Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

SISTEMAS Y SERVICIOS DISTRIBUIDOS

LAB 3: PROGRAMMING OF DISTRIBUTED SYSTEMS (2)

IMPLEMENTATION OF CONCURRENT SERVERS

Teachers:

Antonio Guillén Pérez

Esteban Egea López

INDEX

Contenido

1. Goals.

- Understanding the problem of attending concurrent requests.
- Knowing design alternatives for concurrent servers.
- Learning the use of Java Threads.

2. Concurrent requests

In the prior lab you programmed a simple server able to attend the request of a single customer using a Socket. The problem is that the server runs sequentially. When it receives a request it is directly processed. If, while processing the request, another one is received, the server is not able to deal with it at the same time. In a real system, attending a request can take considerable time and if the system supports a significant load it is clear that we need a mechanism to address concurrent requests. The solution to the problem is to use a mechanism that allows us to attend requests in parallel. Since the S.O is multitasking (although it actually works sequentially, using temporal multiplexing of process, unless it has multiple CPUs), we can use different processes to serve concurrent requests in a "parallel" way. In practice, there are several alternatives, as discussed in the next section.

3. Processes and threads.

In most systems there are two mechanisms to implement independent execution units: processes and threads. Typically, a process is defined as an instance of a running program. A thread, however, is generally defined as a sequence of instructions that are independently managed by the OS scheduler. The implementation of threads may differ depending on the OS. To narrow the definition, in our particular case we will say that: *the fundamental difference between threads and processes is that the first shared memory while the latter are completely isolated by the OS*. Or else: a process can consist of multiple threads, which run in parallel and share the memory space, whereas different processes are completely isolated from each other by the OS and they require inter-process communication mechanisms to exchange information.

In Figure 1.1, 2 processes are shown, made of two thread and how the OS temporally multiplexes the CPU cycles allocated to each of them. That is, if a process is given some runtime of CPU, the OS will divide the time among all the threads that make up the process. Such distribution does not have to be equal. In fact, the allocation algorithm depends on the OS. In addition, in current multiprocessor systems, execution of threads is split between the different processors available.

Threads are also called lightweight processes. The reason is that the cost in resources (processing power) of the context switching (the replacement of a running thread by another one) of a thread is much lower than that of a process.

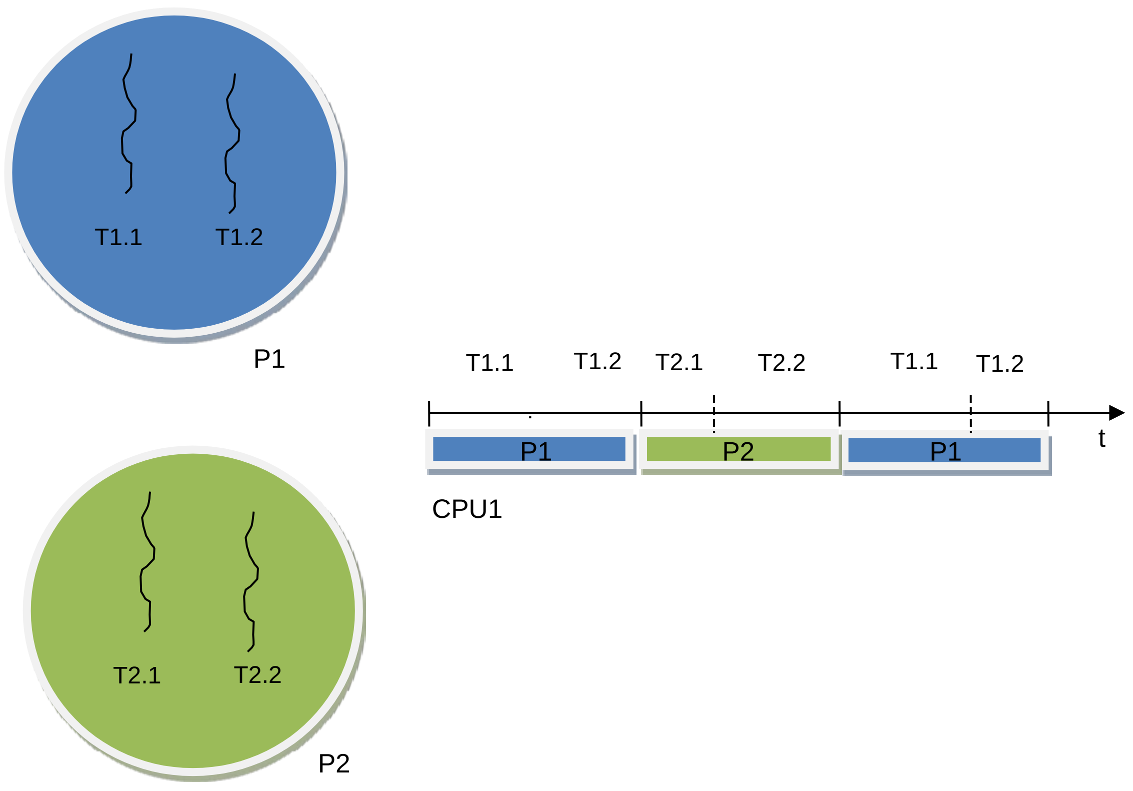

Figura 1.1

4. Design of concurrent servers

We need to deploy servers that can serve concurrent requests, ie, perform tasks in parallel. To do this, we have the possibility of using processes or threads. In this section we will see that these two alternatives can be refined to improve the performance, resulting in the following:

1. A process per client.
2. A thread per client.
3. Multiple pre-forked processes.
4. Multiple pre-allocated threads.
5. Using asynchronous input and output.

4.1 A process per client

This is the classic and basic option in Unix servers. The use of threads is a feature that has not been available in processors and OS until a few years ago. What has been available is the ability to replicate a process in memory. For this purpose a system function called fork() is used. This function replicates the code in memory of the calling process in another area of memory and the execution is resumed in the next instruction. That is, an exact replica of the process is generated, called a child process. The only difference is that the generated child process receives a different process identifier.

In practice, the usual implementation is that the instruction following a call to accept () is a call to fork(). Thus, each new request is served by a child process. The pseudocode is asfollows:

```
while (true)
accept ();
if (pid = fork ()) == 0) {// This is the child. The process
// replicated (child) attends the request while the father returns to accept ()
     attendRequest ();
    break;
}
}
```

4.2 A thread per client

Generating and running a new process is costly in resources. In this case, the mechanism is the same as in the previous case, with the difference that it now creates a new thread for each client. The advantage is that creating and executing a thread is much less expensive. There must be, however, careful management of shared memory and use of appropriate synchronization mechanisms.

In a Unix system (POSIX, actually), the call would be to pthread_create (). In Java is different as discussed below. The pseudocode is as follows:

```
while (true)
     accept ();
     pthread_create (attendrequest & ...) // The thread attends the request
      // While the father returns to accept ()
      break;
     }
}
```

4.3 Multiple pre-allocated processes or threads

Both generating a process and a thread for each request is expensive and adds a delay to the execution of the request. A more efficient alternative is to create a set or pool of children of process or threads ready to handle requests. That is, at the start, a set of child processes or threads ready to serve requests is generated. Each request is attended by one of the children and when it ends, it becomes available to attend the next request. Requests are allocated among the available processes. The parent process is responsible for managing the creation of new processes, in a dynamic way in many cases. There are multiple options for the implementation: child processes can be finished or reused, the algorithm to assing requests or generate new processes, etc. Wewill not go into details here.

Usually, the use of threads is more efficient in terms of memory and processing load. However, it is necessary to properly manage the shared memory and, as we said, the possibility of using threads has not traditionally been available. Many libraries do not guarantee correct operation when used with threads. When they do, they are called thread-safe. Therefore, due to lack of required libraries it is necessary often to resort to the use of child processes instead of threads.

4.4 Using asynchronous input and output.

A traditional alternative to using threads is the use of asynchronous input and output. In this case, the OS select() function is normally used. This function allows to monitor the status of multiple sockets simultaneously, which are passed as a parameter to the function. The process goes to sleep when invoking the function and it is waked up when there is a change in their status, either because of new connection requests, or because new channel data have arrived. The process then iterates through the set of monitored sockets serving those with new data or requests.

This alternative can be combined with the use of threads, ie, threads can be executed to handle requests when the select() function has returned.

4.5 Java Threads

In the case of Java, the Java Virtual Machine (Java Virtual Machine, JVM) runs as a process and supports the use of threads, so it is usual to use Java Threads and implement servers based on multiple threads.

The threads are responsible for processing specific tasks to run in parallel, therefore, it is necessary to implement this task (thread task). To this purpose, the Runnable interface declares a run() method. Any class that is used to perform a thread task must implement this interface and therefore the method run(). That is, the method run() is invoked when the thread is started, it is the entry point. That is, it would be the equivalent of the main() method in a regularclass.

To really create a new thread, you have to create an instance of the Thread class. The Thread class requires in its constructor to be passed an instance of a class that implements Runnable. When the start() method is invoked in Thread a new thread will be created and the run() method of the class passed to the constructor will be executed. Immediately, the control is returned to the caller of the start(). At this time we have at least two running threads and their execution will be multiplexed.

The lifecycle of a Thread has to be managed. From Java version 5, the Executor class was introduced, which simplifies the management of multiple threads and allows the execution of asynchronous tasks. A ExecutorService is generated by a static call to Executors, such as:

```
ExecutorService = Executors.newCachedThreadPool exec ();
```

It is afterwards passed tasks (classes that implement Runnable) to be executed in different threads. That is, a set of threads is pre-allocated and the manager (ExecutorService) use them to process tasks, assigning them dynamically, and reusing threads.

You can create different types of sets or pools of threads. You can use FixedThreadPools, CachedThreadPools and even SingleThread, ie, with a single active thread. The choice depends on the needs of the application.

The resources must be released in an orderly manner with a call to the shutdown() method ExecutorService.

Java.util.concurrent package documentation is located at:

http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

6. Selectors and channels in Java

Java provides asynchronous input and output with the java.nio package. It works similarly to the Unix select() [1]. The operation is as follows: a Channel object representing an input or output source such as a file or a socket is used to read and write. A Channel object is associated (registered) a Selector object. This object is responsible for monitoring multiple channels. That is, you can register multiple channels with it. Registration is necessary to indicate what type of events are to be monitored.

The typical sequence would be: instances of ServerSocketChannel and Selector are created. The selector and channels are registered, indicating the event type to be monitored with a SelectionKey object. Afterwards, select() is invoked, which is a blocking call. When an event registered occurs on the channel, the function returns. At that time, the keyset of events selected is traversed with an Iterator<SelectionKey> and the event isprocessed.

Note that this mechanism may be combined with the use of threads for processing the events, for example.

7. Exercises

Use Eclipse to develop the following programs. In this lab, you will extend the simple echo server from the previous lab to make it concurrent, using Java threads, with Executors. **For this lab you can reuse the client that you implemented in the previous lab**. You will need to develop the following classes, which are detailed in the following sections:

- A class that represents the task that the server executes when it serves a request. Notice that a server could execute multiple tasks depending on the request made, for example, ReadFile, RegistrarUsuario, etc. Each of these specific tasks will be implemented in a different class and the server will execute them in a thread when it serves the different requests. In our case, we will only have one task that will be in charge of reading the string it receives from the client and returning it to them.
- A class that implements the concurrent server. The class simply takes care of generating pre-launched threads, accepting connections and starting the task that attends them in a thread.
- As usual, a class that starts the server. In this case, we will reuse the StartServer class from the previous lab adapted to this one.

a) Implement a class called EchoTask that will attend each new connection or request. This class must implement the interface Runnable since it will run on a thread. This class will be associated with

a corresponding thread. This class will have a static identifier which will serve to distinguish each instance created.

- This class has to implement the run() method as it implements the Runnable interface. Inside this method you simply have to read / write from the Socket as you did in the previous practice.
- The run() method is declared in the Runnable interface and its declaration cannot be modified to make it throw exceptions. Therefore, in this method we will have to use try / catch blocks to handle the possible read / write exceptions on the Socket.
- You will need to provide a constructor for this class to which the server will pass whatever it needs to do its job and initialize the variables. What does the server need to pass to it in order to implement the class?

b) Develop a server that uses multiple threads to address the request. This class will be called <u>ConcurrentServer</u>. Use a CachedThreadPool first and then a FixedThreadPool.

- This class will have a runServer() method that will create the thread pool and then implement the basic server loop you did in the previous lab. When it receives a new connection it will instantiate an EchoTask and pass it to the ExecutorService to be executed by one of its threads.
- To which design alternative of those described in section 4 would each case correspond approximately? Justify it.
- Run your server using the StartServer class.
- Use telnet or your Java client to create multiple concurrent connections and check if there is indeed a match with the mentioned design alternatives. Describe your experiment, the results, and explain them.

c) Develop a simple chat server, which retransmit the data sent by each of the clients connected to other ones. Use the model you prefer. Describe yoursolution.

This exercise supports several solutions. You can try your own. One possible solution is the following:

- The server will not serve the connections with tasks on threads, but will store all the instances of ChatTask in a data structure and will forward the messages from one client to another. You will implement two classes with various methods: ConcurrentChatServer and ChatTask.
- ChatTask is the task that a handle client requests and is passed to a thread. When it receives a message over the socket, it will invoke the server's deliverMessage(String s) method. In this method, the server will invoke the sendMessageToClient(String s) method in the rest of the ChatTask instances (clients).
- Additionally, the server will have a method that can be invoked to remove a client from the list and ChatTask will have a method so that the server can establish an index for it.

d) Finally, have your peers use their chat from different machines. Capture exchanges with wireshark.

e) Develop a server that serves multiple concurrent request using asynchronous input and output, ie, using Java NIO Selector and Channel. Call this class NonBlockingIOServer.

- This class will implement the runServer(int port) method. You will have to use NIO to read / write Sockets. So you will have to use the SocketChannel class and ByteBuffer to read / write to the channel.
- It is necessary to use SelectionKey to register the events that we are interested in attending, among all the ones which can happen in the channel. In our case, both OP_ACCEPT and OP_READ.
- The select() blocking function will be called to wait for events to occur.
- Once an event occurs, the select() function will return and all the SelectionKey keys will be traversed. If the event was an OP_ACCEPT, it will be registered with the OP_READ read event, which warns when there is data. If this event occurs, the channel string will be read and rewritten to the channel.

References

[1] http://beej.us/guide/bgnet/output/html/multipage/index.html

Bibliography

1. Bruce Eckel, Thinking in Java, 4th Edition, Prentice Hall, 2006
2. (Las versiones anteriores están disponibles gratuitamente en formato electrónico en http://www.mindviewinc.com/Books/downloads.html)
3. David Reilly, Java Network Programming and Distributed Computing, Addison-Wesley, 2002.
4. Al Williams, Java 2 network protocols black book, Coriolis Groups Books, 2001.
5. Stevens, R., Unix Network Programming, vol. 1, Addison Wesley, 2004, 3rd Edition.