2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops

Distributed Control Loop Patterns for Managing Distributed Applications [*]

Ahmad Al-Shishtawy,[1] Joel Höglund,[2] Konstantin Popov,[2] Nikos Parlavantzas,[3]
Vladimir Vlassov,[1] and Per Brand[2]

[1] Royal Institute of Technology, Stockholm, Sweden. {ahmadas, vladv}@kth.se
[2] Swedish Institute of Computer Science, Stockholm, Sweden. {joel, kost, perbrand}@sics.se
[3] INRIA, Grenoble, France. nikolaos.parlavantzas@inria.fr

Abstract

In this paper we discuss various control loop patterns for managing distributed applications with multiple control loops. We introduce a high-level framework, called DCMS, for developing, deploying and managing component-based distributed applications in dynamic environments. The control loops, and interactions among them, are illustrated in the context of a distributed self-managing storage service implemented using DCMS to achieve various self- properties.*

Different control loops are used for different self- behaviours, which illustrates one way to divide application management, which makes for both ease of development and for better scalability and robustness when managers are distributed. As the multiple control loops are not completely independent, we demonstrate different patterns to deal with the interaction and potential conflict between multiple managers.*

1 Introduction

Deployment and run-time management of applications constitute a large part of software's total cost of ownership. These costs increase dramatically for distributed applications that are deployed on dynamic environments such as unreliable networks aggregating heterogeneous, poorly managed resources.

The autonomic computing initiative [7] advocates self-configuring, self-healing, self-optimizing and self-protecting (self-* thereafter) systems as a way to reduce the management costs of such applications.

Self-management of a hardware and/or software resource (managed resource thereafter) is achieved through control loops [8]. A control loop continuously monitors the managed resource and acts accordingly. A control loop consists mainly of an autonomic manager and touch points. Touch points enable autonomic managers to sense and affect the managed resource. The autonomic manager function is divided into four phases. Monitoring the managed resource through sensors to find symptoms. Analyzing symptoms and request appropriate change. Planning the requested change. Finally executing the change plan.

Distributed applications require multiple control loops to manage them. Multiple control loops are needed for scalability, robustness, and to simplify programming. In its simplest form there might be one loop per self-* aspect, per application nonfunctional requirement, or per application's subsystems. Usually these loops are not independent but interact and affect each other.

In this paper we present different patterns for constructing distributed control loops to manage distributed applications. These patterns are discussed in the context of a simple component based distributed storage service called *YASS*[1] and implemented using distributed component management system (DCMS) [1][4].

The rest of this paper is organized as follows. In Section 2 we briefly introduce DCMS management system followed by a description of YASS architecture in Section 3. Then in Section 4 we discuss control loop patterns used to self-manage YASS. Followed by related work in Sections 5 and finally conclusions in Section 6.

2 The Distributed Component Management System

DCMS[1][4] is a distributed component management system that facilitates self-management of component based applications deployed on dynamic distributed environments such as community-based Grids.

DCMS separates application's functional and nonfunctional (self-*) code. DCMS is a runtime system that supports both parts. It provides a programming model and

*This research is supported by the FP6 Project Grid4All funded by the European Commission (Contract IST-2006-034567) and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

a matching API for developing application-specific self-* behaviours (control loops). It also supports the functional part by extending the Fractal component model [5] with the concept of component groups and bindings to groups that results in "one-to-all" and "one-to-any" communication patterns , which support scalable, fault-tolerant and self-healing applications [4].

The DCMS runtime system consists of a set of distributed containers that can host components (MEs and application components). The distributed containers are connected together using the Niche overlay network [4]. Niche is self-organising on its own and provides overlay services used by DCMS such as name based communication, distributed hash table, and Publish/Subscribe mechanism.

The self-* code is organized as a network of *management elements* (MEs) interacting through events. This enables the construction of distributed control loops. The self-* code *senses* changes in the environment and can *affect* changes in the architecture – add, remove and reconfigure components and bindings between them. MEs are subdivide into watchers (`W1`, `W2` .. on Figure 1), aggregators (`Aggr1`) and managers (`Mgr1`). There are no exact boundaries but usually watchers are used for monitoring (they try to find symptoms), aggregators are used to analyse symptoms and issue change requests while managers do planning and executing change requests.

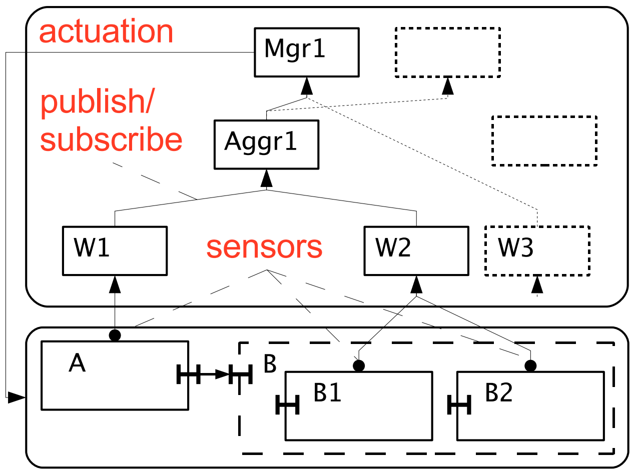

Figure 1. Application architecture.

An application in the framework consists of a component-based implementation of the application's functional specification (the lower part of Figure 1), and a component-based implementation of the application's self-* behaviors (the upper part). The management platform provides for component deployment and communication, and supports sensing and affecting of components.

3 YASS Functional Architecture

YASS [1] is a simple distributed storage service that we will use to illustrate and reason about control loops. YASS stores, reads and deletes files on a set of distributed resources. The service replicates files for the sake of robustness and scalability.

A YASS instance consists out of *front-end components* and *storage components* as shown in Figure 2. The front-end component provides user interface that is used to interact with the storage service. Storage components are composed of *file components* representing stored files. The ovals in Figure 2 represent the available resources in a distributed environment such as resources contributed to a Virtual Organization (VO) or resources in a cluster. Some of the available resources are used to deploy storage components depending on the required total storage capacity. The front-end components are deployed on user machines.

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) that 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 dynamically created new file components. 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 file 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.

4 Control Loops

YASS can be deployed in different distributed sittings with different properties such as reliable clusters or dynamic Grids. This is possible because DCMS separates functional parts (Front-end, Storage, and File components) from non-functional parts (MEs forming control loops). Only the non-functional part needs to be modified in order to run YASS in different distributed settings. In the following subsections we discuss the control loops needed to manage YASS in a dynamic community-based Grid environments, where resources can join, gracefully leave, or fail at any time.

4.1 Basic Control Loops

A basic control loop is a single loop that senses, manages, and affects a managed-resource. Basic control loops

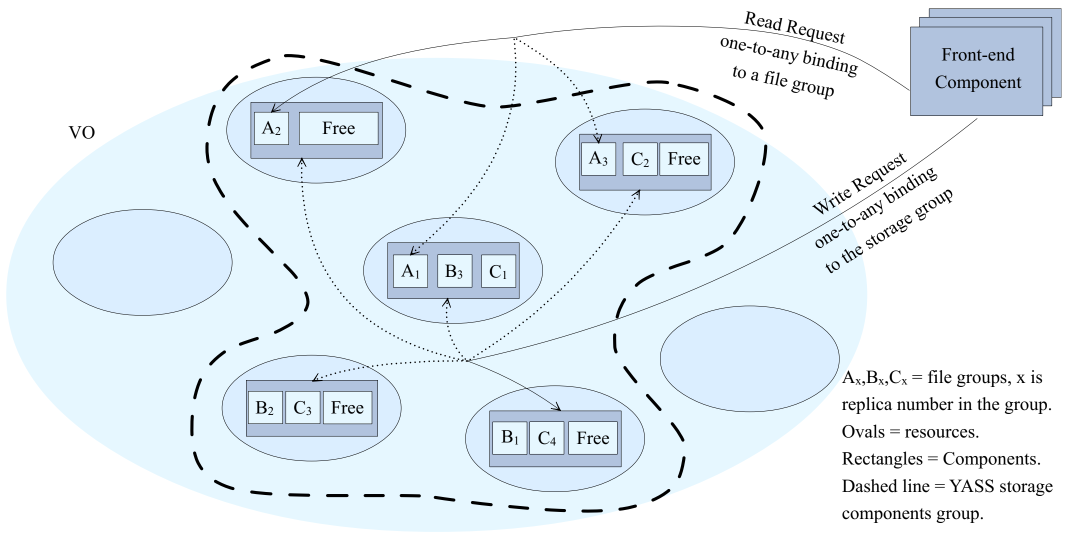

Figure 2. YASS Functional Part

work independently form each other and they form the basic building blocks of more complex loops by linking them together. Below we present two basic control loops used in YASS.

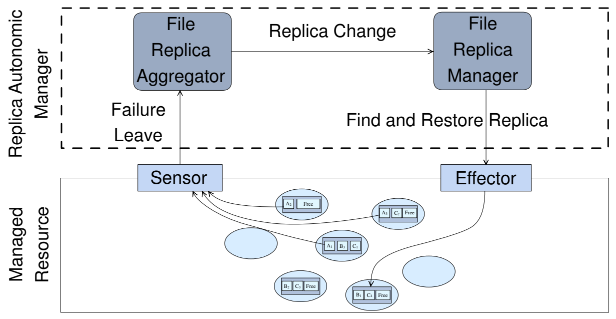

Figure 3. Self-healing control loop.

4.1.1 Self-Healing

Self-healing is concerned with maintaining the desired replication degree for each stored file. The self-healing control loop consists of File-Replica-Aggregator and File-Replica-Manager (Figure 3).

The File-Replica-Aggregator monitors a file group for fail or leave events of its members. This event is triggered when the resource where the file component deployed is about to leave or has failed. The File-Replica-Aggregator response to these events by triggering a replica change event to the File-Replica-Manager that will issue a find and restore replica command.

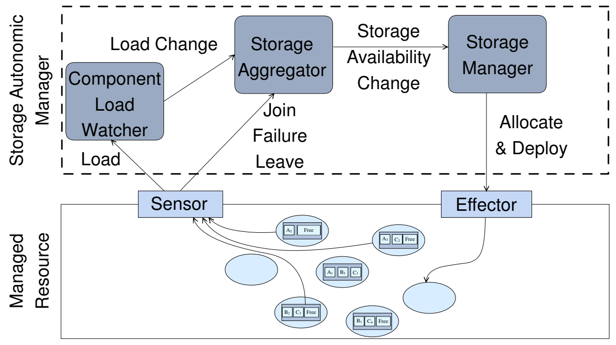

Figure 4. Self-configuration control loop.

4.1.2 Self-Configuration

With self-configuration we mean the ability to adapt YASS in the face of dynamism, thereby maintaining its total storage capacity and total free space to meet functional requirements. The self-configuration control loop consists of Component-Load-Watcher, Storage-Aggregator, and Storage-Manager (Figure 4).

The Component-Load-Watcher monitors the storage group for the total free space available and triggers a load change event when the load is changed by a predefined delta. The Storage-Aggregator is subscribed to the Component-Load-Watcher and fail, leave, and join events. The Storage-Aggregator analyses these events and triggers a storage availability change event when the total capacity or total free space drops below predefined thresholds. The Storage-Manager response to these events by trying to allocate more resources and deploy storage components on them.

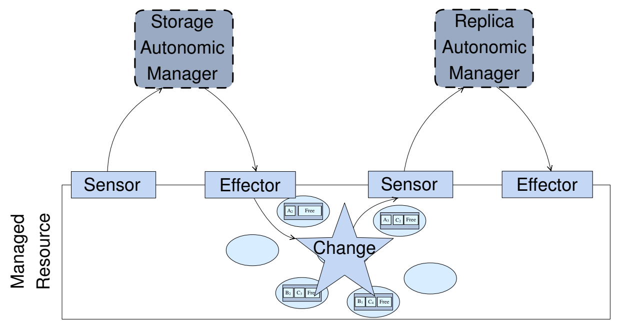

Figure 5. The stigmergy effect.

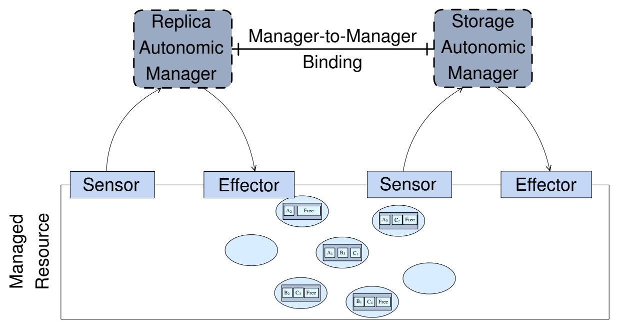

Figure 6. Peer-to-peer management interaction.

4.2 Coordinating Multiple Control Loops

Multiple control loops within the same application are usually not independent because they are managing the same system. Therefore they need to interact and coordinate their actions to avoid conflicts. The interaction take place either through stigmergy, peer-to-peer management interaction, or hierarchical composition, all which are examplified in the following sections.

4.2.1 Stigmergy

Stigmergy is a way of indirect communication and coordination between agents [3]. Agents make changes in their environment and these changes are sensed by other agents and causes 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-resource.

When the utilization of the storage components drops, i.e. the total capacity is above initial requirements and free space is more than a predefined ratio, the Storage-Manager will plan to deallocate some resource. This will be executed using stigmergy (Figure 5). The Storage-Manager will issue a resource leave command. This change in the managed-resource will be sensed and handled by the File-Replica-Manager that will move the file components from the leaving resource to other resources.

4.2.2 Peer-to-Peer Management Interaction

Basic control loops are simple way to achieve self-management. However having multiple independent loops managing the same resource can sometimes cause undesired behaviour. For example, when a resource fails, the Storage-Manager may detect that more storage is needed and start allocating resources and deploying storage components. Meanwhile the File-Replica-Manager will be restoring the files that where on the failed resource. The File-Replica-Manager might fail in restoring the files due to space shortage since the Storage-Manager did not have time to finish. This may also prevent the user temporary from storing files.

If the File-Replica-Manager waited for the Storage-Manager to finish this problem could be avoided. P2P management interaction is used to achieve such behaviours, and can be implemented in DCMS using bindings between the MEs (Figure 6).

Before restoring files the File-Replica-Manager informs the Storage-Manager with 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 until more storage is allocated.

P2P management interaction is a way for managers to cooperate to achieve self-management. It does not mean that a manager controls the other. For example if there is only one replica left of a file, the File-Replica-Manager may ignore the request to wait from the Storage-Manager and proceed with restoring the file anyway.

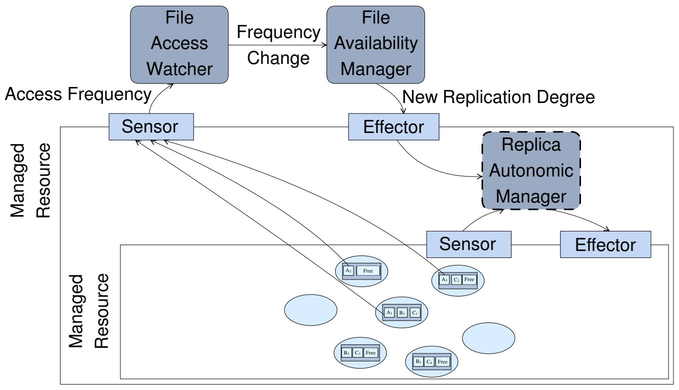

Figure 7. Hierarchical management.

4.2.3 Hierarchical Management

By hierarchical management we mean that some managers can control other managers. Lower level autonomic managers considered as part of the managed-resource for the higher level autonomic managers. Communication is performed using touch points. Higher level managers can sense and affect lower level managers.

A higher level File-Availability control loop can be used to achieve self-optimization. With self-optimization we mean that popular files should have more replicas in order to increase their availability. The control loop consists of File-Access-Watcher and File-Availability-Manager (Figure 7).

The File-Access-Watcher monitors the file access frequency. If a file is popular and accessed frequently then it issues a frequency change event. The File-Availability-Manager may decide to increase the replication degree of that file. This is achieved by increasing the value of the replication degree parameter in the File-Replica-Manager that will start storing more replicas and then maintaining the new replication degree.

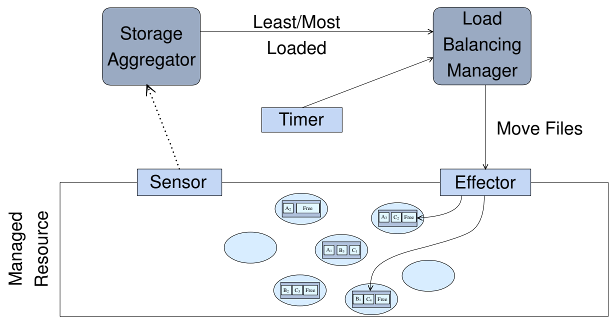

Figure 8. Proactive manager.

4.3 Proactive Managers

All autonomic managers we discussed so far are reactive. They receive events and act upon them. Sometimes proactive managers might be also required. Proactive managers are implemented in DCMS using timer abstraction.

A Load-Balancing control loop can be used for self-optimization by trying to balance the storage among storage components. The Load-Balancing-Manager (Figure 8) wakes up every x minutes and queries the Storage-Aggregator to get the most and least loaded storage components. Then it will move some files from the most to the least loaded storage component. This will achieve lazy load balancing. This example also shows that some parts of the control loop might be shared among different control loop as the Storage-Aggregator.

5 Related Work

The vision of autonomic management as presented in [7] has given rise to a number of proposed solutions to aspects of the problem.

An attempt to analyze and understand how multiple interacting loops can manage a single system has been done in [9] by studying and analysing existing systems such as biological and software systems. By this study the authors try to understand the rules of a good control loop design. A study how to compose multiple loops and ensure that they are consistent and complementary is presented in [6]. The authors presented an architecture that supports such compositions.

A reference architecture for autonomic compoting is presented in [10]. The authors present patterns for applying their proposed architecture to solve specific problems common to self-managing applications. Behavioural Skeletons is a technique presented in [2] that uses algorithmic skeletons to encapsulate general control loop features that can later be specialized to fit a specific application.

6 Conclusions

Our distributed component management system (DCMS) provides a high-level programming framework for constructing basic management control loops. It is a event-based framework that provides sensing/affecting support in a distributed environment and makes use of groups and group bindings to simplify designing and constructing management feedback control loops.

In this paper we demonstrate various patterns for control loops, and discuss possible interactions between them. For reasons of scalability, robustness, ease of use and separation of concerns the management of most distributed applications should consist of multiple control loops, logically separate and running on different machines. The interactions discussed range from harmless stigmergy to harmful competition and disruption. In the latter case the solution patterns of peer-to-peer management interaction and hierarchical management are illustrated.

To summarize, we use the DCMS to develop separate control loops, over, for instance, different aspect of self-*. We study the interaction between the managers and where needed make use of one of the described management interaction patterns to deal with contention. Our studies, as described in this paper, support the position that this approach is a useful one for developing self-managing distributed applications.

References

[1] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand. Enabling self-management of component based distributed applications. In *CoreGRID Symposium*, Las Palmas, Spain, August 2oo8.

[2] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto. Behavioural skeletons in gcm: Autonomic management of grid components. In *PDP'08*, pages 54–63, Washington, DC, USA, 2008.

[3] E. Bonabeau. Editor's introduction: Stigmergy. *Artificial Life*, 5(2):95–96, 1999.

[4] P. Brand, J. Höglund, K. Popov, N. de Palma, F. Boyer, N. Parlavantzas, V. Vlassov, and A. Al-Shishtawy. The role of overlay services in a self-managing framework for dynamic virtual organizations. In *CoreGRID Workshop, Crete, Greece*, June 2007.

[5] E. Bruneton, T. Coupaye, and J.-B. Stefani. The fractal component model. Technical report, France Telecom R&D and INRIA, Feb. 5 2004.

[6] S.-W. Cheng, A.-C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste. An architecture for coordinating multiple self-management systems. In *WICSA '04*, page 243, Washington, DC, USA, 2004.

[7] P. Horn. Autonomic computing: IBM's perspective on the state of information technology, Oct. 15 2001.

[8] IBM. An architectural blueprint for autonomic computing, 4th edition. http://www-03.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf, June 2006.

[9] P. V. Roy, S. Haridi, A. Reinefeld, J.-B. Stefani, R. Yap, and T. Coupaye. Self management for large-scale distributed systems: An overview of the selfman project. In *FMCO '07: Software Technologies Concertation on Formal Methods for Components and Objects*, Amsterdam, The Netherlands, Oct 2007.

[10] J. W. Sweitzer and C. Draper. *Autonomic Computing: Concepts, Infrastructure, and Applications*, chapter 5: Architecture Overview for Autonomic Computing, pages 71–98. CRC Press, 2006.