# Endogenous Versus Exogenous Self-Management

Danny Weyns, Robrecht Haesevoets, Bart Van Eylen,
Alexander Helleboogh, Tom Holvoet, Wouter Joosen

DistriNet Labs
Katholieke Universiteit Leuven
3001 Leuven, Belgium
{danny.weyns}@cs.kuleuven.be

## ABSTRACT

Self-management is considered as one of the crucial means for software systems to deal with changing demands at runtime. Self-management endows a software systems with the ability to adapt its structure or behavior without human intervention. Two different approaches are put forward for self-management: (1) the system components adapt their structure or behavior to changing requirements and cooperatively realize system adaptation—this approach can be considered as *endogenous* self-management; (2) the system is adapted through a control loop, i.e. the system is monitored to maintain an explicit representation of the system and based on a set of high-level objectives, the system structure or its behavior is adapted—this approach can be considered as *exogenous* self-management.

In this paper, we introduce a hybrid software architecture that combines both approaches. A multi-agent system architecture allows agents to flexibly adapt their behavior to changes in their context providing cooperative system adaptation. Then, we extend the multi-agent system architecture with a decentralized control loop adding self-healing properties to the system. We use intelligent monitoring of traffic jams as an illustrative case.

## Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Design, Software Architectures

## General Terms

Design

## 1. INTRODUCTION

Self-management is considered as one of the crucial means for software systems to deal with changing demands at runtime. The general idea of self-management is to endow computing systems with the ability to manage themselves according to high-level objectives specified by humans. To enable self-management, two different approaches are put forward. In the first approach, the system components themselves adapt their structure or behavior to changing requirements and cooperatively realize system adaptation. This approach can be considered as *endogenous* self-management. In the second appraoch, an additional subsystem monitors the system at runtime maintaining a representation of the system, and based on a set of high-level objectives, the subsystem adapts the structure or behavior of the system. In this approach, a control loop is added which monitors the system and adapts the system accordingly. This approach can be considered as *exogenous* self-management.

The two approaches for self-management are often considered as two extreme poles. In practice, the line between both is rather blurred, and compromises will often lead to an engineering approach incorporating representatives from these two extreme poles [1]. A major challenge is to accommodate a systematic engineering approach that integrates both approaches.

This paper aims to contribute to a better understanding of the two approaches for self-management. In particular, we introduce a hybrid software architecture that combines both approaches. A multi-agent system architecture allows agents to flexibly adapt their behavior to changes in their context providing cooperative system adaptation. Then, we extend the multi-agent system architecture with a decentralized control loop adding self-healing properties to the system. We use intelligent monitoring of traffic jams as an illustrative case.

The remainder of the paper is structured as follows. We start by introducing the traffic case in section 2. In section 3, we explain the multi-agent architecture that allows agents—supported by a middleware—to adapt their behavior to changing operating conditions. Section 4 shows how we have extended the multi-agent system architecture with support for self-healing. We discuss related work in section 5. Finally, we draw conclusions.

## 2. INTELLIGENT MONITORING OF TRAFFIC JAMS

The monitoring application we consider fits in the domain of intelligent transportation systems, a worldwide initiative to exploit information and communication technology to improve traffic [11, 5]. The system consists of a set of intelligent cameras which are distributed evenly along the highway, as shown in figure 1. Each camera has a limited viewing range and cameras are placed to get an optimal coverage of the highway with a minimum in overlap. A camera is able to measure the traffic conditions within its viewing range and determine whether there is a traffic jam or not in its viewing range. Each camera is equipped with a data processing unit, capable of processing the monitored data, and a communication unit to communicate with other cameras. The task of the cameras is to detect and monitor traffic jams on the highway in a decentralized way, avoiding the bottleneck of a centralized control center. Possible clients of the monitoring system are traffic light controllers, driver assistance systems, etc.

Traffic jams can cover the viewing range of multiple cameras and can dynamically grow and dissolve. To monitor a traffic jam,


*SEAMS'08,* May 12–13, 2008, Leipzig, Germany.

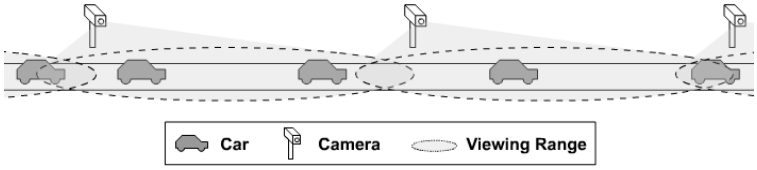


**Figure 1: An example of a highway with traffic cameras.**

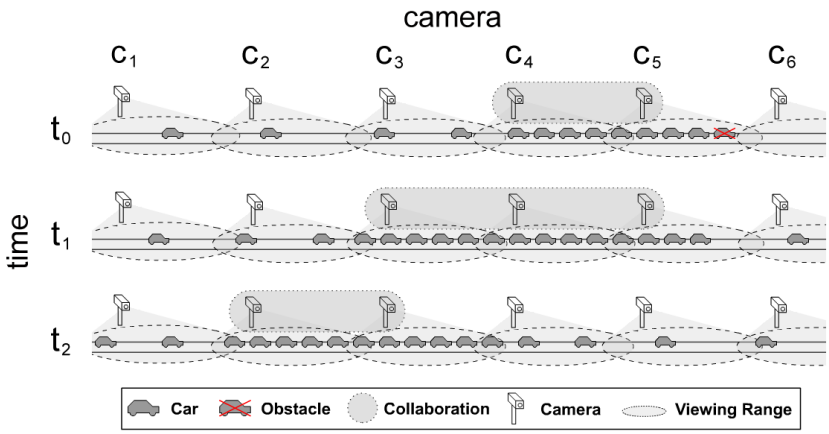


**Figure 2: An example of collaborations between traffic cameras.**

data observed by multiple cameras has to be aggregated. Because there is no central point of control, cameras have to collaborate and distribute the aggregated data to the clients.

By default each camera monitors the traffic conditions of the traffic within its viewing range. When a traffic jam occurs, the camera has to collaborate with other cameras detecting the same traffic jam. In the collaboration, the data each camera is monitoring is aggregated to get a complete image of the traffic jam. Cameras will enter or leave the collaboration whenever the traffic jam enters or leaves their viewing range.

An example of such a collaboration is shown in figure 2. The traffic jam, situated between $C_4$ and $C_5$, has entered the viewing range of camera $C_4$ at $t_0$. Camera $C_4$ and $C_5$ start to collaborate because they are now both monitoring the same traffic jam. At $t_1$, the accident is solved but the traffic jam has further grown and entered the viewing range of camera $C_3$. Therefore, camera $C_3$ now participates in the collaboration between cameras $C_4$ and $C_5$. At $t_2$, the traffic jam has entered the viewing range of camera $C_2$ but has dissolved in the viewing range of cameras $C_4$ and $C_5$. $C_4$ and $C_5$ have stopped collaborating while camera $C_2$ is collaborating with camera $C_3$. This example scenario illustrates how the collaboration between the cameras is driven by the context.

The dynamic nature of the traffic phenomena demands for dynamic collaborations between the cameras. Organizations will evolve dynamically according to the current traffic conditions, which make up the context of the highway.

# 3. A MULTI-AGENT SYSTEM ARCHITECTURE FOR SELF-ADAPTATION

A multi-agent system structures the software as a number of interacting autonomous entities (agents) that are situated in an environment. Control in multi-agent system is decentralized, the system functionalities and qualities result from the local decisions of the agents, their actions in the environment, and the interactions among the agents. Multi-agent systems provide a way to model self-adapting systems. The self-adapting properties of a multi-agent system are based on the agents' capabilities to flexibly adapt their behavior to dynamic and changing circumstances. As such, the self-managing properties of a multi-agent system are endogenous to the system.

Structuring and managing interactions among agents is a crucial part of the design of any multi-agent system. A typical way to manage these interactions is by means of organizations in which agents play roles [14]. To deal with the ongoing dynamics and changes, organizations have to adapt. Most of the existing work on organizations in multi-agent systems defines roles and organizations at the level of agents [6, 15, 4]. As such, agents have a dual responsibility: on the one hand agents play roles providing the associated functionality in the organization, on the other hand agents are responsible to set up and manage organizations, and deal with the complexity of organization dynamics.

In our research, we have developed an approach called *context-driven dynamic organizations* that considers an organization as a first-class abstraction which is explicitly supported by dedicated multi-agent system middleware [10]. The middleware takes the burden of managing organizations and their dynamics. Driven by the context, the middleware manages the evolution of organizations and actively advertises roles to agents, supporting the necessary collaborations between agents needed in the current context. The proposed approach separates the management of dynamic evolution of organizations from the actual functionality provided by the agents playing roles in the organizations. Separating these concerns makes it easier to understand, design, and manage organizations in multi-agent systems.

## 3.1 Software Architecture

Figure 3 shows a layered view of the software architecture of the multi-agent system. The main drivers behind the architecture are scalability and robustness. Each layer is allowed to use ser-

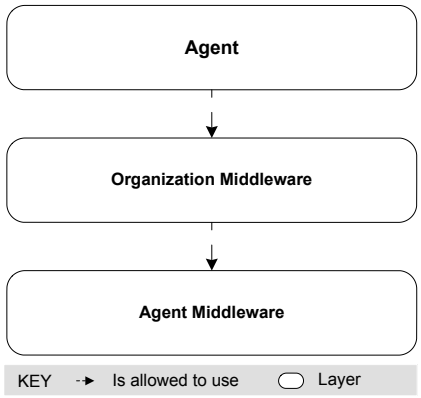


**Figure 3: Layered Module View.**

vices offered by the layer directly beneath it. The layered style allows for a clean integration of the organization concepts with basic agent middleware support. The `Agent Middleware Layer` provides basic services in multi-agent systems [19], including support for perception, action, and communication. Perception provides a service to agents for sensing the context in which they are situated. Action provides a service to act in the environment. The communication service supports the exchange of messages in the distributed setting. On top of the agent middleware layer, we have the `Organization Layer` which provides support for dynamic organizations. The layer encapsulates the management of dynamic evolution of organizations and it provides role-specific services to the agents for perception, action, and communication.

## 3.2 Organization Management

Figure 4 shows the top-level decomposition of the organization middleware. We briefly explain the responsibilities of the components and their relationships.

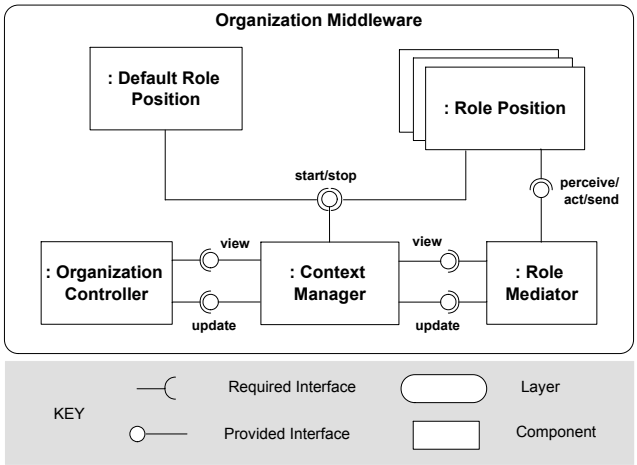


**Figure 4: Components of Organization Middleware Layer.**

The `Context Manager` is responsible for managing context information relevant for the organizations. Context information includes information about the roles agents play in organizations and additional organization-specific data. On the one hand, the context information is used to provide specific services to agents playing particular roles. For example, in the traffic case, for each organization there is one agent playing the role of data aggregator. Agents in this role have the capability to interact with all the other agents in the organization playing the monitoring role. On the other hand, context information is used to dynamically adapt organizations. For example, organizations in the traffic case are adapted based on the current traffic congestion and the spatial position of cameras. The context manager keeps the context information up to date using the services of the agent middleware. Besides, the context manager exports the `view` interface to access the context information and the `update` interface that allows other components to update the context information.

The `Role Mediator` mediates agent activities. Examples of mediation are sending a message to all agents playing a particular role in an organization, or enforcing agents to follow particular interaction patterns, i.e. protocols. To this end the role mediator accesses the context information using the view and update interfaces of the context manager.

The `Organization Controller` is responsible for dynamically adapting existing organizations. The organization controller contains a number of application-specific `evolution laws`. Evolution laws determine the organization changes in the system based on the actual context that is maintained by the context manager. The organization controller uses the view interface of the context manager to inspect the actual context information. When the organization controller decides to change an organization, it updates the state of the organization using the update interface of the context manager.

Interaction with the agents happens via the `Role Position` components. A role position provides a role-specific interface for an agent to act in the environment and interact with other agents. There is one special instance: the `Default Role Position` which offers the necessary primitives to browse open role positions and close role positions. The Role Position delegates agent requests, such as the sending of messages to a specific role type, to the role mediator.

**Deployment View.** Figure 5 shows a deployment view of the system. To make the system scalable and robust, we have chosen for a decentralized approach. Nodes work together to realize the required system goals. Nodes can be in two different modes with respect to a particular organization: master and slave. Each organization consists of one master and zero or more slaves. The master is responsible for the correct adaptation of the organization while the slaves provide the necessary information to the master and perform local adaptations based on instructions of the master.

On each master, an organization controller is deployed. Slaves do not provide such a controller. As stated above, the context manager on a master node keeps his context information up to date using the services of the agent middleware. Local context information is gathered through local perception of the environment, remote context is gathered from context managers on the slave nodes. The context manager on the master in turn keeps the context manager on the slave nodes updated about changes in organization structure.

To ensure consistency across organizations during organization adaptations, the organization controllers on different masters synchronize via the services provided by the agent middleware.

## 3.3 Merging of Organizations

Figure 6 shows a collaborating components view describing how organizations are merged. Only one of the two involved master nodes is shown in the figure. When the organization controller of a master node notices, via the view interface (`1. view context`), that its organization should be merged with another organization according to the evolution laws, the following scenario unfolds. The organization controller uses the agent middleware services to start a negotiation protocol with the master

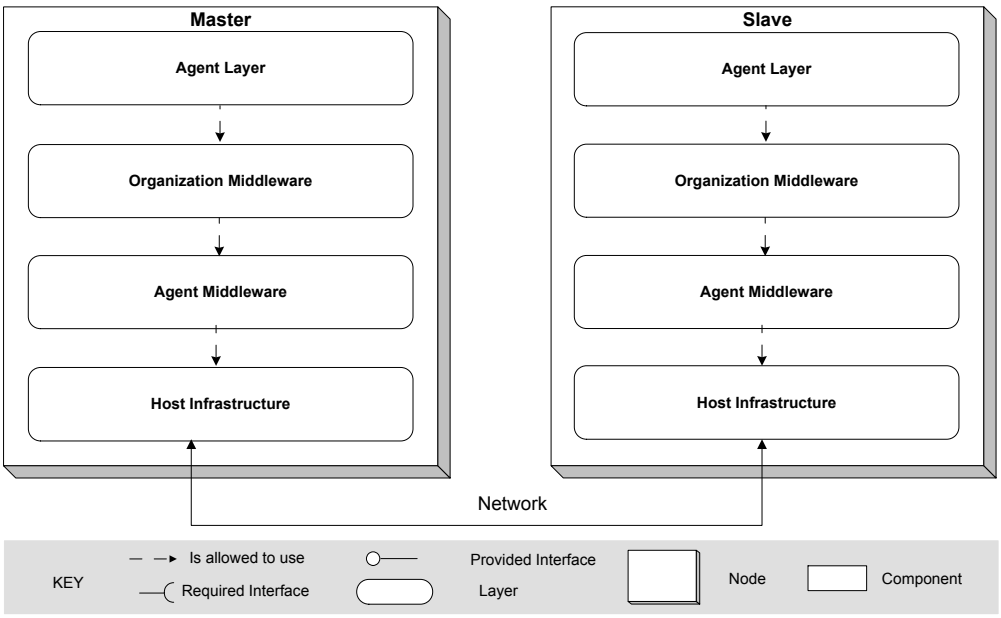


**Figure 5: Deployment View.**

node of the organization it wants to merge with (`2. start negotiation`). When the organization controllers agree to merge the organizations, they start a voting protocol to assign the master of the newly merged organization. The controller that is elected master, updates the context manager through the update interface. First the agents involved in the old organization are requested to stop playing their roles (`3. stop playing roles`). This request is forwarded to the slave nodes (`4. terminate organization`), to the local role positions and further to the agents (`5. stop role position` and `6. stop role`). After performing possible cleanup tasks, each agent informs the role position that it has stopped playing the role (`7. role stopped`). The role position notifies the context manager and the role position is terminated (`8. role position terminated`). Slaves inform the master node about the termination of role positions (`9. context data slave`). Notice how the ordering of steps 8 and 9 is not fixed. It could well be that remote slaves update the context manager on the master (`9`) before the local role positions do (`8`). Only when the organization controller observes that all role positions in the old organization are terminated (`10. view organization stopped`), it removes the old organization and adds the new merged organization. This includes opening new local role positions (`11. update organizations`). The context manager sends the necessary information about the new organization to the slaves involved (`12. data new organization`). The information includes configuration info and the set of open role positions of the new organization. The default role positions notice the new role positions (`13. new role positions`) and notify the agents about the new open role positions (`14. new role position`). The scenario shows how a decentralized approach is used in which only the relevant masters and slaves are involved, which is crucial for scalability.

## 3.4 Example of Merging Organizations

Figure 7 shows an example of merging organizations in the traffic monitoring case. The example retakes the situation described in figure 2 (with camera $C_6$ omitted). At $t_0$ the cameras on nodes $C_4$ and $C_5$ are merged in one organization. Being in master mode, the organization controller on node $C_5$ manages the merged organization. All other cameras are in separate organizations, each managed by their local organization controller in master mode. In the transition from $t_0$ to $t_1$, the master controllers on node $C_3$ and $C_5$ agree to merge. A voting between the two master controllers, determines the controller on node $C_3$ to be the new master controller of the merged organization at $t_1$. The organization controller on node $C_5$ is now in slave mode. At $t_2$, the cameras on nodes $C_4$ and $C_5$ are again split in separate organizations and their organization controllers are back in master mode. The organization controller on node $C_2$ is now the master of the merged organization.

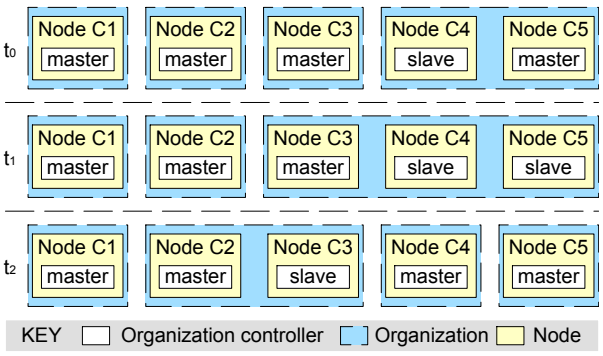


**Figure 7: Merging organizations in the traffic monitoring case.**

# 4. ADDING SELF-HEALING

Self-healing is one of the self-* properties that define autonomic computing [12]. Several definitions of self-healing can be found in literature [18, 20]. We consider self-healing as a software quality in addition to other requirements with an emphasis on autonomy in achieving the quality, reducing the human involvement to no more than high-level policies. Self-healing allows a system (1) to automatically detect and diagnose errors and failures caused by a prede-

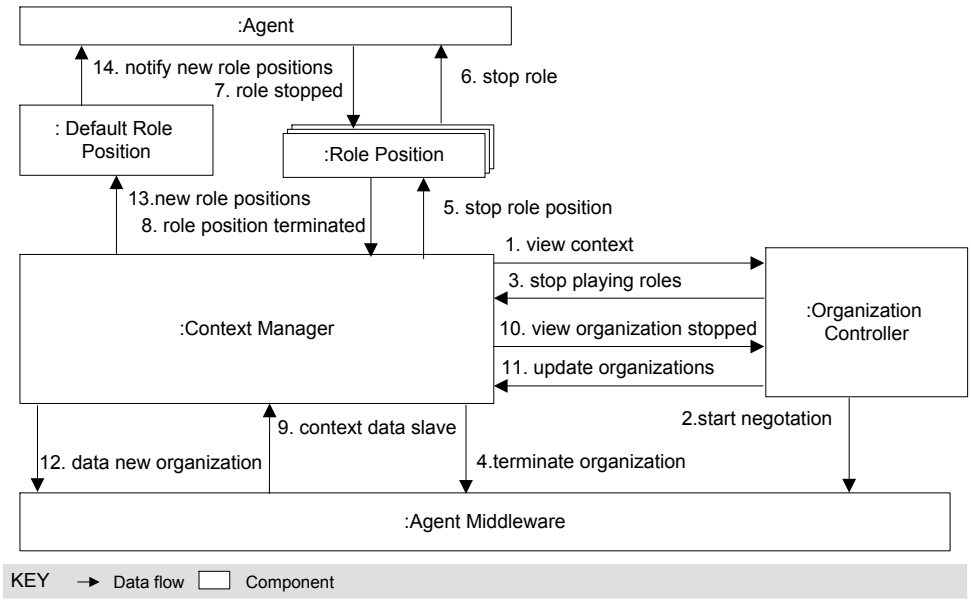


**Figure 6: Merge scenario.**

fined set of faults, and (2) to restore the software system to a healthy state according to a set of predefined healing qualities. Most existing techniques to achieve self-healing properties use some sort of control loop which monitors the system and adapts the system accordingly [9]. An example of a self-healing property in the traffic monitoring case could be the ability the deal with silent node failures. Whenever a node fails, the system is capable of detecting this failure an restoring the system to a degraded (one camera sensor is lost), but functional state.

## 4.1 Self-Healing and Adaptivity

Self-healing can be considered as a special type of adaptivity in addition to the adaptivity the system exhibits in normal operating conditions. Most software systems anticipate a number of changes or dynamics in the environment, which are part of the normal operating conditions of the system. For example, the multi-agent system, introduced in section 3, is capable of coping with changes and dynamics in traffic conditions. When particular events occur in the system or its environment, the system may enter a faulty state, outside the normal operating conditions, as shown in figure 8. From these faulty states, the system is no longer capable of operating according to its specification. For example, the multi-agent system is unable to deal with changes such as node failures. These failures would bring the system to an inconsistent state, in which agents assume role positions that are no longer available and organization controllers are lost.

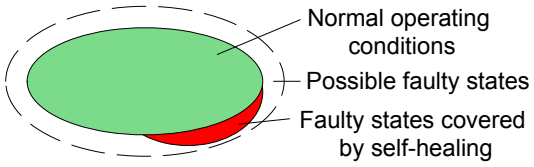


**Figure 8: System and environment states.**

A number of these changes, such as certain errors and failures or more specific a transition to a certain faulty state can be covered by self-healing. Self-healing allows the system to adapt itself in a particular state space outside the normal operating conditions, as shown in figure 8.

One could argue that these previously unanticipated changes can now be considered part of the normal operating conditions of the complete system with self-healing properties. Doing so, however, could make the system more complex. Certainly, when the same mechanisms are used, to deal with the normal operating conditions and the previously unanticipated changes. For example, the mechanism of organization controllers could be expanded to deal with node failures. This, however, would have a negative impact on the understandability, manageability and adaptability of the system.

The main concerns (functional and quality) refer to normal dynamics of the system, such as changing traffic conditions. Special self-healing concerns refer to more intrusive changes, such as node failures. Whenever the system enters a faulty state covered by self-healing, the self-healing subsystem detects this transition and ensures a transition of the system back to a state within the normal operating conditions. Once the system is restored to a state within the normal operating conditions, the main system can continue its operation. The complete system with self-healing covers both the normal operating conditions and the covered faulty states.

## 4.2 Architectural Approach

In this section, we explain how we have extended the main system with a self-healing subsystem that adds a number of self-healing properties to the functionalities and qualities provided by the main system. Figure 9 shows an abstract representation of the approach: the main system provides endogenous self-adaptation properties while the self-healing subsystem provides exogenous self-healing properties.

The self-healing subsystem uses a set of monitoring interfaces to monitor the main system and a set of control interfaces to adapt the main system, forming a closed control loop. The self-healing subsystem also uses a set of predefined self-healing scenarios or repair plans. These scenarios cover a predefined set of faulty states, how they can be detected and how the system can be adapted into a state in which the main system can function according to its specifications. An example of a simplified self-healing scenario in pseudocode is given below. The scenario deals with a node failure and

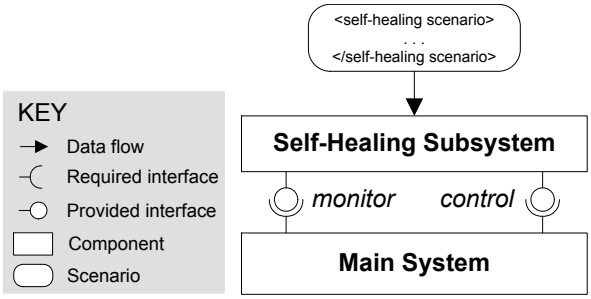


**Figure 9: Abstracted architecture of a self-healing system.**

consists of three phases: monitoring, analyzing and actual healing.

```
//monitoring phase
while(healthy){
    healthy = monitorAliveSignal(nodes);
}

//analyzing phase
failedNode = retrieveFailedNode(nodes);
lostC = retrieveControllerOn(failedNode);
lostRPs = retrieveRolepositionsOn(failedNode);

//healing phase
removeFromContext(lostC);
removeFromContext(lostRPs);
```

## 4.3 Self-Healing and Context-Driven Dynamic Organizations

In our distributed architecture for context-driven dynamic organizations we consider a limited set of faults. In this paper we focus on silent node failures. In a silent node failure, a node becomes unreachable without sending any corrupt messages to other nodes before or after its failure. When a node fails, the system enters an inconsistent state, in which agents, role positions and organization controllers are no longer capable of working according to their specification. Our goal is to add a self-healing property to the system allowing the system to restore to a consistent state in case of a node failure.

This self-healing property is achieved by a self-healing subsystem deployed on every node. The self-healing subsystem interacts with the local agent middleware and organization middleware, and relies on the functionalities of the main system in order to achieve its property. Figure 10 shows the integration of the self-healing subsystem with the rest of the system on one node. Self-healing subsystems periodically exchange alive signals using the communication service provided by the agent middleware. Node failures are detected by monitoring the alive signals. When a failure occurs, two types of inconsistencies may arise: (1) incorrect representations of topologies of nodes, and (2) corrupt representations of organizations. We explain both in more detail.

**Incorrect representations of topologies.** A first inconsistency that will arise in case of a node failure is an incorrect representation of the distribution topology in the context and the agent middleware. In many application domains, such as the traffic monitoring case, the topology is used to determine which organizations can merge. When a node fails, the topology representation in the context and the agent middleware should be updated to keep it consistent with the real world. In order to do so, each self-healing subsystem will monitor the alive signals of all nodes in their local topology, as shown in figure 11. The local topology is retrieved from the local context manager, using the view interface. When a self-healing

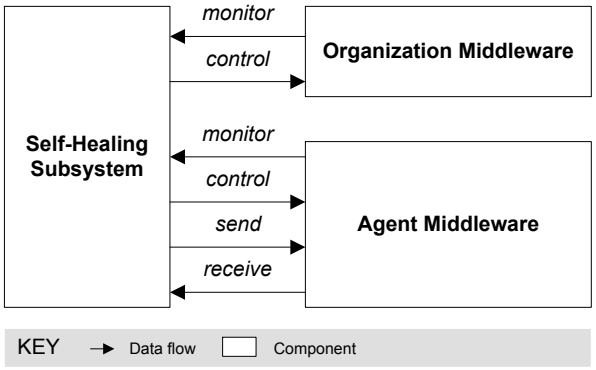


**Figure 10: Integration of the self-healing subsystem with the organization middleware and agent middleware on one node.**

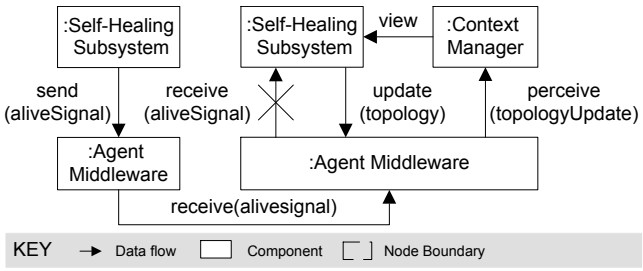


**Figure 11: Self-healing corrupt topology scenario.**

subsystem detects the failure of a node in its local topology, it updates the topology representation in the main system.

**Corrupt representations of organizations.** A second inconsistency that will arise in case of a node failure is a corrupt organization with lost or missing role positions on the failed node. We consider two scenarios. In case a slave node (a node with an organization controller in slave mode) fails, the organization can continue working after some context updates by the self-healing subsystem. In case the master node (a node with an organization controller in master mode) fails, the organization is reset. When an organization is reset, every agent in the organization is split up in a separate organization, and the mechanisms of the main system will ensure that the organizations are re-merged according to the current context. Each self-healing subsystem will use the view interface on the local context manager to determine whether its organization controller is in slave mode or in master mode.

Figure 12 shows the scenario of a slave node failure. The self-healing subsystem on the master mode will monitor the alive signals of all other self-healing subsystems on slave nodes, active in the same organization. When the self-healing subsystem detects the failure of a slave node, it updates the context, using the update interface of the local context manager. The update consists of removing all role positions from the organization context that were active on the failed node. Because the local organization controller itself is in master mode, the main system ensures that this update will be propagated to all other members of the organization.

Figure 13 shows the scenario of a master node failure. Each self-healing subsystem on a slave node will monitor the alive signal of the self-healing subsystem on the master node of the organization. When a self-healing subsystem detects the failure of the master node, it updates the context signaling the local agents to stop their current role positions and that they are split in a separate organization. The self-healing subsystem then starts up the local

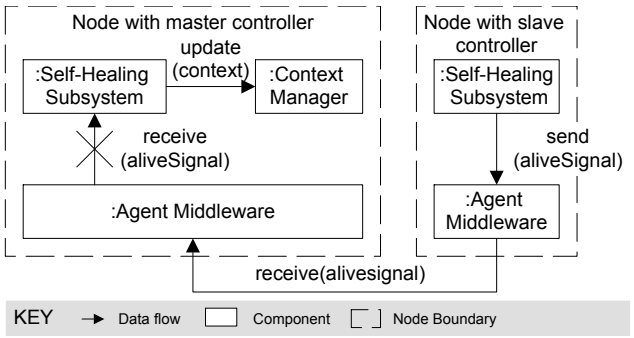


**Figure 12: Self-healing slave node failure scenario.**

organization controller in master mode. Using a consistent context, the main system will now ensure that these organization are re-merged according to the current context.

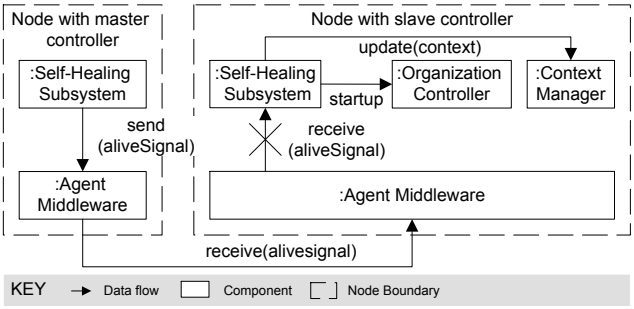


**Figure 13: Self-healing master node failure scenario.**

## 4.4 Self-Healing Example

Figure 14 shows an example of a self-healing scenario in a simplified setting. At $t_1$ there are three nodes, the figure only highlights the most relevant items on each node. Each node has a default role position and one other active role position. All role positions belong to the same organization. At $t_1$, however, $node_1$ has just failed. The context known in $node_2$ and $node_3$ is now inconsistent with the current state of the system. Role positions on $node_2$ and $node_3$ still assume role position $RP_1$ on $node_1$ and the master controller on $node_1$ is lost.

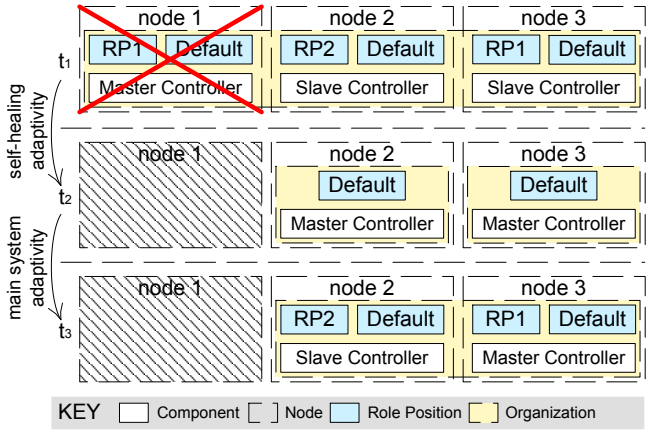


**Figure 14: An example of a self-healing scenario.**

As explained in section 4.3, the self-healing subsystem on the two remaining nodes will detect this failure, because they no longer receive the necessary alive signals. After the detection, the self-healing subsystems update the topology in the agent middleware and reset the organization. They flag the role positions $RP_2$ and $RP_1$ on $node_2$ and $node_3$ to be stopped and remove the organization from the context. Finally, the self-healing subsystems start up their local organization controller in master mode.

At $t_2$, the self-healing subsystems have finished the healing scenario. The default role positions on each node are split up in separate organizations, each managed by the local master controller. The system is again in a consistent state from which the main system (context-driven dynamic organizations) can continue its operation. The main system will ensure the organizations on the remaining nodes to be re-merged, according to the current context. A possible re-merge is shown at $t_3$.

# 5. RELATED WORK

The literature on self-management is extensive. In this paper, we limit the discussion of related work to a number of key papers on architectural approaches for self-managing systems.

Early work of Shaw presents an approach for self-management based on process control loops [17]. This work provides a foundation for what we call exogenous self-management.

Kramer and Magee [13] advocate a component-based architectural approach for self-management, and propose a high-level layered software architecture comprising of three layers: *component control* as the bottom layer, *change management* as the middle layer and *goal management* as the top layer. Compared to this approach, our research currently focusses on the change management layer. However, whereas our work combines an endogenous approach with an exogenous approach for self-management, Kramer and Magee describe a reference model for exogenous self-management.

Oreizy and colleagues [16] propose an architecture-based approach to self-adaptive software. The approach is based on the idea of two simultaneous processes: *system evolution*, i.e. the consistent change of the system, and *system adaptation*, responsible for planning changes and responsive measures. This approach also highlights the possibility of an explicit system representation in the form of an architectural model which can be used by the adaptation mechanisms. As such, this approach also puts forward an exogenous approach to self-management.

The IBM architectural blueprint [2] proposes a hierarchical architectural approach to self-management. Autonomic managers add self-* properties to resources and these managers are, in turn, managed by other autonomic managers. At the highest level a manual manager takes high-level policies from users and delegates these throughout the hierarchy of autonomic managers. The IBM blueprint architecture is basically an exogenous approach to self-management.

Garlan and colleagues [8, 7] focus on the idea of an abstract system representation by introducing an explicit layer for model management. This layer can then be used by high-level adaptation mechanisms. As shown in [3] the idea of an abstract system representation idea can be expanded to an integrated approach, in which an architectural description language (ADL) offers explicit support for adaptation. In our approach, the context manager maintains a model—though a basic model—of the organizations in the system. We have illustrated that this model can offer support adaptation mechanisms for particular self-healing scenarios.

# 6. CONCLUSIONS

In this paper, we presented a hybrid approach for self-management that combines an endogenous self-adaptation approach with an exogenous self-healing approach. In particular, the

multi-agent system provides self-adaptation to deal with dynamics in normal operating conditions, while the self-healing subsystem covers the dynamics involving transitions to faulty states.

We applied the approach to a distributed architecture for context-driven dynamic organizations. In this architecture, the agents of the multi-agent system, supported by a middleware for dynamic organizations, flexibly adapt their behavior to dynamic and changing circumstances. The self-healing subsystem ensures the transition of the system from a faulty state to a state within the normal operating conditions of the system. Mainly by updating context, the self-healing subsystem ensures a consistent state of the system, from which the main system itself can continue its operation, possibly in a degraded but correct mode. The software of the multi-agent system extended with the self-healing subsystem is available for download at http://distrinet.cs.kuleuven.be/agentwise/. The software includes various test scenarios.

Since, our experiences with the approach is currently limited to the traffic monitoring application, we can make no claims about the scope of applicability of the approach. Important properties of the traffic application that determine the class of systems for which the approach may be useful are: (1) there is an inherent distribution of resources and activity in the system, and (2) the speed of cars and thus the pace at which organizations have to evolve is orders of magnitude lower than the speed of communication and the execution of the control software. In addition, the approach assumes continual communication infrastructure.

Although the self-healing subsystem is structurally disjunct to the main system, the subsystems requires access to the main system and detailed knowledge about its behavior. The case shows that adding self-healing to a system is invasive. We explain how we plan to deal with the coupling between the self-healing subsystem and the main system in a disciplined manner. Applying the approach, showed that the self-healing subsystem uses a number of existing interfaces in the main system, such as send and receive interfaces provided by the agent middleware, and the update and view interfaces of the context manager. The self-healing subsystem also requires specific monitoring and control interfaces such as an interface to start up organization controllers in master mode. The integration of the self-healing subsystem with the main system is currently achieved in an ad-hoc manner, the main system just exposes the necessary interfaces. Currently, we study a component model that allows a disciplined integration of the self-healing subsystem with the main system. Our particular interest is on using aspect-oriented software development to enable such an integration. The essential challenges here are: (1) to define a suitable joint point model, and (2) to provide guarantees that no improper interferences will happen.

Finally, our approach currently uses a predefined set of self-healing scenarios. Several researchers have proposed more complete self-healing approaches, covering repair plan generation and goal management. Our approach currently focuses on how to clearly separate the self-healing concern from the rest of the system. We hope, by focusing on a specific class of self-healing requirements in a specific application domain, to uncover a lot of the nuances and problems related with self-healing. The long term objective of our research is to develop a disciplined engineering approach to deal with self-healing in decentralized architectures.

## 7. ACKNOWLEDGMENTS

This research is supported by the DiCoMAS project that is funded by Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). Danny Weyns is funded by the Research Foundation Flanders (FWO).

## 8. REFERENCES

[1] Software Engineering for Self-Adaptive Systems, Dagstuhl Seminar 08031. http://www.dagstuhl.de/en/programm /kalender/semhp/?semnr=08031, 2008.

[2] IBM. Computing. An Architectural Blueprint for Autonomic Computing, http://www-03.ibm.com/autonomic/ (3/2008).

[3] E. Dashofy, A. van der Hoek, and R. Taylor. Towards architecture-based self-healing systems. *Proceedings of the first workshop on Self-healing systems*, pages 21–26, 2002.

[4] V. Dignum, F. Dignum, and L. Sonenberg. Towards Dynamic Reorganization of Agent Societies. *Proceedings of Workshop on Coordination in Emergent Agent Societies at ECAI*, 2004.

[5] ERTICO: Intelligent Transportation Systems for Europe. http://www.ertico.com/.

[6] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in MAS. *3rd International Conference on Multi Agent Systems, ICMAS*, 1998.

[7] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[8] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. *Proceedings of the first workshop on Self-healing systems*, pages 27–32, 2002.

[9] D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya. Self-healing systemsŮsurvey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007.

[10] R. Haesevoets, B. V. Eylen, D. Weyns, A. Helleboogh, T. Holvoet, and W. Joosen. Managing Agent Interactions with Context-Driven Dynamic Organizations. In *Engineering Environment-Mediated Multiagent Systems*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

[11] ITS America: Intelligent Transportation Society of America. http://www.itsa.org/.

[12] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[13] J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. *International Conference on Software Engineering*, pages 259–268, 2007.

[14] J. Odell, H. V. D. Parunak, and M. Fleischer. The Role of Roles. *Journal of Object Technology*, 2(1):39–51, 2003.

[15] A. Omicini and A. Ricci. Reasoning about organisation: Shaping the infrastructure. *AI* IA Notizie*, 16(2):7–16, 2003.

[16] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[17] M. Shaw. Beyond objects: a software design paradigm based on process control. *SIGSOFT Softw. Eng. Notes*, 20(1):27–38, 1995.

[18] R. Sterritt and D. Bustard. Autonomic Computing-a means of achieving dependability? *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the*, pages 247–251, 2003.

[19] D. Weyns, A. Helleboogh, T.Holvoet, and M.Schumacher. The Agent Environment in Multi-Agent System: A Middleware Perspective. *International Journal on Multiagent and Grid Systems, Special Issue on Engineering Environments for Multiagent Systems*, 2008.

[20] S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart. An architectural approach to autonomic computing. *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 2–9, 2004.