Autonomic Configuration Adaptation Based on Simulation-generated State-transition Models

Mike Smit and Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, Canada
Email: msmit,stroulia@cs.ualberta.ca

***Abstract*—Configuration management is a complex task, even for experienced system administrators, which makes self-managing systems a particularly desirable solution. This paper describes a novel contribution to self-managing systems, including an autonomic configuration self-optimization methodology. Our solution involves a systematic simulation method that develops a state-transition model of the behavior of a service-oriented system in terms of its configuration and performance. At run time, the system's behavior is monitored and classified in one of the model states. If this state may lead to futures that violate service level agreements, the system configuration is changed toward a safer future state. Similarly, a satisfactory state that is over-provisioned may be transitioned to a more economical satisfactory state. Aside from the typical benefits of self-optimization, our approach includes an intuitive, explainable decision model; the ability to predict the future with some accuracy avoiding trial-and-error; offline training; and the ability to improve the model at run-time. We demonstrate this methodology in an experiment where Amazon EC2 instances are added and removed to handle changing request volumes to a real service-oriented application. We show that a knowledge base generated entirely in simulation can be used to make accurate changes to a real-world application.**

***Keywords*-self-configuring software, autonomic computing, simulation, service-oriented architectures, adaptation**

I. Introduction

The demand for dynamic, distributed, component-driven software has led to increased adoption of service-oriented software systems. The nature of these systems limits the effectiveness of design-time configuration decisions; additional copies of service instances may be deployed at run time; new services may be removed and new services may be introduced in an existing composition; the provisioned network bandwidth can be changed. Deploy-time and even run-time configuration decisions are required to manage the system in the face of varying business requirements, service providers and consumers, and available resources [1].

Like its namesake in biology, the goal of an autonomic system is to be self-managing. According to [2], self-management involves four main activities: *Monitoring*, *Analysis*, *Planning*, and *Execution* (MAPE). The *monitoring* stage uses sensors to measure key load and performance attributes of the system. The *analysis* stage identifies any metrics that are outside the expected/desired ranges or violate rules stated in the system's SLA (Service-Level Agreement). Furthermore, it attempts to identify the cause of the problem: perhaps a server is under-performing, or perhaps the load suddenly increased. Proactive-analysis methods might also attempt to predict the future by identifying trends or matching current behavior to past patterns. In the *planning* stage, the system decides how to react to the fault by identifying a set of actions that may remedy the situation. These actions are implemented in the *execution* stage via actuators.

In this paper, we describe our methodology for autonomic decision-making in the area of self-optimization, focusing on the *analysis* and *planning* stages. An underlying assumption is that monitoring data can be acquired accurately and with enough frequency; similarly, that execution (i.e., reconfiguration) can occur without prohibitive overhead or delay. Our evaluation was conducted in both simulation and in a real-world cloud computing environment where these assumptions hold.

Our self-optimization decision-making methodology involves three steps. The first step uses our simulation framework (see [3], [4]) to collect a representative number of examples of its expected behavior under different conditions of load, configuration and SLA constraints. The sample subject system in this paper is a text-processing service called TAPoRware (referred to as the *simulated system*). The motivating reconfiguration scenarios envision the adaptation of the number of servers on which the system services are deployed, as well as changes to the number of deployed instances of its services.

The second step of our method creates the system's behavioral state-transition model from the data generated by the simulation step. To that end, the metrics recorded in each monitoring step are discretized to produce a more coarse-grained representation of the recorded data, which is clustered into classes, each of which represents a behavioral *state*. Two states are related with a *transition* between them if an instance of the source state is followed in the simulation by an instance of the destination state. These transitions essentially represent changes in demand (when the system load metrics increase) or changes in the system configuration (when the system configuration changes).

The final step of our method involves the autonomic manager, which, at run time, monitors the simulated system by taking periodic snapshots of its load and configuration. This data enables it to identify in which of the behavioral states it is, thus tracking the progress of the system through the state-transition model. When the system is in a state known to violate (or to potentially lead to violations of) its SLA, a search for a configuration change transition (or a series thereof) that will lead to a satisfactory state is initiated. The subject system is reconfigured accordingly, and monitoring resumes.

We evaluated this methodology and our implementation on our test-bed text-processing system by comparing the performance-to-cost ratio of a series of static configurations, a manually adapted configuration, and our autonomic manager. We found that the autonomic manager was capable of significantly improved performance compared to a baseline static system, and came close to meeting a configuration manually adapted by an expert user at an even cheaper cost.

The rest of this paper is organized as follows. The model-creation step is described in Section II. The autonomic manager is discussed in Section III. The relation of our work to the state of the art is review in Section V. We conduct an evaluation first in simulation and then in a real-world environment (Section IV). Finally, Section VI summarizes the contributions.

II. System Behavior Modelling

Using the existing simulation, we ran a variety of simulations with different configurations and different incoming request loads and recorded the performance. We used a set of static starting configurations (one load balancer, 1-6 servers available, each server running a web service with 6 available operations), then modified the configuration manually in response to changing demands. We identified and recorded several metrics as relevant to performance, including queue length for each server, CPU utilization for each server, response time for each request, the total time required to process the sample set of requests, a rolling average response time and standard deviation, an overall average response time and standard deviation, the request arrival rate, and the current system configuration. These are captured and stored in *snapshots* every 5 seconds. The final product for this paper is a dataset of 15 simulation traces, each covering 120 minutes of simulated time, for a total of 21,600 snapshots from which a state-transition model of the system behavior was constructed.

Once the simulation step has been completed and a trace of the system behavior under different loads and configurations has been recorded, the behavioral model of the subject system can be constructed. This model, identifying the system's observed behavioral states and the transitions between them, forms the basis of the autonomic decision-making step. This section describes the process of identifying states and transitions, given a series of simulation traces.

A. States

A state is a set of snapshots that are similar. For each recorded snapshot in the simulation traces, we calculate and record a *snapshot descriptor*. A snapshot descriptor is a triple. The first value of the triple defines the state's compliance with the SLA: *Satisfactory*, *Unsatisfactory*, or *Boundary* (the *SUB metric*). Boundary states are states which meet the SLA, but which have transitions to unsatisfactory states[1]. The second element of the triple is a set of configuration-related parameters, i.e., a set of metrics related to the current environment (number of incoming requests, etc.). The third value is a set of performance metrics.

[1]Note that this value is not static: the same simulation data with a different SLA will produce different states as the SUB metric may change.

The internal structure of the snapshot descriptor, namely the actual metrics included in the configuration-parameter and performance-metrics set, depends on the subject system. The challenge is to identify a tuple that includes measurable information sufficient to characterize the performance of the system without including extraneous information. We used all performance-related metrics captured in the simulation step. To control the size of the state space, we cluster snapshot descriptors into equivalence classes, called *states*. The snapshot descriptors within each state are "similar enough", in that there is no significant difference in their environment, performance, or configuration metrics[2]. Each state is annotated with a cost, the cost of the cheapest configuration of all its snapshot descriptors. The function used to determine the configuration costs could be the cost of hardware, the cost of maintenance, the "green-ness", *etc.*

Whether two values of a performance metric are "similar enough" depends on the level of precision/abstraction at which metrics are examined. At the lowest level of abstraction, any variation in any metric is considered a change. If the metrics involved are measured using real numbers (and not restricted to a range of integers), the state space is theoretically infinite. To restrict the size of the state space while still producing meaningful results, we discretize ranges of values into "windows" where all values within a window are considered equal to each other. For enumerated performance metrics, each enumerated value defines a "window"; for example, the SUB metric has 3 possible values and has window size 3. For numeric metrics, the size of the window is chosen to give the desired number of windows and therefore the desired state space, while remaining appropriate for that metric. We tested three different discretization strategies before choosing a fixed window size based on the actual range of values. The actual range of observed values is divided by the desired number of windows to produce the window size; we used a small desired number of windows (4-10 depending on the metric).

When constructing the model, as each new snapshot descriptor is recorded, its abstracted representation is calculated, substituting metric values for the corresponding intervals. If a state already exists for this abstract representation, the original descriptor is clustered in it, and the state-population counter is incremented. Otherwise, a new state entry is added to the model. The theoretical state space is 195; in practice, metrics tended toward the minimum and maximum values, so although there were five "windows" for each metric, the majority fell in the first or last window. The actual number of states identified is 83.

B. Transitions

The model includes two types of transitions between states. First, *need-based transitions* (n-transitions) occur when the environment changes, and the software is asked to provide out-of-the-ordinary service (this new level may be the new "ordinary"). In our ongoing example of performance, this takes the form of a load change: increased requests, increased request size, network congestion, and so forth. Thus, a need-based transition implies that there is a

[2]Note that the SUB metric is ignored in the clustering process.

pair of states, s_{src} and s_{dest}, such that (a) s_{src} is followed by s_{dest} in the simulation and (b) the load in s_{dest} has significantly different load metrics associated with it from the load metrics associated with s_{src} (namely the former load metrics fall in different windows from the latter load metrics). In addition to the metrics that change between s_{src} and s_{dest}, a $n-transition$ is characterized by its cost differential, namely $cost(s_{src}) - cost(s_{dest})$.

The second type of state transitions, *configuration-based transitions* (c-transitions), occur when the configuration changes (*i.e.*, s_{dest} has a different configuration from s_{src}). This transition may be caused by a system administrator or the autonomic manager. These transitions are also annotated by their cost differential. However, they also imply a secondary cost, namely the cost of actually making the change. For example, adding a second server involves a deployment cost in addition to the ongoing maintenance and energy costs implied by the additional server.

As each state may have any number of these transitions, the number of occurrences of a given transition t in the simulation traces is calculated. Then, the estimated probability of a given transition occurring is given by its occurrence count relative to the other transitions.

The process also recognizes an additional type of transitions, *performance transitions*. While a configuration and the incoming load on the software will define the system's performance, it generally takes some time after a load or configuration change for the system to stabilize, during which the configuration and load will not change, but the performance will (*i.e.*, s_{dest} has the same configuration and load as s_{src} but different performance metrics). The process recognizes these transitions, but they do not alter the conceptual model: if there is a chain of performance transitions following a configuration or need transition, this "chain" can be followed to the eventual end state.

If two subsequent snapshot descriptors are not equivalent, the model-construction process analyzes what has changed: the configuration, the environment, or both. For configuration changes, the deployment cost is calculated and a c-transition is created. For load changes, a n-transition is created. When both types of metrics have changed, first a n-transition is created (where only load-related elements in the snapshot descriptor change), creating a new intermediate state if an equivalent state does not already exist. Next, a c-transition is also created (again creating a placeholder if need be). Two transitions are considered equivalent if they are both the same type (n or c) and they transition between the same two states; a counter is incremented when duplicates are encountered. The probability of each transition out of a given state is determined after state-transition detection completes. We identified 228 unique transitions in our experiment (from a total of 1,525 transitions).

III. Autonomic Management

Our autonomic-management method makes decisions by monitoring the application and periodically recording a snapshot of its metrics and classifying the snapshot as an instance of a corresponding state in the state-transition model of the system behavior. The decision on whether to make any changes to the system configuration, and which change exactly, rests on the SUB metric of the identified state.

If the current system state is classified as an unsatisfactory state, the state space is searched, starting from the current state and following only c-transitions until a satisfactory state is found. The search algorithm used for that purpose is iterative-deepening depth first search (IDDFS). Based on our observations of the generated state spaces, a satisfactory state is likely to be within a few levels of the current state. IDDFS finds such states quickly with space-complexity similar to DFS. It can be aborted and will return the best result found so far (*e.g.*, a state with improved performance but a boundary state) if it does not find a satisfactory state in the time allotted. The c-transition identified by the search algorithm is executed and the application is observed to ensure the target state is reached. A timestamp of the last change is recorded.

If the current state is classified as a boundary state, a search is initiated just as for the unsatisfactory state; however, the configuration changes implied by the transition sequence are only executed if the identified target state is superior to the current boundary state (an S-state, or a B-state with improved performance and/or lower cost).

If the current state is classified as a satisfactory state, a search for a path of c-transitions that leads to another satisfactory state is initiated. If a path is identified, it is executed if (a) the time elapsed since the last change is sufficient (we use 5 minutes), and (b) the new state is cheaper than the current state. This is intended to remove over-provisioned resources while avoiding "churn" of repeated adds/removes for performance levels at or near SLA levels.

If the SLA changes, each of the recorded snapshot descriptors is revisited to recalculate its associated SUB metric. If after this process the system is found to be in an unsatisfactory state, the adaptation algorithm above is invoked in order to transition to a satisfactory state. Recall that every unsatisfactory state must have at least one c-transition to a satisfactory state; if new unsatisfactory states are added as a result of the SLA change, this requirement might be violated and additional simulations may be required to update the state-transition model.

Depending on the completeness of the simulation-generated state-transition model, the running application may encounter states not yet seen in the simulation traces or may transition to existing states via previously undetected transitions. In these cases, our model-construction algorithm adds the appropriate states and transitions to the model. If the new state is unsatisfactory, there will be no strategy to adapt the configuration. Our methodology specifies that an update to the state-transition model must be scheduled, but the immediate reaction is to use a modified state equivalence function to identify similar states that violate the SLA in the same way and use one of their c-transitions to attempt a move to a satisfactory state.

IV. Evaluation

To evaluate our approach, we implemented an autonomic manager and conducted tests a realistic cloud-computing scenario. The autonomic manager is capable of making configuration changes to a TAPoRware configuration. The state model described in the preceding sections was

Config	Resp. Time	Cost	Performance
4 Servers	91.59	0.0%	0.0%
5 Servers	6.36	18.8%	93.1%
6 Servers	3.36	42.1%	96.3%
Autonomic	6.61	-3.6%	92.8%

Table I
COST AND PERFORMANCE METRICS FOR 6 CONFIGURATIONS AND THE VARIABLE DATA SET AS TESTED IN A REAL CLOUD.

used as the knowledge base. The experiments were based on monitoring and adapting the application by adding or removing servers in response to changing request loads. The service level objective was a very ambitious response time of 5 seconds, with a maximum queue length of 20 requests (recall this is not a simple web request, but CPU-intensive analysis of texts up to 7 MB in size).

We installed the TAPoR web service on an Amazon Elastic Compute Cloud[3] (EC2) instance (small), which is a 32-bit system with 1.7 GB of memory and one EC2 compute unit (roughly a 1.0-1.2 GHz 2007 Xeon processor). From this instance, we created an Amazon machine image to enable easy replication. We deployed instances to match the number of servers used in the various simulated experiments (namely 4, 5, or 6 servers). A load balancer acts as a single endpoint to which clients send requests, tracks the number of instances that are provisioned (either fixed or varied by the autonomic manager), and forwards incoming requests to the instance with the smallest number of outstanding requests.

On a local machine, we run the real world testing application which emulates thousands of web service client requests to TAPoR services, based on a recorded set of requests generated from real logs of the deployed service by an individual not familiar with this project. Four different conditions were used; in the first three conditions, a fixed number of servers (4, 5, and 6, respectively), reflecting static configurations, with no configuration changes made at run time. For the fourth condition, our autonomic manager using 1-6 servers, monitored and changed the system configuration as it deemed appropriate, based on management method described above.

For each of the above conditions, we recorded average response time as the performance measure. We also calculated configuration cost. A server contributed to the cost if it was configured to process requests, whether it actually received requests or not. A slight delay was imposed to emulate start-up time for a new server, and another delay was imposed before server shutdown was initiated to give the autonomic manager time to reverse its decision and re-add the server.

We set a fixed four-server configuration as the baseline and made comparisons relative to that. The results are shown in Figure 1 and Table I. We see that the autonomic approach costs less than 4 servers, yet offers performance comparable to having 5 servers. Compared to a 4-server baseline, it reduces response times by 93% while reducing costs by 4%.

It should be noted that the creation of the knowledge base used a request arrival rate and a configuration that were both set manually by an author of this paper over

[3]http://aws.amazon.com/ec2/

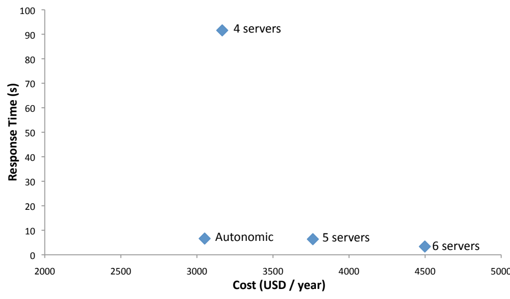

Figure 1. The performance / cost trade-offs for the variable data set as tested in a real cloud.

the 18 hours of simulated time. However, the request set used in the evaluation was entirely disparate and created by a third party. A three hour sequence of requests is not insignificant, but repeated tests and statistical analysis are needed to ensure the correlation between the real and simulated evaluations and the improvements offered by the autonomic manager are statistically significant.

The evaluation of our approach relied on the relatively simple task of adding and removing servers based on load. Further, the service system used 6 web services per server, with the servers all hosting the exact same services. As the configurations grow more complex (distributing services across servers, for instance), the state space will grow and coverage will be more challenging. Later work published in parallel explores the growth of this state space in more detail [5].

V. RELATED WORK

The autonomic adaptation of software systems is studied extensively. Approaches to run-time adaptation vary in their approach, including explicit models of the system's performance [6], [7] like our approach, linear programming formulations [8], [9], and hybrid approaches [10]. Our work distinguishes itself from other approaches in several ways. A popular web services autonomic adaptation is to substitute one service for another equivalent service. Our work is service provider focused: that is, the desire is to ensure compliant performance for the services a single provider offers, ruling out substitution. Much work in autonomic adaptation is at the server level, monitoring load averages and memory. This approach is generally applicable to many applications, but the simulation creation and the knowledge base once created are done at the application level, rather than with system infrastructure. We create our knowledge base entirely in simulation. We deliberately chose a model that can be visualized and used to explain why the autonomic manager made a certain decision, in contrast to black-box approaches.

Calinescu and Kwiatkowska [6] use a Markov model of a software application to generate an autonomic manager for that application, using the PRISM probabilistic model checker as an engine. The Markov model is created during a formal-verification process of the application (or later following the same process). Decisions are chosen by the engine based on the maximizing of a utility function. They describe empirical results for Markov chains with a small number of states, for two scenarios. It is not clear how accurately the deterministic model reflects reality, or how well this method translates to a real system at run time.

Bahati *et al.* [11] use a state-transition model to encode past decisions and their results. A state consists of metrics and the various transitions already tried when in this state. The dividing line between states is the threshold of any metric (if the valid response time threshold is 2 seconds, there are two states: one for response time over 2 seconds, and one for under. A state-transition graph is built over time as the deployed system is manually transitioned from state-to-state. The autonomic manager identifies the state of the system and the transitions that could potentially move the system to an acceptable state. Our methodology builds a behavioral model offline (skipping the expensive learning stage) and offers variable granularity for each metric which allows more fine-tuned control, rather than a simple binary compliant/non-compliant discretization. Their approach does not address the problem of over-provisioned systems.

VI. Conclusion

In this paper we have described our simulation-based method for autonomic reconfiguration of service-oriented systems. Our solution relies on a systematic-simulation method (at pre-deployment time) that develops a state-transition model of the behavior of the system, in terms of its configuration and performance. This model essentially captures how the system's performance is impacted by the changes in the request load that the environment imposes to the system and changes to its configuration. At run time, the system's actual behavior is tracked against this model and when the autonomic manager finds the system in a state that may lead to possible futures that violate service level agreements, its configuration is changed to move the system to a safer future state. Through our case study with TAPoRware, we demonstrated the effectiveness of the process: the autonomic manager achieved results comparable to manual changes by an expert, though not quite at the same level. Given the impracticality of expert non-stop monitoring of an application, our autonomic approach has value.

In addition to the benefits provided by any self-adaptation system, our novel methodology and implementation offers several key benefits.

1) It is based on simulation. Assuming an accurate simulation, we know ahead of time what results our changes will produce without trial-and-error lag time which can be expensive if an SLA is being violated. Simulation data is less accurate than actual run-time data, but a greater volume of data can be acquired at a lower expense.
2) The majority of the "training" of this system occurs offline prior to deployment. There is no learning phase where the system is not useful while it monitors the behavior of a live application. At the same time, the system is still capable of learning at run-time: though it starts with a model built on simulation-generated data, the model grows and "learns" from manual changes or previously unknown states.
3) It is explainable; a state model can be easily displayed showing what state the system was believed to be in, and what remedial action was chosen a a result. Using the dashboard that is part of the simulation framework, a system manager can review simulated "what if" scenarios. This understanding is intended to help resolve issues that administrators may have with allowing an application to make its own decisions.
4) It makes decisions based on the totality of available metrics, rather than using simple thresholds based on single metrics.

The major challenge in our method is it relies on anticipating situations ahead of time, which implies the need for having already seen simulations of these situations. Though we have tools to automatically explore the configuration space, we intend to analytically study the coverage enabled by our method and to work on providing confidence metrics.

Acknowledgments

The authors would like to acknowledge the generous support of iCORE, NSERC, and IBM. Special thanks to Hyeon Rok Lee at the University of Alberta.

References

[1] E. D. Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Automated Software Engineering*.

[2] IBM, "An architectural blueprint for autonomic computing".

[3] M. Smit, A. Nisbet, E. Stroulia, A. Edgar, G. Iszlai, and M. Litoiu, "Capacity planning for service-oriented architectures," in *CASCON '08: Proceedings of the 2008 conference of the Center for Advanced Studies on collaborative research*. New York, NY, USA: ACM, 2008, pp. 144–156.

[4] M. Smit, A. Nisbet, E. Stroulia, G. Iszlai, and A. Edgar, "Toward a simulation-generated knowledge base of service performance," *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, Nov 2009.

[5] M. Smit and E. Stroulia, "Automated state-space exploration for configuration management of service-oriented applications," in *Web Services (ICWS), 2011 IEEE International Conference on*, 2011, p. To Appear.

[6] R. Calinescu and M. Kwiatkowska, "Using quantitative analysis to implement autonomic IT systems," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 100–110.

[7] M. Litoiu, M. Woodside, and T. Zheng, "Hierarchical model-based autonomic control of software systems," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–7, May 2005.

[8] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola, "Qos-driven runtime adaptation of service oriented architectures," in *Proceedings of ESEC and the ACM SIGSOFT FSE*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 131–140.

[9] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *Software Engineering, IEEE Transactions on*, vol. 33, no. 6, pp. 369 –384, Jun. 2007.

[10] M. Litoiu, M. Mihaescu, D. Ionescu, and B. Solomon, "Scalable adaptive web services," in *Proceedings of the 2nd international workshop on Systems development in SOA environments*, ser. SDSOA '08. New York, NY, USA: ACM, 2008, pp. 47–52.

[11] R. M. Bahati, M. A. Bauer, and E. M. Vieira, "Adaptation strategies in policy-driven autonomic management," in *ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems*. Washington, DC, USA: IEEE Computer Society, 2007, p. 16.