ERLANG'S ACTORS CONCURRENCY MODEL IN JAVA

Zbyněk Šlajchrt
Vysoká škola ekonomická v Praze
xslaz900@isis.vse.cz

Keywords

Concurrent programming, scalability, Java, Chaplin ACT, Erlang, Actors model

Introduction

Today one or more multi-core processors may equip even an ordinary home computer. The increasing demand for the newest 3D games, which are so popular among teenagers, or for the multimedia software pushes the prices of the powerful hardware continuously down. Not only a family budget may profit from the affordable prices of the multi-processor hardware, but also enterprise clients may buy high-performance hardware for a reasonable price and begin to require that their software utilize the underlying powerful hardware as efficiently as possible. At this point we are entering the world of enterprise applications where the Java language is well established. So it is legitimate to ask question how Java fits to developing concurrent applications and which tools it provides.

The Java's standard way to concurrency

The Java language uses a concurrency model similar to those in the most popular languages. It is based on locks that control access to shared mutable state. Multiple parallel executions of code are called threads. Java provides a small set of built-in language constructs, which a programmer uses to control threads' access to the shared resources. Although simple and lightweight in its nature, this model embarks a heavy load on the programmer's shoulder because of the constitution of the programmer's mind, which is largely single-threaded. This model makes developing concurrent programming rather challenging than easy and requires high imagination and experience. Inexperienced programmers are often unaware of the pitfalls lurking in the illusory simplicity of the model and their programs annoy users by unpredictable deadlocks and weird behaviour caused by race conditions.

The above-mentioned problems led the Java language designers to decision to incorporate a third-party library to Java 5. Doug Lea, who had developed this library, designed it to make the programming of concurrent applications more high-level and safer [1]. This library contains a number of abstractions like mutex, semaphore, queue, executor, future, channel and others that have their counterparts in the known and proven concepts used in the domain of concurrent applications.

The new concurrent library caused a big leap in the development of concurrent applications in Java and a number of enterprise application servers use it for implementing concurrency.

However, one thing is to provide a high-level library allowing us to write safer concurrent programs, and other thing is to write programs that scale. In other words, we need that the program performs better when it runs on a computer equipped with more processors. Adding processors should lead to better performance of our program. The key characteristic of well-scalable programs is the ability of parallelization. The program must be written in a way that its parts can be run in parallel. The percentage of the code, which cannot be parallelized,

negatively correlates with the maximum speedup, which can be achieved by adding more processors. Amdahl's law expresses this finding.

Amdahl's law

This law quantifies an expected improvement of an overall system in the case that only a certain part of it can be improved. In concurrent programming it is used to predict the maximum speedup when adding more processor power.

If P is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and (1 − P) is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using N processors is [2]

$$\frac{1}{(1-P)+\frac{P}{N}}$$

One can easy conclude from this equation that when the number of processors (N) is increasing to infinity the limit of the formula becomes

$$\frac{1}{(1-P)}$$

For example, if a program runs 10 hours in a single-threaded environment and there is 1-hour subroutine, which cannot be parallelized, then the maximum speedup that can be achieved by adding more processors is 10x (P=0.9).

The conclusion is that our programs do not scale above some limit. The bigger part of the application cannot be parallelized the lower the limit is. The question is whether there is an alternative, which would naturally lead to designing programs that could be more capable of parallelization. The answer is yes and the alternative is the Actors model as implemented in the Erlang language.

Erlang's actor concurrency model

The Actors model became famous thanks to its implementation in the Erlang language. This language was invented in 1986 at Ericsson and the motivation for its birth was an urgent need for a language in which engineers would design software for the Ericsson's high-speed telephone switches. These programs had to be extremely robust and fast, as the switches must run nonstop. Erlang fulfilled all these requirements mainly because of its concurrency model, which has become known under the name Actors model. So, how does this model work?
The main idea behind is that whole system consists of so-called actors, which interact between themselves by means of posting immutable messages. An actor can be viewed as a small service performing asynchronous operations in response to requests posted by other actors. An important feature, which significantly influences robustness and stability, is that the actors never share the state and the messages are immutable. Thus race condition never happens.
Another key design feature is that each actor owns a mailbox to which other actors post messages. Actors cannot be invoked otherwise then through the mailbox. Once a message arrives to the mailbox the sleeping actor wakes up and performs an action according to the pattern matching the incoming message.
Because of such nice properties it makes sense to think about incorporating the Actor model to other languages. Some languages did so and for example Scala has a built-in support for

this concurrency model. Java does not provide itself any construct allowing such programming, however, only few attempts have been made so far trying to provide a framework for the Actors model [3, 4]. The Chaplin ACT software offers such a framework.

Actors in Chaplin ACT

Chaplin ACT is a Java class transformer, which modifies classes in such a way that the classes' instances can form complex composites with minimal coupling. Using this software a programmer is given a tool allowing him to use several modern language concepts like mixins and/or aspect oriented programming. From the programmer's point of view it is seen as en extension to the Java language. Among other features it offers an Actors model framework, which naturally fits to the philosophy of Chaplin ACT [5].

In Chaplin ACT a number of components can form a composite. The interaction between components within the composite is based on messages, which can be either synchronous or asynchronous. Once either message arrives to a target component the behaviour differs according to whether the component is or is not an actor. If the receiving component **is not** an actor then a corresponding method is directly invoked on the component. However, if a component **is** an actor then instead of method invocation the message is stored into the actor's mailbox.

A component becomes an actor by annotating its class with `@Actor` annotation:

```
@Actor
public class ActorA {
```

This annotation instructs the Chaplin ACT to enhance the `ActorA` class by the mailbox. Furthermore, it causes, that an incoming message is stored to the mailbox instead of direct component invocation.

After an actor is spawn it pauses until a message appears in the mailbox. The actor invokes `$receive()` operator and passes a message handler:

```
$receive(new Object() {
    void messageX(int value) {
        System.out.print("Value " + value + " received");
    }
});
```

The message handler can be an instance of `java.lang.Object` and may define an arbitrary number of methods. These methods represent handlers of individual messages. The name of a method corresponds to the name of a message and the method's arguments correspond to the message attributes. Invoking `$receive` publishes information about messages processed by the handler to the composite so that it can route messages here. Once a message appears in the mailbox and fits to one of the published methods the matching method is invoked and the `$receive` operator returns. After that all previously published methods are suppressed.

If a component wishes to post a message to another component it declares the message as an abstract method annotated with `@FromContext`. The name and arguments convention follows the same rule as in the case of message handler method:

```
@FromContext
abstract void messageX(int value);
```

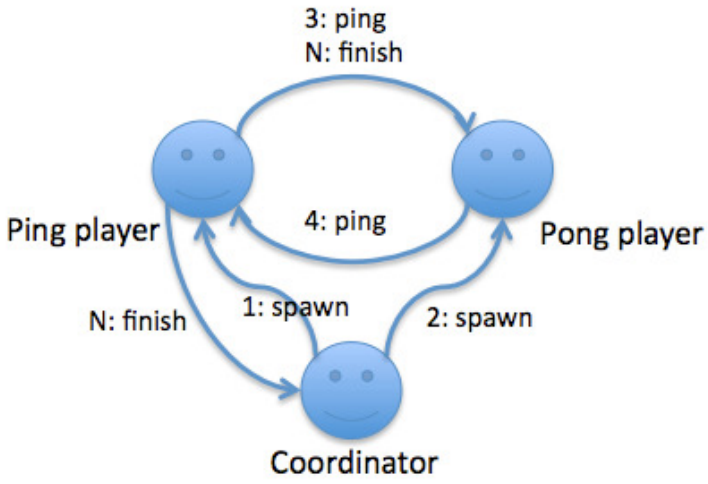
Chaplin ACT silently implements this abstract method and when the method is invoked it sends a message to the routing infrastructure of the composite:

```
public void spawn() {
    messageX(1);
}
```

As soon as the composite's router receives a message it must select the target component or components. In this case that no routing metadata are provided the target component is any component that publishes the matching method. Let's examine the behaviour on the following example.

Ping-pong example

This is a classic example used for explanatory purposes when dealing with the Actors model. In this scenario there are two actors: the ping player who sends the ping message and the pong player who sends the pong message after it receives the ping. There is also a coordinating actor who initiates and finishes the game. The communication between the actors is depicted on the following picture:



After N ball switches the Ping player broadcasts the finish message, which is received by both Pong player and Coordinator.
The Java code for Ping player is as follows:

```
@Actor
@Role
public abstract static class Ping {

    @FromContext(role = Pong.class)
    abstract void ping();
    @FromContext()
    abstract void finish();

    public void spawn(int n) {
        if (n == 0) {
            System.out.println("ping finished");
            finish();
            return;
        }
        // Post the 'ping' message
        ping();
        // Wait until the mailbox receives a reply
        $receive(new Object() {
            public void pong() {
                System.out.println("received pong");
            }
        });
        spawn(n - 1);
    }
}
```

I have not explained yet the function of the `@Role` annotation. By annotating a component's class by this annotation the component is automatically assigned the role when inserted into a composite. The role in this case matches the `Ping` class's name. It is a routing metadata that can be used for targeting messages within the composites. The `Ping` class declares two outbound messages `ping` and `finish`, which can be sent from this class to other components in the same composite. The `@FromContext` annotation at the `ping` abstract method has the property `role` set to the `Pong` class. It is a hint for the message router to deliver the message to the component playing the role Pong in the composite. The method spawn starts the game by sending the ping message and entering into the receive mode where it waits until the pong message returns. After that the spawn method calls itself to continue the game.
The `Pong` class represents the second player:

```
@Actor
@Role
public abstract static class Pong {

    @FromContext(role = Ping.class)
    abstract void pong();

    public void spawn() {
        for (Object msg : $receive(new Receiver() {

            public void ping() {
                System.out.println("received ping");
                pong();
            }

            public void finish() {
                System.out.println("received finish");
                super.finish();
            }
        }));
        System.out.println("pong finished");
    }
}
```

This class is also annotated with `@Actor` and `@Role` annotations that I explained above. The Pong component declares the `pong` message, which is sent back to the Ping component as a response to the `ping` message. The spawn method activates the actor and publishes the `ping` and `finish` message handlers. The mutation of the `$receive` operator used here returns an iterator that provides a sequence of received messages. This `$receive` operator takes an instance of the `Receiver` class which provides the instance with a handful of methods controlling the iteration. The loop continues until the `finish` message is received. The handler of the `finish` message calls the `finish` iteration control method on the super class that instructs the iterator to stop the iteration. The `ping` message handler simply prints out the message and replies by sending the `pong` message to the Ping player.
The Coordinator initiates both players and waits until the Ping component emits the finish message.

```
@Actor
@Role
public static abstract class Coordinator {

    @FromContext(async = true, role = Ping.class)
    abstract void spawn(int n);

    @FromContext(async = true, role = Pong.class)
    abstract void spawn();

    public void spawnAll(int n) {
        // spawns Pong player
        spawn(n);
        // spawns Ping player
        spawn();

        $receive(new Object() {
            public void finish() {
                System.out.println("game over");
            }
        });
    }
}
```

The Coordinator declares two asynchronous `spawn` messages. The first one is sent to the Ping component and the second one to the Pong component. As the both messages are sent asynchronously the `spawnAll` method is not blocked when the `spawn` methods are invoked. After spawning the players it enters into the receive mode where it waits until the `finish` message is received.
The following code shows how the composite is assembled:

```
public static void main(String[] args) {
    Ping ping = $();
    Pong pong = $();
    Coordinator coord = $();
    $$(ping, pong, coord);

    coord.spawnAll(5);
}
```

The `$` operator is a statically imported method that is used in Chaplin ACT for instantiation of components. All components' classes in this example are abstract and they cannot be instantiated directly by means of the `new` operator. The `$$` operator fuses all components into a composite. From this moment all components live together in the context of the composite and they may interact by sending messages to each other. The game starts by calling the `spawnAll` method. The number of switches is 5.

CONCLUSION

The goal of this article was to present an alternative approach for developing scalable concurrent applications in Java. The Actors model became famous thanks to its implementation in the Erlang language and its successful application in the software for the high performance, robust and non-stop running systems. Java itself does not provide any built-in support for the Actors model; however, with the help of the Chaplin ACT it is very straightforward to incorporate the Actors model in a program written in Java.

REFERENCES

[1] Lea, Doug (1999). *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley. ISBN 0-201-31009-0.

[2] http://en.wikipedia.org/wiki/Amdahl%27s_law

[3] http://www.javaworld.com/javaworld/jw-02-2009/jw-02-actor-concurrency1.html

[4] http://www.javaworld.com/javaworld/jw-03-2009/jw-03-actor-concurrency2.html

[5] http://www.iquality.org/chaplin