Martin Weißbach, Thomas Springer

Coordinated Execution of Adaptation Operations in Distributed Role-based Software Systems

Erstveröffentlichung in / First published in:

SAC 2017: Symposium on Applied Computing. Marrakech, 03.04.–07.04.2017. ACM Digital Library, S. 45–50. ISBN 978-1-4503-4486-9.

DOI: https://doi.org/10.1145/3019612.3019624

Final edited form was published in "SAC 2017: Symposium on Applied Computing. Marrakech 2017", S. 45–50. ISBN: 978-1-4503-4486-9
https://doi.org/10.1145/3019612.3019624

Coordinated Execution of Adaptation Operations in Distributed Role-based Software Systems

Martin Weißbach
Technische Universität Dresden
Nöthnitzer Straße 46, 01187 Dresden, Germany
martin.weissbach1@tu-dresden.de

Thomas Springer
Technische Universität Dresden
Nöthnitzer Straße 46, 01187 Dresden, Germany
thomas.springer@tu-dresden.de

ABSTRACT

Future applications will run in a highly heterogeneous and dynamic execution environment that forces them to adapt their behavior and offered functionality depending on the user's or the system's current situation. Since application components in such heterogeneous multi-device systems will be distributed over multiple interconnected devices and cooperate to achieve a common goal, a coordinated adaptation is required to ensure a consistent system behavior. In this paper we present a decentralized adaptation middleware to adapt a distributed software system. Our approach supports the reliable execution of multiple adaptation operations that depend on each other and are performed transactionally even in unsteady environments coined by message loss or node failures. We implemented our approach in a search-and-rescue robot scenario to show its feasibility and conduct first performance evaluations.

CCS Concepts

•**Social and professional topics** → *Software selection and adaptation;* •**Software and its engineering** → *Software post-development issues;*

Keywords

self-adaptive systems, decentralized coordinated adaptation, multi-device systems, roles

1. INTRODUCTION

Software systems are more often distributed over multiple physical or virtual devices collaborating to achieve a common goal or to provide services to users or other software systems. Moreover, future software systems will have to adjust themselves to either the user's current situation, their computational environment or even both. In order to cope with the high degree of distribution of such adaptive software systems, the subsystems tasked with adaptation management will have to be decentralized as well [15]. A solid body of research work is existing (cf. the survey conducted in [8]) that focuses mainly on design approaches of such software systems, reasoning and decision making including formal foundations supporting these parts of an adaptive software system. Decentralization aspects have already been investigated on the level of the decision making [10,12] in such systems to determine which parts of the system have to be adapted in response to changes in the computational environment. The execution of such agreed changes has not been subject to detailed investigation. In [7, 14] a formal model has been proposed to determine a state of run-time components at which it is safe to exchange, update or remove them. In [4] changes are limited so single devices without the need for further coordination with other adaptation management instances. None of the approaches considered the execution of adaptations as a sequence of adaptation steps that are related to or depend on each other.

We use the concept of roles to describe static and dynamic parts of an adaptable entity [2]. The entity's static part is modeled as *Player* while dynamic parts are modeled as *Roles*. A role, therefore, encapsulates context-dependent behavior of an entity, i.e., an adaptable entity's behavior can be accommodated to changes in its computational environment by having the player acquire or drop roles dynamically at run time. Roles may also be migrated from one player to another in response to context changes. On the application level, *Compartments* set a collaboration context for roles. Players may play roles in different compartments simultaneously, but only expose the behavior of the currently active compartment. So far, the usage of the role concept has focused on the abstraction of collaborations between system entities, e.g., agents [1, 11] or services [5], but has not considered the contextual nature of roles.

In the remainder of this paper, we present an approach to perform multiple related adaptations of a distributed software system without the need for a central coordinator. First, we will discuss related work in the field in detail (section 2). Second, we will present our approach (section 3) that incorporates (1) a high-level abstraction of the adaptive software system in which the adaptive subsystem as well as the interaction with the managed subsystem is described; (2) a set of platform-independent adaptation operators used to describe the planned changes our solution is able to per-

DOI: http://dx.doi.org/10.1145/3019612.3019624

Final edited form was published in "SAC 2017: Symposium on Applied Computing. Marrakech 2017", S. 45–50. ISBN: 978-1-4503-4486-9
https://doi.org/10.1145/3019612.3019624

form; and (3) a decentralized coordination protocol that is used by the managing subsystem to conduct the execution of the planned changes. Third, we will evaluate our approach to show its feasibility to coordinate adaptations in distributed environments. Finally, we will discuss the results of the evaluation and conclude our presented approach.

2. STATE OF THE ART

A solid research body exists that focuses on the design and architecture of self-adaptive software systems (see [8]) including the MAPE-K feedback loop [6] as an architectural means to enable self-adaptability of software systems through self- and context-awareness. With an increasing degree of distribution of adapted applications, centralized adaptation management approaches become impractical, which leads to several efforts to distribute the adaptation management subsystem. Patterns of how to distribute phases of the feedback loop [15] or the decentralization of the decision making process have been the main research focus.

FlashMob [12] and DecAp [10] are examples of approaches that share information about the system's current configuration to decide in a decentralized manner when and how to (re-)assemble the system. A different approach by Georgiadis et al. [4] discusses the distributed execution of such changes through distributed component managers. The component managers synchronize using reliable broadcasts to publish join/ leave messages of components entering or leaving the system. Locks on the system, that ensures only one component manager at a time to be able to perform local changes on a globally consistent configuration, are also acquired through broadcasting. However, a component manager can only change its local component and a discussion how several component managers would collaborate to perform multiple changes in a consistent way is missing. Our approach takes the results of decentralized decision making approaches as input to execute multiple, possibly dependent changes in a coordinated and consistent manner.

Since adaptations are performed at run time, a state has to be found in which it is safe to adapt the system or an entity. In [7] this state is called the *quiescent* state of a component and means the component not to be involved in any communication and is, thus, safe for removal or update, for example. Since the concept of quiescence imposes the restriction of several system components for one component to reach such a state, the conditions to reach this state have been relaxed to achieve less system interruption sacrificing the property to ever reach such a state; the resulting concept is referred to as *tranquility* [14]. Different adaptation semantics have been proposed that describe the system behavior formally when adaptations are about to be performed. In [3], three semantics are distinguished: (1) one-point adaptation (system behavior changes from one point in time to the next), (2) guided adaptation (the program is restricted to reach a state at which it can be adapted), and (3) overlap adaptation (a program temporarily exposes both old and new behavior with the old system being restricted in its functionally until the new behavior eventually takes over completely). Our approach makes use of these foundations to determine a point in time in the applications execution that allows the safe modification of role bindings.

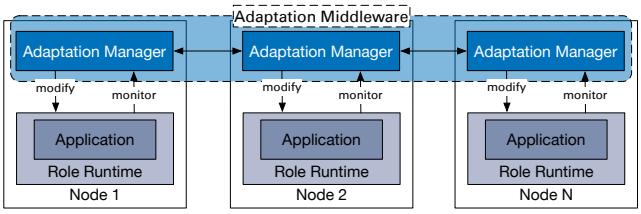

Figure 1: A decentralized adaptation middleware for role-based software systems.

In *multi-agent-systems*, roles serve as abstraction to describe collaborations of agents within a group of agents [1,11]. Similarly, roles abstract from the concrete capabilities an agent has to provide in order to play a role. For example, in a peer-to-peer network, super peers can be established to structure the network, but a super peer is essentially only a role played by a peer with enough computational resources. Apart from that, the collaboration between super peer and its set of child peers clearly describes communication flows in the system. Due to the collaboration-centric notion of roles, an agent's possible behaviors remain static at run time and are either executed actively or not depending on the collaborations the agent participates in, but it is not possible to dynamically add behavior or roles to an agent that was not foreseen to be played by the agent at design time.

In summary, to the best of our knowledge, no current research effort exists that investigates how to perform multiple changes of a highly distributed software system in a coordinated and consistent manner across several devices. Please note that we are not concerned with the decision making process itself as it was discussed in [10,12], but with the execution of such calculated change prescriptions at run time.

3. APPROACH

Our approach relies on a clear separation of application and adaptation logic as it was presented in [6]. First, we briefly discuss the architecture of the overall system including our proposed *adaptation middleware*. Subsequently, we describe the *adaptation operators* supported by the adaptation middleware that describe how the application can be changed. Lastly, we discuss the decentralized coordination protocol used to execute the adaptation of the application.

3.1 System Overview

Relying on a clear separation of application and adaptation logic, the two main components of our adaptive system's architecture are: (1) a role-based application and its respective runtime and (2) the adaptation middleware, which both are distributed over multiple devices (see Figure 1). Due to our focus on the coordinated execution of adaptations, we omitted other necessary components of (self-)adaptive software systems, such as monitoring of the computational environment or reasoning and planning components to derive adaptation decisions in Figure 1.

The *Adaptation Middleware* is composed of autonomous *Adaptation Managers* that control the adaptation of the parts of the distributed role-based application located on the same

Final edited form was published in "SAC 2017: Symposium on Applied Computing. Marrakech 2017", S. 45–50. ISBN: 978-1-4503-4486-9
https://doi.org/10.1145/3019612.3019624

device. Since the adaptation managers collaborate to perform the adaptation, we consider the adaptation middleware not only distributed but also decentralized [15]. The local role-based application can be changed through the *modify* interface of role-runtime and adaptation manager. Modifications such as creating or destroying role instances as well as binding them to or unbinding them from players are realized by that interface which adheres mainly to role features three to five of [9], i.e., players may play different roles simultaneously and multiple times while players may acquire and abandon roles at run time dynamically. The current configuration of the role-based application in terms of roles being played by which players can be obtained through this interface, which is represented by the *monitor* arrow in Figure 1. The details of the interface between the adaptation managers will be presented in section 3.3.

3.2 Adaptation Operators

In response to changes in the computational environment of the distributed application, modifications may have to be applied to the application to meet the new situation. Such modifications are specific steps, represented by *adaptation operators* that describe the transition of an application from a source configuration to a valid target configuration [3]. The adaptation middleware is responsible for the execution of this transition and supports operators to (1) create and bind new instances of roles of a specific type to players, (2) remove role instances from players as well as a combination of both, namely, (3) exchanging a bound role instance with a newly created instance of a different type of role, e.g., to exchange data encoding context-dependently. Furthermore, we support the (4) migration of a role from one player to another, which might be located on a different node, and the (5) replication of a role to other players.

Adaptation operators consist of a set of parameters (cf. Figure 2) to unambiguously describe the modifications to be performed by the adaptation middleware: The unique *Identifier* identifies a single operation at run time; the *Order* number allows to partially order multiple adaptation operators, e.g., to setup predecessor/successor relationships between operators; the *Type* identifies the concrete operator and currently supports to establish and release collaborations between roles in addition to operations (1) through (5) mentioned above; the *State* is present if internal state information of the role is supposed to be transferred from the source to the target configuration; the *Target Node* and *Source Node* specify the system configuration afterwards and before operator's execution, respectively. Both source and target node are composed of four parameters: The player and compartment parameters identify the core object on which the respective role operation is supposed to be executed on and in which compartment, respectively. The role parameter either specifies only the type of the role or an instance identifier as well. If a new role is to be created, only the type information is provided, the latter case is used for operations, e.g., a role's migration or removal. We use the term *operation* or *adaptation operation* if we speak about the fully parameterized adaptation operator at run time.

The adaptation operators provided by the adaptation middleware are abstract operations that provide a specific semantic to the component that is responsible for planning the changes. Each adaptation operation is decomposed into a sequence of lower level adaptation operators provided by the role runtime which we mentioned in the last section.

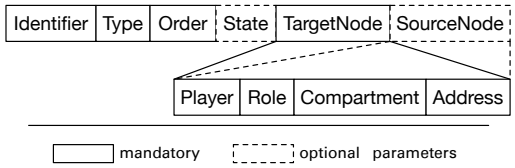

Figure 2: Structure of Adaptation Operators.

3.3 Coordinating Adaptation Operations

Using the previously described adaptation operators, the configuration of the role-based application can be changed at run time in response to changes in the computational environment. We introduce the term *Transaction* for a set of related adaptation operations that defines a scope to reliably transfer the adapted application from a source to a target configuration. We assume each adaptation manager to have complete knowledge about the current transaction, which means that the transaction has to be distributed before the adaptation process can commence.

Subsequently, we present the coordination protocol used by the adaptation managers to perform the adaptation of the distributed application and subsequently discuss error scenarios in distributed environments and the protocol features addressing these scenarios.

3.3.1 The Coordination Protocol

The coordination protocol specifies messages and a scheme on which adaptation managers exchange messages in order to adapt a distributed role-based application. The protocol specifies two types of coordination messages: *Report* and *StateTransfer* messages. A *Report* message contains a reference to the identifier of the transaction it is part of as well as at least one tuple of $\{operationID, status\}$. The *operationID* refers to the adaptation operation on which progress is reported and the boolean *status* indicates whether the operation was conducted successfully or not. The *StateTransfer* message contains the same information as the report message, but the *status* information on the operations progress is replaced with a serialized representation of the role's internal *state*. Consequently, StateTransfer messages are only transmitted from the source to the target configuration, e.g., as part of the migration of a role from one (source) node to another (target) node.

In the following, we use a small example of a role's migration from one player to another on a different node to outline the exchange of *Report* and *StateTransfer* messages between adaptation managers to coordinate the adaptation. The migrating role is assumed to collaborate with other roles, i.e., communication channels exist between the roles.

A planning component within the overall system is responsible to calculate the resulting transaction, which consists of *Disconnect* and *Connect* operations to release and reestab-

Final edited form was published in "SAC 2017: Symposium on Applied Computing. Marrakech 2017", S. 45–50. ISBN: 978-1-4503-4486-9
https://doi.org/10.1145/3019612.3019624

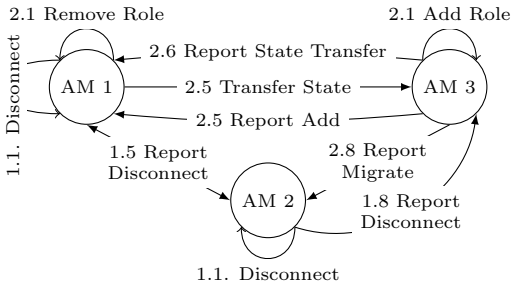

Figure 3: Exchange of Coordination messages for a migrate operation.

lish collaborations of the migrating role, and a *Migrate* operation. The previously discussed adaptation operators are parameterized by the planning component to instruct the adaptation middleware with the modifications that are supposed to executed, e.g., specify operation IDs for each operation that is unique within the scope of the transaction, or use the Order Number parameter to ensure the operations to be performed in the order $Disconnect < Migrate < Connect$.

After receiving a transaction, each involved adaptation manager calculates *Order Groups* based on the adaptation operations' order number. This step is required to structure to execution of dependent operations and contributes to reduce the amount of transmitted coordination messages. Adaptation managers that are addressed within the source or target parameter of operations that belong to the first order group commence the execution of the transaction while other adaptation managers wait. In our example, AM 1 and AM 2 perform the disconnection of the role collaboration (step 1.1) and report the local results to each other (step 1.5). Please note that these *Report* messages are only exchanged between the AMs of the source and target node of the operation. Since AM 2 was specified as target node of the operation, it propagates the overall result to all other AMs that are part of the transaction, which is shown representatively in step 1.8 in Figure 3.

Upon receiving all Report messages, an adaptation manager continues immediately with the execution of the next Order group. In our case, AM 3 creates a new local instance of the role under migration, while AM 1 performs the removal of the role, which consists of a passivation of the role and the retrieval of its internal state after it was passivated. Both actions are displayed as step 2.1 in Figure 3 to indicate the parallel execution of these sub-tasks of the migration. Step 2.5 in Figure 3 shows the Report and StateTransfer messages exchanged between the two. After the received state information was injected into the newly created role instance, AM 3 sends another Report message to AM 1 (step 2.6) and notifies all peers about the successful migration of the role (step 2.8). Subsequently, roles would be reconnected, which we will not present since the general idea follows the previously discussed disconnection. Local changes are activated after all Order Groups were executed successfully and the application resumes to work unrestricted. The passivation of roles can be considered a restriction of the application's functionality since this passive state can be mapped to a quiescent state of a component [7]. Please note that parts of the application not affected by the adaptation continue to carry out their tasks. Roles of higher Order Groups remain active, too, until their Order Group is executed.

3.3.2 *Handling Error Scenarios*

We foresee the loss of coordination messages exchanged between adaptation managers, software failures of adaptation managers or parts of the adapted application, and adaptation failures due to insufficient resources available on a node as infrequently occurring errors during the execution of adaptation processes. Leaving these errors unattended can result in an inconsistent application configuration, which may render the application inoperable. In the remainder of this section, we discuss how these error scenarios are addressed by the coordination protocol. Previously received Report messages of the currently executed Order Group are accumulated in outgoing Report messages. If an adaptation manager is still missing Report messages, a *RequestReport* message is transmitted to the adaptation managers of which reports are missing to request the transmission of the report. Ultimately, a *RequestReport* message is broadcast to peers of the Order Group to obtain the missing report information. If no other peer could provide status information about the progress of the operation on the node, the adaptation is continued assuming optimistically that the node will be back online soon. However, the decision whether to continue or abort the transaction also depends on the concrete adaptation operation. Assume the source node of a migrate operation is unavailable during the adaptation process. Since no state information can be obtained in such a case, the collaborating adaptation manager of the target node would have the transaction abort immediately after the request for the report retransmission fails.

In the case of lost *StateTransfer* messages, the procedure is similar, but if the state information could not be obtained after a requested retransmission, the target node's adaptation manager broadcasts a negative Report message to all adaptation managers participating in the transaction, which aborts the transaction immediately.

In case of a failed transaction, all changes that have already been performed have to be reverted. The steps necessary are calculated by the adaptation managers autonomously upon the receipt of a negative report message. A rollback of a transaction requires only minimal further coordination since passive roles only have to be reactivated and newly created but still passive role instances are removed. Reestablishing collaborations between roles follows the scheme of (dis-)connecting roles, which we already discussed.

For the recovery from failures of single adaptation managers or the adaptable application, the execution of operations and the progress of the transaction is logged locally by each adaptation manager. Having knowledge about the last executed transaction and the state of the execution allows the adaptation manager to reintegrate and finish the execution of the transaction. However, in a first step, reports from peer adaptation managers are requested to determine if the transaction is supposed to be continued or if it has been aborted in the absence of the local adaptation manager.

Final edited form was published in "SAC 2017: Symposium on Applied Computing. Marrakech 2017", S. 45–50. ISBN: 978-1-4503-4486-9
https://doi.org/10.1145/3019612.3019624

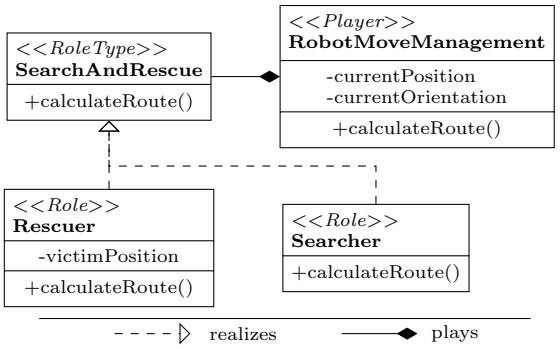

Figure 4: Role Diagram for the Search-And-Rescue Robot scenario.

Since local adaptation errors due to resource constraints result in a negative *Report* message sent to all adaptation managers involved in the transaction, the transaction is immediately aborted and already performed changes are reverted.

4. EVALUATION

We implemented the coordination protocol in the adaptation middleware using Java 1.8 and LyRT [13] to implement the managed role-based application. LyRT is a role-based framework that uses dynamic instance binding to modify a player's run-time behavior through role bindings, which meets our role requirements. An extension to LyRT was implemented that allows our adaptation managers to modify the role bindings of the managed application. A robot's movement is controlled by the *RobotMoveManagement* player that can play roles of the type *SearchAndRescue* to determine and modify the exact movement behavior at run time. If a victim's location is unknown, the player plays the *Searcher* role (random movement) whereas the robot moves straight to the victim and on the shortest way out of the disaster site after the victim has been recovered (*Rescuer* role) if the location is known (see Figure 4). Assume, one robot finds a victim and exchanges the Searcher with the Rescuer role, but detects itself to be unable to finish the rescue due to low battery power. An adaptation is triggered that migrates the Rescuer role to another robot nearby. Since both roles provide behavior that is specific to the robots movement, the Searcher role has to be removed from the second robot before the migration can be performed. Please note that finding a nearby robot or calculating adaptation plans are responsibilities of the self-adaptive infrastructure built on top of our adaptation middleware, e.g., [10, 12].

We conducted the experiment in a partially virtualized environment on one of our office's computers. That means the second robot was executed directly on the host system whereas the source robot was executed within a virtual machine. A third system simulated the decision making process by issuing the transaction to the nodes involved in the experiment. It was hosted on a different virtual machine on the same physical hardware. Delays to the adaptation process introduced by the network communication of the adaptation managers are purposefully neglected using such a virtualized environment for the experiment. The underlying network's

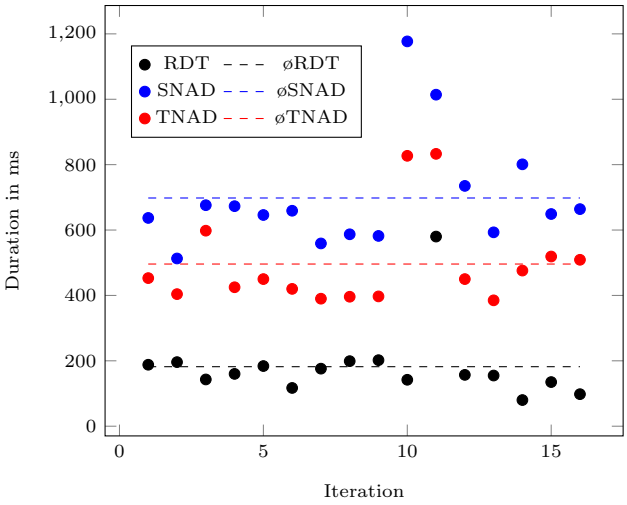

Figure 5: Results of the Adaptation Measurements; RDT... Role Downtime; S/TNAD... Source/Target Node Adaptation Duration.

performance cannot be influenced by the adaptation protocol and a mitigated network communication allows for a better assessment of the protocol's performance with respect to the execution of the adaptation.

In the experiment we conducted, the total time the transaction was running on both source (*SNAD*) and target (*TNAD*) nodes was measured. We also measured the downtime of the role (*RDT*) that was migrated. The role downtime is a measure for the interruption of the specific behavior, and hence the managed application. This downtime of the roles begins when the role is passivated on the source side, thus, ceases to perform its designated behavior and ends after the activation of the role on the target side. Figure 5 displays the results of the experiment. The downtime of the migrated Rescuer role averages at 182ms over all performed 16 iterations of the experiment. Iterations 10 and 11 show an increased adaptation duration time and role downtime for which we deem Java's garbage collection mechanisms responsible.

5. DISCUSSION

Our approach can be classified as a *Guided Adaptation* [3] using the notion of *quiescence* [7] to reach a safe state that allows for the safe execution of adaptation. The concept of quiescence is applied to roles only, which means players continue to operate as long as the role-specific behavior is not performed. Consequently, the time of system interruption can be reduced, which is beneficial for the measured results of our approach.

Due to the decentralization of our adaptation middleware, our approach could easily be integrated into approaches relying on a decentralized planning of changes, e.g., [10, 12]. Our approach is not limited to role-based software systems if an abstraction layer, which maps to the local role-runtime, can be built on top of the adaptable application. The role-abstraction allowed us to define adaptation operators independently from the managed application's platform.

Final edited form was published in "SAC 2017: Symposium on Applied Computing. Marrakech 2017", S. 45–50. ISBN: 978-1-4503-4486-9
https://doi.org/10.1145/3019612.3019624

We consider the loss of coordination messages the most common error source during ongoing adaptations. The aggregation of report messages and the optimistic continuation of the adaptation process are the coordination protocol's means to cope with message loss. Furthermore, the reintegration of failed nodes during an adaptation process is addressed by the coordination protocol. However, we want to stress that mechanisms known from *self-healing* approaches are expected to detect node failures or application crashes and to restart them in order for our adaptation middleware to be able to recover the transaction from the failure.

6. CONCLUSION

In this paper, we presented an approach to coordinate the adaptation of a distributed application in a consistent and decentralized manner. The execution phase of a self-adaptive system has not been subject to major research efforts except for [4] that ensures consistent local adaptations of a distributed system. However, performing several adaptations that depend on each other in a consistent manner has not been investigated yet. The decentralized coordination protocol presented in this paper fills this gap. We use the abstraction between static and dynamically exchangeable application parts provided through the concept of roles [2] as a foundation for the adaption operators that serve as input for the coordination protocol and describe the changes supposed to be made to the managed application.

We conducted experiments in the domain of autonomous search-and-rescue robots to show the feasibility of our approach. The first measures we obtained indicate a general feasibility of our idea for adaptations that encompass a few changes with respect to local execution times of the actual adaptations. Our evaluation environment was coined by low network latency which allowed us to get insight about the general performance of local execution of the coordination protocol. Exhaustive experiments with respect to the discussed error scenarios still have to be conducted.

The concept of roles allows the coexistence of old and new application behavior due to the invocation resolutions at run time. Taking advantage of this behavior through a refinement of the adaptation protocol beyond the plain passivation and activation of roles would allow us to reduce the time the system is interrupted throughout the adaptation. Future work in this direction is likely to result in improvements of the proposed coordination protocol.

Acknowledgements

This work is funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907).

7. REFERENCES

[1] M. Becht, T. Gurzki, J. Klarmann, and M. Muscholl. ROPE - Role Oriented Programming Environment for Multiagent Systems. *CoopIS*, 1999.

[2] G. Boella and F. Steimann. Roles and Relationships in Object-Oriented Programming, Multiagent Systems and Ontologies. In *Object-Oriented Technology. ECOOP 2007 Workshop Reader*, pages 108–122. Springer, Berlin, Heidelberg, July 2007.

[3] B. H. C. Cheng and J. Zhang. Specifying adaptation semantics. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, May 2005.

[4] I. Georgiadis, J. Magee, and J. Kramer. *Self-organising software architectures for distributed systems*. ACM, New York, New York, USA, Nov. 2002.

[5] R. Haesevoets, D. Weyns, and T. Holvoet. Architecture-centric support for adaptive service collaborations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):2–40, Feb. 2014.

[6] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, Jan. 2003.

[7] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11):1293–1306, 1990.

[8] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive Mob. Comput.*, 17(PB):184–206, Feb. 2015.

[9] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering*, pages 141–160. Springer International Publishing, Cham, Sept. 2014.

[10] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. In *Component Deployment*, pages 99–114. Springer Berlin Heidelberg, Berlin, Heidelberg, Nov. 2005.

[11] J. Odell, M. Nodine, and R. Levy. A Metamodel for Agents, Roles, and Groups. In *Agent-Oriented Software Engineering V*, pages 78–92. Springer Berlin Heidelberg, Berlin, Heidelberg, July 2004.

[12] D. Sykes, J. Magee, and J. Kramer. *FlashMob: distributed adaptive self-assembly*. distributed adaptive self-assembly. ACM, New York, USA, May 2011.

[13] N. Taing, T. Springer, N. Cardozo, and A. Schill. *A dynamic instance binding mechanism supporting run-time variability of role-based software systems*. ACM, New York, USA, Mar. 2016.

[14] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, Dec. 2007.

[15] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. On Patterns for Decentralized Control in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II*, pages 76–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.