nc/
Fcgi examples

- A simple request with no data on stdin, and a successful response:

```
{FCGI_BEGIN_REQUEST, 1, {FCGI_RESPONDER, 0}}
{FCGI_PARAMS, 1, "013\002SERVER_PORT80\013\016SERVER_ADDR199.170.183.42 ... "}
{FCGI_PARAMS, 1, ""}
{FCGI_STDIN, 1, ""}
{FCGISTDOUT, 1, "Content-type: text/html\n\n<html><head> ... "}
{FCGISTDOUT, 1, ""}
{FCGISTDOUT, 1, ""}
{FCGI_END_REQUEST, 1, {0, FCGI_REQUEST_COMPLETE}}
```

- Similar to example 1, but this time with data on stdin. The Web server chooses to send the parameters using more FCGI_PARAMS records than before:

```
{FCGI_BEGIN_REQUEST, 1, {FCGI_RESPONDER, 0}}
{FCGI_PARAMS, 1, "013\002SERVER_PORT80\013\016SER"}
{FCGI_PARAMS, 1, "VER_ADDR199.170.183.42 ... "}
{FCGI_PARAMS, 1, ""}
{FCGI_PARAMS, 1, "quantity=100&item=3047936"}
{FCGI_PARAMS, 1, ""}
{FCGI_STDIN, 1, "Content-type: text/html\n\n<html><head> ... "}
{FCGISTDOUT, 1, ""}
{FCGISTDOUT, 1, ""}
{FCGI_END_REQUEST, 1, {0, FCGI_REQUEST_COMPLETE}}
```
Similar to example 1, but this time the application detects an error. The application logs a message to stderr, returns a page to the client, and returns non-zero exit status to the Web server. The application chooses to send the page using more FCGI_STDOUT records:

```c
{FCGI_BEGIN_REQUEST, 1, {FCGI_RESPONDER, 0}}
{FCGI_PARAMS,        1, "\013\002SERVER_PORT80\013\016SERVER_ADDR199.170.183.42 ... "}
{FCGI_PARAMS,        1, ""}
{FCGI_STDIN,         1, ""}
{FCGI_STDOUT,       1, "Content-type: text/html\r\n\n<ht"
{FCGI_STDERR,       1, "config error: missing SI_UID\n"
{FCGI_STDOUT,       1, "\ml><\n<head> ... "}
{FCGI_STDOUT,       1, ""}
{FCGI_STDERR,       1, ""}
{FCGI_END_REQUEST,  1, {938, FCGI_REQUEST_COMPLETE}}
```
FcgI examples

- Two instances of example 1, multiplexed onto a single connection. The first request is more difficult than the second, so the application finishes the requests out of order:

```plaintext
{FCGI_BEGIN_REQUEST, 1, {FCGI_RESPONDER, FCGI_KEEP_CONN}}
{FCGI_PARAMS, 1, "\013\002SERVER_PORT80\013\016SERVER_ADDR199.170.183.42 ... "}
{FCGI_PARAMS, 1, ""}
{FCGI_BEGIN_REQUEST, 2, {FCGI_RESPONDER, FCGI_KEEP_CONN}}
{FCGI_PARAMS, 2, "\013\002SERVER_PORT80\013\016SERVER_ADDR199.170.183.42 ... "}
{FCGI_STDIN, 1, ""}

{FCGI_STDOUT, 1, "Content-type: text/html\r\n\r\n"}
{FCGI_PARAMS, 2, ""}
{FCGI_STDIN, 2, ""}

{FCGI_STDOUT, 2, "Content-type: text/html\r\n\r\n<html><head> ... "}
{FCGI_STDOUT, 2, ""}
{FCGI_END_REQUEST, 2, {0, FCGI_REQUEST_COMPLETE}}
{FCGI_STDOUT, 1, "<html><head> ... "}
{FCGI_STDOUT, 1, ""}
{FCGI_END_REQUEST, 1, {0, FCGI_REQUEST_COMPLETE}}
```
scgi example

- scgi is another "faster" cgi, using a protocol between http server and external program

"70:" "CONTENT_LENGTH"
<00> "27" <00> "SCGI" <00>
"1" <00>
"REQUEST_METHOD" <00>
"POST" <00>
"REQUEST_URI" <00>
"/deepthought" <00> "," "What is the answer to life?"

"Status: 200 OK"
<0d 0a>
"Content-Type: text/plain" <0d 0a> "" <0d 0a> "42"

https://python.ca/scgi/protocol.txt