2009 International Conference on Computational Science and Engineering

A Design Methodology for Self-Management in Distributed Environments

Ahmad Al-Shishtawy*, Vladimir Vlassov*, Per Brand†, and Seif Haridi*†
*Royal Institute of Technology, Stockholm, Sweden
{ahmadas, vladv, haridi}@kth.se
†Swedish Institute of Computer Science, Stockholm, Sweden
{perbrand, seif}@sics.se

***Abstract*—Autonomic computing is a paradigm that aims at reducing administrative overhead by providing autonomic managers to make applications self-managing. In order to better deal with dynamic environments, for improved performance and scalability, we advocate for distribution of management functions among several cooperative managers that coordinate their activities in order to achieve management objectives. We present a methodology for designing the management part of a distributed self-managing application in a distributed manner. We define design steps, that includes partitioning of management functions and orchestration of multiple autonomic managers. We illustrate the proposed design methodology by applying it to design and development of a distributed storage service as a case study. The storage service prototype has been developed using the distributing component management system *Niche*. Distribution of autonomic managers allows distributing the management overhead and increased management performance due to concurrency and better locality.**

***Keywords*-autonomic computing; control loops; self-management; distributed systems;**

I. Introduction

Autonomic computing [1] is an attractive paradigm to tackle management overhead of complex applications by making them self-managing. Self-management, namely self-configuration, self-optimization, self-healing, and self-protection (self-* thereafter), is achieved through autonomic managers [2]. An autonomic manager continuously monitors hardware and/or software resources and acts accordingly. Managing applications in dynamic environments (like community Grids and peer-to-peer applications) is specially challenging due to high resource churn and lack of clear management responsibility.

A distributed application requires multiple autonomic managers rather than a single autonomic manager. Multiple managers are needed for scalability, robustness, and performance and also useful for reflecting separation of concerns. Engineering of self-managing distributed applications executed in a dynamic environment requires a methodology for building robust cooperative autonomic managers. The methodology should include methods for management decomposition, distribution, and orchestration. For example, management can be decomposed into a number of managers each responsible for a specific self-* property or alternatively application subsystems. These managers are not independent but need to cooperate and coordinate their actions in order to achieve overall management objectives.

The major contributions of the paper are as follows. We propose a methodology for designing the management part of a distributed self-managing application in a distributed manner, i.e. with multiple interactive autonomic managers. Decentralization of management and distribution of autonomic managers allows distributing the management overhead, increasing management performance due to concurrency and/or better locality. Decentralization does avoid a single point of failure however it does not necessarily improve robustness. We define design steps, that includes partitioning of management, assignment of management tasks to autonomic managers, and orchestration of multiple autonomic managers. We describe a set of patterns (paradigms) for manager interactions.

We illustrate the proposed design methodology including paradigms of manager interactions by applying it to design and development of a distributed storage service as a case study. The storage service prototype has been developed using the distributing component management system *Niche*[1] [3]–[5].

The remainder of this paper is organized as follows. Section II describes Niche and relate it to the autonomic computing architecture. Section III presents the steps for designing distributed self-managing applications. Section IV focuses on orchestrating multiple autonomic managers. In Section V we apply the proposed methodology to a distributed file storage as a case study. Related work is discussed in Section VI followed by conclusions and future work in Section VII.

II. The Distributed Component Management System

The autonomic computing reference architecture proposed by IBM [2] consists of the following five building blocks.

- **Touchpoint:** consists of a set of sensors and effectors used by autonomic managers to interact with managed resources (get status and perform operations). Touchpoints are components in the system that implement a uniform

This research is supported by the FP6 projects SELFMAN (contract IST-2006-034084) and Grid4All (contract IST-2006-034567) funded by the European Commission.

[1]In our previous work [3], [4] our distributing component management system *Niche* was called DCMS

978-0-7695-3823-5/09 $26.00 © 2009 IEEE
DOI 10.1109/CSE.2009.301

IEEE computer society

management interface that hides the heterogeneity of managed resources. A managed resource must be exposed through touchpoints to be manageable.

- **Autonomic Manager:** is the key building block in the architecture. Autonomic managers are used to implement the self-management behaviour of the system. This is achieved through a control loop that consists of four main stages: monitor, analyze, plan, and execute. The control loop interacts with the managed resource through the exposed touchpoints.
- **Knowledge Source:** is used to share knowledge (e.g. architecture information and policies) between autonomic managers.
- **Enterprise Service Bus:** provides connectivity of components in the system.
- **Manager Interface:** provides an interface for administrators to interact with the system. This includes the ability to monitor/change the status of the system and to control autonomic managers through policies.

The use-case presented in this paper has been developed using the distributed component management system *Niche* [3], [4]. Niche implements the autonomic computing architecture described above. Niche includes a distributed component programming model, APIs, and a run-time system including deployment service. The main objective of Niche is to enable and to achieve self-management of component-based applications deployed on dynamic distributed environments such as community Grids. A self-managing application in Niche consists of functional and management parts. Functional components communicate via bindings, whereas management components communicate mostly via a publish/subscribe event notification mechanism.

The Niche run-time environment is a network of distributed containers hosting functional and management components. Niche uses a structured overlay network (Niche [4]) as the enterprise service bus. Niche is self-organising on its own and provides overlay services used by Niche such as name-based communication, distributed hash table (DHT) and a publish/subscribe mechanism for event dissemination. These services are used by Niche to provide higher level communication abstractions such as name-based bindings to support component mobility; dynamic component groups; one-to-any and one-to-all bindings, and event based communication.

For implementing the touchpoints, Niche leverages the introspection and dynamic reconfiguration features of the Fractal component model [6] in order to provide sensors and actuation API abstractions. Sensors are special components that can be attached to the application's functional components. There are also built-in sensors in Niche that sense changes in the environment such as resource failures, joins, and leaves, as well as modifications in application architecture such as creation of a group. The actuation API is used to modify the application's functional and management architecture by adding, removing and reconfiguring components, groups, bindings.

The Autonomic Manager (a control loop) in Niche is organized as a network of *Management Elements* (MEs) interacting through events, monitoring via sensors and acting using the actuation API. This enables the construction of distributed control loops. MEs are subdivided into watchers, aggregators, and managers. Watchers are used for monitoring via sensors and can be programmed to find symptoms to be reported to aggregators or directly to managers. Aggregators are used to aggregate and analyse symptoms and to issue change requests to managers. Managers do planning and execute change requests.

Knowledge in Niche is shared between MEs using two mechanisms: first, using the publish/subscribe mechanism provided by Niche; second, using the Niche DHT to store/retrieve information such as component group members, name-to-location mappings.

III. Steps in Designing Distributed Management

A self-managing application can be decomposed into three parts: the functional part, the touchpoints, and the management part. The design process starts by specifying the functional and management requirements for the functional and management parts, respectively. In the case of Niche, the functional part of the application is designed by defining interfaces, components, component groups, and bindings. The management part is designed based on management requirements, by defining autonomic managers (management elements) and the required touchpoints (sensors and effectors).

An Autonomic Manager is a control loop that senses and affects the functional part of the application. For many applications and environments it is 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. Decomposition of management can be motivated by different reasons such as follows. It allows avoiding a single point of failure. It may be required to distribute the management overhead among participating resources. Self-managing a complex system may require more than one autonomic manager to simplify design by separation of concerns. Decomposition can also be used to enhance the management performance by running different management tasks concurrently and by placing the autonomic managers closer to the resources they manage.

We define 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 (e.g. tasks are defined based which self-* properties they implement) or spacial (e.g. tasks are defined based on the structure of the managed application). The major design issue to be considered at this step is granularity of tasks assuming that a task or a group of related tasks can be performed by a single manager.

Assignment: The tasks are then assigned to autonomic managers each of which becomes responsible for one or more management tasks. Assignment can

be done based on self-* properties that a task belongs to (according to the functional decomposition) or based on which part of the application that task is related to (according to the spatial decomposition).
Orchestration: Although autonomic managers can be designed independently, 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, i.e. to nodes of the distributed environment. A major issue to be considered at this step is optimized placement of managers and possibly functional components on nodes in order to improve management performance.

In this paper our major focus is on the orchestration of autonomic managers as the most challenging and less studied problem. The actions and objectives of the other stages are more related to classical issues in distributed systems such as partitioning and separation of concerns, and optimal placement of modules in a distributed environment.

IV. Orchestrating Autonomic Managers

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

A. Stigmergy

Stigmergy is a way of indirect communication and coordination between agents [7]. 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.

The stigmergy effect is, in general, unavoidable when you have more than one autonomic manager and can cause undesired behaviour at runtime. Hidden stigmergy makes it challenging to design a self-managing system with multiple autonomic managers. However stigmergy can be part of the design and used as a way of orchestrating autonomic managers (Fig. 1).

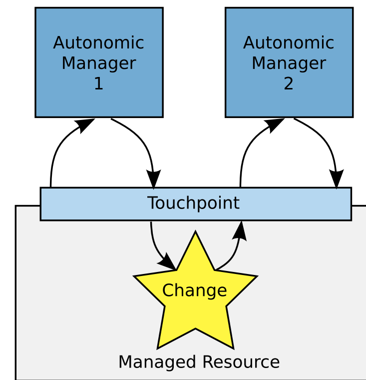

Fig. 1. The stigmergy effect.

B. Hierarchical Management

By hierarchical management we mean that some autonomic managers can monitor and control other autonomic managers (Fig. 2). The lower level autonomic managers 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.

Autonomic managers at different levels often operate at different time scales. Lower level autonomic managers are used to manage changes in the system that need immediate actions. Higher level autonomic managers are often slower and used to regulate and orchestrate the system by monitoring global properties and tuning lower level autonomic managers accordingly.

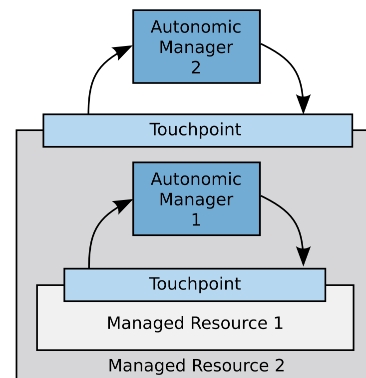

Fig. 2. Hierarchical management.

C. 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 (Fig. 3). Cross autonomic manager bindings can be used to coordinate autonomic managers and avoid undesired behaviors such as race conditions or oscillations.

D. Shared Management Elements

Another way for autonomic managers to communicate and coordinate their actions is by sharing management elements (Fig. 4). This can be used to share state (knowledge) and to synchronise their actions.

V. Case Study: A Distributed Storage Service

In order to illustrate the design methodology, we have developed a storage service called YASS (Yet Another Storage Service) [3], using Niche. The case study illustrates how to design a self-managing distributed system monitored and controlled by multiple distributed autonomic managers.

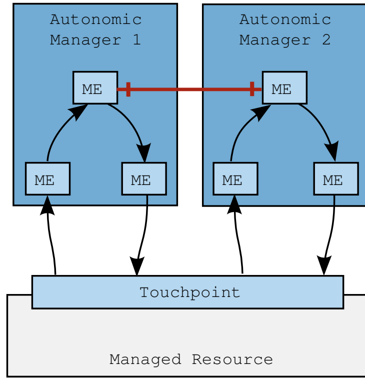

Fig. 3. Direct interaction.

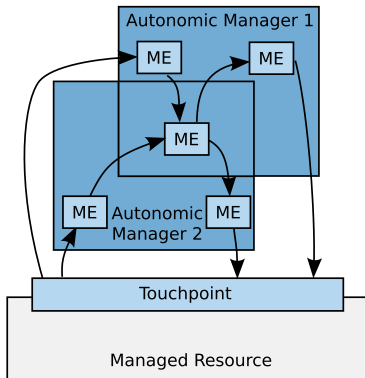

Fig. 4. Shared Management Elements.

A. 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 functions are required in order to make the storage service self-managing in the presence of dynamicity in resources and load: the service should tolerate the resource churn (joins/leaves/failures), optimize usage of resources, and resolve hot-spots. We define the following tasks based on the functional decomposition of management according to self-* properties (namely self-healing, self-configuration, and self-optimization) to be achieved.

- 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;
- 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.

B. YASS Functional Design

A YASS instance consists of *front-end components* and *storage components* as shown in Fig. 5. 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.

C. Enabling Management of YASS

Given that the functional part of YASS has been developed, to manage it we need to provide touchpoints. Niche 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.

D. Self-Managing YASS

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

1) 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 Fig. 6.

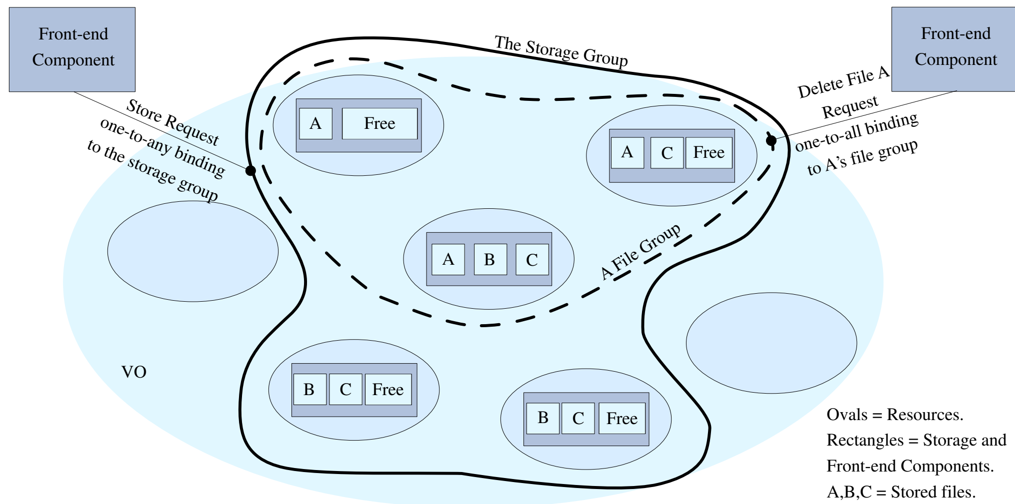

Fig. 5. YASS Functional Part

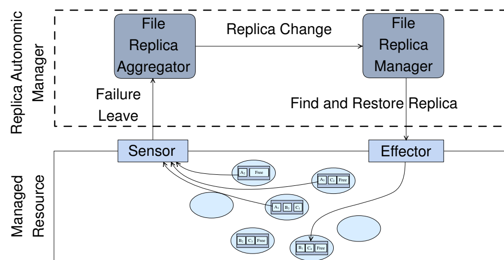

Fig. 6. Self-healing control loop.

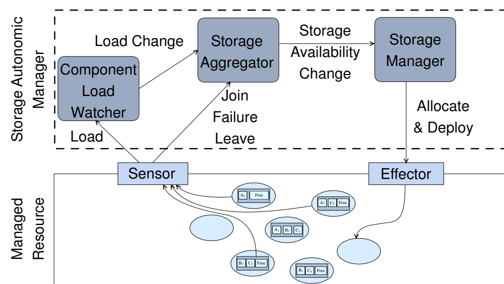

Fig. 7. Self-configuration control loop.

The File-Replica-Aggregator monitors a file group, containing the subset of storage components that host the file replicas, by subscribing to resource fail or leave events caused by any of the group members. These events are received when a resource, on which a component member in the group is deployed, is about to leave or has failed. The File-Replica-Aggregator responds to these events by triggering a replica change event to the File-Replica-Manager that will issue a find and restore replica command.

2) Storage Autonomic Manager: The storage autonomic manager is responsible for maintaining 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 reconfigures 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 YASS. The storage autonomic manager consists of Component-Load-Watcher, Storage-Aggregator, and Storage-Manager as shown in Fig. 7.

The Component-Load-Watcher monitors the storage group, containing all storage components, for changes in the total free space available by subscribing to the load sensors events. The Component-Load-Watcher will trigger a load change event when the load is changed by a predefined delta. The Storage-Aggregator is subscribed to the Component-Load-Watcher load change event and the resource fail, leave, and join events (note that the File-Replica-Aggregator also subscribes to the resource failure and leave events). The Storage-Aggregator, by analyzing these events, will be able to estimate the total storage capacity and the total free space. The Storage-Aggregator will trigger a storage availability change event when the total and/or free storage space drops below a predefined thresholds. The Storage-Manager responds to this event by trying to allocate more resources and deploying storage components on them.

3) 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.

The direct coordination used here does not mean that one 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.

4) Optimising Allocated Storage : Systems should maintain high resource utilization. The storage autonomic manager allocates additional resources if needed to guarantee the ability to store files. However, users might delete files later causing the utilization of the storage space to drop. It is desirable that YASS be able to self-optimize itself by releasing excess resources to improve utilization.

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.

We believe that this is better than adding another autonomic manager for following two reasons: first, it allows avoiding duplication of functionality; and second, it allows avoiding oscillation between allocation and releasing resources by keeping the decision about the proper amount of storage at one place.

5) 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

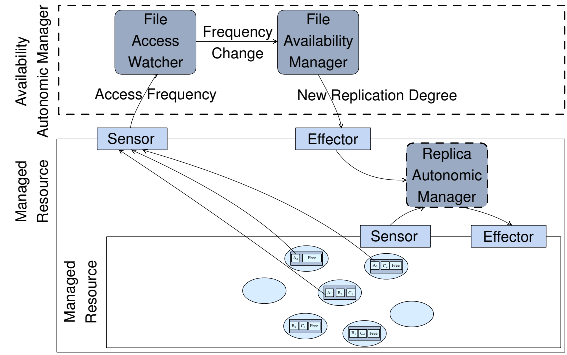

Fig. 8. Hierarchical management.

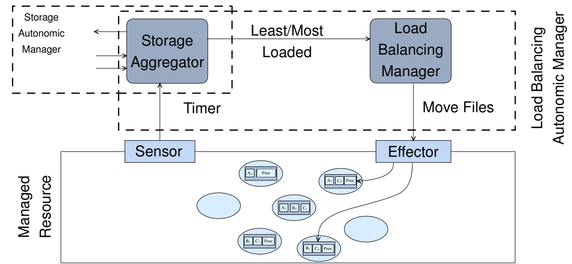

Fig. 9. Sharing of Management Elements.

File-Access-Watcher and File-Availability-Manager shown in Fig. 8 illustrate hierarchical management.

The File-Access-Watcher monitors the file access frequency. If the popularity of a file changes dramatically it issues a frequency change event. The File-Availability-Manager may decide to change the replication degree of that file. This is achieved by changing the value of the replication degree parameter in the File-Replica-Manager.

6) 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 Fig. 9.

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 Niche using a timer abstraction.

The load balancing autonomic manager is triggered, by a timer, every x time units. The timer event will be received by the shared Storage-Aggregator that will trigger an event containing the most and least loaded storage components. This event will be received by the Load-Balancing-Manager that will move some files from the most to the least loaded storage component.

VI. Related Work

The vision of autonomic management as presented in [1] 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 [8] 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 [9]. The authors presented an architecture that supports such compositions.

A reference architecture for autonomic computing 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 [11] that uses algorithmic skeletons to encapsulate general control loop features that can later be specialized to fit a specific application.

VII. Conclusions and Future Work

We have presented the methodology of developing the management part of a self-managing distributed application in distributed dynamic environment. We advocate for multiple managers rather than a single centralized manager that can induce a single point of failure and a potential performance bottleneck in a distributed environment. The proposed methodology includes four major design steps: decomposition, assignment, orchestration, and mapping (distribution). The management part is constructed as a number of cooperative autonomic managers each responsible either for a specific management function (according to functional decomposition of management) or for a part of the application (according to a spatial decomposition). We have defined and described different paradigms (patterns) of manager interactions, including indirect interaction by stigmergy, direct interaction, sharing of management elements, and manager hierarchy. In order to illustrate the design steps, we have developed and presented in this paper a self-managing distributed storage service with self-healing, self-configuration and self-optimizing properties provided by corresponding autonomic managers, developed using the distributed component management system Niche. We have shown how the autonomic managers can coordinate their actions, by the four described orchestration paradigms, in order to achieve the overall management objectives.

Dealing with failure of autonomic managers (as opposed to functional parts of the application) is out of the scope of this paper. Clearly, by itself, decentralization of management, might make the application more robust (as some aspects of management continue working, while others stop), but also more fragile due to increased risk of partial failure. In both the centralized and decentralized case, techniques for fault tolerance are needed to insure robustness. Many of these techniques, while ensuring fault recovery do so with some significant delay, in which case a decentralized management architecture may prove advantageous as only some aspects of management are disrupted at any one time.

Our future work includes refinement of the design methodology, further case studies with the focus on orchestration of autonomic managers, investigating robustness of managers by transparent replication of management elements.

Acknowledgements

We would like to thank the Niche research team including Konstantin Popov and Joel Höglund from SICS, and Nikos Parlavantzas from INRIA.

References

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

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

[3] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Enabling self-management of component based distributed applications," in *From Grids to Service and Pervasive Computing*, T. Priol and M. Vanneschi, Eds. Springer US, July 2008, pp. 163–174.

[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 *Making Grids Work*, M. Danelutto, P. Fragopoulou, and V. Getov, Eds. Springer US, 2007, pp. 153–164.

[5] Niche homepage. [Online]. Available: http://niche.sics.se/

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

[7] E. Bonabeau, "Editor's introduction: Stigmergy," *Artificial Life*, vol. 5, no. 2, pp. 95–96, 1999. [Online]. Available: http://www.mitpressjournals.org/doi/abs/10.1162/106454699568692

[8] 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.

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

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

[11] 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*, Washington, DC, USA, 2008, pp. 54–63.