Appeared in Nenad Medvidovic and Tetsuo Tamai (eds.), Proceedings of the 2nd ACM/IEEE 2010 ICSE Warm Up Workshop (WUP 2009), pages 37–40. ACM. April 2009.

An Adaptation Framework Enabling Resource-Efficient Operation of Software Systems

André van Hoorn, Matthias Rohr, Asad Gul
Graduate School TrustSoft*
University of Oldenburg
26111 Oldenburg, Germany
{van.Hoorn,Rohr,Gul}@informatik.uni-oldenburg.de

Wilhelm Hasselbring
Software Engineering Group
University of Kiel
24098 Kiel, Germany
wha@informatik.uni-kiel.de

ABSTRACT

This paper gives an overview about our current work on a framework which aims at operating component-based software systems more efficiently. Efficiency, in terms of the number of allocated data center resources, is improved by executing architecture-level runtime adaptations based on current workload situations. The proposed framework, called SLAstic, is described and open questions to be answered in future work are raised.

1. INTRODUCTION

Today's enterprise applications are complex, business-critical software systems. An important extra-functional characteristic of these systems is performance, consisting of timing behavior and resource utilization. Particularly interactive software systems which are accessible through the Internet are exposed to highly varying and bursty workloads, e.g., in terms of the number of concurrent users. For example, Figure 1 sketches the varying number of users which we derived from monitoring data of an online portal. The timing behavior of software systems is significantly influenced by the workload conditions. In order to continuously satisfy the contractually specified service level agreements (SLAs), a continuous capacity management strategy is required. Based on the anticipated workload conditions, this includes the provision of an appropriate computing and storage infrastructure as well as the deployment of the software components to this infrastructure.

Over the last years, capacity management was performed in a static and overprovisioning manner, i.e., the software components are deployed to a fixed infrastructure of application and database servers which satisfy the needs for the anticipated worst-case workload conditions. Future infrastructure demands are satisfied according to a "kill-it-with-iron" mentality: adding additional resources to the infrastructure or replacing existing resources by more powerful ones. The shortcoming of this approach is that during medium or low workload periods, the allocated resources may be heavily underutilized causing unnecessarily high operating costs due to power consumption or infrastructure leases.

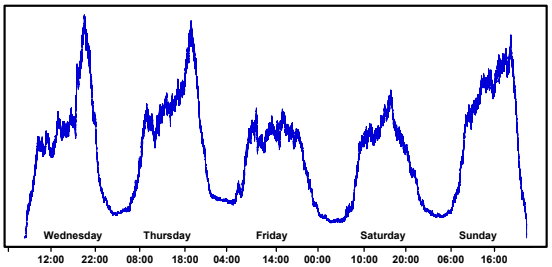


Figure 1: Varying workload intensity

In this paper, we present our work in progress on a framework called SLAstic (sɪˈlæstɪk) which aims at reducing operating costs for application service provision by adapting the architecture of distributed, component-based enterprise systems including its deployment with respect to the current and expected near-future workload situation while continuously satisfying the SLAs. Considered runtime adaptation operations are (1) the dynamic allocation and deallocation of data center resources as well as component-level (2) migration and (3) load-(un)balancing.

The remainder of this paper is structured as follows. Section 2 presents background and discusses related work. The proposed framework is described in Section 3. In Section 4, the conclusions are drawn and future work is discussed.

2. BACKGROUND AND RELATED WORK

The work presented in this paper is embedded into the research fields of self-managed systems and software performance engineering.

Self-managed or autonomous systems are those systems which are able to adapt themselves accordingly to their environment. The term autonomic computing was first used by IBM [6]. They proposed a reference model for autonomic control loops called MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) [6]. More recently, Kramer and Magee [7] proposed a software architecture for self managed

*This work is supported by the German Research Foundation (DFG), grant GRK 1076/1.

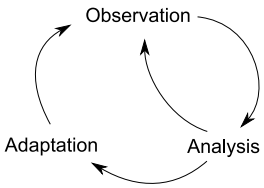


Figure 2: Self-adaptation cycle [13]

systems consisting of the following three layers: component control, change management, and goal management.

Self-adaptation can be described as a cycle of three logical phases [13]: Observation, analysis, and adaptation (Figure 2). The observation phase is concerned with monitoring (e.g., system behavior or system usage). Monitoring approaches have to deal with the trade-off between information quality (detail and timeliness) and performance overhead resulting from data transport and analysis.
The analysis phase detects triggers for adaptation and selects suitable adaptation operations, if adaptation is required. A straightforward means to implement this analysis are rules consisting of conditions and adaptation operations (e.g., "if disconnected, then reconnect"). A promising approach for defining rule sets for self-adaptive software systems was presented by Garlan et al. [5]: Conditions are specified not directly on architectural elements of the software system but on architectural styles. Specifying conditions on the higher abstraction level of architectural styles reduces the complexity of the analysis and allows one to develop general rules that may be reused in other software systems. During the adaptation phase, the selected adaptation operations are executed. Typical concrete adaptation operations for component-based software architectures are, for instance, component microreboot [3], and the replacement of components by alternative variants depending on current execution conditions [4]. Architectural runtime reconfiguration concepts (e.g., [9] and [12]) address the challenge to change a system without requiring a system restart.

Software engineering activities directed to meeting performance requirements are the subject of research in software performance engineering [14]. The state of the art is that architectural design models of the software architecture are annotated with performance information. Performance indicators like response times are computed by first transforming these high-level models into the well-known and mature performance analysis models like queueing models or stochastic Petri nets and then solving these models analytically or by simulation [1]. The SPT/MARTE profiles [10, 11] can be used to annotate UML diagrams. The Palladio approach for performance prediction of component-based software systems [2] uses UML-like models. Different approaches for adapting the software architecture for performance reasons through the use of self-adaptation exist (e.g., [4, 8]).

3. OUR SLAstic APPROACH

With SLAstic we propose an approach for reducing the operating costs of software systems by architecture-based run-

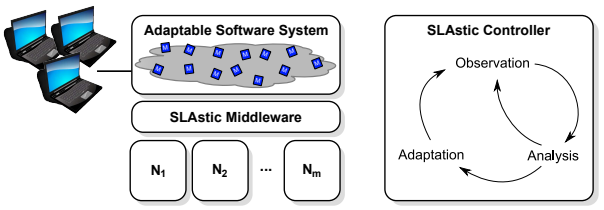
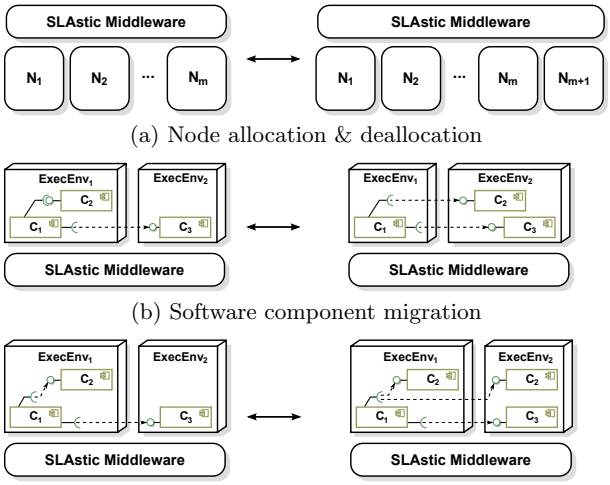


Figure 3: Integration of the SLAstic framework



(a) Node allocation & deallocation

(b) Software component migration

(c) Component-level load-(un)balancing

Figure 4: The three SLAstic adaptation operations

time adaptation for the resource-efficient operation while continuously satisfying the SLAs. The approach covers the complete software lifecycle, particularly design time models of the architecture are used during runtime for system adaptations through reconfigurations of the software architecture. Under increasing workload conditions adaptation operations are executed in order to fulfill the SLAs. Under decreasing workload conditions adaptation operations are executed in order to avoid resource underutilization.

Figure 3 illustrates how the SLAstic framework is integrated into a software system. The SLAstic middleware provides the instrumentation infrastructure and executes the adaptation operations triggered by the SLAstic controller. The SLAstic controller executes a control loop which is based on the self-adaptation cycle described in Section 2.

The following sections describe the proposed adaptation operations (Section 3.1), discuss architecture-level aspects and policies to be modeled (Section 3.2), and sketch the activities executed within the control loop of the SLAstic controller (Section 3.3).

3.1 Adaptation Operations

1. Node Allocation & Deallocation. This system-level operation consists of the allocation or deallocation of a server node. In case of an allocation, this includes the installation of an execution environment but it does not involve any (un)deployment operation of software components. Fig-

ure 4(a) illustrates this adaptation operation: Node N_{m+1} is allocated or deallocated, respectively.

2. Software Component Migration. This application-level operation consists of the migration of a software component, i.e., the undeployment from one execution context and the deployment into another. As an example, Figure 4(b) shows the migration of component C_2 between the execution environments $ExecEnv_1$ and $ExecEnv_2$.

3. Component-level Load-(un)balancing. This application-level operation consists of the duplication of a software component and its deployment into an execution context (as well as the reverse direction). Future requests to the component are distributed between the two component instances. Figure 4(c) illustrates the application of this operation to the component C_2.

3.2 Architectural Models & Adaptation Policies

The approach requires the explicit modeling of relevant aspects of the software architecture in design models. During runtime, these models are refined, kept synchronized with the runtime architecture and used for the analysis activities as described in Section 3.3. Another important purpose of the models is to automatically derive the instrumentation probes in order to obtain measurements of the modeled parameters which are used to calibrate and update the model. The following paragraphs give an overview of the aspects of the software architecture to be modeled.

Components and Assembly. The aspects to be modeled for each software component include component interfaces (required and provided) and performance-relevant concerns like estimates of calling frequencies for required interfaces, or the allocation and deallocation of software and hardware resources. For the application, the component assembly, i.e., the connection of components among each other through their interfaces, must be modeled. The components to which adaptation operations can be applied need to be labeled, e.g., using UML stereotypes.

Deployment Environment. The infrastructure on which the application will be deployed needs to be modeled. This requires the modeling of hardware and software resources on different levels of granularity as well as multiplicities and performance-relevant characteristics. Examples for hardware resources are server nodes and their interconnection (e.g., network links), CPUs and harddisks. Examples for software resources are execution environments (e.g., JBoss application server), and thread pools.

Component Deployment. This consists of the (initial) assignment of software components to execution environments. At runtime, the deployment model is affected by structural changes, caused by executed adaptation operations.

Performance Requirements. This includes the system-level specification of SLAs to be satisfied as well as internal performance specifications at the level of software component interfaces.

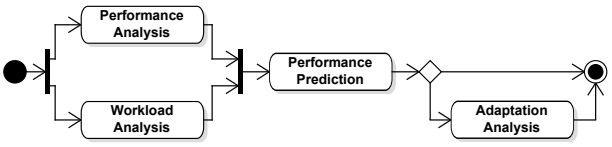


Figure 5: Activity diagram of the analysis phase

Adaptation Policies. The approach requires the definition of policies which specify under which circumstances, i.e. *when*, an evaluation of possible adaptations shall be performed. An example would be that such an analysis is to be performed in case the SLAs are expected to be violated in near future.

3.3 Control Loop

The SLAstic controller continuously navigates through the self-adaptation cycle described in Section 2. Thus, it periodically executes the phases *Observation*, *Analysis*, and *Adaptation*. The activities performed in these phases are outlined in the following Sections 3.3.1–3.3.3.

3.3.1 Observation

The software system is continuously monitored, as described before. During the *Observation* phase, the measurement data which is relevant for the elapsed period is extracted and preprocessed for the subsequent phases of the control loop. The preprocessing includes the mapping of lower level measurement data to the architectural entities.

3.3.2 Analysis

The *Analysis* phase is the core phase of the control loop. It results in the decision whether or not an adaptation is performed during the following *Adaptation* phase. The runtime models are updated and used for the evaluation of performance and adaptation alternatives. Figure 5 shows an activity diagram of the *Analysis* phase including the activities described in the following paragraphs.

Performance Analysis. The performance of the current system architecture is evaluated. Particularly, this includes whether or not the internal and external performance requirements are satisfied. Moreover, the performance parameters of the runtime models are updated and calibrated.

Workload Analysis. The workload of the elapsed observation period is determined and the near-future trends of the workload are estimated. This includes the analysis of the external (system interface) and internal (component interfaces) workload conditions.

Performance Prediction. Based on the performance and workload analysis activities of the previous activities, the performance of the current architectural configuration for the next observation period is predicted. Performance analysis models, which are derived from the architectural runtime models, are used to perform these predictions. An important result of this activity is a prediction of whether the SLAs are expected to be satisfied during the next observation period.

Adaptation Analysis. Based on the specified adaptation policies, this activity is only executed in case an adaptation shall be performed. The first step within this activity is a thorough performance evaluation, e.g., involving the identification of over- or underutilized resources and software components. The effect of possible adaptations on the performance is evaluated using similar techniques as they were used during the performance prediction activity. The result is a selection of adaptation operations to be executed during the subsequent *Adaptation* phase.

3.3.3 Adaptation

In case the *Analysis* phase resulted in the decision to perform an adaptation, the SLAstic middleware is triggered to execute the selected operations. Changes of the architecture which are due to the adaptation will be reflected in the runtime models as soon as the execution of the adaptation has been committed by the middleware. These changes include structural changes in the interconnection of components but also the resetting and adjustment of model parameter values which are no longer valid for the changed context. For example, after a component has been migrated to another execution environment, the response times corresponding to its operations are no longer meaningful.

4. CONCLUSIONS AND FUTURE WORK

In this paper we presented the design of our adaptation framework SLAstic which aims at operating software systems more efficiently. Efficiency, in terms of the number of allocated data center resources, is improved by executing architecture-level runtime adaptations based on current workload situations. Future work regarding the approach can be divided into four parts:

1. The three adaptation operations need to be formally specified and implemented as a proof-of-concept. Simulations shall be executed to examine whether they are a valuable means for improving the resource-efficiency.
2. The activities involved in the three phases described in Section 3.3 as well as the models used for the evaluation tasks have to be detailed. This includes to determine which aspects and runtime data of the software architecture need to be known for the analyses, which analysis models are usable for runtime evaluations, and what is a feasible time-granularity for the adaptation.
3. A language for modeling the relevant architectural aspects as well as the adaptation policies has to be specified.
4. An evaluation of the overall approach needs to be performed in order to examine whether it is both valuable and feasible for realistic applications. We plan to perform an evaluation by simulation in the first place before setting up a case study with a realistic application in the lab.

5. REFERENCES

[1] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[2] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A technique for cheap recovery. In *7th Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 31–44, 2004.

[4] A. Diaconescu and J. Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *Proceedings of the 2005 International Conference on Automated Software Engineering (ASE '05)*, pages 44–53. ACM, 2005.

[5] D. Garlan, S.-W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. In *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 23–46. Springer, 2003.

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

[7] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering (FOSE '07)*, pages 259–268. IEEE, 2007.

[8] Y. Liu and I. Gorton. Implementing adaptive performance management in server applications. In *Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '07)*, pages 12–21. IEEE, 2007.

[9] J. Matevska and W. Hasselbring. A scenario-based approach to increasing service availability at runtime reconfiguration of component-based systems. In *Proceedings of the 33rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA '07)*, pages 137–144. IEEE, Aug. 2007.

[10] Object Management Group. UML Profile for Schedulability, Performance, and Time. `http://www.omg.org/cgi-bin/doc?formal/2005-01-02`, Jan. 2005.

[11] Object Management Group. UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE), Beta 1. OMG Adopted Specification ptc/07-08-04. `http://www.omg.org/cgi-bin/apps/doc?ptc/07-08-04.pdf`, Aug. 2007.

[12] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, pages 177–186. IEEE, 1998.

[13] M. Rohr, S. Giesecke, W. Hasselbring, M. Hiel, W.-J. van den Heuvel, and H. Weigand. A classification scheme for self-adaptation research. In *Proceedings of the International Conference on Self-Organization and Autonomous Systems In Computing and Communications (SOAS'2006)*, Sept. 2006.

[14] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering (FOSE '07)*, pages 171–187. IEEE, 2007.