Multithreading technique improvement in processing Client Server communication by utilizing object oriented language

Eda Tabaku  
*Aleksander Moisiu University, e.tabaku@hotmail.com*

Kliv Shapllo  
*University of Tirana, k.shapllo@fti.edu.al*

Senada Bushati  
*Aleksander Moisiu University, bushatin@yahoo.com*

Follow this and additional works at: [https://knowledgecenter.ubt-uni.net/conference](https://knowledgecenter.ubt-uni.net/conference)

Part of the Computer Sciences Commons, and the Digital Communications and Networking Commons

**Recommended Citation**

Tabaku, Eda; Shapllo, Kliv; and Bushati, Senada, "Multithreading technique improvement in processing Client Server communication by utilizing object oriented language" (2018). *UBT International Conference*. 98.  
[https://knowledgecenter.ubt-uni.net/conference/2018/all-events/98](https://knowledgecenter.ubt-uni.net/conference/2018/all-events/98)

This Event is brought to you for free and open access by the Publication and Journals at UBT Knowledge Center. It has been accepted for inclusion in UBT International Conference by an authorized administrator of UBT Knowledge Center. For more information, please contact knowledge.center@ubt-uni.net.
Abstract. A comparison of an application’s performance, when a program is executed through multiple threads and only a thread to complete a specified task by using Java techniques. In this paper will be expounded how a Java programmer can profit by technology facilities. How to create a program that executes in a parallel way, that solves independent tasks not one after the other, but all at the same time. The application that is developed (the environment used is Eclipse) aims to speed up the time needed to download images from a specific website, but it is flexible for downloading images also from different sites. We have generate downloading of 15 images from a website. It will count the time needed in the two cases, will compare them and lead to the conclusion of the increase in time performance. It will reflect to a faster application for end user. It’s quite obvious that the multithread version of the application is faster. The timing performance is increased nearly by a factor of 6. This means, that building a nearly parallel functional application, is very important and will have a great feedback from the user. Saying all that, must be accepted that programming with threads is necessary, sometimes inevitable, but it needs caution, because complexity increases and new problems should be prevented.

Keywords: Timing performance, process, recourse management, Java, single-threaded, multiple-threaded.

1. Introduction

The earlier computers, could supported only one processor, as the heart of their system. This means that the performance in executing programs and different applications was limited only by the hardware capabilities of that CPU. So, if we had two programs that needed to run, the CPU had to switch execution time between
them, resulting to something that the user called multitasking. But to have true multitasking or true parallelism, one single microprocessor isn’t enough, two or more cores are needed. So, applications execution time is no more sliced by one CPU, but different CPU execute different programs, all at the same time, in a parallel way. This resource allocation is done all by the Operating System, using special algorithms. Going deeper in this subject is not the intention of this paper, because here will be expounded how a Java programmer can profit by these technology facilities. How to create a program that executes in a parallel way, that solves independent tasks not one after the other, but all at the same time. Here comes the concept of multithreading. One special task is done by one special thread. First, let’s explain some terms that are used in this work.

*Process*—is a program in execution.

*Thread*—refers to a sequence of instructions executed inside the context of a process.

*Application-level threads*—threads which are manages by threads libraries in user’s space.

*Task*—work that should be performed.

### 1.1 Threads and multithread concept

As mentioned above, a process is a program in execution and it can be divided into a number of individual or special parts that are called threads. So, a process may be thought as an only one thread in the case of single-threading or as a group of threads in the case of multi-threading. This concept is shown in figure 1:

![Fig. 1](image)

The beauty of programming in a multi-threaded way, is that thread-based multitasking has some great advantages like:

1) threads share the same address space, processes not.
2) inter-thread communication is inexpensive.
3) switching between threads in inexpensive.
At the end of the day, using multithreaded programming, we have this global results that are obvious to the user of the application:

• Improvement of performance.
• Improvement of program structure.
• Improvement of application responsiveness.
• Improvement in processors efficiency.

2. Treads in Java

Java is a powerful programming language, it is based upon processes as the architecture construct of the application. Processes may include one thread or multiple threads that share the same memory space, state and variables. In Java threads are objects and can be created in two different manners: one way is by implementing the java.lang.Runnable interface and the other way is by extending the java.lang.Thread class.

In the figures below, are represented these methods by them snippet code:

```java
import java.lang.*;

class Example extends Thread {
    public void run()
    {
        // body of execution
    }
}

Fig.2. Extending Thread class
```

```java
import java.lang.*;

class Example implements Runnable {
    public void run()
    {
        // body of execution
    }
}

Fig.3. Implementing Runnable interface
```
Each of these solutions has its own advantages. Applying the Runnable interface gives more flexibility on extending other base classes, but extending the Thread class gives a full control over the thread life cycle. Both of this use the run() method, which need to be overridden by the code that the new thread, after creation, needs to execute.

By the way, it’s important to mention something about a thread life cycle in Java.

![Thread life-cycle diagram](image)

**Fig. 4.** Thread life-cycle

Threads might be in five different states which are controlled by JVM. They are listed below:

1) **New** – this is a just born thread, the program hasn’t yet started the thread.
2) **Runnable** – in this state, the thread is considered to execute his task, after the program has started the thread.
3) **Waiting** – this is a transitory state, while the thread is waiting for another thread to finish something.
4) **Timed waiting** – threads enter in this state for a specified amount of time from the runnable state. When the time interval expires the thread goes back to the runnable state.
5) **Terminated** – thread enters in this state after completing its task.

The developing process of this small app doesn’t make use of the extension of other Java classes, but needs a little bit of thread control, so during coding is used extending the Thread class.
3. Application’s theory

The application that is developed (the environment used is Eclipse) aims to speed up the time needed to download images from a specific website, but it is flexible for downloading images also from different sites. The site that’s chosen is an Albanian e-commerce website http://www.123.al. A table of URLs is constructed for 15 images to be downloaded. Every table element contains an image URL. Every download is conceived like a thread task. The idea is that are created two Java classes, one that supports only one thread and one that supports more than one thread (multi-threads). The first script when executes, creates a thread which task is to iterate through the URLs table and download the image that addresses each URL. Is measured the time when thread began running and the time it terminated, meaning that all images were downloaded. Getting time is done by the Java language method System.currentTimeMillis(). The second script, the multi-threaded one, creates 15 threads for 15 images to download. So each download is a different thread task, in this way are created 15 parallel tasks (downloads). The time in milliseconds when the first thread began working and the time when the last thread finished working are measured. Making the difference is taken the time needed for this small multithreading app to end its job. Below is given an example of the application output in the two cases.

- Getting 15 imgs by one thread
- Application terminated in 5629ms

Fig.5. Single-thread application output

- Getting 15 imgs by multiple threads
- Application terminated in 900ms

Fig.6. Multi-thread application output

As it is clear by the screenshots, in the case of a multithreaded app, it produces a huge time profit and a faster app. In the last situation it is used a barrier concept, which means that the main thread creates and starts a number of processing threads and then waits for them to finish. The main thread waits in a barrier. A public function is constructed in a file Lock.java, where are counted all running threads. In the screenshot that follows are presented the lines of code which create the barrier.

```java
while (lkVar.getActiveTh() > 0) {
    synchronized (lkVar) {
        lkVar.wait();
    }
}
```

Fig.7. The barrier concept
First, the main thread waits (stays inside the while loop, the barrier), for all thread to execute and then terminates the program. Below is given the full multi-threaded source code in Java:

```java
public class multipleTh {
    public static void main(String[] args) {
        String[] imgs = new String[]{
            "http://www.123.al/uploads/coupons/_06Xq.jpg",
            "http://www.123.al/uploads/coupons/_06Xq.jpg",
            "http://www.123.al/uploads/coupons/_06Xq.jpg",
            "http://www.123.al/uploads/coupons/_06Xq.jpg",
            "http://www.123.al/uploads/coupons/_06Xq.jpg",
            "http://www.123.al/uploads/coupons/_06Xq.jpg",
        }

        String destF = "C:\\Users\\user\\Desktop\\";
        try {
            int ind = 0;
            long time1 = 0;
            long time2 = 0;

            Lock lkVar = new Lock();
            System.out.println("Getting "+imgs.length+" imgs by multiple threads");
            time1 = System.currentTimeMillis();

            for (String address: imgs) {
                DownThthr = new DownTh(address, destF + "multi" + (ind++) + ".jpg", lkVar);
                thr.start();
            }

            while (lkVar.getActiveTh() > 0) {
                synchronized (lkVar) {

                }
            }

            time2 = System.currentTimeMillis();
            System.out.println("Time spend: "+ (time2 - time1) + " ms");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```
lkVar.wait();
}  

time2 = System.currentTimeMillis();
System.out.println("Application terminated in: "+ (time2-time1)+" ms");
} 
} 

Fig. 8. Multi-threaded source code in Java

It’s not necessary to show also the source code for the single-threaded application since it is simpler. It doesn’t have a barrier concept etc.

4. Experiment statistics

After explaining some key things of the work and source code, is better to continue with the statistics about the program execution and timing performance. Different downloads are done with 15, 30, 45, 60, 75 and 90 images and measured their download time as shown in the table:

<table>
<thead>
<tr>
<th>Images number</th>
<th>Times in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>5629 ms</td>
</tr>
<tr>
<td>30</td>
<td>10654 ms</td>
</tr>
<tr>
<td>45</td>
<td>15463 ms</td>
</tr>
<tr>
<td>60</td>
<td>27144 ms</td>
</tr>
<tr>
<td>75</td>
<td>33086 ms</td>
</tr>
<tr>
<td>90</td>
<td>39066 ms</td>
</tr>
</tbody>
</table>

Fig. 9. Download time of single-threaded app
<table>
<thead>
<tr>
<th>Images number</th>
<th>Times in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>900 ms</td>
</tr>
<tr>
<td>30</td>
<td>1407 ms</td>
</tr>
<tr>
<td>45</td>
<td>2141 ms</td>
</tr>
<tr>
<td>60</td>
<td>4059 ms</td>
</tr>
<tr>
<td>75</td>
<td>4510 ms</td>
</tr>
<tr>
<td>90</td>
<td>6203 ms</td>
</tr>
</tbody>
</table>

Fig. 10. Download time of multi-threaded app

It’s important to be noted that sometimes, for example if the same download is executed (same number of pictures and same code) more than once, download time may differ with a specific amount of milliseconds. This is explained only by the fact of a changing network load in time, that affect downloading speed and packets forwarding. To make the comparison between the two tables more visible, let’s see the following chart:

In the chart above, in the X axis is presented the number of images to download and in the Y axis is the time in milliseconds.

5. Conclusions

If only superficially analyzing the data from tables or if only looking at the presentation in the chart, it’s quite obvious that the multithread version of the
application is faster. The timing performance is increased nearly by a factor of 6. This means, that building a nearly parallel functional application, is very important and will have a great feedback from the user. Even that programming with multithreads that run concurrently to make solve a specific task has this massive time profit and an improved structural frame of the app, improved responsiveness etc, here arises two problems that are typical for programs with more lines of code and more complex that the one showed in this paper so far. One problem is the read/write problem and the second one is the producer and consumer problem. The first, has to do with the fact that one thread tries to update the data that one other thread is reading. This leads to an inconsistent state of the shared data and the only solution is the synchronization of the thread access in the data. The second problem with threads has to do with two threads, that put and read data from a buffer. What arises here is that is must be provided a way that a thread can’t read an empty buffer and can’t write on a full one. Saying all that, must be accepted that programming with threads is necessary, sometimes inevitable, but it needs caution, because complexity increases and new problems should be prevented.

References

2. Effective Java – Joshua Block
3. Concurrency Practice in Java – Brian Boetz, Tim Pkierlin, Joseph Boweer, David Holmes, Dous Lea.
4. The New Wave – Concurrent Programming – Charless W. Kann
11. Towards Detecting Thread Deadlock in Java Programs with Introspection – Yan Wen, Jinjing Zhao, Minhuan Huang, Hua Chen.