Multi-threaded Active Objects

Ludovic Henrio, Fabrice Huet, Zsolt István

Multi-threaded Active Objects

Ludovic Henrio[1], Fabrice Huet[1], Zsolt István[2]

[1] INRIA-I3S-CNRS, University of Nice Sophia Antipolis, France
[2] Department of Computer Science, ETH Zurich, Switzerland
ludovic.henrio@cnrs.fr, fabrice.huet@inria.fr, zistvan@ethz.ch

Abstract. Active objects offer a paradigm which simplifies writing distributed applications. Since each active object has a single thread of control, data races are prevented. However, this programming model has its limitations: it is deadlock-prone, and it is not efficient on multicore machines. To overcome these limitations, we present an extension of the active object model, called multi-active objects, that allows each activity to be multi-threaded. The new model is implemented as a Java library; it relies on method annotations to decide which requests can be run in parallel. It provides implicit parallelism, sparing the programmer from low-level concurrency mechanisms. We define the operational semantics of the multi-active objects and study the basic properties of this model. Finally, we show with two applications that our approach is easy to program and efficient.

Keywords: Concurrency and distribution, active-objects, multicore architectures.

1 Introduction

Writing distributed applications is a difficult task because the programmer has to face both concurrency and location-related issues. The active object [1–3] paradigm provides a solution by abstracting away the notions of concurrency and of object location. An object is said to be active if it has its own thread of control. As a consequence, every call to such an object will be some form of remote method invocation – a *request* – that is handled by the object's thread of control. Active objects are partly inspired by Actors [4, 5]: they share the same asynchronous treatment of messages, and ensure the absence of data race-conditions. They do differ however in how the internal state of the object is represented. Active objects are mono-threaded, which prevents data races without the use of synchronized blocks. Distributed computations rely on the absence of sharing between processes, allowing them to be placed on different machines.

In classical remote method invocation, the invoker is blocked waiting for the result of the remote call. Active objects, on the other hand, return futures [6] as placeholders for the result, allowing the invoker to continue its execution. Futures can be created and accessed either explicitly, like in Creol [7] and JCoBox [8], or implicitly as in ASP [3] (Asynchronous Sequential Processes) and AmbientTalk [9]. A key benefit of the implicit creation is that no distinction is made between synchronous (i.e., local) and asynchronous (i.e., remote) operations in the program. Hence, when the accessed object is remote, a future is immediately obtained. Similarly to their creation, the access to futures can happen either explicitly using operations like *claim* and *get*, or implicitly,

in which case operations that need the real value of an object (*blocking* operations) automatically trigger synchronisation with the future update operation. In most active object languages, future references can be transmitted between remote entities without requiring the future to be resolved.

There are different models related to active objects. The ASP calculus [3] is a distributed active object calculus with futures; the ProActive library is its reference implementation. ASP has strong properties and is easy to program, but the strict sequential service of requests creates deadlocks. A different approach is provided by active objects languages with cooperative multithreading (Creol [7, 10], Jcobox [8]). In this model a thread serving a request can stop in order to let the service of another request progress. While no data race condition is possible, interleaving of the different request services triggered by the different release points makes the behaviour more difficult to predict; in particular Creol does not feature the determinism properties of ASP. Explicit future access, explicit release points, and explicit asynchronous calls make JCobox and Creol richer, but also more difficult to program.

This paper provides a new extension of the active object model with a *different trade-off* between expressiveness and ease of programming compared to the existing approaches. Our approach replaces the strict mono-threading that avoids data-races in active-objects, by an expressive and simple mechanism of request annotation that controls parallelism. Our approach also achieves better performance than classical active-objects on multi-core machines. Our contribution can be summarised as follows:

- A new programming model called *multi-active objects* is proposed (Section 2). It extends active objects with *local multi-threading*; this both enhances efficiency on multicore machines, and *prevents* most of the *deadlocks* of active objects.
- We rely on declarative *annotations* for expressing potential concurrency between requests, allowing easy and high-level expression of concurrency. However, the expert programmer can still use lower level concurrency constructs, like locks.
- We define the *operational semantics* of multi-active objects in Section 3. This semantics allows us to prove properties of the programming model.
- We experimentally show that *multi-active objects make writing of parallel and distributed applications easier*, but also that the *execution of programs based on multi-active objects is efficient* (Section 4). The programming model has been implemented as an extensions to a Java middleware[3].

Additionally, Section 5 compares our contribution with the closest languages, and finally Section 6 concludes the paper. Details on the semantics, the implementation, and the experiments are available in a research report [11].

2 Multi-active Object Programming Model

Illustrating example We will illustrate our proposal with an example inspired by a simple peer-to-peer network based on the CAN [12] routing protocol. In a CAN, data are stored in peers as key-value pairs inside a local datastore. Keys are mapped through a bijective function to the coordinates of a N-dimensional space, called the key-space.

[3] available at: `www-sop.inria.fr/oasis/Ludovic.Henrio/java/PA_ma.zip`

The key-space is partitioned so that each key is owned by a single peer. A peer knows its immediate neighbours, and when an action concerns a key that does not belong to this particular peer, it routes the request towards the responsible peer according to the coordinates of the key. Our CAN example provides three operations: *join*, *add*, and *lookup*. When a new peer *joins* the network, it always joins an existing peer. This joined peer splits its partition of the key-space in two; it keeps one half and gives the other to the new peer. The *add* operation stores a key-value pair, and *lookup* retrieves it.

Active objects are a natural choice to implement this application by representing each peer with an active object. Locally, requests will be served one-by-one. This limits the performance of the application, and also possibly leads to deadlocks if re-entrant requests are issued. With multi-active objects, a peer will be able to handle several operations in parallel. We will illustrate multi-active objects by showing how to simply program a safe parallelisation of the CAN peers.

2.1 Assumptions and Design Choices

To allow active objects to serve several requests in parallel, we introduce multi-active objects that enable local parallelism inside active objects in a safe manner. For this, the programmer can annotate the code with information about concurrency by defining a compatibility relationship between *concerns*. This notion of annotation for concurrency share some common ideas with JAC[13]. For instance, in our CAN example we distinguish two non-overlapping concerns: one is network management (*join*) and another is routing (*add*, *lookup*). For two concerns that deal with completely disjoint resources it is possible to execute them in parallel, but for others that could conflict on resources (e.g. joining nodes and routing at the same time in the same peer) this must not happen. Some of the concerns enable parallel execution of their operations (looking up values in parallel in the system will not lead to conflicts), and others do not (a peer can split its zone only with one other peer at a time).

In the RMI style of programming, every remote invocation to an object will be run in parallel, as a result, data-races can happen on concurrently accessed data. A classic approach to solve this problem in Java is to protect concurrent executions by making all methods *synchronized*. This coarse-grain approach is inefficient when some of the methods could be run in parallel. Alternatively, the programmer can protect the data accesses by using low-level locking mechanisms, but this approach is too fine-grained and possibly error-prone.

By nature, active objects materialise a much safer model where no inner concurrency is possible. In this work, we look for a concurrency model that is more flexible than the single threaded active-objects, but more constrained and less error-prone than Java concurrency. We extend active-objects by assigning methods to *group*s (concerns). Then, methods belonging to compatible groups can be executed in parallel, and methods belonging to conflicting groups will be guaranteed not to be run concurrently. This way the application logic does not need to be mixed with low-level synchronisations. The idea is that *two groups should be made compatible if their methods do not access the same data, or if concurrent accesses are protected by the programmer,* and *the two methods can be executed in any order*. The programmer should declare as compatible both the non-conflicting groups, and the groups where the conflicting code has been

protected by means of locks or synchronised blocks. We chose annotations to express compatibility rules because we think that this notion is strongly dependent on the application logic, and should be attached to the source code.

We start from active objects *à la* ASP, featuring transparent creation, synchronisation, and transmission of futures. We think that the transparency featured by ASP and ProActive helps writing simple programs, and is not an issue when writing complex distributed applications. We also think that the non-uniform active objects of ASP, and JCoBox, reflect better the way efficient distributed applications are designed: some objects are co-allocated and only some of them are remotely accessible. In our model, only one object is active in a given activity but this work can easily be extended to multiple active-object per activity (i.e. cobox). An active object can be transformed into a multi-active object by applying the following design methodology:

- Without annotations, a multi-active object behaves identically to an active object, no race condition is possible, but no local parallelism is possible either.
- If some parallelism is desired, e.g. for efficiency reasons or because dead-locks appeared, each remotely invocable method can be assigned to a group. Then compatibility between the groups can be defined based on, for example, the variables accessed by each method. Methods belonging to compatible groups can be executed in parallel and out of the original order. This implies that two groups should only be declared compatible if the order of execution of methods of one group relatively to the other is not significant.
- If even more parallelism is required, the programmer has two non-exclusive options: either he protects the access to some of the variables by a locking mechanism which will allow him to declare more groups as compatible, or he realises that, depending on runtime conditions, such as invocation parameters or the object's state, some groups might become compatible and he defines a compatibility function allowing him to decide at runtime which request executions are compatible.

We assume that the programmer defines groups and their compatibility relations inside a class correctly. Dynamic checks or static analysis should be added to ensure, for example, that no race condition appear at runtime. Verifying that annotations written by the programmer are correct or even inferring them, e.g. [14], is out of scope here.

2.2 Defining Groups

The programmer can use an annotation, `Group`, to define a group and can specify whether the group is `selfCompatible`, i.e., two requests on methods of the group can run in parallel. The syntax for defining groups in the class header is shown on lines 1–5 of Figure 1.

Compatibilities between groups can be expressed as `Compatible` annotations. Each such annotation receives a set of groups that are pairwise compatible, as illustrated on lines 6–9 of Figure 1. A method's membership to a group is expressed by annotating the method's declaration with `MemberOf`. Each method belongs to only one group. In case no membership annotation is specified, the method belongs to an anonymous group that is neither compatible with other groups, nor self-compatible. This way, if no method of a class is annotated, the multi-active object behaves like an ordinary active object. The `MemberOf` annotation is shown on lines 11, 13, 15, and 17 of Figure 1.

```
@DefineGroups({
  @Group(name="join", selfCompatible=false)
  @Group(name="routing", selfCompatible=true)
  @Group(name="monitoring", selfCompatible=true)
})
@DefineRules({
  @Compatible({"join", "monitoring"})
  @Compatible({"routing", "monitoring"})
})
public class Peer {
@MemberOf("join")
  public JoinResponse join(Peer other) { ... }
@MemberOf("routing")
  public void add(Key k, Serializable value) { ... }
@MemberOf("routing")
  public Serializable lookup(Key k) { ... }
@MemberOf("monitoring")
  public void monitor() { ... }
}
```

Fig. 1: The CAN Peer annotated for parallelism

Figure 1 illustrates the annotations in the context of a CAN peer active object in which *add*s and *lookup*s can be performed in parallel – they belong to the same self-compatible group `routing`. Since there is no compatibility rule defined between them, methods of `join` and `routing` will not be served in parallel. To fully illustrate our annotations, we added *monitoring* as a third concern independent from the others.

2.3 Dynamic Compatibility

Sometimes the compatibility of requests can be more precisely decided at run-time. For example, two methods writing in the same array can be compatible if they don't access the same cells. For this reason, we first introduce an optional *group-parameter* which indicates the type of a a parameter which will be used to decide compatibility. This parameter must appear in all methods of the group and in case a method has several parameters of this type, the leftmost one is chosen. In Figure 2, we add `parameter="can.Key"` to the *routing* group to indicate that the parameter of type *Key* will be used. Overall, at runtime, compatibility between two requests can be decided as a function depending on three parameters: the group-parameter of the two requests, and the status of the active-object. We describe below how we integrated this idea into our framework and allowed the programmer to define compatibility functions inside his/her objects.

To actually decide the compatibility, we add a condition in the form of a *compatibility function* which takes as input the common parameters of the two compared groups and returns *true* if the methods are compatible. The general syntax for this rule is:

```
  @compatible{value={"group1","group1"}, condition="SomeCondition"}
```

The compatibility function can be defined as follows:

- when `SomeCondition` is in the form `someFunc`, the compatibility will be decided by executing `param1.someFunc(param2)` where `param1` is the parameter of one request and `param2` is the parameter of the other.

```
@DefineGroups({
@Group(name="routing",selfCompatible=true,parameter="can.Key",condition="!equals")
  @Group(name="join", selfCompatible=false) })
@DefineRules({@Compatible(value={"routing", "join"},condition="!this.isLocal") })
public class Peer {
  private boolean isLocal(Key k){
    synchronized (lock) { return myZone.containsKey(k); }       }
}
```

Fig. 2: The CAN Peer annotated for parallelism with dynamic compatibility

– when `SomeCondition` is in the form `[REF].someFunc`, the compatibility will depend on the results of `someFunc(param1, param2)` with the group parameters as arguments. `[REF]` can be either **`this`** if the method belongs to the multi-active object itself, or a class name if it is a static method.

Additionally the result of the comparator function can be negated using "!", e.g. `condition="!this.isLocal"`. Since the compatibility method can run concurrently with executing threads, the programmer should ensure mutual exclusion, if necessary. One can define dynamic compatibility even when only one of the two groups has a parameter, in that case the compatibility function should accept one less input parameter. It is even possible to dynamically decide compatibility when none of the two groups has a parameter (e.g. based on the state of the active object); in that case the compatibility function should be a static method or a method of the active object, with no parameter.

As an example, we show how to better parallelise the execution of joins and routing operations in our CAN. During a *join* operation, the peer which is already in the network splits its key-space and transfers some of the key-value pairs to the peer which is joining the network. During this operation, ownership is hard to define. Thus a *lookup* (or *add*) of a key belonging to one of the two peers cannot be answered during the transition period. Operations that target "external" keys, on the other hand, could be safely executed in parallel with a join. Figure 2 shows a modified version of the `Peer` class which supports dynamic compatibility checks. For the sake of clarity, we omit unmodified code. The modifications are as follows. 1) a group parameter, `can.Key` has been added to the `routing` group; 2) a compatibility rule has been defined for groups `routing` and `join` with condition `"!this.isLocal"`; 3) a method **`boolean`** `isLocal(Key k)` has been created, which checks whether a key falls in the zone of a peer At runtime, the method `isLocal(Key k)` will be executed when checking the compatibility of groups `routing` and `join`. This method is selected because it is the only one matching the name of the condition and the group parameter type. We also defined a condition for self-compatibility in the `routing` group: to guarantee that there is no overtaking between requests on the same key, we configure the group to be `selfCompatible` only when the key parameter of the two invocations is not equal, see Figure 2, line 2.

3 A Calculus of Multi-active Objects

This section describes MultiASP, the multi-active object calculus. We present its small step operational semantics and its properties. In MultiASP, there is no explicit notion of place of execution, but the calculus is particularly adapted to distribution because, first,

inter-activity communication behaves like remote method invocation, and second, each object belongs to a single activity. Overall, each active object can be considered as a unit of distribution. The operational semantics is parametrised by a function deciding whether a request should be served concurrently on a new thread or sequentially by the thread that triggered the service.

3.1 Syntax and Runtime Structures

While x, y range over variable names, we let l_i range over field names, and m_i over method names (m_0 is a reserved method name; it is called upon the activation of an object and encodes the service policy). :: denotes the concatenation of lists; we also use it to append an element to a list. $\varnothing$ denotes an empty list or an empty set. The syntax of MultiASP is identical to ASP [3] (except for the clone operator that is of no interest here). Active objects and futures can be created at runtime. New objects can be allocated in a local store (there is one local store for each active object). Thus, we let ι_i range over references to the local store (ι_0 is a reserved location for the active object), α, β, γ range over active object identifiers, and f_i range over future identifiers. Among the terms above, static terms are the ones that contain no references to futures, active objects, or store location. Note that every term is an object and [] is an empty object:

$a ::= x$	variable (y also ranges over variables),
$\mid [l_i = a_i; m_j = \varsigma(x_j, y_j)a'_j]^{i\in 1..n}_{j\in 1..m}$	object definition (x_j binds self; y_j is the parameter),
$\mid a.l_i$	field access,
$\mid a.l_i := a'$	field update,
$\mid a.m_j(a')$	method call,
$\mid Active(a)$	creates an active object from a,
$\mid Serve(M)$	serves a request among M, a set of method labels.$M = \{m_i\}^{i\in 1..p}$
$\mid \iota$	location (only at runtime)
$\mid f$	future reference (only at runtime)
$\mid \alpha$	active object reference (only at runtime)

A *reduced object* is either a future, an activity reference, or an object with all fields reduced to a location. A store maps locations to reduced objects; it stores the *local state* of the active object:

$$o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} \mid f \mid \alpha \qquad\qquad \sigma ::= \{\iota_i \mapsto o_i\}^{i\in 1..k}$$

To ensure absence of sharing, the operation *Copy&Merge*$(\sigma, \iota\,;\, \sigma', \iota')$ performs a *deep copy* of the object at location ι in σ into the location ι' of σ', and ensures that communicated values are "self-contained". This is achieved by copying the entry for ι at location ι' of σ' (if this location already contains an object it will be erased). All locations ι'' referenced (recursively) by the object $\sigma(\iota)$ are also copied in σ' at fresh locations (See [11, 3] for the formal definition).

F ranges over *future value association lists*; such a list stores computed results where ι_i is the location of the value associated with the future f_i: $F ::= \{f_i \mapsto \iota_i\}^{i\in 1..k}$. The list of *pending requests* is denoted by $R ::= [m_i; \iota_i; f_i]^{i\in 1..N}$, where each request consists of: the name of the *target method* m_i, the location of the *argument* passed to the request ι_i, the *future* identifier which will be associated to the result f_i. $(f \mapsto \iota) \in F$

means $(f \mapsto \iota)$ is one of the entries of the list F and similarly $[m; \iota; f] \in R$ means $[m; \iota; f]$ is one of the requests of the queue R.

There are two parallel composition operators: $\|$ expresses *local parallelism*, it separates threads residing in the same activity, and $|||$ expresses *distributed parallelism*, it separates different activities. A request being evaluated is a term together with the future to which it corresponds $(a_i \mapsto f_i)$. An activity has several parallel threads each consisting of a list of requests being treated: the leftmost request of each thread is in fact currently being treated, the others are in a waiting state. C is a *current request structure*: it is a parallel composition of threads where each thread is a list of requests. By nature, $\|$ is symmetric, and current requests are identified modulo reordering of threads:

$$C ::= \varnothing \mid [a_i \mapsto f_i]^{i \in 1..n} \| C$$

Finally, an activity is composed of a name α, a store σ, a list of pending requests R, a set of computed futures F, and a current request structure C. A configuration Q is made of activities. Configurations are identified modulo the reordering of activities.

$$Q ::= \varnothing \mid \alpha[F; C; R; \sigma] ||| Q$$

An *initial configuration* consists of a single activity treating a request that evaluates a static term a_0: $b_0 = \alpha_0[\varnothing; a_0 \mapsto f_\varnothing; \varnothing; \varnothing]$.

Contexts. Reduction contexts are terms with a hole indicating where the reduction should happen. For each context $\mathcal{R}$, the operation $(\mathcal{R}[c])$ replaces the hole by a given term c. Contrarily to substitution, filling a hole is not capture avoiding: the term filling the hole is substituted as it is.

Sequential reduction contexts indicate where reduction occurs in a current request:

$$\begin{aligned}\mathcal{R} ::= {} & \bullet \mid \mathcal{R}.l_i \mid \mathcal{R}.m_j(b) \mid \iota.m_j(\mathcal{R}) \mid \mathcal{R}.l_i := b \mid \iota.l_i := \mathcal{R} \\ & \mid [l_i = \iota_i, l_k = \mathcal{R}, l_{k'} = b_{k'}; m_j = \varsigma(x_j, y_j) a_j]_{j \in 1..m}^{i \in 1..k-1, k' \in k+1..n} \mid Active(\mathcal{R})\end{aligned}$$

A *parallel reduction context* extracts one thread of the current request structure (remember that current requests are identified modulo thread reordering):

$$\mathcal{R}_c ::= \quad [\mathcal{R} \mapsto f_1] :: [a_j \mapsto f_j]^{j \in 2..n} \quad \| C$$

3.2 Operational Semantics

Our semantics is built in two layers, a local reduction $\rightarrow_{\text{loc}}$ defined in [3] and in [11] that corresponds to a classical object calculus, and a parallel semantics that encodes distribution and communications. Activities communicate by remote method invocations and handle several local threads; each thread can evolve and modify the local store according to the local reduction rules. Thanks to the parallel reduction contexts $\mathcal{R}_c$, multiple threads are handled almost transparently in the semantics. The main novelty in MultiASP is the request service that can either serve the new request in the current thread or in a concurrent one; for this we rely on two functions: *SeqSchedule* and *ParSchedule*. Given a set of method names to be served (the parameter of the *Serve* primitive), the set of futures calculated by the current thread, and the set of futures calculated by the other threads of the activity, these functions decide whether it is possible to serve a request sequentially or in parallel. The last parameter is the request queue that will be

LOCAL

$$\frac{(a,\sigma) \rightarrow_{\text{loc}} (a',\sigma')}{\alpha\big[F;\mathcal{R}_c[a];R,\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F;\mathcal{R}_c[a'];R;\sigma'\big] \,|||\, Q}$$

ACTIVE

$$\frac{\gamma \text{ fresh activity name} \qquad f_\varnothing \text{ fresh future} \qquad \sigma_\gamma = \mathit{Copy\&Merge}(\sigma,\iota\,;\,\varnothing,\iota_0)}{\begin{array}{c}\alpha\big[F;\mathcal{R}_c[\mathit{Active}(\iota)];R;\sigma\big] \,|||\, Q \longrightarrow \\ \alpha\big[F;\mathcal{R}_c[\gamma];R;\sigma\big] \,|||\, \gamma\big[\varnothing;[\iota_0.m_0([]) \mapsto f_\varnothing];\varnothing;\sigma_\gamma\big] \,|||\, Q\end{array}}$$

REQUEST

$$\frac{\sigma_\alpha(\iota) = \beta \qquad \iota'' \notin \mathit{dom}(\sigma_\beta) \qquad f \text{ fresh future} \qquad \sigma'_\beta = \mathit{Copy\&Merge}(\sigma_\alpha,\iota'\,;\,\sigma_\beta,\iota'')}{\begin{array}{c}\alpha\big[F;\mathcal{R}_c[\iota.m_j(\iota')];R;\sigma_\alpha\big] \,|||\, \beta\big[F';C';R';\sigma_\beta\big] \,|||\, Q \longrightarrow \\ \alpha\big[F;\mathcal{R}_c[f];R;\sigma_\alpha\big] \,|||\, \beta\big[F';C';R'::[m_j;\iota'';f];\sigma'_\beta\big] \,|||\, Q\end{array}}$$

ENDSERVICE

$$\frac{\iota' \notin \mathit{dom}(\sigma) \qquad \sigma' = \mathit{Copy\&Merge}(\sigma,\iota\,;\,\sigma,\iota')}{\alpha\big[F;\iota \mapsto f::[a_i \mapsto f_i]^{i\in 1..n} \| C;R;\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F::f \mapsto \iota';[a_i \mapsto f_i]^{i\in 1..n} \| C;R;\sigma'\big] \,|||\, Q}$$

REPLY

$$\frac{\sigma_\alpha(\iota) = f \qquad \sigma'_\alpha = \mathit{Copy\&Merge}(\sigma_\beta,\iota_f\,;\,\sigma_\alpha,\iota) \qquad (f \mapsto \iota_f) \in F'}{\alpha\big[F;C;R;\sigma_\alpha\big] \,|||\, \beta\big[F';C';R';\sigma_\beta\big] \,|||\, Q \longrightarrow \alpha\big[F;C;R;\sigma'_\alpha\big] \,|||\, \beta\big[F';C';R';\sigma_\beta\big] \,|||\, Q}$$

SERVE

$$\frac{\begin{array}{c}C = [\mathcal{R}[\mathit{Serve}(M)] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \| C' \\ \mathit{SeqSchedule}(M,\{f_i\}^{i\in 0..n},\mathit{Futures}(C'),R) = ([m,f,\iota],R')\end{array}}{\alpha\big[F;C;R;\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F;[\iota_0.m(\iota) \mapsto f]::[\mathcal{R}[[]] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \| C';R';\sigma\big] \,|||\, Q}$$

PARSERVE

$$\frac{\begin{array}{c}C = [\mathcal{R}[\mathit{Serve}(M)] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \| C' \\ \mathit{ParSchedule}(M,\{f_i\}^{i\in 0..n},\mathit{Futures}(C'),R) = ([m,f,\iota],R')\end{array}}{\alpha\big[F;C;R;\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F;[\iota_0.m(\iota) \mapsto f] \| \mathcal{R}[[]] \mapsto f_0]::[a_i \mapsto f_i]^{i\in 1..n} \| C';R';\sigma\big] \,|||\, Q}$$

Table 1: Parallel reduction (used or modified values are non-gray)

split into a request to be served and the remaining of the request queue. If no request can be served neither sequentially nor in parallel, both functions are undefined. We define $Futures(C)$ as the set of futures being computed by the current requests C. Then, parallel reduction $\longrightarrow$ is described in Table 1. We will denote by $\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$. Table 1 consists of seven rules:

LOCAL triggers a local reduction $\rightarrow_{\text{loc}}$ described in [3] and in [11]. **ACTIVE** creates a new activity: from an object and all its dependencies, this rule creates a new activity at a fresh location γ. The method m_0 is called at creation, the initial request is associated with $f_\varnothing$, a future that is never referenced and never used. **REQUEST** invokes a request on a remote active object: when an activity α performs an invocation on another activity β this creates a fresh future f, and enqueues a new request in β. The parameter is deep copied to the destination's store. **ENDSERVICE** finishes the service of a request: it adds

an entry corresponding to the newly calculated result in the future value association list. The result object is copied to prevent further mutations. **REPLY** sends a future value: if an activity α has a reference to a future f, and another activity β has a value associated to this future, the reference is replaced by the calculated value[3]. **SERVE** serves a new request sequentially: it relies on a call to *SeqSchedule* that returns a request $[m, f, \iota]$ and the remaining of the request queue R'. The request $[m, f, \iota]$ is served *by the current thread*. The *Serve* instruction is replaced by an empty object that will be stored, so that execution of the request can continue with the next instruction. *SeqSchedule* receives, the set of method names M, the set of futures of the current thread, the set of futures of the other threads, and the request queue. **PARSERVE** serves a new request in parallel: it is similar to the preceding rule except that it relies on a call to *ParSchedule*, and that a new thread is created that will handle the new request to be served $[m, f, \iota]$. The particular case where the source and destination are the same require an adaptation of the rules REQUEST and REPLY [11].

We can show that $\longrightarrow$ does not create references to futures or activities that do not exist, and thus the parallel reduction is well-formed. Quite often an active-object will serve all the requests in a FIFO order: m_0 consists of a loop: *while* (*true*) *Serve*(M_A) where M_A is the set of all method names[4] and the other methods never perform a *Serve*. m_0 is compatible with all the other methods and thus all services can be done in parallel, but the service of a request might have to wait until another request finishes. We call this particular case *FIFO request service*.

3.3 Scheduling Requests

Several strategies could be designed for scheduling parallel or sequential services. Future identifiers can be used to identify requests uniquely; also it is easy to associate some meta-information with them (e.g. the name or the parameters of the invoked method). Consequently, we rely on a compatibility relation between future identifiers: *compatible*(f, f') is true if requests corresponding to f and f' are compatible. We suppose this relation is symmetric.

Table 2 shows a suggested definition of functions *SeqSchedule* and *ParSchedule* which *maximises parallelism while ensuring that no two incompatible methods can be run in parallel*. The following of this section explains in what sense this definition is correct and optimal. The principle of the compatibility relation is that two requests served by two different threads should be compatible:

Property 1 (Compatibility). If two requests are served by two different threads then they are compatible: suppose $Q_0 \longrightarrow^* \alpha\big[F; C; R; \sigma\big] \,|||\, Q$ then:

$$C = [a_i \mapsto f_i]^{i \in 1..n} \| [a'_j \mapsto f'_j]^{j \in 1..m} \| C' \Rightarrow \forall i \in 1..n.\, \forall j \in 1..m.\, \mathit{compatible}(f_i, f'_j)$$

We consider that parallelism is "maximised" if a new request is served whenever possible, and those services are performed by as many threads as possible. Thus, to maximise parallelism, a request should be served by the thread that performs the *Serve* operation only if there is an incompatible request served in that thread. The "maximal parallelism" property can then be formalised as an invariant:

[4] *while* and *true* can be defined in pure ASP [3]

$$\frac{\begin{array}{c}\forall f \in F'.\, \mathit{compatible}(f_j, f) \qquad \exists f \in F.\, \neg\mathit{compatible}(f_j, f) \\ m_j \in M \qquad \forall k < j.\, m_k \in M \Rightarrow \big(\mathit{compatible}(f_j, f_k) \ \wedge\ \exists f \in F'.\, \neg\mathit{compatible}(f_k, f)\big)\end{array}}{\mathit{SeqSchedule}(M, F, F', [m_i, f_i, \iota_i]^{i \in 1..N}) = ([m_j, f_j, \iota_j], [m_i, f_i, \iota_i]^{i \in 1..j-1} :: [m_i, f_i, \iota_i]^{i \in j+1..N})}$$

$$\frac{\begin{array}{c}\forall f \in F'.\, \mathit{compatible}(f_j, f) \qquad \forall f \in F.\, \mathit{compatible}(f_j, f) \\ m_j \in M \qquad \forall k < j.\, m_k \in M \Rightarrow \big(\mathit{compatible}(f_j, f_k) \ \wedge\ \exists f \in F'.\, \neg\mathit{compatible}(f_k, f)\big)\end{array}}{\mathit{ParSchedule}(M, F, F', [m_i, f_i, \iota_i]^{i \in 1..N}) = ([m_j, f_j, \iota_j], [m_i, f_i, \iota_i]^{i \in 1..j-1} :: [m_i, f_i, \iota_i]^{i \in j+1..N})}$$

Table 2: A possible definition of *SeqSchedule* and *ParSchedule*

Property 2 (Maximum parallelism). Except the leftmost request of a thread, each request is incompatible with one of the other requests served by the same thread (and that precede it). More formally, if $Q_0 \longrightarrow^* \alpha\big[F; C; R; \sigma\big] \,|||\, Q$ then:

$$\big(C = [a_i \mapsto f_i]^{i \in 1..n} \| C' \ \wedge\ k > 1\big) \Rightarrow \exists i' < k.\, \neg\mathit{compatible}(f_k, f_{i'})$$

The properties above justify the two first premises of the rules in Table 2. The last premise decides which request to serve. For this, we filter the request queue by the set M of method labels. Then we serve the first request that is compatible with all requests served by the other threads. These definitions ensure that any two requests served inside two different threads are always compatible, and also that requests are served in the order of the request queue filtered by the set of methods M, except that *a request can always overtake requests with which it is compatible*. We say that parallelism is maximised because we eagerly serve new requests on as many parallel threads as possible.

We conclude this section by explaining why it is reasonable to let a request overtake compatible ones. Indeed, whenever two compatible requests are served in parallel compatibility implies that the operations of these requests can be freely interleaved. Overtaking is just a special case of interleaving, namely when all operations of one request are executed before the operations of the other request. Thus, even if requests were served in the exact incoming order, a request may actually overtake a compatible one. More generally, if all requests are necessarily served at some point, then allowing a request to be overtaken by compatible ones is a safe decision.

4 Evaluation

In this section we show that our proposal provides an effective compromise between programming simplicity and execution efficiency of parallel and distributed applications: multi-active objects achieve the same performance as classical concurrent approaches while simplifying the programming of distributed applications. A detailed analysis of the results and the description of the multi-active API is provided in [11]

Our proposal is implemented on top of ProActive, the reference implementation of ASP. No preprocessing or modified Java compiler are required, all decisions are made at runtime by reifying constructors and method invocation. The flexibility and portability

we obtain using these techniques induce a slight overhead, but experience shows it has no significant impact at the application level. The implementation of multi-active objects comes with an API for customising the request service policy, the possibility to limit the number of threads inside a multi-active object, and an inheritance mechanism for compatibility annotations [11].

NAS Parallel Benchmarks We first compare multi-active objects with Java threads based on a well-known parallel benchmark suite: the NAS Parallel Benchmarks. To achieve a multi-active implementation, we modify the Java-based version of the benchmark [15] and create a multi-active object version from each kernel. A single multi-active object replaces all worker threads, and futures and wait-by-necessity replace *wait()* and *notify()*. This makes the code easier to understand and to maintain. Code related to parallelism in the multi-active version is much shorter than the original (between 35% and 70%), depending on the benchmark. Also, our annotations make synchronisation much more natural, and on a higher level of abstraction than the original program. The performance of the modified application is very close to the original.

CAN Experimental Results We implemented the CAN illustrating example to show the efficiency brought by multi-active objects compared to active-objects in a distributed multicore environment. The purpose of these experiments is to evaluate the benefits of multi-active objects, not the performance of our CAN implementation. Therefore, our experiments were designed to provide an interesting workload for the active objects themselves, not necessarily for the CAN network. We created and populated a CAN using `join` and `add` operations, then we measured the benefits of `lookup` request parallelisation in the following situations:

- *All from two*: In this scenario, two corners send `lookup` requests to all other nodes, and then wait until all the results are returned. This experiment gives an insight about the overall throughput of the overlay.
- *Centre from all*: In this test case, all the peers lookup concurrently a key located in a peer at the centre of the CAN. This experiment highlights the scalability of a peer under heavy load.

We repeated each scenario 50 times and measured the overall execution time (the difference between successive runs was found to be negligible). The creation and population of the network was not measured as part of the execution time. We used up to 128 machines located at the Sophia-Antipolis site of the Grid5000[5] platform. All hosts are interconnected through Gigabit Ethernet, and are equipped with quad-core CPUs (Intel Xeon E5520, AMD Opteron 275 and AMD Opteron 2218), running Java 7 Hotspot 64-Bit Server VMs. Each requested value was 24KB in size.

Figure 3 shows the execution times and speedup (when turning active objects into multi-active ones) for several sizes for the two scenarios. Both scenarios achieve significant speedup thanks to the communication and request handling performed in parallel, however, the gain in the first scenario is smaller because the lookups are issued from the two corners in sequence; the sequential sending of the initial lookups limits the number of lookups present at the same time in the network. In the second case, the active object version has a bottleneck because the centre peer can only reply to one request at a time whereas those requests can be highly parallelised with our model. As shown before,

[5] `http://www.grid5000.fr/`

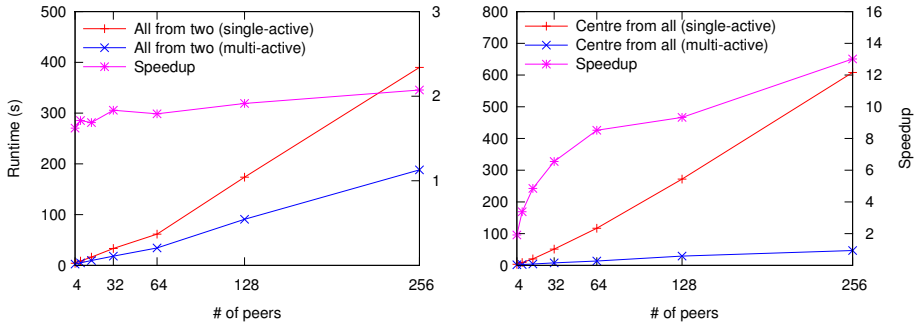

Fig. 3: CAN experimental results

these speedups are simply achieved by adding a few annotations to the class declaration without changing the rest of the code.

5 Comparison with Related Work

Parallel actor monitors [16] (PAM) provide multi-threading capacities to actors based on explicit scheduling functions. However, we believe that the compatibility annotations of multi-active objects provide a higher level of abstraction than PAM, and that this high-level of abstraction is what makes active-objects and actors easy to program.

The main difference between our approach and active-objects with cooperative multi-threading like JCoBox [8] and Creol [7] is twofold. On one hand, Creol-like languages are not really multi-threaded (only one thread is active at a time [7]), thus they do not necessarily address the issue of efficiency on multicore architectures; JCoBox proposes a shared immutable state that can be used efficiently on multicore architectures but as the distributed implementation is still a prototype, it is difficult to study how an application mixing local concurrency and distribution like our CAN example would behave. On the other hand, concerning synchronisation, in cooperative multi-threaded solutions between explicit release points (`awaits`) the programs are executed sequentially. Adequately placing those release points is the main challenge in programming in Creol or JCoBox: too many release points leads to a complex interleaving between sequential code portions, whereas not enough of them will probably lead to a deadlock.

Multi-active objects provide an alternative approach by allowing local concurrency in active objects: with annotations, the programmer can reason on high-level compatibility rules, and parallelism can be expressed in a simple manner. Compatible methods are run concurrently, with potential race-conditions but also local multi-threading. Overall, in ASP and MultiASP distribution and concurrency are much more transparent than in JCoBox and Creol; this difference in point of view explains most of the differences between the two models: transparent vs. explicit futures and compatibility annotations vs. explicit thread release. However, the principles of multi-active objects could be applied to an active object language with explicit futures and explicit release

points, but in this case thread activation (after an await statement) must take into account compatibility information.

Cunha and Sobral [17] use Java annotations to parallelise sequential objects in an OpenMP fashion. A method can be called asynchronously if it is flagged with the `Future` annotation, but the programmer must follow the flow of futures carefully and declare which methods can access them. There is also an `ActiveObject` annotation that creates a proxy and a scheduler, but its exact semantics is not well-defined in [17]. In our opinion, JAC's and our compatibility rules offer a greater control and a higher abstraction level than OpenMP style fork-join blocks, they are also better adapted to active-objects.

Our annotation system looks like JAC's proposal, and this paper could also be seen as an adaptation of a concurrency model *à la* JAC to the active object model. The inheritance model of JAC annotations is well designed [13] and resembles the way our annotations are inherited from class to class. However, multi-active objects offer a simpler annotation system and a higher synchronisation logic encapsulation. Moreover, the dynamic compatibility rules of multi-active objects are not directly translatable into JAC annotations: JAC provides a precondition mechanism that can be used to express dynamic compatibility but it does not guarantee safe access to shared variables. Compared to JAC, we think multi-active objects are simpler to program, have stronger properties, and are better suited to distribution. In particular, the transparent inclusion of annotations leads in our opinion to a powerful and interesting programming model.

Now that our active objects are equipped with multiple threads, strategies for optimizing the number and utilization of threads will have to be considered (e.g. [18, 11]).

6 Conclusion

In active object languages, programming efficiently on multicore architectures is not possible. Indeed, to have multiple threads on the same machine, all the other languages require to create multiple active objects (or coboxes). Then, the communication between those objects is either costly (it relies on a request invocation and heavy parameter passing), or restricted (JCoBoxes can share a state but it must be immutable). In our case, several threads inside an active objects can co-exist and we provide an annotation system to control their concurrent execution. The annotations can be written from a high-level point of view by declaring compatibility relations between the different concerns an active object manages. The operational semantics defined in Section 3 allowed us to prove that request services can be scheduled such that *parallelism is somehow maximised while preventing two incompatible requests from being served in parallel*.

The originality of our contribution lies in the interplay between the formal and precise study of the MultiASP language, and a middleware implementation efficient enough to compete with classical multi-threading benchmarks. The use of dynamic constraints for compatibility allows a fine-grain control over local concurrency and improves expressiveness. We implemented the proposed model in Java and ran experiments to ensure that our approach is efficient. The experiments showed that the performance of multi-active objects is similar to the manually multi-threaded version while code dedicated to parallelism is much simpler when using multi-active objects. We also illustrated the performance gain brought by multi-active objects compared to a classical

active object version. Overall, *multi-active objects outperform simple active objects, and are easier to program than classical multi-threading*. Interleaved execution and race-conditions can of course appear due to multithreading, but, on one side, annotations provide a good way to control this non-determinism, and on the other side, we defined an operational semantics that will allow us to study the possible executions of a multi-active objects and extend properties that the authors proved in the past for ASP.

References

1. Lavender, R.G., Schmidt, D.C.: Active object: an object behavioral pattern for concurrent programming. In: Pattern languages of program design 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1996) 483–499
2. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming ABCL/1. In: Conference proceedings of OOPLSA '86, NY, USA, ACM (1986)
3. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous sequential processes. Information and Computation **207**(4) (2009) 459–495
4. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)
5. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming **7**(1) (1997) 1–72
6. Halstead, Jr., R.H.: Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems (TOPLAS) **7**(4) (1985) 501–538
7. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science **365**(1–2) (November 2006) 23–66
8. Schafer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. ECOOP 2010–Object-Oriented Programming (2010) 275–299
9. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented programming in ambienttalk. ECOOP 2006–Object-Oriented Programming (2006)
10. Boer, F.S.D., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: Proc. of ESOP'07. Volume 4421 of LNCS., Springer (2007)
11. Henrio, L., Huet, F., István, Z.: A language for multi-threaded active objects. Research Report RR-8021, INRIA (July 2012)
12. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, ACM (2001) 161–172
13. Haustein, M., Lohr, K.: Jac: declarative Java concurrency. Concurrency and Computation: Practice and Experience **18**(5) (2006) 519–546
14. Shanneb, A., Potter, J., Noble, J.: Exclusion requirements and potential concurrency for composite objects. Science of Computer Programming **58**(3) (2005) 344–365
15. Frumkin, M., Schultz, M., Jin, H., Yan, J.: Implementation of NAS parallel benchmarks in Java. In: a Poster session at ACM 2000 Java Grande Conference. (2000)
16. Scholliers, C., Tanter, É., De Meuter, W.: Parallel actor monitors. In: 14th Brazilian Symposium on Programming Languages. (2010)
17. Cunha, C., Sobral, J.: An annotation-based framework for parallel computing. In: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, IEEE Computer Society (2007) 113–120
18. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410**(2-3) (2009) 202–220