Autonomic Mechanisms for Building Scalable Services in Wide-Area Shared Computing Platforms

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Vinit A. Padhye

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Anand Tripathi

Feb, 2012

ABSTRACT

Cooperatively shared wide-area computing platforms, such as PlanetLab, provide a large pool of geographically distributed resources which can be utilized for building highly available and scalable services. In this thesis, we present mechanisms and models for building autonomically scalable and resilient services on such platforms. In such platforms resources at a node are allocated to competing users on fair-share basis, without any reserved resource capacities for any user. There is no platform-wide resource manager for the placement of users on different nodes. The users independently select nodes for their applications. Moreover, a node can become unavailable at any time due to crashes or shutdown. Our focus is on the PlanetLab platform which exemplifies the platform level characteristics considered here. Building scalable services in such environments poses unique challenges due to fluctuations in the available resource capacities and node crashes. The service load may surge in a short time due to flash crowds. We present here models for estimating the service capacity under varying operating conditions. Autonomic scaling of service capacity is performed by dynamic control of the degree of service replication based on the estimated service capacity and the observed load. This requires selection of appropriate nodes for the placement of new replicas. Furthermore adaptive load distribution mechanisms are needed because of the varying service capacities of the individual replicas. We present the experimental evaluations of these mechanisms on PlanetLab.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The availability of large scale distributed infrastructures such as cloud and grid computing platforms [Douglis and Foster, 2003] gives opportunity for shared computing and service hosting where services and applications can utilize the large pool of resource provided by such platforms. Unlike dedicated hosting, where entire pool of resources are dedicated to a single service or application, in shared hosting platforms the available computing resources are concurrently used by many applications and services. In such platforms often multiple applications may be co-hosted on a single physical machine. Cloud platforms such as Amazon EC2 [Amazon,] or Microsoft Azure [Microsoft,] are examples of a type of shared hosting platforms where the hosted services and applications are allocated resources with certain capacity guarantees. We consider the shared computing platforms which do not provide any resources or resource abstractions with fixed capacity guarantees. In such shared platforms applications may be co-hosted on a node and such applications would compete for resources available on that node. The platform level resource management would allocate resources on fair-share basis to the competing applications and there is no reservation of resources. Examples of such platforms include Planetlab [Bavier et al., 2004] and Grid computing environments. In such platforms, typically the unused resource capacities at a node can fluctuate due to the resource demands of the applications hosted on that node. However, the abundance of computing resources provided by such platforms can be utilized to build scalable and highly available services. The goal of our work

is to utilize the resources provided by such platform for elastic and dynamic scaling of services.

1.1 Characteristics of Shared Hosting Platforms

We describe here the characteristics of the kind of shared hosting platforms in question and state our assumptions about the platform level resource provisioning policies.

- **No provision of fixed resource capacity:** Typically such platforms provide a cooperatively shared pool of resources which are used by the platform users. However, these platforms do not provide any dedicated resources or resource abstractions with fixed capacity guarantees to the users. There is no reservation of resource capacities.
- **No central resource manager:** In such platforms, there is no central resource manager or scheduler which can do platform-wide resource allocation in order to balance the resource utilization and load. The users deploying the applications select the nodes to be used. A single node can co-host many applications or services of multiple users and they compete for the resources available on that node.
- **Fair-share based resource allocation:** The resources on a single node are allocated on fair-share basis to competing users. For example, in Planetlab a user is given a *slice* on each node and the resources are allocated to slices on fair-share basis. An application can consume the unused resources on a node as long as the other users do not compete for it. However, the unused resource capacities on a node can fluctuate due to the fluctuating resource demands of the applications hosted on it. Moreover, the number of users using the node can also change dynamically. Due to this, there is no guarantee of the resource capacities available to a service.
- **Lack of guarantees of node availability:** Availability of nodes is also not guaranteed. A node may be shutdown at any point of time, or it may crash due to software or hardware failures, or it may become unavailable due to network connectivity issues. Since the platform does not provide any guarantees, the crash or unavailability of a node would result in unavailability of the service or service component hosted on it and abrupt reduction in the service capacity.

These characteristics raise a number of issues that need to be addressed to meet the availability and performance requirements of services under widely varying load conditions. These issues are discussed below.

1.2 Issues in building scalable services

The lack of guarantees for resource capacity and availability require agile and dynamic scaling mechanisms to ensure the availability and scalability of a service. The requirement of resiliency and scalability both demand replication and regeneration of services or service components. Replication is required for service scaling under fluctuating load conditions as well as to ensure the availability of the service in case of node crashes. Regeneration is required to restart the failed replicas or components. However, due to the intrinsic characteristics of these platforms a number of issues need to be addressed in building scalable services on such platforms as discussed below.

The fluctuations in resource capacities available at a node would affect the request handling capacity of the service replica hosted on that node. This necessitates the need for estimating the request handling capacity of the replica at a given time according to the resource capacities available at that node. The models for estimating the service capacity at a replica also need to consider the workload profiles to accurately estimate the average resource demands of the requests. These workload characteristics may also change with time. Furthermore, fluctuations in resource capacities also demand prediction of resource capacity that is likely to be available in the near future.

The load generated by the clients typically changes with time and such changes can be significant especially in events such as *flash crowds* [Arlitt and Jin, 1999, Bodik et al., 2010]. Studies done by [Arlitt and Jin, 1999, Bodik et al., 2010] show that significant changes in client workload can occur over the duration of few hours. The analysis presented in [Arlitt and Jin, 1999] shows that increase of up to five fold in the client workload can occur in the duration of couple of hours. 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.

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.

The deployment of service replicas requires selection of suitable nodes for hosting them. Such a set of nodes needs to be picked based on their available resource capacities and the likelihood that they would remain suitable for service hosting for some time in the near future. Furthermore, since the network locations of service replicas would change with time, suitable mechanisms are needed for clients to locate the currently active service replicas. Such mechanisms themselves must be highly available.

1.3 Research Problem

The unique characteristics of these autonomously managed platforms without any centralized resource management raise various research questions that need to be addressed for building scalable and resilient services under dynamically varying operating conditions. Specifically, we investigate the following issues

- How to estimate the average resource demands for handling a request? These average resource demands would depend on the characteristics of the workload, which may change with time. The estimation of resource demands of processing a request in conjunction with the estimated resource capacities available on a node would determine the service capacity of the replica at that node.

- How to predict the resource capacity that is likely to be available (i.e. not used by other users) on a given node in the near future? Can the recent history of resource usage on the

node be utilized in guiding the prediction models?

- Due to the fluctuations in available resource capacities as well as the lack of guarantees of the node availabilities, sufficient service capacity must be maintained to tolerate the service capacity fluctuations as well as to meet the fluctuations in load demands. Since these fluctuations are unpredictable, a certain amount of excess service capacity must be maintained in the system. However, such capacity should not be provisioned above certain limit to avoid overprovisioning. What are the different models and mechanisms to provision sufficient service capacity without overprovisioning?

- What are the mechanisms for clients to identify and locate the currently active replicas? How to ensure the robustness of such mechanisms? What is the impact of the unavailability of these mechanisms on the system performance?

- Since the request handling capacity of a replica may change with time, the amount of client load distributed to a service replica must be proportional to the request handling capacity of the replica at that time. What are the techniques that would adaptively distribute the client load based on the service capacities of the replicas?

- What are the different strategies for selecting the nodes that are likely to remain suitable for hosting the service replicas for a long duration?

Our approach for addressing these issues is based on developing models for capacity estimation and service capacity provisioning and then using these models to drive the dynamic replication of the service. We present here the models and autonomic mechanisms that we have developed for dynamic service scaling. We evaluated these models and mechanisms over Planetlab environment. Specifically, our contributions are in the following areas.

1. Development of models for estimating average resource demands of requests taking into account the changing workload characteristics

2. Models for predicting the resource capacities likely to be available at a node in the near future and using the predicted resource capacities to estimate the service capacity that would be provided by that node;

3. Models for provisioning the aggregate service capacity in order to maintain some target amount of excess capacity in the system.

4. Mechanisms for adaptive load distribution of client load across service replicas;

5. Mechanisms and protocols for clients to access the deployed service replicas.

6. Prototype framework for building autonomically scalable services over the Planetlab environment.

We develop autonomic mechanisms for deploying and scaling services on such platforms as well as distributing the load according to the service capacities of the individual replicas. These mechanisms can be used and extended by service developers to build highly available and scalable services. We consider services and applications that do not require complex update synchronization protocols and where weak consistency based update protocols are applicable. Examples of such services include content distribution services where the content updates are relatively infrequent, and personal data sharing services where an object is updated by one client but read by many. For such services, the client requests are overwhelmingly read-only in nature. Thus, our work here does not address issues related to update coordination protocols. A service can include any suitable update synchronization mechanism in its design.

1.4 Related Work

- **Cluster-based services:** Our goal of building scalable and available services over the Internet is similar to those for cluster-based services, but our underlying 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 [Fox et al., 1997, Pai et al., 1998, Aron et al., 2000b, Shen et al., 2002b, Aron et al., 2000a, Zhou and Yang, 2006, Shen et al., 2002a]. Soft state based recovery mechanisms for building resilient components in large-scale systems was proposed and used in [Fox et al., 1997]. 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 balancing operations, and the service replicas are deployed over a wide-area network. Moreover, there is no guarantee of the available resource capacities on the nodes hosting the service replicas. Such nodes are not under the control of the service administrator, and they may be shutdown or become unavailable at any time.

- **Dynamic provisioning in shared platforms:** Model-based approaches for autonomic provisioning of services using online internal models for characterizing workload have been studied in [Doyle et al., 2003]. 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. Our dynamic provisioning model can be compared to elastic resource provisioning techniques in cloud platforms. However, unlike cloud platforms which provide well provisioned resources with fixed capacities, the PlanetLab environment does not provide any guarantees of resource availability. The dynamic provisioning model developed by us addresses these challenges. Resource provisioning techniques for a cluster-based shared hosting platform using online profiling of an application's resource consumption are presented in [Urgaonkar et al., 2002]. The focus of that work is on maximizing the utilization of the shared cluster resources. Rather than examining the problem of resource management on a hosting platform from the viewpoint of platform provider, our work is focused on the management and control of a service deployment. The approach used by us for autonomic service capacity management includes feedback-based mechanisms controlling the degree of service replication. Such control system based approaches have been investigated in the past in web server designs [Abdelzaher et al., 2002].

- **Dynamic service replication and relocation:** The notion of dynamic service replication and relocation of a service for fault-tolerance has been studied in the past in the HydraNet-FT system [Shenoy et al., 2000]. HydraNet-FT design requires specially

equipped routers as redirection points. Other researchers have developed server fault-tolerance techniques based on TCP connection redirection or migration [Sultan et al., 2002, Sultan et al., 2003, Marwah et al., 2003]. Such techniques require customized modifications to the operating systems kernel or the routers. For directing client requests to any of the replicas of a service over the Internet, approaches based on *anycasting* [Freedman et al., 2006, Wu et al., 2007, Zegura et al., 2000] have been proposed in the past. Various approaches to build this functionality range from DNS level modifications [Shaikh et al., 2001], network-layer anycasting requiring router level modifications, or building a service similar to DNS at the application level [Zegura et al., 2000]. We present here a registry-based redirection mechanisms which operate at the application-level and does not require any modifications to the existing network infrastructure. In contrast, the DNS based and network level solutions [Partridge et al., 1993] are tedious to deploy and they are slow to react to changes in the replication configuration [Shaikh et al., 2001] because of addition or removal of replicas. Our mechanisms are agile and able to react quickly to such changes, as shown by our evaluations.

- **Load distribution:** The topic of load balancing [Shivaratri et al., 1992] techniques has been extensively studied in the past for environments where 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.

- **Resource monitoring in large scale systems:** Several other research projects, such as CoMon [Park and Pai, 2006] and Sophia [Wawrzoniak et al., 2004], have investigated monitoring of PlanetLab nodes for their resource consumption. In Sophia system, event aggregation and inference model is presented. CoMon periodically collects and provides node-level statistics such as the number of active slices, per slice utilization of CPU, memory, and bandwidth. A number of research projects have analyzed this data for characterizing the resource utilization [Oppenheimer et al., 2006, Cardosa and Chandra, 2008]. The work in [Cardosa and Chandra, 2008] presents statistical methods for resource discovery and for characterization of nodes based on their resource usage. It classifies

nodes into different groups based on the similarities in their resource availability characteristics. The focus of the work in [Oppenheimer et al., 2006] was mainly on the characterization of resource availability of the PlanetLab nodes based on long-term observation data. Our focus is on characterization of nodes based on their recent resource availability. In [Warns et al., 2008], analysis of the CoMon data is presented for characterizing node failures and availability. In contrast to these previous works, our focus is on online monitoring and selection of PlanetLab nodes for hosting service replicas. Dynamic prediction models for forecasting network performance have been investigated in [Wolski et al., 1999, Wolski, 1998]. We present here an online model for predicting available resource capacities for nodes in PlanetLab environment based on their recent load conditions.

Chapter 2

Service Capacity Estimation And Scaling

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.

- First, we estimate the average resource usage demand of a request through online 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.

2.1 Online Benchmarking of Workload

For estimating the average resource usage demand of a request, each service replica performs continuous monitoring of its resource usage and workload. A service replica collects resource usage information of different types of resource such as CPU, memory and bandwidth. A service replica collects two types of information, one is about its own resource usage and the other is about the cumulative resource usage of other applications executing on that node. A replica's own resource usage information in conjunction with the workload characteristics is used in estimating the average resource requirement for handling a request. The information about cumulative resource usage of other applications is used in estimating the available resource capacities on that node. We assume that certain mechanisms are provided by the platform to collect such information. In our experimental prototype over Planetlab this information is collected by probing the *Slicestat* service executing on that node. Based on such information each service replica maintains the following statistics for resource usage at interval i.

- p_i - Average CPU usage (measured in MHz) of replica's own slice over the interval i. 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 node's intrinsic CPU capacity we take into consideration the number of cores and the CPU speed.

- m_i - Average physical memory usage (measured in MB) of replica's own slice over the interval i.

- b_i - Average bandwidth usage (measured in KBps) of replica's own slice over the interval i.

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. This is measured as the time since the request was received to the time when the response was sent.

- n_i - Average amount of per request data communication over the network (number of bytes sent and received).

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

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

$$D_{p_i} = p_i / s_i \tag{2.1}$$

- $D_{m\,i}$ - 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:

$$D_{m\,i} = (m_i - B_m)/s_i \tag{2.2}$$

- $D_{n\,i}$ - Per request network usage, it is same as the average amount of data sent and received over the network for handling a request.

$$D_{n\,i} = n_i \tag{2.3}$$

To accurately capture the workload characteristics and the average resource requirement of processing a request, we need to estimate these per-request demands over a sufficiently large sample of requests. In order to determine that a sufficient number of samples is collected, we continuously calculate the sample mean and variance of the per-request resource demands for currently observed requests and then using the Student's t-distribution we determine the number of samples required such that the population mean is within the 5% margin of the sample mean with 95% confidence. We assume that the changes in workload characteristics tend to be gradual over the duration of five minutes. Therefore, we consider the average of per-request resource demand values over past five minutes interval in estimating the service capacity. The average values of the above per-request resource demand measures computed over the past five minutes

are represented by D_p and D_m. Similarly, per request network usage D_n is obtained based on the five minute average of the n_i values.

2.2 Prediction of Available Resource Capacity

The problem that we address here is how to estimate for a given resource the amount of its capacity that is likely to be available (i.e. not used by other users) in the near future with some given probability. In order to predict the resource capacity that is likely to be available in the near future we need to observe the fluctuations in the available resource capacities over time. To characterize such fluctuations, for each resource type we observe the average available resource capacities over some period, called *observation period (w_o)* and the average available resource capacity over some period in the immediate future, called *prediction period (w_p)*. We then calculate the ratio of average available resource capacity R_{w_p} observed over w_p to average available resource capacity R_{w_o} observed over w_o. We call this ratio the *capacity modulation ratio (θ)*.

$$\theta = \frac{R_{w_p}}{R_{w_o}} \tag{2.4}$$

A capacity modulation ratio greater than 1 indicates increase in the available resource capacity by some fraction and a ratio value less than 1 indicates decrease in the available resource capacity. $P[\theta \geq x]$ is the probability that the average available resource capacity over the next prediction period w_p is at least $R_{w_o} \cdot x$. Therefore, to predict the fraction of the available resource capacity that is likely to be available with a specified confidence level C, we calculate x such that $P[\theta \geq x] = C$. We use w_p as the period for the prediction cycle and also as the control and reporting interval for periodic execution of service capacity estimation and scaling mechanisms.

The prediction of the available resource capacity needs to be done for individual nodes. This is because we observed that the fluctuations in the available resource capacities vary for individual nodes depending on their load conditions. This necessitates the need for a dynamic model that predicts the available capacities for individual nodes based on their current load conditions. We assume that while the available resource capacity itself may change significantly over short durations, such changes (that is the θ values) are statistically predictable over duration

of few minutes. Therefore, our dynamic model for resource capacity prediction is based on observing the history of θ values over some period called *history window (w_h)*. Our prediction model estimates the cumulative distribution (CDF) of the θ values observed over a sliding window of period w_h and calculates the value x for some given confidence level C, such that $P[\theta \geq x] = C$. This value is used to estimate the resource capacity for the next prediction period. For example, suppose that x_i is the value calculated, as described above, at the ith prediction cycle for CPU resource. Let p_i be the observed average available CPU capacity over the immediately preceding observation period of w_o duration at the i'th prediction cycle. The predicted CPU capacity P_i for the following prediction period w_p is estimated as:

$$P_i = p_i \cdot x_i \tag{2.5}$$

The goodness of the prediction model can be determined by observing the ratio of resource capacity observed to be available in a given interval to the capacity predicted for that interval. We call it the *prediction ratio (ϕ)*. A value of ϕ close to 1 indicates that the observed capacity is close to the predicted capacity, whereas values higher or lower than 1 indicate underprediction and overprediction, respectively. Since, in the equation (2.5), x_i is chosen such that $P[\theta \geq x_i] = C$, we expect that the resource capacity observed to be available during the immediately following prediction period w_p is at least $p_i \cdot x_i$ with probability C. Therefore, we expect that $P[\phi \geq 1] = C$. Using this observation, the goodness of the prediction model can be evaluated based on the value of ϕ at which this required confidence C is achieved.

The performance of the prediction model would depend on the specific values configured for the different parameters - w_h, w_o, w_p, and the level of confidence used to select the θ value for prediction. An important question is how to choose the values for these parameters. For the prediction period parameter (w_p), we want it to be of small duration (around 60-90 seconds). The capacity prediction over 60 or 90 seconds would guarantee that the requests would be completed within that period and the backlog of requests would not build over minutes range. This is desirable, since we want the response times to be in few seconds range. However, a very small prediction period is also not desirable. This is because the monitoring and service scaling functions would be performed at every prediction interval. A very small prediction period may not be adequate enough for executing the scaling functions. For the higher values of confidence

level parameter C, $P[\phi \geq 1]$ is higher and hence it may result in significant underprediction. Similarly, a lower confidence level may overpredict the resource capacity significant number of times.

For evaluating the performance of the resource capacity prediction model under various parameters, we collected the traces of approximately 300 Planetlab nodes. These traces include the available capacity information of various resource types collected every 10 seconds for each node. This evaluation is independent of the load imposed on a deployed service by its clients as the resource capacity prediction model depends only on the available resource capacities of individual nodes. Table 5.2 describes the datasets we used for this evaluation. Table 2.2 shows the values configured for different parameters for comparative evaluation. We addressed the following questions in this evaluation

- How to determine the length of the observation period w_o and the prediction period w_p?
- What is the impact of the parameter history window w_h on the prediction performance?
- How to determine the value of the confidence level parameter C?

Table 2.1: Datasets used for Evaluating Resource Capacity Prediction Model

Dataset	Time	Duration	Number of Nodes
Dataset1	Feb 11-12, 2011	38 hours	291
Dataset2	Oct 25-28, 2010	3 days	257
Dataset3	Sep 1-2, 2010	18 hours	305

Table 2.2: Configuration Parameters for Prediction Model

Parameter Name	Values
history window (w_h)	(10,20,30,40,50,60) mins
observation period (w_o)	(1,3,5) mins
prediction period (w_p)	(30,60,90) secs
confidence level	(60%,70%,80%,90%)

2.2.1 Impact of parameters w_o and w_p on prediction performance

To determine the length of w_o and w_p, we first fixed the length of the w_p to be 1 minute and observed the impact of w_o on prediction performance. For evaluating the prediction performance, we observed the CDF of the prediction ratio. Since the prediction ratio value close to 1 is desirable, we determined the goodness of prediction in terms of the probability mass between 0.9 and 1.1. We evaluated the prediction performance with the w_o values of 1, 3, and 5 minutes. We observed that larger observation periods result in less accurate prediction. This data is shown in Table 2.3 for Dataset1 for CPU and network bandwidth. Similarly, we evaluated the prediction performance for w_p values of 30 and 90 seconds. We observed that w_o and w_p value of 1 minute give better prediction performance. We observed similar trends for all other datasets.

Table 2.3: Comparison of Prediction Performance for various values of w_o and w_p

observation period (w_o)	prediction period (w_p)	$P[0.9 \leq \phi \leq 1.1]$	
		CPU	Network Bandwidth
5 min	1 min	0.236	0.82
3 min	1 min	0.247	0.82
1 min	1 min	0.262	0.83
1 min	30 secs	0.254	0.83
3 min	90 secs	0.249	0.82

Table 2.4: Comparison of Prediction Performance for various values of history window and confidence level for CPU

history window	confidence level	P[$\phi \geq 0.95$]	P[$\phi \leq 1.5$]	Difference
30	70	0.32	0.88	0.56
40	70	0.32	0.89	0.57
60	70	0.31	0.89	0.57
30	80	0.22	0.76	0.54
40	80	0.22	0.78	0.56
60	80	0.21	0.75	0.54
30	90	0.13	0.66	0.53
40	90	0.12	0.65	0.53
60	90	0.11	0.66	0.55

Table 2.5: Comparison of Prediction Performance for various values of history window and confidence level for Network Bandwidth

history window	confidence level	P[$\phi \geq 0.95$]	P[$\phi \leq 1.5$]	Difference
30	70	0.32	1.0	0.68
40	70	0.32	1.0	0.68
60	70	0.31	1.0	0.69
30	80	0.22	1.0	0.78
40	80	0.22	1.0	0.78
60	80	0.21	1.0	0.79
30	90	0.12	0.98	0.86
40	90	0.11	0.98	0.87
60	90	0.11	0.99	0.88

2.2.2 Impact of parameters w_h and C on prediction performance

The second question we addressed in our evaluation was how to determine the size of the history window (w_h). For the history window parameter, we did not want it to be very large as a larger window may not be able to capture the fluctuations occurring over a short duration. Therefore, the maximum history window size we configured in this evaluation was 60 minutes. As discussed earlier, for the confidence level parameter the goodness of the prediction model can be evaluated by observing the value of ϕ at which the required confidence level C is achieved, that is the value x such that $P[\phi \geq x] = C$. We observed that for all w_h values except for 10 and 20 minutes this was achieved at ϕ value of approximately 0.95. For w_h values of 10 and 20, the required confidence level was achieved for ϕ values of 0.9 and 0.92 respectively. We also observed that w_h value of 60 minutes performs better in achieving this required confidence level but only with marginal improvements over w_h values of 30, 40, and 50 minutes. This data is shown in Table 2.4 for Dataset1 for CPU capacity. Table 2.5 shows the same data for network bandwidth capacity.

For evaluating the effect of confidence level parameter C , we observed the amount of underprediction in terms of the probability of ϕ being greater than 1.5. We observed that for CPU resource the higher confidence levels result in significant underprediction. For example, as shown in Table 2.4, for the confidence level of 90 the prediction ratio was greater than 1.5 for 33% of the times. We can observe from Table 2.4 and Table 2.5 that confidence level of 70% gives less underprediction than other confidence levels for CPU capacity. However, for network bandwidth the confidence level parameter does not have any effect on amount of underprediction. In this

case confidence level of 90% is more desirable since it gives less overprediction. We observed similar trends for history window and confidence level parameters for all other datasets.

Based on these observations, we make the following conclusions

- Observation period w_o and prediction period w_p of 1 minute give more accurate prediction than other values.
- Window size of 60 minutes gives better accuracy in achieving required confidence level but only with marginal improvements compared to window size of 30, 40 and 50 minutes. Window size of 10 and 20 minutes gives significantly lower accuracy in achieving the required confidence level.
- For CPU resource the confidence level of 70% is desirable as it gives less underprediction than other values. However, for network bandwidth the amount of underprediction is less for all the values of confidence level parameter and therefore, for network bandwidth confidence level of 90% is more desirable since it gives less overprediction.

2.3 Estimation of Service Capacity

The capacity estimation model and online 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 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

- P_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_p) can be calculated as follow

$$r_p = P_i / D_p \tag{2.6}$$

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

$$r_n = B_i / D_n \tag{2.7}$$

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_p, 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.

2.4 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 this 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 benchmarking models in accurately capturing the resource requirement of a request.

We refer to the service capacity predicted for a particular interval i as *model-based predicted service capacity (π_i)*. The service capacity actually observed according to our model during interval i is called *model-based observed service capacity (Ω_i)*. The Ω_i value is calculated based

on the resource capacity observed to be available during interval i. The *actual service capacity (a_i)* of the service replica during interval i is the maximum number of requests that could be successfully handled without causing saturation. For evaluating the performance of the service capacity estimation models we define two measures. One is the *service capacity prediction ratio* (ρ), which is calculated as

$$\rho = \Omega_i / \pi_i \tag{2.8}$$

And the other measure is the *estimation ratio (δ)*, which is calculated as

$$\delta = a_i / \pi_i. \tag{2.9}$$

The actual request handling capacity of a service replica during a particular interval can only be determined by generating load close to the saturation point. We observed that in our testbed environment queuelengths close to 10 for a service replica indicated operating conditions close to the saturation point. For determining the saturation capacity during an interval, we imposed sufficient load on each replica to operate at queuelength of 10, but imposed admission control so it did not increase beyond this number. The sustained rate at each interval indicates the actual service capacity during that interval.

We deployed over 100 service replicas on different PlanetLab nodes and observed the distribution of ρ and δ values. We used E-commerce workload specifications of SPECweb2009 benchmark [SPEC,] to generate the client workload.

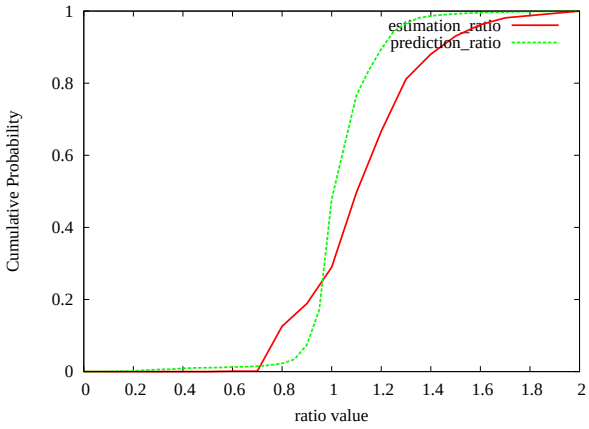


Figure 2.1: Distribution of prediction ratio ρ and estimation ratio δ

Figure 2.1 shows the distribution of over 3000 observations of the ρ values and the distribution of δ values of around 500 observations. The average δ value was observed to be 1.22 and the median was 1.14. This means that on average the actual service capacity is 22% more than the predicted value. We also observed that the probability of the actual capacity being lower than the predicted value was 0.28, and about 70% of the predicted capacity was available with probability close to 1.

2.5 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:

- 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?
- Under what conditions additional capacity should be generated?
- When to reduce the provisioned service capacity in order to avoid over-provisioning?

Our capacity management model is based on the notion of *capacity slack*, which attempts to maintain a certain amount of excess capacity in the system with respect to the currently observed client load. Maintaining such additional service capacity in the system ensures that the client load can be successfully handled under the fluctuating conditions described above.

We define *target slack fraction (f_s)* as the fraction of the capacity that must be maintained as slack. 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.

$$C \geq L * (1 + f_s) \tag{2.10}$$

When the aggregate service capacity is not sufficient to satisfy the target slack requirement, additional service capacity should be generated by adding new service replicas. However, the slack capacity must also be maintained below certain level to avoid over-provisioning. The requirements of maintaining the required slack capacity as well as avoiding overprovisioning may result in frequent addition and removal of replicas due to the fluctuations in the aggregate service capacity. This would make the system unstable. To avoid this problem, we enforce that the aggregate service capacity should be reduced only when it exceeds certain level called as *high watermark (f_h)*. Therefore, the aggregate service capacity C maintained in the system should satisfy the following criteria.

$$C < L * (1 + f_h) \tag{2.11}$$

The aggregate service capacity can reduce abruptly due to the crash of a service replica. This abrupt reduction in the capacity can be significant if the crashed replica contributed significant fraction of the aggregate service capacity and therefore the resulting aggregate service capacity may not be able to handle the current load conditions. 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 *1-replica crash vulnerable*.

$$C - C_m \geq L * (1 + f_l) \tag{2.12}$$

Based on the capacity management model described above, the capacity scaling is performed as described in 2.2. At every control 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 (2.10) and (2.12), then the sufficient amount of additional capacity required to satisfy these two criteria

```
C= total capacity of all replicas
L= total load of all replicas
C_max= Capacity of the largest capacity replica;
if (C < L * (1 + f_s)) {
    C_add = L * (1 + f_s) − C;
    addCapacity(C_add);
}
else if (C − C_max < L * (1 + f_l)) {
    C_add = L * (1 + f_l) − (C − C_max);
    addCapacity(C_add);
}
else if (C > L * (1 + f_h)) {
    while (C > L * (1 + f_h)) {
        r= getMinCapacityReplica(); /*remove lowest capacity replica*/
        C_new = C- r.capacity
        if (C_new > L * (1 + f_s) ∧ C_new − C_max > L * (1 + f_l)) {
            /* Removal will not cause 1-replica crash vulnerability */
            removeReplica(r);
            C = C_new;
        }
}
```

Figure 2.2: Algorithm for Capacity Scaling

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 (2.10) and (2.12). In our prototype framework, the capacity scaling functions are performed by a special component called as *DeploymentManager Agent (DA)*. The details of the DA functions are described in Chapter 3

The level of the capacity slack provisioned affects the availability and performance of the service that the clients would experience. However, higher capacity slack would also increase the deployment costs. The three parameters of this model – target slack, high watermark, and low watermark – affect the cost-performance trade-offs. We evaluated the cost-performance trade-offs for different levels of capacity slack. These evaluations are presented in Chapter 6

In this chapter we presented online models for workload benchmarking and prediction of available resource capacity. The workload benchmarking model estimates the average per-request resource requirement. The online prediction model predicts for a given resource the capacity

likely to be available in next prediction cycle with a given confidence level. The predicted resource capacities along with the per-request resource demands are used to predict the service capacity for the next prediction cycle. We also presented here a model for scaling service capacity based on the notion of capacity slack which tries to maintain some excess capacity in the system.

Chapter 3

Prototype Framework

We present here the prototype framework called Ellora, which we have built for evaluating the models and mechanisms presented earlier for building scalable services. We have built the Ellora framework over Planetlab. However, the architectural components and deployment framework of Ellora can be used for service deployment over any shared hosting platforms that have the characteristics described earlier. We assume that certain platform level mechanisms for extracting and monitoring resource usage information are provided. For example Planetlab provides *Slicestat* data for each node which gives the information about resource usage of various slices hosted on that node. Mechanisms which provide such information are required for service capacity estimation. Our framework does not depend on the availability of aggregate or global information about the available resource capacities or node availabilities.

3.1 Overview of the Ellora Framework

Service scaling in Ellora is achieved through dynamic replication of service components and dynamic adjustment of the degree of replication. The degree of replication is adjusted based on the service capacities of the deployed replicas and the load conditions. Dynamic replication is driven by the capacity scaling models presented earlier. For deployment, replication and regeneration of service replicas or components we use the mobile agent technology. For this

purpose, we use the Ajanta [Tripathi et al., 1999] framework for programming and deploying mobile agents.

Figure 3.1 shows the organization of the service deployment environment of Ellora. A service replica is implemented as a *service agent* which is a mobile agent implemented using Ajanta system. A service agent can be created and dispatched to a remote node for execution. It can be remotely controlled, and terminated if needed. Agents can be located and accessed using their location-independent names, which are based on the URN (Uniform Resource Naming) scheme [Sollins and Masinter, 1994]. Using the URN of an agent, access to its RMI interface is obtained. The Ajanta Name Service provides this facility. An agent can also communicate using TCP connections. The service agents provide TCP-based interface to the clients.

3.1.1 Registry Service

A Registry Service is used for maintaining the current configuration information for each deployed service. For each deployed service, the registry service maintains a record containing the list of all replica agents, their network addresses, and their currently estimated service capacities. Each service in our system is assigned a unique service-id (SID), and the record for a service is queried or updated using this id. The primary function of the registry is to direct a client to one of the replicas. Each replica periodically reports to the registry its estimated service capacity. Based on the recently reported service capacities, it determines the fraction of the total load that should be directed to it. The probability of selecting a replica is proportional to its load fraction. The load distribution fractions are updated continuously based on the periodic reports from the replicas. A replica is considered as crashed if no reports are received in three consecutive periods. The locations of the registry service replicas themselves may change with time, and for that purpose, as a bootstrap mechanism, several "lightweight" *Registry Locators* are provided at some well-known network locations.

The clients query the Registry Service for locating a service replica. On receiving a query from a client, it selects a replica and returns to the client its network address. The client then sends its requests directly to this service replica. Whenever a client fails to contact a replica, it once again queries the registry.

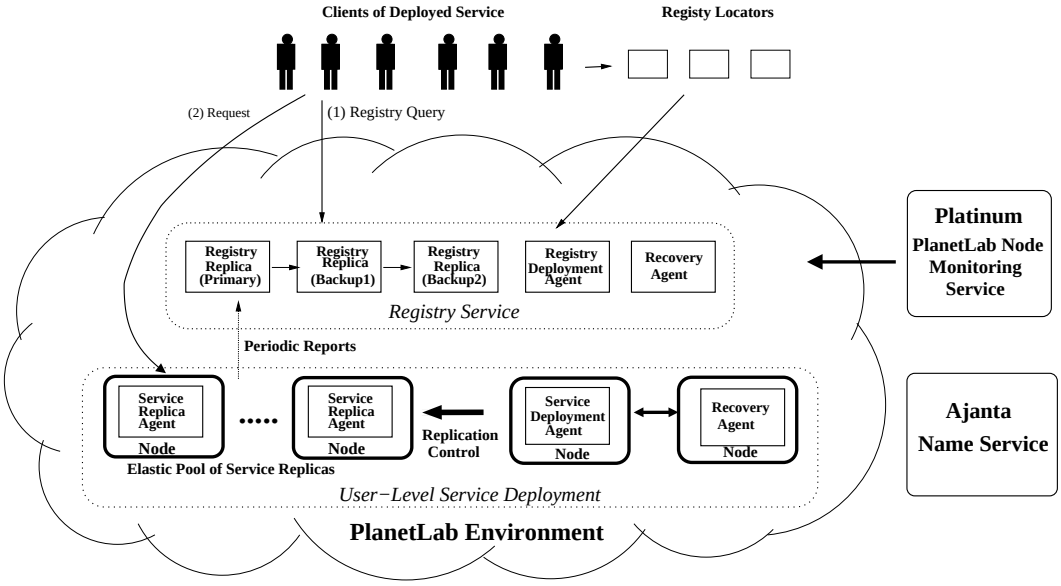


Figure 3.1: Ellora Framework for the Deployment of Resilient and Scalable Services

3.1.2 Service Replica Agent

The service replica agent component of Ellora provides the basic building-block for constructing a replicated service. It is extended by service developers to include request processing logic and service-specific replication management protocols. The service agent performs several generic functions central to the service deployment model of Ellora. The service agents include mechanisms for monitoring the available resource capacities and estimating the service capacity. Each service replica exchanges periodic heart-beat messages for decentralized group configuration management. These messages also include the load information to be used for load balancing. The heart-beat period T is set such that $w_p = k \cdot T$, for $k \geq 1$. In our experiments w_p is set to 1 minute and T is set to 10 seconds. A service agent contains mechanisms to monitor its load, and it also includes mechanisms to detect overload or saturation conditions. Based on the observed load conditions a service replica agent makes autonomic decisions such as shedding client load by redirecting the clients to other service replica agents. The details of these mechanisms are presented in Chapter 4.

3.1.3 Deployment Agent

For each deployed service, Ellora creates a special agent, called *Deployment Agent (DA)*, whose function is to perform dynamic control of the degree of the service replication. It creates and launches the service replica agents, and also detects their crashes. Each service replica agent sends a report message to the DA at each control cycle of w_p period, containing information about the number of requests received in the current cycle and the estimated service capacity for the next cycle. Using this information, the DA determines the aggregate load and the estimated aggregate service capacity. The load information reported by service replicas may not indicate the actual aggregate service load as it does not indicate the requests for which the clients failed to connect to the service replica. As for each failed request a client would perform registry relookup, the number of lookup indicates the number of failed requests. Therefore, DA queries the service registry to get the number of lookups performed in the given cycle and determines the aggregate service load by taking into account the number of registry lookups.

Based on the slack capacity model presented earlier, the DA decides if it should add more service capacity or reduce the existing capacity. If additional capacity is needed, DA 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 DA picks a set of high capacity nodes. The DA uses the packet probe method [Paxson, 1998] to ensure that the selected nodes have sufficient end-to-end bandwidth. 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 DA creates one or more service replica agents on the appropriate nodes. When the capacity is to be reduced, the DA selects the lowest capacity service agent and sends a 'terminate' message to the agent.

The DA uses the periodic reports from the service replica agents to detect their crashes. If no report is received from a replica in the current period, the DA suspects it as failed and

marks its contribution to aggregate capacity as zero. However, for such replicas it still uses the load value in the last report received from that replica. If no reports are received from a service replica agent for a certain number of intervals, the DA probes the agent. If the probe fails, it removes the failed agent from its configuration list. The DA then checks the aggregate service capacity to determine if a new agent needs to be created.

A failure of the DA results in the absence of the capacity scaling and the replication control functions for the service. The DA functions by maintaining as *soft state* the information about its current configuration of service replicas, the current load and the estimated service capacity. This state is based on the recently received reports from the service agents. The DA does not maintain any state on the stable storage. On crashes, this agent is simply restarted on any available node. For detecting the crash of the DA and to perform its restart, we pair it with another agent called *Recovery Agent (RA)*. The DA-RA pair of agents execute on different hosts and they exchange periodic heart-beat messages. Each agent in the pair is responsible for the relaunch of the other. The Recovery Agent has no other functions to perform, so it is a relatively a lightweight agent.

During the time when the DA is not available, at each reporting cycle the service replica agents would fail to make an RMI connection with the DA. In case of such RMI failures, they relookup the Ajanta Name Service for the DA and try once more. When the DA is restarted, it updates its RMI information with the Ajanta Name Service. In the next reporting cycle, the service agents would be able to communicate with this restated DA. After one cycle of reporting, the DA has complete information about all of the service replica agents and it is fully functional at that point.

The placement of service replicas requires selection of suitable nodes based on their available resource capacities. In order to profile nodes based on their resource usage behavior, we need accurate estimation of their resource usage characteristics. We have developed a system *Platinum* for monitoring PlanetLab nodes for their available capacities. It monitors available resource capacities on every node and assists the DA in selecting suitable nodes for deployment of service replicas. The details of this system are presented in Chapter 5

3.2 Registry Service Design

The Registry Service is a critical component because its presence is important for the clients to locate any of the currently operational replica agents of the target service. We design this service using the deployment mechanisms described above. The general mechanisms underlying the resiliency of this service are the same as those used for user-deployed services in Ellora, except for some bootstrap mechanism for the clients to locate this service initially.

The Registry Service is implemented using a replicated group of *Registry Agents*. The replication management of registry agents is based on the primary-backup model [Budhiraja et al., 1993, Wiesmann et al., 2000]. The deployment and replication of this service is dynamically controlled using the *Registry Deployment Agent (Registry DA)*. To locate the registry agents, the clients contact the *Registry Locators* which run at known locations. The registry locators are lightweight components and are used by clients only for bootstrapping purpose to locate the registry agents. The registry locators periodically query the Registry DA or any currently deployed registry agents to get the list of the currently deployed registry agents. A client can also query registry service agent to obtain the current configuration of the deployed registry agents.

3.2.1 Primary-Backup Model

The group of replicated registry agents operates in the primary-backup mode, with the backup agents operating in the active mode. All periodic reports from the service replicas are sent to the primary registry agent. However, a read request such as client query for service replica lookup can be served by any of the registry agents. The updates are propagated to the backup agents in a lazy manner, sequentially from one replica agent to the next, in a daisy-chaining fashion. Due to this lazy update propagation, a registry agent may have its registry data bit lagging for a few seconds, however, such inconsistencies are not critical to performance, as demonstrated in our evaluations.

The group of registry agents is dynamically configured in an ordered set, where the first agent in the set is the primary while the other agents are active backups. The registry updates are propagated to the agents in this order. The registry service agents coordinate amongst themselves in peer-to-peer manner and maintain the configuration in decentralized way. The

configuration is changed when an existing registry agent crashes or a new one joins the group. The configuration changes are identified by monotonically increasing version numbers. The primary agent is responsible for making the configuration change decisions and communicating the new configuration to the other registry agents.

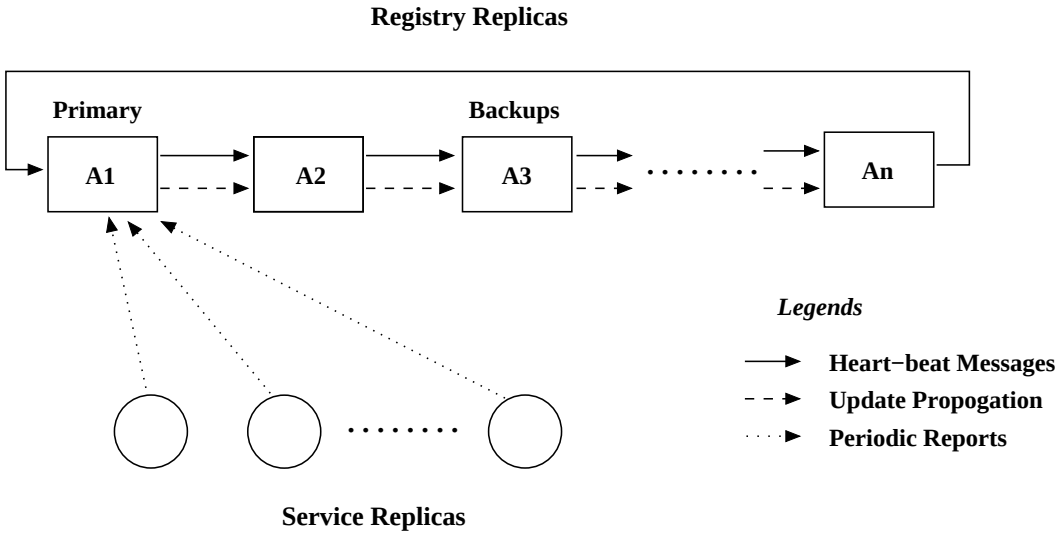


Figure 3.2: Service Registry Architecture

For detecting the crashes of the registry agents, each agent sends periodic heart-beat messages to the downstream agent that follows it in the ordered set. Therefore, if there are n registry agents denoted by A_1 to A_n, where A_1 is the primary, then agent A_i sends heart-beat messages to agent A_{i+1}. The failure of an agent is detected by its downstream agent if the heart-beat messages are not received for some period, i.e. A_i detects failure of A_{i-1}. Once the failure of an agent is detected, the primary agent is notified of the failure. The primary agent then removes the failed agent from the group, and sends configuration change message along with the new configuration to the current group members, and also to the Registry DA. If a primary agent cannot contact a registry agent while updating the new configuration, it marks it as failed and the agent is removed from the group configuration. In order to detect simultaneous failures of multiple agents, when an agent detects the failure of its upstream neighbor it also probes other upstream agents to check if they are failed. This is performed in following manner. When A_i detects failure of A_{i-1}, in the same cycle it also probes other upstream agents from A_{i-2} until

it finds an agent which is not failed. It then reports all the intermediate failed agents to the primary agent. Thus failures of multiple agent can be detected in a single cycle. The failure of primary agent is detected by the first backup agent and upon detecting the primary failure the first backup agent (if it is running) assumes the role of primary. The new primary then reports the failure of the primary and the new configuration to other agents. If first k agents are failed then the $k+1$th agent assumes the role of primary agent.

3.2.2 Correctness of Primary-Backup protocol

We want to ensure that the following properties hold to guarantee the correctness of primary-backup mode of operation.

- P1: Only one agent should be primary at any given time
- P2: All the agents in the replicated group should have the same view of the group configuration.

We assume that network level partitions do not occur. As discussed in our protocol all the group configuration changes are reported by primary to all other agents. Therefore, a backup agent's view of the current group configuration changes only when the primary reports a new group configuration. Hence ensuring the property P1 would also ensure that P2 holds.

In case of false detection of primary failure due to network connectivity issues the first backup agent would assume primary role while the primary is still running. In such cases our protocol ensures that only one agent acts as primary agent as follow. The first backup agent (if not crashed) would assume the primary agent is failed and report the new configuration to other agents. All the other agents would now consider this first backup agent as the new primary agent. When the old primary contacts any of the agents, it is notified of the new primary agent. The old primary then contacts the new primary agent. The old primary is then put at the end of the ordered list and the new primary agent notifies all the agents of the new configuration. We can see that in case of simultaneous failures of first k agents in the order this protocol would also guarantee that only one agent ($k+1$'th agent) acts as the primary agent. Thus it ensures that no two agents acts as the primary agent and hence properties P1 and P2 are satisfied.

During the update propagation, if a registry agent crashes after it has received an update but before propagating the update to the downstream agent then the update would not be propagated to the remaining agents. In case of Registry Service, the updates are basically information about service replica location, load, etc. Since such information is overwritten by periodic updates from service replicas, a lost update can be tolerated in this case. Therefore, we do not address the issue of lost updates here. However, mechanisms using sequenced updates and pulling the updates after group configuration changes can be implemented to avoid the lost update problems.

3.2.3 Recovery and Restart Mechanisms

All the components of the Registry Service – such as Registry DA, Registry Locators or Registry Agents – are lightweight and require minimal or no checkpointing. The fault tolerance and recovery of these components is discussed below.

The crash of a registry agent is reported by the primary registry agent to the Registry DA for immediate recovery of the failed agent, which then launches a new agent. The new agent contacts the primary agent in order to join the group. It joins at the last index in the group configuration and starts with an empty registry. Therefore, it waits for one reporting cycle before it starts responding to the client queries. During this period it redirects the client queries to other replicas. After one reporting cycle, it has information about all of the different deployed services and their replicas. If a service replica agent of some deployed service fails to contact the primary registry agent during a reporting cycle, it contacts any of the other registry agents to get the new configuration of the Registry Service. After that it attempts to communicate with the new primary agent for the Registry Service.

The failure and recovery of the Registry DA is handled using the Recovery Agent mechanism as discussed in the previous section. The Registry DA does not require any checkpointing. While the Registry DA is down, the Registry Service is still available. The registry agents can perform the group management functions in absence of the Registry DA. The only problem while the Registry DA is down is that no new agent can be created if any existing agent fails.

When a registry locator is started, it contacts the Registry DA to get the current configuration of registry agents. The URN of the Registry DA is the only information needed to execute it. A registry locator periodically refreshes its view of the current configuration of the Registry Service by querying any of the registry agents.

In this chapter we presented experimental framework we developed over PlanetLab for evaluation of our service scaling models and mechanisms. In this framework, the service deployment and scaling functions are performed by the Deployment Agent created for each service. We developed a registry service which is accessed by clients to locate the currently deployed service replicas. The fault-tolerance and high availability of this service itself is realized through scaling and resiliency mechanisms provided for user-deployed services. The different components of our framework such as the Deployment Agent or registry service operate by maintaining *soft-state* which can be easily generated upon their recovery.

Chapter 4

Adaptive Load Distribution

We address in this chapter the problem of how to distribute the load according to the service capacities of replicas which may fluctuate significantly over time. As discussed earlier, the fluctuations in service capacities of the individual service replicas necessitate the need for adaptive load distribution. In our system model, a client first queries a *Registry Service* for accessing a particular service. In response to the client query, the Registry Service selects one of the replica and returns its network address to the client. The client then sends subsequent requests to that service replica. A registry lookup is performed again by the client only after some number of requests. We have two different levels at which load distribution is performed:

- Registry-Level: This is the load distribution performed by the registry while selecting the service replicas for client queries. Since a registry lookup is performed by the client only after a certain duration, this level of load distribution operates at relatively coarse level.
- Replica-Level: For more fine-grain distribution of load, each service replica performs the load distribution based on its current load conditions.

The load distribution at replica-level is essential for two reasons. First, since the service capacities of replicas can fluctuate within short duration, load distribution mechanisms which operate at replica-level are needed for more adaptive load distribution. Second, with the replica-level mechanisms the load distribution can be performed even when the registry service is unavailable.

4.1 Registry-Level Load Distribution

The load distribution at registry 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(α)* of that replica. This load distribution fraction is given by the fraction of the total service capacity provided by that replica. Thus for replica k, if C_k is the service capacity, then load distribution fraction α_k of that replica is given by

$$\alpha_k = \frac{C_k}{\sum_{i=1}^{n} C_i} \tag{4.1}$$

At every control interval each service replica reports its service capacity to the registry service. Based on this reported service capacity, registry service calculates the load distribution fraction for each replica. When a client performs a lookup, the registry service selects one of the replicas randomly with the probability of the selection of a replica being proportional to its load distribution fraction.

4.2 Replica-Level Load Distribution

At a replica-level the load distribution is primarily performed by redirecting the load to another service replica. A replica may become overloaded due to the fluctuations in its service capacity or due to an increase in the client load distributed to it. In such cases, the replica needs to redirect a fraction of its load to another replica. For this purpose, we have following mechanisms of load redirection at replica-level.

1. *Request Redirection*: A service replica may redirect a single request to another replica. This is also called *temporary redirection*, since the subsequent requests are sent to the original service replica.

2. *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 registry lookup again.

3. *Forced Lookup*: A service replica may force the client to perform registry lookup again.

For the purpose of load redirection, 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 lookup?

The first problem is related to accurately assessing the current load conditions at a replica. 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 a control interval. As described in Section 2, in each control interval 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 request handling rate over the next interval. Based on this rate, at the beginning of an interval, the replica computes the number of tokens T_i representing the maximum number of requests that it can handle over that interval. 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 interval, 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 time t, then the replica is considered overloaded if

$$x_t > r_i * t \tag{4.2}$$

Similarly the replica is considered underloaded if

$$x_t << r_i * t \tag{4.3}$$

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

When a replica detects an overload situation, it needs to decide if it should perform request-redirection or client-redirection. In our model, a replica performs request-redirection in case of

low overload conditions. A client-redirection is performed in case of high overload conditions, which happen when the number of redirected requests exceed some threshold. In order to find target replicas for redirection, the replica needs information about load conditions of other replicas. For this purpose, every heart-beat period the replicas exchange load information such as the number of requests served, the number of tokens left and the current load status (underloaded, overloaded etc). 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 the 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_k be the number of tokens remaining of the target replica k then *redirection quota* q_k for the target replica k is calculated as

$$q_k = \frac{T_k}{(N/2)} \tag{4.4}$$

This *redirection quota* is calculated by considering that some other replicas may also redirect their requests to the given 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, it decides to force the client to perform registry lookup again. A 're-lookup' message is sent to the client in this case.

We also investigated a model based on characterizing the overload conditions as high and low overload and underload conditions as high and low underload based on certain thresholds. In this model a replica would perform client redirection only when it is in high overload condition and it would only select replicas in low underload condition for client redirection. However, we observed that this model only complicated the load distribution mechanisms and did not perform well. We realized that the load redirection based on the *redirection quota* provides more fine-grain and balanced load distribution.

For evaluating the load distribution mechanisms, we observe the deviation of the load shared by each replica from its expected load during each control interval. Let α_k be load distribution fraction of replica k for a given interval calculated using equation 4.4. Let S_k be the number of requests served by that replica in that interval and L be the total client load. Then the load deviation (β_k) of replica k is calculated as follow

$$\beta_k = |S_k - (\alpha_k * L)| \tag{4.5}$$

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:

$$\beta = \frac{\sum_{i=1}^{n} \beta_i / 2}{L} \tag{4.6}$$

The details of the evaluation of load distribution mechanisms are presented in Chapter 6. In this evaluation we also observe the impact of unavailability of service registry on the performance of load distribution mechanisms. Our load distribution mechanisms are agile in adapting to fluctuations in service capacities and load conditions as well as changes in replication configuration as demonstrated in our evaluations.

In this chapter we described the mechanisms for adaptive load distribution across the service replicas. These mechanisms operate at registry and replica level. We developed a token based model for fine-grain distribution of load at replica level. Based on this token model, the mechanisms at replica level perform load redirection to balance the load across service replicas.

Chapter 5

PlanetLab Monitoring Service

We have developed Platinum - a system for monitoring PlanetLab nodes for their available capacities for various resource such as CPU capacity, memory and network bandwidth. We observe that *CoMon* [Park and Pai, 2006], the node monitoring service provided by the PlanetLab, cannot be used for our purposes directly. This is because of the following reasons. We are interested in the average values for these resource capacities, and also in their variation over time. CoMon provides average values over system-defined monitoring intervals of one minute and five minutes. In our experiments, we need node-level resource utilization data that is collected at a higher frequency (such as at every 10-20 seconds) and aggregated to determine statistics over configurable observation intervals. This is important in order to obtain accurate measurements of a node's behavior over such intervals. The resource usage information of PlanetLab nodes provided by CoMon is relatively coarse grain for this purpose.

The Platinum node monitoring system is used to select nodes based on their available resource capacities for hosting service replicas. The *DeploymentManager Agent (DA)* queries this service to obtain the list of eligible nodes for deploying service replicas. We also use this service to study the resource usage characteristics of PlanetLab nodes. The Platinum service collects the data about resource consumption of every monitored node by probing its *SliceStat* [Park and Pai, 2006] data every periodic interval (in our experiments we set this interval to 10 seconds). We compute the available resource capacity at a node for a particular resource

type as the difference between the node's intrinsic resource capacity and the total usage for that resource for all the slices running on that node. We measure the average and variance of the available resource capacities over a sliding window of configurable interval (we use interval value of 5 minutes).

We select a set of nodes that have the average available resource capacity greater than a given requirement. These nodes define the *eligibility set* for the given requirement. The size of the eligibility set at a particular time for a given requirement indicates the number of nodes satisfying the given requirement at that time. A node is dropped from the eligibility set when it fails to satisfy the given resource requirement. The *eligibility period* of a node for a given resource requirement is measured as the time between the node's entry in the eligibility set for that resource requirement and departure from the eligibility set. A node may enter and leave the eligibility set multiple times during the observation period. Thus a node may have multiple eligibility periods. For such nodes, we consider the average value of their individual eligibility periods.

We investigated the following two approaches for selecting a node for inclusion in the eligibility set for a given requirement:

Basic Method: If C is the average idle capacity on a node and σ is its standard deviation, then for a given resource requirement R we select the node if it satisfies the following condition:

$$C - 2 * \sigma > R \tag{5.1}$$

A node is dropped from the eligibility set if the idle capacity at that node falls below the resource requirement R. When considering the CPU and memory requirements together for selecting nodes, we select the node only if it satisfies the above condition for both the CPU requirement and the memory requirement. We drop a node from the eligibility set, if either the available CPU capacity or available memory capacity on that node falls below R, the requirement threshold.

Profiling-based Method: In this approach we wanted to eliminate those node that show highly frequent and significant variation in their available capacity for a given requirement. In this

approach we build a *profiled eligibility set* from the basic *eligibility set* constructed using the basic approach presented above. The following rules are used for including a node in the *profiled eligibility set*. The rules use a parameter T, which is a time period value. We set it to 30 minutes in our experiments. A node in the basic *eligibility set* is considered for inclusion in the *profiled eligibility set* if its previously observed eligibility period was greater than T minutes. If the previously observed eligibility period of the node was less than T, then we include this node in the profiled set only after it has been in the basic eligibility set for the past T minutes. When a node is dropped from the basic *eligibility set*, it is also removed from the *profiled eligibility set*. In this approach, the eligibility period of a node is defined as the duration for which it remains in the *profiled eligibility set* for a contiguous interval.

We observe the distribution of eligibility set and eligibility periods for various resource requirements to characterize the resource availability in PlanetLab environment. We perform this study for CPU capacity requirements, memory requirements as well as CPU and memory conjoined requirements. For network bandwidth, the SliceStat service provides only average bandwidth usage over 1 minute, 5 minutes and 15 minutes. We need the resource usage data over a higher frequency to determine the average idle capacity as well as its standard deviation. The first important goal of this study is to compare the distribution of eligibility periods for CPU requirements with that of memory requirements. The key questions in this study are : How the size of eligibility set varies over time? How does the behavior of nodes vary for CPU and memory requirements? Do the nodes show more availability in terms of larger eligibility period and eligibility set size for memory requirements than those for CPU requirements? The second goal of this study is to determine whether the node availability is dominated by either the CPU requirement or the memory requirement, when both CPU and memory requirements are considered together. We also want to see if any relationship between the average eligibility of a node and the fraction of the time it is present in the eligibility set. Finally, another important goal of this work is to investigate how selecting nodes based on their recent profile affects the expected eligibility periods.

5.1 Evaluation of Node Selection using Basic Method

We present here the distribution of eligibility periods and eligibility set sizes for a spectrum of resource requirements selected using the basic method.

In the experiments discussed here we monitored about 200 PlanetLab nodes for their available resource capacities at different time periods. Table 5.1 shows the capacity requirements used in these experiments. We present here the observations for two datasets collected for duration of approximately 3 to 4 days. Dataset-1 was collected for duration of 75 hours from November 18-21, 2009 and Dataset-2 was collected for duration of 98 hours from December 1-4, 2009. During the period for which the Dataset-1 was collected, the monitored PlanetLab nodes were highly loaded while in the case of Dataset-2 they were relatively lightly loaded.

CPU	1GHz, 2GHz, 3GHz, 4GHz
Memory	512MB, 1GB, 2GB, 3GB
CPU+Memory	(1GHz + 512MB), (2GHz + 1GB) (3GHz + 2GB), (4GHz + 1GB)

Table 5.1: Capacity Requirements

Dataset	Time	Duration	Number of Nodes
Dataset-1	November 18-21, 2009	75 hours	200
Dataset-2	December 1-4, 2009	97 hours	200

Table 5.2: Datasets and their observation times

Figure 5.1 shows the CDFs of eligibility periods for CPU and memory capacity requirements for the datasets mentioned above. Table 5.3 presents statistics such as average, median and standard deviation for eligibility periods and eligibility set sizes for Dataset-1 and Dataset-2.

From Table 5.3, we observe that typically the median values for the eligibility periods tend to be always less than the average values. The standard deviation also tends to be high, comparable to the average values. This indicates that some nodes tend to exhibit significantly large eligibility periods. This also indicates that the available resource capacities at a node may fluctuate significantly, and there is a large variation of eligibility periods across the nodes.

The impact of memory requirements on node eligibility can be understood from the statistics presented in Table 5.3 and the cumulative distributions given in Figure 5.1. We found that very few nodes could satisfy memory requirement of 3GB. We can observe that nodes show high eligibility periods and eligibility set sizes for memory requirements than CPU requirements. This indicates that typically the available CPU capacity varies significantly compared to the available memory. The impact of combined CPU and memory requirements can be observed by comparing the cumulative distributions of eligibility periods given for the combined requirement in Figure 5.2 with those for the corresponding CPU and memory requirements shown in Figure 5.1. We observe that the distributions of eligibility periods for combined requirements tend to be close to the distribution of corresponding CPU requirements indicating that the availability of nodes for combined CPU and memory requirements is dominated by the CPU requirement.

To understand the distribution of eligibility set sizes we look at the statistics presented in Table 5.3. We find that the eligibility set sizes decrease with the increasing capacity requirements. However, one cannot draw such a generalization for eligibility periods. We observed that for a higher capacity requirement, fewer number of nodes become eligible but some of them remain in the set for a long time. We also observe that the variation in eligibility set sizes tends to be small. This indicates that there is always some constant number of nodes that can satisfy a given requirement. For example, in case of the 4GHz CPU requirement there are always more than 18 nodes available, and for 2GHz at least 36 nodes were in the eligibility set.

	Dataset-1					Dataset-2				
	Eligibility Period (minutes)			Eligibility Set Size		Eligibility Period (minutes)			Eligibility Set Size	
	Avg	Median	Std Dev	Avg	Std Dev	Avg	Median	Std Dev	Avg	Std Dev
1GHz CPU	315	46	472	103	9.29	522	145	874	64	15.3
2GHz CPU	103	34	221	51	6.24	367	50	553	35	6.4
3GHz CPU	218	31	376	38	3.15	423	356	412	30	4.02
4GHz CPU	163	40	284	25	3.72	799	359	1362	21	3.0
1GB Memory	650	438	494	105	2.7	1061	1022	578	84	8.4
2GB Memory	335	284	281	34	1.75	910	787	564	20	5.0
2GHz+1GB	119	48	256	30	4.8	392	53	552	20	5.06
3GHz+2GB	218	108	305	13	1.0	518	577	502	5	1.7

Table 5.3: Eligibility Period and Set Size Statistics for Basic Method

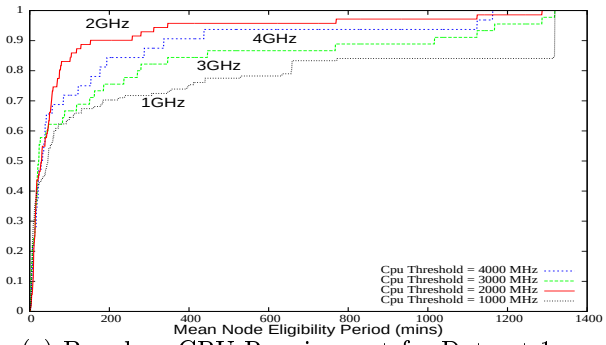
5.2 Evaluation of Node Selection using the Profiling Method

The results of our evaluations of the profiling approach for Dataset-1 and Dataset-2 are shown in Figure 5.3 and Table 5.4. We present the CDF of eligibility period for CPU requirements in Figure 5.3 and statistics for eligibility period and set size for both CPU and memory requirements in Table 5.4. These results show a clear and remarkable benefit of using profiling. For example, comparing the eligibility period values for the 2GHz requirement for Dataset-1 using the basic method with those with the profiled method, one can notice that the average period increases from 103 to 496 minutes, and the median value also increases from 34 to 258 minutes. As expected, the eligibility set sizes are always smaller in case of the profiled approach. This means we have a smaller set of nodes in the eligibility set but they are of higher "quality", i.e. they are likely to meet the given requirement for a longer time. In the data presented in Table 5.4, there was only one case where the values for the eligibility period using the profiled method were smaller than those with the basic method. This occurred for 1GB memory requirement in case of Dataset-1. We have not found any clear explanation for this case. Nonetheless, there is clear evidence otherwise that the profiling method identifies better quality nodes for the given requirement.

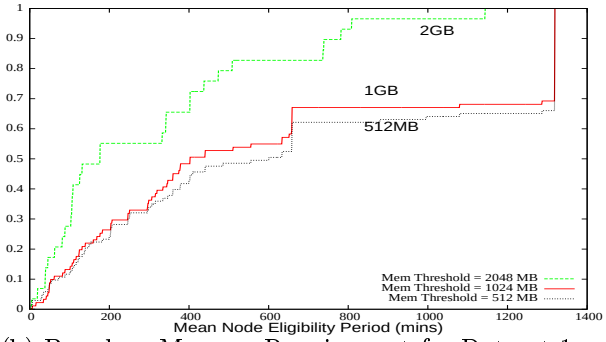
	Dataset-1					Dataset-2				
	Eligibility Period (minutes)			Eligibility Set Size		Eligibility Period (minutes)			Eligibility Set Size	
	Avg	Median	Std Dev	Avg	Std Dev	Avg	Median	Std Dev	Avg	Std Dev
1GHz CPU	740	359	895	50.46	16.4	786	195	1107	42.8	16.49
2GHz CPU	496	258	512	20.67	9.42	951	216	1287	34.6	8
3GHz CPU	552	386	557	15.6	8	1312	709	1681	24.3	5.5
4GHz CPU	356	134	452	11.9	4.6	1406	1129	1567	12.45	2.72
1GB Memory	455	340	393	28.39	14.4	2203	2126	1398	72.9	22.8
2GB Memory	957	1241	541	12.47	6.31	2469	2310	1232	21.74	6.95
2GHz+1GB	701	711	481	12.33	6.52	983	347	1127	26.93	7.28
3GHz+2GB	885	1160	599	4.95	2.82	1784	1784	1790	5.46	1.7

Table 5.4: Eligibility Period and Set Size Statistics with Profiling

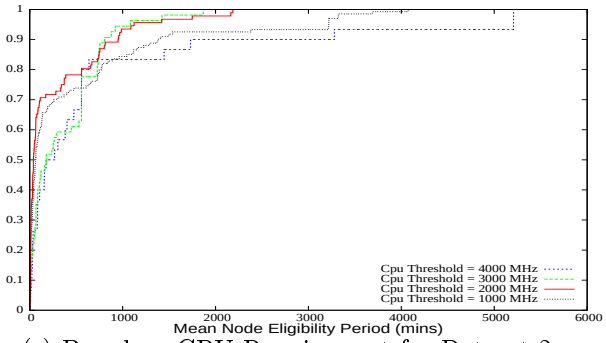
We found similar observations for other datasets collected at different times. Based on these observations we found that the profiling approach selects nodes which satisfy a given resource requirement for a longer duration compared to the basic method of node selection.



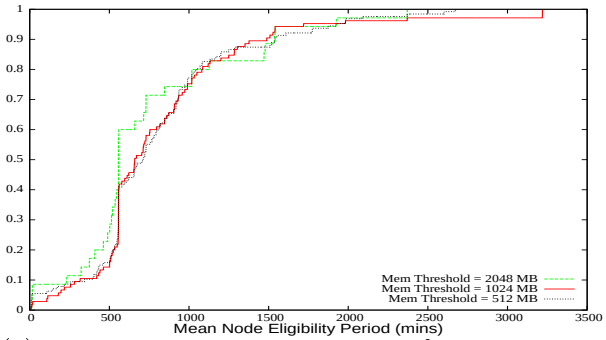
(a) Based on CPU Requirement for Dataset-1



(b) Based on Memory Requirement for Dataset-1

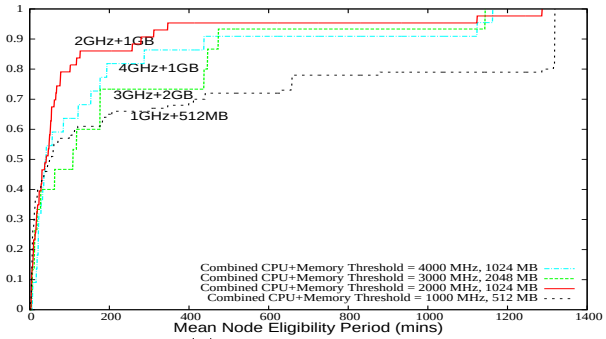


(c) Based on CPU Requirement for Dataset-2

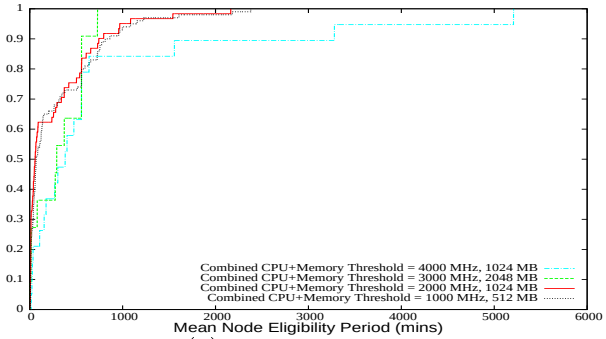


(d) Based on Memory Requirement for Dataset-2

Figure 5.1: CDF of Eligibility Periods for CPU and Memory Requirements

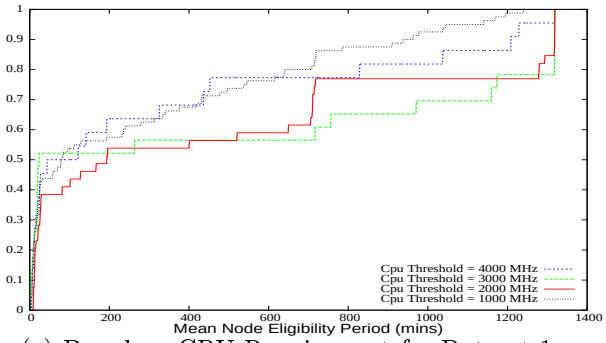


(a) For Dataset-1

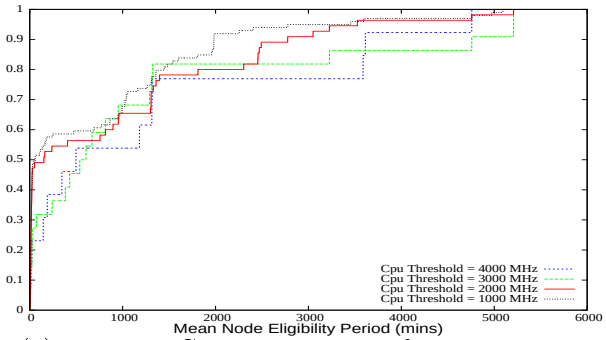


(b) For Dataset-2

Figure 5.2: CDF of Eligibility Periods for Combined Requirements



(a) Based on CPU Requirement for Dataset-1



(b) Based on CPU Requirement for Dataset-2

Figure 5.3: CDF of Eligibility Periods Based on Profiling (for CPU requirements)

Chapter 6

Evaluations

In this chapter we present the system-level evaluation of the autonomic service scaling mechanisms. Our first goal here is to evaluate the scalability of the dynamic capacity provisioning models and mechanisms The second goal is to evaluate the fault-tolerant aspect of our system. Specifically, we measure impact of replica crashes, performance of failure detection and recovery mechanisms, and impact of registry unavailability on load distribution.

6.1 Evaluation of Scaling Mechanisms

We evaluated the scalability of our system under load conditions similar to flash-crowds. We observed the impact of slack capacity level on the scalability and performance of the system. Our main performance measure in these evaluations was the average and 90-percentile response times observed by the clients. The load distribution mechanisms were evaluated by observing distribution deviation (β).

For this evaluation, the workload characteristics as well as the file set at the server side were generated according to the specification of SPECweb2009 E-commerce benchmark [SPEC,]. The total number of files on the server was around 15,000 with average file size of 16KB, variance of 18.5KB, and median and max sizes of 16.2KB and 40.5KB, respectively. The average file size for client requests was 2.8KB with variance of 5.5KB and median size of 1.3KB. For scalability

evaluation we generated flash crowd conditions as follow. During the initial phase, called the *low load phase*, we induced a constant load of 200 requests/second for 30 minutes. In the second phase, called the *flash-crowd phase*, the client load was increased every minute by 20 requests/second until it reached up to ten-fold, i.e. 2000 requests/second. The load was then kept constant at this level for about 30 minutes. We refer to it as *high load phase*. We measured the performance of the system in all three phases. We also observed the ratio of the load on the registry to the total service load. Since the clients were configured to perform registry lookup after every 100 requests, we expected this ratio to be at least 0.01. A registry load ratio value greater than 0.01 indicates registry relookups performed by clients due to connection failures or forced lookups issued by service replicas.

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, it 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 communication 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.

First, we present our evaluation of the impact of slack capacity levels. We evaluated the performance of slack capacity model for target slack levels of 30%, 20% and 10%. For 30% slack the low and high watermarks were set to 10% and 50% respectively. For 20%, they were set to 10% and 30%, whereas for 10% target slack low and high watermarks were 0% and 20% respectively. Table 6.1,6.2, and 6.3 presents the results of this evaluation for these slack levels for low load, flash-crowd, and high load phases denoted by P1, P2, and P3 respectively.

Table 6.1: Performance Statistics for Slack Level 30%

	30% slack (f_l=10%, f_h=50%)					
	Response Time(secs)			β		registry load ratio
Phase	Avg	Std.Dev	90 %ile	Avg	Std.Dev	
low load (P1)	0.466	0.538	0.812	0.13	0.15	0.028
flash crowd (P2)	0.972	0.875	1.42	0.14	0.12	0.059
high load (P3)	0.934	0.316	1.13	0.16	0.11	0.037

Table 6.2: Performance Statistics for Slack Level 20%

	20% slack (f_l=10%, f_h=30%)					
	Response Time(secs)			β		registry load ratio
Phase	Avg	Std.Dev	90 %ile	Avg	Std.Dev	
low load (P1)	1.34	1.26	2.35	0.16	0.13	0.024
flash crowd (P2)	1.91	1.89	4.51	0.16	0.08	0.068
high load (P3)	2.32	8.23	4.11	0.14	0.25	0.041

Table 6.3: Performance Statistics for Slack Level 10%

	10% slack (f_l=0%, f_h=20%)					
	Response Time(secs)			β		registry load ratio
	Avg	Std.Dev	90 %ile	Avg	Std.Dev	
low load (P1)	0.956	1.91	2.09	0.18	0.42	0.033
flash crowd (P2)	4.54	5.41	5.50	0.16	0.15	0.119
high load (P3)	2.57	0.698	4.07	0.29	0.17	0.067

Table 6.4: Replica Addition and Removal Statistics

	30% slack			20% slack			10% slack		
	P1	P2	P3	P1	P2	P3	P1	P2	P3
Avg. replicas	5	19	44	6	16	32	5	20	25
Added	3	43	10	6	40	7	4	39	5
Removed	1	6	5	4	16	4	3	19	4

Figure 6.1 shows the 90-percentile response times observed every 1 minute for 30% slack. Figure 6.2 and 6.3 show the same data for 20% and 10% slack. We can observe that slack level of 30% performs better in terms of response times in all three phases; the average and 90-percentile response times were bounded and only increased by a factor of two with ten-fold increase in the load. From Figures 6.2 and 6.3 we can observe that slack level of 20% and 10% performed poorly in the flash-crowd phase. In this phase the average response times increased by a factor of four for both 20% and 10% slack. Figure 6.4, 6.5, and 6.6 show the capacity generation and number of replicas for 30%, 20%, and 10% slack levels. Table 6.4 presents the statistics of replica creation and removal for these slack levels. We did not observe any node crashes in these experiments. For target slack of 30%, the average slack capacity provisioned was 42% whereas for target slack of 20% and 10% the average slack capacity was 27% and 16% respectively. The

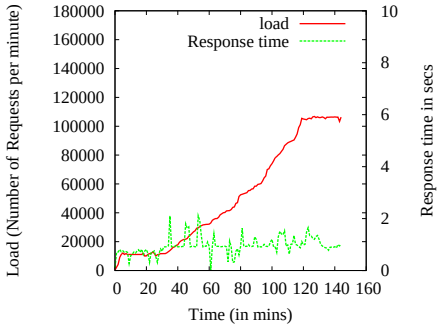


Figure 6.1: Response Times under 30% Slack with SPECWeb benchmark workload

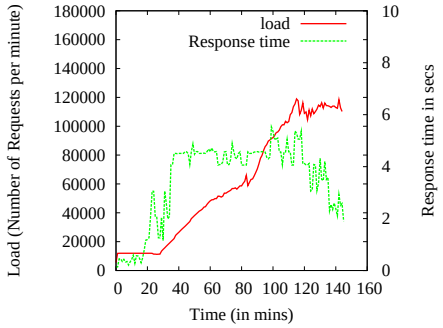


Figure 6.2: Response Times under 20% Slack with SPECWeb benchmark workload

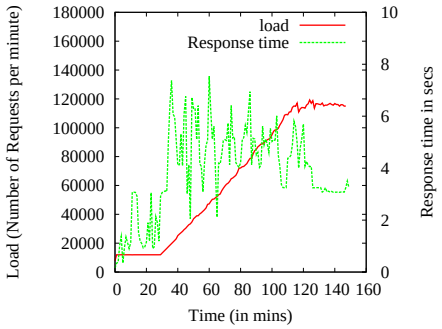


Figure 6.3: Response Times under 10% Slack with SPECWeb benchmark workload

registry load ratio was greater than 0.01 during all the phases and it increased in flash-crowd phase compared to the low load phase. The average β value observed for 30% slack was 0.138 indicating that on average 13.8% of the load deviated from the target distribution. Such a level of distribution deviation is tolerable since the additional capacity maintained in the system is at least 30% for this slack level. For 20% and 10% the average β value was 16% and 20%. For 10% slack the average deviation of 20% may not be tolerable because of the smaller slack value. We found similar observations for other experiment runs.

We also evaluated the scalability of our system with a different workload resembling the workload characteristics of 1998 World Cup website presented in [Arlitt and Jin, 1999]. We generated the files at server side resembling the file size distribution given in [Arlitt and Jin, 1999]. The average file size at the server side was 9.7KB with standard deviation of 9.6KB and median

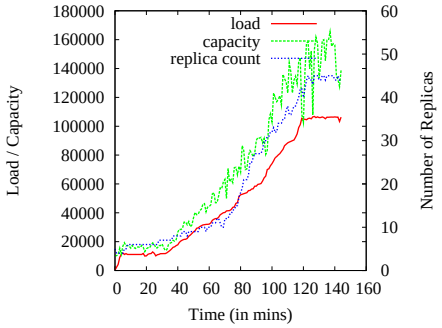


Figure 6.4: Capacity Generation under 30% Slack with SPECWeb benchmark workload

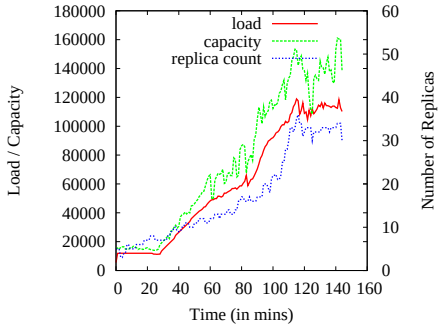


Figure 6.5: Capacity Generation under 20% slack with SPECWeb benchmark workload

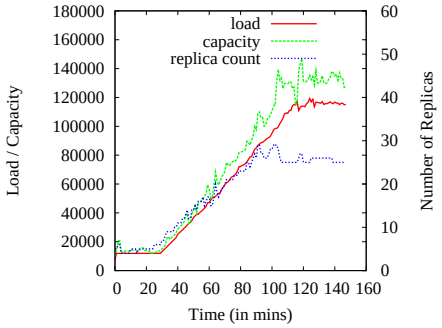


Figure 6.6: Capacity Generation under 10% Slack with SPECWeb benchmark workload

4.1KB. The average file size for client requests was 5.3KB with standard deviation 3.1KB and median 2.2KB. The flash crowd load was generated as discussed above. The slack level configured for this experiment was 30%. Table 6.5 presents the results of this experiment. Figure 6.7 and Figure 6.8 show the 90 percentile response times and capacity generation respectively.

6.2 Evaluation of Fault Tolerance

For evaluating the fault tolerance aspect of our system, we conducted following experiments. First we evaluated the impact of service replica crashes on the client side performance. Second we evaluated the impact of unavailability of the registry service on the load distribution. Finally we evaluated the failure detection and recovery mechanisms of the registry service.

Table 6.5: Performance Statistics for WorldCup workload

	30% slack (f_l=10%, f_h=50%)					
	Response Time(secs)			β		registry load ratio
Phase	Avg	Std.Dev	90 %ile	Avg	Std.Dev	
low load (P1)	2.56	1.91	3.09	0.11	0.34	0.03
flash crowd (P2)	5.8	6.11	7.61	0.19	0.25	0.065
high load (P3)	4.73	4.68	5.7	0.17	0.21	0.038

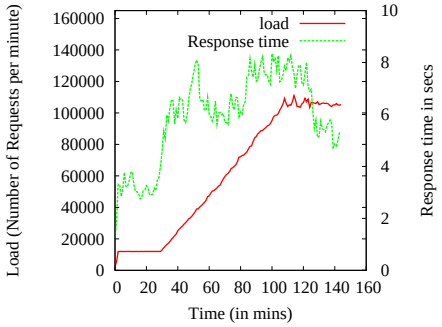


Figure 6.7: Response Times under WorldCup workload

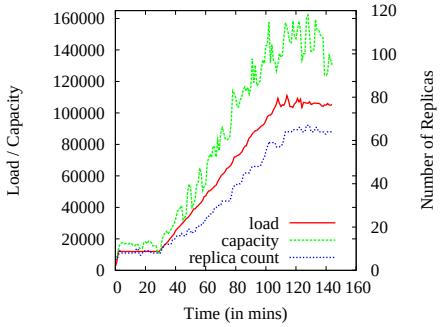


Figure 6.8: Capacity Generation under WorldCup workload

For evaluating the replica crash recovery mechanisms and to observe the impact of replica crashes on performance, we injected periodic crashes of randomly selected replicas. We induced a client load of 400 requests/second, and at every 30 minutes we injected a crash of a randomly chosen replica. We observed the 90-percentile values for response times over one minute intervals. This data is shown in Figure 6.9. The slack level set for this experiment was 30% with low and high watermarks values of 10% and 50%. The replica crashes were detected within two reporting intervals (i.e. 2 minutes) and additional capacity was generated within 30-60 seconds. The 90-percentile response times during normal conditions with no crashes were observed to be in the range of 0.4 to 0.7 seconds. From Figure 6.9 we can observe that the 90-percentile response times increased immediately after the replica was crashed but became normal again within three intervals. Since our capacity model ensures the system is not *1-replica crash vulnerable*, certain amount of extra capacity (10% in this case) would be still available after the replica crash. The increase in 90-percentile response times after the replica crash was mainly due to the clients which accessed the failed replica and had to timeout and relookup the registry.

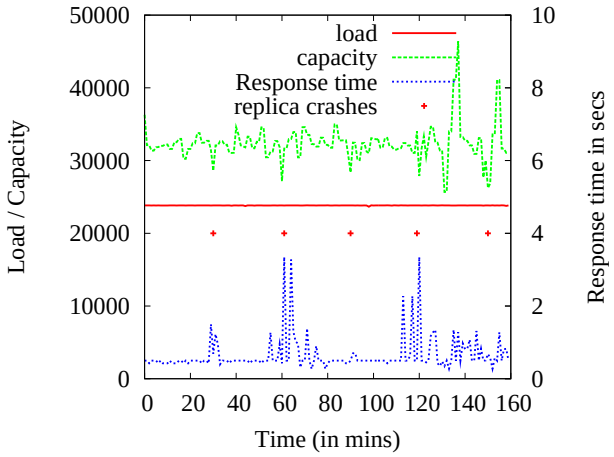


Figure 6.9: Impact of Replica Crashes

For evaluating the impact of registry unavailability on the load distribution we shutdown the Registry Service every 15 minutes and kept it unavailable for 10 minutes. We performed this experiment with 30% target slack and induced a constant load of 400 requests per second. The average β value for the period when the registry was available was observed to be 15.9%, whereas the average β value was 26.2% for the period when registry was not available. This increase in deviation may be tolerable due to the 30% slack capacity, however any new client would not be able to locate service replicas if the Registry Service is not available.

In evaluating the fault tolerance mechanisms of the registry service we wanted to measure the performance of failure detection and recovery mechanisms and to check the the correctness of group management protocols under various failure conditions. We deployed 5 registry agents on PlanetLab nodes. To evaluate the fault tolerance and recovery of registry agent failures, periodically a randomly selected registry agent was terminated. We observed that the failure of a replica is detected within 30 seconds and it requires additional 30 to 40 seconds for generating a new registry agent and updating the new configuration to all the registry agents.

To test the recovery mechanisms in case of multiple simultaneous failures, we terminated multiple registry agents at the same time. We injected failures of the primary and its $k-1$ immediate successor agents. In these cases the $k+1'th$ registry agent correctly detected these failures and assumed the role of the new primary. We also tested the system under catastrophic

failures where all registry agents except one were terminated. In these cases within 3 heart-beat periods the failures of all these agents were detected, and it took about 100 seconds additional for the new configuration with five registry agents to be operational.

For update requests, we measured the time taken to propagate the updates to all the registry agents. The average value for the update propagation delay was observed to be 23.7 seconds for 5 registry agents. We also measured the distribution of client requests across the deployed registry agents. As a client would randomly pick a registry agent for lookup, all the deployed registry agents are expected to share equal amount of load. Thus for 5 registry agents the expected fraction of load shared by each agent is 0.2. We observed that average deviation from this expected fraction of load was 6% for the duration of 2 hours with total 19 registry agent crashes.

The experiments described above performed evaluation of the integrated system and measured the overall performance and scalability of all the mechanisms discussed earlier such as service capacity estimation, capacity scaling and load distribution mechanisms.

Chapter 7

Conclusion

We have demonstrated here the performance of the autonomic mechanisms we developed for building scalable services on shared computing platforms without any node availability guarantees and resource capacity reservations.

Dynamic replication of service components is essential in such environments to cope with the fluctuations in available resource capacities as well as fluctuations in client load. We have presented here a model for service scaling through dynamic control of the degree of service replication. This model is based on maintaining a certain level of slack (excess) capacity in the system. We evaluated the impact of different slack levels on service performance. Since the resource capacity available at a node is not guaranteed, we need online models for predicting the resource capacity likely to be available in the near future at a given node as well as estimating the request handling capacity of the service replica hosted on that node. We presented here a model for predicting resource capacity available on a node based on its recently observed load conditions. We developed an online model for workload benchmarking to capture the per-request resource requirements. For adaptive load distribution, we developed mechanisms which operate at the registry as well as replica level. We developed a token-based model for fine-grain load balancing across the service replicas based on their service capacities.

For experimental evaluation of our service scaling mechanisms, we developed a framework over the PlanetLab. The components of this framework are designed for resiliency by relying

on soft-state based operations. We evaluated the performance of our service scaling models and mechanisms using this framework. Following are the important conclusions of our work.

- In this paper we presented an online model for predicting resource capacities likely to be available in the near future. Our experiments showed that for any desired confidence level, our dynamic prediction model can predict available resource capacity with 95% accuracy.
- We demonstrated here the benefits of the *capacity slack* model. We showed scalable performance with 30% slack for a workload exhibiting flash crowds with ten-fold increase in the load.
- We presented here mechanisms for adaptive load distribution which operate at two levels; registry-level (centralized) and replica-level (decentralized). These mechanisms limit the load deviation to around 15% even under conditions such as flash crowds. The load distribution mechanisms at replica level can distribute load with 75% accuracy even when the registry service is unavailable.

In summary, our work demonstrates that a large number of shared resources without any guarantee of available resource capacities can be utilized for building autonomically scalable and resilient services.

Bibliography

[Abdelzaher et al., 2002] Abdelzaher, T. F., Shin, K. G., and Bhatti, N. (2002). Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96.

[Amazon,] Amazon. Amazon ec2, http://aws.amazon.com/ec2/.

[Arlitt and Jin, 1999] Arlitt, M. and Jin, T. (1999). Workload Characterization of the 1998 World Cup Web Site. URL http://www.hpl.hp.com/techreports/1999/HPL-1999-35R1.html.

[Aron et al., 2000a] Aron, M., Druschel, P., and Zwaenepoel, W. (2000a). Cluster reserves: A mechanism for resource management in cluster-based network servers. In *In Proceedings of the ACM SIGMETRICS Conference*, pages 90–101.

[Aron et al., 2000b] Aron, M., Sanders, D., Druschel, P., and Zwaenepoel, W. (2000b). Scalable content-aware request distribution in cluster-based networks servers. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*.

[Bavier et al., 2004] Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., and Wawrzoniak, M. (2004). Operating System Support for Planetary-scale Network Services. In *NSDI'04: Proc.of the 1st Symp. Networked Systems Design and Implementation*, pages 253–266.

[Bodik et al., 2010] Bodik, P., Fox, A., Franklin, M. J., Jordan, M. I., and Patterson, D. A. (2010). Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 241–252.

[Budhiraja et al., 1993] Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). The Primary-Backup Approach. In Mullender, S., editor, *The Primary-Backup Approach*, pages 199–216. ACM.

[Cardosa and Chandra, 2008] Cardosa, M. and Chandra, A. (2008). Resource Bundles: Using Aggregation for Statistical Wide-Area Resource Discovery and Allocation. In *Distributed Computing Systems, 2008. ICDCS '08. The 28th International Conference on*, pages 760–768.

[Douglis and Foster, 2003] Douglis, F. and Foster, I. (2003). The Grid Grows Up. *IEEE Internet* , pages 24–26.

[Doyle et al., 2003] Doyle, R. P., Chase, J. S., Asad, O. M., Jin, W., and Vahdat, A. M. (2003). 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*, pages 57–71.

[Fox et al., 1997] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. (1997). Cluster-based scalable network services. *SIGOPS Oper. Syst. Rev.*, 31(5):78–91.

[Freedman et al., 2006] Freedman, M. J., Lakshminarayanan, K., and Mazières, D. (2006). 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.

[Marwah et al., 2003] Marwah, M., Mishra, S., and Fetzer, C. (2003). TCP Server Fault Tolerance using Connection Migration to a Backup Server. In *Proceedings of IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 373–382.

[Microsoft,] Microsoft. Microsoft azure, http://www.microsoft.com/windowsazure/.

[Oppenheimer et al., 2006] Oppenheimer, D., Chun, B., Patterson, D., Snoeren, A. C., and Vahdat, A. (2006). Service Placement in a Shared Wide-Area Platform. In *ATEC '06: Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, pages 273–288, Berkeley, CA, USA.

[Pai et al., 1998] Pai, V. S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. (1998). Locality-aware request distribution in cluster-based network servers. *SIGPLAN Not.*, 33(11):205–216.

[Park and Pai, 2006] Park, K. and Pai, V. S. (2006). CoMon: A Mostly-scalable Monitoring System for PlanetLab. *SIGOPS Oper. Syst. Rev.*, 40(1):65–74.

[Partridge et al., 1993] Partridge, C., Mendez, T., and Milliken, W. (1993). RFC 1546: Host Anycasting Service.

[Paxson, 1998] Paxson, V. (1998). On calibrating measurements of packet transit times. *SIGMETRICS Perform. Eval. Rev.*, 26:11–21.

[Shaikh et al., 2001] Shaikh, A., Tewari, R., and Agrawal, M. (2001). On the effectiveness of dns-based server selection. In *In Proceedings of IEEE Infocom*.

[Shen et al., 2002a] Shen, K., Tang, H., Yang, T., and Chu, L. (2002a). Integrated resource management for cluster-based Internet services. In *Proc.of the 5th Symposium on Operating Systems Design and Implementation*, pages 225–238.

[Shen et al., 2002b] Shen, K., Yang, T., and Chu, L. (2002b). Cluster load balancing for fine-grain network services. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 93.

[Shenoy et al., 2000] Shenoy, G., Satapati, S. K., and Bettati, R. (2000). HYDRANET-FT: Network Support for Dependable Services. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, page 699, Washington, DC, USA. IEEE Computer Society.

[Shivaratri et al., 1992] Shivaratri, N., Krueger, P., and Singhal, M. (1992). Load Distributing for Locally Distributed Systems. *IEEE Computer*, 25(12):33–44.

[Sollins and Masinter, 1994] Sollins, K. and Masinter, L. (1994). RFC 1737: Functional Requirements for Uniform Resource Names. Available at URL http://www.cis.ohio-state.edu/htbin/rfc/rfc1737.html.

[SPEC,] SPEC. SPECweb2009 benchmark. Available at URL http://www.spec.org/web2009/.

[Sultan et al., 2003] Sultan, F., Bohra, A., and Iftode, L. (2003). Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions. In *SRDS*, pages 177–186.

[Sultan et al., 2002] Sultan, F., Srinivasan, K., Iyer, D., and Iftode, L. (2002). Migratory TCP: Connection Migration for Service Continuity in the Internet. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 469, Washington, DC, USA. IEEE Computer Society.

[Tripathi et al., 1999] Tripathi, A., Karnik, N., Vora, M., Ahmed, T., and Singh, R. (1999). Mobile Agent Programming in Ajanta. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 190–197.

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

[Warns et al., 2008] Warns, T., Storm, C., and Hasselbring, W. (2008). Availability of Globally Distributed Nodes. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 279–284.

[Wawrzoniak et al., 2004] Wawrzoniak, M., Peterson, L., and Roscoe, T. (2004). Sophia: An Information Plane for Networked Systems. *SIGCOMM Comput. Commun. Rev.*, 34(1):15–20.

[Wiesmann et al., 2000] Wiesmann, M., Schiper, A., Pedone, F., Bettina, K., and Alonso, G. (2000). Database replication techniques: A three parameter classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 206–215.

[Wolski, 1998] Wolski, R. (1998). Dynamically Forecasting Network Performance using the Network Weather Service. *Cluster Computing*, 1(1):119–132.

[Wolski et al., 1999] Wolski, R., Spring, N. T., and Hayes, J. (1999). The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768.

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

[Zegura et al., 2000] Zegura, E. W., Ammar, M. H., Fei, Z., and Bhattacharjee, S. (2000). Application-layer Anycasting: A Server Selection Architecture and Use in a Replicated Web Service. *IEEE/ACM Transactions on Networking*, 8(4):455–466.

[Zhou and Yang, 2006] Zhou, J. and Yang, T. (2006). Selective early request termination for busy internet services. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 605–614.