CIS 455/555: Internet and Web Systems

Server Architectures

September 8, 2021
Announcements

- HW0 due Monday
- HW1 is up now!
  - New: Milestone 0 due next Friday
  - See website for handout. Should probably start now!

- Reading for next time:
  - "Flash: An Efficient and Portable Web Server"
  - "HTTP Made Really Easy"
  - Also, a brief intro to the Spark Framework (for HW1)
The road ahead

- I will begin by introducing some basics
  - Example: Client/server

- Some of you have seen some of this already
  - E.g., in NETS212, CIS553

- If so, stay tuned!
  - This is necessary to bring everyone to the same page
Agenda for the next two lectures

- Brief discussion of the Lampson paper
- The client/server model
- An example service: Web/HTTP
- Servers with threads
- Consistency issues with shared resources
- Event-driven servers
- State, and where to keep it
- Extensible servers
- Multi-tier services and replication
System design

- To this point, you’ve probably had significant experience designing programs to solve specific, relatively small tasks.

- It’s often a very difficult job to build a system!
  - What is a computing system?
  - Why is it harder to build?

- We will explore the process in this course:
  - Architectural aspects [Butler Lampson article]
  - Algorithmic aspects [e.g., two-phase commit]
  - Engineering aspects [e.g., build management]
About Butler Lampson

- Butler Lampson is a Distinguished Engineer at Microsoft Corporation and an Adjunct Professor of Computer Science and Electrical Engineering at MIT.
- He was one of the designers of the SDS 940 time-sharing system, the Alto personal distributed computing system, the Xerox 9700 laser printer, two-phase commit protocols, ...
- He received the ACM’s Software Systems Award in 1984 for his work on the Alto, the Turing Award in 1992, the IEEE Computer Pioneer award in 1996, and the Von Neumann medal in 2001.

http://research.microsoft.com/en-us/um/people/blampson/
Historical note: Xerox Alto

1972-78

- Personal computer for research
- The first GUI-based computer (note the mouse!)
- 128KB RAM, 2.5MB hard disk
- Ethernet

In many ways, the forerunner to the Xerox Star

- ... Which begat the Apple Lisa, and the rest is history!
Designing a computer system is very different from designing an algorithm:

   The external interface (that is, the requirement) is less precisely defined, more complex, and more subject to change.

   The system has much more internal structure, and hence many internal interfaces.

   The measure of success is much less clear.

The designer usually finds himself floundering in a sea of possibilities, unclear about how one choice will limit his freedom to make other choices, or affect the size and performance of the entire system. There probably isn’t a ‘best’ way to build the system, or even any major part of it; much more important is to avoid choosing a terrible way, and to have clear division of responsibilities among the parts.
Lampson’s advice

<table>
<thead>
<tr>
<th>Why?</th>
<th>Functionality</th>
<th>Speed</th>
<th>Fault-tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Does it work?</td>
<td>Is it fast enough?</td>
<td>Does it keep working?</td>
</tr>
<tr>
<td>Where?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Completeness</strong></td>
<td>Separate normal and worst case</td>
<td>Shed load End-to-end Safety first</td>
<td>End-to-end</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td>Do one thing well: Don’t generalize Get it right Don’t hide power Use procedure arguments Leave it to the client Keep basic interfaces stable Keep a place to stand</td>
<td>Make it fast Split resources Static analysis Dynamic translation</td>
<td>End-to-end Log updates Make actions atomic</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>Plan to throw one away Keep secrets Use a good idea again Divide and conquer</td>
<td>Cache answers Use hints Use brute force Compute in background Batch processing</td>
<td>Make actions atomic Use hints</td>
</tr>
</tbody>
</table>
Agenda for the next two lectures

- Brief discussion of the Lampson paper
- The client/server model
- An example service: Web/HTTP
- Servers with threads
- Consistency issues with shared resources
- Event-driven servers
- State, and where to keep it
- Extensible servers
- Multi-tier services and replication
How can we implement a system like the Web?
The peer-to-peer model

- How the Web **could** work (but doesn't):
  - Each machine locally stores some documents
  - If a machine needs a new document, it asks some other machines until it finds one that already has the document
  - No machine is special - they are all 'peers'

- Pros and cons of this approach?
The client-server model

- How the Web actually works today:
  - Some machines (servers) offer documents
  - Other machines (clients) use documents
  - Clients can request documents from servers
  - Model is used for many other services, not just for the web

- Pros and cons of this approach?
  - Every design choice has pros and cons! ("It depends!")
  - Need to think about the application's requirements!
Servers

- **Server**: A machine that offers services to other machines
  - Examples: Mail server, file server, chat server, print server, terminal server, web server, name server, game server, ...
- Protocol often uses request/response pattern
What a server does

- Toy example: An 'arithmetic server' that performs computations
  1. Wait for an incoming request for computation
  2. Read the request
  3. Parse the request
  4. Verify authorization
  5. Perform the requested computation
  6. Generate a document with the response
  7. Return the response to the requester
  8. Repeat
Port numbers and well-known ports

- A single machine can host multiple services
  - Typically distinguished by TCP port number
  - Many services have well-known port numbers, e.g., web servers use port 80, SSH servers use port 22, ...
Challenges for server designers

Major issues:

- **Concurrency**
  - How do we handle multiple simultaneous requests?

- **Statefulness and sessions**
  - Are requests self-contained, or do they require the server to keep around state?

- **Communication and consistency**
  - Today, servers are typically replicated!
  - What state is shared across requests?
  - Do all requests need the same view?

- ... And, of course, **security!!!**
Agenda for the next two lectures

- Brief discussion of the Lampson paper
- The client/server model
- An example service: Web/HTTP
- Servers with threads
- Consistency issues with shared resources
- Event-driven servers
- State, and where to keep it
- Extensible servers
- Multi-tier services and replication
URIs, URNs, and URLs

- **Uniform Resource Identifier (URI)**
  - Comes in two forms: URN and URL

- **Uniform Resource Name (URN)**
  - Specifies *what* to find, independent of its location
  - Example: urn:isbn:0596521974 (for the course textbook)

- **Uniform Resource Locator (URL)**
  - Specifies *where* to find something
  - `<scheme>://[user[:password]@]<server>[:port]/[path][/resource][?param1=value1&param2=value2&...]`
The HTTP protocol

- How to communicate with a web server?
  - Use the HyperText Transfer Protocol (HTTP)

- A very simple protocol
  - First specified in 1990; runs on top of TCP/IP
  - Default port 80 (unsecure), or 443 (secure, with SSL)
  - Originally stateless (HTTP/1.0), but next version (HTTP/1.1) added support for persistent connections
    - Why?

- Development continues
  - HTTP/2 specification was adopted in 2015!
  - Supports compression, server push, connection multiplexing, scheduling... TLS encryption now (de facto) mandatory
Example: A simple HTTP request

Method
GET /~cis455/foo.html HTTP/1.1
Accept: */*
Referer: http://www.cis.upenn.edu/~cis455/
User-Agent: Mozilla/5.0 (Macintosh)

URI

Status
HTTP/1.1 200 OK
Date: Wed, 17 Jan 2018 10:14:20 GMT
Server: Apache
Last-Modified: Wed, 17 Jan 2018 10:14:04 GMT
Content-Type: text/html
Content-Length: 107

Content (optional)
<html><head><title>Test document</title></head><body><h3>Test</h3><p>This is a test</p></body></html>
Common HTTP methods

- **GET**
  - Retrieve whatever information is identified by the URI

- **HEAD**
  - Like GET, but retrieves only metadata, not the actual object

- **PUT**
  - Store information under the specified URI

- **DELETE**
  - Delete the information specified by the URI

- **POST**
  - Adds new information to whatever is identified by the URI
  - Intended, e.g., for newsgroup posts; today, used mostly to implement dynamic content via forms
Status codes

- Server sends back a status code to report how the request was processed

- Common status codes:
  - 200 OK
  - 301 Moved Permanently
  - 304 Not Modified
  - 401 Unauthorized
  - 403 Forbidden
  - 404 Not Found
  - 500 Internal Server Error
Headers

GET /~cis455/assignments.html HTTP/1.1
Host: 158.130.53.72:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13.2; rv:43.0)
  Gecko/20100101 Firefox/57.0
Referer: http://www.cis.upenn.edu/~cis455/
Accept: text/html, application/xhtml+xml, application/xml;q=0.9,*/*;q=0.8
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
Keep-Alive: 115
Connection: keep-alive

HTTP/1.1 200 OK
Date: Wed, 17 Jan 2018 10:13:25 GMT
Server: Apache
Last-Modified: Tue, 16 Jan 2018 10:13:23 GMT
Content-Length: 10528
Connection: close
Content-Type: text/html; charset=UTF-8

<html><head><title>...
Media types ("MIME types")

- Used to specify the nature of the data that is being transferred
  - Defined in RFC 2046
  - Included in the 'Content-type' header
  - At least two parts: Type and subtype

- Some common examples:
  - text/plain
  - application/octet-stream
  - audio/mpeg
  - image/png
  - image/jpeg
  - video/quicktime
'Stateless services'

- Many of you will have heard that HTTP is 'stateless'
- What does this mean?

- Need to distinguish between:
  - **Service state**: State that is maintained independent of particular clients, such as the files on a file server or the web pages on a web server
  - **Session state**: State that is maintained for specific client connections or client requests
  - Typically, 'stateless' refers only to the latter
Statefulness

- Very early HTTP was essentially stateless
  - Make a request; the response is the page that is specified by the URL

- More recent HTTP, and other protocols:
  - Some amount of state is maintained
  - In HTTP, this requires cookies (more later)
  - Connection can be kept open, with state preserved at both ends ('persistent connection') - also in many other protocols

- Pros and cons of statefulness?
Recap: HTTP

- A simple example for the client/server model

- What does it do?
  - Transfer web pages from server to client (browser), and input data from client to server

- What are some of its features?
  - Common methods: GET, HEAD, PUT, DELETE, POST
  - Status codes: 200 OK, 404 Not found, ...
  - Headers (for additional information)
  - MIME types
  - Stateless (in its original form)
Agenda for the next two lectures

- Brief discussion of the Lampson paper ✓
- The client/server model ✓
- An example service: Web/HTTP ✓
- Servers with threads
- Consistency issues with shared resources
- Event-driven servers
- State, and where to keep it
- Extensible servers
- Multi-tier services and replication
A simple web server

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

public class HttpServer {
    public static void main(String args[]) throws Exception {
        ServerSocket serverSocket = new ServerSocket(80);
        while (true) {
            Socket socket = serverSocket.accept();
            InputStreamReader reader = new InputStreamReader(socket.getInputStream());
            BufferedReader in = new BufferedReader(reader);
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

            String request = in.readLine();
            out.println("HTTP/1.1 200 OK\n\n<html><body>Hello world!</body></html>\n");
            socket.close();
        }
    }
}

- Is this a good (web) server? If not, why not?
The need for concurrency

- What if a server receives >1 request?
  - Idea #1: Process them serially
  - Problem: Slow client can block everyone else
  - Idea #2: One server for each request
  - Problem: Wasteful

- Server needs to handle requests concurrently
  - Available resources (CPUs, disks, etc...) are multiplexed between requests
  - Available techniques: Threads/processes, events, cooperative scheduling, thread pools, ...
Refresher: Threads and processes

- Physical machine has some fixed number of processor cores - say, c
  - What if we need more than c threads of execution?

- Idea: Time-share cores
  - A single core works on one thread for a while, then context-switch to another one
  - Switching can be done cooperatively: Each thread yields the core to another thread when it has nothing to do
  - Switching can also be preemptive: Each thread gets to run for a fixed amount of time (quantum, typ. 10-100ms)
  - Pros and cons of preemption?
  - Difference between a thread and a process?
Thread-based servers in Java

- Relevant Java constructs:
  - Worker can be a subclass of `Thread + implement run()`
  - Worker `w = new Worker(); w.start();`
  - Alternative: Worker implements Runnable
  - Thread `t = new Thread(w); t.start();`

- What if the threads share some resources?
  - Potential race conditions + consistency issues
  - Can use synchronization and locking
  - Need to be careful about deadlock, livelock, starvation (more about this later)
A simple thread-based web server

```python
worker(connection) {
    request = connection.read();
    if (request == "GET <document>") {
        data = filesystem.read(document);
        connection.write("HTTP/1.1 200 OK");
        connection.write(data);
    } else {
        connection.write("HTTP/1.1 400 Bad req");
    }
    close(connection);
}

main() {
    socket = listen(port 80);
    while (true) {
        connection = accept(socket);
        (new Thread).run(worker, connection);
    }
}
```

Worker thread (one per connection)

Dispatcher thread (accepts new connections and launches new threads)