Multi-Threaded Servers

December 6, 2007

Client-Server Communication

- Many clients; 1 server
- Server starts and then waits for clients to connect
- Client initiates communication
- Server must handle client requests concurrently
- Server must not confuse client requests
Sockets

1. Server creates ServerSocket at a port and waits
2. Client creates Socket to connect to Server
3. Client writes request on Socket
4. Server reads request from socket
5. Server computes response
6. Server writes response on socket
7. Client reads response and processes it

What Server Does

1. Initialize the application
2. Listen on a port by constructing a “server socket”
   
   ServerSocket ss = new ServerSocket (port)

3. Wait for a client to connect
   
   Socket s = ss.accept();

4. Read from the socket to get the client’s request
5. Write to the socket to respond
What Client Does

1. Create a socket connected to the server computer and port
   
   ```java
   Socket s = new Socket("www.google.com", 80);
   ```

2. Write the request to the socket

3. Read the response

Reading Sockets

- A socket has an input stream (to read from) and an output stream (to write to)

  ```java
  InputStream inStream = socket.getInputStream();
  ```

- To get a more convenient and more efficient way to read regular text:

  ```java
  BufferedReader reader = new BufferedReader(new InputStreamReader(inStream));
  ```

- To read a line of text:

  ```java
  String line = reader.readLine();
  ```
Writing Sockets

- Get the output stream (of the same socket)
  
  ```java
  OutputStream outStream = socket.getOutputStream();
  ```

- To get a more convenient and more efficient way to write regular text:
  
  ```java
  PrintStream writer = new PrintStream(outStream);
  ```

- To write a line of text:
  
  ```java
  writer.println(line);
  ```

Multi-Threaded Server

- Start a new thread to handle each request that comes in

```java
while (true) {
    Socket s = ss.accept();
    Worker ws = new Worker(root, workerNumber);
    ws.setSocket(s);
    (new Thread(ws, "worker " + workerNumber)).start();
    workerNumber++;
}
```
Worker Thread

```java
public void run() {
    System.out.println("Starting worker " + id);
    try {
        handleClient();
    } catch (IOException e) {
        e.printStackTrace();
    }
    System.out.println("Done worker " + id);
    // Release the socket
    s = null;
}
```

Apache, Tomcat and Java Servlets

- **Apache**
  - A popular, open source Web server
  - Handles HTTP requests
  - Responds directly to normal file/directory requests

- **Tomcat**
  - Interfaces between Apache and servlets
  - Manages a pool of threads that servlets run in

- **Java Servlet**
  - Java program accessible via the Web
  - Runs on the server
Worker Pool

- Better response to requests:
  - No need to create Worker object
  - If keep Worker thread running, no need to start and stop threads
  - Too many worker threads can rapidly degrade performance due to thrashing