Project Overview & Network Programming Guide

18-345 Spring 2015
Project

• 35 % of your grade
• 2 person teams
  – Find your buddy and your team name
  – Choice of programming languages: C/C++/Java
• 4 projects over the semester
  – Build upon previous projects
  – Need your last project to move on
  – Start early! Testing will be critical...
    • Your server will eventually talk with ours over the network
Project Overview

- P2P Video-on-Demand
  - Distributed “YouTube”
- Eventually grows to a full peer-to-peer VoD system
- Apply network concepts
  - Socket/Server programming
  - Congestion control
  - Failure detection/recovery
  - Distributed hash
Video-on-demand

- Video on demand
  - View what you want whenever you want
  - Usually not ‘live’ stream

- Content serves from a single provider
  - Not necessarily from the same server
Content Distribution Network

- Service providers have many servers distributed geographically.
- Contents are duplicated from the master server to other locations.
- Content is served from a single server located near the client.
- Need to concern about what/where/when to duplicate the contents.
P2P Video-on-demand

- Peer-to-Peer network
  - All nodes provide resource (bandwidth, content)
  - Many nodes may share the same (popular) contents
- P2P Video on demand
  - Need to be able to query and locate the desired content
  - Clients may concurrently send/retrieve content to/from multiple peers
- Performance consideration
  - Content search time
  - Content retrieval time
  - Content “ready-to-view” time
  - Network bandwidth required
Serving Video over IP

Streaming Media

- The content is constantly received and processed by the client
- Small client buffer needed (client buffer may not hold the entire content)
- Protocols provide facility for jitter compensation / handle packet loss / synchronization
- Utilize multiple protocols
  - Transport:
    - Real-time Transport Protocol (RTP)
  - Control:
    - Real Time Control Protocol (RTCP)
    - Real-time Streaming Protocol (RTSP)
    - Session-initiation Protocol (SIP)
- Example
  - Flash Media Server
  - Quicktime Streaming Server
  - VLC

Progressive Download (Pseudo Streaming)

- Allows media playback while downloading the content
- Client uses content metadata to enable early playback
- Client buffer used to store the entire content
- Protocol
  - Hypertext Transfer Protocol (HTTP)
- Example
  - Youtube / Flash
  - JPEG (back when we use modem)
  - HTML 5 audio/video (use in our project)
HTTP Pseudo Streaming

- Client requests metadata part of the content
  - Usually at the beginning of the file
- Client downloads the content to its buffer and starts playing the media
  - Same as downloading any HTTP content
- Content seek provided by selectively downloading parts of the content
  - Compatible HTTP server support partial byte-range requests for the content
Our P2P VoD Daemon

- Peer status/discovery
- Content search
- Content transfer
- Error control

UI / Html

Local File

HTTP Server

Request Handler

Peer Process

Video Transfer

Project 1
Project 1: HTTP-streaming server

- HTML 5-capable browsers (Firefox, Chrome) provide built-in video clients (HTTP pseudo streaming)
- Our first task is to provide an effective way to transfer our buffered video to browsers
  - Introduction to socket & server programming
  - Partial HTTP implementation (with byte range request)
  - You should be able to run the server on a cluster machine and view your video on a laptop
- Performance index
  - Maximize number of simultaneous viewers (no. clients who can retrieve X bytes within Y seconds after connection initiation)
  - The score will heavily depend on your server design
Hypertext Transfer Protocol

- HTTP provides communications between web browsers & web servers
  - **Web**: framework for accessing documents & resources
  - **Hypertext documents**: text, graphics, images, hyperlinks
- RFC 1945 (HTTP 1.0), RFC 2616 (HTTP 1.1)
  - You will implement a subset of HTTP 1.1
- Documents prepared using Hypertext Markup Language (HTML)
  - HTML 5 introduces embedded media tag `<video>`, `<audio>`
  - Supported browsers: Firefox, Chrome
HTTP Protocol

• HTTP servers use well-known port 80
• Client request / Server reply
• Stateless
  – Server does not keep any information about client
  – Each request is independent
• HTTP 1.0
  – Non-persistent connection
  – Create new TCP connection per request/reply
• HTTP 1.1
  – Persistent connection by default
  – Subsequent request/reply performed on the same connection
  – Used in today’s browser/web server
HTTP Typical Exchange

![HTTP Capture]

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source IP Address</th>
<th>Destination IP Address</th>
<th>Protocol</th>
<th>Length</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>737</td>
<td>29.542064</td>
<td>208.80.152.201</td>
<td>10.251.0.5</td>
<td>TCP</td>
<td>1514</td>
<td>[TCP segment of a reassembled PDU]</td>
</tr>
<tr>
<td>738</td>
<td>29.542094</td>
<td>10.251.0.5</td>
<td>208.80.152.201</td>
<td>TCP</td>
<td>54</td>
<td>58609 &gt; http [ACK] Seq=12908 Ack=65692 Win=48180</td>
</tr>
<tr>
<td>739</td>
<td>29.347331</td>
<td>10.251.0.5</td>
<td>208.80.152.212</td>
<td>HTTP</td>
<td>530</td>
<td>GET /wikipedia/commons/d/5/SRI_First_Internet.txt</td>
</tr>
<tr>
<td>740</td>
<td>29.393913</td>
<td>10.251.0.5</td>
<td>208.80.152.201</td>
<td>TCP</td>
<td>1514</td>
<td>[TCP segment of a reassembled PDU]</td>
</tr>
<tr>
<td>741</td>
<td>29.399479</td>
<td>10.251.0.5</td>
<td>208.80.152.201</td>
<td>TCP</td>
<td>54</td>
<td>58609 &gt; http [ACK] Seq=12908 Ack=67940 Win=45932</td>
</tr>
<tr>
<td>742</td>
<td>29.600454</td>
<td>10.251.0.5</td>
<td>208.80.152.201</td>
<td>TCP</td>
<td>54</td>
<td>58609 &gt; http [ACK] Seq=12908 Ack=67940 Win=45932</td>
</tr>
</tbody>
</table>

**HTTP Transfer Protocol**

```
HTTP/1.0 200 OK
Date: Tue, 24 Jan 2012 20:13:52 GMT
Server: Apache
X-Content-Type-Options: nosniff
Cache-Control: private, s-maxage=0, max-age=0, must-revalidate
Content-Language: en
Vary: Accept-Encoding, Cookie
Content-Encoding: gzip
Content-Length: 9548
Content-Type: text/html; charset=UTF-8
Age: 9746
X-Cache: HIT from sq62.wikimedia.org
X-Cache-Lookup: HIT from sq62.wikimedia.org:3128
X-Cache: HIT from sq73.wikimedia.org
```

From (812 bytes) | Reassembled TCP (1006 bytes) | Uncompressed entity body (3435 bytes)
HTTP Message Formats

- HTTP messages written in ASCII text
- Request Message Format
  - Request Line (Each line ends with carriage return + line feed)
    - Method  URL HTTP-Version\r\n    - Method specifies action to apply to object
    - URL specifies object
  - Header Lines (Ea. line ends with carriage return + line feed)
    - AttributeName:  AttributeValue \r\n    - E.g. type of client, content, identity of requester, ...
    - Last header line has extra carriage return + line feed (\r\n)
  - Entity Body (Content, optional)
    - Additional information to server
- Minimal HTTP 1.1 request
  GET /path/file.html HTTP/1.1
  Host: www.host1.com:80
  [blank line here]
# HTTP Request Methods

<table>
<thead>
<tr>
<th>Request method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Retrieve information (object) identified by the URL.</td>
</tr>
<tr>
<td>HEAD</td>
<td>Retrieve meta-information about the object, but do not transfer the object; Can be used to find out if a document has changed.</td>
</tr>
<tr>
<td>POST</td>
<td>Send information to a URL (using the entity body) and retrieve result; used when a user fills out a form in a browser.</td>
</tr>
<tr>
<td>PUT</td>
<td>Store information in location named by URL</td>
</tr>
<tr>
<td>DELETE</td>
<td>Remove object identified by URL</td>
</tr>
<tr>
<td>TRACE</td>
<td>Trace HTTP forwarding through proxies, tunnels, etc.</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Used to determine the capabilities of the server, or characteristics of a named resource.</td>
</tr>
</tbody>
</table>
Universal Resource Locator

• Absolute URL
  – scheme://hostname[:port]/path
  – http://www.cmu.edu/

• Relative URL
  – /path
  – /
HTTP Request Message

```
GET /wikipedia/commons/thumb/1/1d/OTRS_wikimedia.svg/70px-OTRS_wikimedia.svg.png HTTP/1.1
Host: upload.wikimedia.org
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:9.0.1) Gecko/20100101 Firefox/9.0.1
Accept: image/png, image/*; q=0.8, */*; q=0.5
Accept-Language: en-us, en; q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8; q=0.7, *; q=0.7
Connection: keep-alive
```

[Full request URI: http://upload.wikimedia.org/wikipedia/commons/thumb/1/1d/OTRS_wikimedia.svg/70px-OTRS_wikimedia.svg.png]
HTTP Response Message

• Response Message Format
  – Status Line
    • HTTP-Version Status-Code Message
    • Status Code: 3-digit code indicating result
    • E.g. HTTP/1.1 200 OK
  – Headers Section
    • Information about object transferred to client
    • E.g. server type, content length, content type, …
  – Content
    • Object (document)

• Sample Response
  HTTP/1.1 200 OK
  Date: Fri, 31 Dec 2010 23:59:59 EST
  Content-Type: text/plain
  Content-Length: 42

  abcdefghijklmnopqrstuvwxyz1234567890abcdef
HTTP Response Message

```
HTTP/1.0 200 OK
Content-Type: image/png
Last-Modified: Sun, 03 Apr 2011 17:39:21 GMT
Connection: keep-alive
```

---

Frame (665 bytes) Reassembled TCP (2476 bytes)

```
HTTP/1.0 200 OK.
Server: nginx/0.7.65
Date: Tue, 24 Jan 2012 19:22:30 GMT
Content-Type: image/png
Content-Length: 2071
```

---


```
HTTP/1.0 Reassembled TCP Segment (2476 bytes)
```

---

Internet Protocol Version 4, Src: 208.80.152.211 (208.80.152.211), Dst: 10.251.0.5 (10.251.0.5)

```
TCP Segment of a reassembled PDU
```

---

Ethernet II, Src: 02-54-00-1E-00-1F (00:10:11:00:1E:00), Dst: 08:00:20:00:00:1F (00:10:00:00:1E:00)

```
TCP Segment of a reassembled PDU
```

---

HTTP/1.0 200 OK (PNG)

```
HTTP/1.0 200 OK
```

---

HTTP/1.0 200 OK

```
HTTP/1.0 200 OK
Server: nginx/0.7.65
Date: Tue, 24 Jan 2012 19:22:30 GMT
Content-Type: image/png
Content-Length: 2071
```
Network Programming
A Client-Server Transaction

- Most network applications are based on the client-server model:
  - A server process and one or more of client processes.
  - Server manages some resource.
  - Server provides service by manipulating resource for clients.

Note: clients and servers are processes running on hosts (can be the same or different hosts).
The Socket Interface

• What is a socket?
  – A descriptor that lets an application read/write from/to the network
  – Similar abstraction for network I/O as file I/O
  – Clients and servers communicate by reading/writing from/to socket descriptors
Socket API

- API (Application Programming Interface)
  - Provides a standard set of functions that can be called by applications

- Berkeley UNIX Sockets API (C)
  - Abstraction for applications to send & receive data
  - Applications create sockets that “plug into” network
  - Applications write/read to/from sockets
  - Implemented in the kernel
  - Facilitates development of network applications
  - Hides details of underlying protocols & mechanisms

- Also in Windows, Linux, and other OS’s

- Socket API is also available in Java (java.net.socket)
Communications through Socket Interface

- Application references a socket through a descriptor
- Socket bound to a port number

Underlying communication protocols
Internet Connections

- Clients and servers communicate by sending streams of bytes over connections.
- Socket address is identified by an **IP address:port** pair.
- A **port** is a 16-bit unsigned integer identifying a process (ephemeral, not physical port.)
- Ports below 1024 are reserved for standard protocols.

**Client socket address**
128.2.194.242:18345

**Server socket address**
208.216.181.15:80

**Client host address**
128.2.194.242

**Server host address**
208.216.181.15

*Note: 18345 is an ephemeral port allocated by the kernel*

*Note: 80 is a well-known port associated with Web servers*
Stream mode of service

Connection-oriented
- First, setup connection between two peer application processes
- Then, reliable bidirectional in-sequence transfer of *byte stream*
- Multiple write/read between peer processes
- Finally, connection release
- Uses TCP

Connectionless (Datagram)
- Immediate transfer of one block of information
- No setup overhead & delay
- Destination address with each block
- Send/receive to/from multiple peer processes
- Best-effort service only
  - Possible out-of-order
  - Possible loss
- Uses UDP
Client & Server Differences

- **Server**
  - Specifies well-known port # when creating socket
  - May have multiple IP addresses (net interfaces)
  - Waits passively for client requests

- **Client**
  - Assigned ephemeral port #
  - Initiates communications with server
  - Needs to know server’s IP address & port #
    - DNS for URL & server well-known port #
  - Server learns client’s address & port #

- **Peer-to-Peer**
  - A process performs as both server and client
Socket Calls for Connection-Oriented Mode

Server does Passive Open
- **socket** creates socket to *listen* for connection requests
- Server specifies type: TCP (stream)
- **socket** call returns: non-negative integer *descriptor*, or -1 if unsuccessful

Server:
- socket()
- bind()
- listen()
- accept()
  - Blocks
- read()
- write()

Client:
- socket()
- connect()
- write()
- read()
- close()
Socket Calls for Connection-Oriented Mode

Server does Passive Open
- **bind** assigns local address & port # to socket with specified descriptor
- Can wildcard IP address for multiple net interfaces
- **bind** call returns: 0 (success); or -1 (failure)
- Failure if port # already in use or if reuse option not set

Client
- socket()
  - connect()
    - write()
      - Data
      - read()
    - Data
  - close()
Socket Calls for Connection-Oriented Mode

Server does Passive Open

- **listen** indicates TCP readiness to receive connection requests for socket with given descriptor
- Parameter specifies file descriptor and max number of requests that may be queued while waiting for processing (backlog)
- **listen** call returns: 0 (success); or -1 (failure)
- **Java**: the backlog size is specified thru **bind**

Client

- **socket()**
- **connect()**
- **write()**
- **read()**
- **close()**
Server does Passive Open
- Server calls `accept` to accept incoming requests
- `accept` blocks if queue is empty

---

**Server**
- `socket()`
- `bind()`
- `listen()`
- `accept()` (Blocks)
  - Connect negotiation
  - Data
- `read()`
- `write()`

**Client**
- `socket()`
- `connect()`
  - Data
- `write()`
- `read()`
- `close()`
Socket Calls for Connection-Oriented Mode

Client does Active Open
- **socket** creates socket to connect to server
- Client specifies type: TCP (stream)
- **socket** call returns: non-negative integer *descriptor*; or -1 if unsuccessful

---

**Server**
- `socket()`
- `bind()`
- `listen()`
- `accept()`

Blocks
- `read()`
- `write()`

**Client**
- `socket()`
- `connect()`

Connect negotiation
- Data

---

Data
- `write()`
- `read()`

`close()`
### Socket Calls for Connection-Oriented Mode

**Client does Active Open**
- `connect` establishes a connection on the local socket with the specified descriptor to the specified remote address and port #
- `connect` returns 0 if successful; -1 if unsuccessful

**Server**
- `socket()`
- `bind()`
- `listen()`
- `accept()`

**Client**
- `socket()`
- `connect()`
- `write()`
- `read()`
- `close()`

**Note:** `connect` initiates TCP three-way handshake
Socket Calls for Connection-Oriented Mode

- **accept** wakes with incoming connection request
- **accept** fills client address & port # into address structure
- **accept** call returns: *descriptor of new connection socket* (success); or -1 (failure)
- Client & server use new socket for data transfer
- Original socket continues to listen for new requests
Socket Calls for Connection-Oriented Mode

Data Transfer
- Client or server call `write` to transmit data into a connected socket
- `write` specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior
- `write` call returns: # bytes transferred (success); or -1 (failure); blocks until all data transferred
Socket Calls for Connection-Oriented Mode

Data Transfer
- Client or server call `read` to receive data from a connected socket
- `read` specifies: socket descriptor; pointer to a buffer; amount of data
- `read` call returns: # bytes read (success); or -1 (failure); blocks if no data arrives

Note: `write` and `read` can be called multiple times to transfer byte streams in both directions
Socket Calls for Connection-Oriented Mode

Connection Termination

- Client or server call `close` when socket is no longer needed
- `close` specifies the socket descriptor
- `close` call returns: 0 (success); or -1 (failure)

Note: `close` initiates TCP graceful close sequence
Example: TCP Echo Server

/* A simple echo server using TCP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_TCP_PORT 3000
#define BUFLEN 256

int main(int argc, char **argv)
{
    int n, bytes_to_read;
    int sd, new_sd, client_len, port;
    struct sockaddr_in server, client;
    char *bp, buf[BUFLEN];

    /* Create a stream socket, last parameter is the protocol  
     * 0 = any (usually, there's only one for a protocol family) */
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket\n");
        exit(1);
    }

    /* Bind an address to the socket */
    bzero((char *)&server, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(SERVER_TCP_PORT);
    if (bind(sd, (struct sockaddr *)&server, sizeof(server)) == -1) {
        fprintf(stderr, "Can't bind to server\n");
        exit(1);
    }

    /* queue up to 5 connect requests */
    listen(sd, 5);

    while (1) {
        client_len = sizeof(client);
        if ((new_sd = accept(sd, (struct sockaddr *)&client, 
            &client_len)) == -1) {
            fprintf(stderr, "Can't accept client\n");
            exit(1);
        }
        bp = buf;
        bytes_to_read = BUFLEN;
        while ((n = read(new_sd, bp, bytes_to_read)) > 0) {
            bp += n;
            bytes_to_read -= n;
        }
        printf("Rec'd: %s\n", buf);
        write(new_sd, buf, BUFLEN);
        printf("Sent: %s\n", buf);
        close(new_sd);
    }

    close(sd);
    return(0);
}
Example: TCP Echo Server

```java
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer {
  static final int BUFLEN = 256;
  static final int SERVER_PORT = 3000;

  public static void main(String[] args) throws Exception {
    ServerSocket serverSocket = new ServerSocket();
    serverSocket.bind(SERVER_PORT, 5);

    byte[] buffer = new byte[BUFLEN];

    while (true) {
      Socket sock = serverSocket.accept();
      InputStream sIn = sock.getInputStream();
      OutputStream sOut = sock.getOutputStream();

      while (true) {
        try {
          int count = sIn.available();
          if (count > 0) {
            if (count > buffer.length) count = buffer.length;
            count = sIn.read(buffer, 0, count);
            sOut.write(buffer, 0, count);
            sOut.flush();
            sock.close();
          }
        } catch (IOException ioe) {
          break;
        }
      }
    }
  }
}
```
Example: TCP Echo Client

/* A simple TCP client */
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_TCP_PORT 3000
#define BUFLEN 256

int main(int argc, char **argv)
{
    int n, bytes_to_read;
    int sd, port;
    struct hostent *hp;
    struct sockaddr_in server;
    char *host, *bp, rbuf[BUFLEN], sbuf[BUFLEN];

    switch(argc) {
    case 2:
        host = argv[1];
        port = SERVER_TCP_PORT;
        break;
    case 3:
        host = argv[1];
        port = atoi(argv[2]);
        break;
    default:
        fprintf(stderr, "Usage: %s host [port]\n", argv[0]);
        exit(1);
    }

    /* Create a stream socket */
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket\n");
        exit(1);
    }

    bzero((char *)&server, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "Can't get server's address\n");
        exit(1);
    }
    bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);

    /* Connecting to the server */
    if (connect(sd, (struct sockaddr *)&server,
                sizeof(server)) == -1) {
        fprintf(stderr, "Can't connect\n");
        exit(1);
    }
    printf("Connected: server's address is %s\n", hp->h_name);

    printf("Transmit:\n");
    gets(sbuf);
    write(sd, sbuf, BUFLEN);

    printf("Receive:\n");
    bp = rbuf;
    bytes_to_read = BUFLEN;
    while ((n = read(sd, bp, bytes_to_read)) > 0) {
        bp += n;
        bytes_to_read -= n;
    }
    printf("%s\n", rbuf);

    close(sd);
    return(0);
}
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.Socket;

public class EchoClient {
    static final int BUFLEN = 256;
    static final int SERVER_PORT = 3000;
    static String SERVER_NAME = "localhost";

    public static void main(String[] args) throws Exception {
        InetAddress inetAddress = InetAddress.getByName(SERVER_NAME);
        Socket socket = new Socket(inetAddress, SERVER_PORT);
        byte[] buffer = new byte[BUFLEN];

        InputStream sIn = socket.getInputStream();
        OutputStream sOut = socket.getOutputStream();

        // Write "hello" to the socket
        sOut.write("hello".getBytes());
        sOut.flush();

        while ((sIn.read(buffer)) > 0) {
            System.out.println(new String(buffer));
        }
    }
}
Key data structures

IP Address

32-bit IP addresses are stored in an IP address struct in `<netinet/in.h>`
- IP addresses are always stored in memory in network byte order (big-endian byte order)
- True in general for any integer transferred in a packet header from one machine to another.
  - E.g., the port number used to identify an Internet connection.

Handy network byte-order conversion functions:
- `htonl`: convert long int from host to network byte order.
- `htons`: convert short int from host to network byte order.
- `ntohl`: convert long int from network to host byte order.
- `ntohs`: convert short int from network to host byte order.

You will have to use these functions in the project.
Byte-Ordering

• Consider a hexadecimal 4A3B2C1D at address 100. The bytes could be stored within the address range 100 through 103 in the following order:
  • **Big-endian (“big end first”)**

    | 100 | 101 | 102 | 103 |
    |-----|-----|-----|-----|
    |     | 4A  | 3B  | 2C  |
    |     | 1D  |     |     |

  – The most significant byte (msb, 4A) is stored at the lowest address.
  – Used by Motorola/SPARC and network devices.

• **Little-endian (“little end first”)**

    | 100 | 101 | 102 | 103 |
    |-----|-----|-----|-----|
    |     | 1D  | 2C  | 3B  |
    |     | 4A  |     |     |

  – The least significant byte (lsb, 1D) is stored at the lowest address
  – Used by Intel x86, DEC VAX

• Endianness does *not* denote what the value *ends* with when stored in memory, but rather *which end* it begins with.
Key data structures
Internet Socket Address

struct sockaddr_in  {
    unsigned short  sin_family;  /* address family (always AF_INET) */
    unsigned short  sin_port;    /* port num in network byte order */
    struct in_addr  sin_addr;    /* IP addr in network byte order */
    unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};

• Also defined in <netinet/in.h>
• IP address and port number must be stored in network byte order.
Resources

• Network Programming Tutorial
  – http://beej.us/guide/bgnet/

• Internet
  – http://www.ietf.org/
  – http://www.iana.org
  – www.wikipedia.org

• Books