A Library of Modeling Components for Adaptive Queuing Networks

Davide Arcelli[1], Vittorio Cortellessa[1], and Alberto Leva[2]

[1] Università dell'Aquila, L'Aquila, Italy
[2] Politecnico di Milano, Milano, Italy

Abstract. Self-adaptive techniques have been introduced in the last few years to tackle the growing complexity of software/hardware systems, where a significant complexity factor leans on their dynamic nature that is subject to sudden (and sometime unpredictable) changes. Adaptation actions are aimed at satisfying system goals that are often related to non-functional properties such as performance, reliability, etc. In principle, an adaptable software/hardware system can be considered a controllable plant and, in fact, quite promising results have been recently obtained by applying control theory to adaptation problems in this domain.
Goal of this paper is to provide a design support for introducing adaptation mechanisms in Queuing Network models of software/hardware systems. For this goal, we present a consolidated library of modeling components (in Modelica) representing Queuing Network elements with adaptable parameters. Adaptive Queuing Networks (AQN) can be built by properly assembling such elements. Once feedback control loop(s) are plugged into AQNs, it is possible to analyze and control (before the implementation) the system performance under changes due to external disturbances.
We show the construction of an AQN example model by using our library, and we demonstrate the effectiveness of our approach through experimental results provided by the simulation of a controlled AQN.

Keywords: Performance Modeling, Control Theory, Adaptive Systems

1 Introduction

The growing complexity of computing systems is placing increased burden on application developers. This situation is worsened by the dynamic nature of modern systems, which can experience sudden and unpredictable changes (e.g., workload fluctuations and software component failure). It is increasingly up to software/system engineers to manage this complexity and ensure applications operate successfully in dynamic environments [18]. The use of self-adaptive techniques has been proposed to help engineers in managing this burden. Adaptable systems modify their own behavior to maintain goals in response to unpredicted changes. While adaptation of functional aspects (i.e., semantic correctness) often requires human intervention, non-functional ones (e.g., performance) represent

a challenging opportunity for applying self-adaptive techniques [3]. For example, customers may require continuous assurance of agreed quality levels, which usually map to specific metrics used to trigger adaptations for guaranteeing requirements are met even in the face of unforeseen environmental fluctuations [6]. Such adaptations have been studied for decades in the context of control theory, where control systems have achieved widespread usage in many engineering domains that interact with the physical world [15]. In such domains, a controller measures quantitative feedback from a sensor (e.g., a speedometer) and determines how to tune an actuator (e.g., a fuel intake) to effect the controlled plant behavior (e.g., an engine). One major advantage of using control theory is that such techniques emit analytical guarantees of the system dynamic behavior [21].

In the last few years, control theory has been applied to build adaptable software/hardware systems. Such systems can be in fact considered as controllable *plants* fitting in a basic feedback control loop scheme [15]. The target of a feedback control loop is not necessarily a running system. In fact, control theory is often applied to design system models [13], with the goal of studying (before the implementation) if their dynamic behavior can be controlled/adapted under changes (namely external disturbances), while satisfying non-functional requirements. Here we focus on performance requirements, with the goal of providing support to the design of Queuing Network (QN) models with plugged feedback control loops, for the purposes discussed above. In [4] we presented a first attempt in this direction, where few typical elements of QNs were modeled in Modelica [14], with the purpose of combining them with controllers. QNs assembled on top of these elements have: (i) performance indices, such as response time, which can be observed (as sensed variables), (ii) parameters, such as CPU shares among classes of jobs, that can be modified (as actuators), and (iii) environmental parameters, such as workload and operational profile, which can be modeled as external disturbances whose changes may induce QN adaptations. The first experimental results that we have reported in [4] were promising.

In this paper we consolidate our work by extending the library of Modelica components to represent a large variety of QN elements. We show that such an extended library enables to build a whole class of open QN models that we name here as Adaptive Queuing Networks (AQN). We also show how combining an AQN with controllers allows to investigate the performance of a system under external disturbances and to identify appropriate control laws that have to be implemented in order to guarantee the performance requirements satisfaction through adaptation actions. This paper is organized as follows: in Section 2 we discuss related work, Sections 3 and 4 describe the paper contribution, respectively for the AQN modeling library and for the control modeling, Section 5 presents simulation results and Section 6 concludes the paper.

2 Related Work

Adaptation is becoming a key concern in software applications [18]. An adaptive application must select, from many configurations, the one that is mostly appro-

priate to satisfy specific requirements. There are many examples, from hardware to software development. The evaluation of a new microprocessor design requires studying the impact of input data sets and workload composition [12]. Compiler-level advancements have been developed to support adaptive implementations for performance [2] or power [5]. In [10], a study on tuning Fast Fourier Transformations on graphic processing units is presented, whereas Rahman et al. [26] studied the effect of compiler parameters on performance and power consumption for scientific computing.

Besides, control theory [15] is capturing an increasing interest from the software engineering community that looks at self-management as a mean to meet Quality of Service (QoS) requirements despite unpredictable changes of the execution environment [23]. Examples of this trend can be seen in research on control of web servers [22], data centers and clusters management [11], operating systems [8], and across the system stack [17].

The application of control theory in software engineering, however, is still in a very preliminary stage. Developing accurate system models for software is in fact hard, mostly due to the strong mathematical skills needed for dealing with complex non-linear dynamics of real systems [9]. These difficulties usually lead to the design of controllers focused on particular operating regions or conditions and ad-hoc solutions that address a specific computing problem using control theory, but do not generalize. For example, in [16] the specific problem of building a controller for a .NET thread pool is addressed. More in general, the approach presented herein has the peculiarity of not just closing control loops *around* an existing system, or in other terms, adding a control layer *on top* of a fully functional one. Controllers are here part of the system itself, thus in fact enabling elements to provide the required functionality. The interested reader can find in [20, Chapter 1] a discussion on the benefits (and for completeness, the new design challenges) that such an approach brings into the *arena*.

All the related work discussed up to this point aims at controlling running adaptable applications. In this paper, we raise the level of abstraction, in that our contribution concerns *model-based* performance control of adaptive software. In this domain, some effort has been spent to raise adaptation techniques driven by performance (or more general QoS) requirements at the software architecture level [24], where adaptive verification techniques have been also studied [6]. Adaptation approaches for specific architectural paradigms have been introduced, such as Service-Oriented-Architecture [7]. An interesting work has been recently introduced in [27] for automatically extracting adaptive performance models from running applications. However, none of these papers applies control theory to the control of performance models, like we do in this paper.

3 AQN Modeling Library

The first step to define a control methodology for QN models is the provisioning of a suitable mean for their formalization as dynamic systems. To this end, we

have extended and consolidated a Modelica library for our purposes, i.e. for enabling the design of AQNs.

Modelica [14] is a modeling and simulation environment widely used by control practitioners to define and study dynamic models of physical phenomena and engineered systems, and to support the design and synthesis of suitable controllers. Our library includes at today several basic QN types, including queues, service centers, routing nodes, workload generators (both deterministic and probabilistic), and it also supports multiple job classes. Instances of these types can be created and connected together to seamlessly draw a QN model. The definition of each type includes peculiar inputs/outputs and state equations. The former determine the interaction with the other components (e.g., input/output rates, control input), while the latter represent a (parametric) dynamic model specified as a system of differential equations that capture the time behavior of the component instance, according to its initial state and the input it receives.

When a QN element is instantiated, the designer needs to set some parameters (e.g., the service rate of a server) and the connection to other components (e.g., reflecting the system control flow). Our library has been conceived to be an instrument that can be extended with other features/types and possibly ported to different contexts. This goal is achieved by exploiting the fact that Modelica is an object-oriented framework with constructs for type hierarchies and inheritance. These mechanisms can be used to extend our library either by adding other QN types (e.g., passive resources) or by extending existing types with additional features (e.g., service centers with special scheduling policies).

We describe in the following all the AQN constructs in the current version of our library that, with respect to the previous one in [4], introduces the following novel aspects: (i) multiple classes of jobs for all elements, (ii) definition of required level of services for each class, (iii) multi-servers for M/M/c models, (iv) ready-to-use predefined stations, (v) more complex split nodes and (vi) corresponding merge nodes.[3]

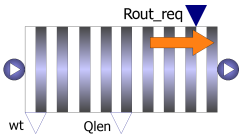


Fig. 1. Station queue.

Station Queue. This element represents a controllable queue of a server, and its state is defined, as usual, by the number of enqueued jobs. The queue capacity is infinite, hence it is possible to push as many jobs as desired. It is also possible to pop an arbitrary number of requests, less or equal than those enqueued, due to the server scheduling mechanism (illustrated later). The graphical representation of a controllable queue, showing its input/output parameters is in Figure 1. The orange arrow indicates the job flow direction. The main input/output parameters, used for controlling purposes, are explicitly shown. In particular, `wt` and `Qlen` represent output "pins" where a controller can be plugged to, in order to observe their current values, i.e., the waiting time per job class and the number of jobs in the queue, respectively. `Rout_req`, instead, represents an input "knob" of the

[3] Note that each parameter described in this section refers to a specific class of jobs, where not differently specified.

queue (i.e., a controlled variable), where again a controller can be plugged to, in order to control the queue length (or the waiting time) with respect to its target value, by actuating on the rate at which jobs are extracted from the queue.

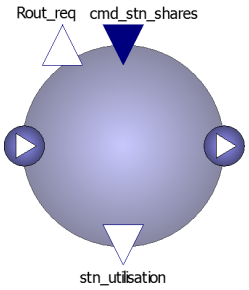


Fig. 2. Station server.

Station Server. This element represents a cluster of S controllable service centers that can process incoming requests at different predefined processing rates for each class of jobs. The graphical representation of a controllable server, showing its input/output parameters, is in Figure 2, where again jobs flow from left to right. A controllable server has three main parameters: `Rout_req` and `stn_utilisation` represent output "pins" which a controller can be plugged to, in order to observe their current values, i.e., its throughput and its utilization, respectively; the input knob `cmd_stn_shares` can regulate the shares that the server allocates to each class of jobs. We have devised, up today, a single scheduling policy for servers, that is a Generalized Processor Sharing (GPS) one [1]. In particular, we envisage that a (non-uniformly distributed) share of processing is held by each class of jobs, where different classes of jobs may have different resource demands. This is a general modeling approach to represent the case of a service center processing jobs by the same "functional" type (e.g. jobs that represent the same software function/operation) that can be partitioned in classes, where each class provides a different quality level. From a software viewpoint, this is a scenario where different adaptations are available for a certain operation, and each adaptation requires a different amount of resources. The processing shares among classes of jobs may be controlled by actuating on the `cmd_stn_shares` knob. In particular, such a knob corresponds to a $K \times S$ matrix, namely CMD, where K is the number of job classes. Each element (i, j) of that matrix, with $i = 1..S, j = 1..K$, is in the interval $[0, 1]$ and represents the fraction of the share of the i-th station server in the cluster that is assigned to the j-th job class. To this aim, we have also defined a "pivotal" $K \times S$ matrix, namely MAX_SERV_RATES, where each entry (i, j), with $i = 1..S, j = 1..K$, is a real number greater or equal to zero and represents the *maximum computational effort*, i.e. the maximum rate of the i-th station server in the cluster for the j-th job class. This means that, for example, if the i-th station server has a maximum computational effort of 20 and the fraction of its share for that class is 0.5, then the actual service time of the i-th station server for the j-th job class is 10 jobs per second.

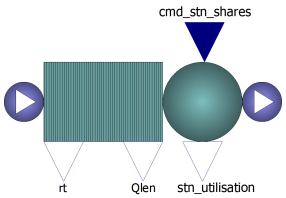
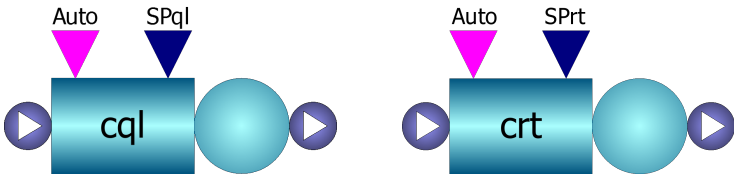


Fig. 3. Station.

Station. This element puts `StationQueue` and `StationServer` together, thus composing a complete service center [19], as illustrated in Figure 3. A station has three output and one input parameters. The `Qlen` output parameter is inherited from the StationQueue element, whereas `stn_utilisation` from StationServer. The `rt` output parameter, instead, represents the current

residence time for a job in the whole station, as defined in standard QN theory as the sum of waiting time in the queue plus the service time in the server. The input "knob" is inherited from the Station Server.[4]

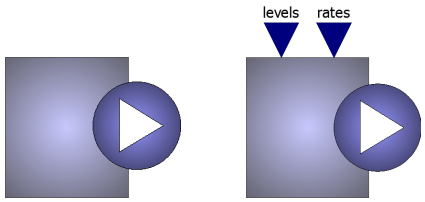
On the basis of the `Station` element, we have built two types of "ready-to-use" stations, that are shown in Figure 4. In particular, the one in Figure 4(a) allows to control the queue length through its `SPql` knob, whereas the residence time is controlled through the `SPrt` knob of the station in Figure 4(b). Both these `Station`s are equipped with an additional input, named `Auto`, which represents a boolean value that can be manipulated to switch on/off over time the control of these elements, namely "automatic mode".



(a) Controlled queue length. (b) Controlled residence time.

Fig. 4. Predefined `Station`s.

Workload Sources. JobSource element represents an open workload source with two internal parameters corresponding to: (i) the rate at which jobs of each class are generated conforming to a probabilistic distribution (possibly varying over time), and (ii) the service levels requested by each class of jobs (in case one would like to assume that each job class may require a specific service level). Figure 5(a) shows its graphical representation. JobSourceVar, in Figure 5(b), represents instead a controllable variant of JobSource, where two additional input parameters are introduced, i.e. `rates` and `levels`, representing input knobs for controlling the two above mentioned internal parameters, respectively.



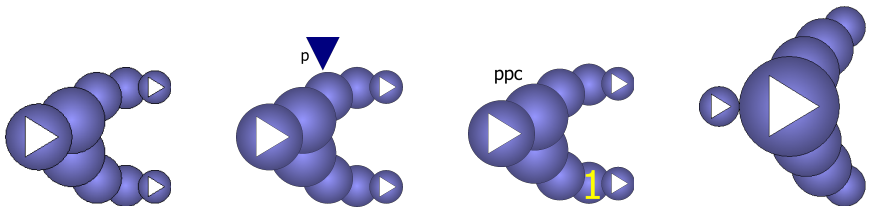
(a) JobSource. (b) JobSource-Var.

Fig. 5. Job sources.

Job Split Nodes. As represented in Figure 6, these elements are stochastic branchings of jobs, where internally-defined distribution functions regulate the

[4] Note that in a Station the `Rout_req` input of the StationQueue element has been joined to the homonym output of the StationServer to implement the policy of job extraction from the queue, so they disappear from the figure.

probability that a job is routed along one of the outgoing paths. In particular: JobsSplit2 (Figure 6(a)) and JobsSplit2Var (Figure 6(b)) have two outgoing paths and an internal parameter, common to all job classes, corresponding to the probability that a job is routed along the first path (i.e., the one on the top). In addition, JobsSplit2Var exposes the `p` input parameter for externally setting such probability. JobsSplit2_probPerClass (Figure 6(c)) is similar to JobsSplit2, but it allows to specify a vector of probabilities (namely `ppc`), i.e. one for each job class. Finally, JobsSplitN (Figure 6(d)) represents a generalization of JobsSplit2, i.e. a N-way job splitting; hence, it allows to specify the number N of outgoing paths, together with a vector of N-1 routing probabilities $p_1..p_{N-1}$, as the N-th one can be obtained by complementing their sum to 1.[5]

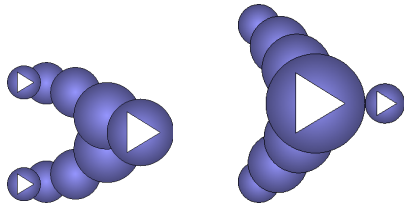


(a) JobsSplit2. (b) JobsSplit2-Var. (c) JobsSplit-2_probPerClass. (d) JobsSplitN.

Fig. 6. Job splitting.

Note that split node probabilities can be associated to either software characteristics or hardware ones. The former case is introduced in the example QN of Section 5. The latter case can be used for routing control in the platform, and in particular for load balancing purposes.

Jobs Merge Nodes. As graphically represented in Figure 7, these elements allow to merge previously split jobs. In particular, JobsMerge2 (Figure 7(a)) can be coupled to any split among JobsSplit2, JobsSplit2Var, and JobsSplit2_probPerClass, whereas JobsMergeN (Figure 7(b)) couples to JobsSplitN.



(a) JobsMerge2. (b) JobsMergeN.

Fig. 7. Job merges.

[5] For sake of simplicity, only `p` has been defined as a knob in the current implementation of our library. The library is open to make all other probabilities as knobs.

4 Assembling AQN and Controllers

4.1 AQN Assembly

There are several ways to model a QN as a dynamic system suitable for the application of control-theoretical management techniques. The one we refer to herein is the combination of two entities. The first one is a set of dynamic systems representing each a processing node, irrespectively of its position in the network. The second one is a probability transition matrix (i.e., a non-dynamic object) that dictates the network topology and the job routing.

The approach just sketched can be applied in both the continuous and the discrete time. In this paper we choose the first case, which implicitly corresponds to assume that job rates are high enough that representing in the model the history of each individual job would be impractical and computationally not advisable. Incidentally, the use of continuous-time models allows to exploit variable-step solvers [25], which can significantly speed up simulation.

The dynamic systems corresponding to nodes, in turn, are the compound of a queue and a server that takes jobs from the queue. The dynamic character comes form the queue, where jobs can accumulate, while the operation of the server is memoryless (i.e., the time to process a job does not depend on the past history of the server). If the queue – for which we assume infinite capacity – does not get emptied, the node model is linear and time-invariant. The transition matrix can be constant, thereby not destroying linearity and time invariance, or some probabilities may be time-varying, in which case the latter property is lost.

If there is more than one job class, all the above is repeated for each of them. In particular, a node has a server for each class it manages, and the probabilities in the transition matrix are expressed for each class—equivalently, the said matrix has a third dimension, and its size in that dimension equals the number of job classes.

As anticipated, the only dynamic element in the library, except controllers, is the model of the server queue. For space reasons we thus concentrate on that model, spending just a few words on the server, and devoting a more comprehensive description of the entire library to specialized works.

We assume the queue occupation $n(t)$ to be a real number, given again the hypothesis of "large" amounts of jobs (e.g., n could be measured in *kilo*jobs). We make the same assumption for the input and output rates $r_i(t)$ and $r_o(t)$. In addition we consider $r_i(t)$ as a purely exogenous input, while we assume that for $r_o(t)$ a required value $r_{o,req}(t)$ comes from the job server. The queue thus emits jobs at (instantaneous) rate $r_{o,req}(t)$ if there are some, otherwise the actual output rate $r_o(t)$ is zero. This immediately reflects into the general differential-algebraic model $\frac{dn(t)}{dt} = r_i(t) - r_o(t)$. Such model can be detailed as follows:

$$\frac{dn(t)}{dt} = \begin{cases} 0 & n(t) \leq 0 \; and \; r_i(t) - r_{o,req}(t) < 0 \\ r_i(t) - r_{o,req}(t) & otherwise \end{cases} \tag{1}$$

As for the server, the model just consists of two computations:

- determine the actual CPU shares by taking the commanded ones, either constant or dynamically computed by a controller, and managing possible over-utilizations (in this paper we simply scale the shares linearly if their sum exceeds the unity, but there are plenty of alternatives);
- determine the required output rate for each job class queue by dividing the corresponding share by the (assumed) processing time for that job class.

4.2 Control Modeling

In this paper we propose a decentralized control strategy, i.e., each node has a local controller and these do not communicate with one another. The purpose of this controller is to regulate the length of the node's queue, job class by job class, by acting on the shares of the node CPU devoted to the servers for the classes. In the treatise we try to use as few control-theoretical concepts as possible, but nonetheless a minimum background on the matter is advisable. The reader needing such information in a compact form can refer, e.g. to [20, Chapters 2 and 4]

In the Laplace transform domain, and limiting the notation to one job class for lightness, the linear behavior of a queue is represented by the transfer function

$$N(s) = P(s)\big(R_i(s) - R_o(s)\big), \quad P(s) = \frac{1}{s}, \tag{2}$$

where $N(s)$ is the transform of the occupation and $R_i(s)$, $R_o(s)$ respectively those of the input and the output job rate. The latter rate is the control variable, as it can be altered by acting on the server CPU share, while the former rate is a disturbance for the local controller $C(s)$. Selecting a PI structure for that controller, we have

$$R_o(s) = C(s)\big(N^\circ(s) - N(s)\big), \quad C(s) = -K\left(1 + \frac{1}{sT_i}\right), \tag{3}$$

where $N^\circ(s)$ is the transform of the desired (set point) occupation, $T_i > 0$ and $K > 0$, both for the Bode criterion and – more intuitively – because to increase the queue occupation one has to decrease the output rate. The open-loop transfer function is thus

$$L(s) = K\frac{1 + sTi}{s^2 T_i}. \tag{4}$$

Accepting to evaluate the cutoff frequency ω_c and the phase margin φ_m on the asymptotic Bode diagrams, we obtain

$$\omega_c = \begin{cases} \sqrt{K/Ti} & K \le 1/Ti \\ K & K > 1/Ti \end{cases} \quad \varphi_m = \begin{cases} \arctan\sqrt{KTi} & K \le 1/Ti \\ \arctan(KTi) & K > 1/Ti \end{cases} \tag{5}$$

To select K and T_i, recall that the closed-loop settling time t_s (which is the time to recover the occupation set point after a step-like disturbance) approximately equals $5/\omega_c$, while the phase margin is a measure of the loop stability

degree and robustness. In this case robustness is not an issue as the model of the controlled system (2) is uncertainty-free. However it is advisable to avoid oscillatory responses, hence to have a "high" φ_m. Since the two cases in (5) correspond respectively to the φ_m ranges $(0°, 45°]$ and $(45°, 90°)$, we select the second one. Given desired values $\bar{t}_s$ and $\overline{\varphi}_m \in (45°, 90°)$ for t_s and φ_m, tuning is performed by setting

$$K = \frac{5}{t_s}, \quad T_i = \frac{t_s}{5} \tan \overline{\varphi}_m. \tag{6}$$

The local controller can either be used to just maintain the queue occupation below a certain level, or can have its set point dynamically calculated as a desired waiting time multiplied by the measured inlet rate. In this case, the effect is to adapt the server processing speed so as to guarantee that the waiting time does not exceed the set point (it can of course be lower if there are no waiting jobs, but this is managed naturally by the PI saturation and anti-windup mechanism).

5 Evaluation

In this section we show the presented Modelica library at work by representing and simulating a network subjected to disturbances.[6] In the example we also briefly illustrate how a quite basic control scheme can improve the behavior of the network, dynamically allotting computational resources so as to maintain a prescribed operation quality in the face of the mentioned disturbances.

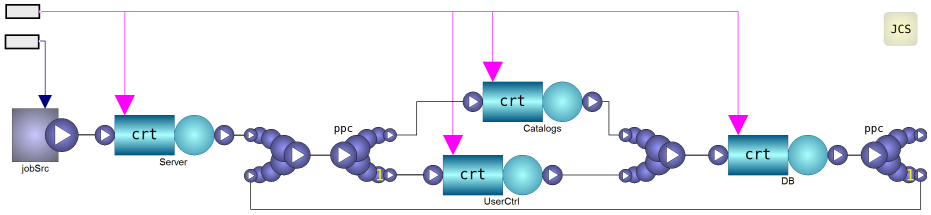


Fig. 8. Simulation example – Modelica diagram for the considered network.

The Modelica diagram for the network is shown in Figure 8, and describes a simple online shop. There are three job classes, namely MakePurchase, BrowseCatalog, and Register. All enter the network through the Server node, and then are split: MakePurchase and BrowseCatalog jobs go to the Catalogs node, while Register jobs are served by the UsertCtrl one. All jobs then go to a database (DB) node. After that, jobs of Register, MakePurchase, and BrowseCatalog, are recycled to merge with the output of the Server node, with probabilities equal to 0.1, 0.3, and 0.5, respectively.

[6] A resource can be downloaded at `http://www.di.univaq.it/davide.arcelli/resources/EPEW2016.zip` including the current version of our library and the usage example in this paper.

The network is subjected to time-varying inputs, as shown in Figure 9. In this figure and the following ones the job classes are numbered, 1, 2, and 3 corresponding to MakePurchase, BrowseCatalog and Register, respectively. All the input rates are composed of a constant baseline value, to which a disturbance made of a double (up/down) step plus a sine-like variation, is superimposed three times in the simulated experiment duration. For sake of simplicity, we do not assume each job class may require a specific service level. Table 5 summarizes the cost in terms of CPU time for processing one job of each class on the four nodes in the network (recall that not all nodes serve all job classes).

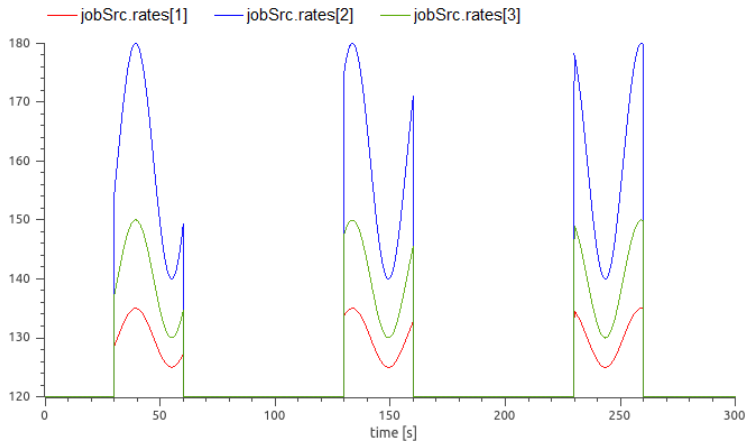


Fig. 9. Simulation example – Input rates.

Table 1. Computational cost of one job of the classes on the network nodes.

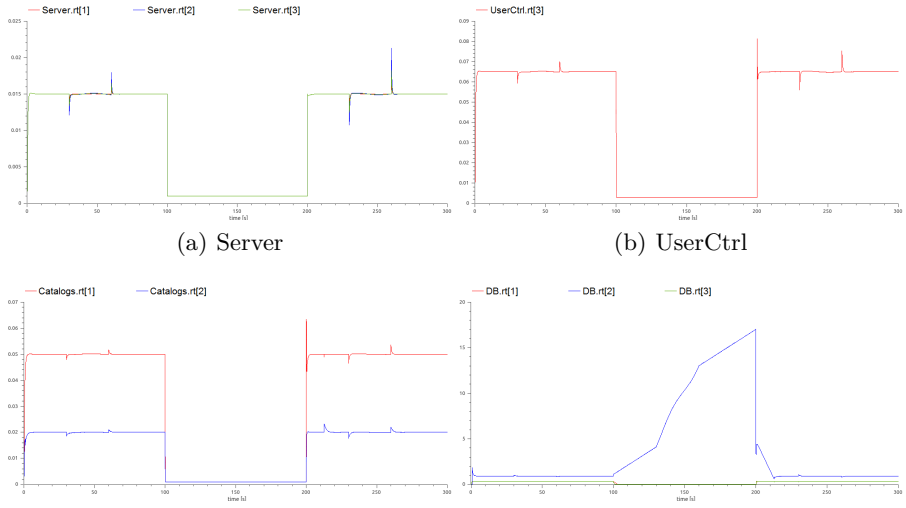
	MakePurchase (sec)	BrowseCatalog (sec)	Register (sec)
Server	0.015	0.015	0.015
UserCtrl	n/a	n/a	0.065
Catalogs	0.5	0.2	n/a
DB	0.3	0.9	0.3

The network is operated in automatic mode for $t < 100$ and $t \geq 200$, while for $100 \leq t < 200$ it is disabled. In automatic mode, each station is governed by a local controller to guarantee a job residence time (i.e., waiting plus processing time) not exceeding 100ms; to do that, the controller can act on the shares of the station CPU allotted to each job server aboard it. When automatic mode is disabled, conversely, each job class server is allotted a fixed station CPU share. The said shares were computed statically for the baseline input rates, plus some over-provisioning for the safe side, with the additional constraint of not exceeding 60% of the entire station CPU availability.

Figure 10 shows the residence time for each class of job on each node in the network. When in automatic mode, local controllers keep the required residence times. The match to the required value is very good, if not for some effects of

the step-like components of the input rate disturbances—the harshest possible *stimulus* in this respect, anyway. All in all, if the required residence times were part of a service level agreement, violations would be practically negligible.

Things are quite different when automatic mode is disabled. The statically allotted CPU shares, given the necessary over-provisioning, cause the residence time with the baseline input rates to be lower than required, which is plain obvious. However, since a reasonable over-provisioning was applied not to unduly steal resources for possible other applications using the nodes, as soon as the disturbance becomes too significant – albeit tractable, as shown by the operation in automatic mode – the requirement on residence times cannot be fulfilled anymore. In the absence of feedback, the network thus drifts away from the goal, as shown in Figure 10(d) for the DB node. The network moves back toward the goal only as the disturbance goes away—or, more in general, comes back within a range that the *a priori* over-provisioned resources can manage. Of course one may think of adapting the over-provisioning amount from time to time based on some analysis of the past behavior and/or some forecast, but as can be seen, feedback control does the job with less complexity than any mechanism of the type just envisaged.



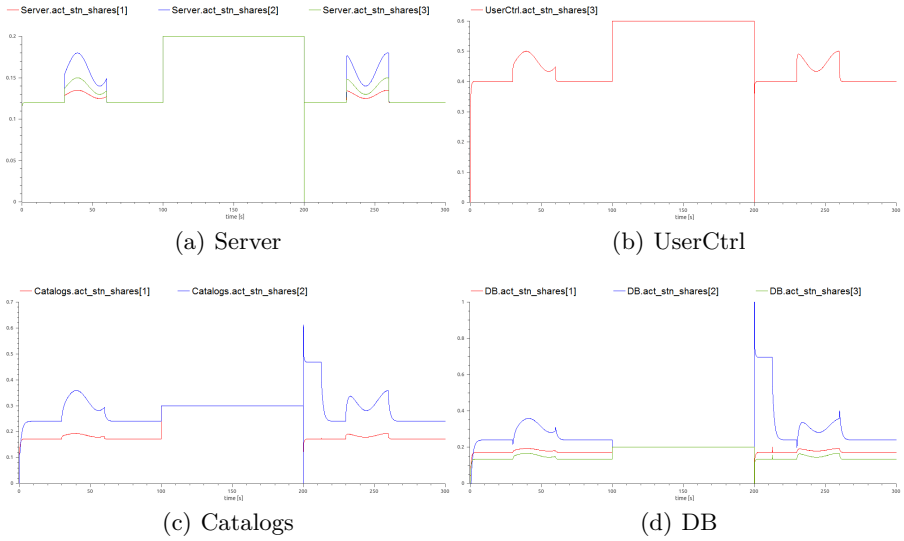
(a) Server (b) UserCtrl

(c) Catalogs (d) DB

Fig. 10. Simulation example – Residence times for each job class on the four nodes.

Carrying, Figure 11 shows the CPU shares allotted to each job class server on each node in the network. The first thing to notice is that with a thoroughly performed tuning – which we do not discuss here as doing so would require space and stray from the scope of this paper – the action of the proposed control, that it is worth recalling to be completely reactive, is fast enough to contrast disturbances very effectively. In other words, considering reactive policies to be "slow" irrespective of how they are set up and tuned, is a conceptual error. If one adds that reactive policies are the only ones usable in the absence of *reliable*

forecasts, and considers how difficult it can be to ensure the reliability of a forecast, then many designs can benefit. Moreover, referring again to Figure 11, the applied local control is inherently capable of moving resources from one server to another when needed, i.e., to not unduly over-provision some server when some other is starving.



(a) Server (b) UserCtrl

(c) Catalogs (d) DB

Fig. 11. Simulation example – CPU shares for each job class server on the four nodes.

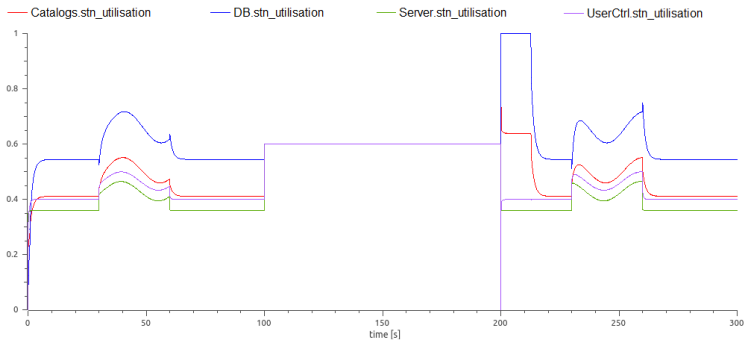


Fig. 12. Simulation example – Total CPU utilizations of the four nodes.

To complete the presentation and the statements just made, Figure 12 reports the CPU utilization for the network nodes. It is worth to be noticed that for most of the time, including when the disturbance is acting, the CPU utilizations under automatic mode are lower than those with static provisioning, and only very sparingly go above the 60% assumed bound. The advantages of this should be apparent in terms of both management simplicity and economy, as no complicated evaluations are necessary to find the best (minimum) over-provisioning, and resources are not allotted unless really needed.

6 Conclusion and Future Work

In this paper, we have presented a library of Modelica components representing QN elements with adaptable parameters. The design of a simple controller have allowed to show an usage example of such library, where an AQN has been built with embedded controllers for satisfying performance goals under disturbances.

We like to remark that our library has been conceived to be open and extensible, so that performance experts and control engineers can define new modeling components both for the AQN and the control modeling sides. For example, in this paper we applied the control mechanism to single QN stations, and controllers do not communicate. A significant short-term extension of our library would be the introduction of hierarchic control for enabling centralized strategies aimed at global performance requirements fulfillment. Another extension that we intend to introduce in the near future would be to support also finite queue capacities, in addition to infinite ones that we have addressed so far.

In the mid-term future, we will relay on our library for tackling many more control problems of AQNs, such as: load balancing and allocation of virtual machines, software, and hardware resources, admission control, temperature/QoS trade-offs, etc. The target here is a broadly applicable methodology for designing control systems based on AQN models with formal quality guarantees.

Finally, as a long-term future work, we intend to implement control systems designed by means of our methodology and embed them into real hardware/software systems, thus "closing a loop" among theory, design, implementation, and practice.

References

1. S. Aalto, U. Ayesta, S. C. Borst, V. Misra, and R. Núñez-Queija. Beyond processor sharing. *SIGMETRICS Performance Evaluation Review*, 34(4):36–43, 2007.
2. J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *ACM PLDI*, 2009.
3. D. Arcelli and V. Cortellessa. Challenges in applying control theory to software performance engineering for adaptive systems. In *ICPE*, pages 35–40, 2016.
4. D. Arcelli, V. Cortellessa, A. Filieri, and A. Leva. Control theory for model-based performance-driven software adaptation. In *QoSA'15*, pages 11–20, 2015.
5. W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *ACM PLDI*, 2010.
6. R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, 2012.
7. V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. L. Presti, and R. Mirandola. Moses: A framework for qos driven runtime adaptation of service-oriented systems. *IEEE Trans. Software Eng.*, 38(5):1138–1159, 2012.
8. C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Performance and environment monitoring for continuous program optimization. *IBM J. Res. Dev.*, 50(2/3):239–248, 2006.

9. R. Dorf and R. Bishop. *Modern control systems*. Prentice Hall, 2008.
10. Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. In *PPoPP*, pages 257–266, 2011.
11. X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck. From data center resource allocation to control theory and back. *CLOUD*, 0:410–417, 2010.
12. L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *J. Instruction-Level Parallelism*, 5, 2003.
13. A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements. In *ASE*, pages 283–292, 2011.
14. P. Fritzson and V. Engelson. Modelica: A unified object-oriented language for system modeling and simulation. In *ECOO*, volume 1445, pages 67–90. 1998.
15. J. L. Hellerstein. Self-managing systems: A control theory foundation. *ECBS*, pages 708–708, 2005.
16. J. L. Hellerstein, V. Morrison, and E. Eilebrecht. Applying control theory in the real world: experience with building a controller for the .net thread pool. *SIGMETRICS Perf. Ev. Rev.*, 37(3):38–42, jan 2010.
17. H. Hoffmann, J. Holt, G. Kurian, E. Lau, M. Maggio, J. Miller, S. Neuman, M. Sinangil, Y. Sinangil, A. Agarwal, A. Chandrakasan, and S. Devadas. Self-aware computing in the angstrom processor. In *DAC*, 2012.
18. J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE*, pages 259 –268, 2007.
19. E. Lazowska, J. Kahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.
20. A. Leva, M. Maggio, A. V. Papadopoulos, and F. Terraneo. *Control-based Operating System Design*. Institution of Engineering and Technology, London, UK, 2013.
21. W. Levine. *The control handbook*. 2005.
22. C. Lu, Y. Lu, T. Abdelzaher, J. Stankovic, and S. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *Parallel and Distributed Systems, IEEE Transactions on*, 17(9):1014–1027, sept 2006.
23. T. Patikirikorala, A. Colman, J. Han, and L. Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *SEAMS*, pages 33–42, 2012.
24. D. Perez-Palacin, R. Mirandola, and J. Merseguer. On the relationships between qos and software adaptability at the architectural level. *SoSyM Journal*, 87:1–17, 2014.
25. L. R. Petzold et al. A description of dassl: A differential/algebraic system solver. In *Proc. IMACS World Congress*, pages 430–432, 1982.
26. S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *HiPEAC*, pages 107–116, 2011.
27. T. Zheng, M. Litoiu, and C. M. Woodside. Integrated estimation and tracking of performance model parameters with autoregressive trends. In S. Kounev, V. Cortellessa, R. Mirandola, and D. J. Lilja, editors, *ICPE*, pages 157–166. ACM, 2011.