Autonomic Mechanisms for Service Scaling in Wide-Area Shared Computing Environments[1]

Vinit Padhye and Anand Tripathi
Department of Computer Science
University of Minnesota Minneapolis, 55455 Minnesota USA

Abstract

In this paper we present mechanisms and models for building autonomically scalable and resilient services in wide-area shared computing environments. Wide-area shared computing platforms as exemplified by PlanetLab provide a cooperatively shared pool of computing resources. Building services in such environments poses unique challenges due to the fluctuations in available resource capacities and node availability. Autonomic mechanisms for building scalable and resilient services in such environments need to appropriately scale the service capacity under fluctuating available resource capacities, node crashes, and fluctuations in client load. Towards addressing these issues, we have developed models and mechanisms for estimating the service capacity under varying available resource capacities through online micro-benchmarking. The mechanisms for autonomic scaling of service capacity perform dynamic replication and regeneration of service replicas based on the estimated service capacity and observed load. This requires selection of appropriate nodes for the placement of new replicas. Furthermore due to the fluctuations in resource capacities, the load distribution for such services need to adapt to the varying service capacities of the individual service replicas. For this purpose we develop mechanisms for adaptive load distribution. We present here the evaluation of these mechanisms through experiments conducted over the PlanetLab environment.

1 Introduction

We present here the results of our investigation of models and mechanisms for building resilient and scalable services using cooperatively shared pool of computing resources over the Internet. Our study is conducted on the PlanetLab platform, which exemplifies such environments. In such an environment, typically a large pool of globally distributed computing resources is available that can be used for building resilient and scalable services. However, a number of challenges arise due to the intrinsic characteristics of such environments. The availability of a node and its resource capacities are not guaranteed at any time. The available resources at a node are allocated to different users on a fair-share basis. A node may shutdown at any time, and the available computing capacity at a node may fluctuate significantly over a short time [8, 18] because of the load imposed by other users and applications running on that node. In building a service over such an environment, the requirements of resiliency and scalability both demand a service to be replicated at different nodes in the network, and the degree of replication needs to be dynamically controlled for scalability under widely varying operating conditions.

For building scalable services using a globally shared infrastructure such as the PlanetLab, several challenging problems need to be addressed. The available service capacity may diminish abruptly due to the crash or shutdown of a replica's node. Moreover, the load generated by the clients typically changes with time, and sometimes it may change quite significantly over a short duration, as characterized by the phenomenon of *flash crowds*. Building a scalable service requires dynamic provisioning of service capacity to meet the current load demands by maintaining an appropriate number of service replicas to handle the client requests. The service deployment mechanisms need to ensure that the aggregate service capacity provided collectively by the service replicas is sufficient to handle the current and projected load conditions, but at the same time there should not be over-provisioning beyond some level. This requires dynamic creation of new replicas or shutting down of some existing ones, and dynamic adjustment of load distribution. The deployment of a new service replica requires selection of a suitable node for hosting it. Such a node needs to be picked based on its available resource capacity and the likelihood that it would remain suitable for service hosting for some time in the near future.

[1] This work was supported by National Science Foundation grants 0834357 and 0708604

In order to determine the aggregate service capacity, the service deployment mechanisms need models to estimate the request handling capacity provided by each service replica at any given time. The service capacity of a replica depends on the available capacities of resources such as the CPU cycles, memory, network bandwidth, and I/O rates at its node. The available resource capacities can fluctuate significantly in a short time, such as a few minutes, due to the presence of other users. Because the service replicas have different service capacities, which typically fluctuate, we need adaptive and agile mechanisms to distribute the client requests according to the capacities of individual replicas. The distribution of client requests to different service replicas needs to be determined dynamically based on the capacities of all service replicas.

We present here the design and evaluation of the mechanisms that we have developed for addressing the above problems for building autonomically scalable services. We evaluate and demonstrate the performance of these mechanisms on the PlanetLab environment. Specifically, our contributions are in the following areas.

First, we develop models for estimating the request handling capacity of a service replica by using online microbencharking of the workload and then using this information in relation to the currently available resource capacities at its host. The workload characteristics and resource usage are continually monitored for this purpose. We develop models for predicting the available resource capacity at a node. For making the service capacity estimation, we use techniques based on operational analysis models [7] to determine the bottleneck resource at any given time.

Second, we develop mechanisms for dynamic provisioning of the service capacity based on the load condition. We present here a model for dynamic control of the degree of service replication. This model utilizes the estimated service capacity of each replica to estimate the aggregate service capacity. This model is based on the notion of *capacity slack*, and it aims at maintaining in the system some target amount of overall slack capacity to deal with fluctuations in the load and the aggregate service capacity. This model also strives to avoid over-provisioning by shutting down some service replicas if the slack capacity exceeds certain limit.

Third, we develop mechanisms for adaptive distribution of client requests to different service replicas based on the estimated service capacities of the individual replicas. We present here mechanisms that operate at two levels. At a relatively coarse level, a distributed hash table (DHT) is used to direct client requests to different replicas according to the desired load distribution. The second set of mechanisms operate at the level of replicas. Here, each replica assesses it own load, and dynamically determines if it needs to redirect some of the client requests to other replicas.

For selecting a node for the deployment of a new service replica, we have developed an infrastructure to monitor a subset of the PlanetLab nodes for their available resource capacities. Some of the initial results of our observations in the characterization of the resource availability of the PlanetLab nodes were presented in [18].

In the next section we present an overview of the framework which we have developed over the PlanetLab platform for experimental evaluation of the models and mechanisms for building autonomically scalable services. Section 3 presents the models we have developed for microbenchmarking, capacity prediction and service capacity estimation. In this section, we also present our evaluation of the capacity estimation models. In section 4 we present the models and mechanisms for dynamic service capacity management. Section 5 describes the mechanisms for adaptive load distribution. Section 6 presents the results of our experiments over the PlanetLab environment to evaluate the mechanisms for dynamic service capacity management and load distribution. The related work is discussed in Section 7, and conclusions are presented in the last section.

2 A Framework for Deploying Replicated Services

The research presented here was conducted using the experimental framework that we presented in [18] for building migratory services on the PlanetLab environment. We extended this framework to include mechanisms for adaptive load distribution and resource-aware load-dependent replication control, which we present in this paper. A service is implemented by a group of replicated service agents. Each service agent is implemented as a mobile agent using the Ajanta system [19, 17]. A mobile agent is capable of autonomously migrating to another host in the network. The architecture of the service deployment framework is shown in Figure 1.

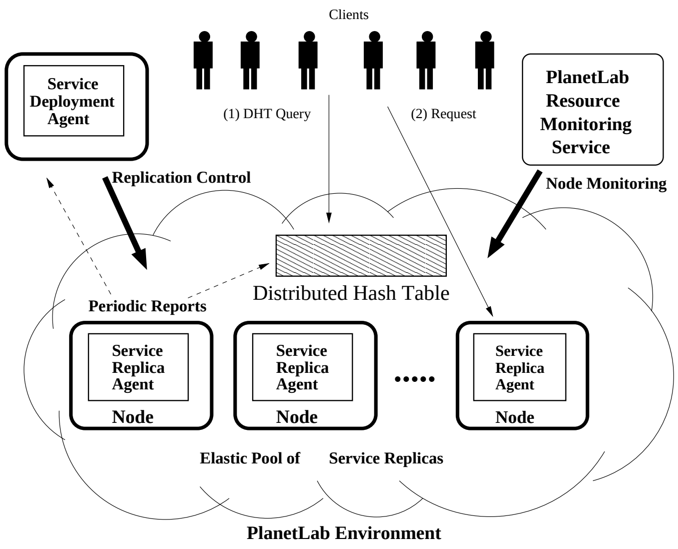

Figure 1: Service Deployment Framework

Each service in our system is assigned a unique service-id (SID). A DHT is used to maintain for each service a record containing its deployment configuration information, such as the list of all replicas, their network addresses, and their currently estimated service capacities. The record for a service is queried or updated using the SID of the service. The DHT is implemented using the Free Pastry system [12]. The DHT maintains the network location information of all replica agents of a registered service, and it also has information indicating for each replica the fraction of the load that should be directed to it for the desired load distribution across the system. The primary function of the DHT is to direct a client to one of the replicas. DHT directs clients to service replicas in proportion to their estimated service capacity. The load distribution fractions may need to be changed continuously because of the fluctuating availability of resource capacities at the replica agents' hosts. For this, each service agent periodically reports its estimated service capacity to the DHT. The number of replicas, their network locations, and the corresponding load distribution information can change with time. Service replica agents may be added or removed for a service.

In order to access a service, a client first queries the DHT, providing the SID of the service, to find one of the service replicas. The DHT randomly chooses one for the replicas, with the probability of selecting a replica being proportional to its load fraction. It returns to the client this agent's name, its IP address, and the port number. The client then sends its requests directly to this service agent. If at any time it is unable to contact the service agent, it once again queries the DHT. In response to a client request, a service agent sends one of the two types of responses: *success*, or *redirection*. In case of redirection, the client is directed to send its request to another service agent; the response contains the name and network address of that agent. The details of the redirection policies for load distribution are discussed later in Section 5.

The service replicas monitor their operating conditions and make autonomic decisions such as shedding client load by redirecting the clients to other service replicas. We present in the next section the mechanisms used by a service agent for monitoring the available resource capacities and estimating the service capacity. We utilize for this purpose the *SliceStat* [10] data provided by the PlanetLab host, indicating the resource utilization by different slices at that host. Slices represent different users on the system, and a service agent is executed under the privileges of a particular slice. A service agent contains mechanisms to monitor its load. We present in Section 5 the models and mechanisms that we have developed for a service replica to assess whether it is underloaded or overloaded in relation to its estimated service capacity. A token-based model is developed for this purpose.

In this framework a special agent, called *Deployment Agent (DA)*, is used for creating and launching the replica agents of a service. This agent is responsible for controlling the pool of the service replicas in the system to maintain certain level of aggregate resource capacity in the system in relation to the currently observed load condition. It strives to maintain a service capacity slack in the system. The Deployment Agent obtains from each service replica agent periodic status reports. It includes the number of requests served by it during the current reporting period and its estimated service capacity for the next period. It also includes per request resource utilization information obtained by continuous monitoring of the resource utilization of the service agent. The Deployment Agent continually monitors the load conditions and the aggregate service capacity. We develop in Section 4 the *capacity slack* model which determines if the aggregate idle resource capacity is at least equal to some target threshold in relation to the current load as reported by the service agents. If not, then it adds additional service capacity by launching new service agents on some PlanetLab nodes that have sufficient idle resource capacity. It is also responsible for detecting the crashes of any service replicas and a creating new ones to ensure the desired slack capacity in the system.

In order to find and select a suitable node for hosting a service agent, the Deployment Agent needs to find a node on the PlanetLab with sufficient idle resource capacity. For this we utilize the node monitoring service which is presented in [18]. This service periodically monitors a large set of nodes and maintains the following node-level statistics: average utilization and free capacity based on 10-second observations over a sliding window of 5 minutes, the standard deviation of these average utilization values, and exponentially weighted utilization.

3 Service Capacity Estimation

Accurate estimation of service capacity at the replica level is crucial for appropriately scaling the aggregate service capacity in the system. The request handling capacity of a particular replica at a given time is based on the available capacities of the resources on the replica's node and the average resource usage demand of a request. The estimation of service capacity is needed to be done continuously since the available resource capacities and the workload characteristics may change significantly with time. Our model for service capacity estimation is based on the following three aspects, utilizing operation analysis techniques [7]: First, we estimate the average resource usage demand of a request through online micro-benchmarking of requests. Second, we develop a model for predicting the available resource capacities at a node in the near future based on the node's recent behavior. Finally, we estimate service capacity of the replica based on the predicted available resource capacity and the per request resource usage demand.

3.1 Online Micro-benchmarking

For estimating the average resource usage demand of a request, each service replica performs continuous monitoring of its resource usage and workload. In our experimental system over the PlanetLab environment, a service replica collects resource usage information of different types of resource such as CPU, memory and bandwidth by probing the *SliceStat* service on the PlanetLab. These probes are performed at 15 seconds interval by the service replica to obtain information about its own slice and also for other slices executing on its host. This information is used for determining the service replica's own resource usage and the unused resource capacities. A service replica also monitors its workload to maintain information about the number of requests served over the past one minute observation interval. Each service replica maintains the following statistics for resource usage at interval i.

- c_i Average CPU usage (measured in MHz) of replica's own slice over the past 1 minute window. It is calculated by observing the percentage CPU usage of replica's own slice and the node's intrinsic CPU capacity measured in MHz. In determining a nodes intrinsic CPU capacity we take into consideration the number of cores and CPU speed.

- m_i Average memory usage (measured in MB) of replica's own slice over the past 1 minute window

- b_i Average bandwidth usage (measured in KBps) of replica's own slice over the past 1 minute window

For monitoring the workload characteristics, the following statistics are maintained for the $i'th$ interval:

- s_i Number of requests served per second
- t_i Average service time per request
- n_i Average amount of per request data communication over the network (number of bytes)

The above information is used for characterizing the average resource usage demand per request, as follows.

- D_{ci} Per request CPU consumption, measured in terms of the number CPU cycles required to service a request. This is given by:

$$D_{ci} = c_i / s_i \tag{1}$$

- D_{mi} Per request incremental memory consumption, measured in MB. This is the incremental amount of memory required for handling a request since some base amount of memory B_m is always used by a service replica. The amount of base memory usage is given by the memory usage under no load condition. We calculate the incremental per request memory requirement as follow:

$$D_{mi} = (m_i - B_m)/s_i \tag{2}$$

The average values of the above per-request resource demand measures computed over the past five minutes are represented by D_c and D_m. Similarly, per request network usage D_n is obtained based on the five minute average of the n_i values. These values are used for estimating the service capacity in the near future, as detailed below.

3.2 Prediction of Available Resource Capacity

Since available resource capacity at a node may fluctuate significantly, the currently available capacity may not be an accurate estimate of the available capacity in the near future. For this purpose, we develop models for predicting resource capacity that is likely to be available in the near future with high probability. We then use this predicted resource capacity for estimating the service capacity. Our model for capacity prediction is based on developing heuristics that take into consideration the average available resource capacity observed in the recent past and determine what fraction of it is likely to be available with high probability in the near future. The heuristics developed here are independently applied to different resources types such as CPU and network bandwidth.

We developed these heuristics by collecting traces of a large collection of PlanetLab nodes and observing how the available resource capacity at a node changes with time. These traces were collected by probing PlanetLab nodes every 10 seconds. For each 1 minute interval we calculated the average available capacity over the past 5 minutes and observed the available capacity in the next 1 minute. Our goal was to observe what fraction of the currently available capacity (observed over the 5 minute window) remains available in the next 1 minute with a given confidence level. For developing the desired heuristics we analyzed the distribution of the ratio of the capacity available in the next 1 minute to the average available capacity over the last 5 minute. We refer to this ratio as *conditional idle capacity ratio*. Based on the observation of 309 PlanetLab nodes for a duration of four hours, in Figure 2 we show the complement of the cumulative distribution of the probability for different values of this ratio for available CPU capacity. For a given value on the x-axis, it gives the probability that this ratio would be larger than or equal to that value. Since this ratio may also depend on whether the node is heavily loaded or lightly loaded, we show this distribution separately for the three different classes of load conditions of the observed nodes: heavily loaded, moderately loaded, and lightly loaded. Table 1 shows how node were characterized in these categories with respect to their CPU and network usage.

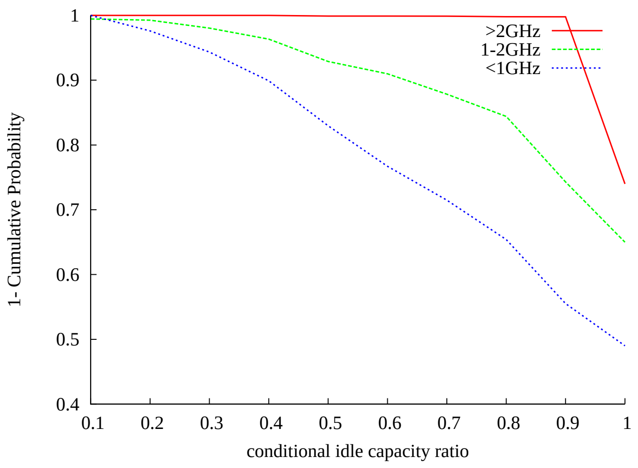

Figure 2: Conditional Availability of CPU Capacity

We can see that the 90 percentile value for heavily loaded nodes (i.e., those with available CPU capacity less than 1GHz) is 40%. which indicates that at least 40% of the currently available CPU capacity was available in next minute with 90% probability. For lightly loaded nodes, at least 95% of the currently available CPU capacity was available in the next minute with 90% probability.

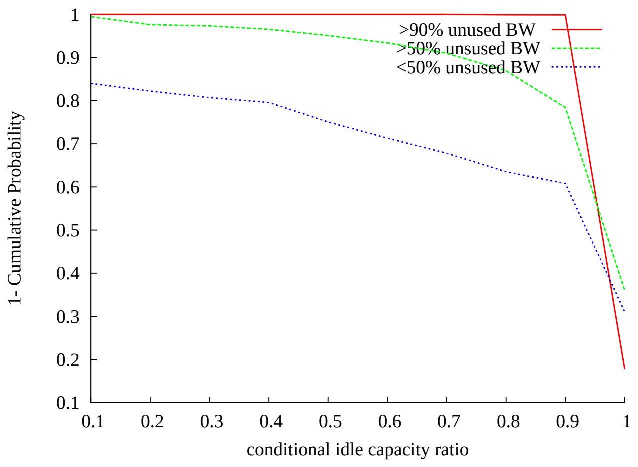

Figure 3: Conditional Availability of Network Bandwidth

Figure 3 shows the same data for bandwidth capacity. These 90-percentile values can then be used as cutoff points to determine how much capacity can be assumed with 90% confidence level to be available in the next minute. Based on this observed data, we take the cutoff points for different categories of nodes as given in Table 1 . We show here cutoff values for CPU and bandwidth. Similarly cutoff data can be obtained for memory.

For predicting the available capacity, each replica observes its 5 minute average of available capacity and categorizes the node into one of the three categories accordingly. It then uses the cutoff point for a particular resource for that category and assumes that much fraction of currently available capacity will be available in the next minute. This predicted resource capacity is then taken into account for estimating the service capacity.

We refer to this model of predicting available resource capacity as a *static model*. One can extend such

Table 1: Node Categories and Cut-offs used

	CPU		Bandwidth	
	Idle Capacity	Cut-off	Unused BW	Cut-off
Heavily loaded	<1GHz	40%	<50%	30%
Moderately loaded	1-2GHz	65%	>50%	75%
Lightly loaded	>2GHz	90%	>90%	90%

a model to an *online model* in which a replica observes how the available capacity at it's host changes and develop this heuristics online and adapt it continuously. In our experiments, we evaluated the *static model* only. Evaluation of an online model will be part of our future work.

3.3 Estimation of Service Capacity

The capacity estimation model and online micro-benchmarking model described above are used in estimating the request handling capacity of a replica at a particular time. We use the average resource demand of a request estimated through online micro-benchmarking and the predicted available capacity for each type of resource in determining the maximum number of requests that can be handled based on each type of resource. This estimation also indicates the bottleneck resource and the maximum number of requests that can be handled by the replica at that time.

A service replica predicts the available capacity at each interval i and calculates the following

- C_i predicted available CPU capacity for next observation interval
- M_i predicted available memory capacity for next observation interval
- B_i predicted available bandwidth for next observation interval
- r_i Maximum number of request that can be handled per second predicted for the next observation interval

The predicted available capacity of a particular type of resource together with the average resource usage demand of a request for that type of resource can be used to determine the maximum number of requests that can be handled based on that type of resource. For example, the maximum number of request that can be serviced per second based on CPU capacity (r_c) can be calculated as follow

$$r_c = C_i/D_c \tag{3}$$

Similarly, the maximum number of request that can be served per second based on predicted available bandwidth is given by

$$r_n = B_i/D_n \tag{4}$$

We can calculate number of request based on memory (r_m) in a similar way. The maximum number of request r_i that can be handled per second is then calculated by taking the minimum value of r_c, r_n and r_m Our current work has mainly focused on CPU, network, and memory demand of the workload because in offline benchmarking of our workload we found that the network and CPU were the bottleneck resources most of the time, and file I/O was never found to be the bottleneck. However, further work needs to be done for developing file I/O resource demand model in this framework. This would require access to I/O utilization data from the host environment.

3.4 Evaluation of Capacity Estimation Models

The evaluation of the service capacity estimation model presented above is performed for two aspects. The first aspect of our evaluation is the accuracy of the prediction model in predicting the service capacity for the next observation interval. The second aspect is to evaluate how accurately the estimated service capacity reflects the actual request handling capacity of the service replica. Precisely, this aspect is related to the evaluation of the micro-benchmarking models in accurately capturing the resource requirement of a request.

We refer to the service capacity predicted for a particular interval as *predicted service capacity.* The service capacity actually observed according to our model at that particular interval is called as *observed service capacity.* The *observed service capacity* is calculated based on the resource capacity observed to be available during that interval. The *actual service capacity* of the service replica during that interval is the maximum number of requests that could be successfully handled without causing saturation. This may be different from the *observed service capacity.* Thus for each interval i we record the following information:

- p_i predicted service capacity for interval i. It is the same as r_{i-1}
- o_i observed service capacity at interval i, It is calculated in same way as r_i but using the actual available resource capacities observed during that interval.
- a_i actual service capacity

Our first set of experiments focused on evaluating the performance of the prediction models. In these evaluations, we observed the ratio of *observed service capacity* to *predicted service capacity*, which we call the *prediction-ratio (ρ).*

$$\rho = o_i/p_i \tag{5}$$

A ρ value greater than 1 indicates that the prediction model is underestimating the capacity while a ρ value less than 1 indicates overestimation. Since our model only assumes some fraction of the currently idle capacity to be available in the next minute with high probability, we expect the prediction ratio to be greater than 1 most of the times. In order to evaluate the goodness of our prediction model in terms of the prediction ratio, we conducted several experiments over PlanetLab and observed the distribution of the prediction ratio.

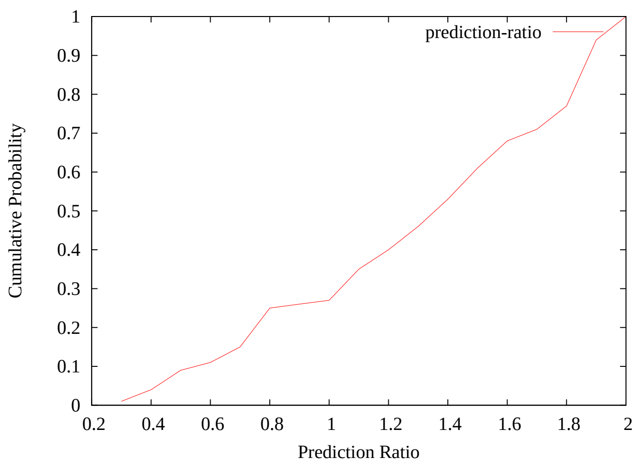

Figure 4: Distribution of prediction-ratio

Figure 4 shows the distribution of over 3300 observations of the ρ value from four experiments of deployment over PlanetLab. In these experiments, we deployed total of 55 service replicas over different PlanetLab nodes. We generated a client load at the level of 50% of the estimated service capacities. The median value of ρ is 1.3 which means that 50% of time the prediction model underestimated the capacity by 30%. As explained above a certain amount of underestimation is expected.

The second aspect of our evaluation was to assess how the estimated service capacity reflects the actual request handling capacity of the replica. However, the actual request handling capacity of a service replica at a particular time can only be determined by generating the load up to the point where it saturates. The immediately preceding level of load where the replica could successfully handle the request can be considered as the actual service capacity of the replica. For this purpose we programmed clients to gradually increase the load and observed the performance of service replica. Since the service capacity may fluctuate significantly in short time, a service replica may saturate earlier or later depending on the fluctuations. We considered the immediately preceding point before saturation[2] where it could successfully handle the load and calculated for those points the ratio of *actual service capacity* to *predicted service capacity*, called the *estimation-ratio (δ)*.

$$\delta = a_i/p_i \tag{6}$$

The *estimation-ratio* indicates the accuracy of our capacity estimation model in estimating the actual service capacity. We considered predicted service capacity instead of observed service capacity in calculating the *estimation-ratio*, since our capacity scaling and load distribution mechanisms are to be driven by the predicted service capacity values.

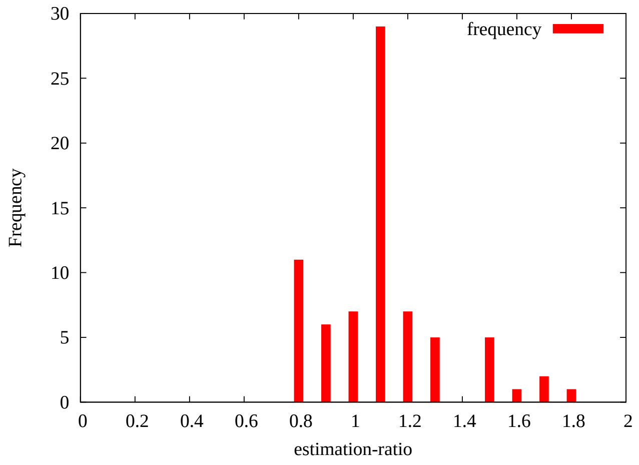

Figure 5: Histogram of estimation-ratio At Saturation

We performed many such experiments of deploying service replicas on PlanetLab nodes and induced client load to the point of saturation of service replicas. We observed the distribution of δ values at the saturation points. Figure 5 shows the distribution of δ values for 75 saturation points. A δ value less than 1 indicates that the service replica saturated at lower load than predicted while a value greater than 1 indicates that service replica could handle higher load than predicted. The average δ values for these saturation points was 1.07 and the median was 1.06. From Figure 4 we can see that the service replica could saturate at a load around 70% of the predicted service capacity.

4 Dynamic Capacity Scaling

Our goal here is to develop models and autonomic mechanisms for dynamic scaling of aggregate service capacity based on the service capacities estimated by each service replica and the currently observed load conditions. The main objective of the capacity management model is to maintain sufficient aggregate service capacity under events such as: fluctuating service capacities of individual service replicas, replica crashes, and fluctuating load conditions. Moreover, the capacity management model should not over-provision the service capacity beyond some level.

Towards these goals, the capacity scaling models and mechanisms need to address the following questions:

[2]We considered a continuous increase in average queue length of the service replica as the indication of saturation and we marked the point when the queue length increased above 10 as the saturation point.

- How much aggregate service capacity should be maintained in order to tolerate fluctuations in available resource capacities, client load and events such as replica crashes?
- When to generate additional capacity to meet the current load requirements?
- When to reduce the provisioned service capacity in order to avoid over-provision?

4.1 Capacity Management Model

Our capacity management model is based on the notion of *capacity slack*, which attempts to maintain a certain amount of slack capacity in the system with respect to the currently observed client load. Maintaining such additional capacity in the system ensures that the client load can be successfully handled when the aggregate capacity falls below some level due to fluctuations in service capacities of individual replicas and the fluctuations in client load. However, the slack capacity must also be maintained below certain level to avoid over-provisioning.

We define the fraction of the capacity that must be maintained as slack as the *target slack fraction (f_s)*. Such a slack capacity must be maintained based on the currently observed load. Let L be the current load, measured in number of requests per second. Let C be the aggregate service capacity which is the sum of the estimated service capacities of the individual service replicas measured as the maximum number of requests that can be served by each replica per second. In order to maintain the desired capacity slack, C should always satisfy the following criteria.

$$C \geq L * (1 + f_s) \tag{7}$$

In order to avoid over-provisioning the service capacity, the provisioned capacity should not be more than certain level called as *high watermark (f_h)*.

$$C < L * (1 + f_h) \tag{8}$$

Furthermore, the capacity management model should also account for the crash of service replicas. For this purpose, we enforce that the aggregate capacity should not fall below certain level due to the crash of a single replica. We refer to this level as *low watermark (f_l)*. Let C_m be the maximum amount of service capacity amongst the individual service replicas, then the following condition ensures that the system is not vulnerable to one replica crash.

$$C - C_m \geq L * (1 + f_l) \tag{9}$$

Based on the above criteria, the capacity scaling is performed as follow. At every observation interval the aggregate service capacity and total client load is calculated using the capacity and load information provided by each replica. If the current aggregate service capacity does not satisfy the criteria specified by equations (7) and (9), then the sufficient amount of additional capacity required to satisfy these two criterion is generated. If the current aggregate service capacity is beyond the high watermark, then the lowest capacity replica is marked for removal to reduce the capacity. However, it is only removed if the remaining aggregate service capacity still satisfies the conditions given by equations (7) and (9).

In the evaluations of our service capacity estimation models described in Section 3, we observed that a service replica may saturate at a load around 70% of the estimated service capacity. Based on this observation, we configured our capacity management model to always maintain 30% slack capacity in the system. Therefore, in our experiments we set the value of f_s to be 0.3. For low and high watermarks, we set the value of f_l to be 0.1 and value of f_h to be 0.5. The performance of the capacity management model can be evaluated by observing the ratio of provisioned capacity to the client load. We refer to it as the *provisioning-ratio (τ)*. According to our model a τ value greater than 1.5 indicates over-provisioning while a τ value less than 1.3 indicates under-provisioning. The evaluation of our capacity management model is detailed in Section 6

4.2 Autonomic Mechanisms for Capacity Scaling

We present here the autonomic mechanisms we have developed for dynamic scaling of service capacity based on the capacity management model presented above. Our capacity scaling mechanisms are based on dynamic replication and adjustment of the number of service replicas. We consider here a content distribution service with read-only requests, therefore we do not address here the issues related to update synchronization. Furthermore we assume that the content distribution files are already present at the host where the service replica is to be deployed. Therefore we do not address in this paper the issue of provisioning service content.

In our experimental system, the dynamic capacity scaling is performed by the *Deployment Agent*. The Deployment Agent continuously monitors the aggregate service capacity and current load conditions. The aggregate service capacity and total load is calculated using the service capacities and load information reported by individual replicas. The Deployment Agent then checks the current load conditions and according to the capacity management model presented above, decides if additional capacity needs to be generated or the existing service capacity needs to be reduced. When more capacity is to be generated, Deployment Agent calculates the additional amount of capacity that needs to be generated and requests a list of available nodes from the monitoring service which satisfy certain minimum resource capacity requirements. The monitoring service also provides information about the currently observed available resource capacities at those nodes. Based on the available resource capacities, the Deployment Agent picks a set of high capacity nodes. It also calculates the average resource demands of a request from the resource demands values reported by the individual service replicas. These average resource demands are used as seed values to assess the approximate service capacity that would be provided if the new service replicas are deployed on the selected hosts. Based on this calculation, the Deployment Agent creates one or more service replica agents on the appropriate nodes. Since the new service replicas would need to perform online benchmarking for some time for service capacity estimation to accurately reflect the actual service capacity, the Deployment Agent waits for two observation intervals before making further decisions of capacity management. When the aggregate service capacity needs to be reduced, Deployment Agent calculates the amount of capacity that needs to be reduced and marks the lowest capacity replica for removal. It then checks if removal of that replica violets conditions given by equations (7) and (9). If not, it sends a 'terminate' message to the service replica, removes the replica record from DHT and recalculates the new aggregate service capacity. If further reduction of capacity is required, the Deployment Agent performs the above procedure again.

For detecting the crashes of the service replicas, the Deployment Agent monitors the periodic reports received from the service replicas. If the Deployment Agent does not receive reports from a service replica for more than two observation intervals, it marks the service replica as *suspect-failed*. It then tries to ping and verify the status of replica. If the Deployment Agent fails to communicate with the replica it marks the replica as *failed* and removes the replica from the system. It then checks if additional capacity is required to be generated due to the replica crashes, and generates capacity as described above.

5 Adaptive Load Distribution

As discussed earlier, one of the challenges in building scalable services over shared computing environments is the distribution of load according to the fluctuating service capacities of the service replicas. The fluctuations in service capacities of the individual service replicas necessitates the need for adaptive load distribution. In this section, we present the mechanisms we developed and evaluated for the adaptive load distribution.

In our system model a client first queries the DHT for accessing a particular service. In response to the client query, the DHT selects one of the replica and returns its network address to the client. The client then sends subsequent requests to that service replica. A DHT lookup is performed again by the client only after some number of requests. Based on this service access model, we have two different levels at which load distribution is performed:

- DHT-Level: This is the load distribution performed by the DHT while selecting the service replicas for client queries. The load distribution mechanisms at DHT level operate at a relatively coarse level.

- Replica-Level: Each service replica performs the load distribution based on its current load conditions. This is at a more fine-grain level.

At a replica-level the load distribution is primarily performed by redirecting the load to another service replica. At replica-level we have the following mechanisms of load redirection.

- Request Redirection: A service replica redirects a single request to another replica. This is also called temporary redirection, since the subsequent requests are sent to the original service replica.
- Client Redirection: A service replica may also redirect the client permanently to another replica. The client then sends the subsequent requests to the new replica until it performs the DHT lookup again. Hence it is also called permanent redirection.
- Forced DHT Lookup: A service replica may force the client to perform DHT lookup again.

5.1 DHT-Level Load Distribution

The load distribution at DHT level is performed based on the service capacities of the individual service replicas. For a balanced distribution of load, each replica should handle the fraction of the load proportional to its service capacity. We define fraction of the load that should be distributed to a particular replica as the *load distribution fraction(F)* of that replica. This load distribution fraction is given by how much fraction of the total service capacity is provided by that replica. Thus for replica r, if C_r is the service capacity, then load distribution fraction F_r of that replica is given by

$$F_r = \frac{C_r}{\sum_{i=0}^{n} C_i} \tag{10}$$

Each replica periodically reports its service capacity to DHT. Based on this reported service capacity, DHT calculates the load distribution fraction for each replica. When a client performs a DHT lookup, the DHT selects one of the replica randomly with the probability of the selection a replica being proportional to its load distribution fraction.

5.2 Replica-Level Load Distribution

The mechanisms for load distribution at the DHT level can only perform distribution of clients and hence only operate at a coarse level. For more fine-grain load distribution we need mechanisms which operate at the level of individual replicas. A replica may become overloaded due to fluctuations in its service capacity or due to increase in client load distributed to that replica. In such cases, the replica needs to redirect a fraction of its load to another replica. A replica may shed its client load through request redirection, client redirection or forced DHT lookups. For this purpose, the load distribution mechanisms at the replica level need to address following questions:

- How to detect an overload situation?
- How to find the target replicas for the redirection of load?
- When to perform request redirection, client redirection or forced DHT lookup?

The first problem is to accurately assess the current load condition at a replica level. For this purpose, we develop a token-based model for effectively characterizing the load and service capacities. In this model *tokens* represent the maximum number of requests that can be serviced by a replica during an observation period. As described in Section 3, in each observation period i a service replica estimates its service capacity for the next period, represented by r_i (as number of requests per second). This represents the estimated maximum service processing rate over the next period. Based on this rate, at the beginning of an observation period, the replica computes the number of tokens T_i representing the maximum number of requests that it can service over that period. A token is consumed every time a request is served. In case of balanced load conditions, the tokens will be consumed at a uniform rate. An overload condition is suspected if the tokens are consumed at a rate higher than the estimated rate. Similarly, a replica is considered to be underloaded when the token consumption rate is significantly below the estimated service rate.

At time t from the beginning of the current period, a replica can detect if it's overloaded by observing the number of tokens consumed by that time. If x_t is the number of tokens consumed by the time t, then the replica is overloaded if

$$x_t > r_i * t \tag{11}$$

Similarly the replica is underloaded if

$$x_t << r_i * t \tag{12}$$

In case of balanced load situations x_t is close to $r_i * t$.

When a replica detects overload situations, it needs to find suitable targets for redirecting its load. A replica also needs to decide if it should perform request-redirection or client-redirection. In our model, a replica performs client-redirection in case of high overload conditions, which happen when the number of redirected requests exceed some threshold, which is set to 30% of token count T_i. Otherwise, in case of low overload conditions it redirects individual requests only. It forces a client to relookup the DHT if it cannot find any suitable target.

In order to find target replicas for redirection, the replica needs information about load shared by other replicas. For this purpose, in our system the replicas periodically exchange load information such as the number of requests served, the number of tokens left and the current load status (underloaded, overloaded etc). This periodic exchange is performed at each quarter of the total observation interval. Only the replicas which have underload status are considered as redirection targets by a replica. For each such potential target replicas, the replica calculates the maximum number of requests that can be redirected to that target replica in a given interval, called as the as the *redirection quota* of that target replica. The redirection quota of a target replica is calculated as follow. Let N be the total number of replicas in the system, and T_r be the number of tokens remaining of the target replica r then redirection quota q_r of the target replica r is calculated as

$$q_r = \frac{T_r}{(N/2)} \tag{13}$$

This redirection quota is calculated by considering that some other replica may also redirect its requests to that target replica. Therefore, we assume that the remaining number of tokens of a target replica can be consumed equally by the potential number of replicas who may perform load redirection. Furthermore we assume that, on average half of the total number of replicas may be overloaded so we divide the remaining number of tokens of a target replica by $N/2$.

For client redirection, only the target replicas with redirection quota more than certain limit (we set this limit to 100) are considered. When a replica decides to redirect a request or a client it selects a replica from the list of target replicas in round-robin manner. For request redirection, it sends a 'request-redirection' message to the client along with the network address of the selected target replica. In case of client redirection, it sends a 'client-redirection' message and the network address of the selected replica.

If a replica can not find any target replicas for redirection or if it has already exhausted the redirection quota of each target replica by redirecting those many number of requests to that replica, it decides to force the client to perform DHT lookup again. A 're-lookup' message is sent to the client in this case.

5.3 Evaluation of the Load Distribution Mechanisms

We present here our methods for evaluating the proposed DHT level and replica level load distribution mechanisms. Our evaluation is based on the deviation of the observed load shared by each replica from its expected load. Let F_r be load distribution fraction of replica r for a given interval calculated using equation (10). Let S_r be the number of requests served by that replica in that interval and L be the total client load. Then the load deviation (β_r) of replica r is calculated as follow

$$\beta_r = |S_r - (F_r * L)| \tag{14}$$

The overall deviation in load distribution across all replicas called as the *distribution deviation*(β) is calculated in terms of the fraction of the total load that deviated from expected distribution. It is calculated as follow:

$$\beta = \frac{\sum_{i=0}^{n} \beta_i/2}{L} \tag{15}$$

In our evaluations over PlanetLab, we observed the distribution deviation. The details of these evaluations are presented in next section.

6 Evaluations

In this section, we present our evaluations of the dynamic capacity scaling and load distribution models and mechanisms. Our main criterion in these evaluations is the client side behavior in terms of the response times. To assess the performance of capacity provisioning models, we observed the provisioning-ratio. Specifically, our evaluations were based on following parameters. These parameters were observed every 1 minute.

- 90 percentile values of client side response times
- fraction of requests that were retried due to errors such as connection failure, socket timeouts etc.
- provisioning ratio (τ)
- distribution deviation (β)
- ratio of DHT load (number of lookup requests sent to DHT) to total service load to assess the load on DHT

We evaluated the performance of our models and mechanisms under a load conditions of constant load and step increase in load in which a constant step increase is added with each step. These evaluations were performed by deploying the service replicas over a pool of 150 nodes on PlanetLab. In PlanetLab, if a slice transfers more than total of 10 GB data in a day on a single node, its bandwidth is capped at a low rate. Since such cases would affect our capacity estimation, we programmed the replicas to measure the amount of total data they have communicated. If a service replica performs more than 9 GB of data communication, such a service replica is shut-down and a new service replica on a different host is created. The Deployment Agent keeps track of the nodes which have exhausted their data limit and such nodes are not used for hosting replicas for that particular day. This limitation is specific to PlanetLab environment only and hence does not affect our capacity models in general. In these experiments, we did not address the fault-tolerance of DHT and Deployment Agent. They were deployed in a controlled environment of our lab cluster.

6.1 Constant Load

The first aspect of our evaluation was to assess the performance of capacity provisioning and load distribution mechanisms under fluctuating service capacities and situation such as replica crashes. To evaluate these aspects of our mechanisms, we induced a constant load and observed the performance of the system. The requests were made for files with variable sizes with average file size of 100KB. Figure 6 shows the 90 percentile values for the client side response times observed every 1 minute for the step load increase from 3000 requests per minute to 25000 requests per minute. The statistics such as average values, standard deviations, min and max for client side response times and other performance parameters are given in the Table 2.

Figure 6 shows that most of the times client side response times were within 1 or 2 seconds. From Table 2, we can see that the average response time was 1.51 seconds with standard deviation of 0.7 and the average fraction of retried requests was 0.07. Figure 7 shows the provisioning ratio observed in this experiment. The average provisioning ratio was 1.36 and it rarely dropped below 1.0. Situations where it dropped below 1.0 were mainly caused by crash of more than one replicas. The performance of load distribution in terms of

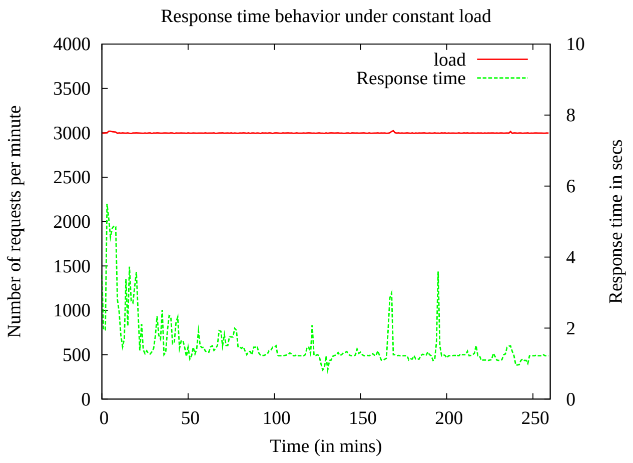

Figure 6: Client Side Response Times under Constant Load

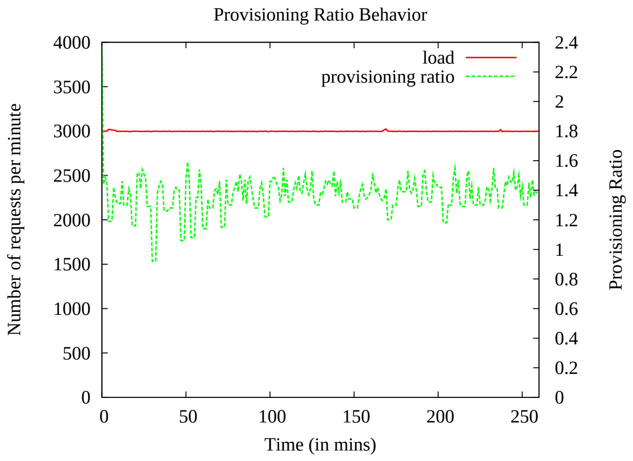

Figure 7: Provisioning Ratio under Constant Load

distribution deviation is shown in Figure 8. The average load distribution deviation was 0.12, which means that only 12% of load was deviated from its expected distribution. The average value of DHT load fraction was observed to be 0.02 which means the load on DHT was 2% of the total service load. In our experiments we set the value of number of requests after which a client performs DHT relookup as 100. Therefore, the DHT load fraction would always be greater than or equal to 0.01.

6.2 Step Increase in Load

The goal of this experiment was to evaluate how the dynamic scaling and load distribution mechanisms perform when the load increases in a short duration. We programmed the load generation such that it will add a constant step increase in load every 10 minutes. Such a step increase in the client load was induced within 1 to 2 minutes. We observed the performance of the system in terms of the parameters mentioned above. The statistics for the performance parameters mentioned above are shown in Table 3. The client side behavior in terms of response times is shown in Figure 9

Figure 10 shows how the capacity was generated as the load increased. The average provisioning ratio

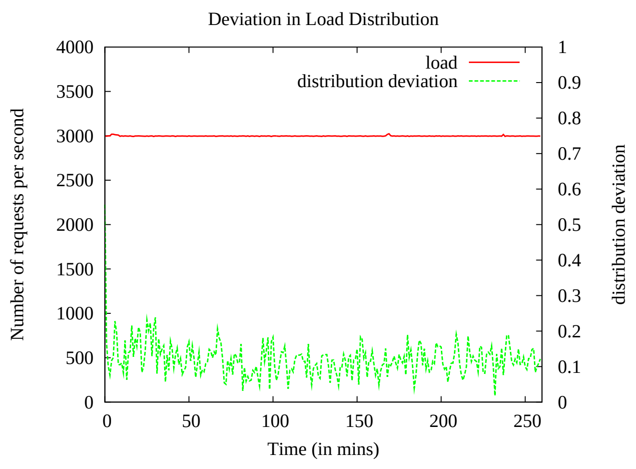

Figure 8: Load Distribution Deviation under Constant Load

Table 2: Performance Statistics for Constant Load Experiment

	Average	Std.Dev	Min	Max
response times (secs)	1.51	0.7	0.82	5.5
fraction of retried requests	0.07	0.06	0	0.3
provisioning ratio	1.36	0.11	0.92	1.59
distribution deviation	0.12	0.05	0.02	0.56
DHT load fraction	0.02	0.01	0.01	0.23

was observed to be 1.32. The provisioning ratio never dropped below 1.0. The average load distribution deviation was 0.18 and load on DHT was found to be 7% of total service load. The DHT load fraction is slightly higher than that observed under constant load. This is expected as during a step increase in load, a number of requests will fail forcing the clients to relookup DHT.

7 Related Work

Our goal of building scalable and available services over the Internet is similar to those for cluster-based services, but our underlying implementation environment is characteristically different. The problems of building highly available and scalable cluster-based network services have been addressed by many research projects in the past [5, 9, 3, 15, 2, 23, 14]. In these systems, the client requests are distributed by the front-end nodes to different servers that are connected to it by a high-speed local area network. The available capacities at the servers are fixed, and utilized solely by the load placed on them by the front-end nodes. Moreover, in the cluster based systems the front-end has a good estimate of the load status of the back-end servers. In our environment, there are no "front-end" nodes for performing request distribution and load

Table 3: Performance Statistics for Step Load Experiment

	Average	Std.Dev	Min	Max
response times (secs)	2.22	0.67	0.95	4.08
fraction of retried requests	0.1	0.11	0	0.36
provisioning ratio	1.32	0.19	1	1.77
distribution deviation	0.18	0.08	0.05	0.42
DHT load fraction	0.07	0.06	0	0.34

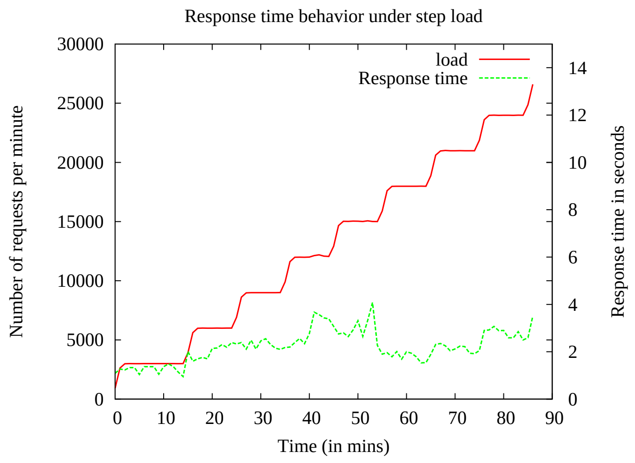

Figure 9: Client Side Response Times under Step Increase in Load

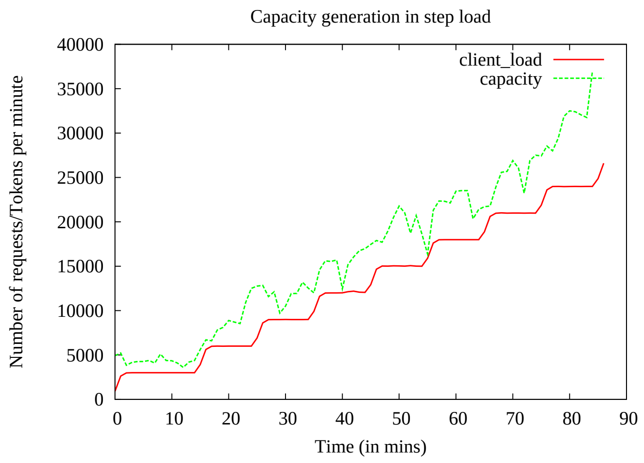

Figure 10: Capacity Generation under Step Increase in Load

balancing operations, and the service replica agents are deployed over a wide-area network. Moreover, there is no guarantee of the available resource capacities on the service agents' hosts. Such hosts are not under the control of the service administrator, and hosts may be shutdown or become unavailable at any time. Our design copes with such situations that may make a service agent unavailable.

Autonomic techniques for resource management and provisioning for services using online internal models for characterizing workload have been developed and studied in [4]. That work focused on provisioning of storage resources on a shared server cluster. Our work has also taken a model based approach for resource provisioning in a large-scale wide-area environment. We use models for characterizing workload demands, capacity prediction and aggregate service capacity management. Resource provisioning techniques for a cluster-based shared hosting platform based on online profiling of an application's resource consumption are presented in [20].. The problem addressed there is from the viewpoint of the hosting platform management, whereas our work is focused on the management and control of a service deployment. A control theoretic approach for resource management on web servers is presented in [1], focusing on limiting the utilization of bottleneck resources. Our models for autonomic service capacity management also include feedback-based control mechanisms controlling the degree of service replication.

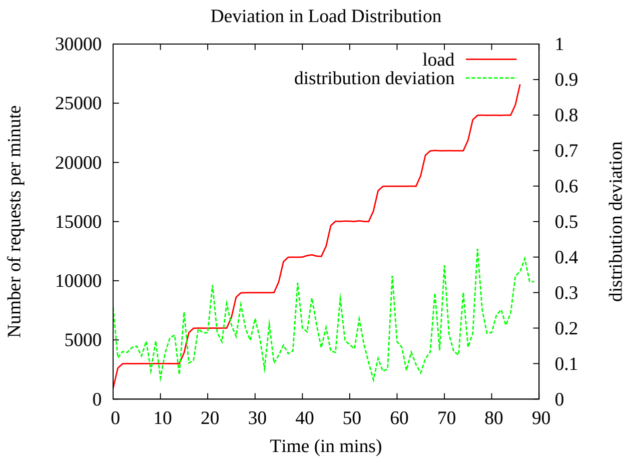

Figure 11: Load Distribution Deviation under Constant Load

Several researchers have proposed and developed system architectures for Internet-based *anycasting* [6, 21, 22]. Various approaches to build this functionality range from DNS level modifications [13], network-layer anycasting requiring routing level modifications, or building a service similar to DNS at the application level [22]. Our DHT-based redirection mechanism is at the application-level and does not require any modifications to the existing network infrastructure. The DNS based and network level solutions [11] are tedious to deploy. As shown in [13], DNS based solution are slow to react to changes in the replication configuration, such addition or removal of replica. In our environment we need mechanisms that are agile to react to such changes. Systems supporting anycast functionality for overlay-based routing have been developed [21, 6]. The topic of load balancing [16] techniques has been extensively studied in the past for environment the processing capacities of nodes are constant. In our environment, the service capacities of the replicas are typically fluctuating, and therefore the load distribution mechanisms have to be adaptive and agile. The load distribution mechanisms developed here address these needs.

8 Conclusions

In this paper, we have presented models and mechanisms for dynamic capacity management and load distribution for scalable services over wide-area shared computing platforms. Our approach was based on model-driven service capacity estimation and provisioning and dynamic control of degree of replication using feedback-based control mechanisms. Our mechanisms for adaptive and agile load distribution were based on continuous adaptation of load distribution fractions and use of token-based mechanisms for overload detection and fine-grain load distribution. We evaluated the performance of such models and mechanisms over PlanetLab. We demonstrated the capability of these mechanisms in autonomically controlling the degree of replication to provide the scalability and resiliency for services. Our evaluations demonstrated that the client side response times were always within some acceptable limits.

References

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance guarantees for web server end-systems: A control-theoretical approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 1, pp. 80–96, 2002.

[2] M. Aron, P. Druschel, and W. Zwaenepoel, “Cluster reserves: A mechanism for resource management in cluster-based network servers,” in *In Proceedings of the ACM SIGMETRICS Conference*, 2000, pp. 90–101.

[3] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel, “Scalable content-aware request distribution in cluster-based networks servers,” in *ATEC ’00: Proceedings of the annual conference on USENIX Annual Technical Conference*, 2000, p. 26.

[4] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat, “Model-based resource provisioning in a web service utility,” in *USITS’03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems.* Berkeley, CA, USA: USENIX Association, 2003, pp. 5–5.

[5] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, “Cluster-based scalable network services,” *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 78–91, 1997.

[6] M. J. Freedman, K. Lakshminarayanan, and D. Mazières, “OASIS: Anycast for Any Service,” in *NSDI’06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation.* Berkeley, CA, USA: USENIX Association, 2006, pp. 10–10.

[7] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance : Computer Systems Analysis using Queueing Network Models.* Prentice Hall, 1984.

[8] D. Oppenheimer, B. Chun, D. Patterson, A. C. Snoeren, and A. Vahdat, “Service Placement in a Shared Wide-Area Platform,” in *ATEC ’06: Proceedings of the Annual Conference on USENIX ’06 Annual Technical Conference.* Berkeley, CA, USA: USENIX Association, 2006, pp. 26–26.

[9] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, “Locality-aware request distribution in cluster-based network servers,” *SIGPLAN Not.*, vol. 33, no. 11, pp. 205–216, 1998.

[10] K. Park and V. S. Pai, “CoMon: A Mostly-scalable Monitoring System for PlanetLab,” *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 1, pp. 65–74, 2006.

[11] C. Partridge, T. Mendez, and W. Milliken, “RFC 1546: Host Anycasting Service,” November 1993.

[12] A. I. T. Rowstron and P. Druschel, “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems,” in *Middleware ’01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg.* London, UK: Springer-Verlag, 2001, pp. 329–350.

[13] A. Shaikh, R. Tewari, and M. Agrawal, “On the effectiveness of dns-based server selection,” in *In Proceedings of IEEE Infocom*, 2001.

[14] K. Shen, H. Tang, T. Yang, and L. Chu, “Integrated resource management for cluster-based Internet services,” in *Proceedings of the 5th Symposium on Operating Systems Designa dn Implementation*, 2002, pp. 225–238.

[15] K. Shen, T. Yang, and L. Chu, “Cluster load balancing for fine-grain network services,” in *IPDPS ’02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, 2002, p. 93.

[16] N. Shivaratri, P. Krueger, and M. Singhal, “Load Distributing for Locally Distributed Systems,” *IEEE Computer*, vol. 25, no. 12, pp. 33–44, 1992.

[17] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh, “Mobile Agent Programming in Ajanta,” in *Proceedings of the 19th International Conference on Distributed Computing Systems*, May 1999, pp. 190–197.

[18] A. Tripathi, V. Padhye, and D. Kulkarni, " Resource-Aware Migratory Services in Wide-Area Shared Computing Environments ," in *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS'09)*, 2009, pp. 51–60.

[19] A. R. Tripathi, N. M. Karnik, T. Ahmed, R. D. Singh, A. Prakash, V. Kakani, M. K. Vora, and M. Pathak, "Design of the Ajanta System for Mobile Agent Programming," *Journal of Systems and Software*, vol. 62, no. 2, pp. 123–140, May 2002.

[20] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," in *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002, pp. 239–254.

[21] C.-J. Wu, R.-H. Hwang, and J.-M. Ho, "A Scalable Overlay Framework for Internet Anycasting Service," in *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2007, pp. 193–197.

[22] E. W. Zegura, M. H. Ammar, Z. Fei, and S. Bhattacharjee, "Application-layer Anycasting: A Server Selection Architecture and Use in a Replicated Web Service," *IEEE/ACM Transactions on Networking*, vol. 8, no. 4, pp. 455–466, 2000. [Online]. Available: citeseer.ist.psu.edu/407466.html

[23] J. Zhou and T. Yang, "Selective early request termination for busy internet services," in *WWW '06: Proceedings of the 15th international conference on World Wide Web*. New York, NY, USA: ACM, 2006, pp. 605–614.