Project no. 034567

Grid4All

Specific Targeted Research Project (STREP)
Thematic Priority 2: Information Society Technologies

Deliverable D1.6: Self-Management – Support and Case Studies

Due date of deliverable: June 2009.

Actual submission date: July 20, 2009.

Start date of project: 1 June 2006 Duration: 30 months

Organisation name of lead contractor for this deliverable: INRIA

Revision: 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level

PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	√

Contents

Grid4All list of participants

Role	Part. #	Participant name	Part. short name	Country
CO	1	France Telecom	FT	FR
CR	2	Institut National de Recherche en Informatique en Automatique	INRIA	FR
CR	3	The Royal Institute of technology	KTH	SWE
CR	4	Swedish Institute of Computer Science	SICS	SWE
CR	5	Institute of Communication and Computer Systems	ICCS	GR
CR	6	University of Piraeus Research Center	UPRC	GR
CR	7	Universitat Politècnica de Catalunya	UPC	ES
CR	8	ANTARES Produccion & Distribution S.L.	ANTARES	ES

1 Introduction

Building self-managing systems with Niche Building self-managing distributed systems raises at least two kinds of issues: architectural ones (how is a self-managed system organized ? how is self-management enabled ?), and decision and control ones (how are decisions for management actions reached ? how are such actions carried out ?). Ideally, under defined architectural templates, given high-level descriptions of the system functional behavior or goals, and of management policies to be observed, one should be able to synthetize (or evolve) a self-managing system. We are far from achieving this ideal, however: architectural principles are still in flux, with major insights coming from architectures of control, and we do not have the tools to synthetize decision and control, the knowledge in this area is very fragmented, with partial insights coming from many disciplines, including control theory [8], discrete event systems [3], distributed algorithms [7], and multi-agent systems [13]. In Grid4All, we focus on a classical view of the design of self-managing distributed systems, where human designers elaborate the system architecture and its decision and control structure, aided with a component model, associated linguistic tools, and target execution environment (Niche).

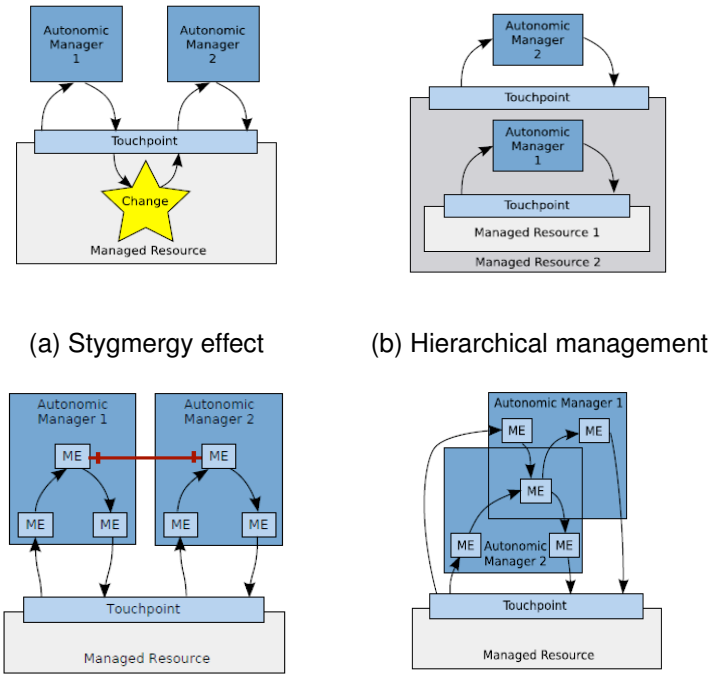
The Niche approach developed in Grid4All (see deliverable D6.7A) constitutes an original contribution to the question of architecting self-managing distributed systems. It combines a control-based and architecture-based approach — as exemplified e.g. by Rainbow [5] or Darwin [9] — with a peer-to-peer substrate: a self-managing system consists of an overlay of interacting components, under the supervision of a set of (possibly interacting, typically feedback) control loops. Each control loop, itself composed of a collection of interacting components, continuously monitors part of the system, analyzes its behavior based on it observations, plans and applies corrective actions if required. Under this approach, a simple self-managing application featuring a single control loop can roughly be considered as having three parts: the functional part, the touchpoints, and the *autonomic manager*. The functional part corresponds to the nominal behavior of the system. The autonomic manager corresponds to the analysis and decision stage of the management control loop. The touchpoints correspond to the sensors and effectors used by the autonomic manager to complete the control loop (monitoring the functional part, and carrying out corrective actions).

To support self-configuration, the Niche programming model is based on the Fractal reflective component model [2]. In the Fractal model, components are bound and interact with each other using two kinds of interfaces: (1) server interfaces offered by the components; (2) and client interfaces used by the components. Components are interconnected by *bindings*: a client interface of one component is bound to a server interface of another component. Different interaction semantics between components can be mediated by binding components, possibly spanning different nodes. Fractal allows nesting of components in composite components and sharing of components. Components have control (management) *membranes*, with introspection and intercession capabilities. It is through this control membrane, that components are started, stopped, configured. It is through this membrane that the components are passivated (as a prelude to component migration), and through which the component can report application-specific events to management (e.g. load). Fractal can be seen as defining a set of capabilities for components, it does not force application components to comply, but clearly the capability of the programmed components must match the needs of management. For instance, If the component is both stateful and not capable of passivation (or checkpointing) then management will not be able to transparently migrate the component. In terms of our control loop approach, a Fractal component can be understood as providing an encapsulation of functional behavior with explicit touchpoints provided by its control membrane as specific interfaces.

Building autonomic managers For many applications and environments, a single control loop structure is not sufficient. It is then desirable to decompose the autonomic manager into a number of cooperating autonomic managers each performing a specific management function or/and controlling a specific

part of the application. We have defined the following iterative steps to be performed when designing and developing the management part of a self-managing distributed application in a distributed manner:

- *Decomposition* The first step is to divide the management into a number of management tasks. Decomposition can be either functional or spatial .
- *Assignment* The tasks are then assigned to autonomic managers each of which becomes responsible for one or more management tasks.
- *Orchestration* Multiple autonomic managers, in the general case, are not independent since they manage the same system and there exist dependencies between management tasks. Therefore they need to interact and coordinate their actions in order to avoid conflicts and interference and to manage the system properly.
- *Mapping* The set of autonomic managers are then mapped to the resources.



(a) Stygmergy effect (b) Hierarchical management

(c) Direct interactions (d) Shared managed element

Figure 1: Orchestrating managers

Autonomic managers can interact and coordinate their operation in the following four ways:

- *Stigmergy*: Stigmergy is a way of indirect communication and coordination between agents (Figure 1(a)). Agents make changes in their environment, and these changes are sensed by other agents and cause them to do more actions. Stigmergy was first observed in social insects like ants. In our case agents are autonomic managers and the environment is the managed application.
- *Hierarchical Management*: By hierarchical management we mean that some autonomic managers can monitor and control other autonomic managers (Figure 1(b)). The lower level autonomic man-

agers are considered as a managed resource for the higher level autonomic manager. Communication between levels take place using touchpoints. Higher level managers can sense and affect lower level managers.

- *Direct Interaction*: Autonomic managers may interact directly with one another. Technically this is achieved by binding the appropriate management elements (typically managers) in the autonomic managers together (Figure 1(c)). Cross autonomic manager bindings can be used to coordinate autonomic managers and avoid undesired behaviors such as race conditions or oscillations.

- *Shared Managed Elements*: Another way for autonomic managers to communicate and coordinate their actions is by sharing managed elements (Figure 1(d)). This can be used to share state (knowledge) and to synchronise their actions.

Deliverable contributions This deliverable is organized in two parts. In the first part, we report on support for self-configuration that was developed during the third year of the project. In the second part, we report on three case studies, also completed during the third year of the project, that exemplify the potential for self-management provided by the Niche approach, and illustrate different cases of autonomic manager orchestration.

To further support self-configuration, we have extended the Fractal model with a formal model for dynamic component deployment and configuration. This model makes explicit the different entities involved in a distributed deployment process, and the dependencies that may exist between executing components and the software executables used to build them and activate them. This is crucial to ensure consistent decisions for component deployment and configuration can be made in a distributed environment (an example of the need to coordinate decisions across the part of a system, or of the need to coordinate local decisions – in this case by configuration managers). We have also extended our Oz Fractal implementation, that comprised the FructOz and LactOz libraries developed during year 2 of the project, with the WorkflOz library to complete our support for constructing self-configurable compnents in Oz.

Our three case studies deal with different examples of self-management that illutrate the design methodology mentioned above, and different examples of autonomic manager orchestration. The first case study, called YASS (Yet Another Storage Service), built using Niche, is a storage service that allows users to store, read and delete files on a set of distributed resources, and that transparently replicates the stored files for robustness and scalability. The second one, called YACS (Yet Another Computing Service), automatically distributes tasks among available distributed resources (masters and workers), monitors task execution and restarts failed tasks. YACS guarantees execution of jobs despite of nodes leaving or failing. The third case study is a self-repair framework, built on the Jade system that inspired Niche, for loosely-coupled cluster systems such as JEE application servers.

2 Support for self-configuration

2.1 Deployment and configuration model

The Fractal deployment and configuration model provides a formal analysis of the main abstractions involved when dynamically deloying and configuring distributed component structures, including relationships between executing components and their supporting software executables. The model has been formally specified with the Alloy specification language, and buids on the Alloy specification of the Fractal component model that was initially presented in Grid4All deliverable D1.4, and further refined [11]. The Fractal deployment and configuration model was developed with three specific objectives in mind:

1. To ensure that one could describe, using the model, *heterogeneous* deployment processes, i.e. deployment processes involving executables in different programming languages, relying on different

deployment tools at different software layers (e.g. deploying Java applications using OSGI bundles for Java code and RPM packages for the supporting C libraries in a Linux environment).

2. To make explicit relations between different entities involved in a deployment process so as to be able to monitor them and to control them in a self-managed distributed environment, where configuration management extends as much to running components than to the executables they depend upon.

3. To develop a formal model that can ultimately be used to reason about deployment processes, deployment-related functions and abstractions, and to characterize correctness conditions associated to such functions and processes.

Interestingly, although there have been numerous works on software deployment, especially, during the past decade, on architecture-based approaches to deployment (approaches that exploit software architecture descriptions to drive deployment processes), there are few works that address the above objectives. The three works most closely related to the Fractal deployment model are the Buildbox model [10], the OpenRec framework [14], and the Deployware framework [4]. The OpenRec framework uses the Alloy specification to formally characterize component configurations and to describe reconfiguration operations, however it supports only a non-hierarchical component model, and does not provide an analysis of deployment concepts and operations. The Builbox model provides a formal analysis of key deployment operations by means of a labelled transition system which are close to those described in our deployment model, but the Buildbow model does not consider deployment in a distributed context, and does not provide a modular analysis of the key components involved in a deployment process, a key requirement to address our second objective above. The Deployware framework addresses heterogeneous deployment processes, and relies on a UML meta-model to describe key deployment abstractions, which are modelled, as in our work, as Fractal components. The Deployaware metamodel is not formally specified, however, and provides an insufficiently detailed analysis in comparison to our model to meet our second objective.

The main concepts in the Fractal deployment model belong roughly to three main categories: software unit concepts, that capture the notion of executable software; transformer concepts, that capture basic forms of operations that can be applied to executables in the process of delivering them in a distributed environment; and support concepts, that capture key functionality required to enact the actual deployment of executables in a distributed environment. All these different concepts are specified as Fractal components, which makes the model fully recursive: components that implement a deployment process can themselves be deployed and configured, using the same abstractions and supporting mechanisms.

The software unit concepts are: software unit (SU), software unit system (SUS), and descriptor. A *software unit* corresponds to some software executable. Each software unit belongs primitively to a *software unit system*, i.e. a set of rules, APIs, or tools that define an executable format, and the operations that can be applied to it. For instance, an RPM package [1] corresponds to a software unit, while the set of tools, conventions, format rules, etc that define how RPM packages are formed and how they can be manipulated (mostly implicitly by the RPM tools), constitute a software unit system. Ditto for OSGI bundles (software units) and the OSGI specifications and tools (software unit system) [12]. From the point of view of formal model, one distinguishing feature of a software unit system is that it constitutes a *naming context*, as defined by the Fractal specification. This allows to freeely combine within the same deployment process, software units from different software unit systems, with no risk of confusion.

A software unit is characterized primarily by its set of *descriptors*. These are typed interfaces, i.e. access points for interaction with a component, according to the Fractal model, and can be server interfaces or client interfaces. The server descriptors of a software unit define the elements that are provided by a software unit (its exports – which can be as simple as sets of procedures or values), whereas the client descriptors define the elements it depends upon (its imports – which will be provided by other software units). As with general Fractal components, software units can be bound with other components

(which may or may not be other software units), and can be composites i.e. contain subcomponents (typically, other software units). Descriptor types abstract constraints attached to imports, such as versioning constraints.

The transformer concepts encompass the following components: launcher, installer, and resolver. A *launcher* takes as input a set of software units and produces a running component. An *installer* takes as input a set of software units and produces other software units. A *resolver* takes as input a set of software units and binds the client and server descriptors of these software units, thus (possibly only partially) resolving the dependencies between the given software units. Deploying an executing component thus typically involves a combination of installers, resolvers, and launchers. The software architecture of an application, as envisaged with the Fractal deployment model, thus encompasses both the relations (binding and containment) between running components, but also the relations between running components and software units that have been transitively involved in their launch.

The last concepts of the Fractal deployment model are: (deployment) node and (software unit) repository. A *node* is an abstraction of a set of computing resources, that are capable of executing components. We require a node to comprise at least a binding factory, to support some form of remote communication with its environment (other computing nodes), and a launcher, to support the creation of executing components from some software units. A *repository* is a store of software units, possibly from different software unit systems, and that provides access to software units through their names or their types.

2.2 Oz support for self-configuration

The FructOz and LactOz Oz/Mozart libraries, developed during year two of the project, have been refined and evaluated in particular in relation to the work which appears closest to it, the Smartfrog system [6]. To complement these two libraries, we have developed the WorkflOz library, that builds on FructOz. Together, these three libraries provide a set of basic operations for self-configuration as follows:

- *Components and reconfiguration*: The FructOz library supports the Fractal component model, and allows dynamic deployment and configuration of a distributed component structure.
- *Observation and navigation*: The LactOz library allows dynamic navigating and monitoring of a distributed component structure.
- *Tasks and Orchestration*: the WorkflOz library allows the coordination structure of a distributed system to be made explicit as a distributed structure of task components. This provides direct and reconfigurable support for manager orchestration, which we provide in WorkflOz in the form of workflow synchronization patterns.

WorkflOz is an Oz-based framework for defining component-based distributed coordination structures or workflows. The main goal of WorkflOz is to enable programmers to write succinct descriptions of complex processes, involving both control flow and data flow, in a compositional and intuitive style. To achieve this goal, WorkflOz provides direct support for all common workflow patterns, which represent widely used, recurring constructs in modern workflow management systems and languages. Unlike most workflow management systems and languages, WorkflOz can be extended with abstractions that capture new patterns, which can be assembled to express arbitrary coordination situations.

WorkflOz is based on the concept of a task, which represents a unit of computation that can be in different states (e.g., executing, terminated). A task has a set of input/output pins through which it receives/emits data values while it is executing. WorkflOz enables composing tasks using operators that capture workflow patterns. For example, consider the operator Seq that captures the sequence pattern. The following expression defines a composite task T3 that represents the sequence of tasks T1 and T2.

```
T3={Seq T1 T2}
```

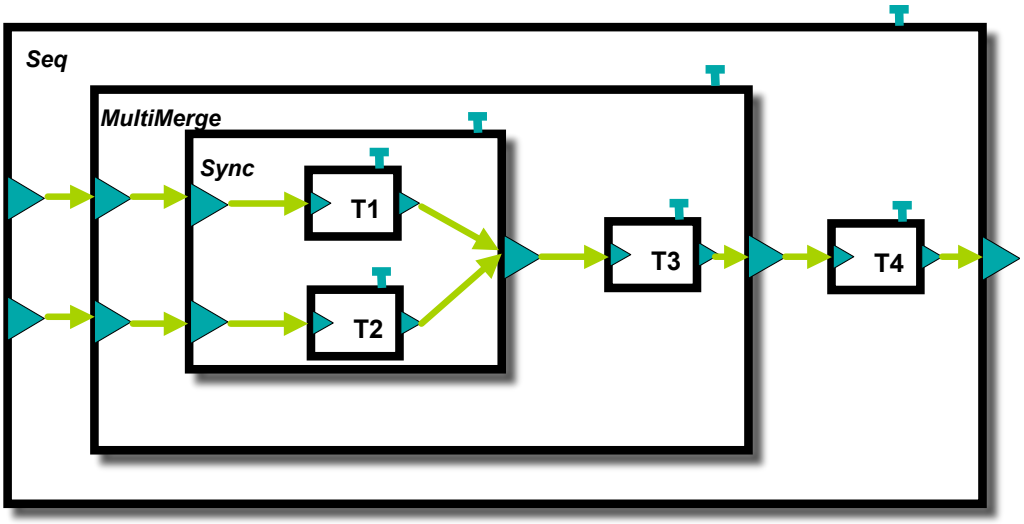


Figure 2: A composition of task components.

WorkflOz provides a set of operators similar to Seq (e.g., Sync, Parallel) which can be combined to express complex workflows and corresponding component configurations. For example, the following simple expression creates the component configuration depicted in Figure 2:

```
T5={Seq {MultiMerge {Sync T1 T2} T3} T4}
```

Primitive tasks can be created using helper functions, such BasicWrapper. This function takes as input an Oz unary function and creates a primitive task with one input pin and one output pin. When this task is executed, it invokes the Oz function with the input pin value, and emits the result to the output pin. Using BasicWrapper, T1 and T2 could be defined as follows:

```
T1={ BasicWrapper fun {$ X}  X+1 end}
T2={ BasicWrapper fun {$ X}  X*2 end}
```

T3 can be executed or cancelled using the following calls:

```
{Execute T3}
{Cancel T3}
```

Pins can be accessed directly to send or read data. For example, one can send the value 8 to the single input pin of T3 (the input pin identifier is 1) through the call:

```
{Send T3 1 8}
```

Tasks are realised as FructOz components with a particular type of controller (i.e., the *task controller*) and specially-marked server and client interfaces representing input and output pins respectively. Composite and primitive tasks correspond to composite and primitive FructOz components. Input and output pins are realised respectively as server and client interfaces. Following the FructOz design, pins are connected through bindings and hold a potentially unbounded list of values (i.e., an Oz stream). Input and output pins are identified using their order (from 1 to the number of input/output pins). When a task is executing, it continuously reads values from its input pins and writes values to its output pins, thus supporting streaming-style data flow. Tasks may be primitive or composite, containing any number of interconnected sub-tasks.

Tasks expose a task controller interface through which their lifecycle is managed. A basic task controller is defined in WorkflOz, in which tasks can be in one of four states: Ready, Executing, Cancelling, Terminated. The task controller exposes operations to trigger state changes (i.e., execute and cancel), to obtain the current state of the task, and to subscribe/unsubscribe to events corresponding to state changes (e.g., from Cancelling to Terminated). These operations are invoked by components external to the task (e.g., task controllers of containing components). The transitions from Executing and Cancelling to Terminated are triggered by the task itself when its computation or the cancellation process are terminated.

WorkflOz provides operators for creating different types of composite tasks, each type corresponding to a particular workflow pattern. A type of composite task (e.g., sequence type) specifies data- and control-flow dependencies between its sub-tasks. Specifically, it specifies the number of pins of the composite and the binding structure among sub-tasks and the composite. Moreover, it specifies control and lifecycle dependencies between the composite and its sub-tasks (e.g., the composite terminates when any of its subtasks terminate). Those dependencies are implemented by the task controller of the composite task using the controllers of the sub-tasks. In the previous sequence example, the task controller of T3 uses the task control interface of T1 to register for state change events; when a termination event is received, the controller triggers the execution of T2 through the T2 task controller.

Extending WorkflOz with support for a new pattern involves (1) defining the control- and data-flow semantics of the corresponding composite task, and (2) implementing the associated task controller and any supporting functionality. This implementation is facilitated by utility classes for managing the task lifecycle, sending events, and accessing and connecting pins.

3 Case study: a distributed storage service

We present in this section the YASS (Uet Another Storage Service) case study, that illustrates how to design a self-managing distributed system monitored and controlled by multiple distributed autonomic managers.

3.1 YASS specification

YASS is a storage service that allows users to store, read and delete files on a set of distributed resources. The service transparently replicates the stored files for robustness and scalability. Assuming that YASS is to be deployed and provided in a dynamic distributed environment, the following management tasks are required:

- Maintain the file replication degree by restoring the files which were stored on a failed/leaving resource. This function provides the self-healing property of the service so that the service is available despite of the resource churn.
- Maintain the total storage space and total free space to meet QoS requirements by allocating additional resources when needed. This function provides self-configuration of the service.

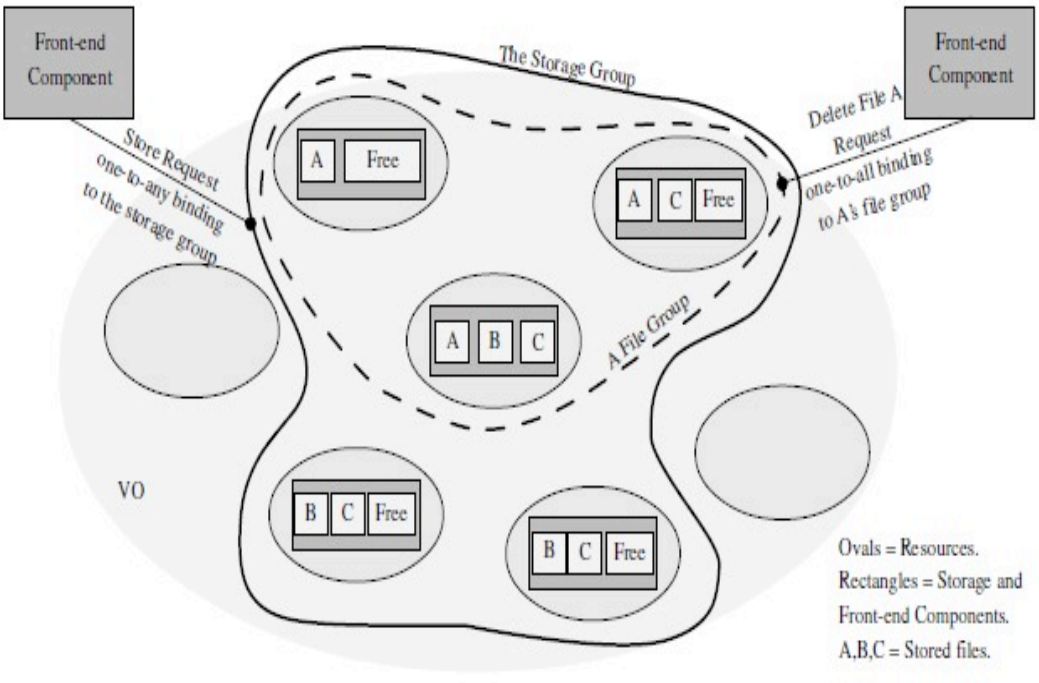


Figure 3: YASS functional part.

- Increasing the availability of popular files. This and the next two functions are related to the self-optimization of the service.
- Release excess allocated storage when it is no longer needed.
- Balance the stored files among the allocated resources.

3.2 YASS design

Functional part

A YASS instance consists of front-end components and storage components as shown in Figure 3. The front-end component provides a user interface that is used to interact with the storage service. Storage components represent the storage capacity available at the resource on which they are deployed. The storage components are grouped together in a storage group. A user issues commands (store, read, and delete) using the front-end. A store request is sent to an arbitrary storage component (using one-to-any binding between the front-end and the storage group) which in turn will find some r different storage components, where r is the file's replication degree, with enough free space to store a file replica. These replicas together will form a file group containing the r storage components that will host the file. The front-end will then use a one-to-all binding to the file group to transfer the file in parallel to the r replicas in the group. A read request is sent to any of the r storage components in the group using the one-to-any binding between the front-end and the file group. A delete request is sent to the file group in parallel using a one-to-all binding between the front-end and the file group.

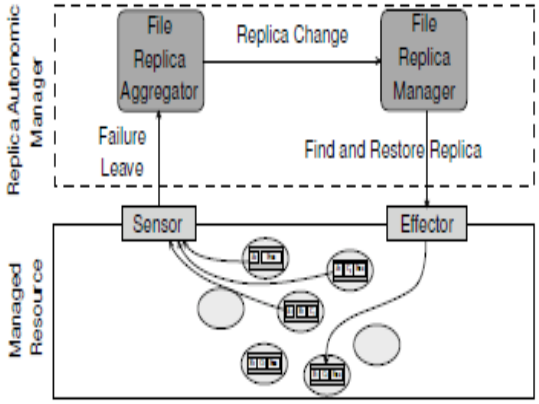


Figure 4: YASS Self-healing control loop.

Enabling management

Given that the functional part of YASS has been developed, to manage it we need to provide touchpoints. DCMS provides basic touchpoints for manipulating the system's architecture and resources, such as sensors of resource failures and component group creation; and effectors for deploying and binding components. Beside the basic touchpoint the following additional, YASS specific, sensors and effectors are required:

- A load sensor to measure the current free space on a storage component.
- An access frequency sensor to detect popular files.
- A replicate file effector to add one extra replica of a specified file.
- A move file effector to move files for load balancing.

Management part

The following autonomic managers are needed to manage YASS in a dynamic environment. All four orchestration techniques described above are demonstrated.

Replica Autonomic Manager The replica autonomic manager is responsible for maintaining the desired replication degree for each stored file in spite of resources failing and leaving. This autonomic manager adds the self-healing property to YASS. The replica autonomic manager consists of two management elements, the File-Replica-Aggregator and the File-Replica-Manager as shown in Figure 4.

Storage Autonomic Manager The storage autonomic manager is responsible for maintain the total storage capacity and the total free space in the storage group, in the presence of dynamism, to meet QoS requirements. The dynamism is due either to resources failing/leaving (affecting both the total and free storage space) or file creation/addition/deletion (affecting the free storage space only). The storage autonomic manager will reconfigure YASS to restore the total free space and/or the total storage capacity to meet the requirements. The reconfiguration is done by allocating free resources and deploying additional storage components on them. This autonomic manager adds the self-configuration property to

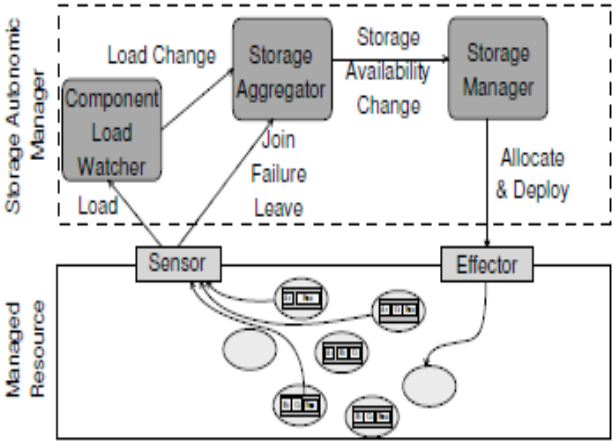


Figure 5: YASS Self-configuration control loop.

YASS. The storage autonomic manager consists of Component-Load-Watcher, Storage-Aggregator, and Storage- Manager as shown in Figure 5.

Direct Interactions to Coordinate Autonomic Managers The two autonomic managers, replica autonomic manager and storage autonomic manager, described above seem to be independent. The first manager restores files and the other manager restores storage. But as we will see in the following example it is possible to have a race condition between the two autonomic managers that will cause the replica autonomic manager to fail. For example, when a resource fails the storage autonomic manager may detect that more storage is needed and start allocating resources and deploying storage components. Meanwhile the replica autonomic manager will be restoring the files that were on the failed resource. The replica autonomic manager might fail to restore the files due to space shortage if the storage autonomic manager is slower and does not have time to finish. This may also prevent the users, temporarily, from storing files. If the replica autonomic manager would have waited for the storage autonomic manager to finish, it would not fail to recreate replicas. We used direct interaction to coordinate the two autonomic managers by binding the File-Replica-Manager to the Storage-Manager. Before restoring files the File-Replica-Manager informs the Storage-Manager about the amount of storage it needs to restore files. The Storage-Manager checks available storage and informs the File-Replica-Manager that it can proceed if enough space is available or ask it to wait.

Optimising Allocated Storage It is possible to design an autonomic manager that will detect low resource utilization, move file replicas stored on a chosen lowly utilized resource, and finally release it. Since the functionality required by this autonomic manager is partially provided by the storage and replica autonomic managers we will try to augment them instead of adding a new autonomic manager, and use stigmergy to coordinate them. It is easy to modify the storage autonomic manager to detect low storage utilization. The replica manager knows how to restore files. When the utilization of the storage components drops, the storage autonomic manager will detect it and will deallocate some resource. The deallocation of resources will trigger, through stigmergy, another action at the replica autonomic manager. The replica autonomic manager will receive the corresponding resource leave events and will move the files from the leaving resource to other resources.

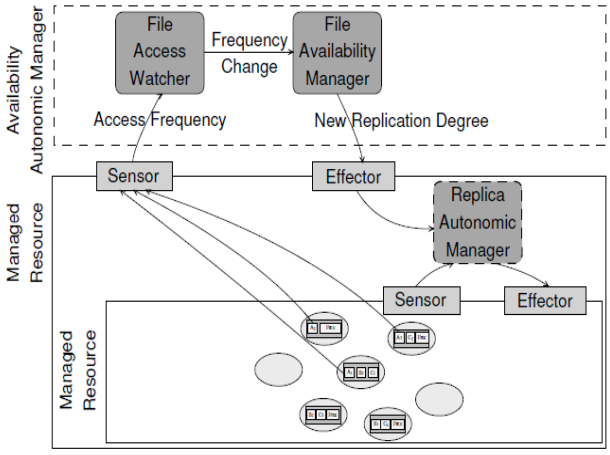


Figure 6: Hierarchical management in YASS.

Improving file availability Popular files should have more replicas in order to increase their availability. A higher level availability autonomic manager can be used to achieve this through regulating the replica autonomic manager. The autonomic manager consists of two management elements. The File-Access-Watcher and File- Availability-Manager shown in Figure 6 illustrate hierarchical management.

Balancing File Storage A load balancing autonomic manager can be used for self-optimization by trying to lazily balance the stored files among storage components. Since knowledge of current load is available at the Storage- Aggregator, we design the load balancing autonomic manager by sharing the Storage-Aggregator as shown in Figure 7. All autonomic managers we discussed so far are reactive. They receive events and act upon them. Sometimes proactive managers might be also required, such as the one we are discussing. Proactive managers are implemented in DCMS using a timer abstraction.

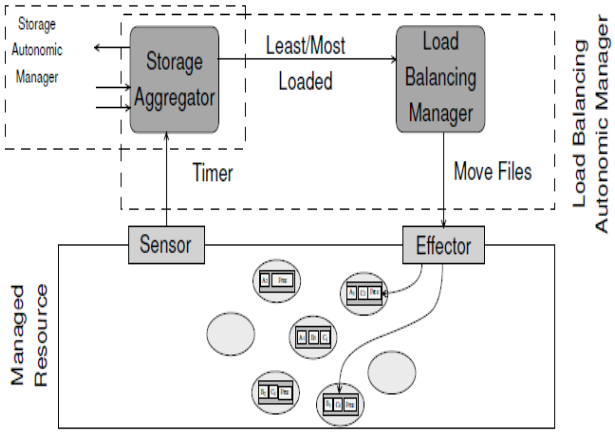


Figure 7: Sharing of managed elements in YASS.

4 Case study: a distributed computing service

The second case study is a computing service called YACS (Yet Another Computing Service). It illustrates how to design a self-managing distributed system with a computational oriented service.

4.1 YACS Specification and design

Yet Another Computing Service, YACS, is a robust computing service that executes jobs submitted by end-users through service front-ends. A job is specified as a bag of independent tasks to be computed on a set of distributed nodes (computers). The service can be used to perform different kinds of batch jobs or bag-of-task applications such as parameter-sweep simulation, video transcoding, ray-tracing or other applications that follow the master-worker paradigm.

The service automatically distributes tasks among available distributed resources (masters and workers), monitors task execution and restarts failed tasks. YACS guarantees execution of jobs despite of nodes leaving or failing. YACS supports checkpointing that allows restarting execution from the last checkpoint. Furthermore, YACS scales, i.e. changes the number of master and workers, when the number of jobs/tasks changes. In order to achieve high availability, the service always maintains a number of free masters and workers so that new jobs can be accepted without delay.

Following the Niche design guidelines the YACS service is designed as consisting of a functional part resposible for job execution, and a management part resposible for self-healing and self-configuration of the service.

Functional part The functional part of YACS contains the following two parts. Job execution management is in charge of managing and executing of jobs. This is done using a master-worker pattern where each job is assigned to a master which collects a set of workers to execute the job; Resource management is in charge of resource allocation to the job execution management. The resource service knows the status of the system's functional resources (masters and workers) by means of monitoring. The functional part of YACS includes a set of functional components: masters, workers and the resource service. A master accepts submitted jobs, finds workers, assigns tasks to the workers, monitors execution and collects results. A worker executes tasks assigned to it and communicates results back to its master. The resource service keeps track of resources, i.e. masters and workers, and allocates them to requesting clients: front-ends and masters, respectively. A user submits a job via a front-end, which contacts the resource service to obtain a free master to execute the job, and sends the job to the master. The master gets free workers from the resource service and assigns tasks to the workers for execution. Upon completion of all tasks and collecting all results, the master communicates the results to the frontend.

Management part The management part of YACS includes a set of management components, watchers, aggregators and the control manager, that are in charge of detecting and healing failures of functional components (workers, masters and the resource service). It is also responsible for scaling and reconfiguring the functional part (i.e. creating/removing and binding functional components) when the service load changes beyond specified thresholds. Watchers monitor functional components; aggregators collect and analyze the monitoring information coming from watchers and issues status events to the control manager, which performs self- management actions.

4.2 Self-management in YACS

There is a need for self-management capabilities within YACS in order to guarantee execution of jobs and service availability despite of node failures, departures and arrivals, as well as changes in load. This requires self-healing of masters, workers and the resource service as well as self-configuration to react on

load changes. The functional part of YACS includes execution management (masters and workers) and resource management (the resource service); therefore the self-management capabilities are split into two parts, one managing the job execution management, the other managing the resource management. Self-management is achieved by placing management elements (watchers, aggregators and managers) to build self-healing and self-configuration control loops. Self-management of execution management is concerned with self-healing. There are watchers that monitor the state of masters and workers and take action upon detecting failures. Upon sensing failures they perform healing and actuation by redeploying jobs or tasks on replacement masters or workers. If the job supports checkpointing, tasks are restarted based on their last known state. The process follows the same pattern when a master component is healed.

Self-management of resource management is concerned with both self-healing and self- configuration. The self-healing control loop is responsible for healing the resource service if any of the components that make up the service fail. This is critical since the resource service is used by all jobs. The self-configuration control loop senses the availability of free functional resources within the system, and upon need deploys additional functional components. The ServiceWatcher senses the state of the system through sensors on resource service components, as well as failures of resource service components. This information is sent to the ServiceAggregator which analyzes the state to see if the availability of free resources is below the configurable threshold, in which case actuation is needed to deployment of additional functional components. The ConfigurationManager performs this actuation by trying to find free physical nodes and deploys the additional components there.

5 Case study: a self-healing JEE cluster

The third case study is a self-healing cluster of legacy JEE servers. It illustrates the application of our self-management approach to legacy systems and a form of distributed design for an autonomic manager based on active replication.

5.1 Design

Functional part

In a JEE multi-tiered architecture, the Web application server is classically divided in several tiers: the HTTP daemon (e.g. Apache), the servlet engine (e.g. Tomcat), the EJB business server (e.g. JOnAS), and the database tier (e.g. MySQL). To manage a cluster of such application servers, JEE Each tier can be independentlyly wrapped by Fractal components that provide the required sensors and effectors for management.

Management part

Each physical machine in the cluster is represented by components called *nodes*. Managed elements executing on these physical machines are subcomponents of nodes. The set of nodes together with their subcomponents constitutes a management domain, i.e. a set of entities under the control of a single management authority and associated set of policies.

To support repair, i.e. the recovery from failures of managed elements corresponding to hardware or software resources, some knowledge must be maintained of the runtime configuration of the system, which should persist even in presence of failures. This knowledge takes the form of a *system map*, a component structure that mirrors the component structure of the managed system. The system map serves as an intermediate between manager components and managed components, and that carries out reconfiguration operations originating with the repair manager component. The repair manager is

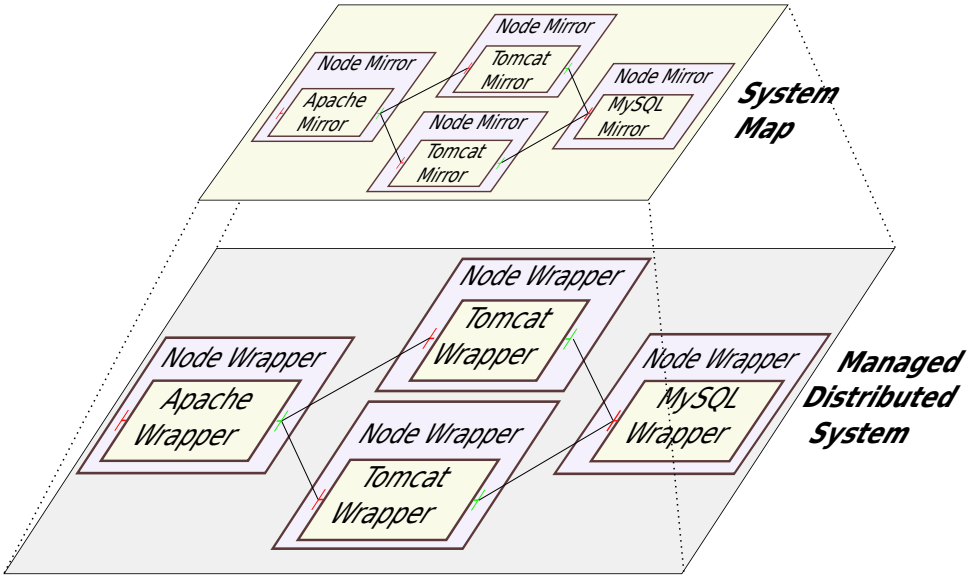


Figure 8: System Map and Managed System

responsible for the analysis of observations on the managed system, the planning and (ultimately) execution of management operations in response to observations, and according to management objectives and policies. A repair manager component and the system map together form an autonomic manager, and execute on a node called a manager node. We call *management subsystem* the set of manager nodes together with their manager subcomponents. To enable self-repair, i.e. to recover from faults that may occur within the management subsystem, the manager nodes are replicated and are an integral part of the managed system.

System Map: basic structure

The runtime configuration of the managed distributed system is captured in the System Map that provides not only an accurate representation of the managed distributed system but also the ability to reconfigure it. In other words, the System Map acts as an intermediate reflective structure between autonomic managers and managed elements. The System Map, depicted in Figure 8 where managed components are wrapped legacy elements, relies on the concept of *mirror components*.

Each managed component has a mirror component in the System Map. Hence, one finds a mirror in the System Map for each managed component and each managed node. Each mirror captures the *complete architectural state* of the managed component it mirrors, which includes the following information:

- The lifecycle status of the managed component (at a minimum: started or stopped) (captured as the lifecycle status of the mirror component).
- The managed component attributes, in the form of ⟨key,value⟩ pairs (captured as attributes of the mirror component).
- The managed component client and server interfaces (captured as interfaces of the mirror component) together with their bindings (which are captured as bindings between mirror components).
- The subcomponents of the managed components (captured as subcomponents of the mirror component)

Through mirrors, manager components can both introspect and reconfigure the architecture of the managed system. By introspecting, we mean that managers can access the mirrors and therefore obtain

the architectural state they mirror. For instance, managers can know which managed components are deployed where, if they are started, and what are the bindings that link them. By reconfiguring, we mean that manager components can change the managed architecture as required. For instance, managers can start-stop mirrors or change managed component attributes. They can also create new managed components (and their mirrors), or remove or create bindings between managed components (and their mirrors).

Since mirrors are Fractal components, such reconfigurations are simply done through the control interface of the mirrors, using the regular management operations available via control interfaces of the components. In other words, manager components reconfigure mirrors as they would the actual managed components. Such reconfigurations of mirrors are done within atomic sessions on the System Map. During each session, only the mirrors are reconfigured, management operations are not propagated to their managed components. When a session is committed by managers, it is the responsability of the System Map to carry out the reconfiguration out to the concerned managed components.

The rationale for our construction of the System Map is worth discussing. We noted above that some knowledge of the managed system and its management state must be available: the System Map provides it. One could have built the System Map as a pure data structure, leaving manager components the responsibility for updating the System Map and maintaining its causal connection with the managed system. Having a System Map as an active entity provides a useful decoupling of responsibilities between System Map and manager components: the System Map is responsible for maintaing its causal connection with the managed system, whereas manager components are responsible for analysis, planning a reconfiguration in response to the analysis, and updating the System Map to launch the execution of the reconfiguration, which will be carried out ultimately by the System Map. With this design, manager components do not need to reimplement each time mechanisms to maintain the causal connection of the System Map with the managed system.

5.2 Self-repair

Principles of repair When a repair manager is notified of a failure (node failure detection is provided via a classical heartbeat protocol, run by the manager nodes), A repair manager recovers from it by opening an *atomic session*, conceptually composed of two phases: analyzing the system map, and reconfiguring the system map to repair the failure. The repair manager only interacts with the system map, never directly with the wrappers. At any time up to the final commit of the session, the repair manager can abort the repair session. Once the repair manager commits a session, the actual commit becomes the responsibility of the system map.

The commit of a session that repairs a single failed component is essentially a two-step process. The first step recreates the managed components lost to a failure on available nodes. The second step updates bindings: it recreates bindings on the new managed component, using the isomorphic bindings between mirrors, and it removes and replaces stale bindings, i.e. bindings that connected the failed components to other ones.

A corresponding simple case is illustrated in Figure 9. The left hand of the figure represents the state of system map and of the managed system right at the end of the repair session, when the repair manager is about to commit. The managed component B has failed on node N'. The repair manager decided to recreate it on the node N, which is simply expressed by removing the mirror of B from the composite mirror for the node N' and adding it to the composite mirror of the node N. The right hand side of the figure depicts the end result of the commit, applying the pseudo-code in Listing **??**. A new managed component B has been created on the node N and all the bindings have been changed accordingly. The management reference from the mirror of B has also been updated accordingly.

This example corresponds to a real situation between an Apache HTTP daemon and its Tomcat servlet engines. When the hardware node where a Tomcat servlet engine runs fails, a new instance of a

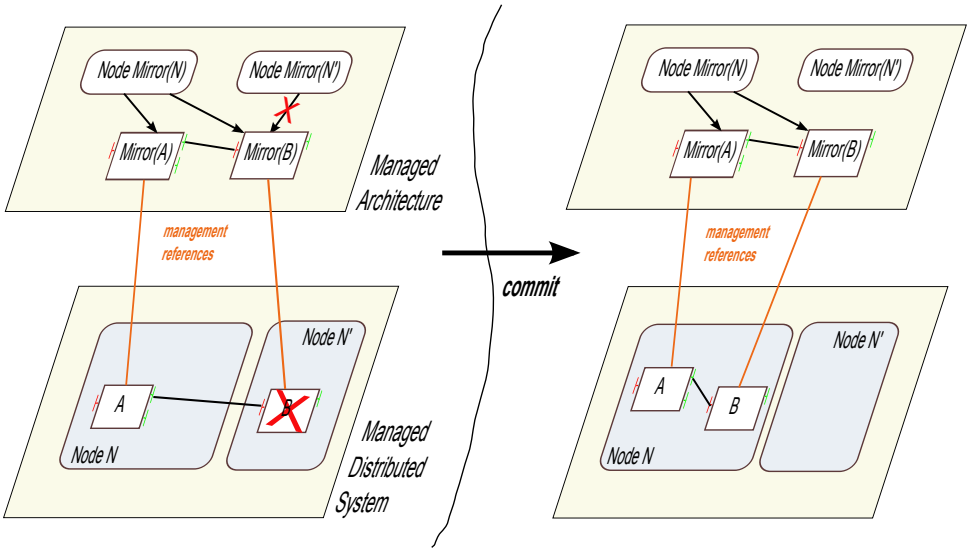


Figure 9: Recreating a managed component

Tomcat servlet engine must be recreated on a new hardware node. Therefore, the Apache HTTP daemon must first close its socket to the failed Tomcat and re-open one to the newly created Tomcat. The unbind operation resets the IP address and port in the Apache configuration file while the bind operation sets the new correct values. It is interesting to point out that the Apache daemon has to be restarted to re-read its configuration file. Hence, to apply the unbind-rebind operations, the wrapper has to actually shutdown and restart the Apache HTTP daemon. This works perfectly well with loosely-coupled legacy systems: clients will reissue the requests lost during the short period of time it takes for the Apache HTTP daemon to restart.

Self-repair To achieve full self-repair, even if failures occur in the management subsystem itself, we use an active replication scheme between manager nodes. We use a uniform atomic broadcast protocol [7] that ensures a one-and-only-one semantics for issued managed operations. Since we replicate manager nodes, there can be redundant repair sessions happening across replicas. Each repair session will issue redundant management operations when committing. The atomic broadcast protocol ensures that each individual management operation forwarded from the redundant session commits onto a managed component is received once and only once. However, replication only hides the failures of individual manager nodes. As in any replication scheme, such individual failures must be repaired to maintain the replication cardinality and thereby preserve the availability of the replicated service over the long term.

In order for the management subsystem to repair itself, we have to deal with two issues. First, the architecture of the management subsystem must be captured within the System Map. Indeed, when a failure occurs, the Repair Manager introspects and reconfigures the System Map in order to understand and repair a failure. While we need no modification to the repair algorithm presented in Listing **??**, we need to enhance our System Map to capture replicated components. Second, we need to extend the repair capabilities to replicated components. So far, our commit protocol knows how to repair a failed component that is not replicated. With failures of individual manager components, we have to repair individual replicas of replicated components.

To deal with the first issue, we introduce the concept of *replicator mirrors*. A replicator mirror is a mirror composite, that mirrors a Fractal composite. Therefore, a replicator mirror is a tree of mirrors since a Fractal composite is actually a tree of composites with components as leaves. The semantics of a replicator mirror is *deep replication*. Akin to a deep copy, a deep replication applies not only to the root composite but also to all its subcomposites and subcomponents. Figure 10 depicts a replicator mirror C

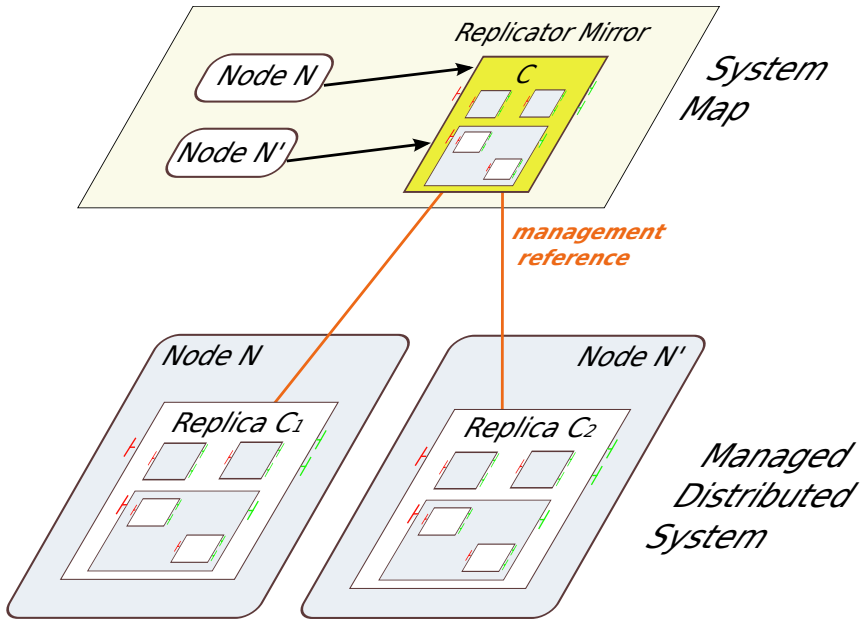


Figure 10: Shared Replicator Mirror

and the corresponding deep replication of managed components. As a composite, the replicator mirror C is a tree of mirrors, represented as nested rectangles. For each mirror in that tree, the corresponding managed component is replicated across the nodes N and N'.

Also note on Figure 10 that the node mirrors for nodes N and N' refer to the replicator mirror for the component C. This captures the knowledge that the replicator mirror has deployed replicas of the composite C on both nodes N and N'. This ensures that our Repair Manager can correctly analyze a failure. For instance, upon a node failure, our Repair Manager introspects the System Map to find components that were deployed on the failed node. It therefore finds the replicator mirrors that have replicas on that node. Our Repair Manager has no concept of a replicator mirror, it treats them as regular mirrors; simply relocating a failed mirror to an available node, as it did before in Figure 9. Removing a mirror from a node mirror and adding it to another node mirror actually updates the list of parent composites of that mirror. In the case of a replicator mirror, the parent composites are nodes and represents the list of nodes where the replicator will replicate components.

Using replicator composites, manager components can be replicated at install time. Once installed and running, each manager replica watches over the other manager replicas, detecting their failure and repairing them. Figure 11 depicts the complete architectural view of a replicated management structure and its distributed managed system. As before, we can see the managed system distributed on several nodes, Node 4 and Node 5 in this case. Both Node 4 and Node 5 have a complex composite with multiple components wrapping different legacy systems (not depicted in the figure). We can also see that the autonomic repair manager, comprising the repair manager and the system map, is replicated on Node 1 and Node 2.

Each replica of the system map has a complete description of the managed distributed sytem. Precisely, we see that the mirrors for Node 4 and Node 5 refer to the mirrors for the components wrapping legacy systems deployed on these two nodes. This description in terms of mirrors enable the repair manager to analyze and repair failures on managed legacy components on either Node 4 or Node 5, as discussed in Section **??**. However, the system map also contains the description of the replicated manager components. Looking at node mirrors for Node 1 and Node 2, once can see that they refer to replicator mirrors, one for the composite of the system map and one for the composite of the repair

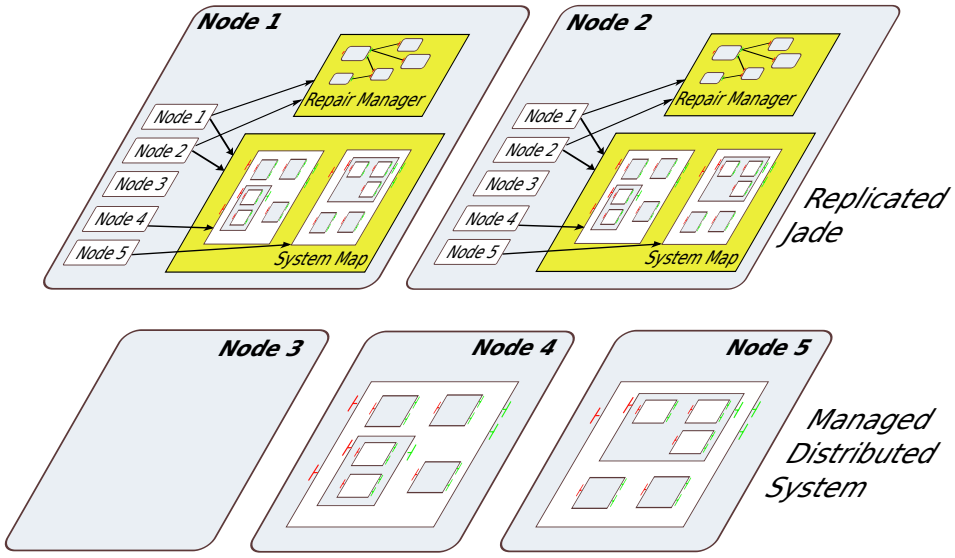


Figure 11: Complete Distributed Architecture

manager.

If Node 1 or Node 2 would fail, the node that is still standing would detect and repair the failure. For instance, if Node 1 fails, the replica of the repair manager on Node 2 will detect the failure and start a repair session. The analysis will happen on the replica of the system map on Node 2, yielding a reconfiguration of the architecture where all replicators on the failed Node1 are relocated to an available node, in this particular case it would be the Node 3. This is all what needs to happen during the repair session. At commit, since failed composites are mirrored through a replicator mirror, the commit will not simply recreate the lost components but actually reinsert them as new replicas. There is nothing special here about this replica reinsertion, we can use any standard reinsertion techniques suited for an active replication scheme [7]. This ensures that the newly created replicas do have a correct state and are reinserted properly in the atomic broadcasts. In the end, Node 3 would look exactly like Node 1, prior to its failure.

5.3 Evaluation

In this section, we evaluate our approach with the management of a cluster of multi-tiered Web servers. This evaluation has two purposes. First, it provides an example of a concrete use of our repair management architecture with a legacy system. Second, this evaluation allows us to demonstrate the negligible overhead of the management framework, and to analyze the Mean Time To Repair (MTTR) of a managed system. Our goal is not instantaneous repair since our repair management system first aims at replacing a human administrator whose MTTR are classically well over dozens of minutes, even for skilled operators. In average, even for complex repairs, our repair management achieves an MTTR well below a minute. In simpler failure cases, it achieves fast MTTR of a few seconds. More importantly, it is compatible with high-availability designs based on replicated servers, maintaining the replication degree by repairing and reinserting replicas in the background. In the following, we first discuss the wrapping of JEE tiers. We then discuss the MTTR achieved by a multi-tier Web server with our repair management, considering different failures, either in a basic multi-tiered architecture or in a more advanced architecture designed for high availability with replicated tiers.

Wrapping JEE Web Servers

In a JEE multi-tiered architecture, the Web server is classically divided in several tiers: the HTTP daemon (e.g. Apache), the servlet engine (e.g. Tomcat), the EJB business server (e.g. JOnAS), and the database tier (e.g. MySQL). Each tier is independently wrapped in a component. Figure 12 illustrates the wrapping of a JEE Web server. In the bottom layer, the figure shows the legacy elements and the way they are connected through legacy communication channels. At the top, the Managed Distributed System level provides control on the legacy elements through wrappers that expose uniform management operations.

Wrappers are bound in a way that reflects the legacy communication channels. In particular, the Apache wrapper requires the *JKConnector* interface that the Tomcat wrapper provides. This enables a binding to be created between the Apache and Tomcat wrappers that captures the presence of a jk connection between the Apache HTTP daemon and the Tomcat servlet engine. The Tomcat wrapper also requires a JNDI interface that captures the connection to a JNDI naming service. In addition, it exposes the *port* and *maxClient* attributes. The wrapper of the EJB Server requires a JNDI interface, as previously described for the Tomcat wrapper. as well as a *JdbcConnector* interface, capturing the use of the JDBC connector. This *JdbcConnector* interface is provided by the wrapper of MySQL.

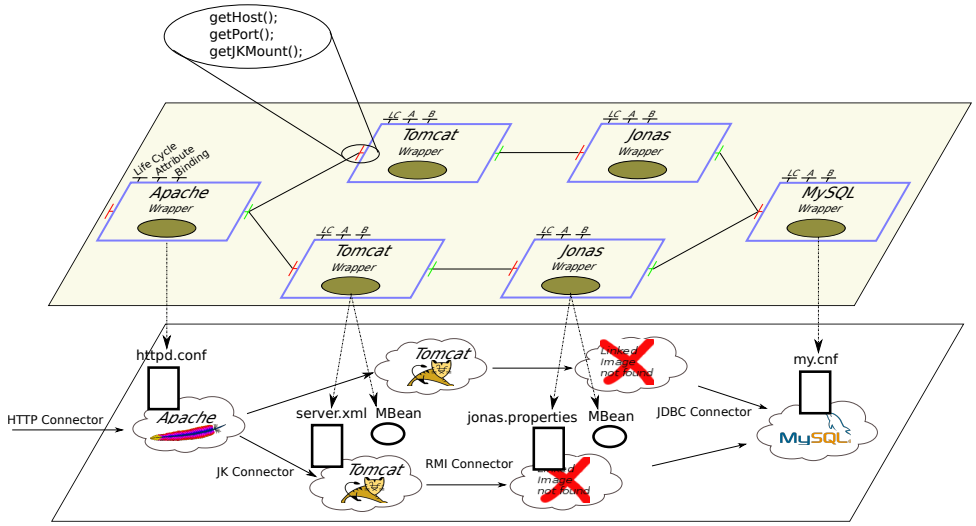


Figure 12: Wrapping a JEE Web Server

It is important to point out that the Java implementation of these wrappers mainly relies on the manipulation of shell scripts and standard configuration files (e.g. *my.cnf* for MySQL, *httpd.conf* for Apache, etc).

Repairing JEE Web servers

To illustrate repairing a JEE cluster, we wrapped and managed a multi-tiered JEE server running the RUBIS benchmark [**?**]. We used the version 1.4.2 that comes with a Web client emulator allowing to simulate a realistic load on the JEE server. This server was composed of a Web tier running Apache version 1.3.29, a servlet engine based on Tomcat version 3.3.2, and a database server based on MySQL version 4.0.17. Experiments were made on nodes running Linux x86, 1.8GHz/1Go, interconnected through a 100Mbps Ethernet network.

We measured the overhead of the repair management system on the CPU of the JEE managed subsystems. As can be seen in Figure 13, this overhead can be considered as negligible. This is due to the out-of-band design of the management system: during nominal operation of the managed JEE

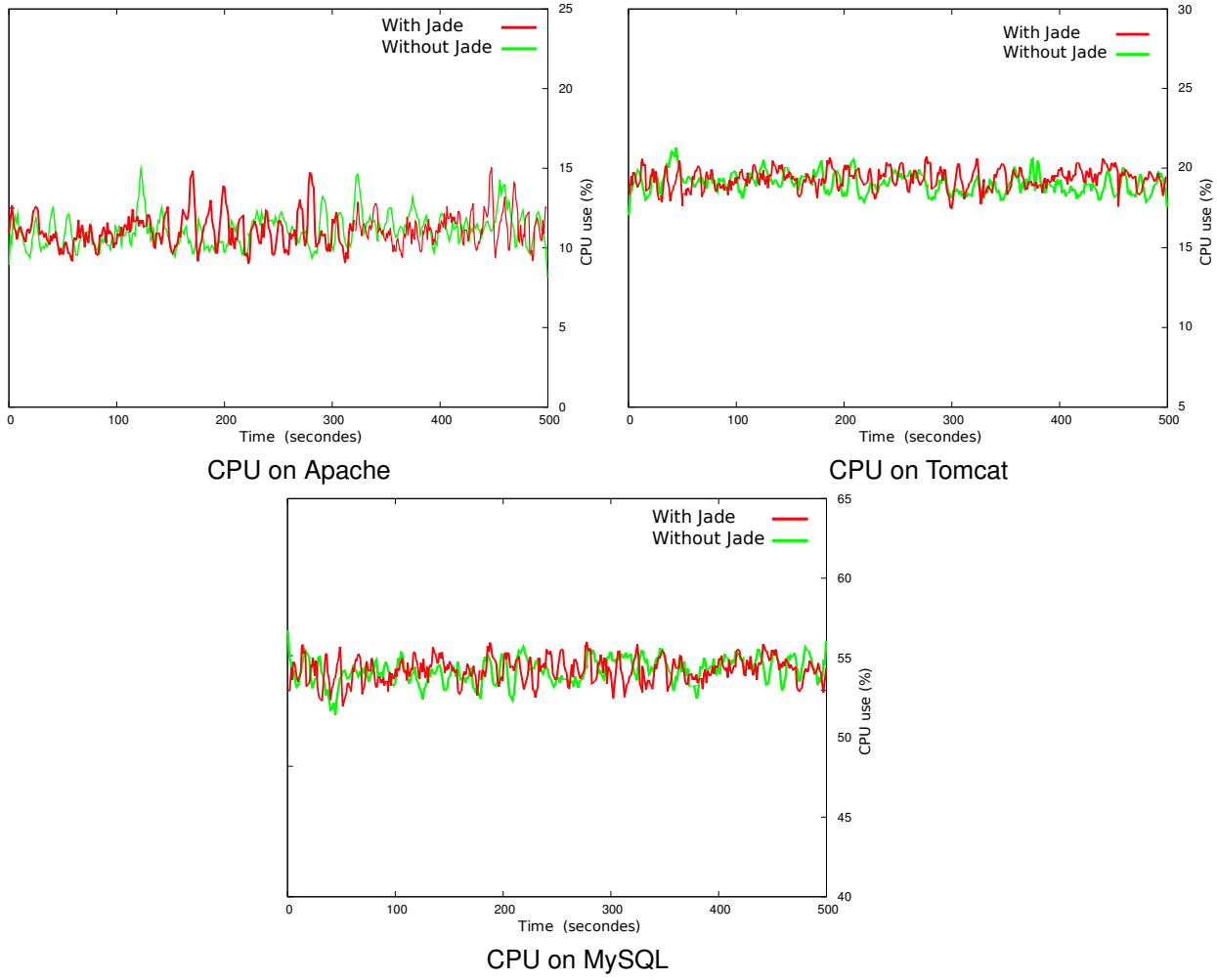


Figure 13: Overhead of the repair management system (CPU)

system, the only cause of overhead is failure detection, since in this case study the communication path between tiers is not modified to enable management. This overhead remained similarly negligible in larger cluster configurations we have experimented with (up to 16 nodes).

We provoked failures on either Apache or Tomcat, as depicted in Figure 14 and Figure 15. the management system detects and repairs the Apache daemon failure within 12 seconds and the Tomcat failure within less than 50 seconds. These numbers include the time for the failure detector to trigger and the time for downloading and installating the necessary software (Rubis, Apache daemon, and Tomcat). They include the installation of the Java wrappers and the application of the management operations, including writing the configuration files from attributes. Ultimately, they also include the time it takes for Apache or Tomcat to start. While Apache is a fast starter, Tomcat is rather slow. While these numbers could be considered large, they are orders of magnitude better than any manual repair time, even by skilled operators.

The repair management system was also applied to a clustered JEE architecture where Apache is used with the *modJK* connector that can load balance requests on multiple replicated Tomcat instances. The management system repaired failed Tomcat instances while maintaining the high availability of the Web server. Because wrappers may actually delay and re-order management operations within the commit of a repair session, the Apache daemon is kept running while the management system repairs the failed Tomcat. It is only stopped and restarted by its wrapper at the very last moment—upon receiving

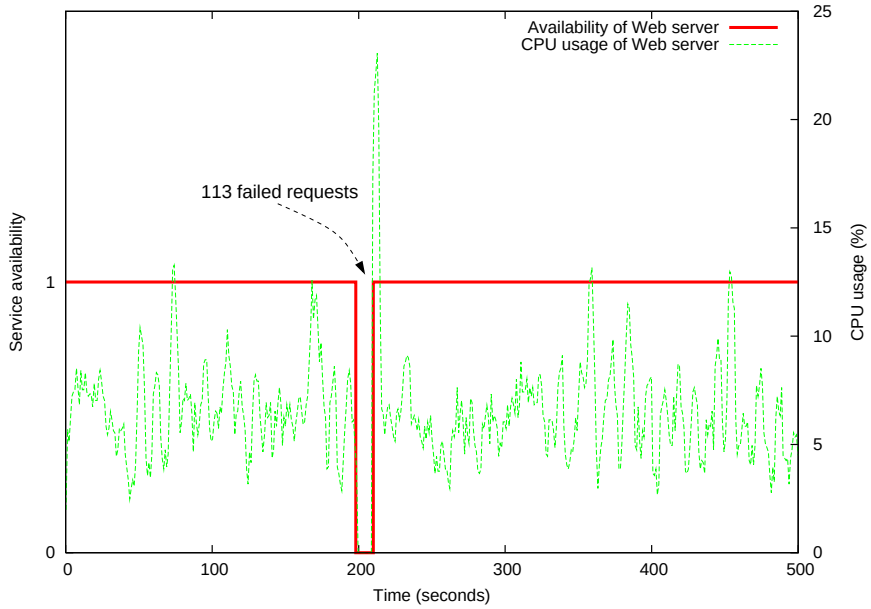


Figure 14: Apache failure

the end-commit. The interruption of service will be down to the time it takes to restart the Apache HTTP daemon. This restart is required by the *modJK* plugin that requires to be stopped to be reconfigured.

Finally, we experimented with the self-repair behavior of the management subsystem itself and its overhead on its ability to repair managed legacy systems. We kept the above failure of a Tomcat but forced a simultaneous failure of one of the autonomic manager replicas. These three failures are detected and handled in this experiment in one repair session. Hence, there is more work to do for repairing not only the lost Tomcat but also the lost replicas of the repair manager and the system map. As above, the repair of Tomcat and of the management subsytem were done without impacting the availability of the Web server (but for the short restart of the Apache daemon).

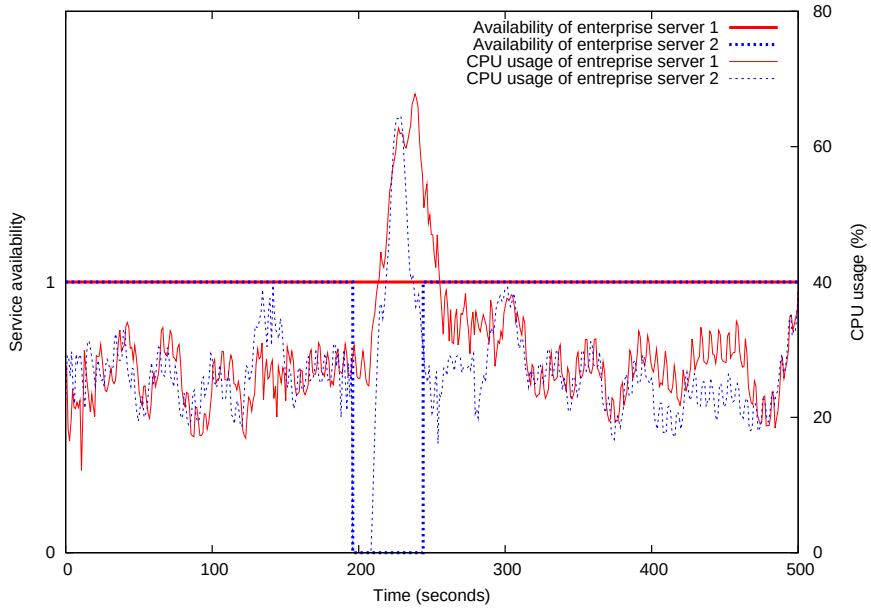


Figure 15: Tomcat failure

References

[1] RPM Package Manager (RPM) v4 Homepage.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.

[3] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd edition, 2008.

[4] A. Flissi, J. Dubus, N. Dolet, and P. Merle. Deploying on the Grid with DeployWare. In *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*. IEEE Computer Society, 2008.

[5] D. Garlan, S.W. Cheng, A.C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10), 2004.

[6] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, Andrew A. Farrell, A. Lain, P. Murray, and P. Toft. The smartfrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43(1), 2009.

[7] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[8] J.L. Hellerstein, Y. Dao, S. Parekh, and D.M. Tilbury. *Feedback Control of Computing Systems*. Wiley-Interscience, 2004.

[9] J. Kramer and J. Magee. Self-managed systems: An architectural challenge. In *Workshop on the Future of Software Engineering (FOSE 2007)*, 2007.

[10] Y. Liu and S. Smith. A Formal Framework for Component Deployment. In *20th ACM Int. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006.

[11] P. Merle and J.B. Stefani. A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA, France, 2008.

[12] The OSGi Alliance. *OSGi Service Platform Release 4, Version 4.1 - Core Specification*, April 2007.

[13] Y. Shoham and K. Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

[14] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe. An Automated Formal Approach to Managing Dynamic Reconfiguration. In *21st IEEE International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006.

Level of confidentiality and dissemination

By default, each document created within Grid4All is © Grid4All Consortium Members and should be considered confidential. Corresponding legal mentions are included in the document templates and should not be removed, unless a more restricted copyright applies (e.g. at subproject level, organisation level etc.).

In the Grid4All Description of Work (DoW), and in the future yearly updates of the 18-months implementation plan, all deliverables listed in Section 7.7 have a specific dissemination level. This dissemination level shall be mentioned in the document (a specific section for this is included in the template, both on the cover page and in the footer of each page).

The dissemination level can be defined for each document using one of the following codes:

PU = Public.
PP = Restricted to other programme participants (including the EC services).
RE = Restricted to a group specified by the Consortium (including the EC services).
CO = Confidential, only for members of the Consortium (including the EC services).
INT = Internal, only for members of the Consortium (excluding the EC services).

This level typically applies to internal working documents, meeting minutes etc., and cannot be used for contractual project deliverables.

It is possible to create later a public version of (part of) a restricted document, under the condition that the owners of the restricted document agree collectively in writing to release this public version. In this case, a new document code should be given so as to distinguish between the different versions.