RZ 3716 (# 99726) 10/07/2008
Computer Science 12 pages

Research Report

Admission Control for Complex Responsive Systems

Luis Garcés-Erice

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

E-mail: lga@zurich.ibm.com

Admission Control for Complex Responsive Systems

Luis Garcés-Erice
IBM Zürich Research Lab
Säumerstrasse 4, 8803 Rüschlikon, Switzerland.
Email: lga@zurich.ibm.com

Abstract

Modern software systems have become too complex to make accurate predictions about their performance under different configurations. Real-time or even responsiveness requirements cannot be met because it is not possible to perform admission control for new or changing tasks if we cannot tell how their execution affects the other tasks already running. Previously, we proposed a resource-allocation middleware that manages the execution of tasks in a complex software system with real-time requirements. The middleware behavior can be modeled depending on the configuration of the tasks running, so that the performance of any given configuration can be calculated. This makes it possible to have admission control in such a system, but the model requires knowledge of run-time parameters. We propose the utilization of machine-learning algorithms to obtain the model parameters, and to be able to predict the system performance under any configuration, so that we can provide a full admission control mechanism for complex software systems. In this paper, we present such an admission control mechanism, measure its accuracy in estimating the parameters of the model, and evaluate its performance to determine its suitability for a real-time or responsive system.

I. Introduction

Traditionally, real-time systems have mainly been used in tightly controlled environments, such as mission-critical and embedded systems. These systems are usually characterized by a set of well-known tasks that have to be executed by a scheduler. The tasks are modeled after the time it takes to execute them and the period with which they are to be executed (or, more generally, a deadline by which the task must have finished execution). The scheduler follows a policy that establishes which task has to be executed next, such that all tasks are finished before their corresponding deadlines. Examples of such policies are Rate Monotonic Analysis (RMA) or Earliest Deadline First (EDF) [1], and many of their variations. In the embedded system case, there is a great amount of control on the tasks that may be executed: even the number of tasks to execute is known *a priori*, having the system fully specified and working by design. In scenarios where more flexibility is allowed, the arrival of tasks for execution is not predefined, although their execution characteristics are specified. In that case a mechanism for *admission control* is needed, such that the scheduler may refuse to execute a task if insufficient resources are available. This lets the tasks already scheduled be processed knowing that their requirements will be met. Again, this approach works because, given the scheduling policy and the specification of the task execution characteristics, a calculation can be carried out on whether a new task can be scheduled, given the tasks currently scheduled and the available resources. This calculation is at the core of any admission control mechanism.

Recently, interest has grown for enterprise systems to become more “real-time”, i.e., the focus has moved from pure average performance to a more controlled evaluation of the performance, where the worst case matters as much as the average. However, for a number of reasons software running on these systems cannot be fully specified in contrast to the embedded counterpart: it is difficult to know at all times what is actually running on a host, as for example background tasks and OS daemons may execute at unpredictable times; even when the applications are known, their characteristics are difficult to describe; and traditional techniques for real-time analysis such as those described before do not fit well with the very complex software stacks run in today’s application servers.

An attempt to make the problem more tractable has been the introduction of real-time messaging middleware [2]. If all or most applications running in a system can conform to a common API and programming model, the execution of the applications can be largely under the control of the middleware. Given that, the middleware has the possibility to play the role of the scheduler in the real-time system, with the applications being the tasks (note that this is not a perfect mapping to the traditional embedded system model). The middleware can account for the temporal consumption of resources by the applications, enforcing a certain partition of the resources much like the scheduler in a real-time system. And, also, some form of admission control is needed, such that a new application or a change in the configuration of one application does not disrupt a correctly working system.

Thus, provided that the middleware manages the execution of all (or the most important) activities in the system, and given a mathematical model of the middleware execution, the performance of the system can be calculated in a given configuration. This allows us to have an admission control in a complex real-time system, because we are able to determine whether the introduction of a new application violates the performance requirements of the other tasks already running. Still, we found in previous work [3] that, even if the middleware allows the modeling of the system, the complexity of the underlying technology (Java$^{\mathrm{TM}}$VM garbage collection, OS internals and daemons, application logic outside the control of the middleware, etc.) makes it impossible to have a closed-form mathematical model of the system that is valid in all circumstances. We could, however, obtain a *family* of models that depend on some parameter that is determined by the configuration of the system (i.e., by the application behavior). For simplicity, in this paper we consider only one parameter, but the same principles can be applied to a system dependent on various parameters.

As we show below, the relation between the configuration of the system and the values of the parameter defining the model is highly nonlinear. The values of the parameter depend on complex interactions at all levels of the software stack. It is not evident (or simply feasible) to establish *a priori* the parameter value corresponding to a given configuration, except for the simplest configurations. However, we assume that we are facing a coherent system, so that the configuration and the parameter describing the performance of the system cannot be randomly related, but are subject to some complex logic. We attempt to grasp this logic by use of machine-learning (ML) techniques. We decide to model our problem as a classification system, using decision trees. The objects for classification are the configurations of the system, and the resulting classes are ranges of values for the parameters of interest in the model. The ranges should be small enough so that they provide a good estimation of the actual parameter value.

There are a number of ML methods that we could have used, such as neural networks or clustering algorithms. We decided on decision trees for the following reasons: Firstly, there is extensive literature on satisfactory results in a number of fields where decision trees have been applied to complex nonlinear problems. Secondly, the extensive use of decision trees has provided mature algorithms that offer very good performance, a critical feature in a responsive or time-critical system, where decisions need to be made fast enough. Finally, the output of the decision tree creation algorithm is a human-readable structure. This is an invaluable analysis tool for a system administrator, who can obtain insight into the behavior of the complex system by looking at the decisive attributes in the decision tree.

The paper is organized as follows: Section II offers an overview of related technologies in real-time complex systems, ML algorithms and their applications. Section III describes the basics of decision tree algorithms. Section IV introduces the modeling of the real-time middleware and the need for a complex nonlinear problem solver. Section V describes the entire system design and how it works; Section VI describes the calibration phase and how the ML algorithm uses the data and Section VII shows the results obtained. Finally, Section VIII presents some possible future lines of research before concluding.

II. Related Work

Quite some work has been developed on admission control for QoS-enabled networking. There are obvious similarities between the purpose of QoS networking and that of our middleware: some resources (bandwidth resp. CPU) are to be shared among competing entities (flows resp. tasks), and the resources have to be allocated according to some reservation (expressing the need for resources). An admission control algorithm for real-time traffic that takes advantage of scheduling information is presented in [4]. Linear programming is used to obtain an admission control policy that preserves the guaranteed QoS and maximizes utility. The authors of [5] measure the maximal rate envelopes of flows. With those measurements, they can predict with some measurable certainty the schedulability of new flows, which are admitted provided the QoS of all flows is respected. Reference [6] describes an admission control algorithm for delay-bounded packet networks that provides a predictive service.

The authors of [7] propose real-time middleware to facilitate the use of the real-time QoS capabilities of the OS by the applications. Thus their approach is different in that they assume the presence of an OS with real-time capabilities, for which the middleware is merely an enabler for applications. The goal of making real-time applications easier to write is shared by both efforts, however.

The real-time literature for complex systems can for the most part be described as an extension to distributed systems of the task-scheduling problem [8]. The assumptions about the tasks are the same as in embedded systems: well-known execution time and well-known available resources. In the middleware literature, QoS requirements are generally used as matching criteria for two end-points to connect [9], but without specifying the mechanisms to actually enforce the QoS requirements. These systems must assume over-provisioning of resources, a scenario where admission control is clearly not required. Other approaches more concerned with the actual system performance are adaptive bandwidth reservation schedulers [10]: Reservation-based schedulers reserve a fraction of the total CPU to each task, but they assume that the bandwidth given to a task is sufficient to fulfill its timing constraints, so that no admission control is needed. Instead of preemptively controlling access of new tasks, the authors of [10] propose to use adaptive reservation for the tasks, such that each allocated fraction of the CPU is determined by a closed control loop. The authors of [11] also present an on-line approach to adapt the QoS of distributed real-time systems, and use distances between states to define the optimal operational point. Control theory is used to bring the system to a desired point of operation, while maintaining some system performance level. The system attempts to allocate resources for all tasks according to their needs, but it can not be ensured that the requirements of all tasks running can be satisfied. In such systems, even if a task is terminated because its requirements cannot be met, some tasks may in the meantime have been negatively affected by the resource allocation.

Lately, there have been various efforts on real-time and responsive systems in complex software stacks [12], but in general admission control has not been considered. We believe that this is mainly because being able to decide whether some task can be accommodated in a complex running system respecting some performance criteria requires too much control on software that is often not fully understood, known, or describable. Such difficulties have been met while trying to introduce empirical performance data into complex software stacks such as service-oriented architectures [13], [14].

There is extensive literature on decision trees. Section III provides an overview of the underlying principles. The seminal work of Quinlan explaining the ID3 algorithm [15] is recommended reading for an in-depth understanding of decision trees.

The algorithm we use in this paper, C4.5 [16], is one of the most widely used, and incremental improvements have also appeared in the literature [17], [18].

Other ML methods that could have been used instead of decision trees are neural networks. Their performance is similar [19], but the resulting neural networks do not produce any intelligible information for system administrators, and have to be used as black boxes. Both approaches are definitely related, as decision trees can in fact be extracted from neural networks [20].

ML techniques have been applied to a number of fields, but it is their recent application by the network community to traffic identification that inspired this work. In [21], clustering algorithms are used to group traffic flows having similar characteristics, using the K-Means algorithm [22].

III. Decision Trees

Decision trees are general-purpose knowledge-based systems that classify objects into a set of classes according to the attributes of those objects. The decision tree itself represents the knowledge acquired by the system. The leaves of the tree correspond to classes. Every other node in the tree is a test on a given attribute of an object, and for every possible outcome of the test there is a branch coming out of the node. A branch is directed to a leaf, classifying the object, or to another node, where the classification process continues.

Decision trees do not have the expressive power of other knowledge formalisms, but their simplicity allows much simpler learning methodologies. This translate into faster algorithms and more responsive obtention of results.

Decision trees are useful for expressing nonlinear relations between a set of objects and a set of classes through the attributes of the objects. Given a sufficiently large set of objects and a mapping to a set of classes, the underlying criteria by which objects are classified are usually nonevident, and their obtention, if at all possible, requires an enormous amount of work. The usual procedure is that of having a pool of observed data, composed of a set of objects with given attributes and the objects' classification, which is observed rather than systematically obtained (otherwise we could already classify any object!). The intention is then to be able to classify another object, given its attributes, following the same logic that permeates the observed data. In other words, the aim is to accurately predict the class corresponding to an unobserved object. The assumption is that the classification logic can be expressed as a decision tree. The goal is to construct, based on the observed data, the decision tree that gives us the right classification for unobserved objects.

The task of building decision trees from already classified objects and their attributes is in general NP-Complete. There are a number of greedy algorithms (such as ID3 or C4.5) that can efficiently compute good approximations to the optimal solution. The main idea is to create the simplest decision tree that correctly classifies all the observed data. This is achieved by making the branching decision at each node based on the attribute that reduces the entropy of the subset of objects classified under that node so far to the greatest extent. The entropy of a set of objects increases with the number of classes the objects of the set belong to. When a node produces 0 entropy for all objects classified under that node, it is said to have reached purity, as all those objects correspond to the same class. That node is a leaf node for the decision tree.

A detailed explanation of the general principle and the ID3 algorithm in particular can be found in [15]. For our experiments, however, we use C4.5, which is based on ID3. Although the basic idea for the construction of the trees remains the same, C4.5 solves a number of problems present in ID3: C4.5 avoids over-fitting of the tree to the observed data by pruning the tree; C4.5 allows the description of attributes by a range of real numbers instead of just discrete values, a property that makes our approach feasible. There is plenty of literature on these algorithms, and their internal details would exceed the scope of this paper. For our experiments we have used the C4.5 algorithm implementation by Quinlan [16], which is available from [23]. We have introduce minor modifications to the application, so that tests for multiple objects can be performed in a batch.

IV. Modeling Complex Middleware

Current software stacks are built from heterogeneous components whose internals usually are not fully understood. This makes the extraction of a useful model that provides any insight into the behavior of the system virtually impossible. This in turn prevents predicting how the system will be affected by the addition of a new task, making it impossible to decide whether the task should be accepted. This is why we propose a middleware that takes care of the execution of the different parts of the system and provides a tractable model that can be analyzed, such that admission control can be implemented. A description of such event-based middleware used in a publish/subscribe system can be found in [2]. At the core there is a scheduler that assigns resources to tasks, detailed in [3]. In a nutshell, credits are assigned to tasks, which implement the application logic and communication means. The total credits allocated to a task are a fraction of a time period T. This fraction is a proxy for a CPU share. The scheduler always chooses to schedule the task with the highest number of credits that has a nonempty event queue, i.e., that has work to do. When a task is scheduled, the run time the associated handling function needs is measured and deducted from the total number of credits for that task. The credits of all tasks are reset when all tasks with work to do have no credits left.

This scheduler enables a processor time allocation that follows a Weighted Max-Min policy, which can be expressed using the following mathematical model: On an N processor machine assume that the share allocated to task i is $0 \leq z_i < 1/N$ (tasks are processed by a single scheduler thread) and that s_i is the number of events arriving at the task queue per second.

The question we wish to address is the rate of events r_i that is actually serviced for task i. The number of events serviced for a task can be viewed as containing two components: the rate guaranteed by the share and any spare capacity that a task can take advantage of. Assuming that the cost to process an event is constant, we denote the total maximum achievable rate by $R^{\max}$ events per second (evt/s). $R^{\max}$ is a function of system settings and the parameter the model depends on.

For a given setting of z_i, s_i and $R^{\max}$, the expected rate r_i is:

$$r_i = Min\left(s_i, z_i.\frac{R^{\max} - \sum_{x_j > x_i} r_j}{1 - \sum_{x_j > x_i} z_j}\right) \tag{1}$$

where

$$u_i = \frac{s_i}{\sum s_j}, \quad x_i = \frac{z_i}{u_i}, \tag{2}$$

and we define u_i to be the fraction of resources that a topic is *attempting* to consume and x_i to be the ratio of z_i to u_i. Equation (1) states that the share that a task gets is its *adjusted* share of what is left over after tasks with higher x_i have been allocated. The amount of additional unused capacity available to a task is the capacity left over by tasks that are less speculative than the task itself.

In order to check the fidelity of the model to the real implementation of the scheduler, we compare the throughput o_i obtained on each of the t tasks with the corresponding throughput calculated by the model. We quantify the difference between the t-dimensional vector of rates $\bar{r} = (r_1, r_2, ..., r_t)$ predicted by the model and the vector of actual rates $\bar{o} = (o_1, o_2, ..., o_t)$ obtained in the experiment by calculating the Euclidean distance between these two vectors, $e = ||\bar{r} - \bar{o}||$. The value e by itself is not a good indicator of the performance of the model, because it depends on the magnitude of $\bar{r}$ and $\bar{o}$. To provide some meaningful reference, we compared this distance to the average distance between two randomly chosen points in the same vector space. We perform a number of experiments that cover a range of system configurations. We vary the following parameters: the sum of the throughput being processed by all tasks, which is important with respect to $R^{\max}$; the number of tasks, which increases the unaccounted overhead; the task shares, and the event rate produced by the tasks, which is in principle independent of the share allocated to that task. We produce 50 different, randomly generated task rate and share settings for each total throughput (30,000, 100,000 and 200,000 evt/sec) and number of tasks (10, 20, 50 and 100), providing a total of 600 scenarios under test using two scheduling threads on a two-core machine. We obtain the $R^{\max}$ parameter value from each experiment.

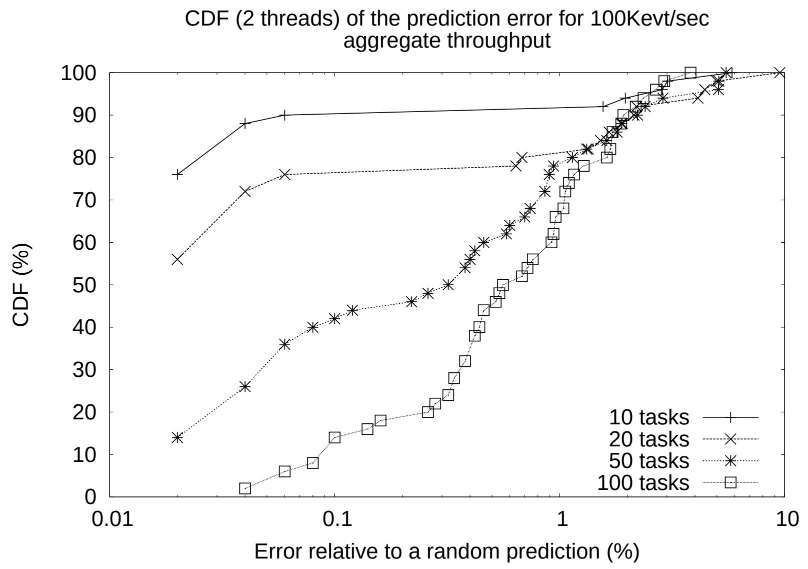

Fig. 1. CDF of the relative error for total aggregate throughput of 100,000 evt/s depending on the number of tasks, for two scheduler threads.

The results in Fig. 1 show that the model reflects the behavior of the actual system with a very high degree of accuracy. The larger the number of tasks, the lower the fidelity. But even for 100 tasks, the prediction in the worst case is 10 times better than the random predictor reference. For 10 tasks, almost 90% of the settings' behavior is described 1000 times more accurately by our model. To summarize, we are able to predict the performance of the system with high accuracy by using a model of the middleware that depends on one parameter.

V. System Description and Utilization

Our middleware provides an execution environment for complex time-sensitive applications. These applications have some timing and performance requirements, which have to be met by using the amount of resources reserved through the middleware.

Our utilization model is such that each application (a task) requests a share of the resources, intended to process a certain rate of events per second. The requirements of a set of tasks form a configuration of the middleware. A configuration is invalid if the amount of resources requested is greater than the total available. Assume a running system in some valid configuration. A new task is introduced (or the requirements of an existing one are changed), resulting in a new configuration ($configuration_x$ in Fig. 2). As can be seen in Fig. 2, the output of the admission control system is either to accept or refuse the new configuration. The admission control mechanism has to decide whether the new configuration can be applied to the system. To do so, the system needs to 1) determine the performance resulting from the application of the configuration and 2) decide whether that performance is satisfactory.

We have seen in Section IV how it is possible to obtain a mathematical model that accurately predicts the performance of the middleware under a given configuration. The model ($Middleware\, model$ in Fig. 2) needs a run-time parameter that depends on the configuration. The value of this parameter ($param\_value_x$ in Fig. 2) is obtained from the new configuration using a knowledge system, in our case a decision tree. Then a prediction for the performance of the system under the new configuration ($model\, performance\, result$) can be obtained.

Defining whether the performance of the system for a given configuration is satisfactory may be performed in a number of ways: in the more straightforward solution, which is the most strict one, performance is only acceptable when all individual tasks in the system meet their performance requirements. In a more flexible solution, a system administrator may decide that some tasks are more important than others, and thus the configuration may be changed to meet the requirements of some tasks, even at the expense of having some other tasks penalized. The criteria by which this is decided may be arbitrarily complex, and in the end it is up to an administrator of the system to define the appropriate policies. Going into the details of how these policies may be set is beyond the scope of this paper. We assume, however, that the policies are set by an administrator (off-line) or some automated means (on-line) and that they are available at any time during execution.

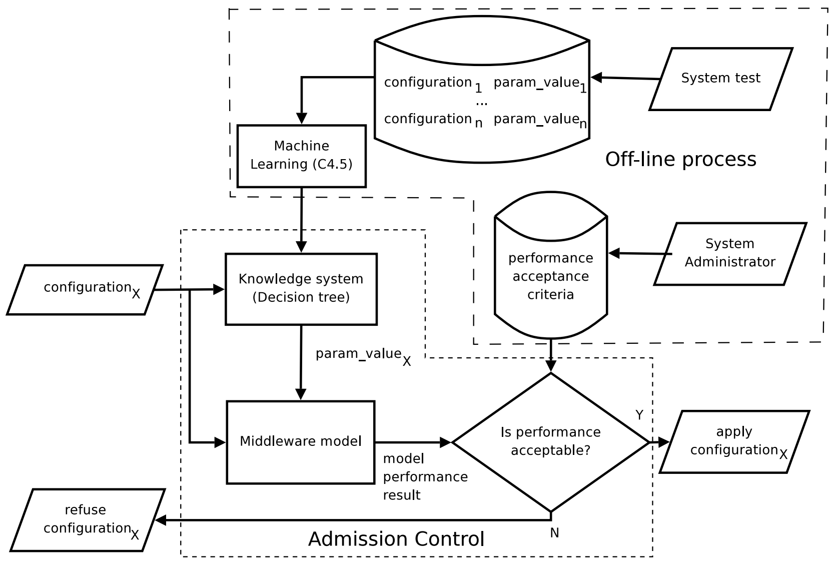

Fig. 2. Diagram of the admission control system using the real-time middleware.

The decision tree needed to estimate the run-time parameter of the model is obtained using an ML algorithm like the one described in Section III. The introduction of ML in the system design means that there has to be a training phase in which the ML algorithm *learns* which are the criteria linking one configuration to a given value of the model parameter. This training phase may be run off-line, by having the system actually run in a number of configurations and measuring the corresponding values of the parameter of interest (top of Fig. 2). Then this data in the form of ($configuration_i$, $param\_value_i$) pairs is fed to the ML algorithm (C4.5 in our case) to obtain the knowledge system (a decision tree in our case) used in the admission control mechanism. We argue that running the training phase is not an exceptionally constraining requirement, as any software platform needs to undergo a test and verification phase before it is deployed in production; even more so if we speak of software with subtle performance requirements. It is conceivable to have such a training phase on-line, so that the ML algorithm learns for some time while the system is in operation. It is difficult to define what the behavior of the admission control system would be during the training phase, as not sufficient knowledge is acquired to produce a sound decision. A possible solution is to have all configurations allowed, even at the expense of violating the performance requirements on occasion. But if this extreme is acceptable during normal operation of the system, then the need for an admission control system (or having the system defined as real-time or time-sensitive) is difficult to justify.

It would also be possible to have the decision tree recalculated periodically as new information about the performance of the system is collected (for clarity, this part has been omitted in Fig. 2). A background process could gather new configuration/parameter value pairs, and when a certain amount of new knowledge is attained, it could be added to the calibration phase data and fed to the ML algorithm to obtain a more accurate decision tree. We evaluate the feasibility of this approach in Section VII-C, by measuring the time required to build a tree.

VI. System Calibration (Learning Phase)

The calibration consists of gathering information about the behavior of the system under a variety of configurations. How these configurations are chosen is discussed below. For now, assume that there is a set of configurations of interest such that it represents the expected range of operation of the system. When the system is being calibrated, under each configuration, the behavior of the system is captured by registering the value of the parameter of interest for the middleware model. In this way, at the end we obtain a set of configuration and parameter value pairs. This data is fed to the C4.5 ML algorithm, which creates a decision tree that reflects the nonlinear relation between the configuration and the model parameter value. The configurations are thus the objects to be classified by the decision tree, while the parameter values are the classes. The model parameter value of a configuration is obtained by classifying the configuration with the decision tree. We describe now how we present the data to the C4.5 algorithm to obtain the classifier.

A configuration is defined as follows: On an N processor machine, assume that the share allocated to task i is $0 \leq z_i < 1/N$ and that s_i is the number of events arriving at the task queue per second. The configuration for a system with t tasks is then given by a vector $W = [(z_0, s_0) \ldots (z_{t-1}, s_{t-1})]$. Note however that, given that we assume the cost of processing an event to be equal for all tasks, there are different configurations that are equivalent. In fact, as far as the middleware is concerned, all permutations of W are exactly the same. Thus all permutations of W are expected to yield the same parameter value, and should be classified the same. We therefore need to represent all permutations of W as a single object, such that the classifier does not become unnecessarily complex. In this way, more information is conveyed by the decision tree as relations between similar objects (configurations) are captured by the common representation. A further advantage is that, knowing that two configurations are equivalent, we do not need to calibrate the system for both of them. Thus the common object representation of all permutations of W is defined by a vector of pairs: V = [$(z_i^0,\ s_i^0) \ldots (z_j^{t-1},\ s_j^{t-1})$], such that $\forall p, q,\ if\, p < q \rightarrow s_i^p > s_j^q \vee s_i^p = s_j^q \wedge z_i^p > z_j^q$. This means that the first setting in the configuration vector is the one having the largest event rate, the second has the second largest rate, and so on. If two settings have the same event rate, the order is decided by having first the one with the larger share. Note that this ordering can be extended to an arbitrary number of elements per setting, provided that each element can be ordered. Also note that this particular ordering is arbitrary and others may work as well (such as ordering first according to the share z_i^p). The important property of this ordering (or any other similar to that) is that it represents two identical configurations equally, independently of which tasks are configured with which setting. Indeed, the system only cares about how the resources are partitioned, not which particular task is getting which share of the resources.

The run-time parameter of the model can have any value within a range in a continuous space, but decision trees have discrete classifications as output. To overcome this apparent mismatch, we divide the range of the parameter value into categories of equal size. The range can be estimated from the calibration phase itself, as the various configurations under test, if properly chosen, provide a good indication of the values expected for the parameter during normal execution. This range may be extended to account for outlier values not seen during calibration. In effect, we are quantizing the estimation of the parameter value. The less quantization error we want to introduce, the more classes we have to define in the output of the decision tree. For simplicity, we opt for a uniform quantization scheme, although with some models a nonuniform scheme may yield better results.

To summarize, in our case, a certain number of configurations and their corresponding $R^{\max}$ are obtained during the calibration period. We denote the k^{th} configuration as vector V_k. Hence the result of the calibration period is a set of pairs of the form (V_k, $R_k^{\max}$). We call this set C. The decision tree has to be trained with the calibration samples in C.

VII. Experimental Results

To validate our admission control mechanism, we perform a number of tests using a publish/subscribe implementation of the real-time middleware described above. The tasks are publishers (information producers) producing a rate of events (messages on a topic) that are sent over the network. Each publisher uses a different topic. A second node runs tasks that are subscribers (information consumers) on those same topics, which we use to measure the rates without altering the execution of the publishers on the first node. Each machine is an IBM®HS20 blade with 2 Intel®Xeon™3.20GHz 64-bit processors with Hyperthreading™enabled and 2 Gbyte of RAM. The machines are connected directly using a Gigabit Ethernet switch. The message size is 128 bytes, so that even at the highest sending rates the network bandwidth is not an issue. Two scheduling threads are run on each two-core machine.

We perform our first test with a two-topic space where we try all possible *different* configurations (4,500) for a total sending rate of 100,000 msg/sec. The range of the parameter $R^{\max}$ we observe is 60,000–100,000 msg/sec. Thus we divide the output into 40 classes, from the lower limit to the upper limit of the observed range, in steps of 1,000 msg/sec. If all configurations where perfectly classified, we would expect an average error of 500 msg/sec in estimating $R^{\max}$, due to the rounding introduced by the quantization of the output space. With the configurations and their corresponding parameter values, we can build C.

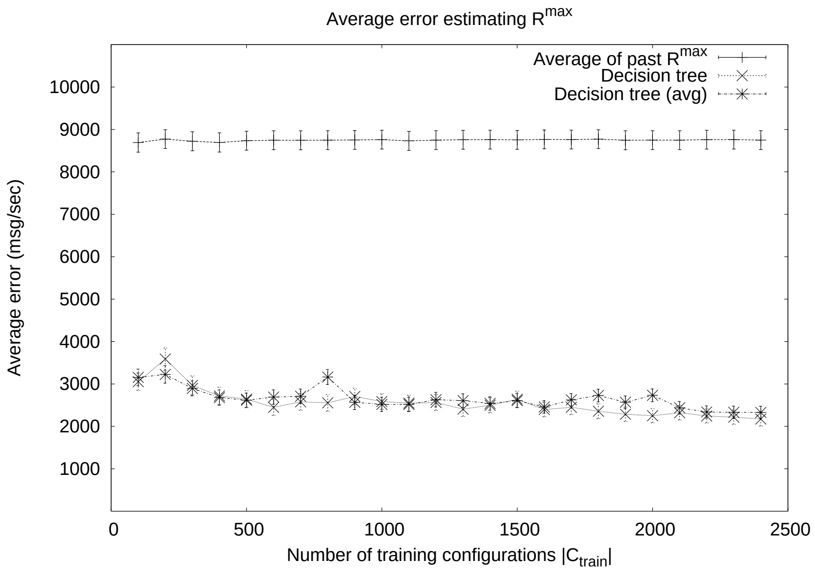

Fig. 3. Comparing the error estimating $R^{\max}$ when using the decision tree and when simply taking the average of the parameter values obtained during calibration.

To validate our approach, we use a fraction C_{train} of the configurations in C to train the decision tree, and use a disjoint set C_{test} to measure the accuracy of the estimated $R^{\max}$ (i.e., the output class of the decision tree) compared with the actual $R^{\max}$ obtained during calibration. In Fig. 3 we compare the error estimating $R^{\max}$ between the decision tree and a naïve predictor that takes the average of the parameter values in C_{train} (confidence intervals are 95%). We use between 100 and 2,400 configuration in C_{train}, and consistently test the resulting decision tree with 2,100 configurations in C_{test}. We can observe that the error using the decision tree is much lower than in the naïve method, namely around three times smaller. We also observe that the decision tree improves its estimation as the number of configurations used for training increases, although not much. Even using half the available configurations for training, the error estimating $R^{\max}$ is still above 2,000 msg/sec. This is not too bad, considering the 500 msg/sec quantization error the classification introduces. There is a trade-off between the number of classes and the possible accuracy obtained by the tree. A few classes introduce a large error in the estimation, whereas many small classes somehow dilute the significance of each class by having few configurations match it.

It is also promising that we obtain significantly good results even by training the algorithm with just a few configurations (the first 300–400 seem to provide most of the improvement in the estimation). The shorter the calibration phase, the easier it is to actually use our approach.

Most of the time, the decision tree does not provide a unique possible classification for a given configuration. A number of possible classes $R_i^{\max}$ are provided with a probability p_i associated with them. Previously we picked the class with the highest probability as result. In an attempt to obtain better estimations, we produce the weighted average of the possible classifications, $\sum_i p_i R_i^{\max}$, as result. This however does not seem to yield better results (see plot *Decision Tree (avg)* in Fig. 3), at least in this configuration space where a lot of information is available.

The result in Fig. 3 is proof that the problem is not trivial and that there is room for an algorithm to fail (as with the naïve method). The value of our approach has been demonstrated, and we now proceed to study its performance more closely.

A. Realistic experiment

The previous experiment is not very realistic in that there is a reduced configuration space and all configurations are possible. In a real system, we expect to find more complex configurations, involving a larger number of tasks. Consider now that in the previous experiment, with only four degrees of freedom, we obtained close to 5,000 possible configurations. This number explodes as configurations get more complex. Thus, we expect a system administrator to know in which regime the system is supposed to operate, providing an indication of which configurations are the most common around which the system works.

We test our approach in three different scenarios. We use the same setup as in the previous experiment, but change the configuration accordingly. In all figures, confidence intervals are 95%.

1) Random walk steps: In a system with 10 publishers, we run 1,000 different configurations. These configurations are generated as a random walk; but the starting point is changed every 100 configurations. First, we use a configuration in which all resources are allocated to one publisher, and all messages are sent through that publisher. Then we introduce random variations to the configuration, obtaining a second one, which we randomly modify in turn, and so on until we have generated the first 100 configurations. Then we pick a configuration in which all the resources are equally divided between two publishers, and the messages sent are split equally between the two. Using the same method, we obtain the next 100 configurations, and so on. Configuration 900 corresponds to all the resources being equally distributed over the 10 publishers, and from there the final 100 configurations are obtained.

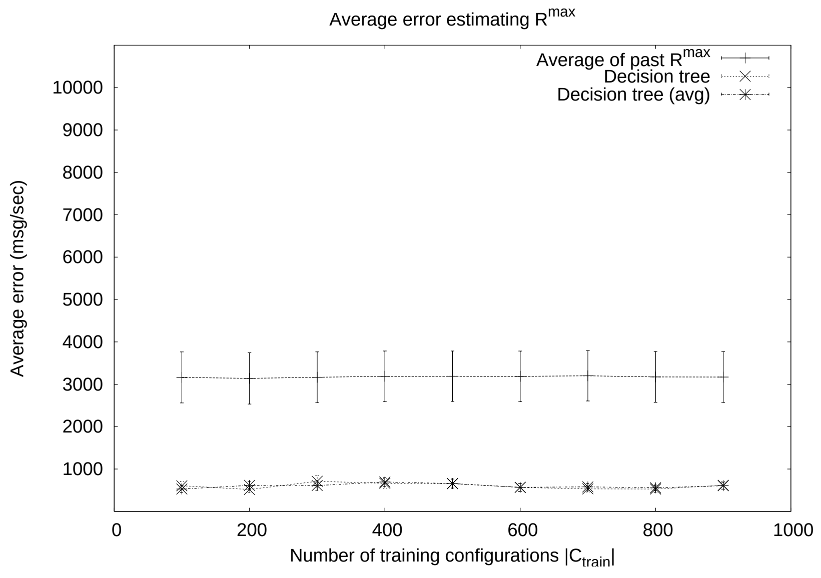

Fig. 4. Comparing the error estimating $R^{\max}$ using the decision tree and simply taking the average.

Fig. 4 shows the error incurred by our algorithm when estimating $R^{\max}$ when using $|C_{train}|$ configurations picked at random to train the algorithm and using another 100 for testing. The $Decision\,tree$ plot shows the error when picking the class with the highest probability, and $Decision\,tree\,(avg)$ shows the error when taking the weighted average as explained above. There is no visible improvement. We also show the performance of the naïve predictor that takes the average of the parameter value in C_{train}. The decision tree stills outperforms the naïve approach, but this time we observe that the error (600 msg/sec) is very close to the quantization error (500 msg/sec). In this test, the decision tree performs almost as well as it can possibly do given the class partition we are using. Moreover, training with the first 100 randomly chosen configurations is as effective as using the entire C_{train}. This vastly reduces the time needed for training in the calibration phase of the deployment of the system.

2) Star walk steps: This time we again use 10 publishers and also run 1,000 different configurations, but they are generated differently. We assume the same 10 starting configurations as in the previous example (1 publisher gets all the resources, 2 publishers get all the resources, ... etc.), and we generate 100 random variations from each of them (thus the name star walk). This setup is closer to a system administrator providing a number of commonly used configurations, and extending the training set by adding similar configurations around the points of operation. The results are shown in Fig. 5, where we train the algorithms with 100 to 900 configurations in C_{train} and test them with another 100 chosen at random. This time the naïve approach performs worse, meaning there is more variation in the values that the model parameter $R^{\max}$ can take. However, the decision tree performs similarly as in the previous experiment, producing an error in the estimation very close to the quantization error. Also in this case, testing for many configurations does not seem to pay off. Again, the average of the tree decisions does not improve the quality of the result.

3) More complex star walk steps: Finally, we try a configuration with 20 publishers, which doubles the number of degrees of freedom in the test. We run the experiment for 2,000 configurations generated like the ones in the previous experiment, but starting with 20 configurations. C_{train} contains from 100 to 1,900 configurations, and we still test with another 100 configurations. With the larger configuration, the error estimating $R^{\max}$ is close to 700 msg/sec, which is worse but still close to the quantization error. Running 200–300 configurations in C_{train} seem to provide all the benefit that the algorithm can bring; so again, even for fairly large configurations the calibration phase should not take long.

All the following experiments use this last, more complex configuration (20 tasks).

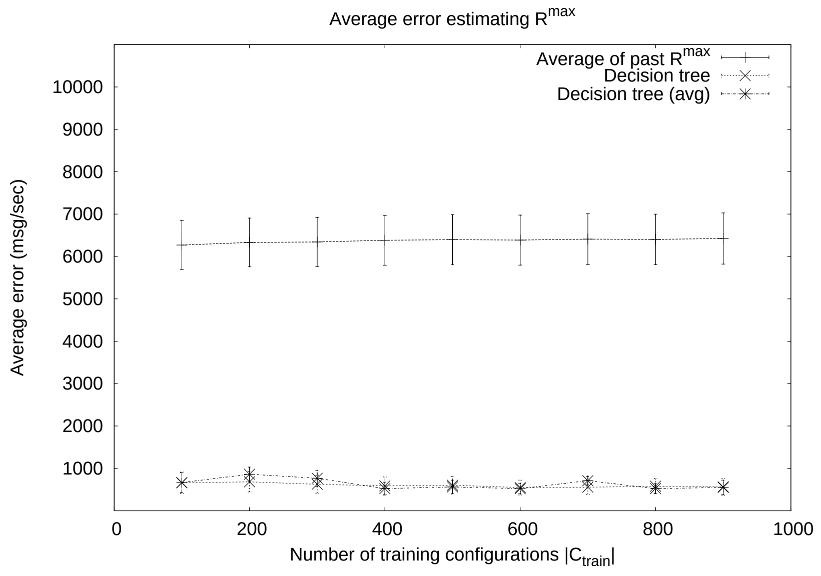

Fig. 5. Comparing the error estimating $R^{\max}$ using the decision tree and simply taking the average.

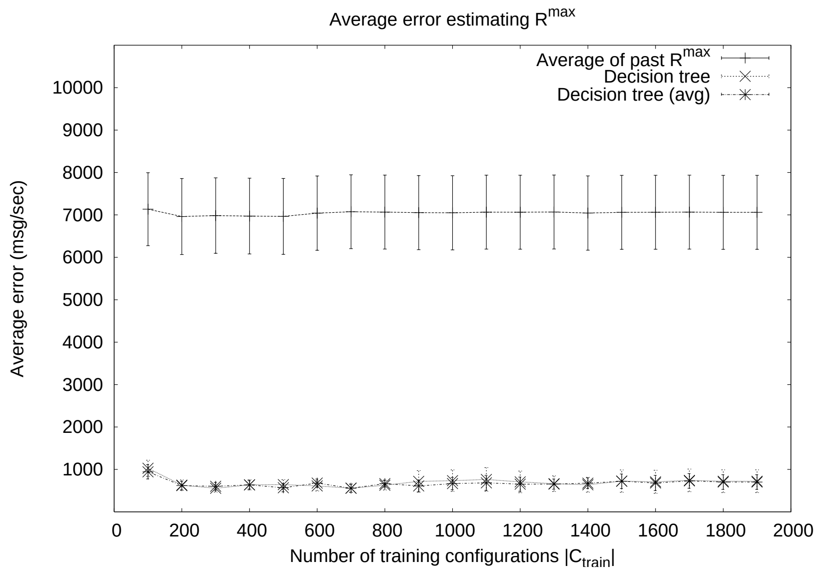

Fig. 6. Comparing the error estimating $R^{\max}$ using the decision tree and simply taking the average.

B. Class granularity

We have seen that the error estimating the parameter $R^{\max}$ is quite close to the quantization error when we use 1000 msg/sec classes. We perform the same tests but using a tree with classes spanning a shorter range: we divide the output into 500, 200, 100 and 10 msg/sec classes, which make for 80, 200, 400 and 4,000 output classes, respectively. In Fig. 7 we show the average improvement of the estimation measured in msg/sec (more is better).

If we look at the results when training the tree with the first few hundred configurations, we see that only the classes of 200 msg/sec provide a significant advantage over the 1000 msg/sec classification we have used in previous experiments. This is, however, probably due to an artifact of the particular configurations under test, because otherwise we would expect some improvement also from the division into 500 msg/sec classes. We observe, though, that when we increase the configurations in C_{train} above to 1,000, all classifications improve their accuracy over the 1,000 msg/sec classes. Reducing the class size to 500, 100, or 10 improves the estimation by close to 200 msg/sec; the improvement is close to 250 msg/sec when using 200 msg/sec classes. Clearly, the 10 msg/sec classes do not make much sense, as they unnecessarily increase the complexity of the tree with 4,000 possible outputs, but have a performance similar to that of the other classifications. All classifications provide an improvement greater than their quantization error, except the one with 500 msg/sec classes, although for the latter it is close.

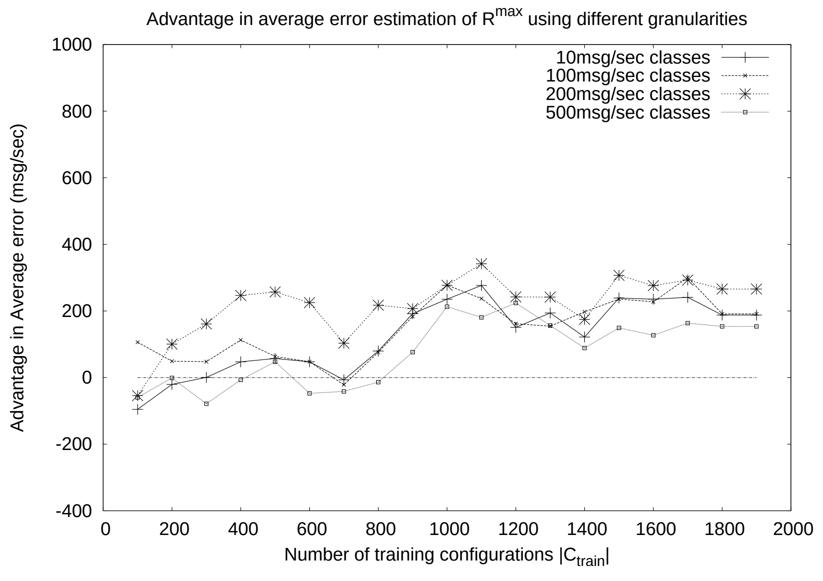

Fig. 7. Advantage in using smaller classes in the decision tree.

Fig. 8 shows the improvement using smaller classes but using the weighted average of the output classes. In general, the algorithm performs worse than when using the 1,000 msg/sec classes. One possibility is that having many more classes, fewer configurations are mapped to each class, and then the chances that a less-than-optimal class is proposed in the solution are greater.

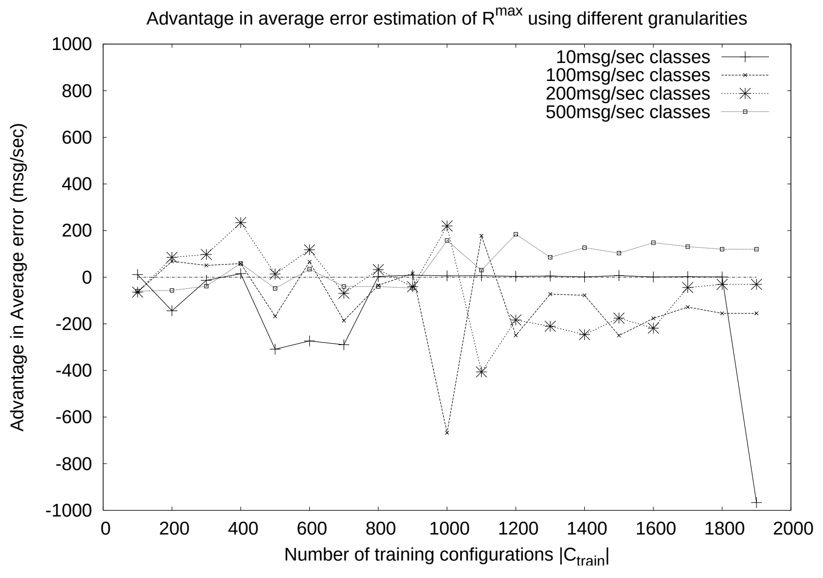

Fig. 8. Advantage in using smaller classes in the average decision tree.

C. Computational cost

Decision tree algorithms in general, and C4.5 in particular, are computationally efficient algorithms. In our approach, this computation is part of the calibration phase of the system, an off-line process where responsiveness is not an issue. However, it may be useful to recalculate the decision tree during the on-line operation of the system, as more and more configurations are being used and the resulting parameter values can be gathered. This assumes that the system changes its configuration on a fairly frequent basis. In this case, recalculation of the decision tree needs to be efficient because the resources in the system should be used by the applications running on it, not the middleware. Fig. 9 shows the time needed to compute the decision tree using different sizes of C_{train}, for different class sizes. The experiment runs the C4.5 algorithm C implementation on a Red Hat®Linux®system with an Intel®Pentium IV CPU at 2.66 GHz.

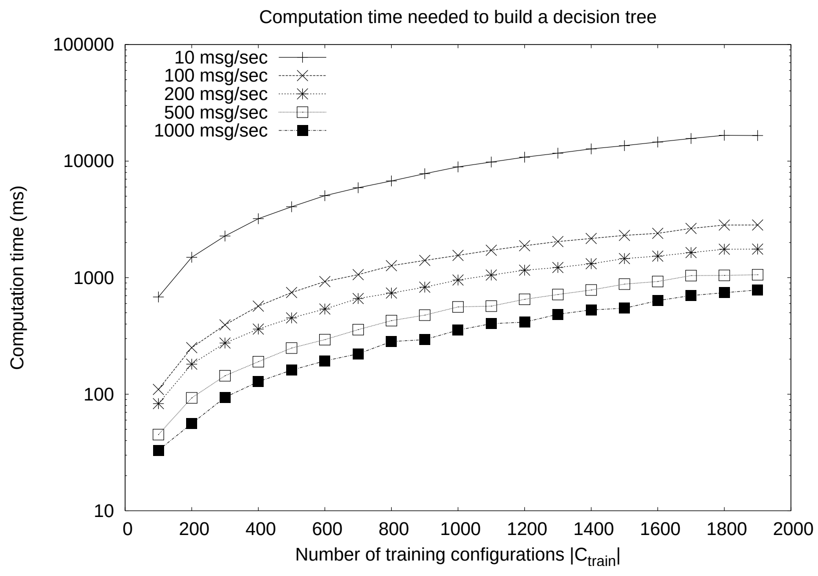

Fig. 9. Computing time needed to train a decision tree with different number of configurations, depending on the number of classes.

We observe that for the initial classification in 40 classes of 1,000 msg/sec, the time needed to compute the decision tree is under 100 ms, using up to 300 configurations. As we have previously seen, more configurations do not bring much benefit to the precision of the parameter estimation. We can then realistically recompute the decision tree on-line with new configurations gathered during operation. Increasing the number of classes increases the precision of the estimation, but only if we train the tree with enough classes (Fig. 7). This brings the computation time into the range of a few hundred millisecond. A very large number of classes (4,000 of size 10 msg/sec) increases the cost significantly (although sub-linearly), while providing no significant performance improvement over larger classes.

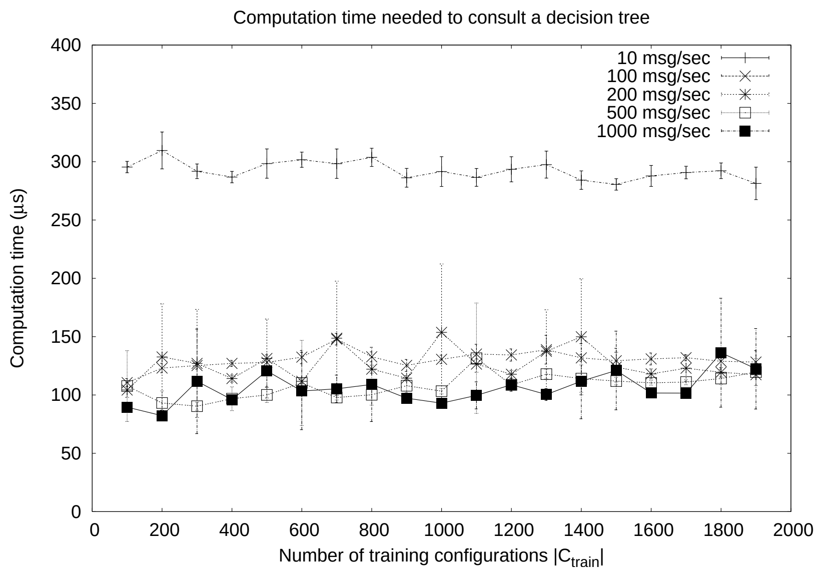

Fig. 10. Computing time needed to consult a given configuration with the decision tree with different C_{train} sizes and depending on the number of classes (confidence interval 95%).

Fig. 10 shows the time taken to query the decision tree with a configuration of 20 tasks. The time remains more or less constant, independently of the number of configurations used to train the decision tree. This is an indication that the C4.5 algorithm does good work at inferring the essence of the training data without unnecessarily increasing the complexity of the tree. The number of classes has a sub-linear influence on the query time. Only the smallest class size in our tests (10 msg/sec, which provide 4,000 output classes) has a significant impact, increasing the average query time to around 300 μs. All other classifications need around 100–150 μs per query. These figures show that the admission control mechanism should not introduce any significant impact in the operation of the entire system.

VIII. Future Work

Our next steps are to try other machine-learning algorithms, and integrate the admission control mechanism into our middleware. Ultimately, we would like to extend this mechanism to a distributed system.

IX. Conclusion

The upcoming responsive and real-time systems require the allocation of resources to tasks in an environment where, because of its complexity, it is not easy to predict which performance will result from a given resource partition. We contend that a resource allocation middleware, by offering a common execution framework for application tasks, can provide a tractable model of the system performance. This model can be used to estimate the performance of a set of tasks under a given configuration. This enables the implementation of an admission control mechanism for these complex systems, where certain tasks can be refused if the estimation is that their execution could damage the requirements of the tasks already running. The middleware model depends however on run-time parameters that can only be captured upon execution of the tasks in a configuration. We propose a calibration phase prior to deployment of the system in production, where machine-learning algorithms build a knowledge system that is used to estimate the parameters of the model. In particular, we have shown experiments with decision trees generated with the C4.5 algorithm. We conclude that, by training the system with a few hundred representative configurations, the parameter models can be estimated to a high degree of accuracy, with error below 1%.

References

[1] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] D. Bauer, L. Garcés-Erice, S. Rooney, and P. Scotton, “Toward scalable real-time messaging,” *IBM Systems Journal*, vol. 47, no. 2, pp. 237–250, April-June 2008.

[3] S. Rooney and L. Garcés-Erice, “Predicting performance on a loosely controlled event system,” in *Proceedings of 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC’08).* Turku, Finland: IEEE Computer Society, July 2008, pp. 1136–1142.

[4] J. Hyman, A. A. Lazar, and G. Pacifici, “Joint scheduling and admission control for ats-based switching nodes,” in *SIGCOMM ’92: Proceedings of the Conference on Communications Architectures & Protocols.* New York, NY, USA: ACM, 1992, pp. 223–234.

[5] J. Qiu and E. W. Knightly, “Measurement-based admission control with aggregate traffic envelopes,” *IEEE/ACM Transactions in Networking*, vol. 9, no. 2, pp. 199–210, 2001.

[6] S. Jamin, P. B. Danzig, S. Shenker, and L. Zhang, “A measurement-based admission control algorithm for integrated services packet networks,” in *SIGCOMM ’95: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication.* New York, NY, USA: ACM, 1995, pp. 2–13.

[7] J. Regehr and J. Lepreau, “The case for using middleware to manage diverse soft real-time schedulers,” in *M3W: Proceedings of the 2001 International Workshop on Multimedia Middleware.* New York, NY, USA: ACM, 2001, pp. 23–27.

[8] R. Bettati and J. W.-S. Liu, “End-to-end scheduling to meet deadlines in distributed systems,” in *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS’92)*, Yokohama, Japan, June 1992, pp. 452–459.

[9] J. He, M. A. Hiltunen, M. Rajagopalan, and R. D. Schlichting, “Providing QoS customization in distributed object systems,” in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001, pp. 351–372.

[10] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli, “QoS management through adaptive reservations,” *Real-Time Systems*, vol. 29, no. 2-3, pp. 131–155, 2005.

[11] S. Abdelwahed, S. Neema, J. P. Loyall, and R. Shapiro, “A hybrid control design for QoS management,” in *Proceedings of the IEEE Real-Time Systems Symposium*, Cancun, Mexico, December 2003, pp. 366–369.

[12] *IBM Systems Journal: Special Issue on Real-Time and Event-Based Systems.* IBM, April-June 2008, vol. 47, no. 2.

[13] J. Savolainen and A. Karhinen, “Matching service requirements to empirical capability models in service-oriented architectures,” in *Proceedings of 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC’08).* Turku, Finland: IEEE Computer Society, August 2008, pp. 1156–1161.

[14] F. Belli and M. Linschulte, “Event-driven modeling and testing of web services,” in *Proceedings of 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC’08).* Turku, Finland: IEEE Computer Society, August 2008, pp. 1168–1173.

[15] J. R. Quinlan, “Induction of decision trees,” *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[16] ——, *C4.5: Programs for machine learning.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[17] S. Ruggieri, “Efficient c4.5,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 2, pp. 438–444, 2002.

[18] P. He, L. Chen, and X.-H. Xu, “Fast c4.5,” in *Proceedings of the International Conference on Machine Learning and Cybernetics*, vol. 5, August 2007, pp. 2841–2846.

[19] L. Atlas, J. Connor, D. Park, M. El-Sharkawi, R. M. II, A. Lippman, R. Cole, and Y. Muthusamy, “A performance comparison of trained multilayer perceptrons andtrained classification trees,” in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, vol. 3, Cambridge, MA, USA, November 1989, pp. 915–920.

[20] O. Boz, “Extracting decision trees from trained neural networks,” in *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD’02).* New York, NY, USA: ACM, 2002, pp. 456–461.

[21] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian, “Traffic classification on the fly,” *SIGCOMM Computer Communication Review*, vol. 36, no. 2, pp. 23–26, 2006.

[22] J. McQueen, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1. Univ. of California Press, 1967, pp. 281–297.

[23] J. R. Quinlan, “C4.5 implementation (release 8).” [Online]. Available: http://www.rulequest.com/Personal/c4.5r8.tar.gz