Assessment of the JADE Component Management Framework

SIMÓN AF FROSTERUS
LEONIDAS PANTZOPOULOS

ROYAL INSTITUTE OF TECHNOLOGY

Master of Science Thesis
Stockholm, Sweden 2007

ICT/ECS-2007-38

ROYAL INSTITUTE OF TECHNOLOGY

Assessment of the JADE component management framework

Master's Thesis

Leonidas Pantzopoulos

Simón af Frosterus

September 2006 – April 2007

Supervisors: Mr. Konstantin Popov and Associate Professor Vladimir Vlassov
Examiner: Associate Professor Vladimir Vlassov

SWEDISH INSTITUTE OF COMPUTER SCIENCE SICS

Abstract

Management of computing systems has always been an important issue that required effort, time and money. Lately, there is a trend towards autonomic computing that is supposed to drastically reduce the aforementioned burden and move management at higher levels. In both cluster based and wide-area networks, the use of component models helps move software construction to higher levels of abstraction and, hence, provide easier management of complex computer systems. There is a plethora of component models available but also a notable number of frameworks that target at providing autonomic management of computing systems. One of them is the JADE component management framework, based on the Fractal component model.

This thesis document gives a deep overview of the JADE component management framework analysing its architecture, its backbone component model, the management authority and the underlying technologies that it is based on. Most importantly it provides an assessment of the framework in terms of programmability, overhead and wide-area network capability. Through the assessment, JADE's weaknesses and strengths are pointed out and a set of conclusions is drawn.

Acknowledgements

We owe some gratitude to our academic advisor and examiner Professor Vladimir Vlassov and our industrial advisor Mr. Konstantin Popov because of their constant support and help throughout the project. We also would like to thank the Swedish Institute of Computer Science (SICS) for providing us with the necessary facilities to conduct our thesis and the Gird4All project.

Leonidas would also like to explicitly thank the Greek State Scholarships Foundation (I.K.Y.) for supporting financially his MSc studies throughout this period.

Separation of Concerns

This master's thesis was performed conjointly by two students and to fulfill KTH requirements regarding this case, the following table will attempt to clarify which student did what. Any section or part not attributed to anyone should be assumed to have been done by both students together.

Leonidas Pantzopoulos	Simón af Frosterus
1.2; 1.4; 1.5.1; 1.5.1.2; 1.5.1.6-8; 1.5.2; 1.5.3.1; 1.5.3.2; 1.5.3.5; 1.5.4; 2 (intro); 2.2 (intro); 2.2.2; 2.2.3; 2.2.7; 2.3 (intro); 2.5; 3.2; 3.4.2; 3.5	1.1; 1.3; 1.5.1.1; 1.5.1.3; 1.5.1.4; 1.5.1.5; 1.5.3.3; 1.5.3.4; 1.5.5; 1.6; 1.7; 2.1; 2.2.1; 2.2.4-5; 2.3.1; 2.4; 2.6; 3 (intro); 3.1; 3.3; 3.4.1; 3.4.3

Contents

List of Figures

List of Tables

List of Abbreviations

ACDL	Adaptation Contract Description Language
ADL	Architecture Description Language
AMM	Architecture Meta Model
API	Application Programming Interface
CBE	Common Base Event
CCA	Common Component Architecture
CCM	CORBA Component Model
CDL	Component Definition Language
CM	Component Model
COM	Common Object Model
CORBA	Common Object Request Broker Architecture
CRL	Collaborative Reinforcement Learning
DCOM	Distributed Common Object Model (COM)
DCUP	Dynamic Component UPdating
DOP	Discrete Optimization Problem
ECA	Event-Condition-Action
EJB	Enterprise Java Beans
GCM	Grid Component Model
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IC2D	Interactive Control and Debugging of Distribution
IDL	Interface Definition Language
IOR	Interoperable Object Reference
IT	Information Technology
J2EE	Java 2 platform Enterprise Edition
JAR	Java ARchive
JDBC	Java DataBase Connectivity
JMS	Java Message Service
JNDI	Java Naming & Directory Services
JVM	Java Virtual Machine
KB	Kilo Byte
KTH	Kungliga Tekniska Högskolan
LAN	Local-Area Network

MAS	Multi-Agent System
OBR	Oscar Bundle Repository
OMG	Object Management Group
ORB	Object Request Broker
OSGi	Open Services Gateway initiative
P2P	Peer To Peer
PDM	Problem Determination Mechanism
RL	Reinforcement Learning
RMI	Remote Method Invocation
SICS	Swedish Institute of Computer Science
SOAP	Simple Object Transfer Protocol
SOFA	SOFtware Appliances
TCO	Total Cost of Ownership
TCP	Transmission Control Protocol
WAN	Wide-Area Network
WS	Web Service
XML	eXtensible Markup Language

Chapter 1

INTRODUCTION

This first chapter will provide an introduction to the concepts of system's management, autonomic computing and component models, which together form the foundations of management frameworks.

1.1 MANAGEMENT

Systems' management is defined as the administration of information technology systems at any level, from requirements' analysis to problem handling to optimizations and everything in between. The cost of systems' management nowadays accounts for much more than the money spent in computer hardware and software together over the life span of almost every large computer system [15] [1]. The real cost of IT solutions is measured over time and includes the upfront investment as well as maintenance, management, and updates, a figure known as total cost of ownership (TCO) [18], that reflects in a more realistic way how much IT solutions cost. As it can be seen in Figure 1.1, the upfront cost of software and hardware, accounting for over 25% of the total cost of ownership, is considerably smaller than the management cost, in this case, of managed and non-managed systems. Current trends show this data to be fairly constant, with similar numbers across different IT scenarios [20] [21].

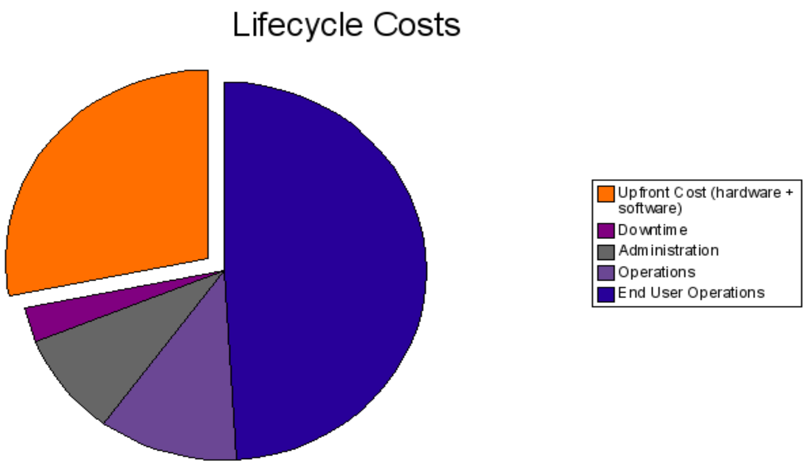


Figure 1.1. Total cost of operation for IT solutions [19]

What started with a centralized and hierarchical approach has now evolved into a much more complex issue spanning many different fields. As computer systems grow in size and spread throughout the world, problems multiply accordingly, and no easy solutions exist. The “manual” approach used in the early stages of systems' management, having physical managers dealing with the problems as they appear, has been obsolete and inefficient for many years, having many flaws. Lacking the ability to scale easily, extreme complexity and the need for inter-disciplinary knowledge are three of the main reasons why the “manual” approach is no longer viable and therefore new techniques are being developed to perform systems' management that meet the requirements of today's systems.

In a world where IT solutions rely on multi-tiered applications distributed and replicated possibly all over the planet, systems' management has become a challenge with no clear-cut solution. The most common technique relies on designing systems that will have the ability to manage themselves automatically, a technique known as autonomic computing or self-management.

1.2 Autonomic Computing

The term “autonomic” was not invented by computer scientists. There are earlier references in other sciences, with Biology being a typical example. On the classic work of Henry Grey back in the 18th century [14], the autonomic (or involuntary) nervous system of a living organism is defined as the part of the nervous system responsible for maintaining the body's stable condition by regulating its internal environment. Functions such as digestion, metabolism and blood pressure modulation are not taking place voluntarily although they are inside our cognizance. As a result, our conscious brain does not have to deal with all these low level activities.

In a similar way, autonomic computing targets at facilitating the management of computer systems by moving management at higher levels. This goal is of major importance in order to cope with the continuous increase in computer systems' complexity. Autonomic computing can not only alleviate the administration burden, which consequently reduces the administrative cost of computer systems, but also reduce erroneous installations/configurations/runs caused by human faults. Human interference is limited to providing policies which are definitely more clear and understandable. However, at that level, a human error on specifying a goal is magnified becoming more influencing and crucial.

Jeffrey O. Kephart and David M. Chess (2003) [1] argue that *"the essence of autonomic computing systems is self-management"* which aims at freeing system administrators from dealing with system operation and maintenance details and to provide users with a system that runs uninterruptedly at full performance. Self-management, according to IBM's autonomic computing initiative introduced in 2001 [16], has four aspects: self-configuration, self-optimization, self-healing and self-protection which are graphically presented in Figure 1.2. These four properties are discussed in the paragraphs that follow and they are summarized in Table 1.

Self-Configuring
Self-Healing
Self-Optimizing
Self-Protecting

Figure 1.2. The four self-management attributes [16]

1.2.1 Self-configuration

In short, self-configuration is dealing with automated system configuration by dynamically adapting to changes in its environment. This is achieved by applying high level policies that describe the desired outcome but not the way it is carried out.

As complexity and scale of today's computer systems continuously increases, installation and configuration of such systems becomes costlier with regard to time and effort. Another important side effect is the raise of errors introduced by the human factor. On the other hand, a self-configurable system can quickly adapt without requiring manual administrative actions, eliminating at the same time human caused faults.

1.2.2 Self-optimization

Self-optimization (or self-tuning) is concerned with optimizing and fine tuning the system in order to achieve peak performance at the minimum cost. A self-optimizing system will continuously be seeking ways to improve its operation by maximizing utilization of resources.

Tuning a computer system to maximize performance was never an easy issue. Indeed, it is becoming even more demanding when dealing with huge systems like databases and application servers where a significant number of parameters has to be tuned and the system's resources should be optimally utilized. In addition to that, in most cases, the cost factor is of major importance and hence optimization decisions should be very well-grounded. Self-optimizing systems can put a bold face on this issue by continuously monitoring, sometimes experimenting, and finally adjusting their own parameters to achieve better operating conditions. A simple, yet typical, self-optimization example is a system that automatically locates and installs updates.

1.2.3 Self-healing

One of the most important properties of autonomic computing is self-healing which aims at improving resilience to failures. The goal of strengthening resiliency is fulfilled by identifying, diagnosing and correcting problems caused by hardware or software failures. In its nicest form, self-healing can be performed proactively, that is, before the failure happens by predicting it.

The importance of the self-healing properties is apparent. Failures of computing systems have a negative business impact that is usually translated into money loss. Moreover, serious problems can take significant amount of time and effort to be located and fixed by human administrators and therefore increase the aforementioned impact. Self-healing computer systems can minimize the downtime either by identifying the problems and automatically apply fixes or by informing administrators to take action.

1.2.4 Self-protection

Last, self-protecting aims at providing correct information to authorized users when they need it through actions based on user's privileges and predefined policies. In other words, self-protecting refers to the ability of a computer system to fight against malicious or intrusive behaviour by first detecting it when it occurs and second acting autonomously to tackle with it, that is to prevent unauthorized access and use of resources and become immune to viruses or attacks such as a denial-of-service.

1.2.5 A Summary on Self-properties

Table 1 intends to summarize the discussion made above.

Attribute	**Feature/Responsibility**
Self-configuration	Automated system configuration by dynamically adapting to changes using high level policies
Self-optimization	Automated fine tuning to achieve peak performance at the minimum cost by maximizing utilization of resources
Self-healing	Improved resiliency by identifying, diagnosing and correcting problems caused by hardware or software failures
Self-protection	Automated protection against malicious or intrusive behaviour by detecting it and taking actions to ensure safe running conditions

Table 1. The four attributes of self-management

In order to build a fully self-managed system one should implement these four attributes. The typical approach is to differentiate between management and application functionality. Object oriented programming, at it is basic form, does not provide any means for applying self-management directly. A more advanced software entity is needed to facilitate the creation of manageable software and the most appropriate entity seems to be a “software component”. An introduction to software components and component models follows in section 1.3, whereas a detailed overview of existing models is given in section 1.5.1.

Still, components on their own can not provide all the required managing functionality for building complex systems. Typically, a “framework” is used which is responsible for the deployment, configuration, bootstrapping and management of the components. Management frameworks are discussed more thoroughly later in section 1.5.2.

1.3 Component Models

Software systems have existed for many years and continue to evolve at a very high pace, increasing in size and complexity, spanning over many different fields (many systems are multi-tiered and distributed) and requiring huge amounts of investments to manage them. As complexity in software systems increases, it is necessary to include mechanisms and approaches to manage and reduce cost. The concept of objects was introduced to fulfill this need. Objects provide a standard way to encapsulate and abstract, creating an easier, more general view of a system, thus simplifying it and breaking it into smaller pieces. The object oriented approach was a step forward in software engineering, providing useful concepts like modularity, encapsulation, or unified functions and data.

However, objects do not provide the levels of abstraction and simplification that modern complex systems may require. Objects are compiler bound, not executable on a stand-alone basis, and are not modifiable at run-time [22], representing a fixed entity directly mappable into code most of the times. Objects, compared to software components, are more fine-grained, and tend to be linked to one concept, while software components are more conceptual, possibly encapsulating many atomic concepts together, thus providing higher abstraction. Actually, software components can be built of objects, but do not have to. Components exist in basically all fields. In construction, when a house is being built, not all the needed components are created from scratch; bricks, pipes or windows are acquired from different sources and then put together. The same approach can be taken in software engineering, albeit a little bit differently, using software components.

Software components greatly help to reduce complexity by separating implementation and interface by making the software architecture explicit [6]. Doing this, programmers are isolated from the real complexity level of the system, letting them focus on their areas of expertise, and later on combining the different components together forming the complete systems from clearly differentiated modules, individually generated, and possibly individually engineered. Component engineered systems, with a clear separation between implementation and design, are used to alleviate the burden of both management and implementation. Well designed components have the ability of being highly reusable, a desirable characteristic when trying to improve performance and cut costs and time. Systems built of many different components can be modified and updated in an easier way by focusing on a component by itself, knowing that if the "front end" of the component is respected, even if the core of the component is completely redesigned and rebuilt, the system should still function properly once the new component is integrated.

From the autonomic computing perspective, a component model should enjoy a set of properties in order to be applicable on a self-manageable system. These requirements are:

a) to support separation between interface and implementation
b) to be reflective
c) to support hierarchical component composition

Extensibility is also a desirable property but it is not considered as a prerequisite. In fact, a reflective and hierarchical component model is expected to be extensible.

Components are defined by their component model, which describes with a set of well-defined directives, how components are built and represented, as well as how they are combined together (composite components), managed, deployed or any other information that may be relevant to the component in question. There exist many different component models at many levels, from component models that do not offer much difference from simpler object oriented paradigms, to advanced models that include support for extension and adaptation like the Fractal Component Model [6]. A deeper review and a categorization of the available component models is given later in section 1.5.1.

1.4 Clusters & Wide-Area Networks

A cluster is a group of independent or loosely coupled computers combined together through software and network to form a unified system. Typically, clusters are deployed to provide high availability, scalability and computational power in a cost effective way [23] [24]. Therefore a usual classification of clusters with regard to usage divides them into two basic categories: High Availability clusters whose main purpose is to improve the availability of a system, and High

Performance Computing clusters which are mainly used in scientific computing and are closely related to parallel computation [25]. Last, there is Grid computing which can be considered as an extension of clustering. The key differences between grids and clusters are, first, that grids have a more heterogeneous nature and, second, that grid computers do not fully trust each other. The latter means that grids give the impression of a service utility and not of a unified computer.

As mentioned above clusters consist of interconnected computers. Usually, they rely on a Local Area Connection (LAN) in order to achieve high speed communication. There are cases though that LAN existence cannot be taken for granted because either the resources are geographically dispersed by nature, or they are intentionally distributed to form a fault tolerant architecture [26]. Under these circumstances computers will be connected through a Wide-Area Network (WAN). However WAN usage introduces new issues that should be carefully considered. Firstly, regarding communication, compared to LANs, the bandwidth is usually reduced, but most importantly, there is an extra overhead due to latency. Obviously, communication becomes also less reliable, this fact becoming the major disadvantage. One might also add security in the list as it is always a crucial matter for consideration.

Wide-area networks, however, are increasingly attractive since they are closer to the nature of today's computing systems. Even if it is feasible to centrally collect all the resources of a modern computing system, the attempt to aggregate them will be extremely expensive and the result of dubious success in terms of flexibility, scalability, and reliability. In fact, all the difficulties WAN introduces and that were mentioned above make the goal of achieving clustering in a wide-area network highly challenging.

1.5 Related Work

A lot of work has been conducted lately on component models and management frameworks. In the following section an analysis on the current state of the art is attempted in both component models and management frameworks.

1.5.1 Overview of Existing Component Models and Categorization

In recent years, the component approach in software engineering has become extremely popular. Initially, simple component models like Sun's JavaBeans and Microsoft's Component Object Model (COM) were introduced targeting mostly code reuse. Later on, the evolution of computing systems leading to the introduction of distributed programming forced the emergence of models such as Sun's Enterprise JavaBeans (EJB), Microsoft's .NET and Object Management Group's (OMG) CORBA Component Model (CCM). All the above models are usually referred to as standard or business component models. These models despite their high penetration – especially inside the industry – have significant shortcomings when the focus is put on high performance or autonomic computing. Thus, targeting at high performance computing, the Common Component Architecture (CCA) was introduced whereas for autonomic computing there are models like Fractal, Sofa and K-Components. Last, there is an interesting work still in progress by CoreGRID: the Grid Component Model (GCM), which is a component model designed specifically for Grids.

As already mentioned, component based development has met gross success inside the industry. It is quite common nowadays for a company to create its own, proprietary component model as a base for their software. Typical examples of this kind are Mozilla's XPCOM [46], Philips' Koala [47], Gnome component model [48] and KParts (KDE's component model) [49]. However these models are not in the scope of this thesis project and hence they are not discussed any further.

1.5.1.1 Sun's Enterprise JavaBeans

Enterprise JavaBeans (EJB) is a component model designed by Sun Microsystems aimed at constructing server-side, enterprise-class applications [27]. Enterprise beans are server-side components written exclusively in the Java programming language, which encapsulate the business logic of an application and are deployed and run in an EJB container [28]. Each single bean is a component, and as such, is reusable and shareable, and can be accessed by clients remotely.

Each EJB has to implement two interfaces and two classes. The “remote” interface represents the methods that are accessible from the outside and thus provides access to the bean itself. This interface represents the business methods. The second interface is the “home” interface, which is in charge of defining the methods that represent the life cycle of the bean: how to create, delete and find beans. The two classes that must be implemented are the Bean Class, which actually implements the business methods defined in the “remote” interface with some of the methods defined in the “home” interface, and the primary key, a simple class that provides access to the persistent storage (database).

There are three types of beans that can be created. Session beans model a workflow and represent a single task [29]. Sessions beans are characterized for not having any persistent state, opposite to the second type of EJBs, called entity beans, which do have persistent state. Entity beans are normally directly related to relational databases and are associated with a row in a database. The last type of bean is the message-driven bean, which is stateless and resides on the server. This type of bean basically acts as a simple message listener [28].

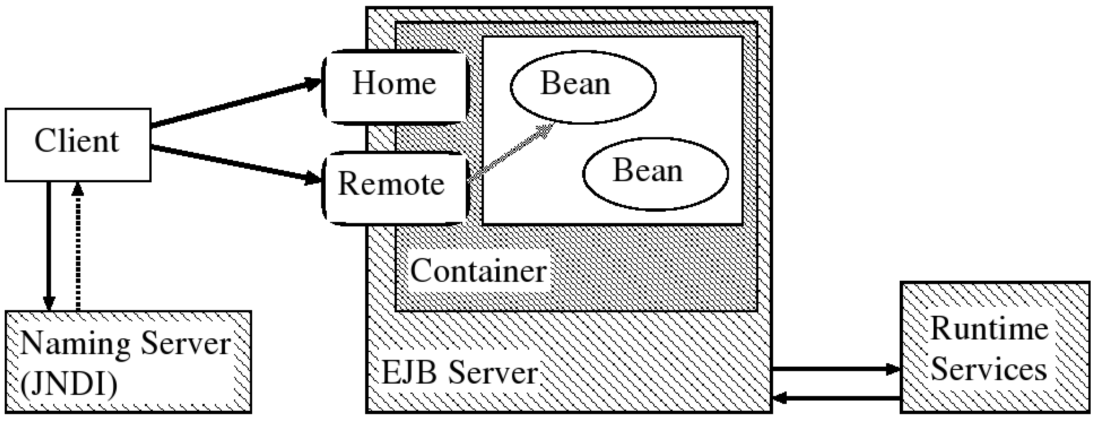


Figure 1.3. EJB architecture [30]

In Figure 1.3 a general overview of the EJB architecture can be seen; the container with the remote and home interfaces and the contained beans.

1.5.1.2 OMG's CORBA Component Model

The CORBA Component Model (CCM) is a specification for building scalable, server-side, multiple-language, transactional, and secure enterprise applications. It provides a consistent component architecture framework for creating distributed n-tier middleware [3]. The CCM specification dictates the complete software development process, starting with the specifications and going all the way to deployment and configuration. CORBA Components in the CCM are defined using an Interface Definition Language (IDL) based on the C++ language type system. This type system used by CORBA has mappings to many existing different type systems, which enables CORBA's ability to interoperate between different software and hardware environments.

The CCM uses an execution scheme based on the component-container pattern [33] which provides the desirable separation between business (components) and non-functional concerns (containers). CORBA's communication scheme is based on making requests to the Object Request Broker (ORB) which potentially can communicate with other ORBs in order to forward the request to the correct method in the server process. Communication between CORBA artifacts is done through four different types of ports as specified in the CCM. These ports give the required interfaces that a component exposes to clients and define the following capabilities of a component:

1. Facets: interfaces provided by a component. Can be called synchronously or asynchronously.
2. Receptacles: components obviously can call other components to delegate some task and in order to do this, components must obtain the object reference to an instance of the component they want to delegate to. The CCM calls these references receptacles.
3. Event sinks and sources: components in the CCM can as well interact with other component by monitoring events asynchronously. A component declares its interest to publish or subscribe to events by specifying event sources and event sinks in its definition [33] enabling it to interact with other components.
4. Attributes: can be used to configure the components.

The CCM standardizes component life cycle management by introducing the “home” IDL element thus specifying a lifecycle management strategy to the Model.

In Figure 1.4, a general overview of the CCM and how two containers would look like with their interfaces and the underlying Object Request Broker (ORB) can be seen.

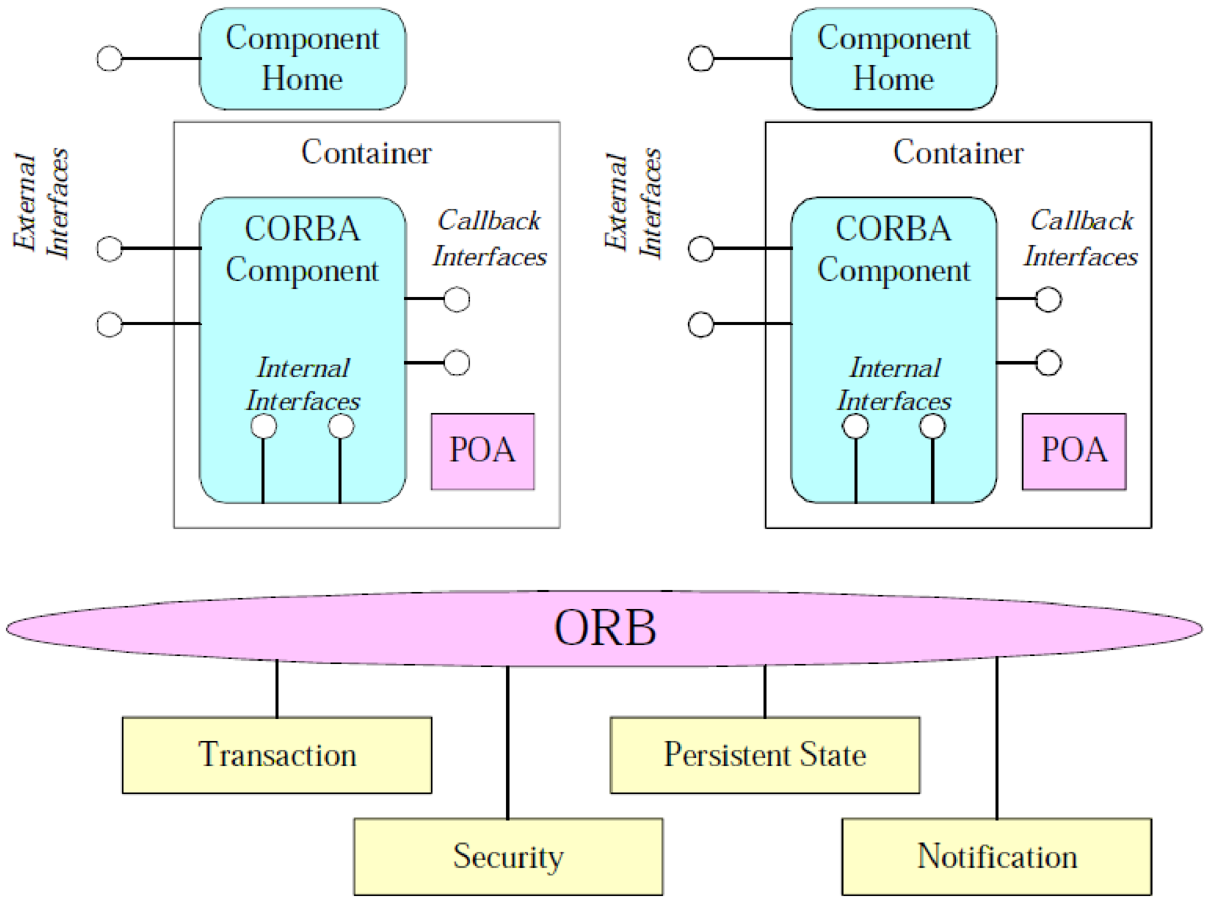


Figure 1.4. The CORBA component model (CCM) [45]

1.5.1.3 Microsoft's .NET

The Common Object Model (COM) is a specification and implementation developed by Microsoft to make interaction between components through a framework possible, having reusability and interoperability between components as its main purpose. To address the lack of support for distributed systems, DCOM (Distributed COM) was developed. DCOM allows for components to interact through networked environments as long as DCOM is available in the given environment (Windows based).

DCOM provides network transparency, enabling calls to methods possibly in other nodes of the network without the calling process knowing if the method is local or, potentially, thousands of kilometres away. DCOM acts as the intermediate layer between processes that are shielded from each other for security reasons. DCOM intercepts the made call and forwards it to the correct node, thus leaving the calling process completely unaware of the networked environment, making the call look like an inter-process interaction. As it can be seen in Figure 1.5, the DCOM architecture allows for complete network transparency as far as the calling process is concerned. The client just makes the call and the DCOM platform transparently handles the call to the recipient, possibly over the network.

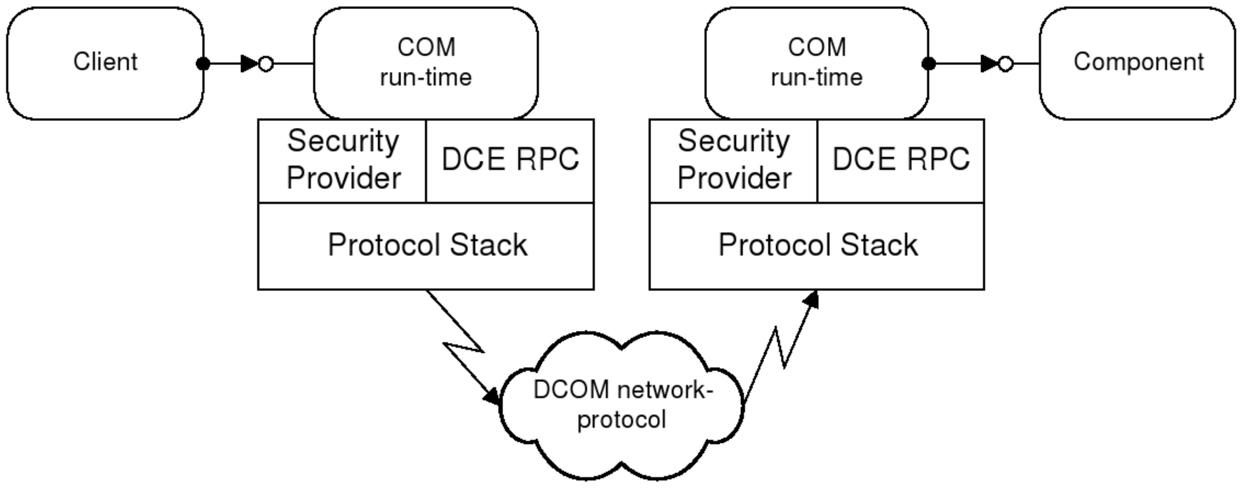


Figure 1.5. DCOM architecture

DCOM is not a very flexible model, having problems with firewalls, only working under Windows environments, and not using the latest, more webservices-oriented technologies like SOAP or TCP [34]. To solve these issues, Microsoft introduced .NET, a platform that is designed with modern distributed systems in mind and addresses their requirements.

1.5.1.4 Common Component Architecture (CCA)

Common Component Architecture (CCA) was introduced in the late 90's by the Common Component Architecture Forum [31], a group of researchers from mainly U.S.-governed laboratories and academic institutions. Their initiative targeted at defining a component architecture for high performance computing, that is a system that sets the rules of linking components together in order to create scientific – mostly parallel – applications. At the same time language independence is promised by allowing components written in different languages to interoperate.

The basic elements of the model are the components, the port and the framework:

- Components are software entities that can be seen as black boxes exposing only well defined interfaces.

- Ports are communication end points that determine the way components interact with each other. By adapting the “provides/uses” interface exchange mechanism (like CORBA 3.0 specification) components may provide or use ports. “Provide” means implementing the functionality described by the port, whereas “use” means making calls to a port offered by another component [32].
- A Framework is required to serve as the container of the components.

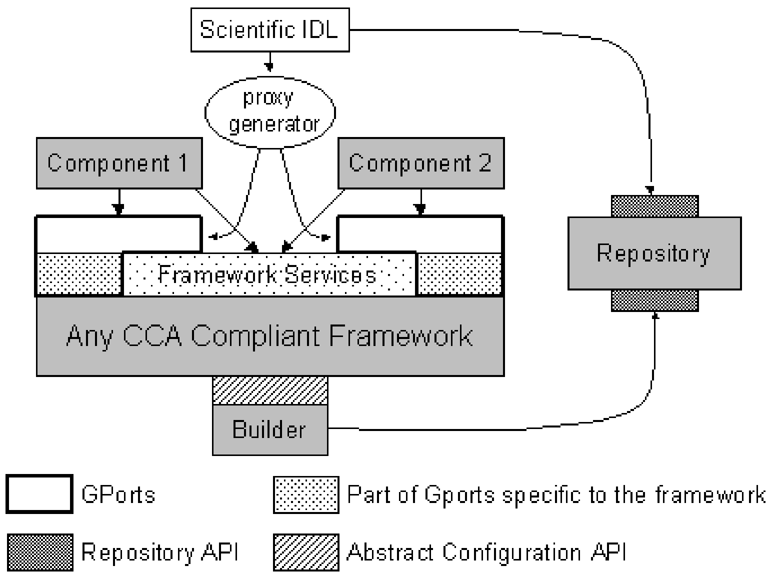


Figure 1.6. Relationships between the various CCA elements [31]

A graphical view of CCA, as defined in [31], is given in Figure 1.6.

The components interact with each other and with the framework using CCA interfaces. Every component of the system defines its ports using the Scientific Interface Definition Language (Scientific-IDL) and all these definitions are stored in a repository using an API called CCA Repository API. Component stubs, called GPorts, are also generated. The services provided by the framework, known as CCA Framework Services, can be used by the components directly through an interface. Lastly, the CCA Configuration API serves as the glue between the builder and the possibly different framework implementations.

1.5.1.5 Fractal Component Model

"Fractal is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces" [41]. The Fractal component model was designed with extensibility and adaptability in mind, which allows effortless and dynamic introduction of control mechanisms for components and eases the burden on programmers when it comes to making trade-offs between configurability and performance [6]. This is done by allowing reflectivity to take place within Fractal components in a flexible way, allowing the possibility to extend and adapt the reflective capabilities to meet a given set of requirements and constraints. Fractal components are equipped with an unrestricted set of control capabilities. The Fractal component model is designed in an attempt to meet any requirements by being flexible and extensible, making it applicable to all types of software, from embedded to large-scale information systems [6].

The objective of the Fractal model is to have a component model that not only implements large systems, but that also deploys, monitors and dynamically reconfigures them, being involved in the complete lifecycle of a system. In Figure 1.7 a basic component based application with the corresponding controllers (binding controller, life cycle controller...) and interfaces can be seen.

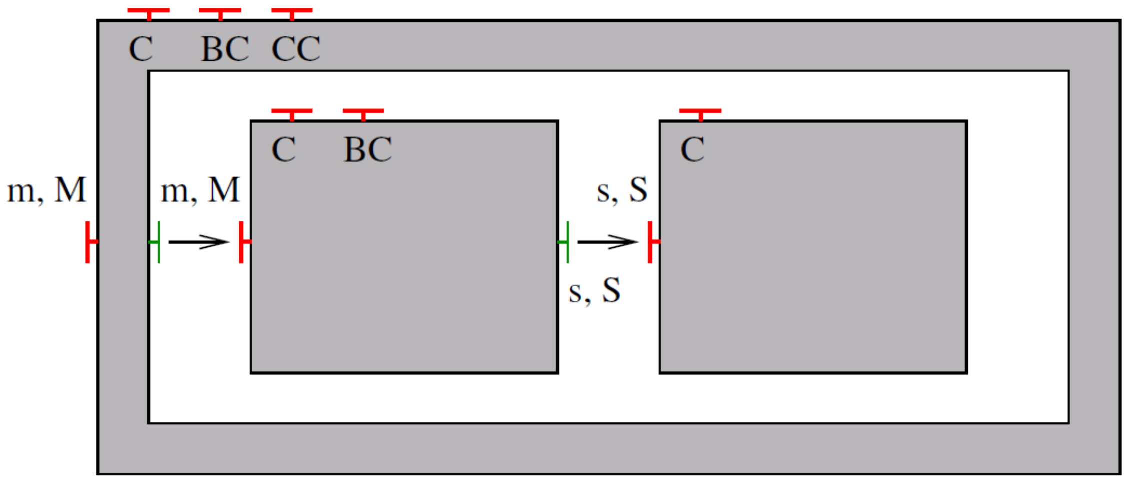


Figure 1.7. A sample component based application [6]

The Fractal component model is a key part of this project and will be discussed in more detail in the following chapter.

1.5.1.6 SOFA Component Model

SOFA stands for Software Appliances and its a project aiming at providing a platform for software components [4]. The project started at Charles University in Prague by the Distributed Systems Research Group [36] and it has later gained support by ObjectWeb. The motivation was to enable development of applications that consist of a set of *"dynamically hierarchical updatable components"*. Main features as presented in [4] are:

a) dynamic downloading of component

b) dynamic update of components at runtime (known as DCUP)

c) hierarchical top-down design

d) distributed deployment of components

e) support for versioning

From its perception, SOFA (version 1.0) was very similar to Fractal. In fact, after 2004 it is considered as a Fractal implementation since it became Fractal compliant on level 2 [37]. According to [4], [5] & [38] SOFA applications are viewed as hierarchically nested components with the notion of primitive and composite being present. A component is divided in two parts: the frame and the architecture. The frame can be seen as a black box where "provides/requires" interfaces are defined using the Component Definition Language (CDL), an OMG IDL based language. The architecture is a gray box view of the (composite) component and it describes its structure by providing information about subcomponents and how they are interconnected trough interface ties (connectors).

Currently there is work in progress on SOFA 2.0 [39] which aims at purifying and extending the existing version.

1.5.1.7 K-Component Model

The K-Component model was originally presented by Jim Dowling on his PhD thesis [34] as a proposal to achieve self-adaptation for autonomic distributed systems in a decentralized fashion. The basic idea behind the model is again Fractal which lends the notion of primitive and composite components.

In more detail, according to [9], [10] & [34], components are defined using an interface definition language called K-IDL (a subset of IDL-3). Dependencies between components are explicitly described through the "provides/uses" interfaces, while base-level adaptation events are provided thanks to "emits" and "consumes" interfaces. A K-Component can be perceived as a runtime with a single address space containing components. Their dependencies inside or outside the runtime are managed using connectors. Components and connectors are reified using architectural reflection as an architecture meta model (AMM). It should be noted here that each K-Component runtime has a partial knowledge of the system; it only knows about the internal components and their connections.

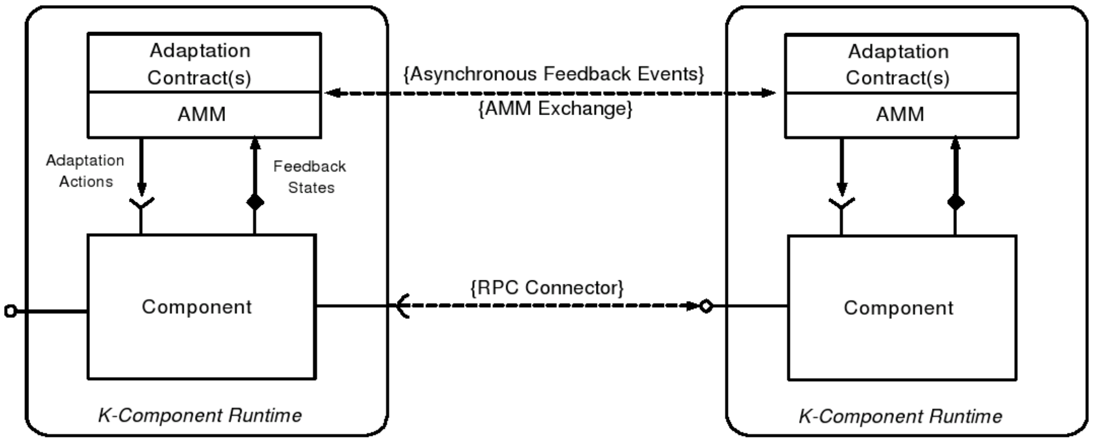


Figure 1.8. Connected components through different K-Component runtimes [9]

An analysis of the model's adaptation capabilities is given on section 1.5.3.1 where K-Components are presented as a self-management framework. Figure 1.8 shows two interconnected components in separate K-Component runtimes.

1.5.1.8 CoreGRID's Grid Component Model

Grid Component Model (GCM) is a specification of a component model for Grids as defined by the Virtual Institute. There is no doubt that Grids have special characteristics, the most typical being heterogeneity. While the goal of a Grid is to provide virtualization of resources, that is to guarantee transparent access to resources hiding heterogeneity and possible dispersion, new challenges show up that common component models cannot adequately face. The model proposed by the Virtual

Institute [43] is required to:

d) be reflective

e) have hierarchical structure

f) be extensible

g) provide support for adaptivity

h) be interoperable

i) allow lightweight implementations

j) have well defined semantics

In the GCM specification, Fractal is taken as a reference model. CoreGRID's [44] proposed model is defined as an extension to Fractal with extra features added in order to target the Grid infrastructure [7]. Fractal has been chosen because it is *"well-defined, hierarchical and extensible"*. Moreover by adopting Fractal, language independence is achieved since there are implementations for various programming languages.

1.5.2 A Summary on Component Models

The overview presented above has brought to light most of the motivations, features and limitations of the popular component models. This section attempts to sum them up and to provide a comparison according to their self-management potentials.

As stated in section 1.3, a component model, in order to be self-management friendly, should enjoy basically three properties: separation between implementation and interface, reflection and hierarchy.

Starting with the standard component models, Sun's Enterprise JavaBeans and Microsoft's DCOM have a major disadvantage that is the lack of contracts between the components. There are no "provides/requires" interfaces as for example in OMG's CCM or Microsoft's .NET. However all of them suffer from the absence of a formal, explicit architecture description and therefore they are not reflective. Reflection is one of the cornerstone properties for addressing autonomic computing.

The Common Component Architecture Forum realized that the usage of Architecture Description Languages (ADL) can help building more reflective systems that can be easier maintained. Nevertheless, despite the benefits of having an explicit software architecture, ADLs do not provide all the necessary facilities to attain self-management. There is another important property a component model should enjoy in order to facilitate self-management and this is the capability for hierarchical component composition. Neither the business component models mentioned above or CCA are hierarchical and therefore they can provide limited support for component introspection and adaptation.

On the contrary, the Fractal component model enjoys all these properties. It is both reflective and hierarchical preserving at the same time the beneficial characteristics of the basic models, e.g. separation between interface and implementation, and explicit architecture description. Its disadvantage is the overhead introduced due to its advanced features.

SOFA can be considered as a Fractal implementation whereas K-Components is a model that shares Fractal's ideas. A drawback of K-Components is their concrete implementation in C++ which puts an obvious constraint. Lastly, the Grid component model is a Grid specific solution based on the Fractal model. Table 2 summarizes the component models and provides a comparison according to their self-management potentials.

	Desirable Properties for Autonomic Computing		
Component Model	**Interface Separation**	**Reflection**	**Hierarchy**
EJB	✗	✗	✗
DCOM	✗	✗	✗
CORBA	✓	✗	✗
.NET	✓	✗	✗
CCA	✓	✓	✗
Fractal	✓	✓	✓
SOFA	✓	✓	✓
K-Components	✓	✓	✓
Grid Component Model	✓	✓	✓

Table 2. A summary/comparison of component models according to their self-management potentials

1.5.3 Management Frameworks

As the trend towards self-managing systems is emerging, there are lots of proposed frameworks available promising to facilitate administrators' life. In the paragraphs that follow five different approaches are presented. Firstly, four frameworks with self-management capabilities are covered: A framework based on K-Components that uses Collaborative Reinforcement Learning (CRL) as a decentralized coordination model [9] [10] [34], a Web Services approach to autonomic computing [11], a Multi-Agent approach [12] and finally the JADE component management framework [8] [53] with the Fractal component model being its backbone [8]. The last framework that is considered is ProActive [69] [70]. ProActive also builds on the Fractal component model but it offers manual management rather than self-management features; nevertheless it is an interesting approach with high penetration in both market and research areas.

1.5.3.1 K-Components and Collaborative Reinforcement Learning

This framework presented in [9], [10] and [34] combines K-Components (described above in 1.5.1.7) with a decentralised coordination model called Collaborative Reinforcement Learning to form a self-adaptive component model. K-Components enable individual components to adapt to changing environment whereas CRL provides components with the mechanism to *"collectively adapt their behaviour to establish and maintain system-wide properties"* in the changing environment. The key feature that this framework promises is decentralised control.

It has been already discussed that in the K-Component model there is no explicit system wide knowledge. Instead, system wide Architecture Meta Model (AMM) is partitioned among the K-Component runtimes and each of them manages its local software architecture. Autonomous behaviour is added to the system by associating reflective programs – known as "adaptation contracts" – to the components which run on the runtime's AMM (Figure 1.8). Adaptation contracts are written in a declarative language known as Adaptation Contract Description Language (ACDL). Developers use ACDL to define actions to events or component's and connector's states using if-then rules or the event-condition-action model (ECA). However, both if-then rules and the ECA model are not satisfactory solutions for large, complex systems because there can be an enormous number of events/actions making it extremely hard for developers to predict and program every single detail. To deal with this issue K-Components are provided with self-learning capabilities. Their self-adaptive behaviour is assisted by a mechanism called CRL.

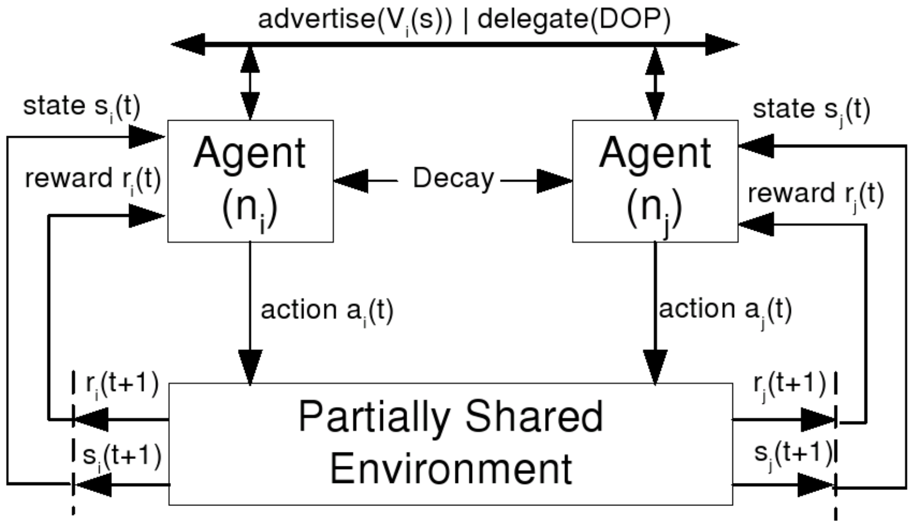


Figure 1.9. The Collaborative Reinforcement Learning (CRL) model [9]

CRL is a distributed version of Reinforcement Learning (RL). RL refers to the machine learning problem where an agent attempts to gradually optimize its behaviour by perceiving the environment and applying actions in a trial-error fashion. The result of the action is then reinforced which in turn leads to the updating of the agent's action-value policy. Reinforcement learning algorithms target at maximizing cumulative reward for the agent over the problem's lifecycle [51]. In CRL there is no system wide knowledge as well. On the contrary, knowledge is distributed among agents that only interact with their neighbours. Indeed agents are advertising their results to their neighbouring agents so that each forms a table of discrete optimisation problem (DOP) solutions. Figure 1.9 graphically presents how the CRL model works.

In conclusion, the K-Components and CRL combination offers a flexible solution for self-management. K-Components provide a self-adaptive component model while CRL models self-management properties at the system level as system optimisation problems. The technique has a long term objective sacrificing possible poor performance in the short run [9].

1.5.3.2 Autonomic Web Services

Sherif A. Gurguis and Amir Zeid propose a solution for achieving self-management using web services [11]. Their approach currently considers only self-healing but they are planing to extend it in order to cover the remaining three self-properties as well. The following paragraphs attempt to summarize the idea proposed in [11].

The four attributes of self-management which are covered also in 1.2 are self-configuration, self-optimization, self-healing and self-protection. According to [1] every component is divided into two parts: a *"managed element"* and an *"autonomic manager"* which features the self-management functionality. Autonomic managers consist of four parts as shown in Figure 1.10. "Monitor" is responsible for keeping an eye on the managed element and gathering information about its state. "Analyser" is used to analyse the collected information to determine the elements' condition and to suggest actions in case they are needed. "Planner" defines the actions that should be taken according to predefined policies. "Executive" is responsible for *"dispatching the proposed actions to the elements"*.

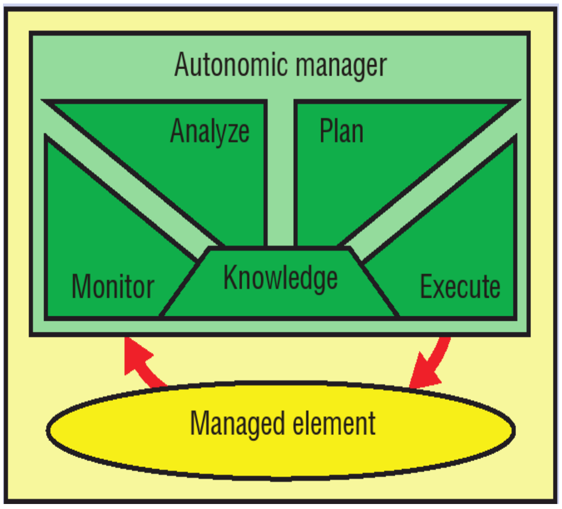


Figure 1.10. Autonomic Computing vision [1]

The classic triangle that describes the roles and operations in a web-service lifecycle is shown in Figure 1.11. There are three discreet roles (requester, provider, registry) and four possible operations (publish, find, bind and use).

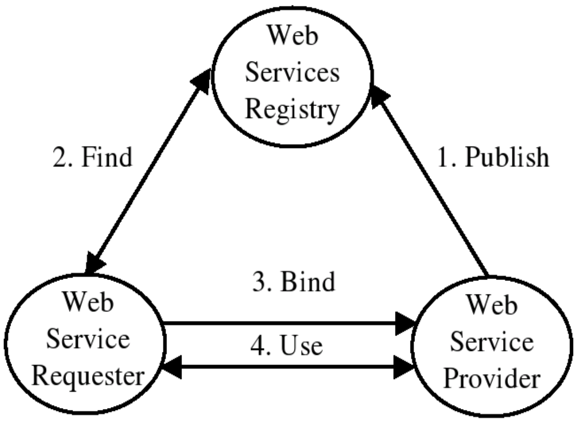


Figure 1.11. Web Services roles and operations

The proposed solution distinguishes the functional aspects of the application from the self-management functionality by defining two groups of web services. System's functionality is provided by *"Functional Web Services"* whereas *"Autonomic Web Services"* are responsible for realizing the autonomic behaviour. Therefore, the vision of the proposal is to have functional web services that through the Internet, manage to locate and use autonomic web services (Figure 1.12).

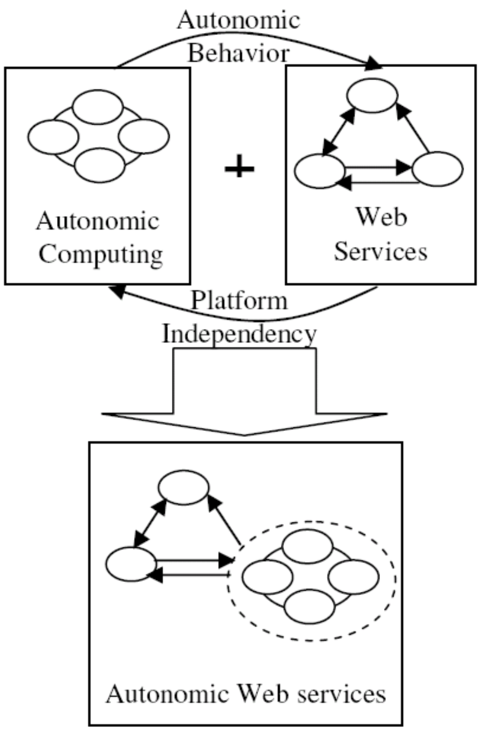


Figure 1.12. The concept of Autonomic Web Services [11]

Putting it all together, an autonomic web service can be assigned to each self-property forming the quartet: Self-Configuring web service, Self-Healing web service, Self-Optimizing web service and Self-Protecting web service. Zooming into an autonomic web service there is a MAPE-cycle consisting of four collaborating web services (Figure 1.13).

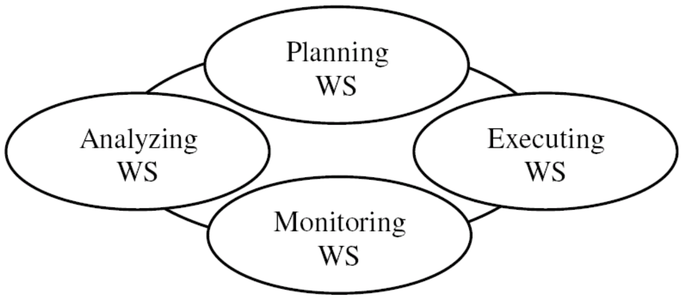


Figure 1.13. The MAPE-cycle of Autonomic Web Services [11]

The initial phase is information collection through monitoring. Analysis is achieved by using a Problem Determination Mechanism (PDM) to automate the procedure. However different applications can produce different logs. To be able to use a common PDM, logs need to have specific syntax and semantics. Thus, Common Base Events (CBE) are used to define log messages in an XML based format. A CBE includes information about the component which reports the event, the influenced component and the cause. CBE classifies the events into eleven predefined and one custom causes, also known as situations.

On the self-healing web service paradigm analysis of logs by the Analyzing WS is done using a Diagnosis Engine that searches for patterns inside the logs and suggests actions according to an XML-based Symptom Database. The planning WS then uses a Rule Engine to determine the right action that satisfies the predefined policy stored in a Policy Database. Finally, the Executing WS is responsible for applying the directed action. Synchronization between the different web services of the self-healing MAPE-cycle is achieved by implementing a Notification Web Service. Each web service subscribes to the events that is interested in.

Concluding, the approach presented above claims to be highly adaptable targeting the fact that there is still a high percentage of legacy systems. Under these circumstances, wrapping legacy code to web services and providing self-management capabilities through Internet based protocols by using autonomic web services is a possibly feasible solution. However, usage of web services is not always the most appropriate solution in terms of performance or security.

1.5.3.3 Multi-Agent Systems Approach to Autonomic Computing

A Multi-Agent System (MAS) approach to self-management is introduced by Tesauro et al. in “A Multi-Agent Systems Approach to Autonomic Computing” [12]. Autonomous agents are self-contained and capable of potentially making independent decisions and taking actions to predefined goals based on the environment they perceive. This same approach can be transposed into large distributed systems, breaking the systems into different modules that can be mapped into autonomous, independent agents in charge of managing themselves and interacting with the other agents composing the system. Using the autonomous agent paradigm, Tesauro et al. devised a software architecture called Unity, which aims at achieving *“self-management of a distributed computing system via interactions amongst a population of autonomous agents called autonomic elements”*.

Unity is built of components implemented as autonomic elements or agents that are in charge of resources and of providing services to other agents or humans. All components in Unity are autonomous, from databases and servers to workload managers to sentinels and brokers. Each autonomic element is, by definition, in charge of its own behaviour like managing its own resources and actions, as well as forming and maintaining relationships with other agents in order to accomplish its own set of goals. And since all the agents that compose the systems follow the same behavioural patterns, dictated by its own goals, the whole system is autonomous by addition of its parts.

Unity supports multiple application environments that provide their own service to the overall system. Each of these environments is represented by an application manager, an element itself that takes care of managing the environment, talking to other environment and making decisions to meet the goals of the environment. Application managers are also responsible for predicting how changes of its environment will affect the environment's ability to reach its predefined goals and act accordingly. Resource arbiter elements are in charge of calculating optimal allocation of resources to application environments; registry elements enable elements to find other elements; policy repository elements allow humans to interact with the system manually through interfaces; and sentinel elements support interfaces that allow an element to monitor another element's state. Finally, Unity also provides a web administrator interface to observe and direct the system.

As a self-management framework, Unity provides self-configuration by means of "goal-driven self-assembly". Each autonomic element is not aware of its environment when it begins to execute and has only a defined target or goal. The self-assembly process consists of contacting the registry to locate other elements of use to achieve the goal and start a relationship with them, and then the element is registered in the registry so it can be contacted by other elements. Unity, for the moment, only provides self-healing properties for the policy repository element. Self-healing, in this case, is achieved by having the policy repository contact an existing cluster and replicate its data across the repository, thus guaranteeing that in case of failure, its data is available somewhere else. Self-optimization is also supported at some scale by Unity. The Unity framework uses the concept of utility to achieve self-optimization, each application environment has a service-level utility function (U_i), obtained from a policy repository, based on the service level provided ($\mathbf{S}_i$) and the current demand of the service ($\mathbf{D}_i$). The goal of the system is to optimize $\Sigma_i U_i(\mathbf{S}_i, \mathbf{D}_i)$ on a continuous basis to accommodate changes on demand of the given service.

The Unity MAS framework is a work-in-progress (as of [6]) prototype that will be expanded and improved, and possibly could provide real value to real world applications but much work will be needed since it has only been tested on a small scale system.

1.5.3.4 The JADE Component Management Framework

JADE – as described in [8], [53] and [54] – is the implementation, in Java, of a prototype component based architecture that describes autonomous repair management in distributed environments. JADE is a management framework built on top of the Fractal component model providing dynamically configurable environments resilient to failure through a repair management scheme, backed by replication, that even manages the management subsystem itself, thus providing fault-tolerance and self-healing in a completely transparent manner.

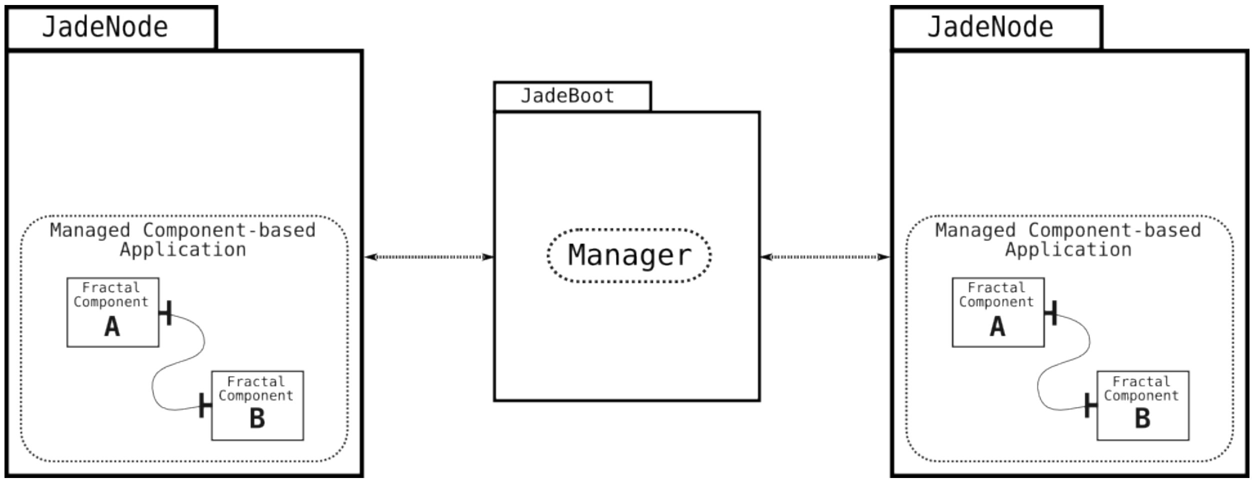


Figure 1.14. The JADE management framework

Figure 1.14 presents a simple view of the JADE framework, which basically consists of a JadeBoot that acts as the centralised manager (possibly replicated) of the whole system, including management of nodes (JadeNode), self-repair and other aspects of the framework. Connected to JadeBoot there can be any number of nodes (JadeNode) in which the centralized manager (JadeBoot) deploys and runs any type of JADE-enabled application architecturally described using the Fractal Architecture Description Language (Fractal ADL). The JADE management framework is discussed in detail in Chapter 2 of this document as it is the main focus of this thesis.

1.5.3.5 ProActive

ProActive itself is not a self-management framework like the other frameworks described in this subsection (1.5.3), but a solution with manual management features. Its strong and modular background, however, should make it easy to introduce autonomous behaviour if needed.

ProActive basically takes advantage of the hierarchical approach to component programming offered by the Fractal component model. In fact, a component consists of one or more medium-grained entities known as active objects, which form together an independent component (Figure 1.15). ProActive is written with parallel, distributed, and concurrent computing in mind and implemented as a GRID Java library. The ProActive library, which permits to minimize the complexity of programming applications that are fully distributed, either over LANs, clusters, P2P networks or even the Internet, is based on the concept of active objects; which are a uniform way to encapsulate [69]:

- a remotely accessible object
- a thread as an asynchronous activity
- an actor with its own script
- a server of incoming requests
- a mobile and secure agent

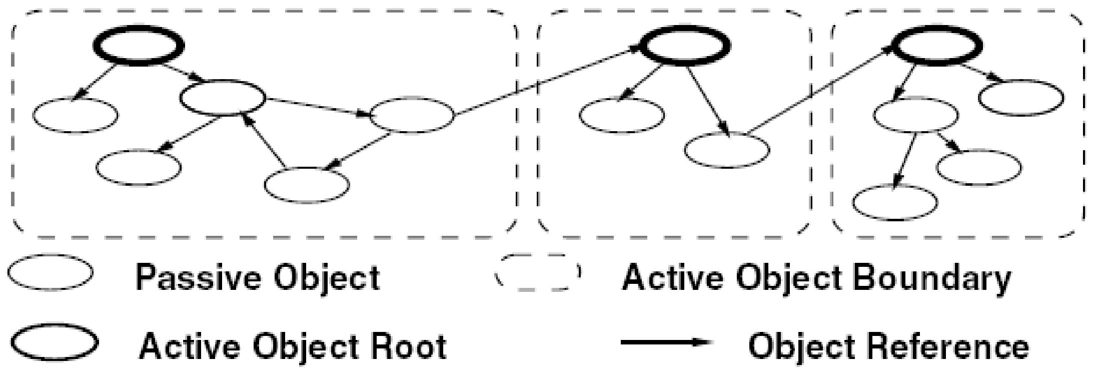


Figure 1.15. A typical object graph with active objects [70]

The ProActive libraries provide an architecture that permits full interoperability with the most predominant technologies available, such as web services, HTTP based transport, Globus, Sun Grid Engine and more. ProActive is composed exclusively of classic Java classes thus making it more practical, and requires no changes to the Java Virtual Machine (JVM), as well as no preprocessing or compiler modification. Writing code for ProActive is done by the programmer just as normal Java code would be written. "*Based on a simple Meta-Object Protocol, the library is itself extensible, making the system open for adaptations and optimizations. ProActive currently uses the RMI Java standard library as default portable transport layer*" [69].

As regards configuration ProActive uses an Architecture Description Language (XML descriptor) for defining "use/provide" ports, contents and bindings of components. ADL is also used to define deployment details such as nodes and JVMs [70].

ProActive offers fault-tolerance through check-pointing and rollbacks, load-balancing, mobility and security only on a manual manner, having an administrator managing the system using the Interactive Control and Debugging of Distribution (IC2D) [69] GUI tool, but not autonomous behaviour.

1.5.4 A summary on Management Frameworks

The first framework presented is the combination of K-Components with the agent based Collaborative Reinforcement Learning (CRL). This framework features decentralised control since there is no explicit system wide knowledge. On the other hand, as it is stated in [9], it *"gives a poor payoff in the short-term in the anticipation of higher payoff in the long term"*. Therefore, it is not suitable for a category of systems that cannot tolerate sub-optimal behaviour, such as real time systems. Finally, K-Component's specific implementation in C++ can in some cases be considered as a disadvantage.

Autonomic Web Services are based on the clear distinction between the functional aspects and the self-management functionality of applications. The "Functional Web Services" use the Internet to locate and use the "Autonomic Web Services". This framework targets at wrapping legacy code to Web Services. Apparently it is a Web Services dependent solution inheriting its pros (e.g. platform independence) and cons (e.g. possible poor performance due to the "chatty" nature of Web Services). Moreover, the proposed framework seems to be dealing only with independent Web Services and not with complex systems that consist of multiple components.

The Multi-Agent Systems approach proposes the use of independent agents that manage themselves and interact with other agents of the system, forming therefore a decentralised architecture. It promises goal-driven self-assembly as an extension to self-configuration, where autonomic elements begin to execute having only predefined targets or goals but no knowledge of their environment. In addition to self-assembly, self-healing and self-optimization are provided; in practice though only self-healing is implemented.

The JADE framework is built on top of the Fractal component model to benefit from its reflective and hierarchical features. A system's architecture is described using the Fractal ADL and there is system wide knowledge and centralised control. The framework is fully implemented in Java but offers the option to wrap any legacy application. In theory it covers all the self-management properties but in practice mainly focuses on self-repair.

The ProActive framework is a special case which mainly focuses on Grid computing. It is also based on the Fractal component model but it does not include self-management capabilities. Instead, it eases manual administration of the system by providing fault-tolerance through check-pointing and rollbacks, load-balancing through migration and finally mobility.

Table 3 summarizes the most important characteristics of the management frameworks covered in the sections above.

	Characteristics			
Framework	**Self-management properties**	**CM/technology**	**Control**	**Target**
K-Components & CRL	all	Fractal based	decentralized	multi-component systems
Autonomic Web Services	self-healing	Web Services	decentralized	independent services
Multi-Agent	self-healing	Software Agents	decentralized	multi-agent systems
JADE	self-healing, self-configuration	Fractal	centralized	multi-component systems
ProActive	-	Fractal based	-	multi-component systems

Table 3. A summary/comparison of the management frameworks

1.5.5 Literature Survey Conclusions

The discussion above leads to the conclusion that the Fractal Component Model is the most suitable component model for building self-manageable systems. Separation between interface and implementation, reflection and hierarchy are the three most important properties a component model should enjoy in order to be the base of a self-management framework, and Fractal offers them both in an extensible and flexible model. The JADE framework fully adapts the Fractal model whereas K-Components are quite close to it by borrowing its ideas. ProActive is a complete, highly active and quite popular framework but it is focusing on Grid computing and provides no self-management features.

The rest of this thesis focuses on the JADE component management framework.

1.6 AIMS & OBJECTIVES

The aim of this thesis is to perform a study of the JADE component management framework and more specifically a study of its structure and architecture as well as of its capabilities and weaknesses. The assessment of the JADE framework will be approached from four main angles: JADE's design in terms of wide-area environment support; the programmability, or how and what is necessary to enable application to run under JADE; how architectural decisions influence the outcome; and the overhead imposed by using the platform.

1.7 METHODOLOGY & EXPECTED RESULTS

The objectives outlined in the previous section will be addressed using both literature study and implementation experimentation. The strategy that will be followed is, at first, to study the JADE component management framework and analyse its structure and architecture as well as how it addresses self management. This phase is expected to help identify the wide-area capabilities of the framework. At the second phase, sample applications will be implemented to evaluate the framework in terms of programmability and overhead. These sample applications will be a simple yet functional HTTP server and a more advanced and realistic Bank RMI application.

The programmability assessment will be divided in two parts: firstly the development and the deployment processes will be evaluated, focusing on time and effort required to create a JADE application as well as pointing out the knowledge and skills required to achieve it. Secondly, the two different approaches in architectural design will be analysed – wrapping legacy software and designing specifically for JADE – using the aforementioned Bank application.

Evaluating the overhead of the framework includes measuring the amount of code required to write a JADE application and the memory usage during runtime. Finally, a theoretical approach on overhead due to the underlying technologies used and the resources needed to run an application under JADE will be attempted based on the study of the framework.

By performing these individual assessments, first a general overview of the current state of the JADE platform is expected to be derived. The sample applications are expected to clarify the requirements in terms of programmability as well as show the extra overhead introduced by using the platform. Moreover, the two different versions of the Bank application are expected to show how the architecture design can influence the resulting JADE application as regards flexibility and extensibility. Finally, a coherent set of conclusions is expected to be drawn and possibly suggestions for improvements will be included.

1.8 LIMITATIONS

As with most projects of considerable magnitude, some limitations apply that restrict both the expected outcome of the project as well as the way it is carried out. There are two main limitations that shape this thesis project: time and incompleteness of the provided framework. As a Master's thesis project, time is limited, in this case to around six months, to complete all the necessary steps of the project. The second considerable limitation is the fact that the JADE component management framework is at a quite early stage and it does not yet provide all the desired functionality. More precisely, the self-repair module of the framework is missing completely, which will prevent a complete assessment of the framework's capabilities with respect to autonomic behaviour.

1.9 EXPECTED READERS' BACKGROUND

In order to be able to properly follow this document, it is recommended to have a basic knowledge of some concepts and technologies. In general, a basic understanding of object orientation and component based design would be helpful to assimilate the different component models and management frameworks that are introduced. Also, specific to some of the management frameworks, the concepts of web services and software agents could ease their understanding. In order to be able to understand the snippets of code that will be introduced both in the main part of the document and the appendices, knowledge of the Java programming language and some understanding of XML would be desirable.

1.10 Roadmap

The rest of this document is formed of three different sections mapped into chapters. Firstly (Chapter 2), the Jade component management framework will be discussed in detail, including explanations of the general architecture and concepts of the framework, the management mechanisms, its underlying technologies and how it operates. Secondly (Chapter 3), an assessment of the framework will be given focusing on Jade's dependency on a cluster topology, the requirements and procedures to adapting and designing an application for Jade and the overhead the framework introduces in terms of memory, amount of code and the underlying component model. Lastly (Chapter 4), a short set of conclusions and some notes on future work that could be carried on following the footsteps of this thesis are given.

Chapter 2

The JADE Component Management Framework

The Jade framework follows an architecture-based approach to realize self-manageable complex systems. Architecture description is causally connected to the managed system, which means that possible changes to system's configuration trigger an update of its description and vice versa.

Jade's purpose is to provide self-management in three aspects: self-repair, self-optimization, and self-protection. Self-properties were discussed in section 1.2. However, there have been four properties: self-configuration, self-healing, self-optimization and self-protection. In Jade's context self-repair is supposed to combine both self-healing and self-configuration properties. In practice so far, the main concern has been self-repair.

Therefore, in its abstract view the Jade framework can be seen as a composition of the following parts [53]:

- Autonomic managers (e.g. repair manager, optimization manager, protection manager) used to detect and react to events.
- Common services (e.g. node discovery, node allocation, deploying service, system representation) that provide basic functionality.
- The monitored system which consists of various interconnected components.

In the sections that follow a detailed description of Jade is attempted, starting from its architecture and wading into its two most important characteristics: the fractal component model and the management authority.

2.1 JADE's Architecture

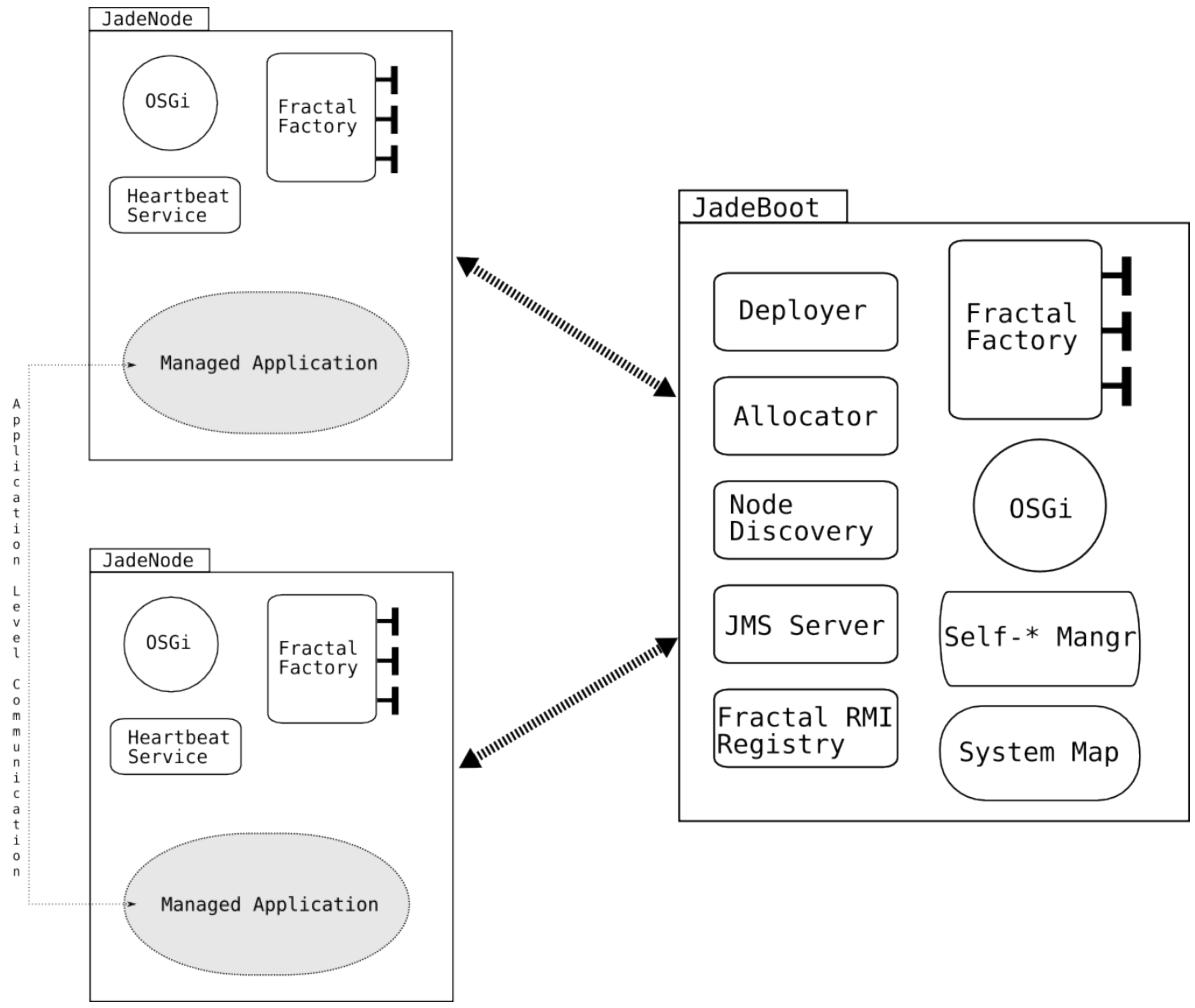


Figure 2.1. Jade framework's architecture

The Jade framework, seen in Figure 2.1, is composed of any number of JadeNodes and a JadeBoot component that behaves as the core of the system, which is itself another JadeNode. Each JadeNode (including JadeBoot) is a Fractal component containing three fundamental services [54]:

- An installer in charge of the installation of legacy software on the nodes. This installer is an implementation of the Open Services Gateway initiative (OSGi) [55] – a deployment framework specification – named Oscar [56].
- A Fractal factory taking care of the instantiation of Fractal components.
- A heartbeat service which is the main attribute of node failure detection. It acts by sending status messages to the Discovery service.

Specific to the JadeBoot node, it is worth noticing the System Map, that keeps an updated view of the whole managed system and enables the possibility of self-repair; the Node Discovery service, which receives nodes added to the system and the Deployer service which communicates to the right JadeNode what to install and where. Also specific to JadeBoot are the Fractal RMI registry [60] that keeps track of registered services, a Java Message Service (JMS) [57] server to allow the Java messaging system to be available, and a Java Naming and Directory Interface (JNDI) registry [58] used by Joram [59], an open source implementation of the JMS specification. The most important underlying technologies are briefly explained in the context of the JADE platform in section 2.4.

2.2 FRACTAL COMPONENT MODEL

The Fractal component model, like other component models (CCM, EJB, etc.) enables the abstraction between implementation and interface, but its strength resides in its ability to support extension and adaptation, which provides the possibility to a) painlessly introduce in a dynamic manner new control facilities to components, b) reduce the tradeoffs that need to be made between space consumption, configurability and performance, and c) reduce the complexity of operating frameworks in different environments. This is done by allowing components to have an open set of control capabilities, also known as non-fixed reflectivity.

The Fractal component model does not have a limited scope and can be applied from embedded to multi-tiered distributed systems. Clearly not all systems require the same functionality from a component model, and Fractal addresses this by not having a fixed specification that all components have to follow. The Fractal component model is *"an extensible system of relations between well-defined concepts and corresponding APIs that Fractal components may or may not implement"* [6]. This set of specifications is defined as increasing "levels of control":

1. Fractal components as runtime entities: no control capabilities to other components, object-like behaviour.
2. External introspection: Fractal components provide a standard interface that permits discovering their external interfaces.
3. Configuration level: at this level, Fractal components provide control interfaces to introspect

and modify their content. This inner content can be composed of subcomponents bound together.

The Fractal component model is very flexible in all its aspects making it suitable for almost any situation. This is achieved by making everything optional, from the control capabilities to the instantiation framework or the type system.

This section of the document is purely based on “The Fractal Component Model Specification” [6] and the “Developing with Fractal Tutorial” [52] and aims at giving an in depth introduction to the Fractal component model, but not a complete overview. For a more complete coverage of the model, see [6] and [52].

2.2.1 Interface Definition Language

At the lowest level, when there is no control capability, Fractal components are like simple objects. This type of components are called “base components” and they can only be used for calling methods on them almost like usual objects. The difference is that the operations are invoked on the components' interfaces. Component interfaces can be seen as access points to components which implement language interfaces [6].

The Fractal component model can achieve language independence and interoperability by adopting Interface Definition Language (IDL) for defining component interfaces. IDL usage however, according to the specification, is not mandatory although its advantages are obvious. The only drawback is the slight overhead introduced when invoking operations through stubs and skeletons, hence in cases where interoperability is not needed, interfaces can be written directly in the target language. Throughout this overview a pseudo-IDL (very similar to the Java language) will be used as defined in the Fractal Component Model specification [6]. Fractal implementations can use any IDL to describe interfaces such as OMG IDL or Java.

Invoking operations on components' interfaces is achieved by first locating the interface and then getting a reference to it. Figure 2.2 shows the IDL code used to realise the above functionality while the text that follows explains the logic behind it.

```
package org.objectweb.naming;

interface Name {
  NamingContext getNamingContext();
  byte[] encode() throws NamingException;
}

interface NamingContext {
  Name export(any o, any hints) throws NamingException;
  Name decode(byte[] b) throws NamingException;

interface Binder extends NamingContext {
  any bind(Name n, any hints) throws NamingException;
}
```

Figure 2.2. Naming API [6]

Names can have many forms, from Java interfaces to Interoperable Object References (IORs) and they always exist inside a specific naming context. The `Name` interface has two operations, `getNamingContext` and `encode`. The former returns the naming context inside which the name is meant to be, whereas the later provides encoding capability that is the ability to convert a name into a byte array. The `NamingContext` interface provides the way to create names from component interfaces using the `export` operation which takes as arguments a component interface and some optional hints. It can also produce a name from an encoded form by using the `decode` operation and passing to it the byte array of the encoded name. Lastly, there is a `Binder` interface which extends the `NamingContext` and provides a `bind` operation that attempts to create a binding to the interface of the given name and return a delegate (or proxy) to that interface (sometimes it can be the interface itself, depending on the location and the underlying technology).

2.2.2 Introspection

Introspection is defined as self-examination, a concept that is part of the Fractal component model. In particular, introspection refers to the capability of Fractal components to introspect their external features (or boundary).

Fractal components, depending on how they are defined, can be seen as a black-box or a white-box.

In the black-box perspective the internals of the components are not known to the outside and they only provide interaction through external interfaces (access points) of two kinds: client (required) that emit operation invocations and server (provided) that receive the invocations.

Introspection can be performed on both types of interfaces (client and server) with the following two language interfaces:

- Component introspection: this is provided to discover the external interfaces of a component, and is done by having components implement the `Component` interface (a component interface implementing `Component` must be named `component`). As it can be seen in Figure 2.3, there are two operations named `getFcInterfaces` and `getFcInterface` that return, respectively, a list of all available external interfaces and the interface for the given name if exists. The `getFcInterface` throws a `NoSuchInterfaceException` in case the requested component interface is not found. These two operations return references that can be cast and accessed directly to invoke operations on the component's server interfaces. The `getFcType` operation returns the type of the component.

```
package org.objectweb.fractal.api;
interface Component {
  any[] getFcInterfaces ();
  any getFcInterface (string itfName)
                throws NoSuchInterfaceException;
  Type getFcType ();
}
interface Type {
  boolean isFcSubTypeOf (Type t);
}
```

Figure 2.3. Component Introspection API [6]

- Interface introspection: the references provided by the above mentioned operations only provide access to the requested interfaces, but not the names of these interfaces. To provide interface introspection capabilities, Fractal components can ensure that the references obtained through `getFcInterfaces` and `getFcInterface` return can be cast to the `Interface` type. As it can be seen in Figure 2.4, there are four operations specified by this interface: `getFcItfName` which returns the name of the interface, `getFcItfType` that returns the type, `getFcItfOwner` to get the component interface of the component it belongs to and `isFcInternalItf`, that tests if the interface is internal or external.

```
package org.objectweb.fractal.api;

interface Interface {
   string getFcItfName();
   Type getFcItfType();
   Component getFcItfOwner();
   boolean isFcInternalItf();
}
```

Figure 2.4. Interface Introspection API [6]

2.2.3 Configuration

On a higher level of control capability Fractal components do not only provide the functionality to discover their external interfaces but can also offer the ability to control (introspect and reconfigure) their internal features through well defined interfaces. A Fractal component consists of two parts. The outer part is called controller or membrane whereas the inner part is the content (Figure 2.5). One of the key concepts of the Fractal model is recursion. Therefore, a component's content can be composed of other components called sub-components. Sub-components are under the control of their parents enforcing a form of hierarchy between components. The enclosing components that expose their content are called composite components. The ones that *"do not expose their content but have at least one control interface"* are called primitive components and the components that do not implement any control functionality are called base components.

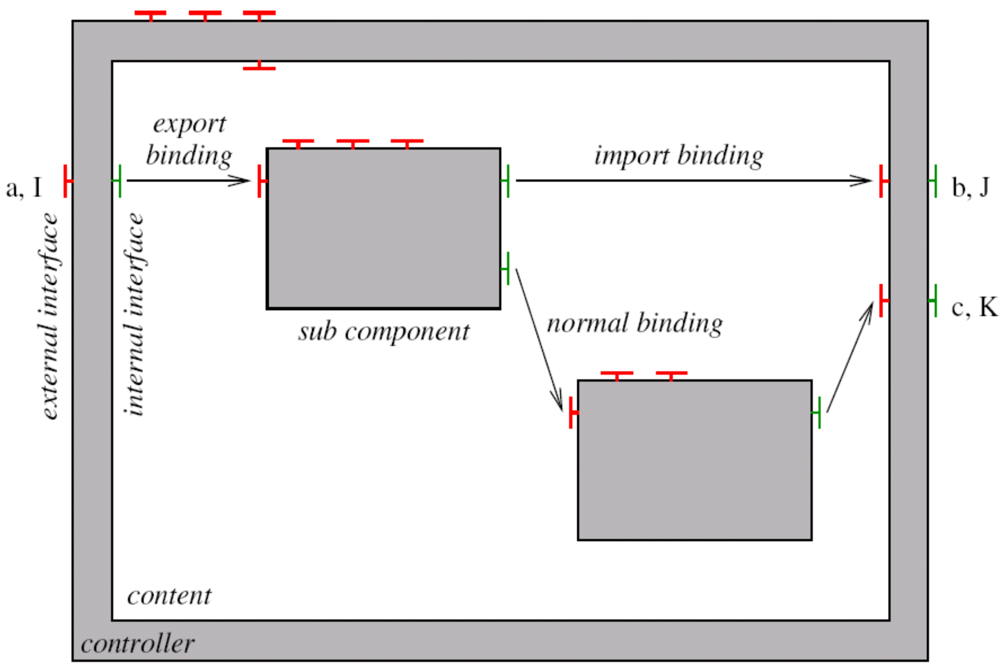


Figure 2.5. A Fractal component [6]

As it can also be noticed in Figure 2.5 there can be external and internal interfaces for each controller. Internal interfaces can be accessed only from within the component that is its sub-components. Interfaces can be also classified into functional and control interfaces where the former are responsible for providing the required functional aspects and the later for implementing control services (i.e. introspection, (re)configuration).

Another interesting feature of the Fractal model is component sharing (section 2.2.3.3) where a component can be shared between components. In such a case, the parent components can manage the shared one through the controller interfaces that it provides.

Components' interfaces are connected with each other through bindings. There are two general categories of bindings: primitive bindings and composite bindings. Primitive are the bindings between client and server interfaces inside a given address space. In Figure 2.5 there are three types of primitive bindings namely normal, export and import bindings. The other category regards bindings made between components residing in different address spaces using stubs, skeletons, adapters and similar patterns. This kind of binding is known as composite binding and is a combination of primitive bindings with binding components. Binding components in turn are normal Fractal components responsible for taking care of communication.

2.2.3.1 Attribute Controller

An attribute can be seen as a *"configurable property of a component"*. Each component can implement an `AttributeController` interface to handle access to this information.

```
package org.objectweb.fractal.api.control;

interface AttributeController {
}
```

Figure 2.6. Attribute Controller API [6]

The specification dictates having a sub interface of the aforementioned empty `AttributeController` where, for every attribute will be a getter and optionally a setter method.

2.2.3.2 Binding Controller

The `BindingController` interface is used from components to bind or unbind their client interfaces to other components. Primitive bindings can be achieved using the API shown in Figure 2.7.

```
package org.objectweb.fractal.api.control;

interface BindingController {
  string[] listFc();
  any lookupFc(string clientItfName)
      throws NoSuchInterfaceException;
  void bindFc(string clientItfName, any serverItf)
      throws NoSuchInterfaceException,
                 IllegalBindingException,
                 IllegalLifeCycleException;
  void unbindFc(string clientItfName)
      throws NoSuchInterfaceException,
                 IllegalBindingException,
                 IllegalLifeCycleException;
}
```

Figure 2.7. Binding Controller API [6]

By calling the `listFc` operation the names of the client interfaces of the component are returned. These names are consumed by the `lookupFc` operation in order to provide the server interfaces that are bound to the given client interfaces. The `bindFc` operation is used to bind two components together by binding their interfaces, a component's client interface with another component's server interface. Lastly, the `unbindFc` operation is used to unbind a component using the client interface of it. Possible exceptions thrown from the aforementioned operations are the `NoSuchInterfaceException` which means that the client interface passed does not exist, the `IllegalLifeCycleException` which is thrown when the component's state prevents it from executing the requested operation and the `IllegalBindingException` which is thrown when a failure in binding occurs.

2.2.3.3 Content Controller

The `ContentController` interface is used to manage the contents of a controller. More specifically it provides the functionality to add components to a component (e.g. subcomponents), to remove them as well as to list the interfaces it contains (Figure 2.8).

```
package org.objectweb.fractal.api.control;

interface ContentController {
  any[] getFcInternalInterfaces();
  any getFcInternalInterface(string itfName)
      throws NoSuchInterfaceException;

  Component[] getFcSubComponents();
  void addFcSubComponent(Component c)
      throws IllegalContentException, IllegalLifeCycleException;
  void removeFcSubComponents(Component c)
      throws IllegalContentException, IllegalLifeCycleException;
}

interface SuperController {
  Component[] getFcSuperComponents();
}
```

Figure 2.8. Content Controller API [6]

The `getFcSubComponents` operation gets the sub-components contained in the component as an array of `Components`. The `addFcSubComponent` operation adds the passed `Component` to the component's contents whereas the `removeFcSubComponent` removes it. The operations `getFcInternalInterfaces` and `getFcInternalInterface` are used to get the component's internal interfaces.

On the `SuperController` interface the `getFcSuperComponents` operation returns the component's “parents”, that is the components that contain this component - known as its super-components. Apparently a component can be added to more than one components as a subcomponent, becoming that way a shared component.

2.2.3.4 Life Cycle Controller

Dynamic (re)configuration, as described above, such as bindings between components, additions or removals of sub-components can not be safely provided without a minimal Life Cycle controller. At its minimum this controller should provide a start and a stop operation to properly start and stop a component. The API suggested for the `LifeCycleController` is shown in Figure 2.9.

```
package org.objectweb.fractal.api.control;

interface LifeCycleController {
  string getFcState();
  void startFc() throws IllegalLifeCycleException;
  void stopFc() throws IllegalLifeCycleException;
}
```

Figure 2.9. LifeCycle Controller API [6]

Apart from the `startFc` and `stopFc` operations there is a `getFcState` operation which provides the state of the component. Two possible states are available: STARTED and STOPPED whose meaning is obvious.

2.2.4 Instantiation

The creation of new components is done through a framework based on component factories, which are themselves components. These factories can be categorized into generic factories and standard factories, and both can provide the `GenericFactory` and `Factory` interfaces. Generic factories can provide different types of components while the standard factory only creates one type of specific component, always of the same type.

In Figure 2.10 the two above mentioned interfaces and the operations they provide can be seen. It is worth noticing that the `Factory` interface does not take any parameters for the `newFcInterface` operation due to the fact that it only generates one type of component, whereas the `GenericFactory` needs to have specified all the necessary parameters to create a component.

```
package org.objectweb.fractal.api.factory;
interface GenericFactory {
   Component newFcInstance(
      Type t, any controllerDesc, any contentDesc)
      throws InstantiationException;
}

interface Factory {
   Type getFcInstanceType();
   any getFcControllerDesc();
   any getFcContentDesc();
   Component newFcInstance() throws InstantiationException;
}
```

Figure 2.10. Instantiation API [6]

Templates are a special type of standard factory component that generate Fractal components with the same functional client and server interfaces as the template, as well as the same attributes, but can have different control interfaces. Templates also allow for subcomponent templates as seen in Figure 2.11. Template components, normally, are useful only when several identical components need to be created, and instead than generating each component by parsing the ADL file as many times as the needed components, it is only parsed once to create the template from which the needed components will be generated. Obviously, this technique is more efficient.

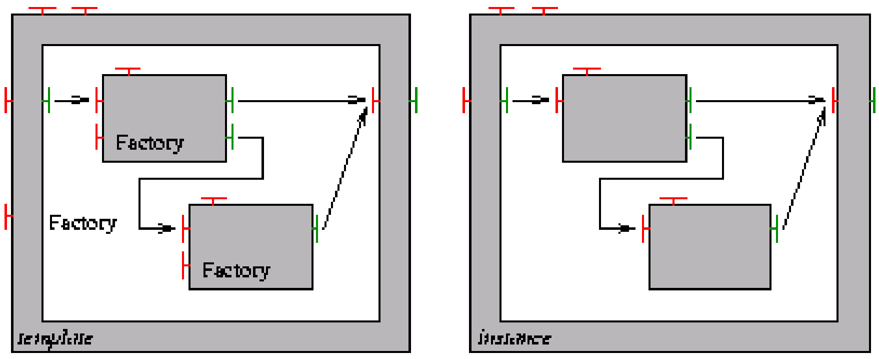


Figure 2.11. Sample template component (left) and component it generates (right)

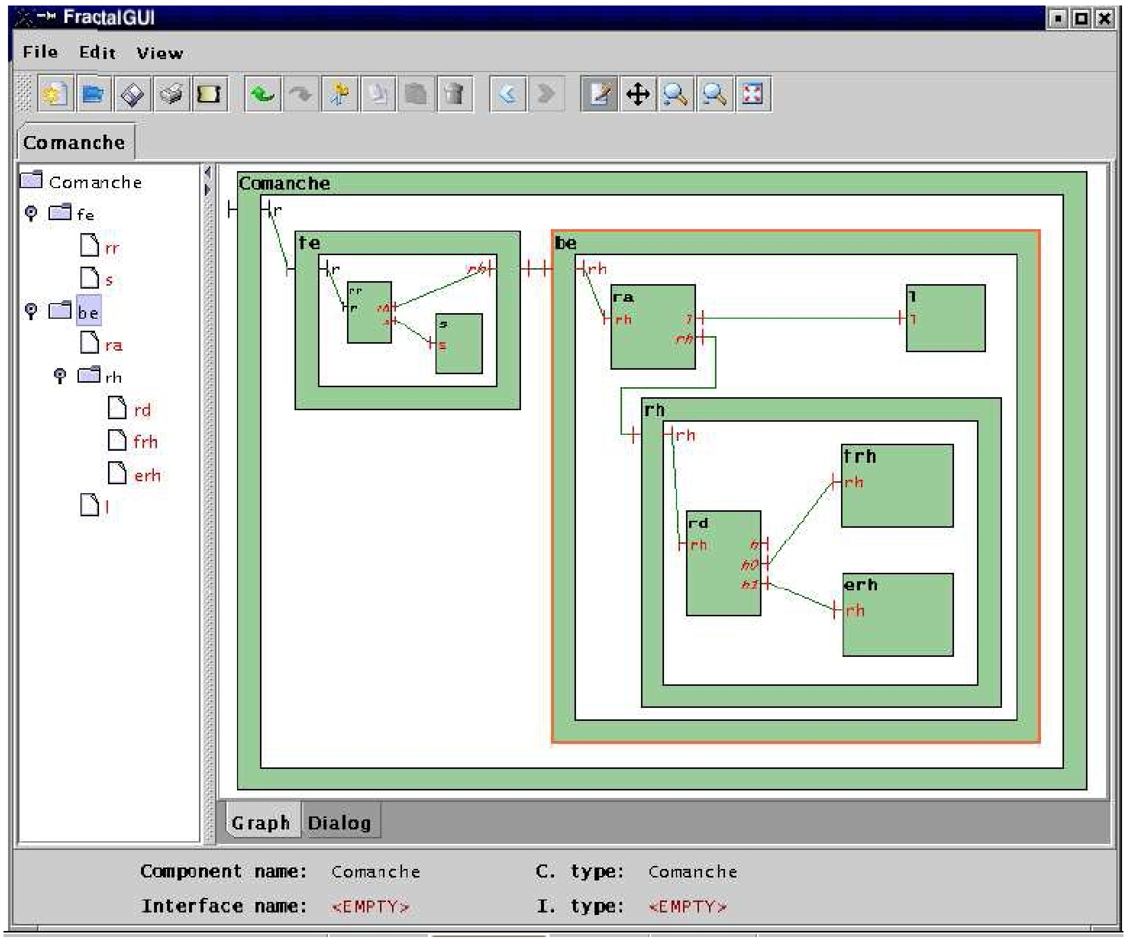
It has been shown that components (factories) can generate other components. However, an "initiator" component should exist. This is achieved through a bootstrap component factory, which is created automatically and is well known and accessible, and which provides the `GenericFactory` interface.

2.2.5 Deployment

There are three main approaches to assemble and deploy Fractal components. All can be used independently or they can be combined forming a hybrid approach. The first approach is the programmatic approach. This is the most direct way and consists of writing a program to create, bind, deploy and start the application.

The second approach is the ADL based configuration. This approach is better in the sense that it results in fewer errors, separates the architecture description from the deployment and abstracts away the component architecture. This is mainly done using Fractal ADL, an Architecture Description Language based in XML for which parsers and verifiers exist, but can be also done with any ADL (even specifically created ones) since ADLs are not part of the Fractal component model. Once the architecture describing the application is complete (using Fractal ADL or any other ADL) it can be compiled into a Java class or directly interpreted.

The last approach is to use a GUI based configuration. This approach helps to generate a visual representation of the architecture, for example using the Fractal GUI tool, from which a screenshot can be seen in Figure 2.12.



Component name: Comanche C. type: Comanche
Interface name: <EMPTY> I. type: <EMPTY>

Figure 2.12. Fractal GUI tool

2.2.6 Conformance Levels

Based on the fact that in the Fractal component model everything is optional, any two given components will most likely not work with each other because they will probably use very different and maybe even incompatible options or extensions of the Fractal model. To approach this problem, a set of conformance levels has been created. Two components with the same conformance level are certified to properly work together, thus making it trivial to find out which components are compatible. In Table 4, a summary of the Fractal model conformance levels is presented.

Level Number	**Description**
level 0	Nothing is mandatory, components are just like objects.
level 0.1	Same as level 0 but all components with configurable attributes must provide `AttributeController` interface; with client interfaces must provide `BindingController`; components that expose their contents must provide the `ContentController` interface; and components that expose their lifecycle must provide the `LifeCycleController` interface.
level 1	Same as level 0. All components must provide `Component` interface.
level 1.1	Same as level 1 plus requirements for level 0.1
level 2	Same as level 1. All component interface references must cast to `Interface`
level 2.1	Same as level 2 plus requirements for level 1.1
level 3	Same as level 2. All components must use the type systems defined in the Fractal component model specification.
level 3.1	Same as level 3 plus the requirements of levels 0.1
level 3.2	Same as level 3.1 plus the requirement of the accessible bootstrap component.
level 3.3	Same as level 3.2 plus the requirement that the `GenericFactory` interface of the bootstrap component must be able to create primitive and composite components

Table 4. Conformance levels of Fractal components

The conformance levels explained in Table 4 are not sufficient to compare two Fractal systems because compatibility issues may appear, for example, with Fractal systems dedicated for a specific language, in which case a new conformance level can be created.

2.2.7 Example

A simple HelloWorld application based on Fractal components is included in Appendix A. Initially, the application's architecture was designed using the Fractal GUI tool. Given the interfaces of the components and the bindings between them, the tool attempted to produce the necessary Fractal files, the Java interfaces and skeleton classes.

The application consists of two primitive components inside a composite component (Figure 2.13). The server component provides an interface to print messages passed to it. The client component just uses the server's service to print a "hello world" message. The application is accessible from outside through an interface bound to the client component.

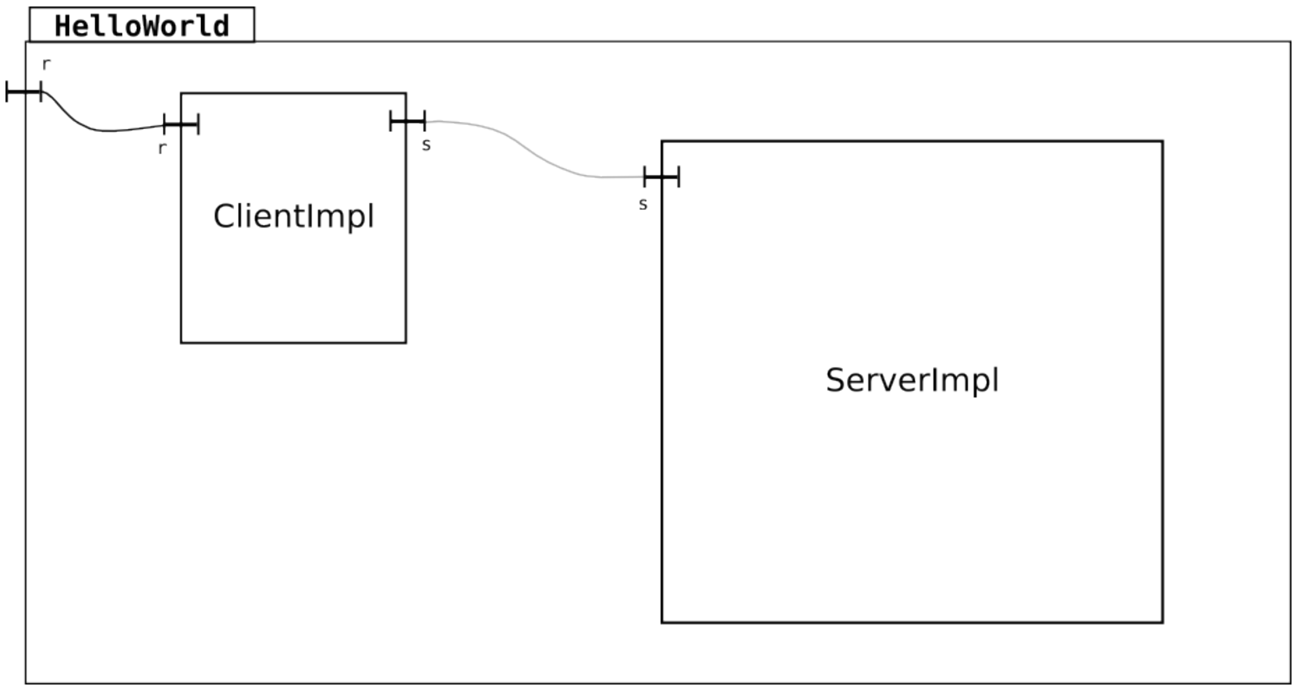


Figure 2.13. Fractal components in the HelloWorld application

2.3 Management Authority

One of the strong points of the Jade management framework is its emphasis on repair management, having the whole managed system under the scrutiny of the repair manager (the repair manager node(s) being also managed). The repair management architecture conforms to the Fractal component model, as everything in Jade does, enabling all Jade components to reside under the repair management domain. This domain is composed of a set of nodes (usually mapped to physical machines) and their subcomponents (typically software components on the nodes), a set that can dynamically change due to node failures and new node introductions.

Nodes can be automatically managed by Jade; for example on a node failure, an identical node to the crushed one can be introduced in the system, to replace the failed one. They can also be manually managed by an administrator using the provided interfaces.

2.3.1 Repair Management Loop

The repair management loop basically consists of a system feedback loop that can be seen in Figure 2.14.

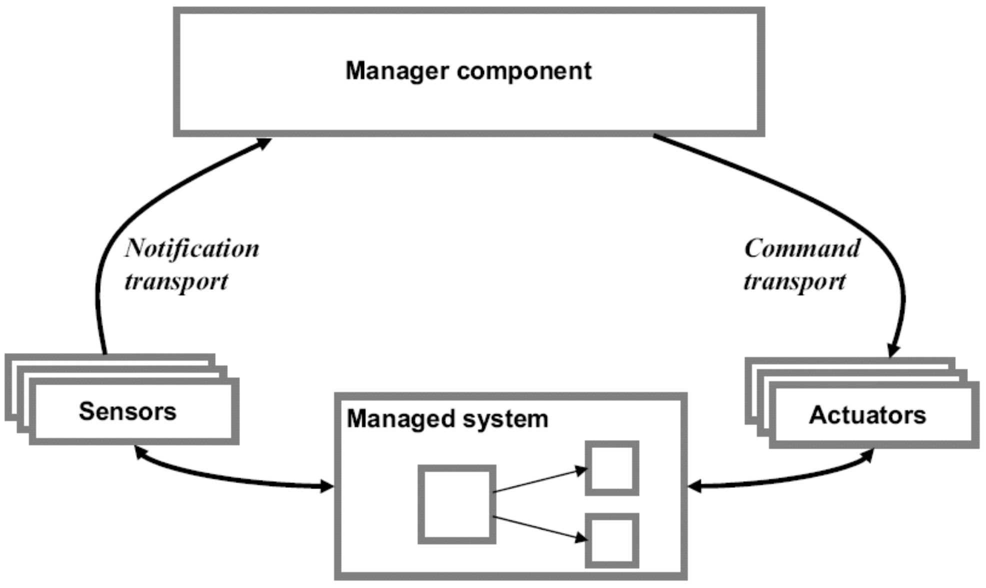


Figure 2.14. Repair management loop [8]

This feedback loop has four main elements:

1. Sensors: in charge of the detection of component state changes, resource change/use and component failure (node failure).
2. Transport subsystem: consists of the bindings between sensors, actuators and the manager component.
3. Manager component: the component in charge of analysing the received data and making decisions based on the system representation, the existing policies and the data received.
4. Actuators: execute the decisions made by the manager component on the managed components. These decisions can be of two types:
   a) Lifecycle actions: stop and resume execution of components in a node and of their bound components.
   b) Configuration actions: to add or remove components from a node and their bindings or to change the code of running components.

The repair policy is established by a set of rules specified by a system policy. In the following figures, a basic failure-repair cycle is described:

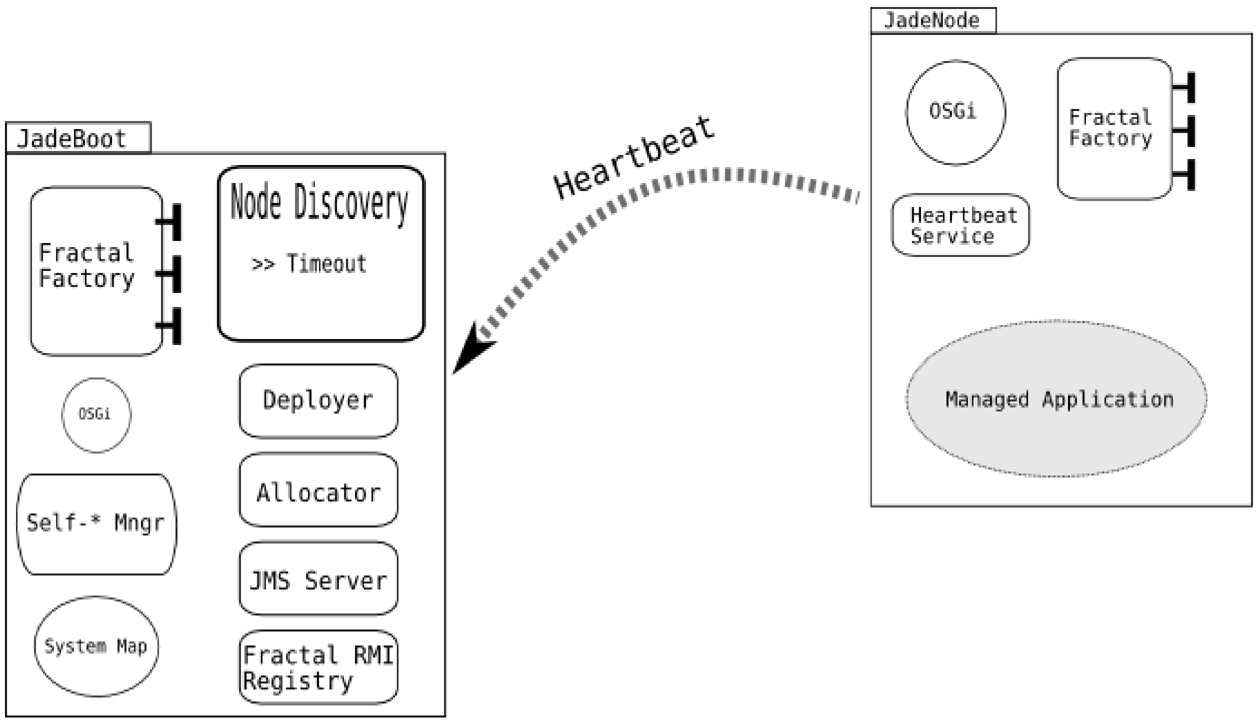


Figure 2.15. Self-repair example - step 1

- In Figure 2.15 the system before failing is presented. The JadeNode with the managed component periodically sends a heartbeat informing JadeBoot of its current state.
- Figure 2.16 represents the failure of the JadeNode where the managed component resides. The Node Discovery module (Manager) is set to wait a period of time (timeout) without receiving a heartbeat to classify a node as failed.

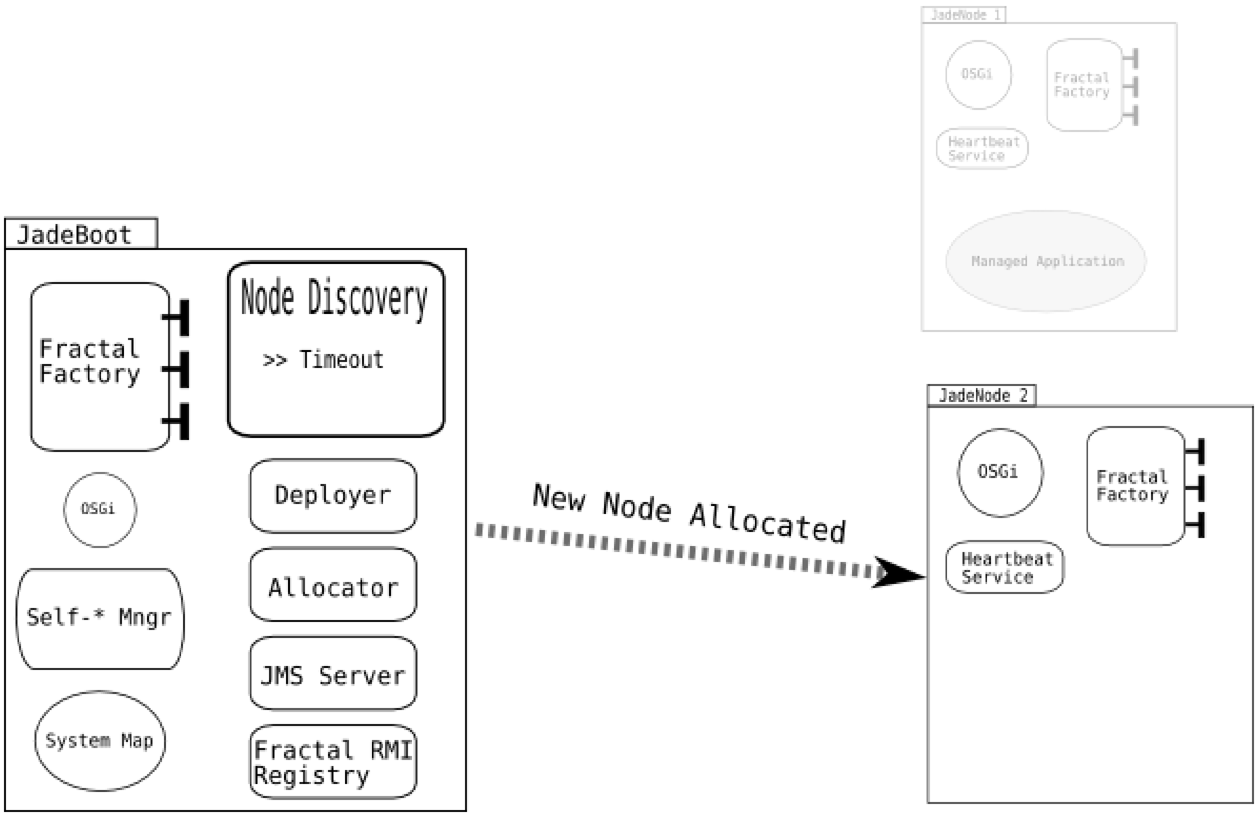


Figure 2.17. Self-repair example - step 3

- Upon realization of the failure of JadeNode 1, the system repair manager, based on the system's policy and the current state of the system, allocates a new empty node, as it can be shown in Figure 2.17.

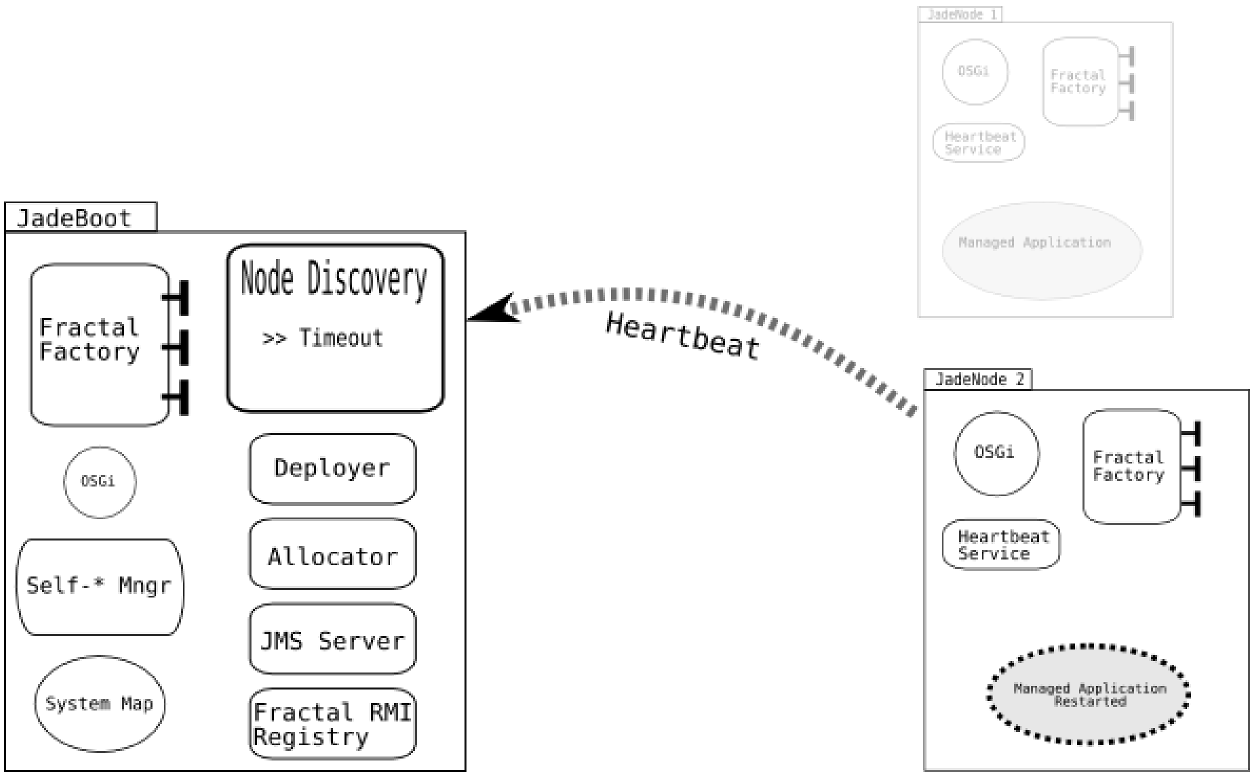


Figure 2.18. Self-repair example - step 4

- Once the new node is allocated, the repair manager deploys and starts the lost component (previously on JadeNode 1) on the new node and restores the system back to normal, everything achieved in an automatic and transparent fashion.

It is important to mention at this point that this is an example of a very simple repair management scenario, and that more complete and sophisticated policies would be needed in more complex cases, mostly due to the stateless properties of the above mentioned failure scenario.

The repair manager, a Fractal component itself, is also vulnerable to failures, so this centralized approach to repair management is not safe as it is. To deal with this, the notion on replication is considered to deal with failures that affect the management component, having several manager components, one of them being the leader and the others acting basically as up-to-date backups. In case of failure of the leader, a leader election algorithm would pick a new leader and continue functioning, achieving failure repair transparency.

2.4 JADE's underlying technologies

The Jade management framework is a fairly large project built on top of many different technologies that provide necessary functionality. This section aims at explaining, to some detail, how these technologies are used in the context of Jade.

2.4.1 Fractal

The Fractal Component Model [62] is basically a specification of a component model [63]. There is a variety of implementations (Julia, AOKell, FractNet, FractTalk, Plasma, ProActive) on various different programming languages (Java, .NET, SmallTalk, C++). In Jade the Julia implementation is used. Julia [64] is the reference implementation of Fractal written in Java and it fully complies with the specification. Julia provides Jade with the necessary framework for creating Fractal components.

In order to make remote method calls on Fractal components Jade uses Fractal RMI [65]. Fractal RMI supplies Jade with *"a set of components to create distributed bindings between Fractal components"*. In practice it *"provides protocol, binder, and stub factory components for remote method calls between Fractal components"*.

Lastly, Fractal ADL [66] is the architecture description language for the Fractal component model. It is an XML based descriptive language, used to define component types, implementations, hierarchies and bindings. In Jade everything is a Fractal component described in Fractal ADL.

2.4.2 Java Message Service

The Java Message Service (JMS) API *"is a messaging standard that allows application components based on the Java 2 Platform, Enterprise Edition (J2EE) to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous"* [57]. In the context of the Jade framework, JMS is used to asynchronously communicate between different (Java implemented) modules. The clearest example of JMS usage in Jade is in the "heartbeat" mechanism used to detect new nodes and node failures.

2.4.3 Oscar

Oscar is an open source implementation of the Open Services Gateway Initiative (OSGi) [55]. The OSGi Alliance is an open standards organization that specifies service platform. This Framework specifies a complete and dynamic component model where applications (as components) can be remotely installed, started, stopped, updated and uninstalled.

Applications are accessible to Oscar in what is known as "bundles", a JAR file containing all the necessary parts to completely configure and install the application (component). Each JAR bundle contains a minimum of a manifest describing the bundle and any other files or archives such as Java classes, legacy application (possibly as binaries), native code and other resources. Actually, anything relative to the application can be included in a bundle.

These bundles are usually placed on Oscar Bundle Repositories (OBR) [67] stored locally, or more commonly, on shared resources, such as network drives or in online-accessible servers, so they can be quickly and uniformly accessed and downloaded to perform a transparent and direct installation of the bundle.

The JADE management framework uses Oscar's functionality in two main ways. Firstly, Oscar is used to start the JADE platform itself. More specifically, and by specifying certain parameters in configuration files, Oscar provides an easy and straightforward way of starting the JADE platform using a technology that is used by JADE anyway. With one single command, and the right configuration files, Oscar starts a JadeBoot or a JadeNode. The second way Oscar is used is the most important one, since it handles all the fetching and installing of bundles on local machines. When a JadeNode has been allocated to the JADE framework and a component is to be installed on it, this is where Oscar comes into play. Oscar accesses the OBRs specified in the configuration files and locates the desired bundle containing the correct application, fetches it from the repository and saves a local copy of the bundle on the proper JadeNode, extracting the JAR file to the specified local folder. The JADE management framework uses Oscar to install and uninstall bundles, but it does not use Oscar's functionality to start and stop resources, this is managed by the JADE Wrappers (section 2.5).

2.4.4 BeanShell

BeanShell is a *"Java source interpreter with object scripting language features, written in Java"* [61]. In the context of the JADE framework, BeanShell provides a command line interface that provides interaction between a JADE user (administrator) and the JADE platform. Even though one of the main goals of the JADE framework is to achieve self-management, an administrator interface is required in order to be able to do any type of manual change to the system, as well as to perform some monitoring.

The interaction between BeanShell and the JADE framework is established by importing a series of commands specific to the JADE platform. These commands are Java-like classes that implement some desired functionality and are usually bundled in JAR files that are referenced through the classpath. A typical example is the "deploy" Beanshell command that takes as an argument the name of the application to be deployed.

2.5 DEVELOPING WITH JADE

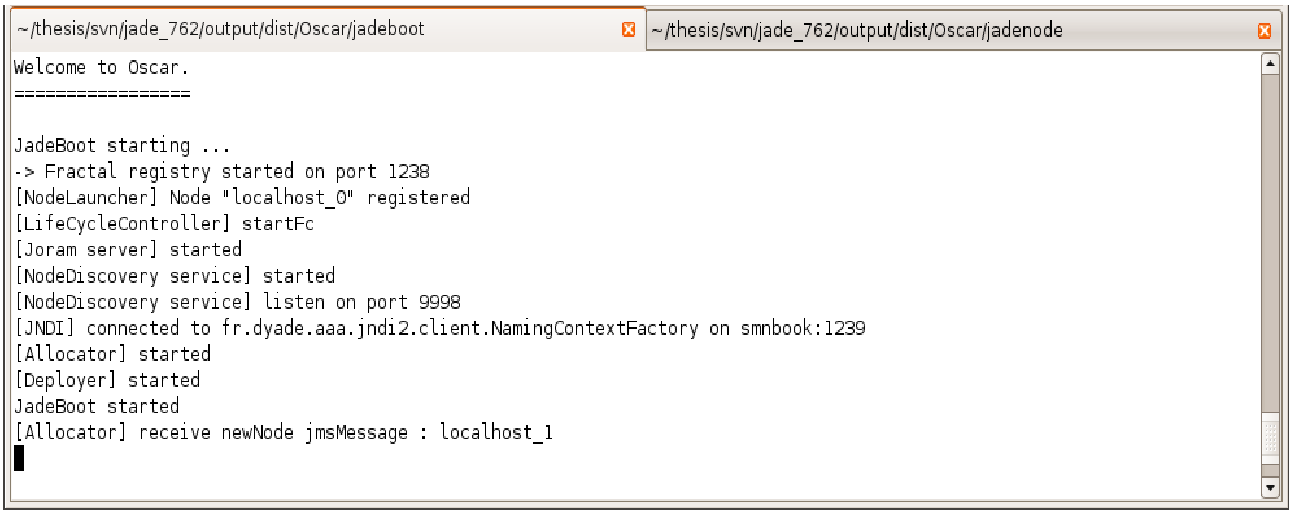
The JADE component management framework is completely based on the Fractal component model and as such, any application that is to be either designed to be executed and managed by JADE or adapted for JADE has first to be architecturally described using an architecture description language. Fractal ADL [66] is the one JADE uses. Details of this procedure can be found by reading the Fractal component model specification [63] and the Fractal ADL project site [66].

Once the application's architecture has been described, the next step is to generate an Oscar bundle [56] [67]. Oscar bundles present a uniform way used by JADE to, exclusively, deploy and undeploy the application itself; more specifically to fetch or download it, extract it to some local folder and install it if required, a process that can vary considerably depending on the type of application. Oscar bundles are most likely to be used under the JADE framework to encapsulate legacy software (binaries or executables) although they can also be used to encapsulate application directly designed and implemented for JADE thus achieving a degree of separation between the application and the way it interacts with the JADE platform. Oscar bundles written specifically for the JADE framework must implement the "`start`" and "`stop`" methods of the `BundleActivator` interface, methods that actually take care of the deploy and undeploy procedures for the given application.

Once the Oscar bundle is complete, the next step consists of “linking” it to the JADE platform itself, which is done through a JADE wrapper. Written in Java and implementing the necessary controllers (AttributeController, BindingController, ContentController, etc.) JADE wrappers provide the interfaces that JADE needs to manage the application. It is also worth mentioning, that the Oscar bundle step can be avoided by including all the source code of the application in the Wrapper itself, but this approach is not recommended because it limits the desired abstraction between the application and the JADE framework.

2.6 Running applications under JADE

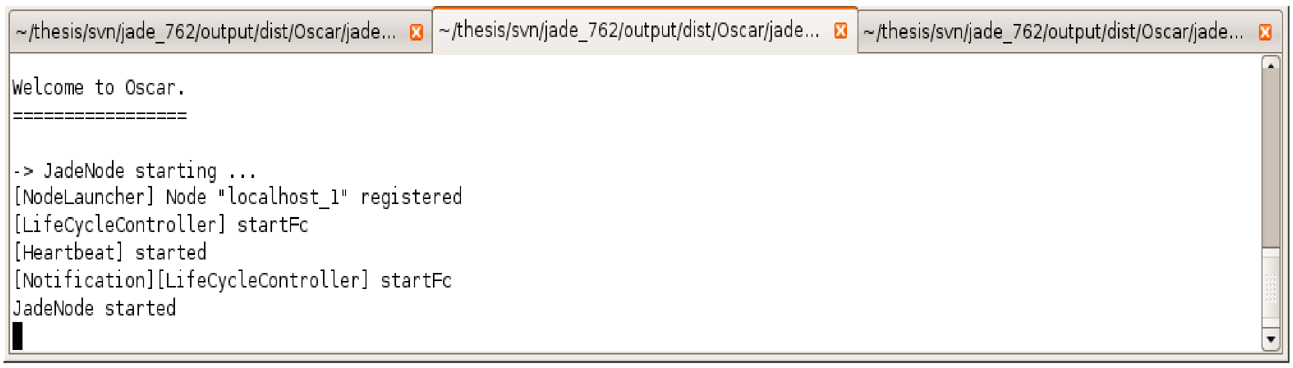
Running an application in JADE consists of three steps. One needs first to start the JadeBoot node. This is done through Oscar. By pointing to the OBR repository and to the JadeBoot bundle in configuration files the appropriate JAR file is downloaded, deployed and started. The second step is to start the available nodes known as JadeNodes. This is also achieved through Oscar by editing properly the configuration files to point to the JadeNode bundle. JadeBoot discovers the new nodes and adds them to the list of available nodes. Figures 2.19 and 2.20 show the output produced by starting a JadeBoot and a JadeNode that connects to that JadeBoot.


~/thesis/svn/jade_762/output/dist/Oscar/jadeboot
~/thesis/svn/jade_762/output/dist/Oscar/jadenode

```
Welcome to Oscar.
=================

JadeBoot starting ...
-> Fractal registry started on port 1238
[NodeLauncher] Node "localhost_0" registered
[LifeCycleController] startFc
[Joram server] started
[NodeDiscovery service] started
[NodeDiscovery service] listen on port 9998
[JNDI] connected to fr.dyade.aaa.jndi2.client.NamingContextFactory on smnbook:1239
[Allocator] started
[Deployer] started
JadeBoot started
[Allocator] receive newNode jmsMessage : localhost_1
```

Figure 2.19. JadeBoot running (screenshot)

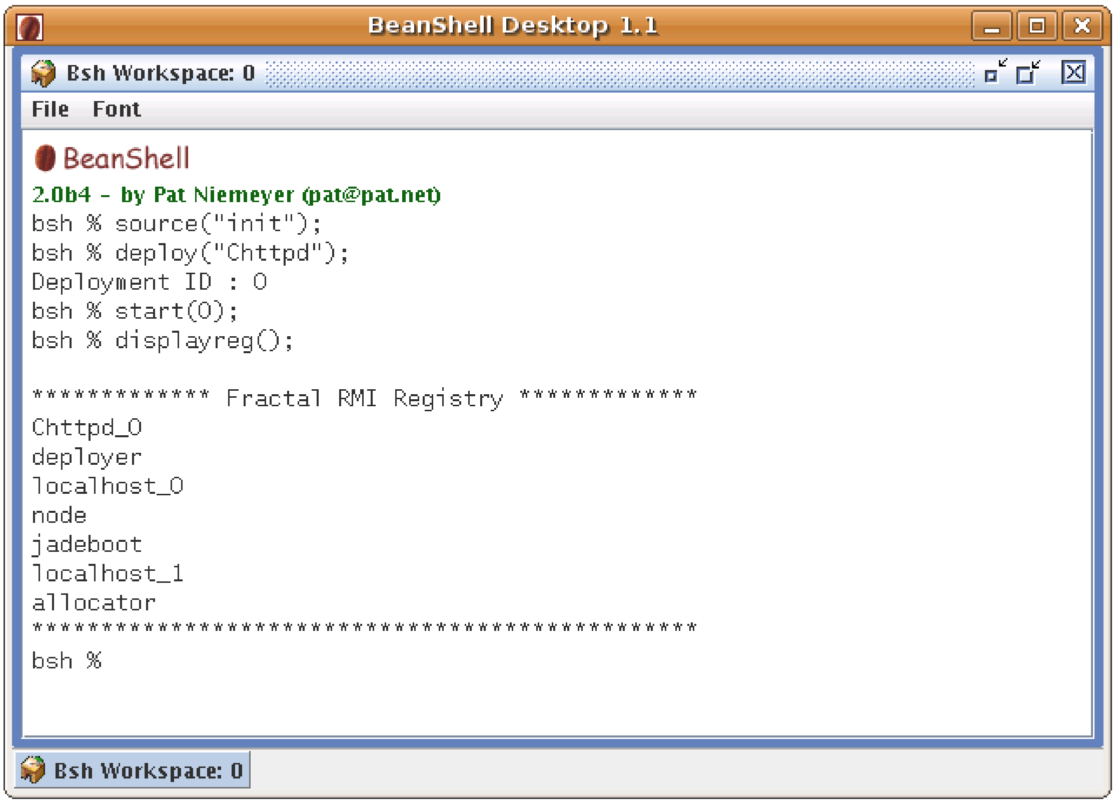

~/thesis/svn/jade_762/output/dist/Oscar/jade...
~/thesis/svn/jade_762/output/dist/Oscar/jade...
~/thesis/svn/jade_762/output/dist/Oscar/jade...

```
Welcome to Oscar.
=================

-> JadeNode starting ...
[NodeLauncher] Node "localhost_1" registered
[LifeCycleController] startFc
[Heartbeat] started
[Notification][LifeCycleController] startFc
JadeNode started
```

Figure 2.20. JadeNode running (screenshot)

Once JadeBoot and JadeNodes are started a shell interface is used to interact with the JADE platform. Beanshell is used for this purpose as described above in 2.4.4. Through Beanshell one can deploy and run applications, as well as monitor the installed components.



```
2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % source("init");
bsh % deploy("Chttpd");
Deployment ID : 0
bsh % start(0);
bsh % displayreg();

************** Fractal RMI Registry **************
Chttpd_0
deployer
localhost_0
node
jadeboot
localhost_1
allocator
************************************************
bsh %
```

Figure 2.21. Beanshell (screenshot)

Chapter 3

Assessment of the JADE Framework

In this chapter of the thesis report, an assessment of the Jade management framework will be made considering several different aspects. First, the capability of the framework to operate under a wide-area environment will be discussed. Then an assessment of the process to create and deploy a basic application for the Jade framework. This will be followed by an evaluation of the process necessary to enable a legacy application to be run under the Jade framework as well as the process of designing an application with Jade in mind from the beginning. The fourth point considered will be the overhead generated by using the Jade framework. To evaluate this, some data will be presented regarding memory usage of the framework and the minimum amount of code necessary to make a legacy application run under Jade. To finalize, some general comments and suggestions regarding the Jade framework will be made.

All the conclusions, comments and remarks made regarding the Jade management framework are based on revision 762 and may not apply to later or earlier versions. Generic concepts and ideas should apply if no changes in design have been made.

3.1 Wide-area capability

In this subsection, the shortcomings of the Jade framework with respect to its possible adaptation to a wide-area network will be discussed.

3.1.1 The Mechanism

The Jade management framework uses a fairly simple failure detection mechanism based on a timeout. When JadeNodes register with a JadeBoot, they start sending what is named a "heartbeat", which is a periodic message sent using the Java Message Service (JMS) protocol integrated into Jade. This heartbeat is expected for all registered JadeNodes every predetermined amount of seconds, a value specified as a property (named "jadenode.heartbeat.pulse") when starting each JadeNode. If the heartbeat is not received for a fixed amount of time (the timeout), then JadeBoot assumes that the JadeNode from which the heartbeat was not received is dead and proper measures are to be taken. It is important to notice that, as of the current version of JADE, the timeout is a hard-coded value.

3.1.2 Evaluation

The JADE framework was designed and implemented with a cluster topology in mind, which makes things easier by making some strong assumptions; the most important is that there are no failures or communication problems. These assumptions, while intended to make the design and implementation process easier, also put strong limitations on several aspects of the JADE platform; limitations that show up when trying to port the platform to a wide-area network where communication is not reliable. Based on the limited amount of insight we have on the platform itself regarding implementation details, one of the main problems that the JADE platform may face when trying to adapt to a wide-area environment is its naïve failure detection mechanism based on a simplistic timeout. This mechanism will assume a node as dead if a heartbeat message is lost or corrupted and it will immediately take action by attempting to allocate a new node to replace the assumed dead and redeploying the components of the dead node on the new node, when in fact, the node assumed dead might be still alive and running. This approach might cause many problems such as delay, loss of service, possible loss of data if the node assumed dead was a stateful node, waste of resources (alive nodes) and many more problems depending on the application being managed.

In order to alleviate this problem it would be recommendable to look into more advanced failure detection mechanisms available that take into consideration the possible communication problems that wide-area networks present. We will not discuss which failure detection mechanism would fit better since it is not in the scope of this project, but we think it is crucial for the JADE framework to drastically improve it. The side effect of a more sophisticated failure detection mechanism might be an increase in overhead and repair time but it will make the framework much more robust.

3.2 Programmability

The programmability assessment will be done through studying the creation and deployment process of a simple application. The application is a trivial HTTP server and is considered as a legacy software ready to be used out of the box in a non Jade environment. In such a case, the creation process includes describing the application's components using Fractal ADL, creating a bundle which will be used by the platform to install the legacy software and finally writing the wrapper that implements the interfaces provided by the application. Deployment is almost the same for every application. In short, it consists of updating the repository by uploading Fractal files, bundles and wrappers.

3.2.1 Developing an Application

A Jade application is typically a composite component. Moreover, a common approach is to include inside every composite component a primitive one called "start" that handles starting and stopping of resources. Finally, components are bound to each other using Fractal bindings which provide signature contracts between components' interfaces.

The Jade application, that is used in this section, consists of a composite component that includes two primitive components: the typical "start" component and the "chttpd" component that provides the server's functionality. Figure 3.1 portrays the application's component architecture.

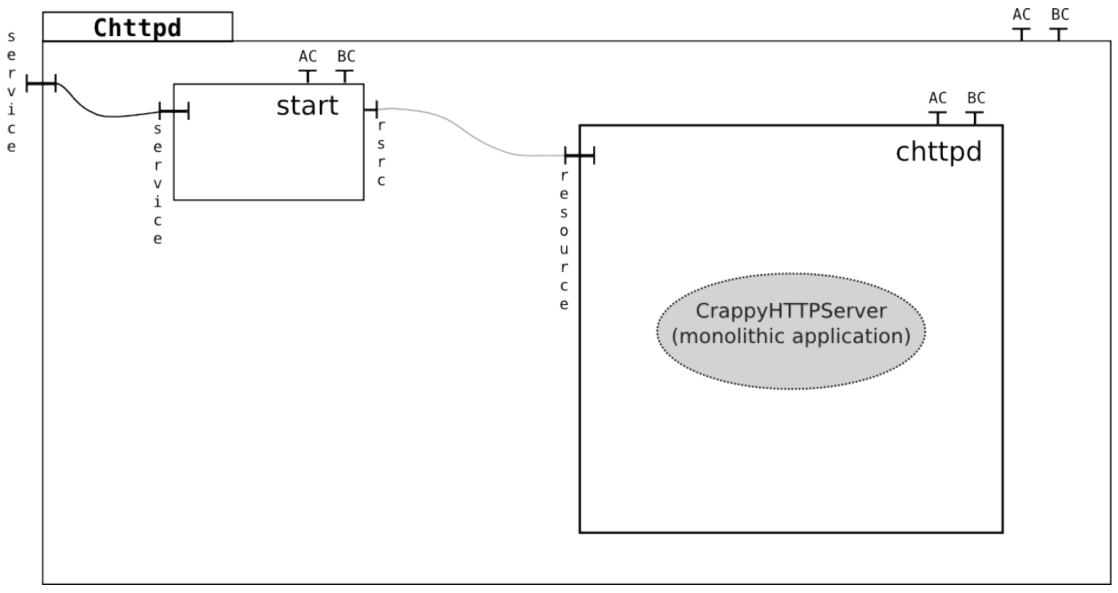


Figure 3.1. Chttpd Architecture

3.2.1.1 Fractal ADL Description

The first step is to convert the architectural representation shown above to a machine readable format. Fractal ADL is used for that purpose and the output is a file that describes the whole architecture in XML. A full version of the Fractal ADL file can be found on Appendix C.

```
<definition name="Chttpd">
  <interface name="service" role="server" signature="fr.jade.service.Service" />

  <component name="start" definition="fr.jade.resource.start.StartType">
    <virtual-node name="node0" />
  </component>

  <component name="chttpd"
       definition="sics.jade.resource.chttpd.ChttpdResourceType">
    <attributes
       signature="fr.jade.fractal.api.control.GenericAttributeController">
      <attribute name="resourceName" value="chttpd" />
      <attribute name="port" value="8080" />
       ...
    </attributes>
    <virtual-node name="node1" />
    <packages>
      <package name="Crappy HTTP Server">
        <property name="dir.local" value="/tmp/chttpd/" />
        <property name="dir.install" value="chttpd/" />
      </package>
    </packages>
  </component>

  <binding client="this.service" server="start.service" />
  <binding client="start.rsrc_chttpd" server="chttpd.resource" />
  <virtual-node name="node0" />
</definition>
```

The composite component is called “Chttpd”. It provides a server interface called “service” that makes the component reachable from the outside by providing two methods “start” and “stop” as defined in the `fr.jade.service.Service` Java interface. This interface is bound to the primitive component “start” which, as mentioned above, is used mainly for controlling the bootstrapping process. The other primitive component is “chttpd” which is responsible for offering the application's service. Note that this component makes use of the `AttributeController` so that the application can be configurable. The “package” element is used to define what bundles should be installed as dependencies of this component. This functionality is provided by Oscar - the OSGi implementation - and in this specific scenario the bundle named “Crappy HTTP Server” will be downloaded and installed from the Oscar bundle repository. Finally the “virtual-node” element defines the node that will host each component. If there is more than one physical node available, then the application can be distributed among nodes by distributing its components.

The definitions of the two primitive components, “start” and “chttpd” can be found in the Fractal ADL code that follows. Full versions of the files are also in Appendix C.

```
<definition name="fr.jade.resource.start.StartType">
  <interface name="service" role="server" signature="fr.jade.service.Service" />

  <interface name="rsrc" role="client" signature="fr.jade.resource.Resource"
    cardinality="collection" contingency="optional" />

  <content class="fr.jade.resource.start.Start" />
</definition>


<definition name="sics.jade.resource.chttpd.ChttpdResourceType">
  <interface name="resource" signature="fr.jade.resource.Resource"
    role="server" cardinality="collection" contingency="optional" />

  <content class="sics.jade.osgi.chttpd.wrapper.CrappyHttpServerWrapper" />
  <controller desc="parametricprimitive" />
</definition>
```

The content class of each primitive component is supposed to implement the server interfaces of the component. As regards the “start” component the implementation class is provided by JADE and specifically by the Java class `fr.jade.resource.start.Start`. Therefore, this class should implement the “start” and “stop” methods defined in the `fr.jade.service.Service` interface. The client interface “rsrc”, as it can be noticed on the main Fractal file, is bound to the server interface of the other primitive component “chttpd”. The content class of the “chttpd” component should implement the `fr.jade.resource.Resource` interface in the `sics.jade.osgi.chttpd.wrapper.CrappyHttpServerWrapper` class.

3.2.1.2 Oscar Bundle

The second step is to create the bundle that will enclose the legacy software. According to OSGi specification [55] a bundle must implement the `org.osgi.framework.BundleActivator` interface which defines two methods: “start” and “stop”. Their name inside the JADE context can be misleading because they are used for different purpose than one would guess. In fact, these two methods play the role of install and uninstall since the JADE wrapper is the one that will take care of starting and stopping components and applications. Usually, the legacy software is included in a compressed archive inside the JAR file of the bundle so it can be easily extracted and installed.

The Java documentation (javadoc) of the bundle can be found on Appendix C.

3.2.1.3 Jade Wrapper

The Jade wrapper that needs to be written by the developer should implement the server interfaces defined by the "chttpd" component. In this particular case, the `CrappyHttpServerWrapper` class of the `sics.jade.osgi.chttpd.wrapper` package must implement the `fr.jade.resource.Resource` interface but also the `GenericAttributeController` and the `BindingController`. The Resource interface defines four methods: configure, loadApp, start and stop. As already mentioned this server interface inside "chttpd" is bound to the "start" component's "rsrc" client interface and is used to activate and deactivate the component. The `GenericAttributeController` (section 2.2.3.1) defines methods to set, get and list the component's attributes whereas the `BindingController` (section 2.2.3.2) defines the necessary methods to bind and unbind components, to list the component's client interfaces and to reach bounded components by using lookups.

It is important to state that a Jade wrapper is an Oscar bundle per se and hence has to implement the `BundleActivator` interface. However, the two methods "start" and "stop" of the implementor class can be empty.

In this specific HTTP server example, the wrapper's start method is trying to run the server installed by the bundle after it updates its configuration file with the parameters read from the Fractal ADL file. It also saves the process ID of the server in order to be able to shut it down later using the stop method. The stop method stops the server by killing its process.

The Java documentation (javadoc) of the wrapper can be also found on Appendix C.

3.2.2 Deploying an Application

The deployment process consists mostly of updating the Oscar Bundle Repository (OBR). OBR is an XML based repository and Jade has adapted it to serve as its main repository for every kind of resource. Even the Jade platform itself is stored in this repository. In such a case Oscar is used first to install the Jade platform (JadeBoot and JadeNode are also Oscar bundles) before one can interact with it.

The necessary deployment steps can be summarized as follows: first, the repository's XML file is edited to include the newly created bundles. The bundles should then be uploaded to the address set on the XML file. The application's Fractal ADL file should also be uploaded into the repository and the JadeBoot's configuration file – the file that is read when starting JadeBoot through Oscar – should be edited to point to the address that contains this ADL description. Normally other Fractal ADL files used to describe primitive components would be uploaded into the same folder, keeping the package names, but revision 762 does not support this feature. Unfortunately they have to be embedded into the platform by editing the JadeBoot bundle.

3.2.3 Evaluation

Both the development and the deployment process have proved to be time consuming and, most importantly, error prone. There are no satisfactory tools to automate any process at all and developers need to spend much time and effort in tasks that could be easily become automated.

First of all, it sounds as a necessity to have a tool that can produce Fractal ADL files and Java skeleton classes from a graphical representation of the application. This would relieve developers from having to manually write XML files and Java methods with repeated content. It would also certainly reduce errors and help produce uniform applications. The only tool available that is supposed to serve partially that purpose is FractalGUI, but it is extremely buggy, inflexible, hard to use and produces wrong output. In addition it does not support automatic implementation of well known interfaces like the ones for attribute and binding controllers.

Another important aspect is the complete lack of a basis to write and build bundles. Starting from scratch was in most of the cases the only option. Having some template bundles and wrappers would at least alleviate some of the problems that one is facing when it comes to making a JADE enabled application. In addition, due to the dynamism and modular nature of the JADE platform, there are a lot of difficulties in referencing the required libraries (Fractal, OSGi and JADE based) since most of the times they are supposed to be exported by the JADE platform itself using manifest files and the OBR repository. The main problem then is the debugging process which is certainly one of the basic drawbacks of JADE. Indeed, taking into account that a single change requires a fully manual deployment makes debugging discouraging.

Obviously, the way JADE's revision 762 handles primitive components' descriptor files is another factor that complicates the deployment process. However a dynamic load of these Fractal ADL files from the repository should be available in later versions.

In conclusion, we believe that, as of now, the creation and deployment process poses a big disadvantage for the JADE framework. It requires extensive knowledge of the platform itself and good debugging skills.

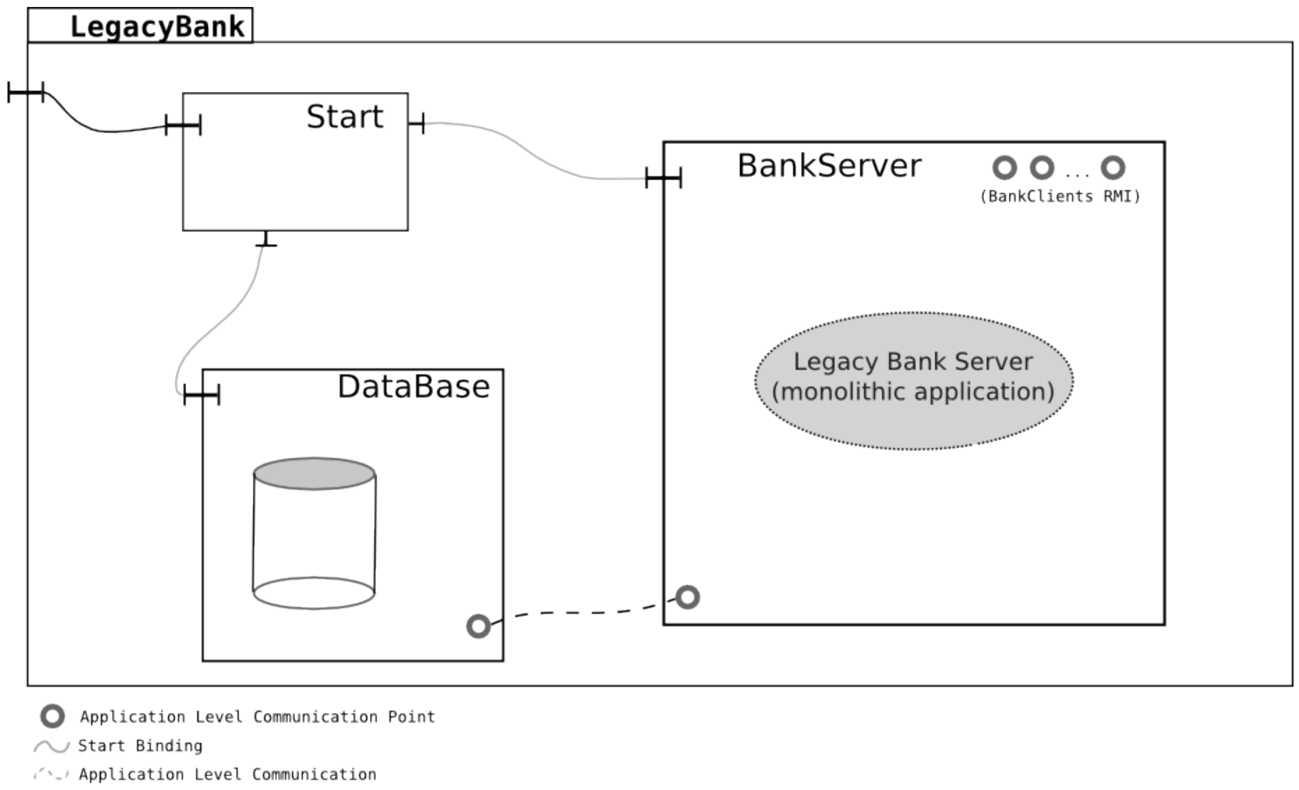
3.3 ARCHITECTURE

The JADE framework is designed to be able to manage any type of application, and since it is written purely in Java, it should be able to run under any environment where Java is supported. For an application to be managed and run under the JADE framework, it has to be adapted for it, following a series of steps, some generic and some specific to the application itself. There are two different approaches to this, either take a complete application and bundle and wrap it so JADE can manage it or design it with JADE in mind, taking into consideration a software component-based architecture and some utilities JADE provides.

In the next two subsections the necessary process for the two above-mentioned approaches will be described using a rather simple client-server application, in this case a BankServer to which BankClients connect via Java Remote Method Invocation (RMI). For the two approaches described (legacy bundling and design with JADE in mind), the interface for the clients to connect to the server will be identical. Actually, the code for the clients is never modified.

3.3.1 Legacy Applications

As mentioned before, any application can be “bundlelized” and wrapped to be manageable by the JADE framework. This is achieved in two steps, first an OSGi compliant [55] bundle has to be generated and then this bundle is wrapped according to the wrapping procedure for JADE, thus obtaining an application that can be deployed and managed under the JADE platform. This process is fairly straight forward and lets any off-the-shelf application to be run under JADE.


LegacyBank
Start
BankServer
(BankClients RMI)
Legacy Bank Server (monolithic application)
DataBase
Application Level Communication Point
Start Binding
Application Level Communication
Figure 3.2. LegacyBank Architecture

As it can be seen in Figure 3.2, the LegacyBank application is composed of three components:

- The "Start" component which is just a boot-strapping mechanism to start the application and actually has nothing to do with the application itself.
- The "DataBase" component which is external to the actual BankServer (any database would be acceptable and does not have to be under the same composite component).
- The "BankServer" component which is the one with all the functionality of the application: RMI server, database interaction and the logic of the application.

Actually, it would be possible to have the LegacyBank application be just one component but for the sake of completeness, and to make it easier to compare with the version designed with JADE in mind we have included the two external components (Start and DataBase).

It is important to see that the communication between the server and the database is achieved using Java DataBase Connectivity (JDBC) (named "application level communication"). Communication at this level is completely transparent to the JADE framework, so in order to let the JADE framework know that there is some communication between the server component and the database component, an "empty" binding is specified between the two as we can see in the snippet of Fractal code provided below. This binding makes JADE aware of dependencies between components even though in reality nothing is ever communicated through the binding itself, and all communication happens at the application level.

```
<definition name="LegacyBank">
  <!-- START -->
  <component name="start" definition="StartType">
   ...
  </component>

  <!-- BANK SERVER -->
  <component name="bankserver" definition="BankServerResourceType">
    <attributes signature="GenericAttributeController">
      <attribute name="resourceName" value="bankserver" />
        ...
    </attributes>
  </component>

  <!-- MySQL -->
  <component name="mysql" definition="MySqlResourceType">
    <attributes signature="GenericAttributeController">
      ...
    </attributes>
  </component>

  <!-- BINDINGS -->
  <binding client="this.service" server="start.service" />
  <binding client="start.rsrc_bankserver" server="bankserver.resource" />
  <binding client="start.rsrc_mysql" server="mysql.resource" />
</definition>
```

The Java documentation (javadoc) for the LegacyBank application together with the Fractal ADL files is included in Appendix D.

3.3.2 JADE Designed Applications

The second approach is to design, or redesign, an application specifically for JADE. This approach would take advantage of the modularity and extensibility that programming with a component-based approach supplies, as well as provide the option to use Fractal Remote Method Invocation (Fractal RMI), similar to Java RMI but already inserted into the JADE platform through the Fractal specification. Fractal RMI provides a transparent and simple way to connect Fractal components without having to worry about application (low level) communication.

Figure 3.3 shows the architecture of the JadeBank application, an example of a Jade-designed application, which is conceptually identical to the one presented in the previous subsection but designed and implemented having Jade into consideration. In this case, the functionality is the same as the LegacyBank (except for the Logger component used to further show the use Fractal RMI) but it is achieved using technologies provided by Jade itself (component and Fractal RMI) rather than legacy mechanisms. The JadeBank application consists of the following components:

- The “Start” component which is just a boot-strapping mechanism to start the application and actually has nothing to do with the application itself.
- The “DataBase” component which is external to the actual BankServer (any database would be acceptable and does not have to be under the same composite component).
- The “BankServer” component which holds the Java RMI server aspect of the application as well as its logic.
- The “DataManager” component, which is the “gateway” to access the database itself. The interesting part is that any number of BankServers could communicate with the DataManager through remote objects via Fractal RMI in a completely transparent fashion.
- The “Logger” component also uses Fractal RMI to “talk” to the DataManager in a transparent way; no effort has to be put to implement this.

All of these components are bound together using Fractal bindings between server and client interfaces, thus indicating to the Jade platform dependencies between them. This approach enables to quickly and effortlessly add a new Logger or a new BankServer just by editing the Fractal file describing the JadeBank composite component, which provides an easy way to control an application's granularity and architecture. Each of the components described above can be placed in any physical node without worrying about ports or addresses since all inter-component communication is handled by Fractal RMI and the Fractal Registry. Therefore, if a node dies and its components are relocated to a new node, other components in different nodes will not notice this and will not have to adapt to it.

The partial Fractal file included below shows the ADL definition, including “real” bindings through which Fractal RMI object references are obtained. This Fractal ADL represents the same architecture that can be seen on Figure 3.3.

```
<definition name="JadeBank">
  ...
  <!-- START -->
  <component name="start">
   ...
  </component>

  <!-- JADE BANK SERVER -->
  <component name="jadebankserver">
    <attributes> ... </attributes>
    <virtual-node name="node0" />
    <packages> ... </packages>
  </component>

  <!-- DATA MANAGER -->
  <component name="datamanager">
    <attributes> ... </attributes>
    <virtual-node name="node1" />
    <packages> ... </packages>
  </component>

  <!-- JADE BANK LOGGER -->
  <component name="jadebanklogger">
    <attributes> ... </attributes>
    <virtual-node name="node0" />
    <packages> ... </packages>
  </component>

  <!-- MYSQL -->
  <component>
    <attributes> ... </attributes>
    <virtual-node name="node0" />
    <packages> ... </packages>
  </component>

  <!-- BINDINGS -->
  <binding client="this.service" server="start.service" />
  <binding client="start.rsrc_jadebankserver" server="jadebankserver.resource" />
  <binding client="start.rsrc_datamanager" server="datamanager.resource" />
  <binding client="start.rsrc_jadebanklogger" server="jadebanklogger.resource" />
  <binding client="start.rsrc_mysql" server="mysql.resource" />

  <!-- bind the jadebankserver to the datamanager -->
  <binding client="jadebankserver.dataservice" server="datamanager.dataservice" />
  <!-- bind the datamanager to the jadebanklogger -->
  <binding client="datamanager.datalogger" server="jadebanklogger.datalogger" />

  <virtual-node name="node0" />
</definition>
```

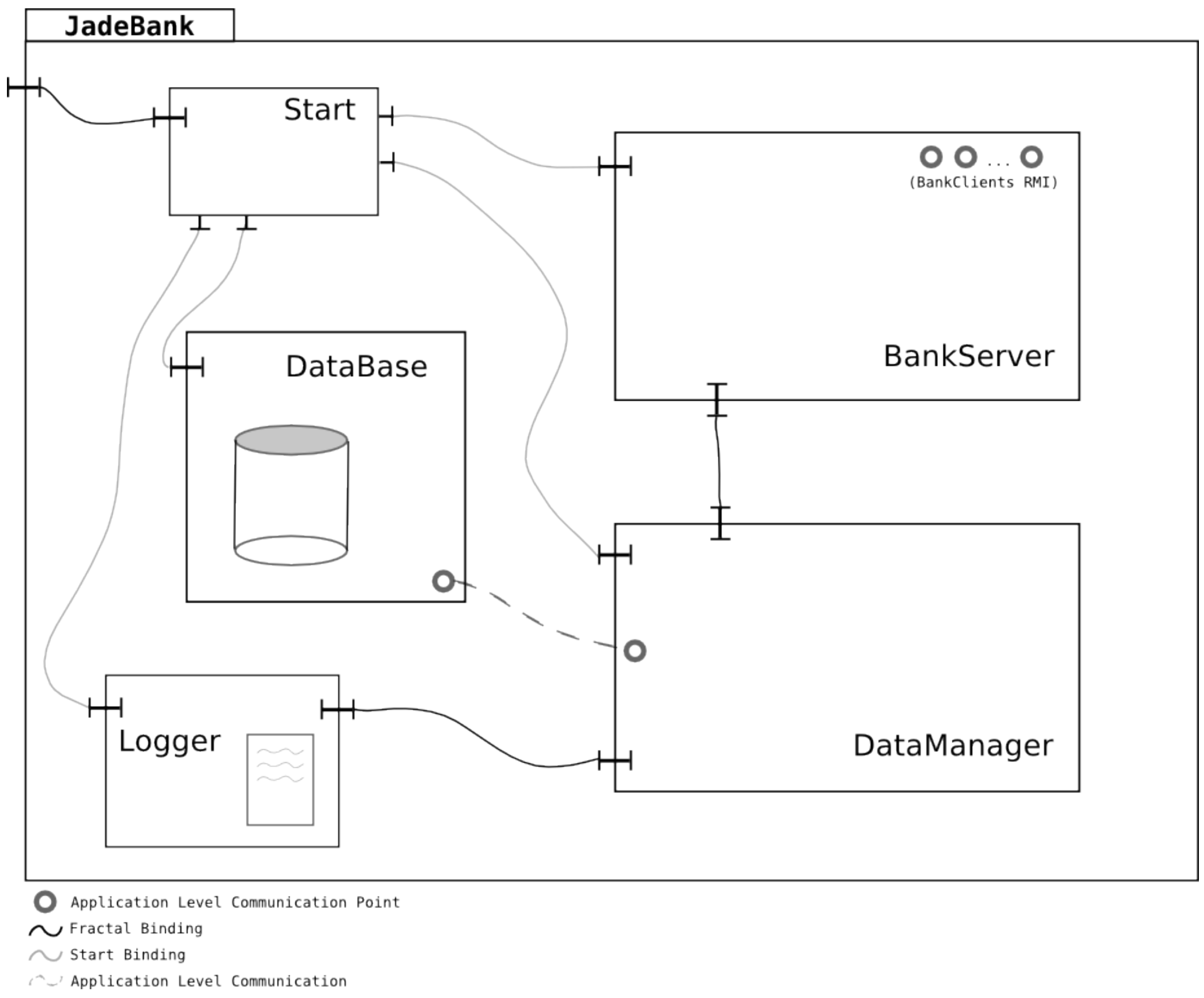


Figure 3.3. JadeBank Architecture

The Java documentation (javadoc) for the JadeBank application together with the Fractal ADL files is included in Appendix E.

3.3.3 Evaluation

The two approaches described in the previous subsection are quite different in many aspects, and both approaches are not suitable for all situations. The "legacy approach" is best suited when trying to run under JADE a closed-source application or one that is monolithic by definition, that can not be split into different components or would not benefit from it. The "legacy approach" can be considered easier than designing an application with JADE in mind since the application itself is already there and will not need to be modified. The process to bundle and wrap a legacy application so it is manageable by JADE is fairly systematic. But it has some drawbacks, like the lack of control over the granularity and modularity of the application, if the application originally is one single piece it will stay like this under JADE, deployed in one node and managed as such.

The "JADE approach" (designing with JADE in mind) makes the application much more flexible, allowing an application to be distributed over many nodes, making it modular and taking advantage of the benefits of modular design (component-based in the case of JADE) like extensibility and flexibility. But the "JADE approach" presents some serious inconveniences; it is in general more demanding and requires more time to create an application that runs under the JADE framework. This approach is also more difficult since it requires a better understanding of the JADE framework itself, but probably, its main drawback is the fact that the application's "entry point" has to be placed directly in the wrapper so it can access the benefits of the framework (i.e. Fractal RMI), and this is better suited to be done in Java.

In a real world example, we would take several legacy applications, write bundles and wrappers for them and bind them together; However, access to the benefits of Fractal RMI would then be restricted to inter-wrapper communications and the applications themselves would have to rely on application-level communications. Only if we were to write an application from scratch, to be managed by JADE, we could take full advantage of it, but it seems that if we were wrapping legacy application, communication between them would have to be done by different means using other networking protocols.

3.4 OVERHEAD

The JADE management framework adds an extra layer of significant magnitude to any application it manages, an overhead that will be analyzed in this section from three different perspectives: the overhead inherent to the Fractal model on which JADE is based, the amount of physical memory that the framework consumes and needs to manage applications as well as the amount of code that is necessary to enable an application so it can be managed under JADE.

3.4.1 Inherent Fractal Component Model Overhead

The Fractal component model is totally based on XML, a standardized way of transmitting and storing data, but at the same time a format that adds a considerable amount of bytes to any network transaction. XML files need to be parsed in order to become “machine ready”, a process that is known to be time consuming and inefficient [68]. The Fractal component model offers a wide range of possibilities, ranging from introspection to reconfiguration and management, but this comes at the price of performance and overhead. For example, a new component has to be created using a component factory, which is a component itself, that has to exist at all times so new components can be created; a process inherited from the Fractal component model that, as can be clearly seen, is intensive in resources.

The Fractal component model was described in section 2.2.

3.4.2 Memory Consumption

Another important aspect regarding overhead is the resources needed for running a JADE application. Nodes are typically mapped to real machines and hence the requirement is as simple as occupying machines in a cluster or wide-area network. A more interesting aspect is the amount of memory consumed by each node to run a JADE application, that is the memory used by JadeBoot and the running JadeNodes.

3.4.2.1 Test Scenario

The testing scenario includes running the JADE applications (or parts of them) described in the previous sections. Three of them – the CrappyHttpServer, the LegacyBank edited to use a remote database and the isolated MySQL – belong to the “wrapping legacy software” category. The last test is an edited version of JadeBank that uses a remote database. The choice to use a remote database is made in order to provide uniform test cases so that the results can be comparable.

Two kinds of tests are performed for each application: on the first one the whole JADE application is deployed and run on one node, whereas on the second, two nodes are used and the application is distributed among them. In the “wrapping legacy software” category, the first node is chosen as the one that always hosts the main application's component. The second node then, when present, will host the composite component and the helping ones such as the start component. In the "JADE-designed" category the JadeBank application is more distributed since there are more components involved.

3.4.2.2 Test Results

The first table (Table 5) presents the fairly constant memory consumption of the root node of the platform, the JadeBoot. Our tests showed that addition of nodes has a slight impact on JadeBoot as the memory increases for only 60KB on average per node addition.

	Memory Consumption in KB						
	no nodes	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**	**12 nodes**	**16 nodes**
JadeBoot	37,600	37,700	37,800	37,900	38,100	38,400	38,600

Table 5. JadeBoot's memory consumption

Table 6 presents the amount of memory occupied by JadeNodes and the applications themselves. Initially, before deployment, a JadeNode uses on average 29,000 KB of memory.
The table shows that regarding legacy applications a node addition occupies approximately 30,000KB of memory. The amount of memory that each node uses depends on the size of the component, that is the size of the application associated with it. This is more obvious on the MySQL example where due to the size of the MySQL bundle the node reaches the significant amount of 117,700KB. We expected this amount to decrease by the garbage collector once the deployment process was completed and the binaries were removed but this did not happen.

	Memory Consumption in KB		
	Application	**JadeNode 1**	**JadeNode 2**
CrappyHttpServer (1 node)	12,300	30,100	-
CrappyHttpServer (2 nodes)	12,300	30,200	29,800
LegacyBank with remote DB including RMI Registry (1 node)	36,200	31,000	-
LegacyBank with remote DB including RMI Registry (2 nodes)	36,200	31,000	29,800
MySQL (1 node)	14,600	117,700	-
MySQL (2 nodes)	14,600	117,700	29,600
JadeBank with remote DB including RMI Registry (1 node)	16,400	33,000	-
JadeBank with remote DB including RMI Registry (2 nodes)	16,400	31,800	30,600

Table 6. Applications' memory consumption under JADE

JadeBank falls into the second category; it was specifically designed with JADE in mind and there is no bundled software with it. The application is thus more distributed and this is also depicted by the memory consumption measurements. Since the application is run under the JADE container (from inside the wrapper) it was expected that the memory usage would be decreased despite the fact that the application's functionality has been expanded by introducing the DataManager and Logger components (see section 3.3.2).

Generally we believe that the JADE framework, as a typical Java application, requires a significant amount of memory even for trivial applications. A big portion of course is occupied by the framework itself and therefore we expect that more complex applications will not linearly increase the memory requirements. However it should be noted that JADE (revision 762) still lacks most of the self-management features (for instance self repair functionality needs to be integrated) that will definitely increase its memory requirements once they will be added.

3.4.3 Amount of Code

The bundles and wrappers that are necessary to make an application (either legacy or JADE-designed), be deployable and runnable under JADE require an amount of code that can not be easily estimated or measured since it is directly related to the application in consideration. If the application is what is considered as a "legacy application", the amount of code necessary to make it enabled for JADE is fairly constant and is basically divided into four sections of code briefly explained in Table 7 below.

Code Location	**Description**
Bundle's start method. Implements BundleActivator interface.	This portion of code is the one in charge of installing the code in the host machine. Installation usually consists of downloading the bundle from an Oscar Bundle Repository (OBR) and extracting the application's binaries to some accessible folder in the local machine.
Bundle's stop method. Implements BundleActivator interface.	Opposite to start, this part of code is the one that takes care uninstalling the binaries installed by the bundle's start method. This process usually includes removing the files and folders created by the start procedure.
Wrapper's start method. Implements Resource interface (optional).	This portion of the wrapper is the one in charge of executing the application when the start command is issued from BeanShell. The process consists of running some executable file, class or script that will start the application.
Wrapper's stop method. Implements Resource interface (optional).	Opposite to the start method, this portion of code stops the running process by killing it or taking some other action available.

Table 7. Four sections of code regarding JADE applications

The four methods described above are very much related to each other and are highly dependent on the type of the application under consideration. A generic approach does not exist for all cases as the type of application as well as what environment it is designed for, will influence the way it is installed/uninstalled and run/stopped. This procedure is very dependent on the type of application in case.

Applications designed directly with JADE in consideration will potentially include large amounts of code directly both in the bundle and wrapper methods, so, depending on the type of application (legacy or not), the amount of code will vary significantly and, thus, the amount of work will also change drastically. Therefore, it is not possible to give an acceptable estimate. It is true, however, that the amount of code required to adapt an application for Jade is not excessive and can be handled, and could be improved with some sort of tool that would generate code skeletons of all the necessary classes and interfaces since a big part of the adaptation process is fairly systematic.

3.5 Other observations

Besides the aspects specifically discussed above, several other issues should be taken into consideration for future versions of the framework. The first and probably most important flaw is the complete lack of documentation. Although there are guides for running the platform and writing wrapper for it, these guides are of no real value. The only useful document regarding Jade is [8], which introduces the framework and its basics, but focuses more on repair management than in the framework itself.

Another issue that seems to not have been considered is security, so we want to mention at this point that, even though more functional aspects of the framework should have higher priority, some security mechanisms should be implemented into the framework if it is ever to be used for any "real" purpose.

On another note, a minimal list of some discovered bugs and flaws is given in Appendix F.

Chapter 4

CONCLUSIONS

4.1 CONCLUSIONS

In this section, some conclusions covering the whole document will be drawn attempting to point out the most relevant aspects of the work done throughout the thesis project.

4.1.1 Literature Study

The problem of systems' management has grown with the complexity of modern systems and to cope with it, autonomous behavior of systems is the most widely accepted technique. In order to gain abstraction and reduce the newly introduced complexity of today's distributed solutions, component models are used. Component models, in order to be suitable for autonomous computing, should separate interfaces from implementation, be reflective and hierarchical, properties shared by the Fractal component model, the SOFA model and K-components but not fully by other models such as EJBs, .NET, CCM and CCA. Component models that support the three aforementioned desirable properties do set strong grounds for autonomic management frameworks to build on top of, and thus achieve some levels of autonomic management (self-configuration, self-healing, self-optimization and self-protection) like the JADE management framework or K-Components with Collaborative Reinforcement Learning.

4.1.2 The JADE Framework

The decision to use the Fractal component model seems to be a well based choice since the Fractal model is a stable and sufficiently supported model that enjoys important properties. It is both reflective and hierarchical preserving at the same time the beneficial characteristics of the basic component models, such as separation between interface and implementation and explicit architecture description. It seems therefore that Fractal is the most appropriate component model for creating self-manageable software. Also, the choice of Java as the programming language gives a strong advantage due to the characteristic of the language itself as platform-independent.

One of the most important parts of the framework itself is the failure detection mechanism. Our study showed that the mechanism used is trivial as it clearly depends on a reliable and robust network, implying a cluster solution. The failure detection mechanism should therefore be drastically improved in order to make JADE wide-area enabled. It is also important to notice that the JADE platform follows a very centralized design, with all nodes being managed by a single manager (JadeBoot); an architectural decision that can create communication bottlenecks at the central node and implies catastrophic consequences may the central manager fail. This negative scenario is considered in theory using replication mechanisms but not implemented in practice.

From the developer's perspective writing a JADE application is a relatively simple procedure as long as there is documentation and a programming guide or enough experience and familiarization with the framework. However, currently, there is no documentation at all and the available programmer's guide is incomplete if not misleading. We have also seen that the deployment process is arduous, time consuming, error prone and lacks dynamism. To facilitate development, an automated deployment process should be created since the procedure is quite systematic.

As regards application architecture, the capability of JADE to bundle and wrap legacy software is useful and gives the opportunity to use non open source software. However, JADE is better utilized when software is built specifically for the framework. In that case the major benefits are transparent communication between components through Fractal RMI, better flexibility and extensibility.

Finally, being a Java based project imposes a significant overhead in terms of memory consumption and performance. As our measurements showed the platform itself is not such a memory hog taking into account today's systems but there are obviously memory leaks after the deployment process starts.

In conclusion we believe that JADE is a sound attempt at a management framework based on strong principals, but as with anything in an immature state, it needs a lot of work to be considered a reliable solution. There are many flaws that need to be addressed and some fundamental decisions to be revised.

4.2 FUTURE WORK

In order to complete the assessment of the JADE framework, a study of the management capabilities of the framework must be performed. Under normal circumstances, this should have been part of the performed individual assessments. However, the self-management (self-repair) capabilities of the studied version of the JADE framework were not integrated, hence they could not be evaluated in practice. We expect the self-repair functionality to become available at some point, and in order to complete our assessment of the framework as a management framework, an evaluation following these basic guidelines could be performed:

- Build a test scenario comprised of multiple distributed JadeNodes interconnected (non-trivial environment).
- Manually fail a controlled set of JadeNodes and observe the failure detection mechanism realize these failures. Once the failure detections have happened, the central manager should take the appropriate actions in order to restore the system to a working, valid state.
- Discover the self-repair actions taken by the self-management module and verify that the new (automatically regenerated) environment is compatible with the original one.
- Report the actions taken by the self-management module and discuss their validity regarding self-repair.

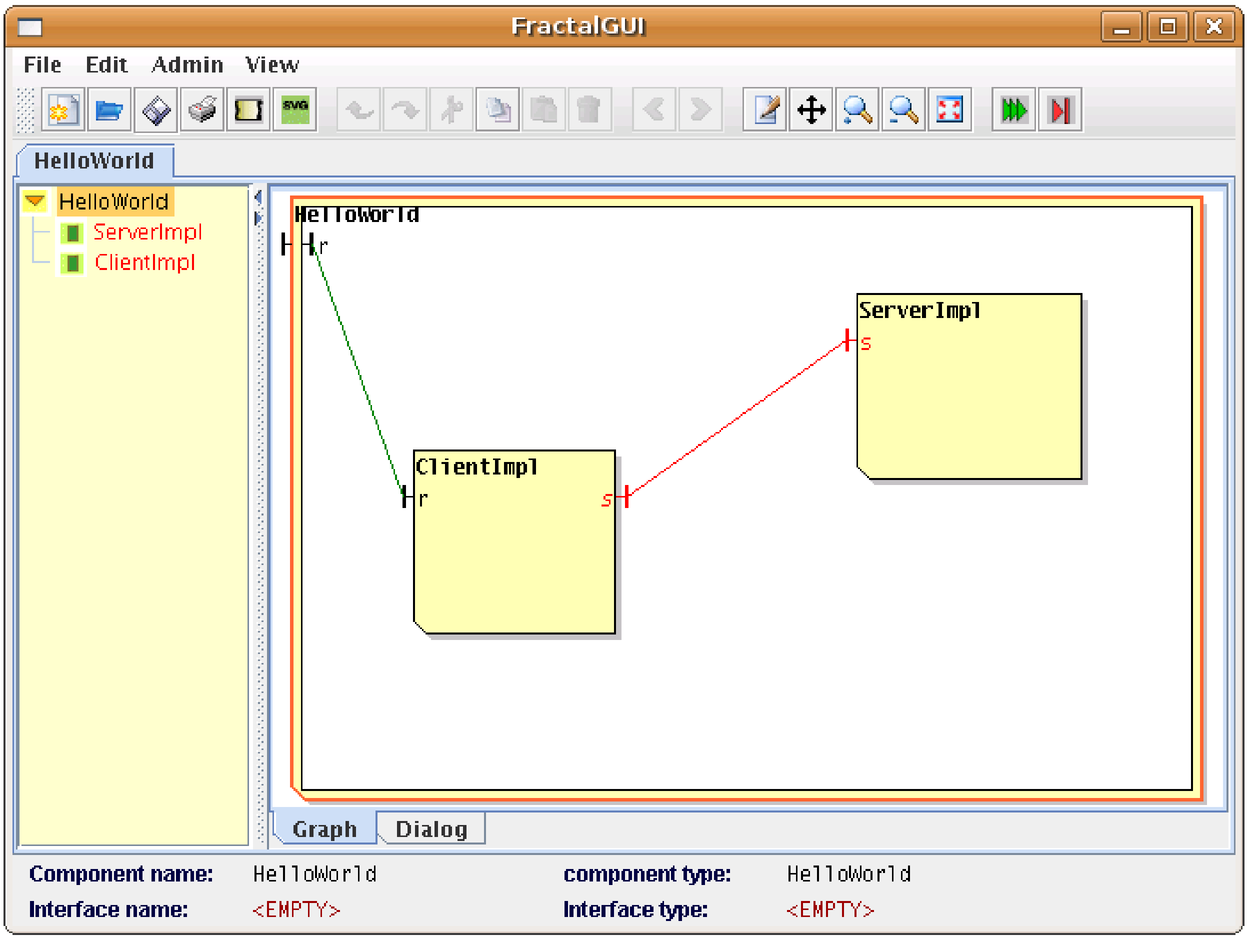
Once this assessment is complete, it can be stated that the the general assessment of the JADE framework is complete. As a side note, it is worth mentioning that the JADE framework, so far, only considers node failures and not application failures on the managed nodes, so the assessment of self-management capabilities would only cover node failures.

Furthermore, the work discussed in this thesis document could be expanded and enhanced by looking into the limitations imposed by the cluster topology assumptions, and more specifically the failure detection mechanism. Time could be spent researching the available failure detection mechanisms for wide-area environments and, certainly, propose a better suited algorithm for failure detection than the basic "heartbeat" approach used now by JADE.

Once the framework is refined by improving its failure detection mechanism, the repair manager is fully integrated and the platform becomes more stable, a battery of tests could be carried out with a relevant number of real nodes (physically distributed) to truly examine the framework's ability to perform under a wide-area network.

Appendix A: A Fractal HelloWorld Application

The design of the application is done using the Fractal GUI tool which produces the necessary Fractal files, the Java interfaces and skeleton classes.



Component name: HelloWorld   component type: HelloWorld
Interface name: <EMPTY>   Interface type: <EMPTY>

As already mentioned there is a composite component which includes two primitive ones. The external interface binds to the client's server interface. This is how the application is accessed. Furthermore, the client (ClientImpl) has a client interface that binds to the server's (ServerImpl) server interface.

The produced Fractal files are attached below. A common approach is to separate the interface definitions from the rest information such as bindings by putting them into different files. The Fractal GUI tool follows this approach.

```
<!-- HelloWorld.fractal. The composite component (extends HelloWorldType.fractal) -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition extends="HelloWorldType" name="HelloWorld">
```

```
  <component definition="ServerImpl" name="ServerImpl"/>
  <component definition="ClientImpl" name="ClientImpl"/>
  <binding client="this.r" server="ClientImpl.r"/>
  <binding client="ClientImpl.s" server="ServerImpl.s"/>
  <controller desc="composite"/>
  <coordinates color="-73" y0="0.14539894321348862"
       x1="0.891350832842055" y1="0.47045537662206655"
       name="ServerImpl" x0="0.6135862668491892"/>
  <coordinates color="-73" y0="0.4168030906370116"
       x1="0.3661714962074936" y1="0.7364419168221132"
       name="ClientImpl" x0="0.11123689454280883"/>
</definition>
```

```
<!-- HelloWorldType.fractal. Interface definition -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition name="HelloWorldType">
<interface signature="java.lang.Runnable" role="server" name="r"/>
</definition>
```

```
<!-- ClientImpl.fractal. The client primitive component (extends ClientImplType)-->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition extends="ClientImplType" name="ClientImpl">
<content class="ClientImpl"/>
<controller desc="primitive"/>
</definition>
```

```
<!-- ClientImplType.fractal. Interface definitions -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition name="ClientImplType">
<interface signature="java.lang.Runnable" role="server" name="r"/>
<interface signature="Service" role="client" name="s"/>
</definition>
```

```
<!-- ServerImpl.fractal. The server primitive component (extends ServerImplType) -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition extends="ServerImplType" name="ServerImpl">
<content class="ServerImpl"/>
<attributes signature="ServiceAttributes">
  <attribute value=">>" name="header"/>
</attributes>
<controller desc="primitive"/>
</definition>
```

```
<!-- ServerImplType.fractal. Interface definition -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
 "classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition name="ServerImplType">
  <interface signature="Service" role="server" name="s"/>
</definition>
```

The Java classes are included below combining the automatically produced skeletons with the implementation code. Some parts are censored in order to emphasize the important ones.

```
/**
 * Main.java. The starting point of the application (manually created class)
 */
public class Main {

  public static void main(String[] args) {
  ClientImpl client = new ClientImpl();
  client.start();
  }
}


/**
 * CilentImpl.java: The client's implementation class
 */
import org.objectweb.fractal.api.control.BindingController;
import java.lang.Runnable;

public class ClientImpl implements Runnable, BindingController {

  public final static String SERVICE_BINDING = "service";
  private Service service;

  public ClientImpl () {}

  // -----------------------------------------------------
  // Implementation of the BindingController interface
  // -----------------------------------------------------
  public String[] listFc () {
    return new String[] { SERVICE_BINDING };
  }
  public Object lookupFc (final String clientItfName) {
    if (SERVICE_BINDING.equals(clientItfName)) {
      return service;
    }
    return null;
  }
  public void bindFc (final String clientItfName, final Object serverItf) {
    if (SERVICE_BINDING.equals(clientItfName)) {
      service = (Service)serverItf;
    }
  }
  public void unbindFc (final String clientItfName) {
    if (SERVICE_BINDING.equals(clientItfName)) {
      service = null;
    }
  }

  // -----------------------------------------------------
  // Implementation of the Runnable interface
  // -----------------------------------------------------
  public void run () {
      service.print("hello world");
  }
}


/**
 * ServerImpl.java: The server's implementation class
 * (implements the Service and indirectly the AttributeController)
 */
public class ServerImpl implements Service, ServiceAttributes {
  private String header = "";
  private int count = 0;

  public ServerImpl () {
    System.out.println("ServerImpl started");
  }

  public void print(String msg) {
    for(int i=0; i<count; i++) {
      System.err.println(header + msg);
    }
  }

  public String getHeader() { return(this.header); }

  public void setHeader(final String msg) { this.header = msg; }
}


/*
 * ServiceAttributes.java: The server's AttributeController interface
 */
import org.objectweb.fractal.api.control.AttributeController;

public interface ServiceAttributes extends AttributeController {
  String getHeader();
  void setHeader(String header);
}


/**
 * Service.java: The server's service interface
 */
public interface Service {
  void print(String msg);
}
```

Appendix B: The CrappyHttpServer Application (Oscar Bundle)

The Oscar bundle described in this section is a simple HTTP Server. However, instead of implementing all the functionality inside the bundle, a different approach has been chosen. The HTTP server is implemented as a legacy software that will be included in a binary form (zipped) inside the Oscar bundle.

Developing an Oscar bundle actually means implementing the `BundleActivator` interface of the `org.osgi.framework` package. The interface has two methods: `start()` and `stop()`. `Start` will be used to run the application and `stop` to stop it. In this specific case `start` will also include the code for unpacking the legacy software (included in the bundle as a TGZ archive) and installing it to the appropriate folder.

The source code and the Java documentation produced from the two projects (the HTTP Server and the Oscar bundle) is shown in the pages that follow.

Package Summary	
sics.jade.osgi.chttpd	

Package sics.jade.osgi.chttpd

Class Summary	
Activator	The bundle activator that installs and runs the Crappy HTTP Server.

Class Activator

sics.jade.osgi.chttpd

```
java.lang.Object
  └ sics.jade.osgi.chttpd.Activator
```

public class **Activator**
extends Object

The bundle activator that installs the Crappy HTTP Server. The precompiled CrappyHttpServer is included in a TGZ file. This bundle is intended to be used in OSCAR, not JADE.

Author:
Simón af Frosterus & Leonidas Pantzopoulos @ SICS

Version:
1.0

Constructor Summary
`Activator()`

Method Summary	
void	`start(BundleContext context)` Implements BundleActivator.start().
void	`stop(BundleContext context)` Implements BundleActivator.stop().

Package Summary	
seds.thesis.chttpd	

Package seds.thesis.chttpd

Class Summary	
CHttpd	A simple HTTP server that handles GET and POST requests.
CHttpdConf	A crappy HTTP server conf parser.
Test	A class for testing the application's functionality (mostly used for debugging).

Class CHttpd

seds.thesis.chttpd

```
java.lang.Object
  └seds.thesis.chttpd.CHttpd
```

---

public class **CHttpd**
extends Object

A simple HTTP server that handles GET and POST requests.

Author:
Simón af Frosterus & Leonidas Pantzopoulos @ SICS

Version:
1.0

Constructor Summary
`CHttpd()`

Method Summary	
`static void`	`main(String[] args)` Main method of the application.

Class CHttpdConf

seds.thesis.chttpd

```
java.lang.Object
  └seds.thesis.chttpd.CHttpdConf
```

---

public class **CHttpdConf**
extends Object

A crappy HTTP server conf parser.

Author:
Simón af Frosterus & Leonidas Pantzopoulos @ SICS

Version:
1.0

Constructor Summary
CHttpdConf()   Initializes a CHttpdConf object and tries to locate the default configuration file.
CHttpdConf(String conf_file)   Initializes a CHttpdConf object with the given configuration file.

Method Summary	
`String`	**getAbsolutDocumentRoot**() Gets the path of the folder the document root points to (absolut path)
`String[]`	**getDefaultFiles**() Gets default files that the server will try to serve if no input is given.
`String`	**getDocumentRoot**() Gets the path of the folder the document root points to (relative to SERVER_ROOT).
`int`	**getPort**() Gets the port the server is currently listening on.
`void`	**setDefaultFiles**(String[] files) Sets default files in proporties file that the server will try to serve if no input is given (e.g. index.html, index.htm)
`void`	**setDocumentRoot**(String root) Sets document_root value in properties file
`void`	**setPort**(int port) Sets the port value in properties file.

APPENDIX C: THE CRAPPYHTTPSERVER JADE APPLICATION (LEGACY)

```
<!-- Chttpd.fractal -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://fr/jade/service/deployer/adl/xml/jade.dtd">

<definition name="Chttpd">
  <interface name="service" role="server" signature="fr.jade.service.Service" />

  <!-- ================================== -->
  <!--               START                -->
  <!-- ================================== -->
  <component name="start" definition="fr.jade.resource.start.StartType">
    <virtual-node name="node0" />
  </component>

  <!-- ================================== -->
  <!--               CHTTPD               -->
  <!-- ================================== -->
  <component name="chttpd" definition="sics.jade.resource.chttpd.ChttpdResourceType">
    <attributes signature="fr.jade.fractal.api.control.GenericAttributeController">
      <attribute name="resourceName" value="chttpd" />
      <attribute name="dirLocal" value="/tmp/chttpd/" />
      <attribute name="dirInstall" value="chttpd/" />
      <attribute name="port" value="8080" />
      <attribute name="document_root" value="htdocs" />
      <attribute name="default_files" value="index.html" />
    </attributes>
    <virtual-node name="node1" />
    <packages>
      <package name="Crappy HTTP Server">
        <property name="dir.local" value="/tmp/chttpd/" />
        <property name="dir.install" value="chttpd/" />
      </package>
    </packages>
  </component>

  <!-- ================================== -->
  <!--              BINDING               -->
  <!-- ================================== -->
  <binding client="this.service" server="start.service" />
  <binding client="start.rsrc_chttpd" server="chttpd.resource" />
  <!-- start.rsrc_* is equal to start.rsrc. The part after "rsrc" is just ignored. -->

  <virtual-node name="node0" />

</definition>


<!-- sics.jade.resource.chttpd.ChttpdResourceType.fractal -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://fr/jade/service/deployer/adl/xml/jade.dtd">

<definition name="sics.jade.resource.chttpd.ChttpdResourceType">

  <interface name="resource" signature="fr.jade.resource.Resource" role="server"
cardinality="collection" contingency="optional" />

  <content class="sics.jade.osgi.chttpd.wrapper.CrappyHttpServerWrapper" />
  <controller desc="parametricprimitive" />

</definition>
```

```
<!-- fr.jade.resource.start.StartType.fractal -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://fr/jade/service/deployer/adl/xml/jade.dtd">


<definition name="fr.jade.resource.start.StartType">
  <interface name="service" role="server" signature="fr.jade.service.Service" />
  <interface name="rsrc" role="client"signature="fr.jade.resource.Resource"
    cardinality="collection" contingency="optional" />

  <content class="fr.jade.resource.start.Start" />

</definition>
```

Package Summary	
sics.jade.osgi.chttpd.wrapper	

Package sics.jade.osgi.chttpd.wrapper

Class Summary	
Activator	
CrappyHttpServerWrapper	

Class Activator

sics.jade.osgi.chttpd.wrapper

```
java.lang.Object
  └sics.jade.osgi.chttpd.wrapper.Activator
```

---

public class **Activator**
extends Object

Constructor Summary
`Activator()`

Method Summary	
`void`	`start(BundleContext context)`
`void`	`stop(BundleContext context)`

Class CrappyHttpServerWrapper

sics.jade.osgi.chttpd.wrapper

```
java.lang.Object
  └sics.jade.osgi.chttpd.wrapper.CrappyHttpServerWrapper
```

---

public class **CrappyHttpServerWrapper**
extends Object

Constructor Summary
`CrappyHttpServerWrapper()` Constructor

Method Summary	
`void`	`bindFc(String itfName, Object itfValue)`

void	**configure**() Configure a Chttpd resource on the local machine.
String	**fixDir**(String dir)
String	**getAttribute**(String name)
String[]	**listFc**()
String[]	**listFcAtt**()
void	**loadApp**() Load an application on that Chttpd resource.
Object	**lookupFc**(String itfName)
void	**setAttribute**(String name, String value)
void	**start**() Start the resource. (=> run)
void	**stop**()
void	**unbindFc**(String itfName)

Package Summary	
sics.jade.osgi.chttpd	

Package sics.jade.osgi.chttpd

Class Summary	
Activator	

Class Activator

sics.jade.osgi.chttpd

```
java.lang.Object
  └sics.jade.osgi.chttpd.Activator
```

---

public class **Activator**
extends Object

Author:
: Leo & Smn @ SICS The bundle activator that installs the Crappy HTTP Server. The precompiled CrappyHttpServer is included in a TGZ file.

Constructor Summary
`Activator()`

Method Summary	
void	`start(BundleContext context)` Implements BundleActivator.start().
void	`stop(BundleContext context)` Implements BundleActivator.stop().

APPENDIX D: THE BANK JADE APPLICATION (LEGACY)

```
<!-- BankServer -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://fr/jade/service/deployer/adl/xml/jade.dtd">

<definition name="BankServer">

  <interface name="service" role="server" signature="fr.jade.service.Service" />

  <!-- ================================== -->
  <!--               START                -->
  <!-- ================================== -->
  <component name="start" definition="fr.jade.resource.start.StartType">
    <virtual-node name="node0" />
  </component>

  <!-- ================================== -->
  <!--            BANK SERVER             -->
  <!-- ================================== -->
  <component name="bankserver"
definition="sics.jade.resource.bankserver.BankServerResourceType">
    <attributes signature="fr.jade.fractal.api.control.GenericAttributeController">
      <attribute name="resourceName" value="bankserver" />
      <attribute name="dirLocal" value="/tmp/bankserver/" />
      <attribute name="dirInstall" value="bankserver/" />
      <attribute name="rmiport" value="1099" />
      <attribute name="rmihost" value="deus.sics.se" />

      <attribute name="bankname" value="6Bank" />
      <attribute name="dbhost" value="tenerife.sics.se" />
      <attribute name="dbport" value="3306" />
      <attribute name="dbname" value="banks" />
      <attribute name="dbusername" value="root" />
      <attribute name="dbpassword" value="12345" />

    </attributes>
    <virtual-node name="node0" />
    <packages>
      <package name="Bank Server">
        <property name="dir.local" value="/tmp/bankserver/" />
        <property name="dir.install" value="bankserver/" />
      </package>
    </packages>
  </component>

  <!-- ================================== -->
  <!--             BINDING                -->
  <!-- ================================== -->
  <binding client="this.service" server="start.service" />
  <binding client="start.rsrc_bankserver" server="bankserver.resource" />

  <virtual-node name="node0" />

</definition>



<!-- sics.jade.resource.bankserver.BankServerResourceType -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://fr/jade/service/deployer/adl/xml/jade.dtd">
```

```
<definition name="sics.jade.resource.bankserver.BankServerResourceType">

  <interface name="resource" signature="fr.jade.resource.Resource" role="server"
cardinality="collection"
    contingency="optional" />

  <content class="sics.jade.osgi.bankserver.wrapper.BankServerWrapper" />
  <controller desc="parametricprimitive" />

</definition>
```

Package Summary	
sics.jade.osgi.bankserver.wrapper	

Package sics.jade.osgi.bankserver.wrapper

Class Summary	
Activator	
BankServerWrapper	

Class Activator

sics.jade.osgi.bankserver.wrapper

```
java.lang.Object
  └sics.jade.osgi.bankserver.wrapper.Activator
```

---

public class **Activator**
extends Object

Constructor Summary
`Activator()`

Method Summary	
`void`	`start(BundleContext context)`
`void`	`stop(BundleContext context)`

Class BankServerWrapper

sics.jade.osgi.bankserver.wrapper

```
java.lang.Object
  └sics.jade.osgi.bankserver.wrapper.BankServerWrapper
```

---

public class **BankServerWrapper**
extends Object

Constructor Summary
`BankServerWrapper()` Constructor

Method Summary

void	**bindFc**(String itfName, Object itfValue)
void	**configure**() Configure a Bank Server resource on the local machine.
String	**fixDir**(String dir)
String	**getAttribute**(String name)
String[]	**listFc**()
String[]	**listFcAtt**()
void	**loadApp**() Load an application on that Bank Server resource.
Object	**lookupFc**(String itfName)
void	**setAttribute**(String name, String value)
void	**start**() Start the resource. (=> run)
void	**stop**()
void	**unbindFc**(String itfName)

Package Summary	
sics.jade.osgi.bankserver	

Package sics.jade.osgi.bankserver

Class Summary	
Activator	The bundle activator that installs the Bank Server.

Class Activator

sics.jade.osgi.bankserver

```
java.lang.Object
  └sics.jade.osgi.bankserver.Activator
```

---

public class **Activator**
extends Object

The bundle activator that installs the Bank Server. The precompiled BankServer is included in a TGZ file.

Author:
Simón af Frosterus & Leonidas Pantzopoulos

Constructor Summary
Activator()

Method Summary	
void	start(BundleContext context) Implements BundleActivator.start().
void	stop(BundleContext context) Implements BundleActivator.stop().

Package Summary	
<unnamed>	

Package <unnamed>

Interface Summary	
Bank	The Bank interface.

Class Summary	
AccountImpl	The Account servant (that implements the Account remote interface).
BankImpl	The Bank servant (that implements the Bank remote interface).
BankServerConf	A bank server conf parser.
Client	The Client.
Server	The Server.

Exception Summary	
Rejected	The Exception thrown when an action is rejected.

Class AccountImpl

<unnamed>

```
java.lang.Object
  └java.rmi.server.RemoteObject
      └java.rmi.server.RemoteServer
          └java.rmi.server.UnicastRemoteObject
              └AccountImpl
```

All Implemented Interfaces:

Remote, Serializable

---

public class **AccountImpl**
extends UnicastRemoteObject

The Account servant (that implements the Account remote interface).

Title: Bank Manager Application

Description: Bank Manager Application

Author:

Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:

1.1

Constructor Summary
AccountImpl(String bankname, String name, float balance, Connection connection) Constructs an Account object.
AccountImpl(String bankname, String name, Connection connection) Constructs an Account object with zero balance.

Method Summary	
float	**balance**() Interface implementation.
synchronized void	**deposit**(float value) Interface implementation.
String	**getName**() Interface implementation.
String	**getString**() Interface implementation.
synchronized void	**withdraw**(float value) Interface implementation.

Interface Bank

<unnamed>

All Superinterfaces:

Remote

All Known Implementing Classes:

BankImpl

---

public interface **Bank**
extends Remote

The Bank interface.

Title: Bank Manager Application

Description: Bank Manager Application

Author:

Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:

1.1

Method Summary	
`boolean`	**`deleteAccount`**`(Account acc)` Deletes the specified Account.
`Account`	**`getAccount`**`(String name)` Gets the Account with the given name.
`Account`	**`newAccount`**`(String name)` Creates a new Account.

Class BankImpl

<unnamed>

```
java.lang.Object
  └java.rmi.server.RemoteObject
      └java.rmi.server.RemoteServer
          └java.rmi.server.UnicastRemoteObject
              └BankImpl
```

All Implemented Interfaces:

Bank, Remote, Serializable

---

public class **BankImpl**
extends UnicastRemoteObject
implements Bank

The Bank servant (that implements the Bank remote interface).

Title: Bank Manager Application

Description: Bank Manager Application

Author:

Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:

1.1

Constructor Summary
`BankImpl``(String dbhost, int dbport, String dbname, String bankname, String dbuser, String dbpassword)` Initializes a Bank object.

Method Summary	
`boolean`	**`deleteAccount`**`(Account acc)` Interface implementation.
`Account`	**`getAccount`**`(String name)` Interface implementation.

Account	**newAccount**(String name) Interface implementation.

Methods inherited from interface Bank
deleteAccount, getAccount, newAccount

Class BankServerConf

<unnamed>

```
java.lang.Object
  └ BankServerConf
```

public class **BankServerConf**
extends Object

A bank server conf parser.

Title: Bank Manager Application

Description: Bank Manager Application

Author:
Simón af Frosterus & Leonidas Pantzopoulos

Version:
1.1

Constructor Summary
BankServerConf() Initializes a BankServerConf object and tries to locate the default configuration file.
BankServerConf(String conf_file) Initializes a BankServerConf object with the given configuration file.

Method Summary	
String	**getBankName**() Return the Bank name (also used in the RMI URL).
String	**getDataBaseHost**() Return the DataBase host
String	**getDataBaseName**() Return the DataBase name
String	**getDataBasePassword**() Return the DataBase Password
int	**getDataBasePort**() Return the DataBase port

String	getDataBaseUserName() Return the DataBase username
String	getRMIHost() Return the RMI host
int	getRMIPort() Return the RMI port the RMI Registry is currently listening on
static void	main(String[] args) TEST MAIN - No purpose at all
void	setBankName(String name)
void	setDataBaseHost(String host) Set DataBase host name value in properties file
void	setDataBaseName(String name) Set DataBase name value in properties file
void	setDataBasePassword(String password) Set DataBase password value in properties file
void	setDataBasePort(int port) Set DataBase port name value in properties file
void	setDataBaseUserName(String username) Set DataBase user name value in properties file
void	setRMIHost(String host) Set RMI host name value in properties file
void	setRMIPort(int port) Set the RMI port value in properties file

Class Client

<unnamed>

```
java.lang.Object
  └Client
```

public class **Client**
extends Object

The Client.

Title: Bank Manager Application

Description: Bank Manager Application

Author:
Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:
1.1

Constructor Summary

`Client(String[] args)` Creates a Clent object and connects to the Bank.

Method Summary

`protected void`	`execute(Command command)` Executes a given Command.
`static void`	`main(String[] args)` Main method.
`protected Command`	`parse(String str)` Parses input commands.
`void`	`run()` Run method.

Class Rejected

<unnamed>

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.io.IOException
              └java.rmi.RemoteException
                  └Rejected
```

All Implemented Interfaces:

Serializable

public class **Rejected**
extends RemoteException

The Exception thrown when an action is rejected.

Title: Bank Manager Application

Description: Bank Manager Application

Author:

Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:

1.1

Constructor Summary

Rejected()
Rejected(String reason)

Class Server

<unnamed>

```
java.lang.Object
  └ Server
```

public class **Server**
extends Object

The Server.

Title: Bank Manager Application

Description: Bank Manager Application

Author:
Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:
1.1

Constructor Summary
Server(String[] args) Creates a Server object and registers a Bank with the given name.

Method Summary	
static void	**main**(String[] args) Main method.

APPENDIX E: THE BANK JADE APPLICATION (JADE-DESIGNED)

```
<!--  JadeBank -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://fr/jade/service/deployer/adl/xml/jade.dtd">

<definition name="JadeBank">

  <interface name="service" role="server" signature="fr.jade.service.Service" />

  <!-- ================================== -->
  <!--              START                 -->
  <!-- ================================== -->
  <component name="start" definition="fr.jade.resource.start.StartType">
    <virtual-node name="node0" />
    <!--
      Host is relative to jadeboot's naming.
      For instance with host="localhost" and number="1" it will try to deploy
      to the node registered in jadeboot as "localhost_1".
    -->
  </component>

  <!-- ================================== -->
  <!--          JADE BANK SERVER          -->
  <!-- ================================== -->
  <component name="jadebankserver"
definition="sics.jade.resource.jadebankserver.JadeBankServerResourceType">
    <attributes signature="fr.jade.fractal.api.control.GenericAttributeController">
      <attribute name="resourceName" value="jadebankserver" />
      <attribute name="dirLocal" value="/tmp/jadebank/" />
      <attribute name="dirInstall" value="jadebankserver/" />
       <attribute name="rmihost" value="localhost" />
      <attribute name="rmiport" value="1099" />

      <attribute name="bankname" value="6Bank" />
      <attribute name="dbhost" value="tenerife.sics.se" />
      <attribute name="dbport" value="3306" />
      <attribute name="dbname" value="banks" />
      <attribute name="dbusername" value="root" />
      <attribute name="dbpassword" value="" />

       <!-- max number of retries to attempt to connect to the datasource -->
       <attribute name="dbConAttempts" value="10" />
       <!-- sleep time in msec between each retry -->
      <attribute name="dbConSleepTime" value="500" />
       <!-- increase factor to apply to sleeptime between each attempt
       (sleeptime of 1000 and factor of 1 will retry every 1 sec) -->
      <attribute name="dbConSleepFactor" value="1.5" />
    </attributes>
    <virtual-node name="node0" />
    <packages>
      <package name="Jade Bank Server">
        <property name="dir.local" value="/tmp/jadebank/" />
        <property name="dir.install" value="jadebankserver/" />
      </package>
    </packages>
  </component>

  <!-- ================================== -->
  <!--           DATA MANAGER             -->
  <!-- ================================== -->
  <component name="datamanager"
definition="sics.jade.resource.datamanager.DataManagerResourceType">
```

```
    <attributes signature="fr.jade.fractal.api.control.GenericAttributeController">
      <attribute name="resourceName" value="datamanager" />
      <attribute name="dirLocal" value="/tmp/jadebank/" />
      <attribute name="dirInstall" value="datamanager/" />
    </attributes>
    <virtual-node name="node0" />
    <packages>
      <package name="Data Manager">
        <property name="dir.local" value="/tmp/jadebank/" />
        <property name="dir.install" value="datamanager/" />
      </package>
    </packages>
  </component>

  <!-- ================================== -->
  <!--          JADE BANK LOGGER          -->
  <!-- ================================== -->
  <component name="jadebanklogger"
definition="sics.jade.resource.jadebanklogger.JadeBankLoggerResourceType">
    <attributes signature="fr.jade.fractal.api.control.GenericAttributeController">
      <attribute name="resourceName" value="jadebanklogger" />
      <attribute name="dirLocal" value="/tmp/jadebank/" />
      <attribute name="fileName" value="JadeBank.log" />
    </attributes>
    <virtual-node name="node0" />
    <packages>
      <package name="Jade Bank Logger">
        <property name="dir.local" value="/tmp/jadebank/" />
      </package>
    </packages>
  </component>

  <!-- ================================== -->
  <!--                MYSQL               -->
  <!-- ================================== -->
  <component name="mysql" definition="sics.jade.resource.mysql.MySqlResourceType">
    <attributes signature="fr.jade.fractal.api.control.GenericAttributeController">
      <attribute name="resourceName" value="mysql" />
      <attribute name="dirLocal" value="/tmp/jadebank/" />
      <attribute name="dirInstall" value="mysql/" />
      <attribute name="user" value="root" />
      <attribute name="password" value="" />
    </attributes>
    <virtual-node name="node1" />
    <packages>
      <package name="MySql Server">
        <property name="dir.local" value="/tmp/jadebank/" />
        <property name="dir.install" value="mysql/" />
      </package>
    </packages>
  </component>

  <!-- ================================== -->
  <!--             BINDINGS               -->
  <!-- ================================== -->
  <binding client="this.service" server="start.service" />
  <binding client="start.rsrc_jadebankserver" server="jadebankserver.resource" />
  <binding client="start.rsrc_datamanager" server="datamanager.resource" />
  <binding client="start.rsrc_jadebanklogger" server="jadebanklogger.resource" />
  <binding client="start.rsrc_mysql" server="mysql.resource" />

  <!-- bind the jadebankserver to the datamanager -->
  <binding client="jadebankserver.dataservice" server="datamanager.dataservice" />
  <!-- bind the datamanager to the jadebanklogger -->
  <binding client="datamanager.datalogger" server="jadebanklogger.datalogger" />

  <virtual-node name="node0" />

</definition>
```

```
<!--  sics.jade.resource.jadebankserver.JadeBankServerResourceType -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://fr/jade/service/deployer/adl/xml/jade.dtd">

<definition name="sics.jade.resource.jadebankserver.JadeBankServerResourceType">
  <interface name="resource" signature="fr.jade.resource.Resource" role="server"
cardinality="collection" contingency="optional" />
  <interface name="dataservice" role="client"
signature="sics.jade.resource.data.DataService" contingency="optional" />

  <content class="sics.jade.osgi.jadebankserver.wrapper.JadeBankServerWrapper" />
  <controller desc="parametricprimitive" />
</definition>


<!--  sics.jade.resource.jadebanklogger.JadeBankLoggerResourceType -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://fr/jade/service/deployer/adl/xml/jade.dtd">

<definition name="sics.jade.resource.jadebanklogger.JadeBankLoggerResourceType">
  <interface name="resource" signature="fr.jade.resource.Resource" role="server"
cardinality="collection" contingency="optional" />
  <interface name="datalogger" role="server"
signature="sics.jade.resource.data.DataLogger" contingency="optional" />

  <content class="sics.jade.osgi.jadebanklogger.wrapper.JadeBankLoggerWrapper" />
  <controller desc="parametricprimitive" />

</definition>


<!--  sics.jade.resource.datamanager.DataManagerResourceType -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://fr/jade/service/deployer/adl/xml/jade.dtd">

<definition name="sics.jade.resource.datamanager.DataManagerResourceType">
  <interface name="resource" signature="fr.jade.resource.Resource" role="server"
cardinality="collection" contingency="optional" />
  <interface name="dataservice" role="server"
signature="sics.jade.resource.data.DataService" contingency="optional" />
  <interface name="datalogger" role="client"
signature="sics.jade.resource.data.DataLogger" contingency="optional" />

  <content class="sics.jade.osgi.datamanager.wrapper.DataManagerWrapper" />
  <controller desc="parametricprimitive" />
</definition>


<!--  sics.jade.resource.mysql.MySqlResourceType -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://fr/jade/service/deployer/adl/xml/jade.dtd">

<definition name="sics.jade.resource.mysql.MySqlResourceType">
  <interface name="resource" signature="fr.jade.resource.Resource" role="server"
cardinality="collection"
    contingency="optional" />

  <content class="sics.jade.osgi.mysql.wrapper.MySqlWrapper" />
  <controller desc="parametricprimitive" />
</definition>
```

Package Summary	
sics.jade.osgi.jadebankserver.wrapper	

Package sics.jade.osgi.jadebankserver.wrapper

Class Summary	
Activator	
JadeBankServerWrapper	

Class Activator

sics.jade.osgi.jadebankserver.wrapper

```
java.lang.Object
  └sics.jade.osgi.jadebankserver.wrapper.Activator
```

---

public class **Activator**
extends Object

Constructor Summary
`Activator()`

Method Summary	
`void`	`start(BundleContext context)`
`void`	`stop(BundleContext context)`

Class JadeBankServerWrapper

sics.jade.osgi.jadebankserver.wrapper

```
java.lang.Object
  └sics.jade.osgi.jadebankserver.wrapper.JadeBankServerWrapper
```

---

public class **JadeBankServerWrapper**
extends Object

Constructor Summary
`JadeBankServerWrapper()`   Constructor

Method Summary

`void`	**bindFc**`(String itfName, Object itfValue)`
`void`	**configure**`()` Configure a Jade Bank Server resource on the local machine.
`String`	**fixDir**`(String dir)`
`String`	**getAttribute**`(String name)`
`String[]`	**listFc**`()`
`String[]`	**listFcAtt**`()`
`void`	**loadApp**`()` Load additional applications on that Jade Bank Server resource.
`Object`	**lookupFc**`(String itfName)`
`void`	**setAttribute**`(String name, String value)`
`void`	**start**`()` Start the resource. (=> run)
`void`	**stop**`()` Stop the resource.
`void`	**unbindFc**`(String itfName)`

Package Summary	
sics.jade.jadebank	
sics.jade.osgi.jadebankserver	

Package sics.jade.jadebank

Interface Summary	
Account	The Account interface.
Bank	The Bank interface.

Class Summary	
AccountImpl	The Account servant (that implements the Account remote interface).
BankImpl	The Bank servant (that implements the Bank remote interface).
Client	The Client.

Exception Summary	
Rejected	The Exception thrown when an action is rejected.

Interface Account

sics.jade.jadebank

All Superinterfaces:
Remote

All Known Implementing Classes:
AccountImpl

---

public interface **Account**
extends Remote

The Account interface.

Title: Jade Bank Manager Application

Description: Jade Bank Manager Application

Author:
Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:
1.2

Method Summary

float	**balance**() Gets the balance of the account.
void	**deposit**(float value) Deposits the selected amount into the account.
String	**getName**() Gets the name associated with the Account.
String	**getSummary**() Gets a text summary of the Account (name, balance, etc).
void	**withdraw**(float value) Withdraws the selected amount from the account.

Class AccountImpl

sics.jade.jadebank

```
java.lang.Object
  └java.rmi.server.RemoteObject
      └java.rmi.server.RemoteServer
          └java.rmi.server.UnicastRemoteObject
              └sics.jade.jadebank.AccountImpl
```

All Implemented Interfaces:

Account, Remote, Serializable

---

public class **AccountImpl**
extends UnicastRemoteObject
implements Account

The Account servant (that implements the Account remote interface).

Title: Jade Bank Manager Application

Description: Jade Bank Manager Application

Author:

Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:

1.2

Constructor Summary
AccountImpl(String name, DataService ds) Constructs an Account object.

Method Summary

float	**balance**() Interface implementation.
synchronized void	**deposit**(float value) Interface implementation.
String	**getName**() Interface implementation.
String	**getSummary**() Interface implementation.
synchronized void	**withdraw**(float value) Interface implementation.

Methods inherited from interface sics.jade.jadebank.Account
balance, deposit, getName, getSummary, withdraw

Interface Bank

sics.jade.jadebank

All Superinterfaces:
Remote

All Known Implementing Classes:
BankImpl

---

public interface **Bank**
extends Remote

The Bank interface.

Title: Jade Bank Manager Application

Description: Jade Bank Manager Application

Author:
Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:
1.2

Method Summary	
boolean	**deleteAccount**(Account acc) Deletes the specified Account.
Account	**getAccount**(String name) Gets the Account with the given name.

Account	**newAccount**(String name) Creates a new Account.

Class BankImpl

sics.jade.jadebank

```
java.lang.Object
  └java.rmi.server.RemoteObject
      └java.rmi.server.RemoteServer
          └java.rmi.server.UnicastRemoteObject
              └sics.jade.jadebank.BankImpl
```

All Implemented Interfaces:

Bank, Remote, Serializable

---

public class **BankImpl**
extends UnicastRemoteObject
implements Bank

The Bank servant (that implements the Bank remote interface).

Title: Jade Bank Manager Application

Description: Jade Bank Manager Application

Author:

Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:

1.2

Constructor Summary
BankImpl(DataService ds, String bankname) Initializes a Bank object.

Method Summary	
boolean	**deleteAccount**(Account acc) Interface implementation.
Account	**getAccount**(String name) Interface implementation.
Account	**newAccount**(String name) Interface implementation.

Methods inherited from interface sics.jade.jadebank.Bank
deleteAccount, getAccount, newAccount

Class Client

sics.jade.jadebank

```
java.lang.Object
  └sics.jade.jadebank.Client
```

public class **Client**
extends Object

The Client.

Title: Jade Bank Manager Application

Description: Jade Bank Manager Application

Author:
Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)

Version:
1.2

Constructor Summary
Client(String[] args) Creates a Clent object and connects to the Bank.

Method Summary	
protected void	**execute**(sics.jade.jadebank.Command command) Executes a given Command.
static void	**main**(String[] args) Main method.
protected sics.jade.jadebank.Command	**parse**(String str) Parses input commands.
void	**run**() Run method.

Class Rejected

sics.jade.jadebank

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.io.IOException
              └java.rmi.RemoteException
                  └sics.jade.jadebank.Rejected
```

All Implemented Interfaces:
Serializable

---

public class **Rejected**
extends RemoteException

The Exception thrown when an action is rejected.

Title: Jade Bank Manager Application

Description: Jade Bank Manager Application

Author:
Simón af Frosterus & Leonidas Pantzopoulos (based on code written by Vlad Vlassov and Thom Birkeland)
Version:
1.2

Constructor Summary
`Rejected()`
`Rejected(String reason)`

Package sics.jade.osgi.jadebankserver

Class Summary	
Activator	The bundle activator that installs the Jade Bank Server.

Class Activator

sics.jade.osgi.jadebankserver

```
java.lang.Object
  └ sics.jade.osgi.jadebankserver.Activator
```

---

public class **Activator**
extends Object

The bundle activator that installs the Jade Bank Server. The precompiled JadeJadeBankServer is included in a TGZ file.

Author:
Simón af Frosterus & Leonidas Pantzopoulos

Constructor Summary
`Activator()`

Method Summary	
`void`	`start(BundleContext context)` Implements BundleActivator.start().
`void`	`stop(BundleContext context)` Implements BundleActivator.stop().

Package Summary	
sics.jade.resource.data	

Package sics.jade.resource.data

Interface Summary	
DataLogger	
DataService	

Class Summary	
DataLogger.Transaction	

Exception Summary	
AccountExistsException	AccountExistsException Title: AccountExistsException Description: AccountExistsException
AccountNotFoundException	AccountNotFoundException Title: AccountNotFoundException Description: AccountNotFoundException
LoggerException	LoggerException Title: LoggerException Description: LoggerException
NegativeBalanceException	NegativeBalanceException Title: NegativeBalanceException Description: NegativeBalanceException
SQLConnectionException	SQLConnectionException Title: SQLConnectionException Description: SQLConnectionException

Class AccountExistsException

sics.jade.resource.data

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └sics.jade.resource.data.AccountExistsException
```

All Implemented Interfaces:
Serializable

---

public class **AccountExistsException**
extends Exception

AccountExistsException

Title: AccountExistsException

Description: AccountExistsException

Author:
Simón af Frosterus & Leonidas Pantzopoulos
Version:
1.0

Constructor Summary
`AccountExistsException()`
`AccountExistsException(String reason)`

Class AccountNotFoundException

sics.jade.resource.data

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └sics.jade.resource.data.AccountNotFoundException
```

All Implemented Interfaces:
Serializable

---

public class **AccountNotFoundException**
extends Exception

AccountNotFoundException

Title: AccountNotFoundException

Description: AccountNotFoundException

Author:
Simón af Frosterus & Leonidas Pantzopoulos
Version:
1.0

Constructor Summary
AccountNotFoundException()
AccountNotFoundException(String reason)

Interface DataLogger

sics.jade.resource.data

public interface **DataLogger**

Author:
Simón af Frosterus & Leonidas Pantzopoulos

Version:
1.0

Inner Class Summary	
final static class	**DataLogger.Transaction**

Method Summary	
void	**log**(String message)
void	**log**(DataLogger.Transaction transaction, float amount, String accountName) Log a transation with amount
void	**log**(DataLogger.Transaction transaction, String accountName) Log a simple transation (without amount)

Class DataLogger.Transaction

sics.jade.resource.data

```
java.lang.Object
  └java.lang.Enum
      └sics.jade.resource.data.DataLogger.Transaction
```

All Implemented Interfaces:
Comparable, Serializable

Enclosing class:
DataLogger

final public static class **DataLogger.Transaction**
extends Enum

Field Summary	
static final DataLogger.Transaction	BALANCE
static final DataLogger.Transaction	DELETE_ACCOUNT
static final DataLogger.Transaction	DELETE_ACCOUNT_ERROR
static final DataLogger.Transaction	DEPOSIT
static final DataLogger.Transaction	DEPOSIT_ERROR
static final DataLogger.Transaction	ERROR
static final DataLogger.Transaction	EXIT
static final DataLogger.Transaction	GET_ACCOUNT
static final DataLogger.Transaction	GET_ACCOUNT_ERROR
static final DataLogger.Transaction	NEW_ACCOUNT
static final DataLogger.Transaction	NEW_ACCOUNT_ERROR
static final DataLogger.Transaction	WITHDRAW
static final DataLogger.Transaction	WITHDRAW_ERROR

Method Summary	
static DataLogger.Transaction	valueOf(String name)

static final DataLogger.Transaction[]	**values**()

Interface DataService

sics.jade.resource.data

---

public interface **DataService**

Author:
Simón af Frosterus & Leonidas Pantzopoulos

Version:
1.0

Method Summary	
float	**balance**(String name) Gets the balance of the account.
void	**connect**(String dbhost, int dbport, String dbname, String dbuser, String dbpassword, String bankName) Connect to the database
void	**deleteAccount**(String name) Deletes the specified account.
void	**deposit**(float value, String name) Deposits the selected amount into the account.
void	**getAccount**(String name) Gets the Account with the given name.
String	**getSummary**(String name) Gets a text summary of the Account (name, balance, etc).
void	**newAccount**(String name) Creates a new Account.
String	**test**() Tests without throwing any exceptions.
void	**withdraw**(float value, String name) Withdraws the selected amount from the account.

Class LoggerException

sics.jade.resource.data

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └sics.jade.resource.data.LoggerException
```

All Implemented Interfaces:
    Serializable

---

public class **LoggerException**
extends Exception

LoggerException

Title: LoggerException

Description: LoggerException

Author:
    Simón af Frosterus & Leonidas Pantzopoulos
Version:
    1.0

Constructor Summary
`LoggerException()`
`LoggerException(String reason)`

Class NegativeBalanceException

sics.jade.resource.data

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └sics.jade.resource.data.NegativeBalanceException
```

All Implemented Interfaces:
    Serializable

---

public class **NegativeBalanceException**
extends Exception

NegativeBalanceException

Title: NegativeBalanceException

Description: NegativeBalanceException

Author:
    Simón af Frosterus & Leonidas Pantzopoulos
Version:
    1.0

Constructor Summary
NegativeBalanceException()
NegativeBalanceException(String reason)

Class SQLConnectionException

sics.jade.resource.data

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └sics.jade.resource.data.SQLConnectionException
```

All Implemented Interfaces:

Serializable

---

public class **SQLConnectionException**
extends Exception

SQLConnectionException

Title: SQLConnectionException

Description: SQLConnectionException

Author:

Simón af Frosterus & Leonidas Pantzopoulos

Version:

1.0

Constructor Summary
SQLConnectionException()
SQLConnectionException(String reason)

Package Summary	
sics.jade.osgi.jadebanklogger.wrapper	

Package sics.jade.osgi.jadebanklogger.wrapper

Class Summary	
Activator	
JadeBankLoggerWrapper	

Class Activator

sics.jade.osgi.jadebanklogger.wrapper

```
java.lang.Object
  └sics.jade.osgi.jadebanklogger.wrapper.Activator
```

---

public class **Activator**
extends Object

Constructor Summary
`Activator()`

Method Summary	
void	`start(BundleContext context)`
void	`stop(BundleContext context)`

Class JadeBankLoggerWrapper

sics.jade.osgi.jadebanklogger.wrapper

```
java.lang.Object
  └sics.jade.osgi.jadebanklogger.wrapper.JadeBankLoggerWrapper
```

---

public class **JadeBankLoggerWrapper**
extends Object

Constructor Summary
`JadeBankLoggerWrapper()` Constructor

Method Summary

void	**bindFc**(String itfName, Object itfValue)
void	**configure**() Configure a Bank Server resource on the local machine.
String	**fixDir**(String dir)
String	**getAttribute**(String name)
String[]	**listFc**()
String[]	**listFcAtt**()
void	**loadApp**() Load an application on that Bank Server resource.
void	**log**(String message)
void	**log**(Transaction transaction, float amount, String accountName)
void	**log**(Transaction transaction, String accountName)
Object	**lookupFc**(String itfName)
void	**setAttribute**(String name, String value)
void	**start**() Start the resource. (=> run)
void	**stop**()
void	**unbindFc**(String itfName)

Package Summary	
sics.jade.osgi.jadebanklogger	

Package sics.jade.osgi.jadebanklogger

Class Summary	
Activator	Empty Bundle of the JadeBankLogger

Class Activator

sics.jade.osgi.jadebanklogger

```
java.lang.Object
  └sics.jade.osgi.jadebanklogger.Activator
```

public class **Activator**
extends Object

Empty Bundle of the JadeBankLogger

Author:
Simón af Frosterus & Leonidas Pantzopoulos

Constructor Summary
`Activator()`

Method Summary	
void	`start(BundleContext context)` Implements BundleActivator.start().
void	`stop(BundleContext context)` Implements BundleActivator.stop().

Package Summary	
sics.jade.osgi.datamanager.wrapper	

Package sics.jade.osgi.datamanager.wrapper

Class Summary	
Activator	
DataManagerWrapper	

Class Activator

sics.jade.osgi.datamanager.wrapper

```
java.lang.Object
  └sics.jade.osgi.datamanager.wrapper.Activator
```

---

public class **Activator**
extends Object

Constructor Summary
`Activator()`

Method Summary	
`void`	`start(BundleContext context)`
`void`	`stop(BundleContext context)`

Class DataManagerWrapper

sics.jade.osgi.datamanager.wrapper

```
java.lang.Object
  └sics.jade.osgi.datamanager.wrapper.DataManagerWrapper
```

---

public class **DataManagerWrapper**
extends Object

Constructor Summary
`DataManagerWrapper()` Constructor

Method Summary

float	**balance**(String name)
void	**bindFc**(String itfName, Object itfValue)
void	**configure**() Configure a Bank Server resource on the local machine.
void	**connect**(String dbhost, int dbport, String dbname, String dbuser, String dbpassword, String bankname) Connect to the database.
void	**deleteAccount**(String name)
void	**deposit**(float value, String name)
String	**fixDir**(String dir)
void	**getAccount**(String name)
String	**getAttribute**(String name)
String	**getSummary**(String name)
String[]	**listFc**()
String[]	**listFcAtt**() List the Fractal Component Attributes
void	**loadApp**() Load an application on that Bank Server resource.
Object	**lookupFc**(String itfName)
void	**newAccount**(String name)
void	**setAttribute**(String name, String value)
void	**start**() Start the resource. (=> run)
void	**stop**()
String	**test**()
void	**unbindFc**(String itfName)
void	**withdraw**(float withdrawAmount, String name)

Package Summary	
sics.jade.osgi.datamanager	

Package sics.jade.osgi.datamanager

Class Summary	
Activator	The bundle activator that installs the Jade Bank Server.

Class Activator

sics.jade.osgi.datamanager

```
java.lang.Object
  └sics.jade.osgi.datamanager.Activator
```

---

public class **Activator**
extends Object

The bundle activator that installs the Jade Bank Server. The precompiled JadeJadeBankServer is included in a TGZ file.

Author:
Simón af Frosterus & Leonidas Pantzopoulos

Constructor Summary
`Activator()`

Method Summary	
void	`start(BundleContext context)` Implements BundleActivator.start().
void	`stop(BundleContext context)` Implements BundleActivator.stop().

Package Summary	
sics.jade.osgi.mysql.wrapper	

Package sics.jade.osgi.mysql.wrapper

Class Summary	
Activator	
MySqlWrapper	

Class Activator

sics.jade.osgi.mysql.wrapper

```
java.lang.Object
  └sics.jade.osgi.mysql.wrapper.Activator
```

---

public class **Activator**
extends Object

Author:
Simón af Frosterus & Leonidas Pantzopoulos

Constructor Summary
`Activator()`

Method Summary	
`void`	`start(BundleContext arg0)`
`void`	`stop(BundleContext arg0)`

Class MySqlWrapper

sics.jade.osgi.mysql.wrapper

```
java.lang.Object
  └sics.jade.osgi.mysql.wrapper.MySqlWrapper
```

---

public class **MySqlWrapper**
extends Object

Constructor Summary
`MySqlWrapper()`

Method Summary	
void	**bindFc**(String itfName, Object itfValue)
void	**configure**() Configure an Mysql Resource on the local machine.
String	**fixDir**(String dir)
String	**getAttribute**(String name)
String[]	**listFc**()
String[]	**listFcAtt**()
void	**loadApp**() Load an application on that Mysql Resource.
Object	**lookupFc**(String itfName)
void	**setAttribute**(String name, String value)
void	**start**() Start the execution of the resource
void	**stop**() Stop the execution of the resource
void	**unbindFc**(String itfName)

Package Summary	
sics.jade.osgi.mysql	

Package sics.jade.osgi.mysql

Class Summary	
Activator	This bundle activator installs on the local machine resources needed by MySQL.

Class Activator

sics.jade.osgi.mysql

```
java.lang.Object
  └sics.jade.osgi.mysql.Activator
```

public class **Activator**
extends Object

This bundle activator installs on the local machine resources needed by MySQL. This bundle is specific for Linux machines. It contains the MySQL resources compressed in a TGZ file.

Author:
Simón af Frosterus & Leonidas Pantzopoulos

Constructor Summary
`Activator()`

Method Summary	
void	`start(BundleContext context)` This method unzips and installs the databse in the local dir
void	`stop(BundleContext context)` This method uninstalls the database from the local directory

APPENDIX F: FLAWS & BUGS

In this appendix, a list of some of the flaws and bugs that have been discovered throughout the realization of this Master's thesis are included.

- No parallel deployment: when deploying an application built of multiple components each deployed on different physical node, this is done on a sequential manner even when there are no dependencies that would justify the sequential deployment. Deploying different components on different nodes simultaneously would improve JADE's performance, mostly when large bundles are involved.
- Node's names in Fractal files are not real host names but machine names. For example, a machine with host name “alice.sics.se” but a machine name of “bob” will be referenced by JadeNode as “bob_0” (for the first node detected from this machine), so the virtual-node attribute in the Fractal file has to be set to “bob” rather than the actual host name.
- It seems that JADE requires the presence of a bundle even if the wrapper contains all the desired functionality.
- Lack of dynamism when adding components, even with only client interfaces, requires a full restart of the whole application. When the system map is integrated this will probably be handled.
- JADE lacks the ability to find classes and Fractal files that are inside packages (subfolders) located in Oscar bundle repositories. For example, if a Fractal file has a definition somewhere else (inside a package), this definition will not be dynamically loaded and has to be included in a JAR file embedded into the JadeBoot prior to starting JADE.

REFERENCES

[1] Jeffrey O. Kephart, David M. Chess (January 2003)
The Vision of Autonomic Computing
IEEE Computer 36 (1):41-52

[2] David Barlett. (2001) [On-line]
CORBA Component Model (CCM)
Accessed on December 2, 2006
URL: http://www-128.ibm.com/developerworks/webservices/library/co-cjct6/

[3] Diego Sevilla Ruiz (2006) [On-line]
The CORBA & CORBA Component Model (CCM) Page
Accessed on November 6, 2006
URL: http://ditec.um.es/~dsevilla/ccm/

[4] SOFA: Software Appliances [On-line]
Accessed on November 21, 2006
URL: http://sofa.objectweb.org or http://nenya.ms.mff.cuni.cz/projects.phtml?p=sofa&q=0

[5] SOFA component model [On-line]
Accessed on November 21, 2006
URL: http://nenya.ms.mff.cuni.cz/projects/sofa/tools/doc/compmodel.html

[6] E. Bruneton, T. Coupaye, J.B. Stefani. (2004).
The Fractal Component Model Specification.

[7] CoreGRID (2006).
Proposals for a Grid Component Model
Deliverable D.PM.02

[8] Sara Bouchenak et al. (2005).
Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters.

[9] Jim Dowling, Vinny Cahill. (2004).
Self-Managed Decentralised Systems using K-Components and Collaborative Reinforcement Learning.

[10] Jim Dowling, Vinny Cahill. (2001).
The K-Component Architecture Meta-Model for Self-Adaptive Software.
In Proceedings of Reflection 2001, LNCS 2192.

[11] Sheriff A. Gurguis, Amir Zeid. (2005).
Towards Autonomic Web-Services: Achieving Self-Healing Using Web Services.

[12] Gerrald Tesauro et al. (2004).
A Multi-Agent Systems Approach to Autonomic Computing.

[13] Peter Van Roy, Ali Ghodsi et al. (2005).
Self Management of Large-Scale Distributed Systems by Combining Structured Overlay Networks and Components.
CoreGRID Technical Report Number TR-0018

[14] Henry Gray (author), Warren H. Lewis (editor) (1918) [On-line]
Anatomy of the Human Body (20th edition)
Philadelphia, US: Lea & Febiger.
Accessed on November 8, 2006
URL: http://www.bartleby.com/107/

[15] IBM [On-line]
Autonomic Computing: IBM's Perpective on the State of Information Technology
Accessed on November 9, 2006
URL: http://www.ibm.com/industries/government/doc/contest/

[16] IBM Research | Autonomic Computing [On-line].
Accessed on November 9, 2006
URL: http://www.research.ibm.com/autonomic/index.html

[17] Paterson D. et al. (2002).
Recovery-Oriented Computing: Motivation, Definition, Techniques, and Case Studies.
Technical Report CSD-02-1175, CS Dept., UC Berkeley

[18] Total Cost of Ownership (TCO) [On-line].
Accessed on November 9, 2006
URL: http://searchdatacenter.techtarget.com/sDefinition/0,,sid80_gci342316,00.html

[19] Ronni Colville, Patricia Adams and Kris Brittain (2005.)
Optimize Change and Configuration Management With People, Process and Tools.
Gartner Research

[20] P. Lowber (2001)
Thin-Client vs. Fat-Client TCO
Decision Framework, DF-14-2800 Research Note

[21] Brian Killian (2006)
H*ow Much Does Your Network Cost? Calculating TCO*
Brintech, INC.

[22] Colin Atkinson et al. (2002)
Component based product line engineering with UML.
Addison-Wesley Longman Publishing Co., Inc.

[23] Gregory F. Pfister. (1997).
In Search of Clusters (2nd ed.).
Prentice Hall

[24] The UNIX Operating System: A Robust, Standardized Foundation for Cluster Architectures [On-line].
Accessed on November 14, 2006
URL: http://www.unix.org/whitepapers/cluster.htm

[25] Beowulf.org: Overview [On-line].
Accessed on November 15, 2006
URL: http://www.beowulf.org/overview/index.html

[26] Hewlett-Packard Company. (1998).
Designing Disaster Tolerant MC/ServiceGuard Clusters
Tech. Rep. No. B6264-90002

[27] D. Blevins (2001)
Overview of the Enterprise JavaBeans Component Model
In Heineman and Councill's *Component-based Software Engineering: Putting the Pieces Together.*
Addison-Wesley.

[28] Yi Liu and H. Conrad Cunningham. (2004)
Mapping Component Specifications to Enterprise JavaBeans Implementations.

[29] R. Monson-Haefel (1999)
Enterprise JavaBeans.
O'Reilly and Associates, Inc., first edition

[30] Shin Nakajima and Tetsuo Tamai. (2001).
Behavioural Analysis of the Enterprise JavaBeans Component Architecture.
NEC Corporation, Kawasaki, Japan

[31] The Common Component Architecture Forum [On-line].
Accessed on November 17, 2006
URL: http://www.cca-forum.org/

[32] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. (1999).
Toward a Common Component Architecture for High-Performance Scientific Computing.
Eighth IEEE International Symposium on High Performance Distributed Computing HPDC-8 '99 page 13.

[33] Sylvain Robert, Ansgar Radermacher and Vincent Seignole. (2005).
The CORBA Connector Model.

[34] Jim Dowling's PhD Thesis (2004).
The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems.
University of Dublin, Trinity College

[35] Rehman Adil [On-line]
Comparison of DCOM and Dot Net Remoting.
Accessed on Thursday, June 17, 2004
URL: http://www.ericsson.com/mobilityworld/sub/articles/other_articles/nl04jun12

[36] Distributed Systems Research Group [On-line].
Accessed on November 21, 2006
URL: http://nenya.ms.mff.cuni.cz/
Charles University, Prague

[37] ObjectWeb Forge: Project Info- SOFA [On-line].
Accessed on November 21, 2006
URL: http://forge.objectweb.org/projects/sofa

[38] SOFA implementation [On-line].
Accessed on November 21, 2006
URL: http://sofa.objectweb.org/userdoc/doc.html

[39] SOFA 2.0 - Projects - Distributed Systems Research Group[On-line].
Accessed on November 21, 2006
URL: http://nenya.ms.mff.cuni.cz/projects.phtml?p=sofa20&q=0

[40] Microsoft Corporation (1996)
Windows NT Server DCOM Technical Overview White Paper

[41] The Fractal Homepage
Accessed on November 21, 2006
URL: http://fractal.objectweb.org

[43] The Grid Component Model: an Overview (2006)
GCM D.PM.02 (in the Programming Model Institute plenary meeting)
URL: http://www.coregrid.net/mambo/content/view/210/292/
London, Jan 2006

[44] CoreGRID Network of Excellence - European Grid Research - Home [On-line]
Accessed on November 21, 2006
URL: http://www.coregrid.net/

[45] OMG (2006)
CORBA Component Model Specification
Version 4.0
Accessed on November 21, 2006
URL: http://www.omg.org/cgi-bin/doc?formal/06-04-01

[46] Mozilla.org: XPCOM [On-line].
Accessed on November 22, 2006
URL: http://www.mozilla.org/projects/xpcom/

[47] Rob C. van Ommering. (1998).
Koala, a Component Model for Consumer Electronics Product Software.
Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families (pages: 76-86).

[48] The GNOME Component Model [On-line].
Accessed on November 22, 2006
URL: http://developer.gnome.org/arch/gnome/componentmodel/

[49] K Desktop Environment - KDE Feature Overview [On-line].
Accessed on November 22, 2006
URL: http://www.kde.org/info/

[51] Richard S. Sutton, Andrew G. Barto. (1998) [On-line]
Reinforcement Learning: An Introduction.
Accessed on November 23, 2006
URL: http://www.cs.ualberta.ca/%7Esutton/book/ebook/the-book.html
London, England: The MIT Press (Cambridge, Massachusetts)

[52] E. Bruneton (2004)
Developing With Fractal (Tutorial).
France Telecom R&D
Version 1.0.3

[53] Sardes Project - Jade: a framework for autonomic management [On-line].
Accessed on February 09, 2007
URL: http://sardes.inrialpes.fr/jade/jade/jade-a-framework-for-autonomic-management.html

[54] Julien Legrand (2006) [On-line]
Jade - User guide
SADRES - INRIA Rhones-Alpes.
Accessed on November 15, 2006
URL: http://sardes.inrialpes.fr/research/jade

[55] OSGi - The Dynamic Module System for Java [On-line].
Accessed on February 09, 2007
URL: http://www.osgi.org/

[56] ObjectWeb: Oscar - An OSGi framework implementation [On-line].
Accessed on February 09, 2007
URL: http://oscar.objectweb.org/

[57] Sun Microsystems, Inc [On-line].
J*ava Message Service (JMS)*
Accessed on February 09, 2007
URL: http://java.sun.com/products/jms/

[58] Sun Microsystems, Inc [On-line].
Java Naming and Directory Interface (JNDI)
Accessed on February 09, 2007
URL: http://java.sun.com/products/jndi/

[59] ObjectWeb: JORAM: Java (TM) Open Reliable Asynchronous Messaging [On-line].
Accessed on February 09, 2007
URL: http://joram.objectweb.org/

[60] ObjectWeb: Fractal RMI [On-line].
Accessed on February 09, 2007
URL: http://fractal.objectweb.org/fractalrmi/index.html

[61] Pat Niemeyer [On-line]
BeanShell. Lightweight Scripting for Java
Accessed on February 15, 2007
URL: http://www.beanshell.org/

[62] ObjectWeb: Fractal [On-line]
Accessed on February 15, 2007
URL: http://fractal.objectweb.org/index.html

[63] ObjectWeb: Fractal Specification [On-line]
Accessed on February 15, 2007
URL: http://fractal.objectweb.org/specification/index.html

[64] ObjectWeb: Julia [On-line]
Accessed on February 15, 2007
URL: http://fractal.objectweb.org/julia/index.html

[65] ObjectWeb: Fractal RMI [On-line]
Accessed on February 15, 2007
URL: http://fractal.objectweb.org/fractalrmi/index.html

[66] ObjectWeb: Fractal ADL [On-line]
Accessed on February 15, 2007
URL: http://fractal.objectweb.org/fractaladl/index.html

[67] OBR - Oscar Bundle Repository [On-line]
Accessed on February 16, 2007
URL: http://oscar-osgi.sourceforge.net/

[68] Matthias Nicola, Jasmi John (2003)
XML Parsing: A Threat to Database Performance
12th Intl. Conference on Information and Knowledge Management, New Orleans CIKM'2003

[69] ProActive - Programming, Composing, Deploying on the Grid [On-line]
Accessed on March 29, 2007
URL: http://www-sop.inria.fr/oasis/proactive/

[70] Laurent Baduel, Francoise Baude et al. (2006).
Programming, Composing, Deploying for the Grid.
GRID COMPUTING: Software Environments and Tools", Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag.