Model-based Performance Self-adaptation: A Tutorial

Emilio Incerto
emilio.incerto@imtlucca.it
IMT School for Advanced Studies
Lucca, Italy

Mirco Tribastone
mirco.tribastone@imtlucca.it
IMT School for Advanced Studies
Lucca, Italy

ABSTRACT

This tutorial presents techniques for self-adaptive software systems that use performance models in order to achieve desired quality-of-service objectives. Main hindrances with the state of the art are the assumption of a steady-state regime to be able to use analytical solutions and the explosion of the state space which occurs when modeling software systems with stochastic processes such as Markov chains. This makes their online use difficult because the system under consideration may be in a transient regime, and the typically large cost of the analysis does not permit fast tracking of performance dynamics. We will introduce fluid models based on nonlinear ordinary differential equations as a key enabling technique to effectively approximate large-scale stochastic processes. This representation makes it possible to employ online optimization methods based on model-predictive control in order to find an assignment of the values of tunable parameters of the model steering the system toward a given performance goal. We will also show how, dually, the same techniques can be used for the online estimation of software service demands. In this tutorial we will focus on software performance models based on queuing networks, with applications to runtime auto-scaling in virtualized environments.

KEYWORDS

Adaptive software, Control-theory, Model predictive control, Resource Sharing, Performance Engineering

ACM Reference Format:
Emilio Incerto and Mirco Tribastone. 2019. Model-based Performance Self-adaptation: A Tutorial. In *Tenth ACM/SPEC International Conference on Performance Engineering Companion (ICPE '19 Companion), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3302541.3310293

1 INTRODUCTION

Performance is fundamental aspect of modern software systems design since it strongly affects user satisfaction. Indeed, it is widely recognized by researchers and practitioners alike that performance should be seen as a new facet of software correctness [14].

Performance-driven self-adaptation consists in the idea of managing runtime variability (due events such as workload fluctuation or hardware degradation) by continuously monitoring a software system and, when appropriate, by triggering suitable reconfigurations. Indeed, due to the uncertainties in the execution environment, an initially optimal configuration with respect to some given quality-of-service (QoS) criterion such as throughput or mean time to failure may suddenly become unacceptable, thus steering the system toward a significantly different operating point.

One possible way of addressing this issue is to *react* to an operating condition that is deemed critical for the system's performance. For instance, the early auto-scaling mechanism in Amazon's cloud platform allows the operator to define policies for scaling up or scaling down the number of servers based on certain observed metrics.[1] Clearly, such a reactive approach may introduce delays in the effectiveness of the policy—one would rather prefer a *proactive* method that anticipates potential drops in the QoS.

At the core of any proactive approach is the availability of a model that can predict the system's future behavior based on its current state as well as under the application of potential reconfigurations. This introduces three main difficulties:

i) One must be able to analyze the model efficiently in order to ensure a fast execution of the adaptation cycle.
ii) One must devise an effective strategy for exhaustively exploring the adaptation space (AS), i.e., the set of all feasible system configurations. Indeed, it has been recognized that the typically huge size of the AS represents a major limitations for state-of-the-art approaches to real-world scenarios [10].
iii) One needs frequent up-to-date estimates of the model parameters, to be provided in a minimally intrusive manner so as not to affect the behavior of the running system significantly [4].

This proposal presents performance-driven self-adaptation techniques for software systems using analytical models.

2 QUEUING NETWORKS

Queuing networks (QNs) are a class of well established analytic models for software performance engineering [9, 11]. Their key idea is to model customers being routed between different service stations, where they compete for a pool of processing resources. Among the benefits of QNs is the fairly immediate association between their constituent elements and the components of software systems (e.g., a service center may be used either for abstracting a software device or a hardware resource like CPU or disk). Typically, designers employ such model for studying the performance of systems at design time in a "what-if" analysis fashion (i.e., combining different workload and queuing parameters) so as to identify the most suitable hardware/software configuration satisfying requirements under worst-case or average execution conditions [16].

ICPE '19 Companion, April 7–11, 2019, Mumbai, India
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6286-3/19/04…$15.00
https://doi.org/10.1145/3302541.3310293

[1] https://aws.amazon.com/autoscaling/

A major limitation in the use of QN models at runtime for self-adaptation purposes is the lack of analytical results for the evaluation of performance metrics in the transient regime. Indeed most of the literature focuses on steady-state measures only, for which a large body of results exists [6]. But the very assumption of modifying the system parameters at runtime inevitably introduces a transient period during which these results do not necessarily hold.

In general, an exact transient formulation of a QN involves a system of linear ordinary differential equations (ODEs), whose size grows exponentially with the number of jobs and stations. This is a prohibitive cost—the well-known state space explosion problem—that blocks any runtime use of such description.

The key enabling mathematical tool to still leverage an analytical representation of the transient dynamics, albeit approximately, is in the use of *fluid* models. We refer to [8] for a thorough review about this topic. Here we provide the intuition behind the fluid model for a single-class QN, where an approximate estimate of the average queue length at each station i, denoted by x_i, is governed by a coupled system of non-linear ODE in the form:

$$\frac{dx_i(t)}{dt} = -\mu_i(t)\min\{x_i(t), s_i(t)\} + \sum_{j\in S} p_{j,i}(t)\mu_j(t)\min\{x_j(t), s_j(t)\}$$

The quantity $\mu_i(t)\min\{x_i(t), s_i(t)\}$ is the *instantaneous throughput* at each station: when the queue length $x_i(t)$ in station i is less than the available servers $s_i(t)$, then the $x_i(t)$ jobs are served in parallel, each with rate $\mu_i(t)$; otherwise some of the jobs are enqueued and only $s_i(t)$ of them are processed at once. Throughputs are weighted by the routing probabilities $p_{j,i}(t)$ because a station may receive only a fraction of the jobs completed elsewhere.

Such approximation has been applied to software performance models described by stochastic process algebra [15, 30] and layered queuing networks [28]. Although it directly provides queue-length estimates only, it is also possible to efficiently estimate other important performance metrics such as throughput, utilization, and response times as appropriate functions of the ODE solution [29].

In the context of self-adaptation, the main benefit of fluid models of QNs is that the resulting ODE system is compact since it requires one equation for each station. Furthermore, a result of asymptotic convergence formally support the abundant empirical evidence of the high quality of the approximation when the size of the QN (intended as the population of jobs and servers) is large enough [8]. Another important consequence is that we have turned the description of a large-scale stochastic process into a deterministic dynamical system: this paves the way for the applicability of several methods from control theory for model learning and adaptation.

3 QN SELF-ADAPTATION FRAMEWORK

Around QN fluid models we build our proposal of performance-based self-adaptation by putting it in the context of the well-known MAPE-K control loop [3]. Its reference architecture considers the monitoring of the behavior of a self-adaptive system to extract the actual runtime parameters (*monitoring* phase). Parameters are used to calibrate a model that allows an accurate analysis of the current execution conditions at runtime (*analysis* phase). If problems are detected, a set of refactoring actions is computed (*planning* phase) that, once reflected on the running system, steer it in a problem-free state (*execution* phase). Finally, *knowledge* is used as the shared repository (among all the different phases) to store all the information representing the awareness of the system about itself. An overview of our proposed approach is depicted in Figure 1 in which we can identify the following components.

- **Monitoring:** we monitor the running software system using a system profiler tool, similarly to [31], with which we periodically extract a set of parameters. More precisely, for each service center i we wish to collect its service rates μ_i and its estimated queue length x_i. In addiction, in order to completely specify the structure of the monitored system, we extract the routing matrix P that will be used for representing the topology of the QN model. Finally, in the case of closed networks, we estimate the number of customers in the system N and the thinking time Z. In the case of open networks, we identify the jobs arrival rate λ. Then we use these parameters to feed the runtime QN model built in order to describe the performance evolution of a running system.
- **Analysis:** the QN model is suitably translated into a program that executes to predict the transient behavior of the running system. The result of this step is the set of performance metrics of interest based on the model prediction made with the previously estimated parameters. Now, giving the computed indices and the performance constraints model as input to the *constraint analysis engine* we detect the performance violations. We envisage this last component as a modeling and analysis tool applicable for checking the QoS requirements against the performance evaluation of each specific system configuration (e.g., a computer-algebra system [22, 23] or an SMT solver [19]).
- **Planning:** starting from the previously computed *performance violations*, an *adaptation engine* component is able to compute a set of refactoring actions that represent the plan for the current adaptation iteration. Concrete examples of such actions can be the addition or removal of a service center (e.g., virtual machines), the modification of the service rates (e.g., assigning different CPU sharing quota) or the change of the load balancing policy in a distributed system.
- **Execution:** the adaptation mechanism is actually implemented by executing the previously defined *adaptation actions* on the running system.
- **Estimation:** goal of this phase is to use the structural information about the system and the monitored quantities for estimating the relevant QN parameters that cannot be directly measured, such as service demands. Then the estimated information is used for feeding the QN model used for predicting the performance dynamics of the systems.

The control loop is repeated at runtime for each system evolution step so to reduce the influence of uncertainty at each adaptation. In the remainder of this paper we overview techniques for planning and estimation based on QNs.

4 MODEL PREDICTIVE CONTROL FOR QNS

The fluid QN model can be interpreted as a set of constraints (in continuous-time) that have to be satisfied by the system's dynamics.

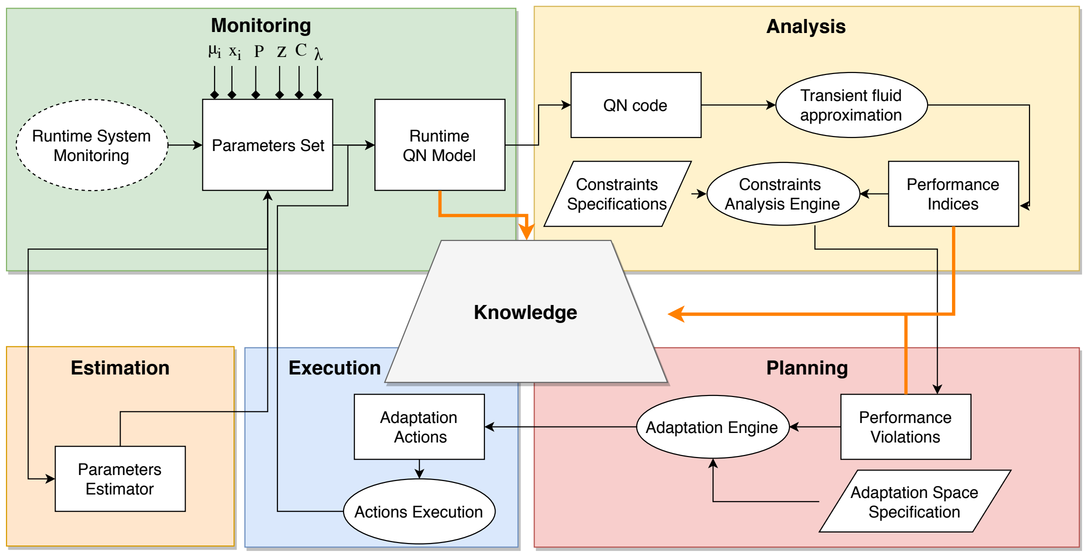

Figure 1: Performance self-adaptation framework based on queuing networks.

By appropriately discretizing time, it can then be turned into a finite set of constraints over a given time horizon of H steps. Indeed, simple algebraic manipulations yield a formulation that reads:

$$x(k+1) = A(x(k)) + x(k), \qquad k = 0, \ldots, H$$

where $x(k)$ is vector such that $x_i(k)$ is the queue lengths of station i at step k and matrix $A(x(k))$ has components $a_{i,j}(x(k))$ given by

$$a_{i,j}(x(k)) = \begin{cases} -\mu_i(k)\Delta t \min\{x_i(k), s_i(k)\} & ,i = j \\ p_{j,i}(k)\mu_j(k)\Delta t \min\{x_j(k), s_j(k)\} & ,i \neq j. \end{cases}$$

These manipulations enable the formulation of the performance-driven self-adaptation problem as model predictive control, a well-known technique based on on-line numerical optimization [13]. Here, the discretized equations represent the constraints of an optimization problem where the objective function encodes a target performance metric and the decision variables are the parameters of the system that can act as *control knobs* such as, for example, the service rate $\mu_i(k)$ in case of a hardware architecture that allows dynamic voltage scaling [25]. The basic idea behind MPC is to perform an optimization at each time step during the system evolution. The model is initialized with the currently measured state of the system. The optimization finds the optimal values of such knobs that steer the system toward the desired QoS requirement over the time horizon H [20]. Thus MPC returns an optimal value for each control signal at each time step across that horizon. Adaptation takes place according to the *receding horizon control* paradigm: only the values for the next time step are applied, whereas all subsequent ones are discarded [1]. When MPC is started at the next iteration the newly measured state will readily feedback the effect of the adaptation into the system, and become the starting point for the optimization over the next time horizon.

Unfortunately, applying MPC is not straightforward. Its main limitation is the typically high computational cost. Indeed, state-of-the-art approaches report significant overheads even for small models and short prediction horizons [1, 32]. For QNs, the main sources of complexity are: *i*) the nonlinearity of the ODE model, as it needs to account for threshold-type service rates that depend on the state of the system; *ii*) the exponential complexity of the optimization due to the multi-dimensionality of the AS. As a consequence, the solution of a single optimization problem can be time-consuming, reducing the maximum frequency at which the controller can operate and therefore its effectiveness.

In this tutorial we present a novel formulation which formally translates the original nonlinear MPC problem for a single-class QN into an equivalent mixed integer linear programming (MILP) one, leading to a quadratic programming problem that enjoys very efficient solution techniques [2].

The MPC approach can be extended to multi-class QN, a relevant model for the prediction and control of multiple applications (e.g., multiple virtual machines) sharing the same execution environment (i.e., the same CPUs or disks) [21]. In particular, in this tutorial we will focus on CPU-bound systems on a virtualized environment. We consider control knobs that can realize *vertical* as well as *horizontal* scaling; the former varies the sharing quota on each individual machine, while the latter determines the number of virtual machines actually employed [26].

We first present a multi-class QN model for the *capped allocation* paradigm [7]. This is a CPU-sharing mechanism available in most modern off-the-shelf hypervisors (e.g., [5, 24]), which defines the maximum share that a VM can receive from the CPU. Starting this model, we formulate an MPC problem which can be efficiently solved by linear programming, in a surprisingly simpler way than the mixed-integer optimization of the single-class QN model.

5 PARAMETER ESTIMATION

For simplicity, until now we have implicitly assumed that all system parameters could be directly measurable in the monitoring phase. In reality, certain parameters are inaccessible because it would be expensive and/or invasive to obtain them. In this tutorial we also present a parameter estimation approach, which has the main objective of obtaining model parameters in a minimally intrusive manner and without assuming that the system under control is in a steady-state regime [17]—a necessary condition for the applicability of any of the parameter-estimation techniques reviewed in [27].

We focus on the estimation of service demands for single-class QNs with exponentially distributed service times and load-dependent (i.e., multiple-server) service rates, using measurements of queue lengths only. The key to achieving this is to interpret the previously discussed MPC problem in a dual manner: the decision variables are the service demands to be estimated and the objective function to be minimized is the error between the predicted queue lengths and the measured ones across the whole observation horizon H. The *moving horizon* strategy that shifts forward the time window at each step allows one to obtain continuously updated estimates of the parameters.

6 CONCLUSION

This tutorial has the objective to propose a model-based approach for the performance-driven adaption of software system, presenting recent work by the authors and colleagues (i.e., [17–21]) in a unified manner and in the broader context of self-adaptive systems.

The tutorial will also touch on a number of open questions. To cite a few, a current limitation is the "flat" nature of the QN models, which does not cover mechanisms such as simultaneous resource possession and layered services (e.g., multi-tiered applications when one tier needs processing from tiers below in order to complete its job). This calls for more expressive MPC formulations based for layered queuing networks [12]. Another key issue is that the fluid QN model provides first-order (i.e., mean value) approximations only. This results in the impossibility to define adaptation strategies driven by distributional requirements, such as response-time quantiles. Finally, the service-demand estimation needs to be extended for generally distributed service demands and multi-class networks. We hope that the tutorial can stimulate further research in this challenging area.

REFERENCES

[1] S. Abdelwahed, J. Bai, R. Su, and N. Kandasamy. 2009. On the application of predictive control techniques for adaptive performance management of computing systems. *IEEE Transactions on Network and Service Management* 6, 4 (2009), 212–225.

[2] Tobias Achterberg and Roland Wunderling. 2013. Mixed integer programming: Analyzing 12 years of progress. In *Facets of Combinatorial Optimization*. 449–481.

[3] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. 2015. Modeling and analyzing MAPE-K feedback loops for self-adaptation. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 13–23.

[4] Mahmoud Awad and Daniel A. Menasce. 2017. Deriving Parameters for Open and Closed QN Models of Operational Systems Through Black Box Optimization. In *Proceedings of the International Conference on Performance Engineering (ICPE)*.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, Vol. 37. 164–177.

[6] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor Trivedi. 2005. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley.

[7] Ethan Bolker and Yiping Ding. 2000. On the performance impact of fair share scheduling. In *International CMG Conference*. 71–82.

[8] Luca Bortolussi, Jane Hillston, Diego Latella, and Mieke Massink. 2013. Continuous approximation of collective system behaviour: A tutorial. *Performance Evaluation* 70, 5 (2013), 317–349.

[9] Vittorio Cortellessa and Raffaela Mirandola. 2000. Deriving a queueing network based performance model from UML diagrams. In *International Workshop on Software and Performance*. 58–70.

[10] Rogério De Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, and others. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. 1–32.

[11] Antinisca Di Marco and Paola Inverardi. 2004. Compositional generation of software architecture performance QN models. In *Working IEEE/IFIP Conference on Software Architectures*. 37–46.

[12] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. 2009. Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Transactions on Software Engineering* 35, 2 (2009), 148–161. https://doi.org/10.1109/TSE.2008.74

[13] Carlos E. García, David M. Prett, and Manfred Morari. 1989. Model predictive control: Theory and practice—A survey. *Automatica* 25, 3 (1989), 335–348.

[14] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*.

[15] Jane Hillston, Mirco Tribastone, and Stephen Gilmore. 2012. Stochastic Process Algebras: From Individuals to Populations. *Comput. J.* 55, 7 (2012), 866–881.

[16] Yu Chi Ho and Xiren Cao. 1983. Perturbation analysis and optimization of queueing networks. *Journal of Optimization Theory and Applications* 40, 4 (1983), 559–582.

[17] Emilio Incerto, Annalisa Napolitano, and Mirco Tribastone. 2018. Moving Horizon Estimation of Service Demands in Queuing Networks. In *International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[18] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. 2015. A proactive approach for runtime self-adaptation based on queueing network fluid analysis. In *International Workshop on Quality-Aware DevOps, (QUDOS)*. 19–24.

[19] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. 2016. Symbolic Performance Adaptation. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 140–150.

[20] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. 2017. Software performance self-adaptation through efficient model predictive control. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 485–496.

[21] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. 2018. Combined Vertical and Horizontal Autoscaling Through Model Predictive Control. In *European Conference on Parallel Processing*. Springer, 147–159.

[22] Matthias Kowal, Ina Schaefer, and Mirco Tribastone. 2014. Family-Based Performance Analysis of Variant-Rich Software Systems. In *Fundamental Approaches to Software Engineering (FASE)*. 94–108.

[23] Matthias Kowal, Max Tschaikowski, Mirco Tribastone, and Ina Schaefer. 2015. Scaling size and parameter spaces in variability-aware software performance models. In *International Conference on Automated Software Engineering (ASE)*. 407–417.

[24] Parallels. 2016. OpenVz User Guide. https://docs.openvz.org/openvz_users_guide.webhelp/

[25] Padmanabhan Pillai and Kang G Shin. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM SIGOPS Operating Systems Review*, Vol. 35. 89–102.

[26] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. 2017. Auto-scaling web applications in clouds: A taxonomy and survey. *Comput. Surveys* 9, 4 (2017).

[27] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. 2015. Evaluating approaches to resource demand estimation. *Performance Evaluation* 92 (2015), 51–71.

[28] Mirco Tribastone. 2013. A Fluid Model for Layered Queueing Networks. *IEEE Trans. Software Eng.* 39 (2013), 744–756.

[29] Mirco Tribastone, Jie Ding, Stephen Gilmore, and Jane Hillston. 2012. Fluid rewards for a stochastic process algebra. *IEEE Transactions on Software Engineering* 38, 4 (2012), 861–874.

[30] Mirco Tribastone, Stephen Gilmore, and Jane Hillston. 2012. Scalable Differential Analysis of Process Algebra Models. *IEEE Transactions on Software Engineering* 38, 1 (2012), 205–219.

[31] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM, 247–248.

[32] Xiaoqiang Wang, Vladimir Mahalec, and Feng Qian. 2017. Globally optimal nonlinear model predictive control based on multi-parametric disaggregation. *Journal of Process Control* 52 (2017), 1–13.