Adaptive Internet Services Through Performance and Availability Control

Jean Arnaud
INRIA
Grenoble, France
jean.arnaud@inria.fr

Sara Bouchenak
Grenoble Universities
Grenoble, France
sara.bouchenak@inria.fr

ABSTRACT
Cluster-based multi-tier systems provide a means for building scalable Internet services. Building adaptive Internet services that are able to apply appropriate system sizing and configuration is a challenging objective for nowadays system administrators. This paper addresses two issues for building adaptive Internet services: (i) the control of service cost, performance and availability, three antagonist and primary aspects of Internet services, and (ii) an adaptive control of Internet services that does not shift the complexity of system administration from the Internet service to its controller.

This paper presents the design and implementation of MoKa - a middleware for controlling performance and availability of cluster-based multi-tier systems. The contribution of the paper is twofold. First, we improve an analytic model to predict the performance, availability and cost of cluster-based multi-tier applications. Second, we define a utility function and use it to build a capacity planning algorithm that calculates the optimal application configuration which guarantees performance and availability objectives while minimizing functioning cost. Finally, we propose a novel approach for dynamic provisioning of multi-tier applications that removes the burden of manual (re)configuration of the controller itself. Our experiments on the TPC-W multi-tier online bookstore show that MoKa provides significant benefits on application performance and availability.

Categories and Subject Descriptors
C.2.4 [Distributed Systems]: Client/server, Distributed applications, Distributed databases; C.4 [Performance of Systems]: Performance attributes, Reliability, availability, and service ability; D.4.8 [Performance]: Modeling and prediction, Queueing theory

General Terms
Algorithms, Management, Performance

Keywords
Multi-tier applications, Cost, Availability, Performance, SLA, Modeling, Capacity planning

1. INTRODUCTION
Data centers host a large variety of Internet services, ranging from web servers to email servers, streaming media services, and enterprise servers. These services are usually based on the client-server architecture, in which a server provides some online service to concurrent clients, such as reading web documents, sending emails or buying the content of a shopping cart. To face the increasing load of such applications, servers are organized in a multi-tier architecture. Figure 1 represents a three-tier web application which starts with requests from web clients that flow through an HTTP front-end server and provider of static content, then to an enterprise server to execute the business logic of the application and generate web pages on-the-fly, and finally to a database that stores non-ephemeral data. However, the complexity of multi-tier applications and their low rate for delivering dynamic web documents – often one or two orders of magnitudes slower than static documents – place a significant burden on data centers [10]. To face high loads and provide higher service scalability, a commonly used approach is the clustering and replication of servers in clusters of machines.

Figure 1: Multi-tier applications

The challenge in cluster-based multi-tier applications stems from the conflicting goals of, on the one hand, high performance and availability, and on the other hand, low cost and resource consumption. In the limit, high performance and availability can be achieved by assigning all available machines in a data center to a multi-tier application. Symmetrically, it is possible to build a very-low cost multi-tier application by allocating very few machines, which induces bad performance and data center downtime. Between these two extremes, there exists a configuration such that cluster-
Based multi-tier applications achieve a desirable level of service performance and availability while cost is minimized. This paper precisely addresses the problem of determining this optimal configuration. It presents MoKA - a middleware for capacity planning and provisioning of cluster-based multi-tier applications. The contributions of the paper are the following.

First, we propose a capacity planning system for cluster-based multi-tier applications that takes into account performance and availability constraints of applications. We believe that both criteria must be taken into account collectively. Otherwise, if capacity planning is solely performance-oriented for instance, this may lead to situations where only 1% of clients are admitted in the application with guaranteed performance while 99% of clients are rejected. To combine performance and availability objectives:

- We define a utility function that quantifies the performance, availability and cost of cluster-based multi-tier applications.
- We develop a capacity planning algorithm that, given SLA performance and availability constraints, calculates a configuration of the cluster-based multi-tier application that guarantees SLA constraints while minimizing the cost of the application (i.e. the number of hosting machines). The capacity planning algorithm is based on an analytic model of cluster-based multi-tier applications.
- We extend a queuing theory-based analytic model, that originally predicts application performance, with the following features: the prediction of application availability, and the handling of cluster-based multi-tier applications where each tier may consist of several replica servers.

Finally, we apply admission control and server provisioning techniques in order to reify the optimal planned capacity in the actual multi-tier application.

To the best of our knowledge, existing approaches for capacity planning of cluster-based multi-tier systems concentrate on performance requirements and do not address availability issues [1, 22, 21]. Furthermore, these approaches require manual off-line calibration of their model to determine its parameter values, and this each time application workload mix changes (i.e. client behavior changes) [22, 21]. This removes the burden of configuration from the multi-tier application, but induces new manual configuration needs at the level of the application capacity planning controller, which may be non-trivial. We believe in adaptive controllers for a practical use of modeling and capacity planning in dynamic distributed Internet services. Thus, we propose a provisioning technique for cluster-based multi-tier applications that uses lightweight monitoring in order to: (i) automatically detect workload mix changes and, (ii) dynamically recalibrate the underlying analytic model to reflect the new mix, before being able to adequately provision the application.

Finally, we implement MoKA , a middleware for modeling, capacity planning and provisioning of cluster-based multi-tier applications. The paper presents experiments conducted with MoKA on an industry standard application, the TPC-W online bookstore. The results of the experiments on a forty machines cluster show that MoKA optimizes the utility of multi-tier applications by providing significant benefits on application performance, availability and cost.

The remainder of the paper is organized as follows. Section 2 defines the necessary background. Section 3 presents design principles of the MoKA middleware for capacity planning and provisioning of multi-tier applications. Section 4 presents the results of our experiments. Section 5 provides a brief overview of the related work, and Section 6 draws our conclusions.

2. BACKGROUND

2.1 Cluster-based multi-tier systems

A multi-tier system is composed of a series of M tiers $T_1, T_2, ..., T_M$. Each tier is tasked with a specific concern. For instance, the multi-tier system in Figure 1 consists of tier $T_1$ responsible of processing the application web content, tier $T_2$ responsible of the application business logic, and tier $T_3$ responsible of the storage of non-ephemeral data of the application. When a client issues a request to a multi-tier system, the request first accesses tier $T_1$, and then may flow through successive tiers $T_2, T_3, ..., T_M$. More precisely, when a request is processed by tier $T_i$, either a response is returned to tier $T_{i-1}$ (or to the client if $i = 1$), or a subsequent request is sent to tier $T_{i+1}$ (if $i < M$).

Multiple clients may concurrently access a multi-tier system. In its basic form, each tier consists of a single server. To prevent a server from thrashing when the number of concurrent clients grows, a classically used technique is admission control [12]. It consists in fixing a limit for the maximum number of clients allowed to concurrently access a server – also known as the Multi-Programming Level (MPL) configuration parameter of servers. Above this limit, incoming requests are abandoned (i.e. rejected). Thus, a client request processed by a multi-tier system either terminates successfully with a response to the client, or is abandoned because of a server’s concurrency limit.

Moreover, a multi-tier application may face a varying workload amount and a varying workload mix over time. The workload amount denoted $N$ is the number of clients that try to concurrently access a multi-tier application. And the workload mix denoted $X$ corresponds to the distribution of different types of interactions issued by clients. Application workload variation reflects different client behaviors at different times; for instance, an e-mail service is likely to face a higher workload amount in the morning than in the rest of the day.

Multi-tier servers are hosted by machines (i.e. computing units), and a machine is exclusively owned by a server. However, for scalability purposes, a tier is usually provisioned with multiple servers in a cluster built atop replication, partitioning and load balancing techniques. In the following, we consider fair load balancing techniques. If not provisioned adequately, cluster-based multi-tier applications may face a bottleneck on one of the tiers; and bottleneck may occur at most on one tier [4]. We consider that adding more machines to an overloaded tier reduces load amount on each machine of the tier, thus improving overall performances. We also made the assumption that machines are homogeneous inside each tier of the application, as it is typically the case in computer clusters of that size [5].
2.2 Performance, availability and cost

SLA – Service Level Agreement – is a contract negotiated between clients and their service provider [13]. It specifies service level objectives (SLOs) that the application must guarantee in the form of constraints on performance and availability. Among the key metrics of interest for quantifying the performance and availability of multi-tier applications, we can cite client request latency and client request abandon rate used in the following.

The latency of a client request is the necessary time for a multi-tier application to process that request. The average client request latency (or latency, for short) of a multi-tier application is denoted as \( \ell \). A low latency is a desirable behavior which reflects a reactive application.

The abandon rate of client requests is the ratio of requests that are rejected by a multi-tier application compared to the total number of requests issued by clients to that application. It is denoted as \( \alpha \). A low client request abandon rate (or abandon rate, for short) is a desirable behavior which reflects the level of availability of a multi-tier application.

Besides performance and availability metrics, the cost of a multi-tier application is another aspect that is taken into account when optimizing the provisioning of the application. The cost of multi-tier systems refers to the economical and energetical costs of these systems. Here, cost is defined as the total number of machines that host a cluster-based multi-tier application, and is denoted as \( \omega \). Saving energy from running servers is always a good point, even if machines amount are not limited, then a low cost always preferable.

3. MOKA DESIGN PRINCIPLES

The objective of MOKA is to provision a cluster-based multi-tier application with a configuration in such a way that the performance and availability constraints are respected and the cost of the application is minimized. Thus, MOKA takes as inputs performance and availability constraints in the form of respectively latency and abandon rate limits not to exceed (i.e. \( \ell_{\text{max}} \) and \( \alpha_{\text{max}} \)). An implicit constraint is the minimization of the cost of the application (i.e. the number of machines hosting the cluster-based multi-tier application). MOKA has also exogenous inputs that are the application workload amount \( N \) and workload mix \( X \). MOKA produces as an output the configuration \( \kappa \) with which the application must be provisioned in order to guarantee the constraints.

As described in figure 2, the MOKA middleware architecture consists of four main parts, namely Monitoring, Analytic model, Model calibration, Model-based capacity planning and provisioning. In the following, we first provide some definitions, before describing MOKA sub-systems.

3.1 Definitions

We define the configuration \( \kappa \) of a cluster-based multi-tier application with a triplet \( \kappa(M, AC, LC) \), where \( M \) is the fixed number of tiers of the multi-tier application, and \( AC \) and \( LC \) are respectively the architectural configuration and local configuration of the application. We define the architectural configuration of a cluster-based multi-tier application as the distributed setting of the application in the form of the number of replica servers at each tier. It is conceptualized as an array \( AC = AC_1, AC_2, ..., AC_M \), where \( AC_i \) is the number of machines at tier \( T_i \) of the multi-tier application. We define the local configuration as the distributed setting of the application in the form of constraints on the number of tiers of the multi-tier application, and \( AC \) is the fixed number of tiers of the multi-tier application, and local configuration of the application. We define the local configuration as the fixed number of tiers of the multi-tier application, and local configuration of the application. We define the local configuration as the fixed number of tiers of the multi-tier application, and local configuration of the application. We define the local configuration as the fixed number of tiers of the multi-tier application, and local configuration of the application. We define the local configuration as the fixed number of tiers of the multi-tier application, and local configuration of the application.

A noticeable difference can be determined either as a relative value (e.g. ±10%) or an absolute value (e.g. ±50 clients).
space limitation, we are not able to fully detail the MVA model and the proposed extended version. In the following, we briefly describe the features added by the proposed extensions.

First, the analytic model is now able to represent cluster-based multi-tier applications where several server replicas may exist per tier, and a fair load-balancing applies to distribute the load between replicas. This results in an additional input to the original model that is the architectural configuration $AC < AC_1, ..., AC_M$ of the multi-tier application. Second, the model is now able to estimate client request abandon rate of the application (as an indicator of availability), where admission control applies at the entry of each tier server and requests may therefore be rejected. This implies adding $LC < LC_1, ..., LC_M$ local configuration (i.e. MPLs) of the multi-tier application to original model inputs, and abandon rate $\alpha$ to original model outputs.

Furthermore, the workload mix given as an input of the model is represented by a triplet $X(TT, V < V_1, ..., V_M >, S < S_1, ..., S_M >)$, where $TT$ is the client think time, $V < V_1, ..., V_M >$ are tiers visit ratios, and $S < S_1, ..., S_M >$ are service times at the tiers [19]. Client think time is the average time between the reception of a response by a client and the sending of the next request by that client. Visit ratios reflect how much client requests visit the tiers of the application. Indeed, when a client request enters tier $T_1$, the request may generate sub-sequent requests to tier $T_2$, tier $T_3$, and so on until tier $T_M$. The visit ratio $V_i$ is the average number of sub-sequent requests at tier $T_i$ generated by a client request entering the multi-tier application. Service times times correspond to the incompressible amount of time necessary to process requests on each tier of the application. Thus, service time $S_i$ is the average incompressible time necessary to process a client request on tier $T_i$. $X(7s, V < 1, 2, 3, S < 5ms, 8ms >)$ is an example of a workload mix where client think time is 7 s, where each request on tier $T_1$ induces in average 2 requests on tier $T_2$ (i.e. visit ratios), and where the incompressible request service time is of 5 ms on tier $T_1$ and 8 ms on tier $T_2$.

### 3.4 Model calibration

In the original MVA model, an initial stage, called model calibration, is necessary to use the model. It consists in determining the application workload mix parameters $X(TT, V < V_1, ..., V_M >, S < S_1, ..., S_M >)$ that are given as an input of the model. To automate this phase, MoKa includes monitoring mechanisms in order to calibrate the model. These mechanisms are based on interception techniques that apply at the entry of servers of the front-end tier for measuring $TT$, and at the entry of servers of each tier $T_i$ for measuring $V_i$ and $S_i$.

In the current prototype of MoKa, an implementation of the monitoring mechanism is proposed for monitoring Java EE multi-tier applications, the industry standard for building Java multi-tier applications. Monitoring in MoKa applies AOP (Aspect-Oriented Programming) techniques - and AspectJ implementation - in order to intercept the information that characterizes application workload mix [8]. Indeed, Java EE defines standard APIs for the entry point of each tier of multi-tier applications, such as HTTP Servlet API for the Web and business tier and JDBC API for the database tier. MoKa makes use of these standard APIs in order to capture the time when a request enters a tier, the time when it leaves it, and the interval between the two. This allows to measure service times $S < S_1, ..., S_M >$ and visit ratios $V < V_1, ..., V_M >$. Client think time $TT$ is measured as the interval between the time when a client request terminates and the time when the next request is made by the same client (i.e. inside the same HTTP session). The data are monitored on the different servers of the multi-tier application, then collected on a central node. A moving average is calculated in order to produce the average values of $(TT, V < V_1, ..., V_M >, S < S_1, ..., S_M >)$ that characterize a workload mix $X$.

Moreover, workload mix parameters must be determined with a low workload amount of the application [19]. This is especially true for estimating service times $S < S_1, ..., S_M >$ which values must not include overhead due to a high load of the application. Thus, the proposed model is automatically calibrated at application start-time with the parameters of the initial workload mix, and then recalibrated each time the workload mix changes. And whenever a (re-)calibration of the model is triggered, this must be done on a lightly-loaded application. However, a workload mix change may occur whereas the application faces a high workload amount. A first approach to face this issue consists in forcing the application to lower its workload amount through aggressive admission control for a period of time in order to monitor new values of $(TT, V < V_1, ..., V_M >, S < S_1, ..., S_M >)$ and recalibrate the model with the new mix $X$. Another approach consists in dividing the machines hosting the multi-tier application into two sub-sets, one for model recalibration and one for continuing serving clients during recalibration. The former sub-set is small (e.g. consisting of $M$ machines, one per tier) and handles a small amount of clients (denoted by $N'$). The latter consists of the remaining machines (e.g. $\sum AC_i - M$) and handles the remaining clients $(N - N')$. While the second approach for model recalibration maintains higher service availability than the first approach, it requires specific load-balancing in order to balance $N'$ clients to the first sub-set of machines and the remaining clients to the second sub-set. Experiments presented in Section 4 follow the former approach.

### 3.5 Capacity planning

Once the model correctly calibrated, and if the workload varies, MoKa performs capacity planning in order to calculate the new configuration $\kappa$ of the multi-tier application that respects performance and availability constraints while minimizing cost, i.e. maximizes utility. In the following, we first define our utility function, before describing the proposed utility-aware capacity planning.

#### 3.5.1 Utility function

Given performance constraint $\ell_{max}$ and availability constraint $\alpha_{max}$ that a multi-tier application must guarantee, we define Performability Preference (i.e. performance and availability preference) as follows:

$$PP(\ell, \alpha) = (\ell \leq \ell_{max}) \cdot (\alpha \leq \alpha_{max}) \quad (1)$$

where $\ell$ and $\alpha$ are respectively the actual latency and abandon rate of the multi-tier application. Note that for $\forall \ell, \forall \alpha, PP(\ell, \alpha) \in \{0, 1\}$, depending on whether Eq. 1 holds or not.

Based on performability preference and cost of the multi-tier application, we now define a utility function that com-
bines both criteria as follows:

$$\Theta(\ell, \alpha, \omega) = \frac{M \cdot PP(\ell, \alpha)}{\omega}$$

(2)

where $\omega$ is the actual cost (i.e. #machines) of the multi-tier application, and $M$ is the number of tiers of the multi-tier application. $M$ is used in Eq. 2 for normalization purposes. Here, $\forall \ell, \forall \alpha, \forall \omega, \Theta(\ell, \alpha, \omega) \in [0, 1]$, since $\omega \geq M$ (at least one server per tier) and $PP(\ell, \alpha) \in [0, 1]$.

A high value of the utility function reflects the fact that, on the one hand, the multi-tier application guarantees service level objectives for performance and availability, and on the other hand, the cost underlying the multi-tier application is low.

### 3.5.2 Utility-aware capacity planning

#### Algorithm 1: Capacity planning of multi-tier applications

**Input:**
- $N$: workload amount (i.e. #clients)
- $X(T, T', V < V_1, \ldots, V_M >, S < S_1, \ldots, S_M >)$: workload mix

**Output:**
- $\kappa^\ast(M, AC < AC_1, \ldots, AC_M >, LC < LC_1, \ldots, LC_M >)$

**Parameters:**
- $\ell_{\text{max}}$: maximum latency
- $\alpha_{\text{max}}$: maximum abandon rate
- $MO$: underlying model algorithm

**Initialization:**
- $\kappa^\ast(M, AC < AC_1, \ldots, AC_M >, LC < LC_1, \ldots, LC_M >)$
- $\ell_{\text{opt}}$: optimal latency
- $\alpha_{\text{opt}}$: optimal abandon rate

**Algorithm:**
1. $\ell_{\text{max}}$ maximum latency
2. $\alpha_{\text{max}}$ maximum abandon rate
3. $MO$: underlying model algorithm
4. $\ell_{\text{opt}}$: optimal latency (i.e. local configuration)
5. $\alpha_{\text{opt}}$: optimal abandon rate
6. $\kappa^\ast(M, AC < AC_1, \ldots, AC_M >, LC < LC_1, \ldots, LC_M >)$
7. $\ell_{\text{opt}} = \ell_{\text{max}} - (1 - \alpha_{\text{max}})$; $\alpha_{\text{opt}} = 0$;
8. $\ell_{\text{opt}}$: optimal latency (i.e. local configuration)
9. $\ell_{\text{opt}} = 1$; $\ell_{\text{opt}}$: optimal latency (i.e. local configuration)
10. $LC_\text{opt} = N' \cdot V_m$; $\ell_{\text{opt}}$: optimal latency (i.e. local configuration)
11. $o^\ast = 0$;
12. $\kappa^\ast = \theta, \theta > 0$;
13. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
14. $\ell_{\text{max}}$: maximum latency
15. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
16. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
17. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
18. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
19. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
20. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
21. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
22. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
23. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
24. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
25. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
26. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
27. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
28. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
29. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
30. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
31. $\ell_{\text{opt}} = \ell_{\text{max}} + \ell_{\text{opt}}$
32. return $\kappa^\ast(M, AC < AC_1, \ldots, AC_M >, LC < LC_1, \ldots, LC_M >)$

We propose a utility-aware capacity planning method based on the above utility function to calculate the optimal architectural and local configuration of a cluster-based multi-tier application in such a way that the application performance and availability preference for latency and abandon rate is guaranteed while the cost of the application is minimized.

Calculating the optimal configuration of a multi-tier application is thus equivalent to calculating the configuration for which the utility function value is maximal, (i.e. optimal, $\Theta^\ast$).

Algorithm 1 describes the proposed capacity planning for cluster-based multi-tier applications. It takes as inputs the application workload amount $N$, the workload mix $X$; and produces as output the application configuration that optimizes utility. The algorithm uses the performance and availability preference parameters $\ell_{\text{max}}$ and $\alpha_{\text{max}}$, and the model denoted by $MO$.

Roughly speaking, given an application workload and a target performability preference, the capacity planning algorithm builds an initial minimal configuration of the multi-tier application, then calculates the performance and availability of this configuration based on the model, and tests the estimated performance/availability against performability preference. If performability preference is verified, the capacity planning algorithm returns that configuration and terminates; otherwise, it iterates on a new augmented configuration of the multi-tier application and repeats the process.

More precisely, the algorithm builds an initial configuration with a minimal architectural configuration (for minimal cost) and the corresponding local configuration in such a way that availability preference based on $\alpha_{\text{max}}$ is guaranteed (cf. Algo. 1, lines 8-13). In the initial configuration, each tier is provisioned with a single server. Admission control is applied based on the maximum abandon rate applied to incoming workload, which produces the workload $N'$ actually entering the system. At each tier, the server's MPL (i.e. local configuration) is simply assigned the number of entering client requests at that tier; this is calculated based on entering workload and visit ratios parameters (cf. Algo. 1, line 11). The capacity planning algorithm then follows two successive stages in order to guarantee performance and availability objectives and to minimize cost. In a first stage (cf. Algo. 1, lines 16-19), a configuration that verifies performability preference of Eq. 1 is rapidly built, though that configuration may not be minimal regarding cost. This first stage simply consists in adding server machines to all tiers until performance and availability objectives are met. At each tier, servers MPL is assigned based on fair load balancing among servers of that tier. Thus, starting from that configuration, the second stage of the algorithm maximizes the utility of the multi-tier system through cost minimization (cf. lines 22-31). To do so, it applies a dichotomic approach for an efficient search of the minimal number of servers at each tier (i.e. architectural configuration). The local configuration is calculated based on load balancing among servers of a tier. In Algo. 1-line 24, the number of concurrent requests that globally enter all replicas of tier $T_m$ is by definition $N' \cdot V_m$. Thus, with load balancing between replicas, the number of requests that enter one replica at tier $T_m$ is $\frac{N' \cdot V_m}{T_m}$. Finally, the algorithm produces an application configuration that verifies performance and availability preference while minimizing cost, thus, a configuration that maximizes utility. As a result, and based on the produced configuration, two techniques are used for reifying the configuration in the application: a technique that applies provisioning through server allocation/deallocation and a technique that uses server admission control to prevent thrashing.

\footnote{Though correct, the way MPL (i.e. local configuration) is calculated here might be sensitive to workload amount variation since it is calculated based on a given $N$; and it might incur frequent MPL reconfigurations. An improved calculation of the MPL consists in augmenting its value to the maximum as long as performability preference is verified.}
4. EVALUATION

4.1 Experimental setup

4.1.1 Testbed application

The evaluation of the extended analytic model and utility-aware capacity planning and provisioning techniques has been conducted using the TPC-W benchmark [20]. TPC-W is an industry standard benchmark from the Transaction Processing Council that models a realistic web bookstore. TPC-W comes with a client emulator which generates a set of concurrent clients that remotely access the bookstore application. They emulate the behavior of real web clients by issuing requests for browsing the content of the bookstore, consulting the best-sellers, etc. The client emulator generates a tunable workload; this allows us to vary the workload amount and workload mix during the experiments. In our experiments, the on-line bookstore was deployed as a cluster-based two-tier system, consisting of a cluster of replicated web/business servers as a front-end and a cluster of replicated database servers as a backend.

4.1.2 Software and hardware environment

Our experiments have been conducted on a set of computers organized as follows: a first computer dedicated to the client emulator, a front-end cluster of replicated servers running Apache Tomcat 5.5 web and application server, and a back-end cluster of replicated servers running MySQL 5.0 database server. The cluster-based multi-tier TPC-W application running with MoKa as deployed as follows. First, aspect weaving was applied to the code of the TPC-W application in order to automatically integrate monitoring features as discussed in Section 3. Second, a proxy-based approach was followed in order to implement admission control, and load balancing among the replicas of a tier. Here, load balancing is able to dynamically integrate/remove replicas upon server (un-)provisioning. The experiments were conducted on x86-compatible machines with bi-2.0 GHz AMD Opteron CPUs and 2 GB RAM, connected via a 10 Gb/s Ethernet LAN.

4.2 Model validation

We first conduct experiments to validate the accuracy of the extended analytic model and its ability to reflect the behavior of the cluster-based multi-tier application. In particular, we evaluate the ability of the model to reflect the variation of latency and abandon rate as the application workload amount varies. Thus, for the same set of input variables, the performance and availability reified by the model is compared with the actual performance and availability of the real system. Figure 4 describes workload variation, both in amount and mix, used in the experiments. Here, two application mixes are considered Mix1 and Mix2; the former is the TPC-W Browsing mix, and the latter is a heavier version of the Browsing mix. Mix2 involves more requests being sent to tier 2 (then a larger visit ratio) and an increased service time on tier 2.

Figure 3 presents the evolution over time of respectively client request latency and abandon rate, for both the real system and the modeled system, when the workload varies according to Figure 4. For comparison with the base system, we use two configurations: κ1 (2, AC < 2, 5 >, LC < 150, 100 >) and a larger configuration κ2 (2, AC < 4, 10 >, LC < 800, 1000 >). The results show that the model is able to render the behavior of the real system, for both latency and abandon rate.

4.3 Capacity planning evaluation

In this section, we evaluate the proposed utility-aware capacity planning techniques. Here, we consider a performance constraint limiting the maximum client request latency to 500 ms, and an availability constraint fixing the limit of client request abandon rate to 5%. Thus, 95% of client requests are handled by the application in less than half a second. The role of the capacity planning is to guarantee these constraints while minimizing the cost, through feedback provisioning and admission control.
Figure 3: Real system vs. modeled system

Figure 6: Local configuration of the controlled system

Figure 7: Latency in presence of control

Figure 8: Abandon rate in presence of control

5. RELATED WORK

Capacity planning and provisioning are critical issues for the performance and availability of Internet services [14, 15, 6]. While the improvement of server performance and availability is usually achieved by system administrators using ad-hoc tuning [2, 17], new approaches tend to appear to ease the management of such systems.

Server admission control (i.e. MPL control) is extensively studied in server systems. Elnikety et. al. apply it at the level of a web server [9], Milan-Franco et. al. use it in database servers [18], whereas Menascé et. al. study it at each tier of a multi-tier system [16]. Some admission control solutions are proposed in the form of heuristics such as the well-known hill-climbing heuristic [16]. Other solutions are based on analytic models for quality-of-service guarantees [11]. Diao et. al. use an analytic model to control MPL in multi-tier systems [7]. Zhang et. al. propose a regression-based modeling of multi-tier applications in order to predict the capacity of applications in terms of allowed concurrent clients [23].

Other approaches control the dynamic provisioning of servers in cluster-based systems. Autonomic provisioning of database servers is presented in [5], and dynamic server provisioning in multi-tier systems is described in [1]. While these systems are based on heuristics, other approaches tend to better characterize multi-tier applications through analytic
modeling. Villela et al. follow a model-based approach for provisioning the business tier in a multi-tier system [22]. Uurgaonkar et al. apply an analytic model calibrated for the underlying workload mix in order to plan the capacity of multi-tier systems in terms of the number of servers to provision at each tier [21]. Both approaches are motivated by performance objectives; however they require calibrating the model with appropriate parameters.

In summary, our present work differs from other projects in many aspects: (i) it performs capacity planning at all tiers of a multi-tier system since the bottleneck tier may differ from one workload to another, (ii) it combines performance with availability objectives, (iii) it handles both workload amount and workload mix variations without requiring a manual recalibration of the model, (iv) it combines admission control with server provisioning for a better usage of resources, and (v) it follows a model-based approach that maximizes the utility of applications, i.e. guarantees performance and availability constraints while minimizing cost.

6. CONCLUSION

In this paper, we present MoKa, a middleware for adaptive modeling, capacity planning and dynamic provisioning of clustered multi-tier applications. The proposed method includes four novel features: (i) A utility function that takes into account performance and availability objectives, and combines them to the cost of multi-tier applications; (ii) A utility-aware capacity planning algorithm that calculates configuration of multi-tier applications that guarantees performance and availability constraints while minimizing application cost of the application; (iii) The extension of a queuing theory-based analytic model that predicts application performance with features for the prediction of application availability in cluster-based multi-tier systems; (iv) An adaptive model calibration that dynamically determines model parameters in order to handle both workload amount and workload mix variations. Our experiments on an online bookstore show that MoKa provides significant benefits on application performance and availability.

7. REFERENCES


