Non-Intrusive Policy Optimization for Dependable and Adaptive Service-Oriented Systems

Christian Inzinger, Benjamin Satzger, Waldemar Hummer, Philipp Leitner and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8, A-1040 Vienna, Austria
{lastname}@infosys.tuwien.ac.at

ABSTRACT

The Service-Oriented Architecture paradigm facilitates the creation of distributed, composite applications. Services are usually simple to integrate, but often encapsulate complex and dynamic business logic with multiple variations and configurations. The fact that these services typically execute in a dynamic, unpredictable environment additionally complicates manageability and calls for adaptable management strategies. Current system control strategies mostly rely on static approaches, such as predefined policies. In this paper we propose a novel technique to improve management policies for complex service-based systems during runtime. This allows systems to adapt to changing environments, to circumvent unforeseen events and errors, and to resolve incompatibilities of composed services. Our approach requires no knowledge about the internals of services or service platforms, but analyzes log output to realize adaptive policies in a non-intrusive and generic way. Experiments in our testbed show that the approach is highly effective in avoiding incompatibilities and reducing the impact of defects in service implementations.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; G.3 [**Probability and Statistics**]: Markov processes; H.3.5 [**Online Information Services**]: Web-based services

General Terms

Algorithms, Experimentation, Management, Reliability

Keywords

Adaptation, Autonomic Management, Dependability, SOA

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'12 March 25-29, 2012, Riva del Garda, Italy.
Copyright 2011 ACM 978-1-4503-0857-1/12/03 ...$10.00.

1. INTRODUCTION

Service-Oriented Architecture (SOA) [17] is a paradigm often used to realize large-scale software systems integrating various services across multiple providers. A basic principle of SOA is that services are loosely coupled and implement business functionality as black boxes. Although services by definition are stateless and solely react to the provided input, service-based systems are generally depending on multiple configuration parameters and operate in dynamic, error-prone environments. All these characteristics render management of SOA systems a highly challenging task.

To ensure that such systems work properly, SOA governance is responsible for monitoring the performance of single services, runtime environments, and the overall system or service composition. This information is used either by human operators that reconfigure the system manually, or as input to policies that provide automated adaptation [20]. However, it is difficult to anticipate each and every possible system state and provide an according policy rule. It is similarly problematic to foresee all effects of reconfiguration actions. Software bugs, for instance, can occur in every service implementation and runtime environment. Furthermore, the management policies or administrator actions may be in conflict, thus forcing the application into unwanted states or oscillation patterns [15]. In the end, human error can never be ruled out resulting in software defects [8], incorrect policies, or wrong adaptation actions, respectively.

We present a novel approach for dependable and adaptive control of service-oriented systems. Our technique poses low requirements on the system, and can be integrated into any SOA in a minimally invasive way. We propose a novel technique that infers a Markov Decision Process (MDP) model by analyzing the effect of system configuration parameters. The MDP model is automatically constructed from log output emitted by the services. The MDP is used to derive optimized policies considering software bugs, incompatibilities, and environmental changes. The main contributions of this paper are (1) a generic framework for dependable and adaptive control of SOA based on log analysis, (2) novel techniques for transforming typical SOA log data into an MDP representation, and (3) real world experiments to quantify the usefulness of this approach.

The remainder of this paper is structured as follows. In Section 2 we introduce a motivating scenario as the basis for discussion of our approach. Section 3 serves as a brief overview on MDP and related algorithms. The main contributions are presented in Section 4. Section 5 provides an evaluation based on the introduced scenario. Related work is discussed in Section 6, and Section 7 concludes the paper.

2. SCENARIO

In this section, we introduce a scenario that serves as the basis for discussion of our approach. Consider a SOA that provides an enterprise travel itinerary service, allowing employees to automatically manage flight, car and hotel reservations for their trips to customers. The application is implemented as a composition of multiple interacting services.

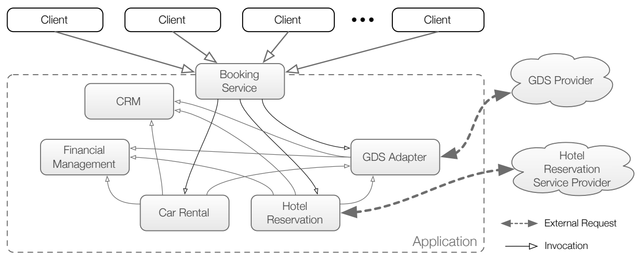


Figure 1: Architecture of the Travel Itinerary Application

A high-level overview of the services and interactions is shown in Figure 1. When an employee requests a new trip, the Global Distribution System (GDS) is requested to book a flight. To that end, the GDS adapter retrieves the customer address from the Customer Relationship Management (CRM) system, and uses external GDS Providers to find a suitable flight to an airport sufficiently close to the customer's site. At the same time the GDS needs to consider the available budget, as retrieved from the Financial Management (FM) system. Next, the Hotel Reservation Service (HRS) contacts external HRS Providers to find a hotel close to the destination address. The Car Rental Service (CRS) then issues a request to book a car for the specified period.

Parameter Type	**Examples**
Application-Specific	Skip Budget Check
Dynamic Binding	Providers for GDS, HRS
QoS Criteria	Max. Response Time of GDS
Runtime Environment	Resource Limits

Table 1: Parameters Defining the System Configuration

The system has various parameters that determine the current configuration state and influence the performance and functionality of the offered service (summarized in Table 1). Firstly, services expose application-specific properties that can be set explicitly (e.g., whether to skip a time-consuming budget check). Next, the composition depends on external services that are dynamically looked up and bound to at runtime (e.g., providers for GDS and HRS). Moreover, Quality of Service (QoS) properties are used to control the composition behavior (e.g., use only GDS providers with a certain response time). Finally, the runtime environment in which services execute influences the composition behavior (e.g., resource limits, request queue lengths). For reliable operation of a dependable system it is crucial to capture these parameters and to define management policies which aim at reconfiguring the application in response to undesired system states.

3. BACKGROUND

In this section, we provide an introduction to Markov decision processes (MDPs) [18], a mathematical framework for decision making, which serves as basis for our adaptive control mechanism. MDPs deal with decision making in an uncertain world, where performance depends on a sequence of decisions. At any point in time a decision making agent chooses among a number of available actions. The execution of an action affects the current state and a reward is given to the decision maker as feedback. How the state is affected probabilistically depends on the current state and the chosen action, but not on previous states. An MDP (S, A, T, R) is formally defined by a set of states S, a set of actions A, a transition model $T : S \times A \times S \to [0, 1]$, and a reward function $R : S \to \mathbb{R}$. The transition function $T(s, a, s')$ determines the probability of reaching state $s' \in S$ when executing action $a \in A$ while being in state $s \in S$. The utility of a state s is defined by the reward function $R(s)$.

The main problem for MDPs is to find a *policy* π that specifies which action to take for any state. In particular, $\pi(s)$ gives the action recommended by the policy for state s. An optimal policy is one that maximizes the expected rewards over time. Some algorithms, such as value iteration and policy iteration [18], are able to compute an optimal policy if transition model and reward function are known in advance. Reinforcement learning on the other hand solves the online variant of an MDP, where transition model and rewards are initially not known. Additional knowledge gathered by observations of transitions and rewards are used to iteratively improve a policy. Prominent reinforcement learning algorithms include Q-Learning [18] and Dyna-Q [21]. A core contribution of this work is to show how MDPs, a mathematically sound and well understood model, can be leveraged to manage complex real world service-based systems.

4. ADAPTIVE POLICY OPTIMIZATION

In this section we present our approach for deriving optimized policies that lead to dependable and adaptive service-oriented systems. Figure 2 illustrates the assumptions we are posing on the system. Since the internal structure of the system is not important for our approach a generic SOA model can be assumed, consisting of a number of runtime environments, each hosting a number of services. We suppose that the system initially is either managed by an administrator or predefined policies, which are to be replaced by an optimized policy. The crucial prerequisite is that status information and management actions are observable, as illustrated by the magnifying glass.

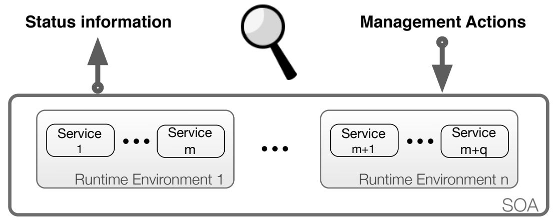


Figure 2: SOA without optimized policy. Information about the system status and management actions need to be observable in order to apply our approach.

We argue that the entity initially managing the system needs data about the system status and performed management actions anyways, hence such information is already available in some form. We do not assume that this information is produced centrally. Our approach is also applicable if each service and runtime environment produces the required data about status and reconfigurations individually. The status data needs to contain all information relevant to the system's performance and reliability. If, for instance, the response time of services matters then the status data should contain that information. However, there is no need for semantic annotations. Status information may be issued either at fixed intervals or triggered by events. It is sufficient, but not necessary, to emit only the indicators that have changed since the last output. At least if a performance indicator changes significantly, then there should be a status information update. We argue that all these requirements are met by virtually any SOA, which typically log the change of performance indicators and management actions.

In a nutshell, based on raw log data containing information about status updates of individual or compound components and management actions taken by the managing entity, we derive an optimized policy that takes over control, as shown in Figure 3.

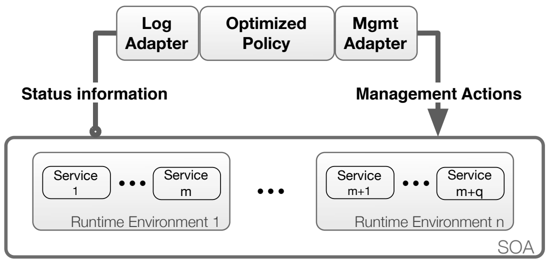


Figure 3: SOA with optimized policy, which takes system status data as input and outputs an optimized management action.

To be able to cover all kinds of systems with different models for providing status information and triggering management actions, we use a log adapter transforming status information into a canonical format and a management adapter responsible for executing a decision generated by the optimized policy.

Figure 4 shows the procedure of generating an optimized policy. Log data, containing information about relevant performance indicators as well as executed management actions, is used as input to the log adapter, which transforms the information into a canonical representation. The log adapter is also used during runtime to preprocess the monitoring data for the optimized policy. The MDP Creator generates an MDP based on the canonical log data. Since MDPs are well known in AI, the Policy Creator can be based on already existing algorithms to finally create the optimized policy.

To the best of our knowledge there is no comparable approach transforming generic log data into an MDP. This transformation is not trivial since, for instance, there is no reward function contained in the log data, which, however, is an essential part of an MDP. The final optimized policy will mimic the successful management actions while avoiding the ones that lead to errors. It is able to absorb complex relationships, and root causes to effects that are not directly linked. In the following we explain the three main components of the policy optimizer.

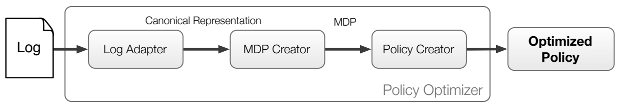


Figure 4: Functionality of the Policy Optimizer. It takes system information – usually in the form of logs – as input and outputs an optimized policy.

4.1 Log Adapter

The log contains information about changes of performance indicators and management actions. However, we cannot assume, that logs from different services or runtime environments adhere to a common standardized format. The log adapter is responsible for the aggregation of all available logs from the system's components, and transforms the applications-specific formats into a canonical representation, as indicated by the arrow labeled ① in Figure 5. Apart from the Management Adapter, which provides a mapping from abstract management actions to concrete calls of components, the implementation of the log adapter is the only functionality that needs to be provided by the user in order to apply our approach. All other parts are generic and provided by our framework.

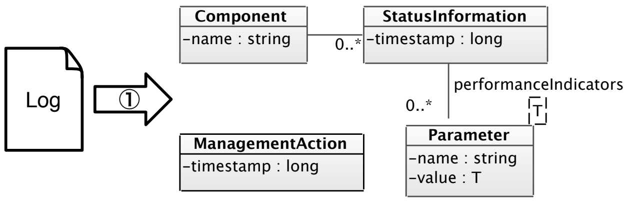


Figure 5: Functionality of the Log Adapter

We propose the model shown in Figure 5 as canonical format for capturing relevant information contained in the log data. *StatusInformation* emitted from system components includes updated performance indicators, captured in *Parameter*, consisting of a generic name-value tuple.

Errors are identified as a special type of performance indicator in the log entries, and an according *Parameter* named '*ERR*' is set. When a configuration change is performed – either by the administrator or by the currently active management policy – the system emits a log message that is captured by *ManagementAction* in the model.

4.2 MDP Creator

In this section we show how the canonically represented log data are transformed into an MDP (S, A, T, R).

MDP States and Actions

The first step towards a complete representation as an MDP is the extraction of states S and actions A as shown by the arrow labeled ② in Figure 6. The captured *StatusInformation* updates for each component are aggregated to *ComponentStates*, and further to a full *State* representation for the

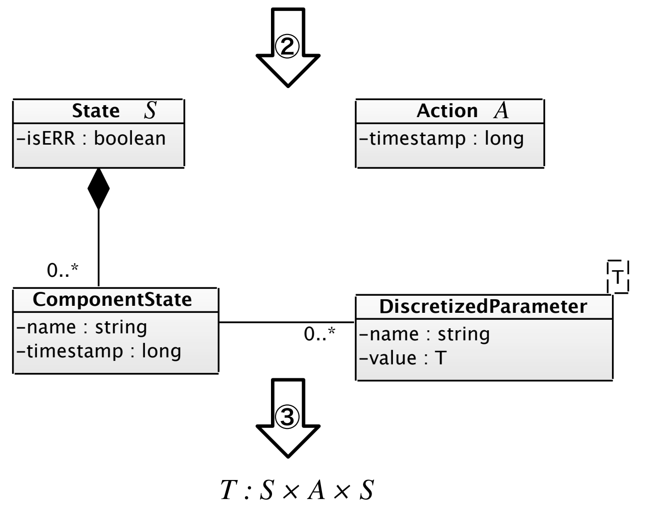


Figure 6: Generation of states, actions and transition model

system. This aggregation is based on either temporal proximity of the *timestamps* or on a correlation by a specified *Parameter* attribute, such as request id. If a state contains a component state with at least one *DiscretizedParameter* signifying an error, the state is labeled as error state by setting *isERR*. States S of the MDP must be finite. *Parameters*, however, contain continuous values and could result in an infinite number of states. Therefore, we conduct an automatic discretization of the observed values for each continuous *Parameter* (e.g., response-time), using a simple equal width interval method [12], resulting in *DiscretizedParameters*. Our framework is designed to allow for the usage of different discretization methods, such as equal frequency binning or Holte's 1R [13], but we found that the simple equal width binning method performed reasonably well. Actions A are constructed from each *ManagementAction* element. Additionally, a no-operation action (*NOP*) is added to the set of actions to allow the policy to remain in the current state, which allows to cover environment changes. Whenever performance indicators change without interference of policy or administrator action, the *NOP* action is used to represent external events not within control of our framework.

MDP Transition Model

In this step, we extract the transition model $T : S \times A \times S$ from the representation generated so far, as shown by the arrow labeled ③ in Figure 6. The transition model $T(s, a, s')$ assigns the probability of reaching state s' from state s when performing action a. We derive the transition model by employing a modified Passive ADP Agent [18] algorithm. Algorithm 1 is invoked for each *State* and *Action* in chronological order, incrementally constructing a transition model from the observed data.

MDP Reward Function

To complete the generation of the MDP (S, A, T, R), we finally need to derive a reward function $R : S \to \mathbb{R}$ from the model. So far, the required data was in some way directly extractable from the log output. The reward function, however, is not readily available, as neither the logs, nor the models generated so far provide any reward signals.

Algorithm 1 Transition model learning

```
Input: $s'$ current state,
        $a$ previously taken action
  persistent: $T(s, a, s')$, a transition model
        $N_{sa}$, a table of frequencies for the state-action
              pairs, initially zero
        $N_{s'|sa}$, a table of outcome frequencies given state-
              action pairs, initially zero
  if $s$ is not null then
    increment $N_{sa}(s, a)$ and $N_{s'|sa}(s', s, a)$
    for each $t$ such that $N_{s'|sa}(t, s, a)$ is nonzero do
      $T(s, a, s') \leftarrow N_{s'|sa}(t, s, a)/N_{sa}(s, a)$
    end for
  end if
```

We propose a novel approach to finding a reward function from preprocessed log data, based on the assumption that a majority of the actions taken by the initial managing entity are reasonable. The basic idea is that if the initial manager performs some action a in state s_1 which leads to state s_2, i.e., $s_1 \xrightarrow{a} s_2$, then we assume that state s_2 is more desirable than state s_1. In the case of contradictory transitions where there are a number of transitions $s_1 \xrightarrow{\bullet} s_2$ and $s_2 \xrightarrow{\bullet} s_1$ we assume that the majority of management actions was beneficial. In any case, failure conditions are to be avoided. Figure 7 graphically illustrates how we generate a reward

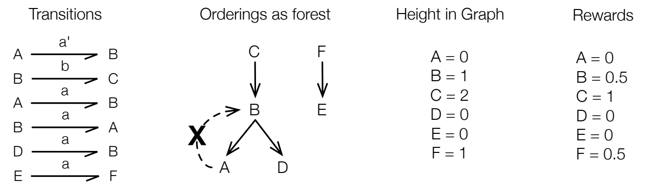


Figure 7: Illustrative example of our reward function

function R that assigns rewards to states. The input is observed transitions $T^O \subseteq S \times A \times S$. As a first step a forest $F = (S, E)$ is created, where the nodes consist of states S and edges E. Furthermore, the following condition holds: $(s_1, s_2) \in E \implies |\{s_1 \xrightarrow{\bullet} s_2\}| \leq |\{s_2 \xrightarrow{\bullet} s_1\}|$. The reward function $R : S \to [-1, 1]$, which maps states to rewards, is defined as follows

$$R(s) = \begin{cases} \frac{height_F(s)}{height(subtree(s))}, & \text{if } s \neq ERR \\ -1, & \text{if } s = ERR \\ 0, & \text{otherwise,} \end{cases} \tag{1}$$

where $height_F(s)$ returns the length of the longest path from node s to a leaf node in F, *subtree*(s) return the tree s belongs to, and *height*(t) returns the height of tree t, i.e., the height of its root node.

The definition ensures that failure states give lower (negative) rewards than all non-failure states. Non-failure states give a higher reward if they have been favored by the initially managing entity. For a state without any occurrence in the log, 0 reward is given.

4.3 Policy Improvement

The output of the MDP creator is a system description in the form of an MDP. In the policy improvement step, well-known decision making algorithms can be applied to optimize the management policy. We have incorporated both Policy Iteration [6] for adapting a policy to avoid error states in an environment, that is not expected to change significantly, as well as the Q-learning [22] algorithm, able to iteratively adjust to changing environments.

Finally, to utilize the optimized policy, it is deployed, replacing the initial policy. The policy optimization can be arbitrarily repeated. This allows to take additionally gathered data into account to refine the policy and react to changes in the environment.

5. EVALUATION

To evaluate our approach, we have created a simulation testbed, allowing for quick and easy specification of complex composite applications, their runtime properties, as well as the initial management policies. The simulation testbed is implemented using Ruby[1]. It provides a domain-specific language (DSL) for the definition of the service behavior, e.g., interaction with other services, and processing cost. Furthermore, it allows for the specification of configuration variants and parameters that can be dynamically changed during runtime. Listing 1 shows a simplified definition of the HRS's "find hotel" method in the "search using all external providers" configuration. It adds a new configuration variant to the HRS's "find hotel" method, which invokes 3 partner services CRM, FM, and GDS. Furthermore, to model interaction with external services, the 'cost' value defines the mean of the normally distributed invocation time.

```
add("hrs#find_hotel").add_config("all#regular") do
  invoke "crm#get_customer_address"
  invoke "fm#get_budget"
  invoke "gds#get_flight_information"
  cost 4 # estimated cost of external requests
end
```

Listing 1: Service method definition: HRS "find hotel", demonstrating basic capabilities of the developed simulation testbed, i.e., invocation of other services and their methods, simulation of external processing costs, and the definition of configuration variants.

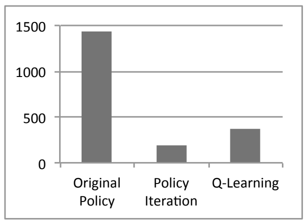
The simulation testbed furthermore allows for the specification of initial management policies for the created services using a similar DSL. Additionally, a management interface is provided, allowing to replace the initial policy with the optimized one. We also provide for several different user interaction patterns to simulate varying numbers of users with different behaviors.

The policy optimization framework is implemented as a Java library. Currently, there is one log adapter[2], compatible with the simulation testbed log, and conforming to the specification. The optimization algorithms, i.e., policy iteration and Q-Learning, are optimized implementations of the algorithms presented in [18].

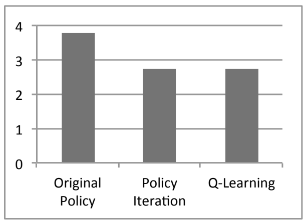
We implemented the scenario application as described in Section 2, with multiple concurrent clients sending requests

[1]`http://ruby-lang.org/`

[2]An exemplary log4j configuration and helper methods conforming to the implemented format are available at `https://gist.github.com/1197839`



(a) Total Errors



(b) Processing Time (s)

Figure 9: Evaluation result summary. Our approach was able to significantly improve the initial management policy using both, policy iteration and Q-Learning. Error occurrence was reduced by more than 70%, and average processing time decreased by over 27%.

to the Booking Service. The initial policy is designed to degrade the quality of services for faster processing times if load rises above a certain threshold. In the interaction of the HRS and GDS services there manifests a hidden incompatibility in one certain configuration constellation. In that scenario, the HRS is configured to only consider major airports for finding hotels which increases the probability of empty results for the travel itinerary in combination with the GDS, configured to look for the cheapest flights, which might use smaller airports. This situation is perceived as an error and mitigated by the HRS querying all available partner services incurring additional processing overhead. This special case is not addressed in the otherwise useful initial policy. In general, as argued before, it is very difficult to anticipate all possible failure scenarios, which calls for adaptive management policies as proposed in this paper.

The policy optimization is performed after a bootstrapping phase which is needed to collect log data. This phase is completed if 3000 requests have been processed. Each single service invocation triggers the output of at least one status update information. The requests are issued in a sawtooth pattern to simulate varying load patterns. The evaluation period, in which the performance of policies is assessed, consists of an equal amount of requests following the same pattern. The evaluation was performed on a machine with a 2.4GHz quad-core Intel Xeon E5620 CPU with 12MB shared L3 cache, 16GB RAM, running Ubuntu 10.04 LTS.

The results in Figure 9 show, that we are able to reduce the occurrence of errors by over 70% using the Q-Learning policy improvement, and by more than 80% using the policy obtained through policy iteration. Furthermore, the average request processing time is reduced by over 27% due to the reduced impact of the performance penalty incurred when errors are encountered. The worse performance of the Q-Learning algorithm with regard to total errors can be attributed to the slower convergence of this algorithm for the given problem. However, Q-Learning is more suitable for iterative, online policy improvement.

Figure 8 presents detailed evaluation results. The section 'Queue Length' shows the system's processing queue and illustrates that the chosen request pattern induces significant stress on the application. The three 'Processing Time' sections show the time it took the system to process each single request. The optimized policies maintain appropriate

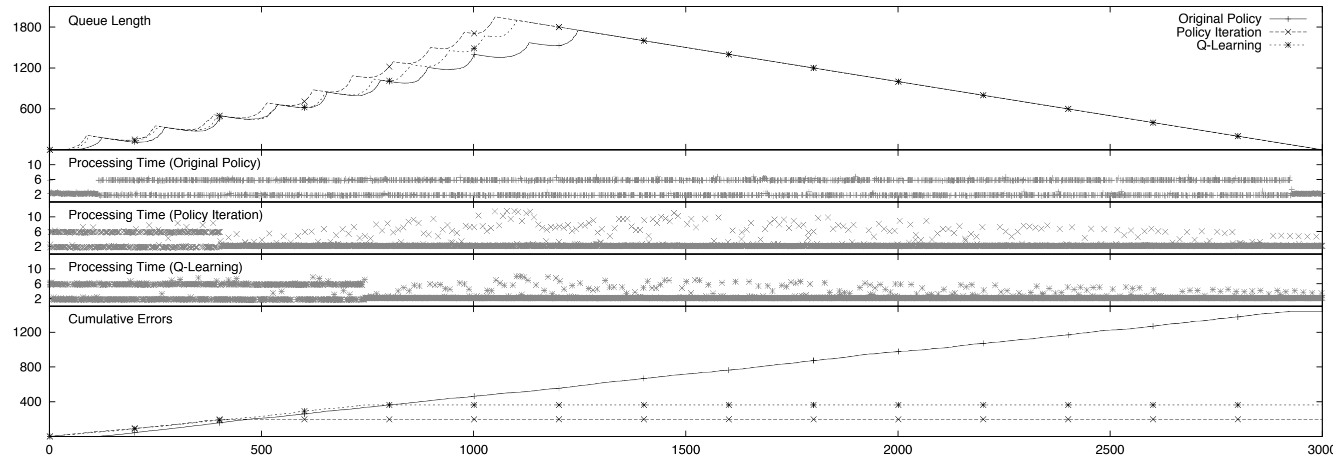


Figure 8: Results of the conducted experiment. The x-axis represents the progress of the evaluation based on the number of request processed. 'Queue Length' shows the number of requests pending for processing. The three 'Processing Time' panels depict the time it took for individual requests to be processed for each of the three evaluated policies. 'Cumulative Errors' shows the aggregated number of errors occurred during the evaluation run.

processing times, and allow for a degradation in processing time to avoid errors. The last section, 'Cumulative Errors', shows the aggregated number of errors encountered during the evaluation. In our scenario, both optimized policies outperform the original management strategy.

6. RELATED WORK

Autonomic and policy-based management, fault tolerance, and self-healing systems research has received a lot of attention in the past and continues to do so until today. Recently, these areas are becoming more and more relevant for SOA environments, as unaided optimization of configuration, collaboration, and error mitigation strategies, are essential for the successful implementation of a loosely-coupled SOA.

An approach to autonomic SLA-based management of distributed systems is presented in [1], proposing a hierarchical architecture of autonomic managers using a traditional MAPE cycle, each responsible for certain non-functional concerns of an application according to a predefined policy. Similarly, [14] presents an autonomic framework for preventing SLA violations. The approach presented in [19] applies automated planning algorithms for system reconfiguration based on user-defined objectives.

Several approaches have been presented in the area of self-healing web service integration and composition (e.g., [9, 11, 10]), as well as self-healing BPEL processes (e.g., [5, 4, 16]). The presented techniques are concerned with optimizing the behavior of integrated and/or composed services and business processes, using static a priori policies, and, contrary to our approach, assume in-depth knowledge of the services to be managed.

An architecture for self-manageable cloud services is presented in [7]. Similar to our approach, services provide management interfaces to allow for the control by the autonomic manager. However, the presented solution requires for the autonomic manager to know the service capabilities ahead of time.

A notable method for policy-driven autonomic management using reinforcement learning techniques is presented in [3, 2]. The approach allows for optimization of runtime behavior of managed applications by analyzing and deconstructing the provided management policies, utilizing a complex management architecture. In contrast to our approach, managed applications and components must be completely controlled using the proposed framework, policies enforced by internal mechanisms cannot be taken into account.

7. CONCLUSION

In this paper we propose a novel approach for optimizing control policies for SOA, leading to dependable and adaptive service-oriented systems. The approach makes minimal assumptions about the structure and capabilities of the system. We present a new technique to transform log data into a Markov Decision Process representation, which is used to generate an improved control policy that takes into account dynamics of the environment and software defects. This is done at runtime without need for human intervention. Experiments conducted in a testbed consisting of real Web services show that the adaptive policies are capable of mitigating the effects of defects and incompatibilities between collaborating components.

As future work we plan to integrate service level objectives, as well as request payload data, into the policy generation in addition to log data. We will also investigate how our approach can be applied to Web service compositions and business process optimization. Another future research direction includes to consider not only the service level but also the resource level, i.e., to control the mapping of services to resources. The techniques presented in this paper allow to manage relatively complex service-oriented systems. However, if large-scale highly complex systems are to be controlled, algorithms for complexity reduction, such as principal component analysis, could be employed. Active learning promises better exploration and faster learning rates for Q-learning.

Acknowledgement

The research leading to these results has received funding from the European Commission's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257483 (Indenica).

8. REFERENCES

[1] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Autonomic management of non-functional concerns in distributed & parallel application programming. In *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009.*, pages 1 –12, May 2009.

[2] R. Bahati and M. Bauer. Modelling reinforcement learning in policy-driven autonomic management. *International Journal On Advances in Intelligent Systems Volume 1, Number 1, 2008*, 2008.

[3] R. M. Bahati, M. A. Bauer, and E. M. Vieira. Policy-driven autonomic management of multi-component systems. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, CASCON '07, pages 137–151, New York, NY, USA, 2007. ACM.

[4] L. Baresi and S. Guinea. Dynamo and Self-Healing BPEL Compositions. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 69–70, Washington, DC, USA, 2007. IEEE Computer Society.

[5] L. Baresi, S. Guinea, and L. Pasquale. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, ESSPE '07, pages 11–20, New York, NY, USA, 2007. ACM.

[6] R. Bellman. *Dynamic programming*. Dover Pubns, 2003.

[7] I. Brandic. Towards self-manageable cloud services. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 2, pages 128 –133, July 2009.

[8] A. Brown and D. Patterson. To err is human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY01)*, 2001.

[9] K. Chan and J. Bishop. The design of a self-healing composition cycle for web services. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on*, pages 20 –27, May 2009.

[10] G. Denaro, M. Pezze, and D. Tosi. Designing self-adaptive service-oriented applications. In *Autonomic Computing, 2007. ICAC '07. Fourth International Conference on*, page 16, june 2007.

[11] G. Denaro, M. Pezze, and D. Tosi. Shiws: A self-healing integrator for web services. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 55–56, Washington, DC, USA, 2007. IEEE Computer Society.

[12] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In *Proceedings of the 12th International Conference on Machine Learning*, pages 194–202. Morgan Kaufmann Publishers, Inc., 1995.

[13] R. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine learning*, 11(1):63–90, 1993.

[14] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. Monitoring, Prediction and Prevention of SLA Violations in Composite Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'10)*, pages 369–376, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[15] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *Software Engineering, IEEE Transactions on*, 25(6):852 –869, nov/dec 1999.

[16] S. Modafferi, E. Mussi, and B. Pernici. SH-BPEL: A Self-healing Plug-in for WS-BPEL Engines. In *Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, MW4SOC '06, pages 48–53, New York, NY, USA, 2006. ACM.

[17] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, 2007.

[18] S. Russell, P. Norvig, J. Candy, J. Malik, and D. Edwards. *Artificial Intelligence: A Modern Approach*. Prentice hall, 2010.

[19] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer. Using automated planning for trusted self-organising organic computing systems. In *ATC*, volume 5060 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2008.

[20] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4):333–360, 1994.

[21] R. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, volume 216, page 224. Citeseer, 1990.

[22] C. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.