Multi-active Objects

Ludovic Henrio Fabrice Huet
INRIA – CNRS – I3S – Univ Nice Sophia Antipolis
{ludovic.henrio,fabrice.huet}@inria.fr

Zsolt István Gheorghe Sebestyén
Technical University of Cluj-Napoca
zsolt.istvan@gmx.net, gheorghe.sebestyen@cs.utcluj.ro

I. Background and Objectives

Even though there are several frameworks that are widely used to program distributed applications (OpenMP, RPC, Java RMI), these deal only with communication transparency, and do not hide all details of the distributed nature of the application. As a result, programmers have to deal with several low level aspects such as remote object registry setup and communication channel management. This gave rise to a new generation of frameworks that make all aspects of object distribution transparent to the user. Such high-level programming models with strong properties turned out to be crucial for programming safe large-scale applications. Some of these frameworks are based on the actor paradigm [1]. Actors are entities defined by their behavior and communicating strictly by message passing, therefore they are decoupled completely from each other [2], [3]. Active objects are inspired by actors, but they are closer to object oriented paradigms [4], [5], [6], [7]. The main goal of active objects is to achieve global concurrency with the help of asynchronous communication and internal scheduling mechanisms for request handling.

All languages based on principles inspired from actors suffer from the same limitation concerning local parallelism. Actors are very efficient and very easy to program when it comes to distribution: actors abstract away the notion of distribution, remote communications occur in the form of messages, which are enqueued and treated by the receiver. This way different computing entities are strongly decoupled and synchronization only concerns the reception of messages. As a result, the programmer does not have to deal with data race-conditions. Unfortunately, when it comes to local parallelism, this programming model is far from being efficient because it entails a lot of data copy between actors while direct memory access would be faster. Some of the actor-like paradigms can be tweaked such that local activities are multi-threaded, but in that case, the programmer loses the benefits of a well-designed programming model, and has to face complex synchronizations and data race-conditions.

Taking into consideration the widespread popularity of multi-core processors and the trend of increasing the number of cores, any framework that does not fully utilize the multithreading capacities of multi-core architectures will seem deprecated. On the other hand, it is rather hard to deal both with the application logic and with the clearly orthogonal task of concurrent synchronization at the same time. Even if concurrent code is supposed to improve the performance of an application, if it is unwisely written it can introduce race-conditions, which make development and testing difficult.

Our approach extends the active object paradigm to support multi-threading of active objects while decoupling the logic flow from synchronization. We provide the user with a solution in which parallelism is transparent, just as the distributed nature of the application. One crucial requirement we impose ourselves is to keep the programming language easy to use for programmers who are not necessarily expert in concurrent programming. We thus decided to allow the programmer to annotate methods corresponding to entry points of the active object with information regarding parallelism. We use this information to run several methods in parallel. Annotations should be simple enough so that the programmer gives only high-level information, that we can use to "schedule" the parallel execution of several threads. We present a set of annotations to provide multi-threading for active objects, while keeping backward compatibility with standard active objects together with a way to use those annotations to synchronize multi-threading inside active objects.

II. Proposal

In order to create a multi-active object model, we decided to reason in terms of request compatibility. Compatibility of two requests means that running them in parallel either a) does not result in any data concurrency, or b) is expected by the programmer (i.e. he/she manages his/herself the data concurrency). In the second case it is supposed that the data in question will be protected in the code (with locks, mutual exclusion blocks, etc.). Since static analysis is out of the scope of this paper, we trust the programmer to define the compatibility rules among methods correctly. These rules can be thought of as contracts between the programmer and the runtime environment, in which the programmer allows the framework to run several methods in parallel. Whenever the programmer specifies that parallelism is harmful, he/she can rely on the runtime environment for assuring the safety of execution. The runtime will provide as much parallelism as possible, unless the programmer has stated otherwise.

A. Annotations

Method compatibility could be expressed in many different ways. We chose an approach where the programmer explicitly states the compatibility between methods, and the mutual exclusion is then deduced. Although being somehow similar to the use of the *synchronized* keyword in Java, it has more semantic flexibility. While the *synchronized* keyword is used to restrict parallelism between several methods, our approach allows for both a restrictive and permissive reasoning.

Obviously, pairwise compatibility relations for a high number of methods could easily become too complex to declare

```
@DefineGroups({
  @Group(name="GroupF", selfCompatible=true),
  @Group(name="GroupB", selfCompatible=false)
})
@DefineRules({
  @Compatible( { "GroupF", "GroupB" } )
})
```

Fig. 1. Annotations for Groups and rules

```
@MemberOf("GroupF")
public int foo_1() {...}

@MemberOf("GroupF")
public int foo_2() {...}

@MemberOf("GroupB")
public int bar() {...}
```

Fig. 2. Annotations for membership

and to maintain. Therefore, we introduce the notion of groups to express compatibility relations on sets of methods rather than on individual methods. A group gathers methods that perform a similar task, thus manipulate the same data. Unless specified otherwise, methods inside a group are mutually incompatible. These groups not only have the goal of reducing the amount of added meta-data, but also help in the logical structuring of an application, since methods working on the same data set can be most probably collected into the same group. To specify which groups can run concurrently, a set of compatibility rules is given by the programmer.

To create groups and to specify rules, the programmer will use source code annotations placed at the beginning of classes. A group is defined by its name, that acts as an identifier used in compatibility rules. If the methods contained in the group can be executed in parallel, its optional *selfCompatible* property can be set to true. Rules define sets of groups that are all compatible with each other. In the example shown in Figure 1 we create two groups and define their relationship as follows:

- Methods which are members of *GroupF* can run concurrently because they are self compatible.
- Methods in *GroupB* are mutually exclusive.
- Any method from *GroupF* can run concurrently with a method from *GroupB*.

The membership to a group is specified using an annotation written directly before a method (Figure 2). A method can only belong to one group because it is simpler to create a new group than to deal with compatibility issues when a method belongs to two groups.

One classical issue of usual active objects is that re-entrant requests systematically lead to deadlocks: if a request sent to an active object require the result of a request to the same active object, then a deadlock occurs. This is particularly unavoidable if the request must go through a second active object. Compatibility annotations allow two requests of the same objects to be run concurrently and thus remove some of the deadlocks due to re-entrance. In particular, if a group is self-compatible then its methods can be re-entrant: they can use the result of a call on the same method to terminate.

```
method runActivity() {
  while (true) {
    serve(requestQueue.removeFirst());
  }
}
```

Fig. 3. Pseudo-code of service loops (FIFO)

```
method runActivity() {
 while (true) {
   if (compatible(requestQueue.peekFirst(),
        activeRequests)) {
     parallelServe(requestQueue.removeFirst());
} } }
```

Fig. 4. Pseudo-code of service loops (Multi-active)

This was typically impossible with the classical active object paradigm.

An advantage of using these annotations is that the amount of work needed to define rules does not depend on the size of a class, but only on the number of groups. If the complexity of performing this operation would grow exponentially with the number of exposed methods, this approach would not be practical in real-world applications. A potential drawback of the annotation-based approach is that, since meta-data is contained inside the Java classes, once they are compiled it can not be changed, and it is not possible to modify the compatibility of methods at runtime. This limitation can be addressed using scheduling policies presented in II-B.

Safely executing a multi-threaded active object then consist in serving a request in parallel with the others *if no incompatible request is currently being served*, to avoid running two incompatible methods in parallel.

B. Multi-active scheduling

In the previous section we introduced the idea of compatibility annotations for active objects. We will proceed by presenting how this information can be used in the context of the ProActive framework to enable multi-threaded local execution.

In the ProActive framework each active object has to implement a method called `runActivity` that constitutes its life-thread. Most of the time this method is provided by the ProActive framework, and does not have to be implemented by the programmer. Since active objects can execute only a single request at a time, the default policy is to handle requests in a first-in-first-out manner.

Figure 3 shows the `runActivity` method provided by ProActive by default. The `serve` method takes a request as parameter and executes it in the context of the caller's thread. As a result, the service loop will return only when the request is served. This way (assuming that `removeFirst()` blocks if the queue is empty), there is no need for more logic inside the loop. To be able to serve several requests in parallel, a mechanism is needed to move their execution to secondary threads, started from the main loop. We introduce `parallelServe`, a non blocking method which will execute a request using a different thread. Figure 4 shows a new version of the `runActivity` method which executes requests concurrently if they are compatible. A list of currently executing requests is maintained (`activeRequests`) and the first request in the request queue is checked for compatibility, and served, if possible.

In the actual implementation this loop is hidden from the user inside the scheduling class. Note that the term "scheduling/scheduler" is used in this paper to mean: "definition of a multi-threading service policy" and is not related to the design of efficient schedulers for distributed computing. The scheduler is a policy based one and it provides two predefined policies (or strategies): multi-active and FIFO. The first aims at starting as many requests in parallel as possible, while maintaining the relative order of their arrival at serving time. The second strategy exists for compatibility purposes, and can be used to reproduce the classic single-active behavior. Besides these two predefined policies we also expose a complete API which the programmers can use to write customized policies, e.g. one which limits the maximum number of parallel threads inside an active object.

The Scheduling API can be split into two parts. One that deals with method compatibility and one that exposes the internal state of the scheduler. The first contains methods for verifying the compatibility of two (or more) requests or method names. The second gives access to the request queue and the set of already executing requests inside the active object. To simplify the policy-writing, a policy is defined as a function, which takes as input the compatibility information and the scheduler state, and produces as an answer the list of requests that can be started right away. The internal state of the scheduler is guaranteed not to change while executing a policy, so the code of the policy does not have to deal with synchronization.

III. Conclusion

We have presented an extension of active objects to support local parallelism and reentrant calls. Based on the use of annotations to indicate the possibility of methods to run in parallel, our approach reduces the need for explicit locking. It brings a high degree of parallelism, is extensible, and compatible with legacy active objects. On the programmer side, the complexity is minimal: only some simple compatibility annotations have to be added inside the code. We have also proposed an API which can be used to implement custom scheduling policies to improve the performance of an application.

We implemented our proposal as an extension of the ProActive library and conducted experiments that revealed promising: we could run up to 41 simultaneously served requests and obtained a reasonable speedup on a graph traversal application. The reader may refer to [8] for a longer version of this paper including experiments and exhaustive related works.

References

[1] G. Agha, "An overview of actor languages," in *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*. ACM, 1986, pp. 58–67.

[2] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2-3, pp. 202–220, 2009.

[3] J. Armstrong, "Erlang-A survey of the language and its industrial applications," in *In Proceedings of the symposium on industrial applications of Prolog (INAP96). 16–18*. Citeseer, 1996.

[4] K. Taura, S. Matsuoka, and A. Yonezawa, "ABCL/f: A future-based polymorphic typed concurrent object-oriented language-its design and implementation," in *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*. Citeseer, 1994.

[5] E. B. Johnsen, O. Owe, and I. C. Yu, "Creol: A type-safe object-oriented model for distributed concurrent systems," *Theoretical Computer Science*, vol. 365, no. 1–2, pp. 23–66, Nov. 2006.

[6] D. Caromel, W. Klauser, and J. Vayssiere, "Towards seamless computing and metacomputing in Java," *Concurrency: practice and experience*, vol. 10, no. 11-13, pp. 1043–1061, 1998.

[7] D. Caromel, L. Henrio, and B. Serpette, "Asynchronous and deterministic objects," in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 2004, pp. 123–134.

[8] L. Henrio, F. Huet, Z. István, and G. Sebestyé, "Adapting active objects to multicore architectures," in *International Symposium on Parallel and Distributed Computing (ISPDC 2011)*. IEEE Computer Society, 2011.