Towards Autonomic Hosting of Multi-tier Internet Applications

Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen
Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
Email: {swami,gpierre,steen}@cs.vu.nl

Abstract

A vast amount of caching and replication solutions have been proposed in the literature to improve the performance of multi-tiered Web applications (which we call as Internet services). These solutions aim to alleviate the scalability bottleneck of only a single tier and different techniques are suitable for services of different nature. However, from the view point of an administrator who wants to host a service scalably, it is not easy to determine the right set of techniques to apply. This leads to either gross over-provisioning of resources or poor performance. We believe that the decision process of choosing the right techniques for a service can be automated. To strengthen our position, we propose the design of an autonomic hosting system that uses a combination of multi queue models and online simulations to achieve our goals. Even though our work is very much in progress, we believe the techniques used in our system can provide a good start in taming the complex problem of scalable hosting of services.

1 Introduction

E-commerce enterprises such as yahoo.com, amazon.com and ebay.com often use complex software systems to serve content to millions of Web clients. For instance, the Web page generated in response to each client request to a Web site like amazon.com is not generated by a single application but by a large number of smaller Web applications operating in parallel. These enterprises build their software systems out of many such Web applications, usually called *services*. Services are applications that perform certain business logic and are exposed through well defined client interfaces usually accessible over the network. Examples of service include order processing services and shopping cart services.

Typically, a service consists of business logic which makes certain queries to a data store and request(s) to other services to generate a response to its clients. Typically, the enterprise systems of e-commerce companies (such as Google, Amazon and Yahoo) run hundreds of these services. These services are typically hosted across thousands of inexpensive PCs in multiple data center(s) possibly located across a wide-area network [3]. Moreover, enterprises usually assign performance and availability goals, commonly known as service level agreement (SLA), to each service individually.

In this paper, we focus on the problem of hosting a service efficiently so that it can meet its SLA. A generalized service hosting architecture is shown in Figure 1. As seen in the figure, deploying a service usually involves replicating its code to a number of application servers and its data to an array of data store machines. Furthermore, different caching layers such as for service response caching and database caching can be deployed to improve performance. Various research efforts have been carried out on each of these solutions [14, 15, 4, 10, 5, 11, 2]. However, all these works aim to alleviate bottlenecks only in one tier of a given service.

For a given service, it is not trivial for an administrator to determine the best set of techniques and the number of nodes to provision for each of them. The goal of our work is to build a system that autonomically provisions the hosting platform for a service and determines the right configuration of techniques to apply so that it can meet its performance goals with *minimal usage of servers*. For instance, for a given service our system automatically determines which set of techniques will help in improving its performance. This problem is challenging due to four reasons. First, multiple techniques per service need to provisioned. Second, the effect of provisioning on performance depends on the characteristics of the service it hosts. For example, service caching is beneficial only if the requests to a service exhibit good temporal locality. Third, provisioning a tier does not always result in a linear performance gain. For example, caches tend to follow the law of diminishing returns (i.e., the benefit of increasing the number of cache servers decreases after a certain threshold). Fourth, the access patterns (e.g., request rate, update rates, and temporal locality of requests) can change continuously.

Our position in this paper is follows: For scalable hosting of a service, there exists a wide range of caching and replication solutions that can be applied at different tiers of a service. We believe that the decision regarding what

are the right set of techniques to apply for a given service and how to provision each of them can be automated. To strengthen our position, we present the prototype design of our system that employs a combination of queueing models and on-line cache simulations to estimate the performance gain of adding/removing a resource in a tier. The proposed system, unlike solutions solely based on queueing models, is "*cache-aware*", which means that it takes into account temporal access patterns of requests.

The rest of this paper is organized as follows. Section 2 describes our system model and our generalized hosting architecture. Section 3 discusses the design of our autonomic hosting system. Section 4 presents the related work and Section 5 concludes the paper with a list of important open issues to be addressed.

2 Background

2.1 System and Application Model of a Service

We assume that each service is assigned a performance and availability goal (usually referred to as an SLA). For sake of simplicity, we will restrict ourselves to performance-related SLAs in this paper. We define the SLA of a service such that its average response time should be within the $[LowRespTime, HighRespTime)$. Typically, large scale e-commerce enterprises deploy their software system across multiple data centers to attain high performance and availability. In this paper, we assume that the resources allocated to a service (and all its tiers) is located within a single data center. Replicating a service across multiple data centers is assumed to be done by replicating the service (and all its tiers) as a whole, so that there is no inter data center communication between the tiers of a service. Furthermore, we assume that each data center has a pre-allocated rescue pool of resources. When a service hosted in a data center does not meet its SLA, the autonomic hosting system adds one or more resources from this pool to the service to ensure that it meets its desired performance. However, a service cannot use all the resources in the pool as the pool is shared with other services hosted in the data center.

2.2 Generalized service hosting architecture

Our generalized service hosting architecture is given in Figure 1. As seen in the figure, there are various techniques that can be applied at different tiers of a service to improve its performance. These techniques include server-side and client-side response caching (e.g., [15, 1]) done at tier 0 and 2^a respectively, business logic replication done at tier 1, database caching (e.g., [4, 2]) done at tier 2^b and database replication at tier 3 (e.g., [10, 5]). We do not explain

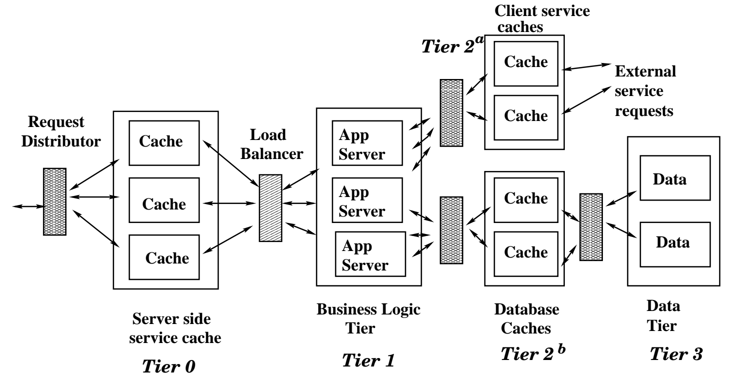

Figure 1. Generalized hosting architecture for a service

these techniques in detail due to space constraints. Interested readers are referred to [12].

Each tier of a service can be provisioned with one or more servers (zero or more for caching tiers). This requires the use of a load balancer to route the requests and/or share the load among replicas in a tier uniformly. In our system, we assume the presence of hardware load balancers (e.g., CISCO GSLBs) for the business logic tier and use a request distributor (e.g., [8]) for routing requests across servers in a caching tier.

Caching service responses (at tier 0 and tier 2^a) and database query responses (at tier 2^b) introduces the problem of consistency maintenance. To simplify the process of consistency maintenance, we assume that the request workload to the database of a service (and also to other services) consists of a fixed set of read and write query templates. A database query template is a parameterized SQL query whose parameter values are passed to the system at runtime. This scheme is deployed, for example, using Java's prepared statement. In our system, we expect the developer to specify a priori which query template conflicts with which update template. For example, consider the following template: $QT1$: "SELECT price, stock, details from book where id=?" and its conflicting update query template $UT1$: "Update price=price+1 from books where id=?". In this example, when a server caches a query result that is an instance of $QT1$, then it subscribes to receive update events generated by $UT1$. Ofcourse, template-based invalidations can be extended to include the parameter of the query template (so that a query to update the price of book "XYZ" only invalidates the instance of $QT1$ with parameter "XYZ". We use the same template-based invalidation technique for service response caching.

3 Design of an adaptive hosting system

The goal of our adaptive hosting system is to ensure that the service is continuing to meet its SLA even when its

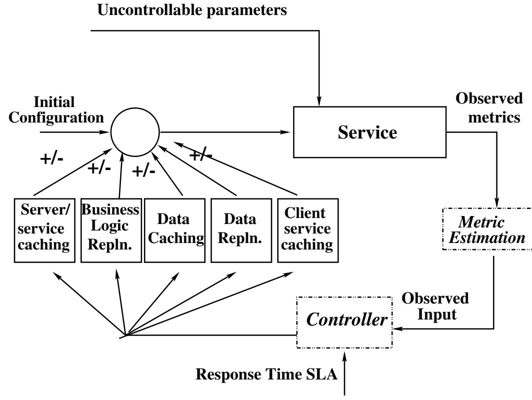

Figure 2. Logical Design of an Adaptive Hosting System for Internet Services

workload (e.g., request rate, temporal locality of requests, or update ratio) may change. In such a scenario, the system needs to take appropriate action by increasing or decreasing the amount of resources allocated to one (or more) of the service tiers in Figure 1. In a sense, we can see this system as a continuous feedback system as the one shown in Figure 2. The system must detect changes in performance (using a metric estimation system) and trigger the controller to change the resource configuration (i.e., the number of servers dimensioned for each tier) to bring the average response time back to the acceptable interval.

The controller is the decision-making element that is responsible for provisioning each tier in a given service. The controller answers the following the question: *what is the right resource configuration for a given service that can help to meet its SLA with a minimum number of servers being used for hosting it?* To design such a controller, we must first be able to estimate the response time of a service for a given resource configuration. This would enable the controller to determine the response time if extra servers are added (or removed) in different tiers and select the best configuration. In this section, we propose a way to model the response time of a service and a method to estimate the gain in response time when a server is added (or removed) in each of the tier. Finally, we explain how this model is used to make a decision to adapt the resource configuration of a service.

3.1 Modeling end-to-end latency of a service

An internet service is multi-tiered. Each incoming request is first received by the first tier which in turns serves the request locally and/or can trigger calls to other tiers. Let us consider a tier T_i that receives a request which can be serviced locally with a probability p_i and/or can trigger multiple requests to more than one tier. Let K_i be the set of tiers that T_i calls for servicing its incoming requests, i.e., $T_j \in K_i$ if T_j is called by T_i. For example, in Figure 1, T_1 makes requests to T_{2^a} and T_{2^b}, so $K_1 = \{T_{2^a}, T_{2^b}\}$. Let $N_{i,j}$ denote the average number of requests sent to T_j by T_i for serving a single incoming request to T_i. For example, if a single request to the business logic tier (T_1) results in 1.5 queries to the data cache tier (T_{2^b}), then $N_{1,2^b} = 1.5$. Now, the average response time to service a request at T_i, $RespTime_i$ is given by:

$$RespTime_i = Q_i + p_i * ExecTime_i + \sum_{j \in K_i} N_{i,j} * RespTime_j \quad (1)$$

where Q_i is the average queueing latency experienced by a request at T_i before being serviced and $ExecTime_i$ is the average time taken by tier T_i to execute the request (and does not include the response times of other tiers). Note that equation 1 can capture the response times of tiers with different characteristics (e.g., caching or computational). For example, for a server-side caching tier (T_0), p_0 denotes the average cache hit ratio, $N_{0,1} = 1 - p_0$ (each request to cache goes to T_1 only if it is a cache miss) and $K_0 = \{T_1\}$ (as all outgoing requests of T_0 are always sent to T_1). For the business logic tier, $p_1 = 1$ as all services always have to do some business logic computation, and $K_2 = \{T_{2^a}, T_{2^b}\}$ as the business logic can make requests to the external service tier (T_{2^a}) and data tier (T_{2^b}).

We can then perceive a service as a 4-tiered system, whose end-to-end response time can be obtained from equation 1 as follows:

$$\begin{aligned} RespTime_0 = Q_0 + p_0 * ExecTime_0 + (1 - p_0)* \\ (Q_1 + ExecTime_1 + N_{1,2^a} * RespTime_{2^a} \\ + N_{1,2^b} * RespTime_{2^b}) \end{aligned} \quad (2)$$

where $RespTime_{2^a}$ and $RespTime_{2^b}$ are the average response time for client-side service caching and database caching tiers respectively. The equations for the variables can be similarly derived from equation 1.

3.2 Service Characterization

In our model, parameters such as p_i, $ExecTime_i$ and $N_{i,j}$ characterize the service and its workload. To estimate the average response time of a service, these parameters must be measured. Here, we discuss how the system can accomplish such measurements.

3.2.1 Estimating different parameters

To estimate the response time of a service, we need to know the execution time ($ExecTime_i$), p_i, and $N_{i,j}$ ($\forall T_j \in K_i$) of each tier. Most of these values can be obtained by instrumenting the cache managers and application servers appropriately. For example, $ExecTime$ of caches can be obtained by instrumenting the cache manager appropriately so that the average latency to fetch an object from cache can be logged. Measuring $ExecTime$ for business logic

tier is harder because mere instrumentation at the business logic tier can only obtain the average time to service a request at the application service, i.e., $RespTime_1$. However, $RespTime_1$ not only includes the computational time at application server ($ExecTime_1$) but also the response time of data ($RespTime_{2^b}$) and external service tiers ($RespTime_{2^a}$). So, to obtain $ExecTime_1$, we need to measure the values of $RespTime_1$, $RespTime_{2^a}$ and $RespTime_{2^b}$ during low loads (to avoid queueing latency). Using these values $ExecTime_1$ can be obtained subsequently.

3.3 Impact of adding/removing server at each tier

An SLA violation can occur when one (or more) tier(s) face a bottleneck that may occur due to a change in its workload. For example, if the cache hit ratio at T_0 decreases (due to low temporal locality or increased number of invalidations) then the request rate to the business logic tier will increase thereby leading to increased response time. In such a scenario, the controller need to answer the following questions: (i) Will provisioning extra server(s) at any of the tiers improve the response time? and if so, (ii) what is the best tier to provision the resource(s)?

To answer these questions, the controller needs to automatically estimate the relative goodness in adding a machine at each tier and choose the tier that gives the highest benefit as the one to provision the resource. Adding an extra resource, i.e., a server, to a tier can reduce the queueing latency as the tier has more processing capacity. Furthermore, for caching tiers, this can improve the hit ratio as the caches have more memory capacity. We describe the process of estimating these gains below.

In general, estimating the improvement in queueing latency by adding another server is easier. Queueing latency, Q_i is given by Little's law as $Q_i = rps_i * ExecTime_i$. In this model, the reduction in queueing latency by adding a new server to a tier that has n servers is: $(rps_i * ExecTime_i) * (1/n - 1/(n+1))$ (note that execution time is not affected if we assume that servers are homogeneous).

3.3.1 Estimating improvement in cache hit ratio

For caching systems, execution time and queueing latency are not the main bottleneck and cache hit ratio is the crucial factor. The amount of memory available for the caches has a direct impact on the cache hit ratio. Estimating the improvement in cache hit ratio when a new server is added is not trivial. Before we present the details of hit ratio estimation, recall that we design our distributed cache using consistent hashing [8].

In our distributed cache, we estimate the possible gain in hit ratio due to addition of a new server using the following technique. Let us assume the memory limit of each cache server is M. Each cache server stores only the list of objects whose size jointly does not exceed M and keeps statistics about its cache hit ratio. In addition to this, the cache manager at each cache server maintains a *virtual cache* list that will hold the identifiers of objects that the server would have cached if it had $M + \Delta$ memory and its corresponding virtual cache hit ratio. The hit ratio of the virtual list is what the server would have obtained if it had been given an extra Δ memory for caching. So, if the caching tier runs N cache servers and Δ is set to M/N, then the average of virtual cache hit ratios of all servers is the possible hit ratio the distributed cache would obtain when an extra M memory is added to it. This is equivalent to adding another server with memory M to the distributed cache. Of course, this estimation is valid only if the requests are distributed uniformly across caches which is the case in our distributed cache.

The cache hit ratio of the virtual cache list is used by the controller to compute the gain in response time due to addition of another server to the distributed cache (using equation 2). Similarly, the possible degradation in response time due to removal of a server from distributed cache can be estimated by maintaining another virtual list in each cache server with a $M - \Delta$ memory threshold. Note that a similar technique of virtual cache list was used in [18].

3.4 Decision process

When the system faces an increase in observed end-to-end response time ($RespTime_0$) beyond the $HighRespTime$ threshold set by the SLA, then the controller needs to adapt. The controller can use one or more servers from the rescue pool to bring the response time back to the acceptable interval. However, the controller must first decide on the best tier to add the new server. To do that, the controller obtains values of $ExecTime_i$, p_i and $N_{i,j}$ for each tier from the metric estimation system. For caching tiers, it also obtains the estimated cache hit ratio for $M + \Delta$ memory. With these values, the controller computes $RespTime_0$ when a server is added to each tier and selects the one that offers the least response time as the tier to add the new resource. This process is continued until the response time falls within the acceptable interval or until the rescue pool is exhausted.

Continuous addition of servers without appropriate scaling down can lead to over-provisioning of resources. To avoid this scenario, the controller must periodically check if the observed response time is lower than the $LowRespTime$ threshold. If so, the service is probably over provisioned. To avoid that, the controller estimates the increase in response time if one (or more) server(s) is removed in any of the tier. Subsequently, it chooses to remove a resource from the tier that offers the lowest estimated response time, provided the estimated value is within the acceptable interval set by the SLA.

4 Related Work

A vast number of solutions have been proposed in the literature for improving the performance of Web applications. These include techniques such as fragment caching (e.g., [6]), XML caching (e.g., [15]), database caching (e.g., [4, 2, 13]) and database replication (e.g., [10, 11]). All these techniques are studied independently and aim to address the bottleneck at different tiers of a service. In contrast, we aim to build a hosting system that automatically chooses the right set of techniques to apply for a given service based on its individual characteristics and to determine the right amount of resources to provision for each tier. The problem studied in this paper is closely related to capacity provisioning and has been well studied in the context of single-tiered applications [9, 7]. A simple transposition of these techniques to our problem is however not suitable as database, business logic and service caches have very different characteristics. Hence, it is imperative to treat each individual tier as a separate entity. A recent work studied the problem of provisioning a 3-tier web site using multi-queueing models [17]. Unfortunately, the study does not include any caching techniques (such as client/server-side service caching or database caching) in its model. This is a very limiting approach as caching is one of the widely used techniques used in boosting the performance of a service.

5 Conclusion and Future Work

A vast number of techniques exist for scalable hosting of networked services. However, the choice of right techniques and number of resources to provision in each tier depends on the characteristics of the individual service. In this paper, we have presented the initial design of a system that performs autonomic hosting of internet services. Our system employs a combination of queueing models and online cache simulations to decide on the right resource configuration to use for a given service. We have implemented our prototype in Tomcat and the Axis[1] platform with a PostgreSQL database backend. We believe the techniques used in our hosting system can help multi-tiered internet services in handling sudden changes of workload that may arise due to events such as flash-crowds.

As a next step, we plan to test our prototype with TPC-APP [16], a service-oriented application benchmark, to validate our model and to refine it further. Subsequently, we plan to look into the issue of proactive adaptation and taking into account the availability SLAs.

References

[1] Akamai Edgesuite Architecture, `http://www.akamai.com/en/html/services/edge_proc_targeting.html`.

[2] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: a dynamic data cache for web applications. In *Proceedings of International Conference on Data Engineering*, pages 821–831, 2003.

[3] L. Barroso, J. Dean, and U. Hlzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, Mar 2003.

[4] C. Bornhvd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *Data Engineering*, 27(2):11–18, June 2004.

[5] E. Cecchet. C-JDBC: a middleware framework for database clustering. *Data Engineering*, 27(2):19–26, June 2004.

[6] J. Challenger, P. Dantzig, and K. Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Transactions on Internet Technology*, 4(4), Nov 2004.

[7] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Web server software architectures. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 2003.

[8] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceeding of the eighth international conference on World Wide Web*, pages 1203–1213, 1999.

[9] D. A. Menasce. Web server software architectures. *IEEE Internet Computing*, 7(6):78–81, 2003.

[10] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the International Middleware Conference*, Toronto, Canada, Oct. 2004.

[11] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: Autonomic data replication for web applications. In *Proceedings of the 14th International World-Wide Web Conference*, Chiba, Japan, may 2005.

[12] S. Sivasubramanian, G. Pierre, and M. van Steen. Towards autonomic hosting of multi-tier internet applications. Technical Report IR-CS-021, Vrije Universiteit, Amsterdam, The Netherlands, May 2006.

[13] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. GlobeCBC: Content-blind result caching for dynamic web applications. Submitted for publication, Oct. 2005.

[14] S. Sivasubramanian, M. Szymaniak, G. Pierre, and M. van Steen. Replication for web hosting systems. *ACM Computing Surveys*, 36(3), Sept. 2004.

[15] J. Tatemura, O. Po, A. Sawires, D. Agrawal, and K. S. Candan. Wrex: A scalable middleware architecture to enable xml caching for web services. In *Proceedings of the International Middleware Conference*, Grenoble, France, Nov. 2005.

[16] Transaction Processing Performance Council. TPC benchmark app (application server). `http://www.tpc.org/tpc_app/default.asp`.

[17] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 291–302, 2005.

[18] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proc. of the USENIX Annual Technical Conference*, pages 161–175, 2002.

[1] `http://ws.apache.org/axis/`