Mini-projet : Serveur Web

6 mars 2013
Rendu: 17 avril 2013

Compte pour 20% de la note finale

En dehors de la description ci-dessous, seules les informations complémentaires données sur Moodle font foi (pas les informations données oralement).

Le projet s'effectuera par groupes de 2 étudiants au maximum. ***La conception de la solution et son implémentation seront faites séparément dans chacun des groupes, et ne seront pas partagées entre les groupes.***

Les classes du paquetage `java.util.concurrent` **ne pourront être utilisées, sauf les classes mentionnées explicitement ci-dessous.**

I Goal of the project

The goal of this project is to implement a simple concurrent web server. This web server will support only static content, which will be read from a local directory in the file system of the host running the web server. Furthermore, it will support only the GET and HEAD methods of the HTTP/1.1[1] protocol.

The focus of this project is to make the web server highly concurrent. On the one hand, it should process several client connections in parallel, so that a client connection is not blocked waiting for an answer while another connection is being processed by the server. Additionally, when a single connection is used by the client for several HTTP requests the web server should also be able to process the individual requests in parallel. The goal of the second requirement is to decrease the response time when the client is using HTTP Pipelining. This is an optional feature of HTTP/1.1, where a browser is allowed to send additional requests in a connection before having received the answer to the previous requests.

II Project outline

The project is split into 3 stages. The purpose of the first stage is to get familiar with the problem while using a minimum of concurrency concepts. In the second and third stage you will improve the server, by using more advanced concurrency techniques. Your submission must include working code for stages 2 and 3.

Submission of your solution will occur only after all three stages, see dates and deadlines below. To facilitate testing please make sure that the two versions of your project can be built and run using the ant[2] script. That is, `ant run-stage2` and `ant run-stage3` shall compile and run the second and third stage of the project. We will provide a template of the ant script (`build.xml`) which you can use directly or which you can adapt to your class names. Your submission must contain the following items:

[1] `http://www.w3.org/Protocols/rfc2616/rfc2616.html`
[2] Instructions on how to install Apache Ant can be found here: `http://ant.apache.org/manual/install.html`

- source-code
- ant script
- a short description of your design choices in a file `readme.txt`.

All files shall be submitted in a single zip file, named by the family names of all the group members (without accents). Failure to adhere to any of the points mentioned above might result in your project not being graded. If in doubt, please ask on the forum. The link to submit the zip file will be provided later on Moodle. The dates for the project stages are the following:

- *Stage 1 :* Starting date : March 6, 2013
- *Stage 2 :* Suggested starting date : March 13, 2013
- *Stage 3 :* Suggested starting date : March 27, 2013
- Project submission date : April 17, 2013

III HTTP protocol

This section is for information only. You will not have to implement the protocol.

III.1 Request format

Une requête HTTP a le format suivant (voir exemple ci-dessous). La première ligne contient (1) la méthode spécifique à la requête, (2) une URL indiquant l'objet de la requête (par exemple, une page internet), ainsi que (3) la version du protocole HTTP utilisé. Pour simplifier, on suppose que toutes les requêtes utilisent uniquement les méthodes `GET` et `HEAD`.[3] Dans le cas de ces deux méthodes, le reste de chaque requête est uniquement constitué d'une liste d'entêtes, chaque ligne de texte correspondant à une entête et étant composée d'un identifiant (nom de l'entête) et d'une valeur :

```
GET http://www.epfl.ch/ HTTP/1.1
Host:  www.epfl.ch
...
Connection:  close
```

} Liste d'entêtes

Pour simplifier, on considère uniquement la version 1.1 du protocole HTTP. Cela implique que chaque requête contient une entête `Host` dont la valeur correspond au serveur devant traiter la requête. Cette entête a le format ci-dessous. Notez que la mention du port est optionnelle. Le port utilisé par défaut est le port 80.

`adresse_serveur` [":" `port`] (par exemple, `www.epfl.ch` ou `www.google.ch:8080`)

Les réponses HTTP ont un format légèrement différent de celui des requêtes (voir exemple ci-dessous). La première ligne contient (1) la version du protocole utilisé, (2) le code du statut de la requête (par exemple, 404 si l'objet de la requête n'a pas été trouvé), et (3) une explication du code de statut utilisé. Similairement aux requêtes, les lignes qui suivent contiennent une liste d'entêtes. Enfin, le reste de la réponse contient l'objet requis par la requête correspondante (dans l'exemple ci-dessous, un fichier HTML) :

[3] Notez que les autres méthodes sont très peu utilisées et ne sont pas tenues d'être implémentées par un serveur HTTP.

```
HTTP/1.1 200 OK
Date:  Thu, 08 Jan 2009 13:05:24 GMT
 ...
Content-Length:  14420
Connection:  close

<html>
 <head>
 ...
 </head>
 <body>
 ...
 </body>
</html>
```

Liste d'entêtes

Résultat de la requête

III.2 Persistent connections

HTTP/1.1 requires that web servers support persistent connections[4]. This means that a TCP connection established between a browser and a server can be used by multiple HTTP requests, for instance, to read multiple images for a single page. Reusing a connection significantly decreases the time needed to load a page and decreases the CPU requirements in both the client and the server.

III.3 HTTP Pipelining

HTTP/1.1 also allows the use of HTTP Pipelining[5], as an optional functionality. Without pipelining, a client must wait for the answer to the previous request before sending another request in the same connection (different connections are independent). Pipelining allows a client to send requests without waiting for previous answers, which can also decrease the page load time in high latency connections. However, when using pipelining the web server **must** send the answers in the same order as the corresponding requests were received.

IV Code provided

In order to simplify the project we provide the following classes. We will publish a zip file that contains the necessary jar files, the aforementioned ant script and the Javadocs for these classes. You can use the content of the zip file directly as a starting point for your project (when using eclipse).

- `HttpRequestStream` - Reads a series of HTTP requests from a Java input stream (i.e., `java.io.InputStream`)
- `HttpRequest` - Represents a request.
- `HttpResponse` - Represents a response.
- `StaticSite` - Generates the response to an HTTP request, by retrieving the resource requested from the local directory.
- `Configuration` - Loads and exposes a set of configuration properties. The properties are read from the file `server.properties` located in the current working directory. This file should follow the standard structure of a Java properties file, i.e., one key-value pair per line written as `key = value`.

  Here is an example of the `server.properties` file:

[4] `http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html#sec8.1`
[5] Section 8.1.2.1 of the HTTP/1.1 specification

```
site.root=site
server.port=8080
stage2_thread_pool_size=4
```

The `site.root` property is required by the class `StaticSite`. It indicates the directory storing the static content provided by this web server. The other properties shown above are not used by the classes provided, they are merely examples of properties that a full implementation of this project may use. Solutions are not required to use the properties file, but it is recommended as a more convenient alternative to either hard-coded values or command line arguments.

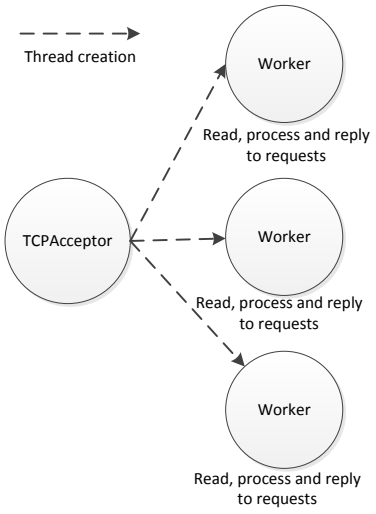
V Description of the project stages

V.1 Stage 1

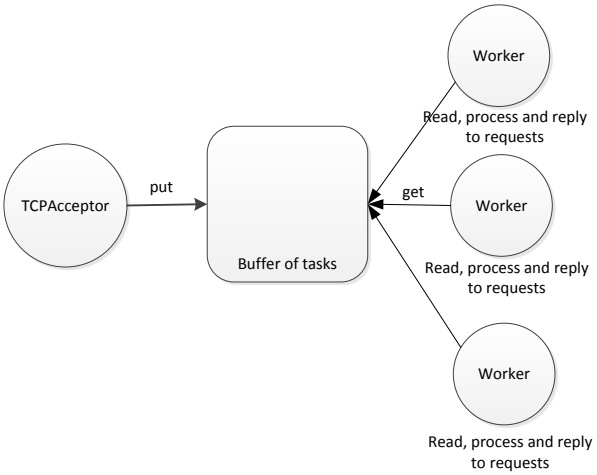
The first stage consists of implementing a minimal multi-threaded web server. The TCPAcceptor thread waits for TCP connections from browsers, and whenever a new connection is established (i.e., `ServerSocket.accept` returns), it creates a new worker thread to handle *all* the requests received from this connection (see Figure 1a). The worker thread should process requests sequentially. Processing a request consists of three steps: Firstly, the request has to be read from the socket, then a matching response has to be generated, which is finally written to the socket. In order to simplify handling requests we have provided classes that help with each of these steps (see Section IV and the classes' API documentation). As we have mentioned in Section III.2, handling one connection can involve handling multiple requests, so handling requests should be done in a loop.

To test the web server in your local machine, type in the web browser the address `http://localhost:8080`. The port number should match the port where your server is listening for new connections.

Hint: To make transition to Stage 2 simpler, when creating worker threads do not extend `java.lang.Thread`, but rather use the constructor `Thread(Runnable target)`.



(a) Threads in Stage 1 (workers are created dynamically)



(b) Threads in Stage 2 (fixed set of workers)

Figure 1: Stages 1 and 2

Remark: As mentioned above, this stage is not mandatory. However, unless you feel comfortable with the notions needed to implement this stage, we recommend that you do not skip stage 1.

V.2 Stage 2

In stage 1 every new connection results in the creation of a new worker thread. This is not efficient, as connections are usually short-lived and creating a new thread is expensive, so that the server may end up wasting many resources creating and destroying threads. The goal of stage 2 is to improve this basic design by reusing worker threads to handle multiple connections.

In stage 2, the server maintains a fixed set of worker threads that are created initially and then handle one connection at a time. Once the connection is closed, the worker thread becomes available to handle a new connection. This can be recognized as an instance of the producer consumer problem with a single producer and a fixed set of consumers. The TCPAcceptor is a producer that generates `Sockets` and worker threads are the consumers.

Since another producer-consumer problem will arise in Stage 3, rather than solving this problem (directly), you will have to implement a generic solution. That is, you have to implement a buffer and producers-consumers for generic tasks. Here a task is any object that implements the interface `java.lang.Runnable` and a consumer can execute the task by calling the `run()` method of the object. Once the `run()` method returns the task is consumed and the consumer, i.e., the worker thread, can obtain the next task. As in the lecture the buffer should have a fixed size, such that the producing thread will block when it is full.

Allowed Classes: Implement the buffer using semaphores (`java.util.concurrent.Semaphore`)

V.3 Stage 3

As discussed in Section III.3, HTTP/1.1 supports pipelining of requests in a single connection. In the implementation of Step 2, requests received from a connection are executed sequentially by the worker thread assigned to this connection. This simple strategy is not optimal, because a pipelined request will only start being processed when all the previous requests in the same connection are processed, unnecessarily increasing the response time. A better solution is to process requests in parallel. In order to do so, we introduce a second buffer of tasks, each task being responsible for generating responses, synchronizing and then writing them to the sockets (see Figure 2).

Recall that pipelining requires that the responses are sent in the same order as the requests were received. Note that even though the requests may start being processed in the order they were received, they can terminate in a different order. Consider a request to retrieve a large image is followed by one to retrieve a thumbnail, it is likely that the second request will finish before the first.[6] In this case, your implementation has to delay sending the answer to the second request until the answer to the first has been written.

We ask you to implement delaying sending the answer as follows. You should coordinate the threads that generate and write responses of the *same connection* using a common `BlockingCounter`, which is defined by the following interface

```
interface BlockingCounter {
   void await(int number);
   void increment();
}
```

Initially, the value of the counter is 1. The task T_k handling the k-th request of the corresponding connection, will use `await(k)` to block until the counter has reached the value k. Moreover, once T_k has written its response, it increments the counter.

As mentioned above, you should use two buffers of tasks, one for tasks that read requests and one for tasks that perform the remaining processing. The solutions to both stages should use the same buffer class.

[6] To make it even more apparent consider the thumbnail file being cached in memory, while the large images has to be fetched from disk.

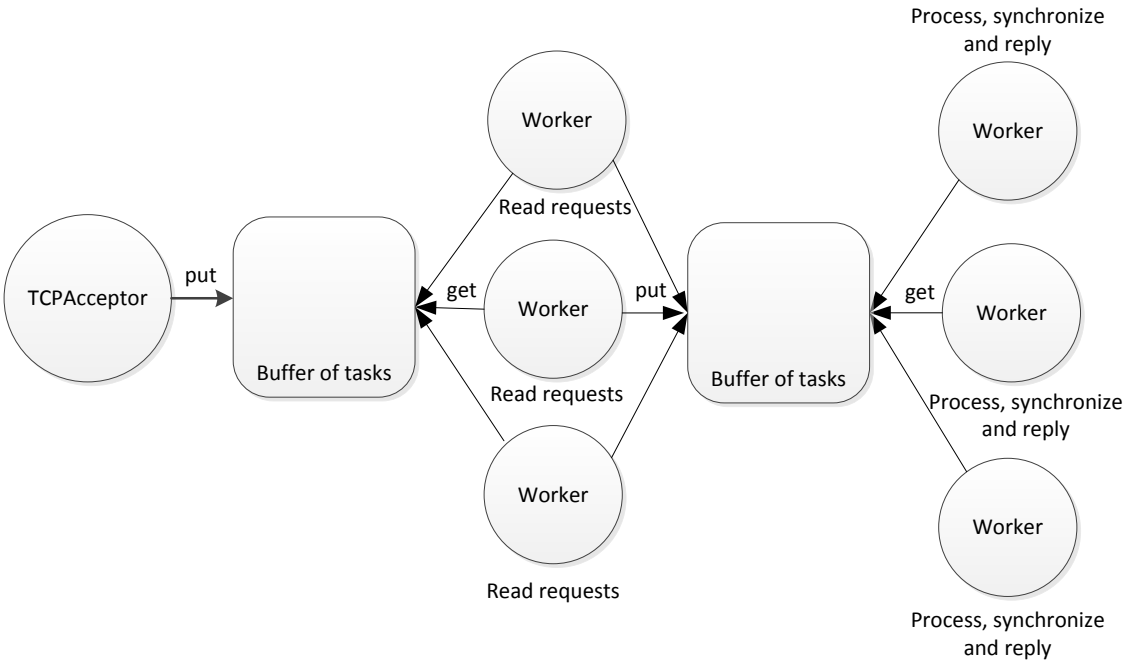


Figure 2: Threads in Stage 3

Question: Is it necessary to have two buffers (and two sets of threads) or is it possible to schedule all tasks using just one buffer? Explain in `readme.txt`.

Allowed Classes: Use either Java conditions (`java.util.concurrent.locks.Condition` and `ReentrantLock`) or Java Monitors to implement the `BlockingCounter`.

VI Testing of pipelining (stage 3)

In order to facilitate testing (and seeing the benefits) of pipelining we provide the `site.delay` property. When `site.delay` is non-zero, `StaticSite` delays the creation of every `HttpResponse` by the duration specified (milliseconds).

The classes provided also include a *client*, which will download all pages in `site.root` and print statistics about the time it took. The client expects two arguments on the command-line: (1) the number of parallel threads to use for downloading (number of simultaneous connections opened) and (2) the amount of pipelining (if the second argument is x, the client waits for the replies after having sent x requests). You can also invoke the client using `ant run-load-tester`. Depending on the sizes of the two sets of threads, playing with these two parameters should give quite different results.