AARHUS UNIVERSITET

Software Engineering and Architecture

Server Threading

AARHUS UNIVERSITET

Motivation

- Servers are *reactive*
  - Process incoming requests for clients
  - … and if we are successful there are many *clients*…

1. Accept connection
2. Process request
3. Goto 1

```
while(! isStopped()){
    Socket clientSocket = null;
    try {
        clientSocket = this.serverSocket.accept();
    } catch (IOException e) {
        if(isStopped()) {
            System.out.println("Server Stopped.") ;
            return;
        }
        throw new RuntimeException(
            "Error accepting client connection", e);
    }
    try {
        processClientRequest(clientSocket);
    } catch (Exception e) {
        //log exception and go on to next request.
    }
}
```

AARHUS UNIVERSITET

Exercise

- What happens in the single-threaded server if
- A) A new request arrives before the last was processed?
- B) One of the requests takes 45 seconds to complete?

Motivation

- This is problematic if
  - *A new request arrives before the last was processed*
  - *The request process is slow to compute*

1. Accept connection
2. Process request
3. Goto 1

```
while(! isStopped()){
    Socket clientSocket = null;
    try {
        clientSocket = this.serverSocket.accept();
    } catch (IOException e) {
        if(isStopped()) {
            System.out.println("Server Stopped.") ;
            return;
        }
        throw new RuntimeException(
            "Error accepting client connection", e);
    }
    try {
        processClientRequest(clientSocket);
    } catch (Exception e) {
        //log exception and go on to next request.
    }
}
```

AARHUS UNIVERSITET

One Solution

- The classic solution is the *thread-per-request*
  - That is, every request creates a new thread

1. Accept connection
2. Spawn new thread
3. Goto 1

The thread then concurrently process the request!

```
public void run(){
    synchronized(this){
        this.runningThread = Thread.currentThread();
    }
    openServerSocket();
    while(! isStopped()){
        Socket clientSocket = null;
        try {
            clientSocket = this.serverSocket.accept();
        } catch (IOException e) {
            if(isStopped()) {
                System.out.println("Server Stopped.") ;
                return;
            }
            throw new RuntimeException(
                "Error accepting client connection", e);
        }
        new Thread(
            new WorkerRunnable(
                clientSocket, "Multithreaded Server")
        ).start();
    }
    System.out.println("Server Stopped.") ;
}
```

AARHUS UNIVERSITET

Exercise

- What happens in the multi-threaded server if
- A) A new request arrives before the last was processed?
- B) One of the requests takes 45 seconds to complete?

All Well Then?

- Classic teaching tells us that
  - Thread creation is expensive
    - slow and require much memory
  - Thread context switching is expensive
    - Slow to switch from thread 1 to thread 87

- Thus, if our server is hit by 10.000 requests in 1 second, it will become sluggish…

- Note: *Maybe classic teaching is not true!!!*

AARHUS UNIVERSITET

Availability Teaching

- Maybe, maybe not; but anyway

- *If the number N of incoming requests is high you get*
  - *N concurrent threads struggling for CPU time*
    - *They are all very slow*
  - *May run out of memory!*
    - *Meaning the server will simply crash*

A Low Availability Server ☹

- Quite fun experiment to make!
  - Java VM will terminate when 95% CPU time is spent on GC

AARHUS UNIVERSITET

Another Take

- The alternative solution is the *Thread-Pooled-Server*

1. Accept connection
2. Queue request for next idle thread
3. Goto 1

Ten threads then concurrently process the requests!

```
protected ExecutorService threadPool =
    Executors.newFixedThreadPool(10);
```

```
public void run(){
    synchronized(this){
        this.runningThread = Thread.currentThread();
    }
    openServerSocket();
    while(! isStopped()){
        Socket clientSocket = null;
        try {
            clientSocket = this.serverSocket.accept();
        } catch (IOException e) {
            if(isStopped()) {
                System.out.println("Server Stopped.") ;
                break;
            }
            throw new RuntimeException(
                "Error accepting client connection", e);
        }
        this.threadPool.execute(
            new WorkerRunnable(clientSocket,
                "Thread Pooled Server"));
    }
    this.threadPool.shutdown();
    System.out.println("Server Stopped.") ;
}
```

Exercise

- What happens in the thread-pooled server if
- A) A new request arrives before the last was processed?
- B) One of the requests takes 45 seconds to complete?

AARHUS UNIVERSITET

Discussion

- Thread pooled servers have characteristics that are important for high availability server construction
  - More responsive as seen from the client
    - Instead of 10.000 requests at time t1 and then 20.000 at t2, the thread pool *will* guaranty that at most 2.000 requests are handled
      - That is, (theoretically) 10 times faster processing at time t2
  - Protection against crashing due to out-of-memory
    - The pool size can be adjusted to the amount of memory available
    - (Give and take, if all requests are of the ‘eat memory’ type…)
  - Queue characteristics means bursts are handled
    - Bursts (sudden steep increase of requests) are handled by *graceful degradation*
      - Server comes slower but then catches up again; does not crash…

And…

- As far as I can read Jetty (= SparkJava engine) is thread pooled
  - But I haven't read the source code ☺