Implementation of a fault-tolerant system for solving cases of numerical computation

Cyril Dumont, Fabrice Mourlin

Algorithmic, Complexity and Logic Laboratory, Paris 12th university, 61 avenue du Général de Gaulle, 94010 Créteil Cedex, France
cyril.dumont@u-pec.fr, fabrice.mourlin@u-pec.fr

Abstract

We have realized MCA framework (for Mobile Computing Architecture), allowing the resolution of cases of numerical calculation in a heterogeneous distributed environment. Its features are adaptability and reactivity. The first facet is to achieve the end of the calculation even if computing resources are not reliable. The second facet means that depending on context, mobile agent can move part of calculation on to other computing resources. Finally, we have mixed mobility and virtual machine use. So, our solution allows developers to use computation codes written with a different programming language (C++, C). Our results validate our approach and provide a showcase for new evaluations.

Keywords: distributed system, mobile agent, numerical calculation, reliability, space computing.

1. Introduction

Distributed systems are fundamentally different from non-distributed systems in the fact that there are situations in which members of a distributed system are no longer able to communicate with other members of the same system. The reason is that a member of the community is down or because the connection between community members no longer works. This partial failure which is the failure of part of the system can occur at any time and may be intermittent or long term.

Then, it becomes clear that the implementation of a distributed system is a challenge to be tolerant to these various failures. Indeed, the ability to react to a failure in this type of system greatly complicates the work of developers. But it is a quality mark and a definite asset to differentiate itself from other systems. Distributed systems link together components that provide resources or services to each other [5, and a failure of one or more of them may lead to resources that are never released, with services that continue to run while the service recipient is waiting for response. These situations can lead to a distributed system to unnecessary consumption of resources or, worse, his blocking total.

First, we present the general architecture of the computing platform based on the MCA framework. Subsequently, we list the various tools and paradigms proposed by API Jini™ framework used in the MCA as the concept of services and mobility and Javaspaces technology. We detail the various elements of this architecture. Finally, we provide an overview of the solutions offered by the framework MCA.

2. Overview of the architecture MCA

In this section we present the software foundation on which MCA framework is based. Initially, we present Jini™ technology (version 2.1) because MCA architecture is essentially based on this technology using different basic services it offers. As an example, *MCAService* service is a Jini™ service, and as such it has the same characteristics as any other Jini™ services, particularly the ability to define a security policy around remote invocations of its methods. In a second step, we use technology Jini™ to implement a system of mobile agents (figure 1) useful for components called *ComputeAgent* and *ComputationCase* in our architecture (2.2). This type of system requires a security policy adapted to the mobility of code that we define in section 2.2.

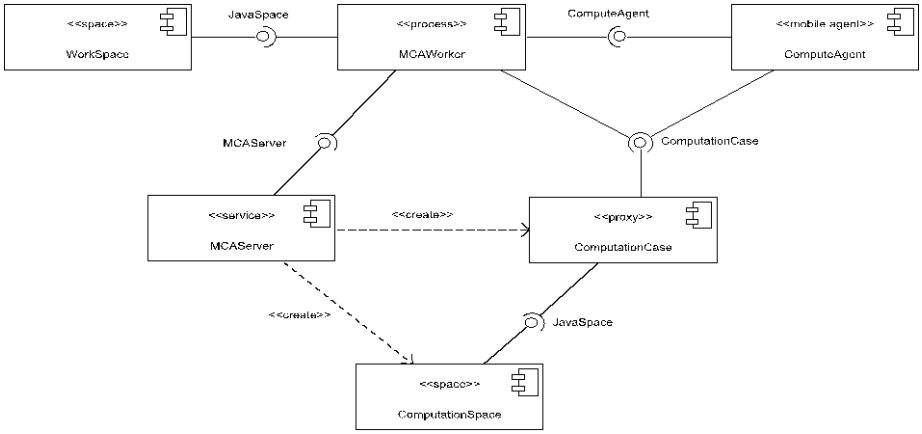

Figure 1: Component diagram of MCA Architecture

2.1. Service Oriented Architecture-based on technology Jini™

Jini™ technology [1] has been defined to support the development of distributed systems in a simple and elegant way. A key feature of this kind of system is adapting to change. Change in a distributed application is the arrival or departure of components that make up the system. One of the problems caused by it and to be absolutely considered, is the partial failure. Because in a distributed system, computers can work while others are stopped and on the other hand, all computers can be operational but the network connection may be faulty.

Today, several implementations are available: JSC, Seven, ServiceHost, Rio, Harvester, H2O and CoBRA. All these implementations were presented by Svetozar Misljencevic in [2]. For our software architecture, our choice was made to the reference implementation as it has already been used by our team about other work [3] and this experience is an advantage. This implementation, developed by Sun, is called Jini™ [4].

2.2. Definition of a Jini™ service

Jini™ is based on the concept of service, which allows you to create systems with a service-oriented architecture. Jini™ based application is composed of various services available on the network. Each service is actually available via a proxy object which we call proxy in the following section. When a client wishes to use a service, it has first to obtain a proxy service. The client then invokes methods

through this proxy. The latter then performs the desired method that supports communications across the network to the service itself.

If the arrival of a service, or transfer to another node on the network, is common in a distributed system, it is important that a client can easily find these services.

Jini™ offers a solution: the Lookup service, through which a customer can find the services they wish to use. In addition, Jini™ provides a set of protocols for discovery (discovery protocols) used by clients to discover services without prior knowledge of their location. If a customer wishes to use a service for the first time, it looks for the proxy for that service in a directory (Lookup) available on the network. If the location of the service has changed, then the customer can always find the service through this directory. Once the client gets the desired service proxy, it uses the proxy to interact directly with the service, and this time without the participation of the directory. Figure 2 provides the different protocols as described above.

2.3. Role of lease

Using leases to avoid Jini™ service using resources unnecessarily. For example, if a client requests via a lease to a service to maintain an accessible state and this client has a fault, then the client will be unable to renew the lease. The lease expires and the service will then release the resources used. Similarly, if the service and the client work normally but the network knows them fails, then the client will be unable to renew the lease which will allow the lease expires release resources used unnecessarily.

The lease term is present when registering for a service, such as service *MCAService*, to the directory service offered by Lookup. Upon registration, the service provider obtains a lease created by the directory corresponding to the service he has recorded. The service will save directory as the provider of this service will renew the lease. If the service provider has to have a failure, the lease expires and the service proxy will be removed from the directory.

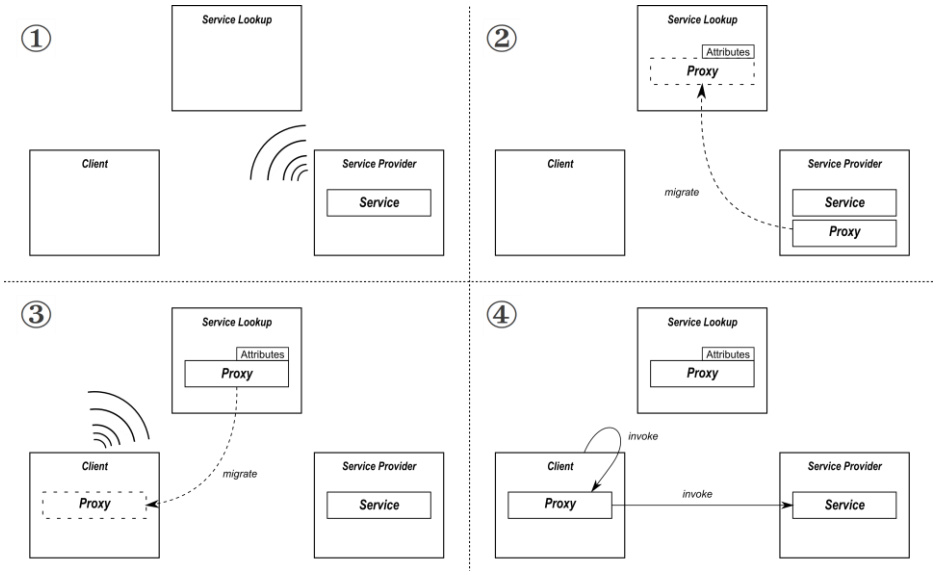

Figure 2: used protocols (1: discovery, 2: subscribing, 3: lookup, 4: invocation)

3. Fault tolerance in the "spaces"

ComputationSpace is a space dedicated to the resolution of a case of calculation. It is, therefore, a central component of the architecture proposed by the MCA framework for processes *MCAWorker* modify the entries contained in a *ComputationSpace* to solve the case of associated calculation. We also find the use of space with the component *WorkSpace*. These two types of components must be fault tolerant. We present in this section the different mechanisms offered by spaces to meet this constraint with distributed transactions, replication, and data persistence.

3.1. Distributed transaction

Distributed transactions are present in most distributed systems. Applied to an architecture based on spaces such as we propose with the MCA framework, distributed transactions can perform a series of operations with a space in a transactional manner. They provide a means to maintain the integrity of the entries contained in a space in the execution of a sequence of operations. *JavaSpaces* technology uses transaction model offered by the Jini™ technology. It offers a simple solution: all transactions are supervised one (or more) service Transaction Manager, an implementation called Mahalo is available with the Jini™ starter kit. An application wishing to perform operations in a transactional context will then ask for the service to create a transaction and the transaction once created, processes can then perform operations in this transaction. A transaction can end in two ways: success (via a commit), in this case all operations in the transaction are effective, or when a failure (via cancel the operation), in which case no operation will be dine.

3.2. Replication

The replication process allows duplicate entries in a space "source" to one (or more) space "target" to respond to failures that may arrive during the execution of a space. All instances of space share the same inputs as a replication group. To make a *ComputationSpace* fault-tolerant, it is defined as a replication group.
There are two kinds of replications. *Synchronous replication* ensures that the client receives the response from his operation on the source space only when all other spaces of the replication group have performed this operation. This mode of replication is most suitable topology "Assets - Liabilities" when the application needs to guarantee that no operation performed on the source space will be lost and not run on other spaces of the group. In return, this type of replication penalizes the performance of various operations.
In *asynchronous replication*, transactions are executed on the source space and the answer is immediately returned to the client. All operations are accumulated in the source space and are sent asynchronously to the target space after a period of time or after a specified number of operations. This type of replication provides better performance than synchronous replication but entries may be lost if a failure occurs source space when sending pending operations to the target space. This mode of replication is another problem: the consistency of data in the source and target spaces is not always guaranteed.

3.3. Memory persistence

A space is a temporary memory for the entries contained in a space are lost to stop it. To allow a space to recharge after a reboot entries, we chose to save a data source external to the space. *NoSQL* type databases (include "Not Only SQL") [6] proposes a solution perfectly suited for storing data. MongoDB [7], which provides a document-oriented *NoSQL* data-

base, stores data in *BSON* format [8]. Manipulation of data in this format is simple with the help of API provided for the Java language. Another advantage of this type of base is to be "schema-less", for example, it is possible to store data without any schema defined in advance.

4. Conclusion

We realize a platform able to respond to a failure in a distributed system. Failure of one or more components can lead to unusable state resources, or services that continue to run while the service recipient is waiting for a response. We show the capabilities of our system to react in such situations.

5. References

[1] J.Waldo and The Jini™ Team. The Jini™ Specifications. Addison-Wesley Professional, 2nd edition, 2000.

[2] S. Misljencevic. Jini™ Service Container. PhD Thesis, Universiteit Antwerpen, (2006)

[3] M. Bernichi. Surveillance logicielle à base d'une communauté d'agents mobiles. PhD Thesis, Paris 12 University, 2009.

[4] J. Newmarch. Foundations of Jini™ 2 Programming. Apress, 2006.

[5] Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Technical report, Global Grid Forum (2002)

[6] S. Tiwari. Professional NoSQL. John Wiley & Sons Ltd, 2011.

[7] M. Dirolf et K. Chodorow. MongoDB: The Definitive Guide. O'Reilly Media, 2010.

[8] Specification BSON. http //bsonspec.org/.