An algorithm for safely stopping a component system

Ludovic Henrio, Marcela Rivera

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An algorithm for safely stopping a component system

Ludovic Henrio — Marcela Rivera

N° 6444

Janvier 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6444--FR+ENG

An algorithm for safely stopping a component system

Ludovic Henrio *, Marcela Rivera *

Thème COM — Systèmes communicants
Projet Oasis

Rapport de recherche n° 6444 — Janvier 2008 — 18 pages

Abstract: This report proposes an algorithm for safely stopping a subsystem of a component assembly. More precisely, it safely stops a component and all its subcomponents in a distributed and hierarchical component model. Our components are distributed, autonomous, and communicating asynchronously. Thus, one of the great challenges addressed by this report is to synchronize those components during a process which involves their deactivation. This algorithm has been prototyped and experimented on a distributed component example. This report also describes the main properties of our algorithm (both correctness and termination) and its requirements on the component system to be stopped.

Key-words: component, synchronization, distribution, reconfiguration

* INRIA, Sophia Antipolis, Université de Nice – I3S – CNRS

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Un algorithme d'arrêt pour un système de composants

Résumé : Ce rapport de recherche présente un algorithme permettant d'arrêter sûrement un système de composants. Plus précisément, cet algorithme arrête d'une faiçon fiable un composant et tous ses sous-composants dans un modèle à composants distribuiés et hiérarchique. Ce rapport décrit aussi les principales propiétés de l'algorithme et les conditions sur le système à composant permettant l'arrêt.

Mots-clés : composant, synchronisation, distribution, reconfiguration

Contents

1 Introduction

This report is situated in the context of distributed component programming, and more particularly it solves the problem of stopping a portion of a hierarchical component assembly in the case of distributed autonomous components.

As underlined in [9], maintaining integrity during reconfiguration and adaptation is crucial. More precisely, in order to be reconfigured, a component assembly should reach a state where components are stopped, and considered as easily reconfigurable. This report proposes a characterization of states where reconfiguration is easy, together with an algorithm to reach it while stopping a component subsystem. Such a general stopping algorithm together with reconfiguration primitives will provide autonomous adaptation capabilities to distributed component systems.

Loosely coupled component models are more adapted to distribution purposes. Indeed, loose coupling reduces the impact of latency, and limits the interleaving between components: each component is responsible for its own state. Moreover, loose coupling allow autonomic management of components. That is why we think that a distributed component system should rely on relatively autonomous component communicating by asynchronous communications, and not sharing any memory. To increase the programmability of the model, we need a mechanism allowing the programmer to use the results of asynchronous calls without the burden of call-backs. We suggest to use *futures* for this purpose: a future [16, 5] is an identifier (kind of empty object) that represent the result of an asynchronous method calls that has not been calculated yet. We consider "first class futures" that can be transmitted between processes.

To sum-up, we consider a component model that separates the components into autonomous, and relatively loosely coupled entities. Thus, our algorithm relies on the following assumptions relatively to the component model:

- Components do not share memory, and there is no shared component, i.e. component hierarchy is a tree.
- Components communicate by asynchronous *requests* which can be remote method calls or any other asynchronous communication.
- All communications are performed over some "bindings" meaning that two components cannot communicate by bypassing the component hierarchy.
- The communication mechanism can be instrumented by adding informations to every message.
- It is possible to identify safe states for any given component: here, a *safe state is a point where the component is idle and has no request to serve.*

The GCM component model [6] is a component model that satisfies most of those prerequisites, that is why we made a prototype implementation of our algorithm over this component model. Relying on the features mentioned above, our algorithm has the following characteristics:

- It is able to stop a subsystem in a safe state, that is in a state where the status of the inner components can be reduced to a minimum, to allow the reconfiguration of the stopped component.
- We consider a state as safe if all the subcomponents of the stopped one have finished their pending requests. If the component is stateless, it can then be replaced by any newly created one.
- It is adapted to a component model where components are autonomous entities communicating by asynchronous messages.
- It is not particularized for a given configuration, or for a predefined component property: it is able to handle any component topology, with possibly state-full components, and any communication pattern.
- It supports *replies* [5] consisting of a value sent for replacing a future object.
- If the system cannot be stopped safely, the algorithm never finishes but does not block, this keeps the integrity of the system and let it run normally.

This report is organized as follows. Section 2 reviews related works, then Section 3 describes the way our algorithm for stopping components works. Section 4 details the algorithm, and its properties are given in Section 5. Section 6 presents the implementation of our algorithm that is evaluated in Section 6.2.1. Finally, Section 7 concludes the report and presents the futur works.

2 Related Works

The main motivations for reconfiguration of systems are mentioned in [12]. The authors present some reasons for which an adaptation may be performed and it also mentions some strategies for dynamic reconfiguration of applications. This work also gives aspects to be considered in order to perform a dynamic reconfiguration, and problems caused by dynamic modifications of the system.

In the work of Rasche and Polze [17], they design and implement a framework for dynamic component reconfiguration on Microsoft .NET environment. They presented an experimental evaluation of their infrastructure for dynamic reconfiguration of component-based applications. Their reconfiguration process is limited to three basic operations: adding or removing a component, and setting its attributes. Their goal was to find an algorithm able to reconfigure an application without disrupting the service. Their proposal consists in reconfiguring an application if its components don't interact and if there are no pending transactions. However, they do not have an algorithm for stopping the system, neither, for dynamically replacing of components involving state transfer. Their algorithm blocks the communications in order to reach a reconfigurable state. This work is essentially valid in mobile applications because they rely on few internal interactions.

The CASA approach [15] also provides a framework for enabling the development and the operation of autonomic applications. In this framework Mukhija and Glinz propose a sequence of steps for dynamic replacement of some components of a system [14]. But this process doesn't apply to our requirements, and their definition of components is rather close to the notion of objects. For example, contrarily to Fractal, to GCM, or to the definition of component given by Szyperski [18], CASA components do not specify the interfaces they require.

In [13], the authors presents a component-based programming framework. In this framework the applications can be autonomically reconfigured to manage the dynamism and uncertainty of the applications and the environment. They enable the description of the dynamic replacement/addition/deletion of components, and the change of interaction relationships. Autonomic composition and evolution is expressed as a list of rules.

In all these works, no support for hierarchical components is provided; although hierarchical component models like SOFA [10] or Fractal [3] have great advantages concerning management and design of component systems. However, the frameworks presented above would benefit from an algorithm for stopping a set of component. As explained above, such an algorithm should stop a component subsystem in a safe state where the reconfiguration is easier or at least possible.

The implementation of our algorithm relies on the GCM [6], which is a Grid extension to the Fractal component model [3, 7]. Fractal is a hierarchical component model, meaning that each component is either composite (if it is composed of other components), or primitive (if its content cannot be decomposed). These components interact through *method calls* between client and server interfaces. From Fractal, GCM inherits a hierarchical structure with strong separation of concerns between functional and non-functional behaviours, including a very basic life-cycle and binding management. GCM extends Fractal with support for deployment, many-to-one and one-to-many communications, autonomic behaviour, and a better structure for non-functional concerns. This report can also be considered as a proposal for an extension of the life-cycle controller of Fractal for distributed systems (e.g. Grids).

3 Brief Description

3.1 Objectives

We propose an algorithm for stopping a GCM component system. It recursively stops a *master component* together with all its subcomponents in a safe state, i.e. without any request to be treated. Depending on the reconfiguration to be handled, and on the components involved, different states can be considered as safe, and a given component can be stopped at different times. This report clearly takes the safest assumptions allowing to stop any assembly in a state that can be considered as safe for any reconfiguration. Of course, this means stopping the component later that it could be, and some component systems could never be able to stop (e.g. if they are involved in a live-lock); but we prefer that a

component never stops instead of stopping it in an inconsistent state where it will not be reconfigurable. Relaxing the hypotheses in order to stop a component sooner is relatively easy from the general algorithm given here.

During a usual run (without stopping a component), a component is considered to receive requests, in an asynchronous manner, serve them in a FIFO order, and return a result, via a *reply* message. A reply [5] consists in a value sent to replace a future identifier. A component is single threaded, except that it can receive replies and requests at any time (requests are enqueued but replies affect the internal state of the component). The stopping algorithm is given as an extension of the usual run able to take into account *stop* messages.

In the whole article, for simplicity, we will present our algorithm supposing that only one component is to be stopped (only one master component). Section 4.5 will explain how to extend this algorithm in order to stop many components, possibly simultaneously.

3.2 Principles

We call *master* the component that receives initially a *stop request*. The algorithm safely stops the master component and all its subcomponents.

We call *re-entrant requests* the ones that are consequences of requests originating from inside the master and that might have to be treated in order to be able to stop the subsystem: one component inside the master might hold a future for the result of this request, and perform a wait for the associated value. The algorithm relies on a marking of requests working as follows: the master component marks some requests, and each time a request is sent during the service of the marked request, this request is marked too. Only the master do not propagate marks. In practice, only requests outside the master component are marked; marking is only useful to identify re-entrant requests, and to avoid dead-locks involving this kind of requests. This addresses the case where a component waits for messages corresponding to requests it has sent, or requests triggered by requests it has sent, recursively.

The global idea is that the algorithm runs in two phases; in a first phase the master component marks all the requests it sends. This phase lasts until all the requests for which it waits the results are marked. In the second phase the master component blocks all the requests it receives (except the marked ones) and the inner components continue processing their requests; when all the components are idle, and all inner components have empty request queues, the components are stopped. During the shutdown process, the components take several states described as:

Started: The component is running normally (initial state).

Wait for stopping: Only the *master* component can enter this state. It lasts from when the master component receives the **stop signal** until it has updated all non-marked futures. In this state, the master component marks all its outgoing requests. At the end of this state, the master sends a **stopping signal** to all its subcomponents.

Stopping: Starts when the component receives a **stop signal** (or exits the **wait for stopping state** for the master). The master component only serves *marked requests*, the other components serve all the requests.

Ready to Stop (R2S): Starts just after the component sends the **ready to stop signal**, until it receives the **stop signal**. To be in the **R2S state** a component must be idle and all its subcomponents must be in the **R2S state**. It can't process requests. Before being able to handle a new request, it must get back to the **stopping state**.

Note that, as soon as all the subcomponents of a component C are in the **R2S state** and C is idle, none of its subcomponent has to handle a request before C has to receive one (this is due to the absence of sharing).

The **not ready to stop state** (**NOT_R2S**) is the complementary of the **R2S state**.

Stopped: The component and its subcomponents are stopped. It is idle and can't emit or receive requests. It can be safely reconfigured.

Definition 3.1 (Strongly idle state) *An idle component composed of idle subcomponents with* empty request queues *is said to be in the* ***strongly idle state****.*

3.3 Requirements

We present here the conditions required for our algorithm to work. First, at the beginning, all components are in the **started state** and consider their subcomponents are in the **NOT_R2S state** (e.g. this information is stored and initialized to the NOT_ R2S value).

Condition 1 (Initial state) *Initially, all components are in the* ***started state****. All components are considered by their parent as in the* ***NOT_ R2S state****.*

Second, markings are propagated by requests calls *outside the master component.*

Condition 2 (Marking calls) *All requests sent by the service of a marked request are marked too (except if it is the master that serves the request).*

Condition 3 (Dead-locks) *In a normal run, every request service terminates.*

Condition 4 (Fairness) *Every received request is served after some finite time.*

As a consequence of fairness and absence of dead-lock, we can ensure that:

Property 3.1 *Every future is updated after sometime.*

4 Algorithm Description

We present in fact two algorithms. One for the master component (Figure 1) and the second for the subcomponents (Figure 2). Each component is responsible for locally storing the **R2S state** of its subcomponents according to the **R2S signal** or **NOT_R2S signal** received. Thus, every component knows whether its subcomponents are in the **R2S state**, and updates its **R2S state** (and sends the **R2S signal** to its parent).

4.1 Master

The master component receives the *master stop signal* and is responsible for sending the **stopping signal** to its subcomponents. The shutdown process of the *master component* is divided into two phases.

The first phase consists in ensuring that all the requests for which a result may be awaited are marked. This is necessary in order for the re-entrant request mechanism to work properly (see Section 4.4).

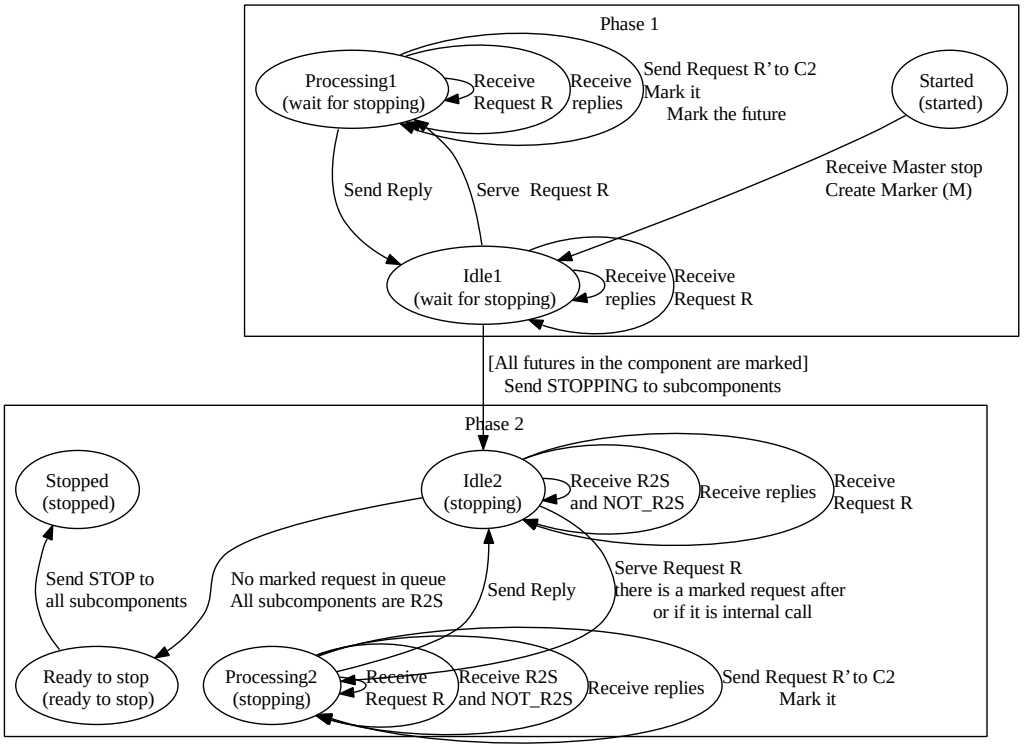

Figure 1: Algorithm of the master component

The second phase is synchronized with the algorithm for subcomponents. It still marks all outgoing requests but filters incoming ones. The second phase starts by notifying all subcomponents by sending a **stop signal**.

In the second phase the *master component* coordinates the process of stopping all the subsystem. It is responsible for knowing whether its subcomponents are in the **R2S state** (using the **R2S signal**). When the component knows that all its subcomponents are in

the **R2S state**, and if the master is idle, then the master component is in a **strongly idle state**. Thus, the master component can send the **stop signal** to all subcomponents, which finally stops the system.

In this phase, the master component acts as a barrier: it doesn't allow its subcomponents to receive external calls, except if these externals calls are marked. If a marked request arrives, the master component serves all functional calls until this marked call in order to respect the FIFO ordering of request serving. Except from this, replies and requests are served, sent and received usually.

Note that, if a component has the additional property that requests can be served in any order then only marked requests can be served, and the other functional requests before the marked ones can be kept stored, which accelerates the stopping procedure. In the general case, for taking into account any request ordering requirements, a partial order between requests could be defined. Such an order would state which requests have to be served before a marked one. We considered here two particular orderings: FIFO (total order), and no ordering.

4.2 Subcomponents

A subcomponent stopping process is simpler, and similar to the second phase of the master. It begins when the master component starts the second phase, and sends to the subcomponents the **stopping signal**. Each subcomponent first propagates the **stopping signal** to its subcomponents, and then continues processing until its queue becomes empty.

In particular, each subcomponent must be in the **stopping state** to accept a request. Thus, before being able to accept a request, a component in the **R2S state** must first change its state to the **stopping state** and send the **NOT_R2S signal** to its parent.

When a component knows that all its subcomponents are in the **R2S state**, is idle, and has a empty request queue, the component can send the **R2S signal** to its parent. At this point, the component and recursively all its subcomponents are in the **strongly idle state**.

4.3 Algorithm Synchronization

We describe briefly the different phases reached by the system, and how those phases are triggered and synchronized:

- Initially all components are in the **started state**.
- When the *master component* receives the **master stop signal** it changes to the **waiting for stopping state**.
- While the *master component* is in the **waiting for stopping state** it marks all the outgoing requests.
- When the *master component* has marked all functional calls it changes to the **stopping state** and it sends the **stop signal** to its subcomponents.

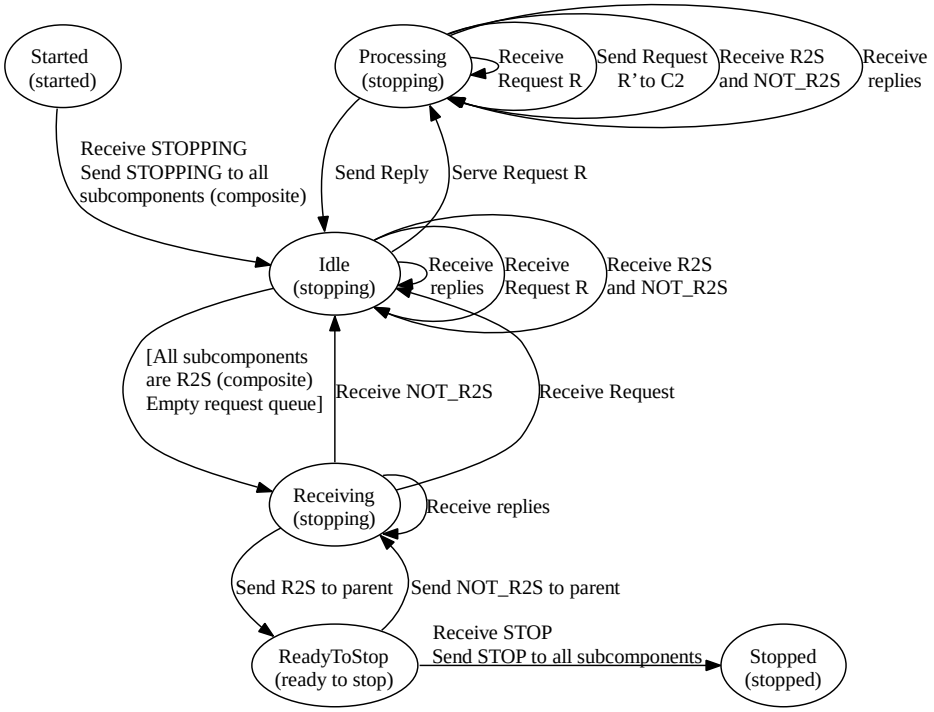

Figure 2: Algorithm of the subcomponent

- When the subcomponents receive the **stop signal** they reach the **stopping state**.
- When there are no more internal calls to process (queues of the subcomponents are empty), each component sends to its parent a **R2S signal**. Then, each composite changes to the **R2S state** as soon as all its subcomponents are in the **R2S state**. At the end of this process the *master component* is in the **R2S state**.
- When the *master component* is in the **R2S state** it sends the **stop signal** to all its subcomponents, which propagate the signal. Then, the *master component* changes to the **stopped state** and, consequently all subcomponents change to the **stopped state**; finally the system is stopped.

4.4 Preventing Deadlocks: the Re-entrant Requests Mechanism

First note that, necessarily, methods that have been sent before the master **stop signal** are not marked. If the future for this request is awaited and the request needs to call-back the master component (re-entrant call), then the call-back request will not be marked and then

not be served, leading to a dead-lock. Once all the futures inside the component correspond to marked requests the *master component* triggers the second phase where only marked requests are served (as described in Section 4.1).

A call is necessarily re-entrant at the master component level because all communications transit by the bindings. Thus once all emitted requests have been marked, all the consequences (requests) of this marked request are marked, and thus are-entrant request is easily identified.

This mechanism is sufficient for ensuring the termination of the algorithm in usual cases; unfortunately, due to the asynchronous and imperative nature of the model, it is possible for a component outside the master to store a future issued before the marking of requests, and return it later, during the second phase of the master. Such a behaviour is not frequent but would break the invariant that all futures awaited by the master are marked in the second phase, and could cause deadlocks if the non-marked future depends on a request addressed to the master. In that case, the solution consists, for the component that stored the future, in sending a message to the master stating that a dead-lock might occur, and the master would restart the stopping algorithm from the beginning (with more marked futures). This mechanism finally ensures the absence of dead-locks without impeding the reachability of the **stopped state**. In general, this mechanism is not needed.

Property 4.1 (No dead-lock) *The algorithm does not create any additional dead lock (it might have to be restarted in case an unmarked future is awaited outside the master in the second phase).*

4.5 Stopping Several Components

It is easy to extend the presented algorithm so that it is implemented by all components and any two components can run it simultaneously. First, this means that every component features both the master component and the subcomponent algorithm. This also requires marking states (and the marked requests) with the identifier of the master component that is being stopped. This way different stopping procedures can be considered as independent.

4.6 Non hierarchical systems

Stopping a simple isolated component is simple, both in a hierarchical system and in a flat one. But our algorithm might be of more interest for synchronizing the stopping process of a set of components, even not hierarchically organized.

Indeed consider a flat system, and a set of components to be stopped. One can adapt our algorithm to stop this *assembly* in a synchronized way. In order to do this, the algorithm for the master component should be split into

- one component chosen to be the central manager, storing the status of each of the component to be stopped and synchronizing them;

- an interception of the calls outgoing the assembly, in order to mark them, this must be added to all the components having a client interface leading outside the assembly;
- an interception of the calls incoming into the assembly, in order to store them, or serve them if they are marked, this must be added to all the server interfaces accepting calls from outside the assembly.

5 Properties of the Algorithm

This section is divided into two parts: we first give properties on the stopped system of components. This ensures that our algorithm stops the components in a safe state and does not modify the usual execution. The second set of properties ensure that the stopping algorithm is able to terminate under reasonable conditions. Each property comes with an informal proof. Due to the complexity of the algorithm, formalizing those proofs is out of the scope of this report.

5.1 Components are Stopped in a Safe State

When the *master component* sends the **stop signal** to its subcomponents, it necessarily has received a **R2S signal** from all subcomponents (and each subcomponent has received a **R2S signal** from all subcomponents, recursively). A primitive component (that has no subcomponent) reaches a **R2S state** if and only if it is idle and its queue is empty. Thus, by induction, a component (which is not the master component) reaches a **R2S state**, if and only if, it is idle, all its subcomponents are in the **R2S state**, and its queue is empty. Consequently, if a component is in the **R2S state**, then it is necessarily in the **strongly idle state** (Definition 3.1). The queue of each component but the master is empty:

Property 5.1 *When stopped, the master component is in the **strongly idle state**.*

If a marked request is to be served, then the *master component* serves all requests before this one. Thus the order of treatment of request is still the order of arrival, which implies Property 5.2. Note that, non marked requests received after this "last" marked request are stored (enqueued) in the master queue:

Property 5.2 *When stopped, the master queue only contains requests received after the last served request.*

5.2 Termination of the Algorithm

From reachability of the **strongly idle state**, reachability of the stopped state can be proven as follows. Suppose each subcomponent has an empty request queue and is idle. In Figure 1 and 2, the components are all in the **Idle state** (stopping). As primitive components are idle with empty queues, they can switch to the **R2S state**. Then, at the level above each component is idle, has an empty request queue (except the master), and its subcomponents

are in the **R2S state**; thus it can reach the **R2S state**. Finally, the master and recursively all the subcomponents send the **stop signal** to their subcomponents, and reach the **stopped state**.

Property 5.3 (Reachability) *If after reaching the **stopping state** the master reaches a state where every subcomponent has an empty request queue and is idle then it can be stopped.*

The above termination property is quite restrictive, and the algorithm terminates under much weaker conditions. But identifying the necessary and sufficient condition for termination is far from trivial, here is a weaker sufficient condition:

Condition 5 (No live-lock) *For each request Q, after sometime, there is no more requests sent as a consequence of Q.*

Suppose the interleaving of request at the entrance of the component ensures that the flow of marked request terminates at some point. As a consequence, after sometime, the master component does not receive requests that require a re-entrant request to be treated: after sometime, no more marked requests are received by the master component.

Property 5.4 (Termination) *If after some point in time, the master component does not receive any more marked requests, and if the components are live-lock free, dead-lock free, and serve all the requests they receive (Conditions 3, 5 and 4), then the stopping process terminates.*

Note that it is easy to adapt the algorithm in order to cancel the "stop" order if the termination cannot be ensured.

6 Implementation

6.1 GCM/ProActive

A GCM reference implementation is based on ProActive [4]. In this implementation each component and each composite membrane is an active object. The controllers are encapsulated in the membrane which also dispatches functional calls to inner components. In this implementation, components communicate through asynchronous method calls with futures. The futures can be forwarded to any component in a non-blocking manner. A constraint inherent to this implementation is the absence of shared memory between components.

6.2 Evaluation

The algorithm presented in this report has been tested on the CoCoME example [1]. This example consists of a "point of sale" system and its surrounding infrastructure. It simulates the interaction between different components (Bank, Inventory and CashDeskLine). The full example, implemented in GCM/ProActive, is described in [8]. Each component implements

the two algorithms (master and subcomponent), but only one is used at a given time, depending on the component that receives the **stop signal**. The implementation of the example is available in [2].

To manipulate additional messages used by the stopping algorithm, we implemented controllers for each component, signals sent between a component and its subcomponents, or parent are exchanged by these controllers. From the point of view of the functional code the manipulation of these signals is transparent.

Actually, we have been able to stop individually all the component of the CoCoME example at "any" time without running into a dead-lock. For primitive components, the algorithm only waits for the arrival of futures (first phase).

6.2.1 Performance Analysis

We show the performance of our algorithm on the CoCoME example. We have performed some preliminary analyses to know the behaviour of our stopping process for different cases. The tests were run on a Intel Core 2 Duo 2.4GHz with 4096 MB of RAM, running ProActive v3.2 under Fedora 5.

We consider two cases, trying to stop two different components. In the first case the component to be stopped has two subcomponents. They receive external requests, but no request is issued by the master component. In the second case a subcomponent of the master component generates requests that goes out of the master, and come back as re-entrant requests. The inner component waits for the completion of the re-entrant request before continuing its execution.

Delay $[s]$	Size of queue $[number\ of\ requests]$	Time to stop Case 1 $[s]$	Time to stop Case 2 $[s]$
0.1	84	23.1	35.2
0.2	142	33.3	52.6
0.4	179	38.2	62.5
0.5	199	43.7	73.5
0.8	210	44.9	74.1
1.0	235	54.5	83.0
1.5	242	55.3	87.8
2.0	247	55.7	88.3
2.5	251	56.6	88.4

Figure 3: Time necessary for the algorithm to reach a safe state

Figure 3 shows the average time to stop a subsystem depending on the number of requests in the queue of the components. For controlling the number of requests in the queue we add a delay in the processing time for one of the requests treated by a subcomponent of the master. For example, in a second, the component can serve (without delay) approximatinely 250 requests per second. Introducing a delay of 1/100 second, less than 100 requests can be treated in a second, delaying the service of approximately 190 requests each second. Consenquently with a delay of 1/100 second the queue grows of 190 requests each second.

7 Conclusion and Future Works

This article presented an algorithm that stops safely a component together with all its subcomponents, belonging to a distributed component system. We are placed in the context of distributed autonomous hierarchical components communicating by asynchronous method calls, and possibly replying by means of replies (future updates). The algorithm can be applied and lead to a safe state in very general conditions: no assumption is made on the components dependencies or the components behaviour. Such a safe state is a prerequisite for any safe reconfiguration of the subsystem: the components are stopped in such a way, that their state is minimally characterized and they are no interrupted in the middle of a request treatment. The algorithm has been presented using GCM as the component model, but it is applicable to any hierarchical component model, and we have shown that it can be adapted to non-hierarchical ones.

The innovative aspects of this algorithm is the way synchronization between components is handled, and the notion of re-entrant request that has been defined in order to deal with futures. Without detection of re-entrant requests, any algorithm trying to stop a component subsystem would result in a dead-lock. Indeed waiting for a future value blocks a component while the corresponding result has not been computed. This computation might involve a request addressed to the component being stopped, if this request is not identified and treated, the component trying to access the future will be indefinitely stuck.

In the near future, our aim is obtain a definition of *safe state* with less constraint and with more guarantees. One of our major objective is to verify formally the stopping algorithm presented here, but the complexity of the algorithm is a hard obstacle.

We also plan to consider hierarchical systems where some of the components can be shared, in the sense that they belong to two components. If a shared component is inside a component to be stopped, it must first mark all the invocations it performs (because a shared component can contact the exterior of the system to be stopped). However, this marking is not sufficient because requests might arrive from a component that is not in the system to be stopped, and the R2S status of the component cannot be maintained simply for these components. We plan on extending our algorithm to shared components in the future. On a longer term, we will develop more deeply the reconfiguration of components and we would achieve the flexibility, safety and efficiency in a very general setting; we could rely on [11] that shows experiences with safe reconfigurations.

References

[1] L. Henrio E. Madelaine M. Rivera E. Salageanu A. Cansado, D. Caromel. *The Common Component Modeling Example: Comparing Software Component Models, ch. A Specification Language for Distributed Components implemented in GCM/ProActive*. 2007. `http://agrausch.informatik.uni-kl.de/CoCoME`, to appear.

[2] Algorithm for stopping a component implemented in CoCoME example. Available at `http://www-sop.inria.fr/oasis/personnel/Marcela.Rivera/`.

[3] E. Bruneton, T. Coupaye, M. Leclercp, V. Quema, and J. Stefani. An open component model and its support in java. In *7th Int. Symp. on Component-Based Software Engineering (CBSE-7)*, LNCS 3054, may 2004.

[4] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[5] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer-Verlag New York, Inc., 2005.

[6] CoreGRID, Programming Model Institute. Basic features of the grid component model (assessed), 2006. Deliverable D.PM.04, `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

[7] E.Bruneton, T.Coupaye, and J.B. Stefani. The Fractal Component Model `http://fractal.objectweb.org/specification/index.html`. Technical report, ObjectWeb Consortium, February 2004.

[8] CoCoME example implemented in GCM/ProActive. Available at `http://www-sop.inria.fr/oasis/Vercors/Cocome/`.

[9] I. Hillman, J.; Warren. An open framework for dynamic reconfiguration. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 594–603, 23-28 May 2004.

[10] Petr Hnětynka and Tomáš Bureš. Advanced features of hierarchical component models. In *Information Systems and Formal Methods*, pages 3–10, 2007.

[11] Jean-Bernard Stefani Juraj Polakovic, Sébastien Mazaré and Pierre-Charles David. Experience with implementing safe reconfigurations in component-based embedded systems. In *The 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2007)*, pages 242–257, Boston, MA, USA, 2007. Springer Berlin/Heidelberg.

[12] Abdelmadjid Ketfi, Noureddine Belkhatir, and Pierre-Yves Cunin. Automatic adaptation of component-based software: Issues and experiences. In *PDPTA '02: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1365–1371. CSREA Press, 2002.

[13] H. Liu and M. Parashar. A component based programming framework for autonomic applications. In *First International Conference on Autonomic Computing (ICAC'04)*, 2004.

[14] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Systems Aspects in Organic and Pervasive Computing - ARCS 2005*, pages 124–138. Springer Berlin / Heidelberg, 2005.

[15] Arun Mukhija and Martin Glinz. The casa approach to autonomic applications. In *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks, ASWN 2005*, pages 173–182. IEEE Computer Society, 2005.

[16] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, November 2006.

[17] Andreas Rasche and Andreas Polze. Configuration and dynamic reconfiguration of component-based applications with microsoft .net. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, page 164, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Clemens Szyperski. *Component software: beyond object-oriented programming.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399