

TCP Programming

RES, Lecture 2 (second part)

Olivier Liechti
Juergen Ehrensberger



HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD

www.heig-vd.ch

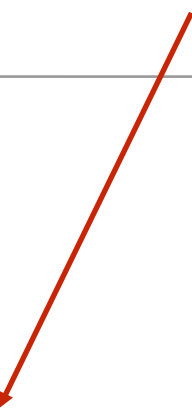


Rappel: Example: **05-DumbHttpClient**



Code walkthrough

**establish a connection
with server**



```
public void sendWrongHttpRequest() {  
    Socket clientSocket = null;  
    OutputStream os = null;  
    InputStream is = null;
```

```
    try {
```

```
        clientSocket = new Socket("www.lematin.ch", 80);  
        os = clientSocket.getOutputStream();  
        is = clientSocket.getInputStream();
```

**get streams to send and
receive bytes**




```
        String malformedHttpRequest = "Hello, sorry, but I don't speak HTTP...\r\n\r\n";  
        os.write(malformedHttpRequest.getBytes());
```

```
        ByteArrayOutputStream responseBuffer = new ByteArrayOutputStream();  
        byte[] buffer = new byte[BUFFER_SIZE];  
        int newBytes;
```

```
        while ((newBytes = is.read(buffer)) != -1) {  
            responseBuffer.write(buffer, 0, newBytes);  
        }
```

**read bytes sent by the
server until the
connection is closed**



```
        LOG.log(Level.INFO, "Response sent by the server: ");  
        LOG.log(Level.INFO, responseBuffer.toString());
```

```
    } catch (IOException ex) {  
        LOG.log(Level.SEVERE, null, ex);  
    } finally {
```

```
    }
```

...



Example: **04-StreamingTimeServer**



Code walkthrough

```
ServerSocket serverSocket = null;  
Socket clientSocket = null;  
BufferedReader reader = null;  
PrintWriter writer = null;
```

```
try {  
    serverSocket = new ServerSocket(listenPort);  
    logServerSocketAddress(serverSocket);  
    clientSocket = serverSocket.accept();
```

bind on TCP port




**block until a client makes a
connection request**



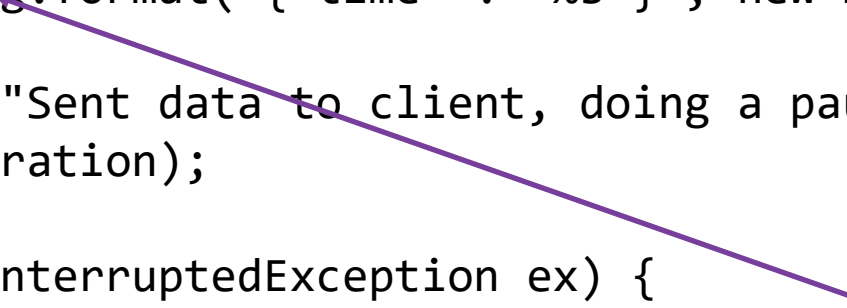
```
    logSocketAddress(clientSocket);  
    reader = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));  
    writer = new PrintWriter(clientSocket.getOutputStream());
```

**we want to exchange
characters with the
clients (we should
specify the encoding!)**



```
    for (int i = 0; i < numberOfIterations; i++) {  
        writer.println(String.format("{ 'time' : '%s' }", new Date()));  
        writer.flush();  
        LOG.log(Level.INFO, "Sent data to client, doing a pause...");  
        Thread.sleep(pauseDuration);  
    }  
} catch (IOException | InterruptedException ex) {  
    LOG.log(Level.SEVERE, ex.getMessage());  
} finally {  
    reader.close();  
    writer.close();  
    clientSocket.close();  
    serverSocket.close();  
}
```

**we make sure to flush
the buffer, so that
characters are actually
sent!**



Handling Concurrency





blocking IO (synchronous)

n employee = n threads

employees are expensive

limited space for employees in the truck

few employees => long queue



non-blocking IO (asynchronous)

there is only 1 employee (1 thread)

customers are called back when the request is fulfilled

no queue





blocking IO (synchronous)

n employee = n threads

employees are expensive

limited space for employees in the truck

few employees => long queue



non-blocking IO (asynchronous)

there is only 1 employee (1 thread)

customers are called back when the request is fulfilled

no queue





blocking IO (synchronous)

n employee = n threads

employees are expensive

limited space for employees in the truck

few employees => long queue



non-blocking IO (asynchronous)

there is only 1 employee (1 thread)

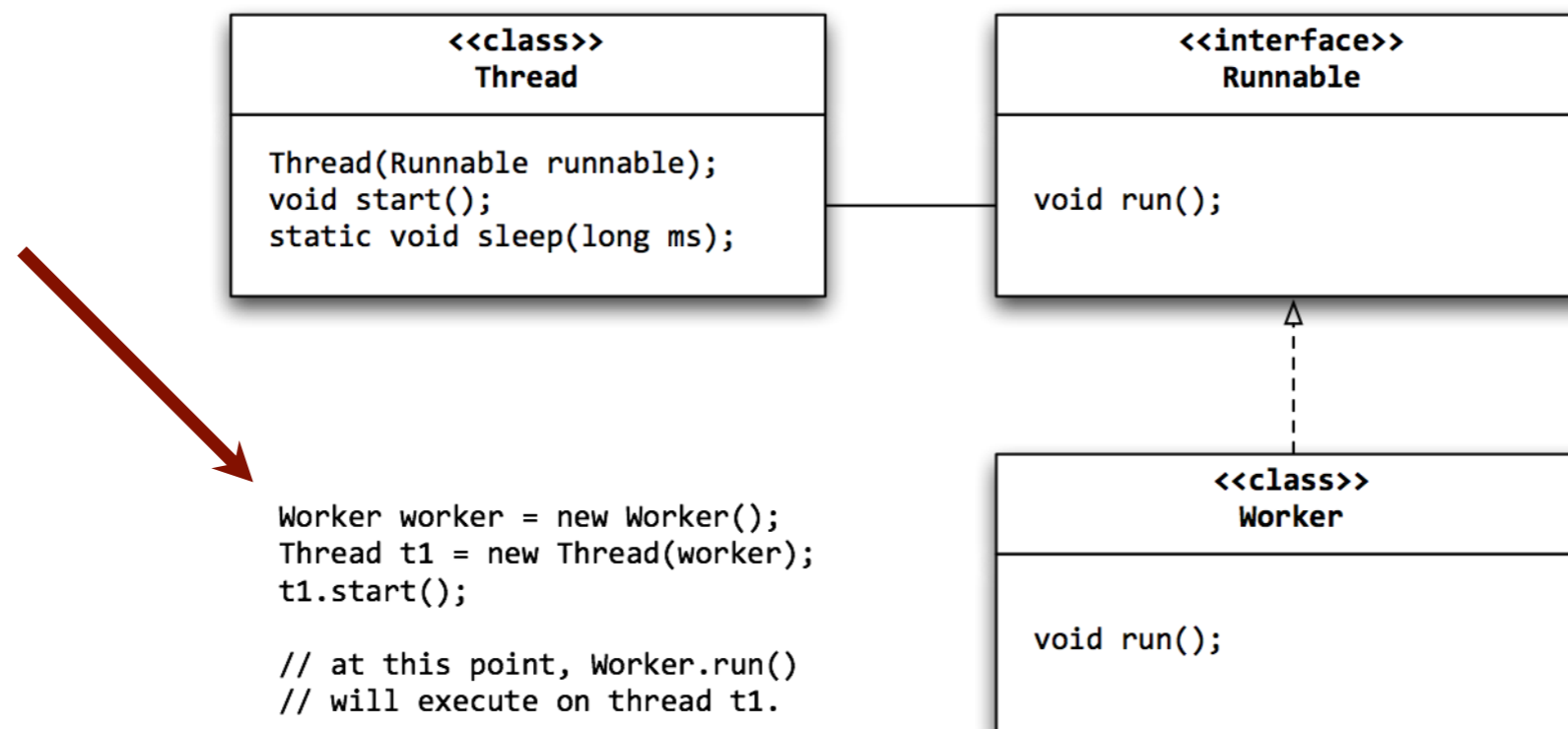
customers are called back when the request is fulfilled

no queue



Concurrent Programming in Java

- In Java, there are two main types
 - The **Thread class**, which *could be extended* to implement the behaviour you want to run in parallel.
 - The **Runnable interface**, which *is implemented* for the same purpose and is passed as an argument to the Thread constructor.



Concurrent Programming in Java

- There are other classes related to threads, in the `java.util.concurrent` package. An important one is the **ExecutorService**, which makes it possible to use **thread pools**.
- A thread pool gives you a way to limit the number of threads spawned (by your server), so that you will not consume all resources. Others are queued.

```
package de.vogella.concurrency.threadpools;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class Main {
    private static final int NTHREDS = 10;

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
        for (int i = 0; i < 500; i++) {
            Runnable worker = new MyRunnable(10000000L + i);
            executor.execute(worker);
        }
        // This will make the executor accept no new threads
        // and finish all existing threads in the queue
        executor.shutdown();
        // Wait until all threads are finish
        executor.awaitTermination();
        System.out.println("Finished all threads");
    }
}
```

<http://www.vogella.com/tutorials/JavaConcurrency/article.html>



Single Threaded Single Process Blocking

Not really an option...

The server implements a loop.

**It waits for a client to arrive.
Then services the client until done.**

**Then only goes back to accept the next
client.**

Can only talk to 1 client at the time

It is only when we reach this line that
a new client can connect

```
serverSocket = new ServerSocket(port);
while (true) {

    clientSocket = serverSocket.accept();

    in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
    out = new PrintWriter(clientSocket.getOutputStream());
    String line;
    boolean shouldRun = true;

    LOG.info("Reading until client sends BYE");

    while ( (shouldRun) && (line = in.readLine()) != null ) {
        if (line.equalsIgnoreCase("bye")) {
            shouldRun = false;
        }
        out.println("> " + line.toUpperCase());
        out.flush();
    }
    clientSocket.close();
    in.close();
    out.close();
}
```

It takes a long time to serve each client 

Single Threaded Multi Process Blocking

How apache httpd did it (with pre-fork, kind of...)

The server implements a loop.
It waits for a client to arrive.
When the client arrives, the server forks
a new process.

The child process serves the client while
the server is immediately ready to
serve the next client.

Forking a process is kind of heavy...
and resource hungry

While the child process serves the client...

... the parent can immediately welcome the next client.

```
while(1) { // main accept() loop
    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
&sin_size);
    if (new_fd == -1) {
        perror("accept");
        continue;
    }

    inet_ntop(their_addr.ss_family,
        get_in_addr((struct sockaddr *)&their_addr),
        s, sizeof s);
    printf("server: got connection from %s\n", s);

    if (!fork()) { // this is the child process
        close(sockfd); // child doesn't need the listener
        if (send(new_fd, "Hello, world!", 13, 0) == -1)
            perror("send");
        close(new_fd);
        exit(0);
    }
    close(new_fd); // parent doesn't need this
}
```

Multi Threaded Single Process Blocking

The 'old' Java way

The server uses a first thread to wait for connection requests from clients.

Each time a client arrives, a new thread is created and used to serve the client.

Millions of clients, millions of threads?

**Resource hungry.
Not scalable.**

The ReceptionistWorker implements a run() method that will execute on its own thread.

```
private class ReceptionistWorker implements Runnable {

    @Override
    public void run() {
        ServerSocket serverSocket;

        try {
            serverSocket = new ServerSocket(port);
        } catch (IOException ex) {
            LOG.log(Level.SEVERE, null, ex);
            return;
        }

        while (true) {
            LOG.log(Level.INFO, "Waiting for a new client");
            try {
                Socket clientSocket = serverSocket.accept();
                LOG.info("A new client has arrived...");
                new Thread(new ServantWorker(clientSocket)).start();
            } catch (IOException ex) {
                LOG.log(Level.SEVERE, ex.getMessage(), ex);
            }
        }
    }
}
```

As soon as a client is connected, a new thread is created.
The code that manages the interaction with the client executes on this thread.

2 types of workers, n+1 threads

Example: **07-TcpServers**



Multi Threaded Single Process Blocking

The 'old' Java way

The server uses a first thread to wait for connection requests from clients.

Each time a client arrives, a new thread is created and used to serve the client.

Millions of clients, millions of threads?

**Resource hungry.
Not scalable.**

The ReceptionistWorker implements a run() method that will execute on its own thread.

```
private class ReceptionistWorker implements Runnable {

    @Override
    public void run() {
        ServerSocket serverSocket;

        try {
            serverSocket = new ServerSocket(port);
        } catch (IOException ex) {
            LOG.log(Level.SEVERE, null, ex);
            return;
        }

        while (true) {
            LOG.log(Level.INFO, "Waiting for a new client");
            try {
                Socket clientSocket = serverSocket.accept();
                LOG.info("A new client has arrived...");
                new Thread(new ServantWorker(clientSocket)).start();
            } catch (IOException ex) {
                LOG.log(Level.SEVERE, ex.getMessage(), ex);
            }
        }
    }
}
```

As soon as a client is connected, a new thread is created.
The code that manages the interaction with the client executes on this thread.

2 types of workers, n+1 threads

Ubuntu-18.04

File Edit View Navigate Code Analyze Refactor Build Run Tools Git Window Help

TcpServers [...\RES\Teaching-HEIGVD-RES-2021-OL\examples\07-TcpServers\TcpServers] TcpServers.java

Project

Commit

Pull Requests

Structure

Favorites

Terminal

Build

Event Log

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

...

package ch.heigvd.res.examples;

/**

* This application shows the difference between a single threaded TCP server

* and a multi threaded TCP server. It shows that the first one is only able to

* process one client at the time, which is obviously not really an option for

* most applications. The second one uses n+1 threads, where n is the number of

* clients currently connected. The extra thread is used to wait for new clients

* to arrive, in a loop.

* The application starts the multi-threaded server on port 2323 and the

* single-threaded server on port 2424. Use several terminals and the telnet

* command to (try to) connect to the servers and compare the behavior.

* @author Olivier Liechti

*/

public class TcpServers {

/**

* @param args the command line arguments

*/

public static void main(String[] args) {

System.setProperty("java.util.logging.SimpleFormatter.format", "%5\$s %n");

MultiThreadedServer multi = new MultiThreadedServer(port: 22222);

multi.serveClients();

SingleThreadedServer single = new SingleThreadedServer(port: 11111);

single.serveClients();

}

}

8 items | 1 item

ASDFASDF

bye

BYE

ASDF

sdf

SDF

asdd

ASDD

IDEA

Git

TODO

Problems

Terminal

Build

Event Log

IdeaVim: Using the Ctrl+R shortcut for Vim emulation. // You can redefine it as an IDE shortcut or configure its handler in Vim Emulation settings. (10 minutes ago)

16:72 CRLF UTF-8 Tab* main

**Single Thread
Single Process
Asynchronous Programming**

The 'à la Node.js' way

The server uses a single thread, but in a non-blocking, asynchronous way.

Callback functions have to be written, so that they can be invoked when clients arrive, when data is received, etc.

**Different programming logic.
Scalable.**

We are registering callback functions on the various types of events that can be notified by the server...

```
// let's create a TCP server
const server = net.createServer();

// it reacts to events: 'listening', 'connection', 'close', etc.
// register callback functions, to be invoked when the events
// occur (everything happens on the same thread)

server.on('listening', callbackFunctionToCallWhenSocketIsBound);
server.on('connection',
callbackFunctionToCallWhenNewClientHasArrived);

//Start listening on port 9907
server.listen(9907);

// This callback is called when the socket is bound and is in
// listening mode. We don't need to do anything special.
function callbackFunctionToCallWhenSocketIsBound() {
  console.log("The socket is bound and listening");
  console.log("Socket value: %j", server.address());
}

// This callback is called after a client has connected.
function callbackFunctionToCallWhenNewClientHasArrived(socket) {
  ...
}
```

... and we code these functions, implementing the behavior that is expected when the events occur.



Select is a blocking operation (with a possible timeout). It blocks until something has happened on one of the provided sets of file descriptors.

Single Thread Single Process IO Multiplexing

The 'select' way

Sockets are set in a non-blocking state, which means that `read()`, `write()` and other functions do not block.

System calls such as `select()` or `poll()` block, but work on multiple sockets. They return if data has arrived on at least one of the sockets.

Watch out for performance.

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    fd_set rfd;
    struct timeval tv;
    int retval;

    /* Watch stdin (fd 0) to see when it has input. */
    FD_ZERO(&rfd);
    FD_SET(0, &rfd);
    /* Wait up to five seconds. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfd, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now.\n");
        /* FD_ISSET(0, &rfd) will be true. */
    else
        printf("No data within five seconds.\n");

    return 0;
}
```

Here, we know that something has happened on one of the sockets. We can iterate over the set of file descriptors and get the data.

End of part 2
End of chapter

