Concurrency in Java and in Erlang

Sven-Olof Nyström
Department of Information Technology,
Uppsala University, Sweden
svenolof@csd.uu.se

May 10, 2004

1 Concurrency in Java

Concurrency (and Java's threads in particular) is a very important tool for writing programs that must communicate with many other programs or with several people. The most important example of a concurrent system is perhaps multi-tasking operating systems which allow several programs to be run at the same time. In a web server, concurrency means that a single web server can server many requests at the same time, and a request that might take very long to serve, for example, downloading a large document over a slow connection, will not block another request that can be served faster.

In situations such as these, threads are not strictly necessary, but it often turns out that the simplest and most efficient solution involves multiple threads. The concept of threads offers an attractive alternative to regular processes, as they are more light-weight (due to the lack of memory protection). However, other types of processes have even lower overhead.

Java's threads are rather nicely integrated with the class system. The standard API defines a class `Thread` which can either be used directly, or inherited from. Unfortunately, the implementations use the threads of the underlying operating system, which means that threads are threads are expensive (threads are cheaper than OS processes, but creating a thread is many orders of magnitude more expensive than, say, creating an object). Many operating systems only allow a very restricted (a few hundred) number of threads. What is even worse is that the behavior of threads depends on the operating system, so that a Java program written for one OS might not work when run on an other OS.

When two Java threads in an application communicate, they normally do so by updating shared objects. An update of an object might involve several operations, and to ensure that no other process can access an object while one process is updating it, Java offers primitives for synchronization.

Unfortunately, synchronization is relatively costly (compared to the same accesses without synchronization). Worse, synchronization may sometimes cause deadlocks. But not doing synchronization will result in race conditions, which is equally bad.

An older version of Sun's tutorial *How to use threads* began with the following not so encouraging words:

> The first rule of using threads is this: avoid them if you can. Threads can be difficult to use, and they tend to make programs harder to debug. To avoid the possibility of deadlock, you must take extreme care that any threads you create don't invoke any methods on Swing components.

2 Concurrency in Erlang

Erlang [2] is a concurrent language for telecom applications developed by Ericsson. Its sequential component is a simple, dynamic functional language. It also has primitives for process creation and message passing.

The advice offered to Erlang programmers regarding the use of processes [3] has a much more optimistic ring than the corresponding comment in Sun's tutorial:

> Assign exactly one parallel process to each true concurrent activity in the system [...] When deciding whether to implement things using sequential or parallel processes then the structure implied by the intrinsic structure of the problem should be used. The main rule is:
>
> "Use one parallel process to model each truly concurrent activity in the real world[.]"
>
> If there is a one-to-one mapping between the number of parallel processes and the number of truly parallel activities in the real world, the program will be easy to understand.

In other words, never hesitate to use processes when it seems that the problem calls for it. This advice may seem irresponsible, but the truth is that Erlang has a good track record regarding reliability. Ericsson has successfully developed and shipped many large and complex products developed using Erlang. For example, Ericsson's ATM switch, the AXD 301, is the most sold ATM switch in the world. Its software contains 1.7 million lines of Erlang code.[1] Measurements indicate 99.99999999% reliability, i.e., a downtime of less than 31 milliseconds per year [1]! Beside Ericsson, its competitor Nortel also ships a number of products based on Erlang.

Let's start with an example of a simple sequential Erlang program.

```
fac(0) -> 1;
fac(N) when N > 1 -> N * fac(N-1).
```

[1]People often express disbelief at lines-of-code counts such as the above. It appears that the complexity of telephone switching systems such as the ATM switch stems from the complexity of the underlying telecom protocols.

More complex data structures are built with lists and tuples. The following function takes a three element tuple and returns the sum of the second and third element.

```
f({_, X, Y}) ->
      X + Y.
```

The primitives for concurrency are straight-forward, Suppose we have a module `m` function with a function `run`.

```
-module(m).

run() -> ....
...
```

The expression `Pid = spawn(m,run,[])` will create a process evaluating `run` and bind the variable `Pid` to its process identifier.

Now, an expression `Pid ! { deposit, 42}` sends a message to the process, and if the function `run` was written

```
run() ->
    receive
      {deposit, X} ->
           .... % add X monetary units to current account.
```

it would handle the message appropriately. Note that any value can be sent as a message.

All Erlang processes under a specific node share the same OS-process. An Erlang process is a data structure, containing a stack, a heap and a program counter. (The stack and the heap are small at creation, and allowed to grow when necessary, so the minimum size of a process is a few hundred bytes.)

A process identifier may refer to a process on other nodes (which may or may not reside on the same machine). To keep downtimes low, Erlang also has mechanisms for restarting failing processes and upgrading code at run-time.

Regarding the cost of process creation, let me relate some figures by Armstrong [1]. He compared implementations of Java and Erlang on a single computer using a simple test program, and measured the cost of message passing and process creation. He found that the Java system he tested never allowed more than two thousand threads. On the same machine, thirty thousand Erlang processes was completely unproblematic. The cost of sending a message (in Java, a synchonized method call) was 50 microseconds in the Java system when the number of threads was low, but rose sharply when the number of threads increased beyond 1000 threads. In Erlang, the time to send a message was never more than 0.8 microseconds, even for 30000 processes. The difference in the cost of process creation was even greater (two orders of magnitude).

Which are the crucial features of Erlang? In my opinion, if you want a concurrent programming language where multi-process programs are easy to program efficiently and reliably, the language should have the following:

- Processes should be very light-weight. Running tens of thousands of processes should be completely unproblematic.
- Don't use OS processes (at least, make sure that the behavior of the processes don't depend on the underlying OS).
- Do not allow shared state between processes.
- Base all communication and sychronization on message passing.

3 Experiences with Java threads

If one needs to write a complex concurrent application in Java, what is the best way to deal with the problems with Java's thread model? In this section, I will give an overview of a student project [4] where the student ended up implementing several of the key features of Erlang in Java (even though he wasn't aware of Erlang).

The project was carried out at a company, Ongame, which develops software for on-line casino games such as poker or blackjack. The game models a virtual casino, in which several players may sit at one casino table and interact with the game and each other. Each client is an applet or a windows application running on the player's machine, and communication with the clients is, of course, over the Internet.

It is desirable that a server should be able to handle hundreds of clients, but a good gaming experience requires response times in the order of a few tenths of a seconds, including network delays. It should should be easy to extended the system with other types of games.

The basic design consists of three classes: Server, which keeps the state of the game server, Game, which models a room where players enter and communicate, and Player, which maintains a connection with a client.

There is a constant need to maintain "awareness" of other players. The system needs to perform many operations take time but not CPU, for example database lookups and client communication. In other words, it is quite representative for the type of problems we want to solve using threads.

3.1 A model for communication

Early in the project, it was recognized that the conventional way of doing communication between threads in Java (with shared objects and synchronization) proved difficult due to deadlocks and often gave rise to bugs that were hard to find.

Instead it was decided to use a communication model based on events. Each core object (Server, Game, Player) maintains a queue of events. An event may contain arbitrary data structures, and all communication between core objects are done via the event system.

It would have been interesting to see a side-by-side comparison between systems written using shared-data communication and message passing. Since it already had been determined (by developers at the company) that writing a game server directly in Java's concurrency model would require much more time, the effort was not undertaken.

3.2 Thread pooling

The other problem with Java's concurrency model is that running many threads is expensive! A single player in the game server does not require much processing power, but expanding the number of players beyond a few hundred proved difficult. Even with only three hundred players the performance degradation was noticeable. The technique applied to solve this problem, *thread pooling*, uses regular Java objects (in the game server, they corresponded to objects of the classes Server, Game and Player) to represent a task. Initially, a limited pool of threads is created. Tasks are placed in a queue at creation, and when a thread has finished its current task it will choose a new one from the queue. With thread pooling the number of clients could be expanded beyond 400.

3.3 Comparison with Erlang

It is interesting to compare the architecture that resulted from these deliberations with Erlang's concurrency model:

- It was found that communication and synchronization by message-passing simplifies development by eliminating problems with deadlock and race conditions.
- By using ordinary Java objects to emulate light-weight processes performance was improved, especially when loads were high.

In other words, to obtain a robust, flexible and scalable game server, it was necessary to implement a few key features of Erlang. This reminds of Greenspun's Tenth Rule of Programming:

> Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.

(To the best of my knowledge, the student's implementation was not bug-ridden.)

4 Conclusions

From the experiences with Erlang and Java it seems that if one wants a programming language where writing efficient multi-threaded applications is no harder than writing single-threaded applications, the language should have the following properties:

- There should be *really* light-weight processes.
- The behavior of multi-threaded applications should be independent of operating system.
- All communication and synchronization should be through message-passing.
- It should be possible to send any value as a message.

The last item is obvious if one considers a language such as Java, Erlang or Lisp, but early data flow languages satisfied the first three requirements but not the last.

Is Erlang the final answer? That is unlikely. Just as Java looks great if we see it as an improvement over C++ but not-so-great if we look further, I'm sure that once we get accustomed to the Erlang process model we'll see many things that could be improved. (Ten years from now we'll have a Post-Erlang workshop?)

Is it possible to implement Erlang-like concurrency as a library in Java? Certainly, but the result is likely to be harder to use and less efficient than the same features integrated in a programming language.

References

[1] Joe Armstrong. Concurrency oriented programming in Erlang. Invited talk at the Lightweight Languages Workshop (LL2) held at MIT, 2002.

[2] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition.* Prentice-Hall, 1996.

[3] Klas Eriksson, M. Williams, and J. Armstrong. Program development using Erlang–programming rules and conventions. Available at `www.erlang.org`.

[4] Viktor Lidholt. Design and testing of a generic server for multiplayer gaming. Master's thesis, Uppsala University, 2002.