CSE 331
Software Design & Implementation

James Wilcox & Kevin Zatloukal

Fall 2022

Servers

Event-driven programming

An *event-driven* program is designed to wait for events:

- program initializes then enters the *event loop*
- abstractly:

```
do {
    e = getNextEvent();
    process event e;
} while (e != quit);
```

Server Programming

- Servers sit around waiting for events like:
  - new client connections
  - new data from the client (high scale servers)

- Simple version (normal scale):

```
while (true) {
  wait for a client to connect
  process the request; send a response back
}
```

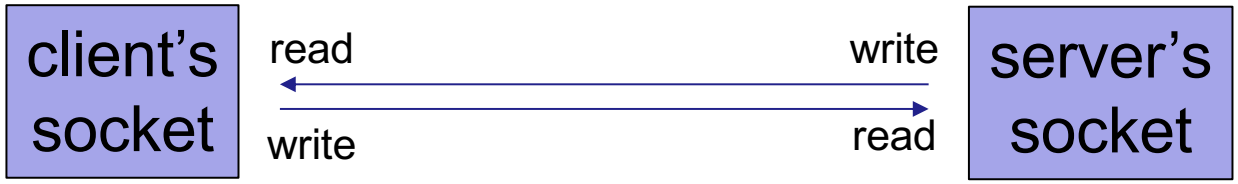
  - probably want to use a new thread for processing
  - high scale web servers might look quite different

Sockets

- Each client connection is represented by a “socket”

- A socket is like a **file**
  - can be read from and written to
  - (in Unix, sockets and files are nearly identical)

- Client and server each have “half” of the socket
  - what the client writes is read by the server
  - what the server writes is read by the client



Server Sockets & Ports

- Server creates a "server socket" and waits for a connection
  - each connection comes with an individual socket
  - allows reading from / writing to that client

- Servers on the same machine distinguished by a **port** number
  - numbers below 1024 require admin privileges

```
ServerSocket ssock = new ServerSocket(80);
```

- Clients indicate the port when trying to connect:

```
Socket sock = new Socket("attu", 80);
```

Ports & Protocols

- Sockets API allows reading & writing of byte data
  - like the File API

- Each server can define its own **protocol** for communication
  - the language it uses to speak to clients

- By convention, ports are associated with particular protocols
  - 80 = HTTP
  - 443 = HTTPS
  - 25 = SMTP relay
  - …

- Client that wants to talk HTTP can try connecting to 80

Example: Chat Server & Client

`ChatServer.java`

`ChatClient.java`

Aside: Java Lambda syntax

- Java 8+ allow a JS-like syntax for making "functions"
  - allowed for arguments that require an interface
  - when that interface has only a single method

- Example: **`Button.addActionListener`**
  - expects an **`ActionListener`** object
  - with a single method **`actionPerformed(ActionEvent)`**

```
button1.addActionListener((e) -> {
  System.out.println("clicked!");
});
```

Example: GUI + sockets

Most modern client applications have to both

- display a GUI
- communicate with one or more servers
- (doing both creates additional difficulties…)

We can make an example by writing a GUI chat client

`ChatClientGUI.java`

Protocols

- HTTP (Hyper-Text Transfer Protocol) is the most important
  - initially created for retrieving HTML documents
  - simple, text-based protocol

- Trend moving away from new protocols toward re-use of HTTP
  - Google (2010s) used HTTP for almost everything

- Allows for re-use of **libraries** for creating HTTP servers…
  - use of libraries reduces bugs, saves time, etc.
  - do not write your own HTTP server

HTTP

HTTP Request 1

```
GET /index.html HTTP/1.1
```

- Request ends with a **blank line**
- Between GET and blank are optional headers of the form

```
Name: Value
```

  - similar to Java properties files
  - common example would be `User-Agent` to describe client

HTTP Response 1

```
HTTP/1.1 200 OK
content-length: 124
content-type: text/html; charset=UTF-8
Date: Wed, 27 May 2020 18:30:00 GMT
Connection: close

<html>
…
```

- 200 status code indicates successful
- 400s for error that is the client's fault
- 500s for error that is the server's fault

Demo

```
(command-line HTTP request)
```

HTTP Request 2

```
POST /register HTTP/1.1
content-type: application/x-www-form-urlencoded
content-length: 25

fname=Kevin&userid=kevinz
```

- **`POST`** request includes client content
- 25 bytes of content after the blank line
  - newlines are just another byte

HTTP

- **`GET`** & **`POST`** requests are by far the most common
  - other types like DELETE also exist

- See CSE 333 for a more complete discussion
  - (no need to memorize the details here)

Uniform Resource Locators (URLs)

- Tells the browser what to get and how to get it

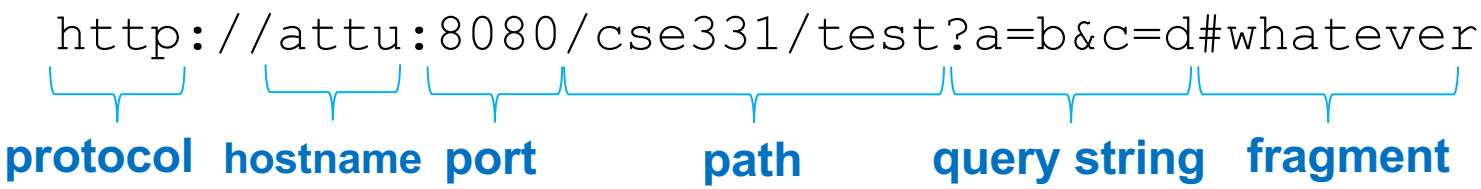
```
http://attu:8080/index.html
```

Connect to server attu on port 8080

Send GET request

```
GET /index.html HTTP/1.1
…
```

Uniform Resource Locators (URLs)



- **Port** is optional (default is 80 for HTTP)
- Optional "`?a=b&c=d`" part of path is called **query string**
  - "&"-separated key=value pairs
  - useful for passing arguments to the server-side code…
- **Fragment** is only kept in the browser
  - client can use this to record its place in the document
  - allows back/forward buttons to work on a single page

HTTP SERVERS

Server Frameworks

- How do we write a modular HTTP server?
  - need to split up the code into multiple classes

- Usual technique is to route requests using the **path**
  - use path to choose class that handles the request
  - used in Java, C++, Python, JavaScript, …
  - pass data to class using:
    - query string
    - POST body
    - (part of) path

Spark Java

- Simple library for writing HTTP servers in Java
  - not to be confused with “Apache Spark” — very different!

- Give Spark paths and corresponding classes
  - latter are called “routes” in this library
  - server will read the request path and invoke appropriate class
    - info about the request passed in request object
    - response can be written to response object or returned

- Library handles the event loop

Spark Java

```
Spark.get("/path", new MyRoute());
```

- `GET` request with this path are sent to this object

- Second argument must implement `Route` interface
  - single required method `handle(Request, Response)`
  - that means it can also be implemented with a **Lambda**

```
Spark.get("/ready", (request, response) -> {
  return "Nah, I'm busy";
});
```

Example: Hello Server

`HelloServer.java`