Towards Autonomic Hosting of Multi-tier Internet Applications

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

Technical report IR-CS-021, Vrije Universiteit, May 2006.

Abstract. Large scale e-commerce enterprises like Yahoo and Amazon use complex software systems made of hundreds of Internet services to serve content to millions of clients. These services are multi-tiered Web applications that perform certain business logic and are exposed through well-defined client interfaces usually accessible over the network. A constant challenge faced by these organizations is to host these services in a scalable fashion to meet performance goals. A vast amount of research has been done by the community on caching and replication solutions that aim to improve performance of a service by addressing the bottlenecks at its different tiers (e.g., database and presentation tiers). However, different techniques are suited for different kinds of services and it is not easy for an administrator to choose the best set of techniques for a given service. Our position in this paper is as follows: We believe that the decision process of choosing the right techniques for a service requires analysis of service and workload at each tier and that to a large extent this process 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.

The application model of one typical service is shown in Figure 1. As shown in the figure, 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. An example of a service calling another service can be the product catalog service of an e-commerce shopping store calling the top-sellers service to display the top sellers in the "Science Fiction" category, while displaying details for the book "Hitchhikers Guide to the Galaxy." These software systems can be viewed as an acyclic forest of services.

Typically, the enterprise systems of e-commerce companies (such as Google, Amazon and Yahoo) are hosted across thousands of inexpensive PCs in multiple data center(s) possibly located across a wide-area network [3]. Hosting such complex software systems so that performance and availability goals are met is a challenging problem. Instead of managing the performance at a global level, enterprises usually assign performance and availability goals, commonly known as service level agreement (SLA), to each service individually. Subsequently, the problem of SLA management of the whole software system breaks down to managing the SLAs of individual services. In this paper, we focus on the problem of maintaining the SLA of an individual service.

To achieve the requested performance, each service is often deployed across multiple machines. 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. Although these techniques have been introduced independently from (and often as alternatives to) each other, we believe that they complement each other and may need to co-exist to obtain the best performance. While much research has been carried out on each of these solutions [21, 22, 13, 12, 26, 19, 2], all these works aim to alleviate bottlenecks only in one tier of a given service. However, different techniques are suited for different kinds of services. For example, if requests to the service exhibits high request locality, service caching might be beneficial. If the underlying data retrieval is a bottleneck, then database caching or replication might be useful depending on the temporal locality of database queries. Sometimes, a combination of these techniques might be needed to achieve a certain SLA.

We propose a generalized service-hosting architecture that combines various such solutions in different tiers to improve the performance of a given service, as shown in Figure 2. As seen in the figure, a wide variety of caching and replication solutions exist to enable scalable hosting of a given service. However, for a given service, it is not trivial for an administrator to determine the best set of techniques and the number of nodes to dimension for each of them. The goal of our work is to build a system that autonomically dimensions[1] 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 for four reasons. First, multiple techniques per service need to dimensioned. Second, the effect of dimensioning on performance depends on the characteristics of the service it hosts. For example, service caching

[1] Dimensioning problem is also known to as provisioning problem.

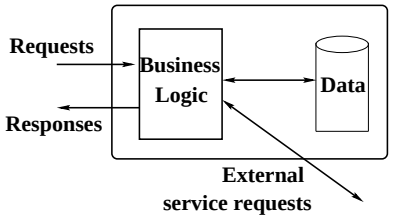

Figure 1: Application Model of an Internet Service

is beneficial only if the requests to a service exhibit good temporal locality. Third, dimensioning 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 dimension each of them can be automated. To strengthen our position, we propose a generalized service hosting architecture that covers the wide range of proposed solutions that aim to improve the performance of a Web application. Even though the individual elements of the architecture has been studied independently extensively, very few works have studied them in entirety. Moreover, we present our prototype design of an autonomic hosting system that adapts to the workload of a service it hosts. It identifies the right set of caching and replication techniques (and the best number of servers to be allocated for these tiers) among those described in our generalized architecture for each service.

Our system 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. In contrast, traditional queueing systems assume that adding a server at any given tier always decreases the queueing time thereby leading to reduced latency. However, such an assumption is not true for caching tiers.

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

The application model of a service is shown in Figure 1. As seen in the figure, a service consists of business logic that receives the requests and makes (zero or more) queries to a database and (zero or more) requests to other services to generate appropriate responses. Usually, the business logic is hosted in an application server. The data tier can use a relational DBMS or object stores such as [9, 13, 15]. In our work, we focus on relational DBMS-driven services. 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

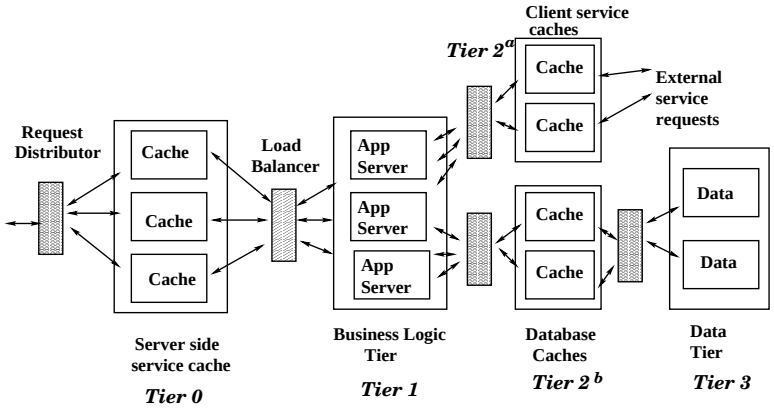

Figure 2: Generalized hosting architecture for a service

average response time should be within the $[LowRespTime, HighRespTime)$[2].

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.[3] 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 2. There are various techniques that can be applied at different tiers of a service to improve its performance. For sake of completeness, we will explain each of these techniques briefly in this section.

Server-side service response caching (tier 0) is a technique that is becoming increasingly popular to improve the scalability of Internet services. The principle behind this technique is simple: the system stores the response to a service request (identified by requested method name and parameters), so that future identical requests are not forwarded to the business logic tier. The cached responses will be usually in the form of XML messages if the service is implemented as a Web service (e.g., [22]). This technique usually improves the throughput of the service as a cache hit in this layer reduces the load on the business logic tier (and other subsequent tiers). This technique is beneficial if the service requests exhibit a good temporal locality.

Business logic replication (tier 1) is a technique used when business logic computation is the bottleneck. An example of a computationally intensive business logic is the page generation logic of an e-commerce site (that combines the responses from multiple services using XSL transformation to generate an HTML page). Usually, replicating business logic translates to running multiple instances of the application server (across multiple servers). This technique is simple provided the hosted business logic is stateless and keeps all the state in the data tier.

Client-side service response caching ($tier\ 2^a$) is used to cache the responses of the requests made by the business logic to the external services. The external services can reside on other data centers possibly across a wide-area network. In such a scenario, deploying a

[2]We do not look at 99 percentiles of response time and restrict ourselves to average response time as a starting point.

[3]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.

client side service response cache can be beneficial as it alleviates the network bottleneck (such as network congestion). Furthermore, client-side caching can be useful if the external service is not meeting its SLA due to temporary failures or poor provisioning, and if the requesting service has no immediate control over them (e.g., if the external service is run by a different department or organization).

Database caching ($tier\ 2^b$) is a technique used to reduce the load on the data tier. Database caching systems store the response of database queries and maintain the consistency of cached query responses. Examples of database caching systems include DBProxy [2] and DBCache [5]. Database caching techniques are useful if the database queries exhibit high temporal locality. In our architecture, we use our in-house database caching solution, GlobeCBC [20].

Database replication ($tier\ 3$) is a widely studied technique to alleviate the database bottleneck. Data are replicated usually to achieve better performance and/or higher availability [10] and several solutions exist for replicating relational DBMSs [17, 19, 12]. If we restrict ourselves to performance, data replication is more beneficial than caching if the database queries exhibit poor locality [20]. However, if the underlying database receives a huge number of updates, then the benefit of database replication reduces due to consistency maintenance.

2.3 Request Distribution and Cache consistency

Each tier of a service can be dimensioned 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., [11]) 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. A cached query response (or service response) might become inconsistent when the underlying database of the service gets updated. 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 invalidations to $QT1$. When the data tier receives an update query that is an instance of $UT1$, it sends out a message to invalidate all instances of $QT1$. 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 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 2. In a sense, we can see this system as a continuous feedback system as the one shown in Figure 3. The system must detect changes in performance (using a metric

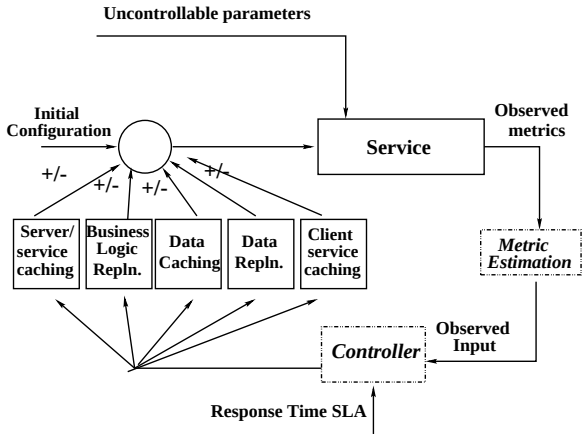

Figure 3: Logical Design of an Adaptive Hosting System for Internet Services

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 dimensioning 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

As seen in Figure 2, a 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., $T_j \epsilon K_i$ if T_j is called by T_i. For example, in Figure 2, 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:

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

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). [4]

[4]Modeling execution times of a database is tricky as read and write queries have different characteristics. As a starting point, we stick to simply modeling average query execution times. We defer refinements to future research.

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:

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

where $RespTime_{2^a}$ and $RespTime_{2^b}$ are the average response time for client-side service caching and database caching tiers respectively. The average response times of these tiers are given as:

$$RespTime_{2^a} = Q_{2^a} + p_{2^a} * ExecTime_{2^a} + (1 - p_{2^a}) * (RespTime_{ext}) \tag{3}$$

$$RespTime_{2^b} = Q_{2^b} + p_{2^b} * ExecTime_{2^b} + (1 - p_{2^b}) * (Q_3 + ExecTime_3) \tag{4}$$

where $RespTime_{ext}$ is the average response time of requests made to external service(s).

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 $RespTime_{2^b}$ and using these values $ExecTime_1$ can be obtained subsequently.

3.2.2 Synchrony vs. Asynchrony

The above equations in general assume that requests from one tier to another are synchronous. While this assumption is usually true for interactions from caches to business logic (or from database caches to databases), it is not necessarily true for calls made by the business logic tier. Applications running in the business logic tier make (multiple) calls to the data tier or external services and it is possible that some calls maybe asynchronous. In such a case, equation 1 is not valid as it assumes all calls are made in a synchronous fashion.

Let us consider the scenario where all the calls made by T_i to other tiers in K_i are asynchronous. In such a case, the response time for requests made to other tiers is limited by the slowest call and its average response time is given by:

$$RespTime_i = Q_i + p_i * ExecTime_i + \max_{j \epsilon K_i}(N_{i,j} * RespTime_j) \quad (5)$$

A similar equation can be envisaged for a hybrid scenario mixing synchronous and asynchronous calls. However, to identify the nature of calls from one tier to another, we have built separate wrappers for synchronous and asynchronous calls and require the developer to use them for making requests to other tiers. These wrappers help the metric estimation system in collecting the average response time of these calls.

3.3 Estimating 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.

3.3.1 Estimating improvement in Q_i

In general, the queueing latency, Q_i is given by Little's law as the product of the number of requests that arrive at T_i per time unit, rps_i, and the time taken to execute them, $ExecTime_i$ [24].

$$Q_i = rps_i * ExecTime_i \quad (6)$$

In this model, the improvement in queueing latency by adding a new server can be estimated as follows: Let us assume that T_i runs n servers and requests are distributed uniformly. By adding another server in T_i, Q_i is reduced from $(rps_i * ExecTime_i)/n$ to $(rps_i * ExecTime_i)/(n+1)$ (note that execution time is not affected if we assume that servers are homogeneous).

3.3.2 Estimating improvement in cache hit ratio

For caching systems, execution time and queueing latency are not the main bottleneck. On the other hand, 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. For this problem, traditional queueing systems that use process sharing discipline are not useful as adding a new server to a distributed cache does not result in improving the hit ratio linearly. Hence, it is imperative for the system to estimate the possible gain in cache hit ratio that can be obtained by adding a new server to its distributed caching layer.

Before we discuss the details of hit ratio estimation, we note that we design our distributed cache using consistent hashing [11]. In such a design, a request to an object X is always routed to a server whose id is numerically close to a hash value of X, $f(X)$,

and only that server can store the object. This design avoids redundant storage of objects across multiple servers and can also be optimized to balance the load among the cache servers [11]. In such a system, 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. We experimented this technique to estimate the improvement in cache hit ratio of our database query caching system, GlobeCBC and the initial results validate our approach.

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 (over time) can lead to over-provisioning of resources (thereby increasing operational costs of the system). 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., [7, 1, 14]), XML caching (e.g., [22]), database caching (e.g., [5, 2, 20]) and database replication (e.g., [17, 19, 12, 6]). All these techniques are studied independently and aim to address the bottleneck at different tiers of a service. The objective of our work is neither to propose an improvement nor an alternative to any of these techniques. Rather, 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.

Our problem is closely related to capacity provisioning and has been well studied in the context of single-tiered applications [16, 8]. 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 [25]. 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: Current Status and Open Issues

A vast number of techniques exist for scalable hosting of networked services. Our position is that many of these techniques are not in conflict with each other and the right configuration depends on the characteristics of the individual service. Furthermore, we believe that the decision regarding the right configuration of a service and how many resources to dimension for each of them can be automated. To this end, 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 on-line 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[5] platform with a PostgreSQL database backend. The database caches are implemented at the JDBC layer and service caches are implemented at the axis web service layer. 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 [23], a service-oriented application benchmark, to validate our model and to refine it further. Our proposed system has still many open issues that are being currently addressed or to be addressed in the future. Some of the important open issues among them are:

- *Proactive adaptation:* The design in this paper talks about how to adapt when we see a SLA violation. However, it would be desirable to predict significant changes in workload well in advance so that SLA violations can be minimized. This requires good prediction models such as [18, 4].
- Global SLAs to local SLAs: One of the fundamental design choices we made is that each service has an individual SLA. However, from the view point of an organization, it has a global end-to-end customer SLA needs to be translated to SLAs of individual services. This requires modeling of the complete software system and is a very challenging problem.
- *Modeling database:* As noted earlier, modeling the response times of a relational database are hard as the read and write queries have different characteristics. Furthermore, replication makes it harder as each update will result in updates to all replicas. This is still an open issue and we are looking for precise models to address this issue.
- *Availability SLAs:* Another important criterion for each service to be met is availability. Usually, availability is met by redundancy at each tiers assuming certain failure

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

models (of the server failures, data center failures etc.). We would like to include these requirements also in our model in the future.

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] Y. Baryshnikov, E. G. Coffman, G. Pierre, D. Rubenstein, M. Squillante, and T. Yimwadsana. Predictability of web-server traffic congestion. In *Proceedings of the Tenth IEEE International Workshop on Web Content Caching and Distribution*, pages 97–103, Sept. 2005.

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

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

[7] 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.

[8] 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.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25(2):173–182, 1996.

[11] 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.

[12] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 156, Washington, DC, USA, 1998.

[13] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, 2000.

[14] W.-S. Li, O. Po, W.-P. Hsiung, K. S. Candan, and D. Agrawal. Engineering and hosting adaptive freshness-sensitive web applications on data centers. In *Proceedings of the Twelfth international conference on World Wide Web*, pages 587–598, 2003.

[15] B. Ling, E. Kiciman, and A. Fox. Session state: Beyond soft state. In *Proc. of Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, San Francisco, CA, USA, Mar 2004.

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

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

[18] J. Rolia, X. Zhu, M. Arlitt, and A. Andrzejak. Statistical service assurances for applications in utility grid environments. *Perform. Eval.*, 58(2+3):319–339, 2004.

[19] 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.

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

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

[22] 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.

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

[24] K. S. Trivedi. *Probability and statistics with reliability, queuing and computer science applications*. John Wiley and Sons Ltd., Chichester, UK, UK, 2002.

[25] 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.

[26] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings of 19th IEEE Symposium on Reliable Distributed Systems*, 2000.