CS434/534: Topics in Network Systems

HTTP/1.1 Selected Features;
Network Server Design: Socket API, Basic UDP
Client/Server Design

Sept. 14, 2021

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

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

- Slides posted on the Files folder
- Programming project 1 (part 1) will be posted on Thursday
Recap: Layered Network Systems

- Layered system design implications
  - Logical communications vs physical communications
  - Each layer can add meta data (packet headers) to implement its protocol
    - Needs a multiplexing field if providing services to higher layer
    - Meta data in a layered system can be considered managed as a stack

- High-level guiding principle of designing layers: the end-to-end principle

- The Internet 5-layer architecture
Recap: HTTP

- The HTTP protocol is designed to support a generic interface protocol supporting the Representational State Transfer (REST) architecture style
  - Stateless
  - Universal resource
    
    ```
    http-URI = "http:" /"" authority path-absolute [ "?" query ]
    [ "#" fragment ]
    ```
  - State transfer as universal operators to manipulate resources
  - Resource can have multiple representations
    - HTTP methods transfer representations of a resource
    - Content negotiation to select representation
    - Self-descriptive messages supporting representation
Recap: HTTP Content Negotiation

- HTTP dimensions to determine the representation
  - Client characteristics
  - Negotiation related fields
    - Client: Accept (mime type), Accept-Charset, Accept-Encoding, Accept-Language
    - Server: Vary

- Example
  - Using curl (curl -v -X <method> --http<ver> -H <header> <uri>)
  - curl -v http://www.google.com
Representation Selection in HTTP: Client Preference

- Representation/Content preference by (1) quality (q) value (higher q, higher preference); (2) concrete over wildcard
  - Exercises:
    - Accept: text/plain; q=0.5, text/html, text/x-dvi; q=0.8, text/x-c
      - What is the preference of representation?
    - Accept: text/*; q=0.3, text/html; q=0.7, */*; q=0.5
      - What is the q value of
        » text/html
        » text/html; charset=utf-8
        » text/plain
        » image/jpeg
    - What if you cannot decode gzip?
Outline

- Admin and recap
- Networked systems (network applications, network-based systems)
  - Overview
  - HyperText transfer system (HTTP/1.x)
    - Basic abstractions
      - Resource
      - State transfer
      - Representation
    - HTTP features
Demo: Transfer-Encoding

- Demo:
  - `curl -v http://www.cs.yale.edu`
    - How does the server indicate the content length?
  - `curl -v http://www.google.com`
    - Capture traffic using wireshark
    - Q: What does www.google.com indicate content length?
    - Q: What might be a reason to introduce the transfer encoding?
Conditional Transfer: Browser Cache and Conditional GET

- **Goal:** don’t send object if client has up-to-date stored (cached) version
- client: specify date of cached copy in http request
  - If-modified-since: <date>
- server: response contains no object if cached copy up-to-date:
  - HTTP/1.1 304 Not Modified

```
http request msg
If-modified-since: <date>

http response
HTTP/1.1 304 Not Modified
```

```
http request msg
If-modified-since: <date>

http response
HTTP/1.1 200 OK
...
<data>
```
Outline

- Admin and recap
- Networked systems (network applications, network-based systems)
  - Overview
  - FTP as comparison
  - HyperText transfer system (HTTP/1.x)
    - Basic abstractions
      - Resource
      - State transfer
      - Representation
    - HTTP features
      - Transfer encoding, conditional transfer
      - Transfer involving intermediates
Demo

- Exercise: curl -v http://www.yale.edu
  - Q: How does it redirect?

- Exercise: curl -v https://www.yale.edu
  - Q: Do you see intermediates on the path?
There are three common forms of HTTP intermediary: proxy (forward proxy), gateway (reverse proxy), and tunnel.

- **Forward proxy:** Typically in the same network as the client.
- **Reverse proxy (gateway):** Typically in the same network as the server.
Forward Proxy

- A message-forwarding agent that is selected by the client, usually via local configuration rules, to receive requests for some type(s) of absolute URI and attempt to satisfy those requests via translation through the HTTP interface.

- Forward proxies are often used for
  - improving security
  - providing annotation services
  - shared caching
Reverse Proxy (Gateway)

- "gateway" (a.k.a. "reverse proxy"): an intermediary that acts as an origin server for the outbound connection but translates received requests and forwards them inbound to another server or servers.

- Gateways are often used to
  - encapsulate legacy or untrusted information services
  - improve server performance through "accelerator" caching
  - enable partitioning or load balancing of HTTP services across multiple machines.
Example: Wikipedia Architecture

What are potential issues that can be caused by introducing intermediates and how HTTP/1.1 handles them?
Admin and recap

Networked systems (network applications, network-based systems)
  - Overview
  - FTP as comparison
  - HyperText transfer system (HTTP/1.x)
    - Basic abstractions
      - Resource
      - State transfer
      - Representation
    - HTTP features
      - Transfer coding, conditional transfer,
      - Transfer involving intermediates
      - Supporting stateful services
Example Stateful User-server Interaction: Cookies

Goal: no explicit application level session

- Server sends “cookie” to client in response msg
  
  Set-cookie: 1678453

- Client presents cookie in later requests
  
  Cookie: 1678453

- Server matches presented-cookie with server-stored info
  - authentication
  - remembering user preferences, previous choices
Offline Example: Authentication of Client Request

**Authentication goal:** control access to server documents

- **stateless:** client must present authorization in each request
- **authorization:** typically name, password
  - Authorization: header line in request
  - if no authorization presented, server refuses access, sends
    - WWW-authenticate: header line in response

Browser caches name & password so that user does not have to repeatedly enter it.
Network Systems Study: Key Questions

Key questions we ask about a network system
- What is the architecture (entities/processes, protocols involved)?

- How does it provide
  - Extensibility
  - Scalability
  - Robustness to failures
  - Security?
Offline Exercise Questions: Extensibility Related

- Why does HTTP/1.1 separate into major.minor version numbers?
- How does HTTP/1.1 define requirement to handle extensible headers?
- Is there a mechanism in HTTP/1.1 to evolve to a new major version number?
- Is there specification in HTTP/1.1 on how to handle issues that intermediates may have different versions?
Outline

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

- High-performance network application server
  - Basic transport API and network server
    - Basic socket API
    - Simple example and a peek inside socket operation
    - Overview of issues
  - High performance network server
    - Multi-threaded network servers
    - select io network servers
  - Operational analysis
  - Example server analysis:
    - netty and nginx on high performance, modularity design patterns
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
Network server is implemented by using the services provided by transport layer below it.

Transport-layer services are realized using transport layer protocol(s).
- A network system can design multiple transport layer services, each using its own transport protocol.

Network server invokes transport-layer services through the transport-layer interface (API).
Transport Layer Service: TCP

- **Services**
  - multiplexing/demultiplexing
  - reliable transport
    - between sending and receiving processes
    - setup required between sender and receiver: a connection-oriented service
  - flow control: sender won’t overwhelm receiver
  - congestion control: throttle sender when network overloaded (do not see)
  - error detection
  - does not provide timing, bandwidth
TCP Protocol Header

TCP design based on transport port number, introduced in BSD Socket.
**Transport Layer Service: UDP**

- A connectionless service
- Does not provide: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee
  - why is there a UDP?
UDP Protocol Header

<table>
<thead>
<tr>
<th>0</th>
<th>15</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit source port number</td>
<td>16-bit destination port number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit UDP length</td>
<td>16-bit UDP checksum</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

data (if any)

Ethernet, Wireless, Cable/DSL, IP, UDP, TCP, Email, FTP, Telnet, WWW, FDDI
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
      - The basic socket API
- Introduced in BSD4.1 UNIX, 1981
- Conceptually quite simple: an interface (a “door”) into which one application process can both send and receive messages to/from another (remote or local) application process
The Basic Socket API

- Two common 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 labelled w/ src/dst port numbers
- Incoming packets are multiplexed to processes according to port binding
Exercise

- What might the UDP socket API look like if you were to design it?
DatagramSocket (Java) (Basic)

- **DatagramSocket()**
  constructs a datagram socket and binds it to any available port on the local host

- **DatagramSocket(int lport)**
  constructs a datagram socket and binds it to the specified port on the local host machine.

  // more methods on multiplexing control: bind, connect; see demos

- **DatagramPacket(byte[] buf, int length)**
  constructs a DatagramPacket for receiving packets of length length.

- **DatagramPacket(byte[] buf, int length, InetAddress address, int port)**
  constructs a datagram packet for sending packets of length length to the specified port number on the specified host.

- **receive(DatagramPacket p)**
  receives a datagram packet from this socket.

- **send(DatagramPacket p)**
  sends a datagram packet from this socket.

  // socket state control

- **close()**
  closes this datagram socket.

https://docs.oracle.com/javase/9/docs/api/java/net/DatagramSocket.html
Example: UDPServer, UDPClient

- A simple UDP server which changes any received sentence to upper case
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        // Check socket state
        %netstat -a -p udp -n
        %java UDPServer
        %netstat -a -p udp -n
System State after the Call

“*” indicates that the socket binds to all IP addresses of the machine:
% ifconfig -a
Exercise: UDP Socket Demultiplexing

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

- Send message to server
  - %java UDPClient localhost 9876

- Check server socket state
  - netstat -a -p udp -n
Per UDP Socket State

- Each Datagram socket has a set of states:
  - binding to IP address, port
  - send buffer size
  - receive buffer size
  - timeout
  - traffic class

See
http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html
Next Class

- TCP socket
- High-performance HTTP server design: thread based
- Please read the staged, event-driven architecture (SEDA) paper
Backup: UDP Server/Client Example Details
Offline: Binding to Specific IP Addresses

```java
InetAddress sIP1 = InetAddress.getByName("localhost");
DatagramSocket ssock1 = new DatagramSocket(9876, sIP1);

InetAddress sIP2 = InetAddress.getByName("128.36.59.2");
DatagramSocket ssock2 = new DatagramSocket(9876, sIP2);

DatagramSocket serverSocket = new DatagramSocket(6789);
```
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = null;

        while(true) {
            DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
        }
    }
}
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        ...
        // process data
        String sentence = new String(receivePacket.getData(), 0, receivePacket.getLength());
        String capitalizedSentence = sentence.toUpperCase();
        sendData = capitalizedSentence.getBytes();
    }
}
Java Server (UDP): Response

- **Java DatagramPacket:**
  - `getAddress()` / `getPort()`
    - returns the source address/port
Get IP addr, port #, of sender

```java
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
```

Create datagram to send to client

```java
DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);
```

Write out datagram to socket

```java
serverSocket.send(sendPacket);
```

End of while loop, loop back and wait for another datagram

```java
}
```

Java server (UDP): Reply
Example: Java client (UDP)

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

class UDPClient {
    public static void main(String[] args) throws Exception {
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        String sentence = inFromUser.readLine();
        byte[] sendData = sentence.getBytes();

        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress sIPAddress = InetAddress.getByName("servname");
        InetAddress sIPAddress = InetAddress.getByName("servname");

        DatagramPacket packet = new DatagramPacket(sendData, sendData.length, sIPAddress, 2000);
        clientSocket.send(packet);
    }
}
```
Example: Java client (UDP), cont.

```
DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, sIPAddress, 9876);
clientSocket.send(sendPacket);

byte[] receiveData = new byte[1024];
DatagramPacket receivePacket =
    new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);

String modifiedSentence =
    new String(receivePacket.getData());

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