Using Reinforcement Learning for Controlling an Elastic Web Application Hosting Platform

Han Li   Srikumar Venugopal

School of Computer Science and Engineering,
University of New South Wales, Australia
{hli,srikumarv}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-1104
April 2011

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Web applications have stringent performance requirements that are sometimes violated during periods of high demand due to lack of resources. Infrastructure as a Service (IaaS) providers have made it easy to provision and terminate compute resources on demand. However, there is a need for a control mechanism that is able to provision resources and create multiple instances of a web application in response to excess load events. In this paper, we propose and implement a reinforcement learning-based controller that is able to respond to volatile and complex arrival patterns through a set of simple states and actions. The controller is implemented within a distributed architecture that is able to not only scale up quickly to meet rising demand but also scale down by shutting down excess servers to save on ongoing costs.

1 Introduction

Enterprises are increasingly turning towards web-based applications to support operations such as customer relationship management and electronic mail. Many web applications also make their interfaces available as web services that are used to construct composite Web applications through paradigms such as service-oriented architecture (SOA) or mashups. These developments imply an increasing dependency on web applications and on the web platform as a whole.

Poor performance or failure of a web application can be disastrous not only to its users but also to other applications that depend on its services. However, unforeseen events such as rapid spikes in demand can lead to heavy load on the web applications and affect their response times and availability. The typical response to this situation is to provision more instances of the web application rapidly over which the excess load is distributed.

In the recent past, such an expansion would be limited by the physical server capacity available at the disposal of the service provider. Recent developments have led to the advent of Infrastructure as a Service (IaaS) (or cloud computing) wherein computing resources can be provisioned and de-provisioned by users on demand through a self-serve interface. The users are charged only for the capacity used on an on-going basis ("pay-as-you-go") with little or no upfront costs. Hence, IaaS has enabled the creation of hosting platforms for web applications that can scale elastically, limited only by the capacity of the infrastructure provider.

However, the main challenge in this environment is to devise a scheme for controlling computing resources used by the web application platform. The control scheme has to be quick to react to load-inducing events such as unpredictable spikes in demand. Since multiple web applications are supported on the same platform, an excessive load on one should not affect the performance of the others. When the demand has reduced, excess capacity should be de-provisioned to reduce ongoing infrastructure costs. Most importantly, the addition and removal of computing capacity should be without cascading effects on the rest of the applications or the platform.

Reinforcement learning [13] has been identified as a suitable technique for systems management for autonomic control of data centers [17]. The main advantage of using this technique is the elimination of the need to predefine explicit models for optimising performance metrics as the system develops its own knowledge base and rules from its interaction with the environment. In recent past, reinforcement learning has been applied to different problems in autonomic computing such as resource allocation [15], power management [5], and system configuration management [2].

In this paper, we apply reinforcement learning to the problem of controlling an elastic web application hosting platform. Our contributions are: i) a controller that is able to respond to complicated request arrival patterns through a series of simple discrete states and combined actions; and ii) a distributed architecture that incorporates a controller in every server and that is able to scale up quickly to meet excess demand and scale down to save on costs when the demand reduces. We demonstrate the efficacy of our scheme through experiments conducted on Amazon EC2.

The rest of this paper is organised as follows. In the following section, we describe our problem in depth and the model for the learning agent in our

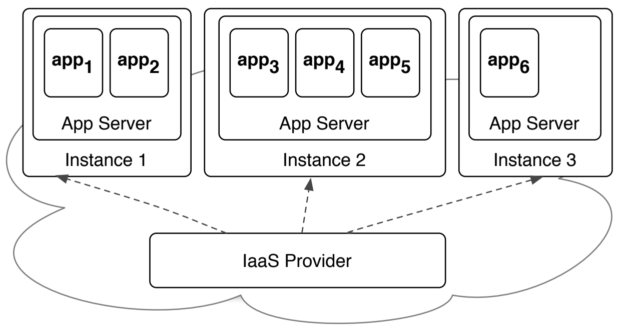


Figure 2.1: Hosting web applications on an IaaS provider

system. Then, we describe the application of reinforcement learning in Section 3. Section 4 discusses the actual implementation of our system. Section 5 describes the experiments and their results. Finally, we compare our system to existing solutions in literature and conclude the paper.

2 Problem Description

In this paper, we have targeted the standard 3-tier architecture for web applications [10]. The presentation tier deals with the user interface as shown via a web browser. The logic tier encapsulates implementation of business logic in modules running in an application server and the data tier deals with the management and storage of data in a separate database server, usually an RDBMS (Relational Database Management System). The web application is hosted in an application server such as Redhat's JBoss[1], IBM's WebSphere[2] or Oracle's Glassfish[3] running on top of Windows or Linux operating systems.

IaaS users are able to instantiate virtual machines using images that contain predefined web application hosting environments. A web application server is able to simultaneously host multiple applications with different requirements but are constrained by the resources of the underlying virtual machine. Therefore, a user may start multiple instances to host different applications leading to the scenario depicted in Figure 2.1.

An increase in the arrival rate for one of the applications results not only in a greater load on the application but also reduces the resources available for the other applications hosted in the application server. One solution to this is to move the heaviest loaded application to an application server on a new machine instance requested from the IaaS provider. This could lead to a sharp increase in costs, especially if the load on every application increases in a short interval leading to proliferation of new instances. Alternatively, it may be possible to deploy a copy of the application to an existing, lightly-loaded server. For example, in Figure 2.1, if *app3* on *Instance 2* is under a heavy load, it could be moved or duplicated to *Instance 3*. However, while this solution saves on

[1]http://www.jboss.org
[2]http://www-01.ibm.com/software/websphere/
[3]http://glassfish.java.net/

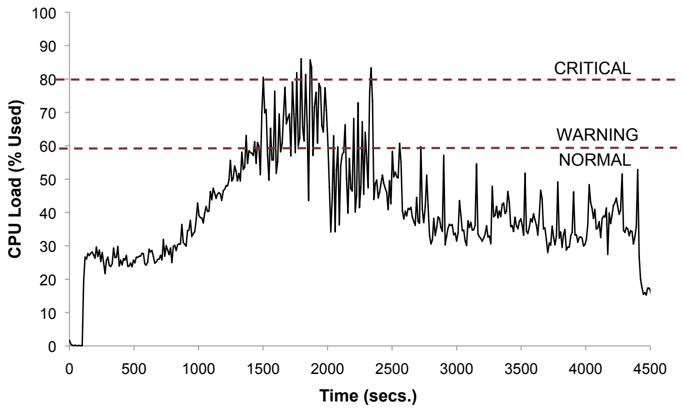


Figure 2.2: An example of CPU load determining state transitions

costs, it may lead to contention between the new and existing applications for the destination server's resources.

The goal of a control scheme for such a system is therefore to ensure that every web application is able to maintain its performance and that the ongoing infrastructure costs should be minimised while maintaining a steady state in which no server is overloaded. This can be split into two objectives. The first is to determine the smallest number of servers required to satisfy resource requirements of all the web applications. This is the *provisioning* problem [18]. The second objective is to distribute the applications among the servers such that each application is able to meet its response time and availability requirements. This is an instance of the *dynamic placement* [8] problem. Thus, the overall problem is a combination of these two problems.

In this system, at any particular instant, the number of instances of any application is variable as is the number of servers hosting them. New applications can be introduced any time into the system. Web traffic is considered "bursty" and heavy-tailed [11] and would differ for different applications. When a server becomes critically overloaded or an application overshoots violates its requirements, the controller has to execute a series of mitigating actions. Also, when a system is under-utilised, the controller must be able to de-provision servers by moving applications to other under-utilised servers.

A centralised controller would, therefore, need to keep track of a dynamic system with multiple variables changing simultaneously, both at the server and application levels. This complexity is reduced in a distributed architecture where monitoring, management and feedback at the server level is performed by a local controller [21]. This also makes the system agile and adaptive to rapid changes in workload. Therefore, in our work, we have created a fully distributed control scheme where each server is a learning agent that contributes to and shares a common knowledge base.

2.1 Learning Model

Consider a scenario with m servers. The overall state of a server is a combination of its system state and the states of the applications running on them. The system state is an indicator of the load on the actual physical or virtual machine. We use the percentage of CPU used (denoted as p_{CPU}) as the primary metric to determine its system state. This is because the CPU is a load-dependent resource, that is, its usage reflects the intensity of the load on the server [8]. However, to make the state space discrete, we have applied an upper and lower threshold to p_{CPU} so that the measures can be classified into three categories: normal, warning and critical. As an example, Figure 2.2 illustrates the classification of CPU usage levels for a single server using data from our experiments (Section 5). When the CPU usage increases above the lower threshold (0.6 in this example), the server's state switches from normal to warning, and then from that to critical when the usage goes above the upper threshold (0.8). It then returns to below the lower threshold, or switches to the normal state.

An application's state is described with respect to a metric that determines its performance, such as its average response time. Here, the response time is the time between the arrival of a request at the server and the time the response is sent. We have applied an upper and lower threshold to the response time and the application state is classified in the same manner as the system state. Therefore, the state of a server, $s =< s_0, s_1, \ldots s_n >$ where s_0 represents the system state while $s_1 \ldots s_n$ are the states of the n applications being hosted by the server.

Reinforcement learning is concerned with how to learn a control policy (a mapping from the states of environment to control actions) so as to maximise a cumulative reward signal. Let s_t be the state of the server at time t. The controller selects an action a_t from a finite set A as a result of which the system transitions to state s_{t+1} at time $t + 1$. The controller earns a reward $R(s_t, a_t, s_{t+1})$ for this transition. The reward of a state-action pair should reflect the satisfaction of performance requirements of the applications, the state of the server after the action is taken and the impact of the action. Therefore, we define the reward function as follows:

$$R(s_t, a_t, s_{t+1}) = R(s_{t+1}) + R(a_t) \tag{2.1}$$

where $R(s_t)$ is the reward for reaching state s_t defined as:

$$R(s_t) = \sum_{0 \le i \le n} R_i \tag{2.2}$$

where

$$R_i = \begin{cases} -c & \text{if } s_i = critical \\ 0 & \text{otherwise} \end{cases}$$

that is, R_0 is assigned based on the state of the server, and $R_i (1 \le i \le n)$ are assigned based on the states of each of the n applications on the server. $R(a_t)$ is the cost brought to the entire system by the execution of a_t and is further detailed in Section 3.2.

We have applied Q-Learning [23], a widely used reinforcement learning technique, to this problem. In this strategy, the agent calculates the quality (or Q-value) of a state-action combination, denoted by $Q(s, a)$, from its interaction with the environment using the formula:

$$\begin{aligned} Q_{t+1}(s_t, a_t) = {} & Q_t(s_t, a_t) + \\ & \alpha_t[R(s_t, a_t, s_{t+1}) + \gamma max Q_t(s_{t+1}, a_t) - Q_t(s_t, a_t)] \end{aligned} \tag{2.3}$$

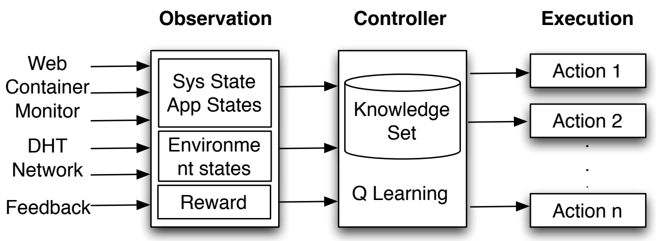


Figure 3.1: Abstract view of the controller

where α is the learning rate determining the extent how the newly acquired information will override the old one. A factor of 0 makes the controller learn nothing, while a factor of 1 makes the controller consider the most recent information only.

We have adopted the policy of drawing the actions with probability ϵ from the Boltzmann distribution [23]:

$$p_t(s,a) = \frac{e^{Q_t(s,a)/\tau}}{\sum_{a^* \in A} e^{Q_t(s,a^*)/\tau}} \tag{2.4}$$

where $p_t(s,a)$ is the probability of selecting action a in state s, and τ is a temperature parameter that tunes the randomness of selecting the actions. τ is an annealing factor that is decreased in every iteration thereby reducing randomness as well. When τ approaches 0, the algorithm becomes a greedy algorithm where the action with highest value is always chosen. By including randomness in the choice of the action made by the agent in all states, we allow the agent to explore the state-action space thoroughly in order to learn better.

3 Applying Q-learning

The web application platform is a dynamic distributed environment in which servers are constantly instantiated and terminated, and which has no constant size. Therefore, instead of having a central controller for the entire system, we decided to embed a controller into each server. The servers are connected in a peer to peer network underpinned by a Distributed Hash Table (DHT) that provides the lookup functionality.

Figure 3.1 illustrates an abstract view of the controller that is part of every server in the system. A monitor component on the server (detailed in the next section) provides observations of the server and the applications on it to determine the local system and application states. Observations from other servers are fetched via the DHT to obtain the environment states. These are then passed to the controller. Another input is the value of reward given to the controller for actions performed in previous steps.

The Knowledge Set (KS) in the controller is a database that maintains the knowledge learned from each iteration of the Q-learning algorithm. The KS is empty at the initial stage. As the learning process goes on, new states emerging at each iteration are inserted into the KS, along with a list of candidate actions

Table 3.1: List of Actions

`DO NOTHING`
`START SERVER`
`FIND SERVER`
`TERMINATE SERVER`
`MOVE`
`DUPLICATE`
`MERGE`
`START SERVER AND MOVE`
`FIND SERVER AND MOVE`
`START SERVER AND DUPLICATE`
`FIND SERVER AND DUPLICATE`
`MOVE AND TERMINATE SERVER`

and the corresponding Q-values. In other words, the KS consists of an array of detected states, and every state has a number of actions with a Q-value for each action.

3.1 Local States

As explained in Section 2.1, for each server and application, we map the usage and performance metrics respectively to one of three states - normal, warning and critical. By categorising the states, heterogenous measurements are mapped into globally uniform values. Therefore, the possible states of a server and application are reduced to three discrete levels which makes the solution tractable. Another merit of this mapping is that the performance measures being observed can be changed without affecting the controller.

A moving average model is applied to p_{CPU}, to smooth out sudden fluctuations that may be mistaken for state changes and to reflect the trend of the CPU load. The moving average value of p_{CPU} is computed as:

$$p_t = \beta p_{cpu} + (1 - \beta) p_{t-1} \tag{3.1}$$

where p_t is the average CPU usage at time t, and β is the moving rate ($0 \leq \beta \leq 1$). p_t is then mapped to one of the states described previously. In practice, the server state has a heavier influence on the selection of actions than the application state and therefore, each of the states is further divided into three sub-states in order to provide a more fine-grained view.

The application state is classified by its average response time, which is the primary performance metric for web applications. The average memory occupied by a web application is used to distinguish it as `LIGHT` or `HEAVY`.

3.2 Actions

We define a set of 7 base actions that are combined in various ways to control server provisioning and application placement operations in the system. Similar to the state space, the base actions consist of three server actions and four application actions.

The server actions are described as follows:

- `START SERVER`: This action requests a new instance from the IaaS provider and sets it up to receive applications from the rest of the system. This action is usually invoked when the existing servers become insufficient for handling the increasing load and consequently leads to an increase in the infrastructure expenditure. This is also termed *scaling up*.
- `TERMINATE SERVER`: This action shuts down a server and terminates the instance, thereby *scaling down* the system. While this action decreases the overall resource capacity, it also reduces the expenditure on servers. This action is executed when the load has decreased and the system has become "over-provisioned". However, this action cannot be executed within the first 30 minutes of a server's existence.
- `FIND SERVER`: When a server is experiencing heavy load, this action induces a negotiation process with other servers so as to find a server that is capable of accepting an application. The negotiation is driven by the reward function described in the following section. The incentive here is to make the best use of the existing resources to avoid adding new servers and incur the extra cost.

The application actions are as follows:

- `DUPLICATE`: Create a copy of a web application on another server. The new instance of the application is then able to share the load of the original instance. This is commonly employed while scaling up.
- `MOVE`: Move a web application from one (source) server to another (destination) server so as to shift the load. This is different from the `DUPLICATE` action as the copy of the application on the source is deleted and all the requests for the application is redirected to the destination server. `MOVE` is employed during scale up to relieve a server of a heavily-loaded application that is depriving other applications of CPU and memory. During scale down, `MOVE` is employed to divest a server of all applications, so that it can be safely shut down.
- `MERGE`: When one web application has multiple copies in different servers, the `MERGE` action undeploys a copy and then shifts all the load to the remaining instances. This is the reverse of `DUPLICATE`, and the eventual effect of this action is to withdraw the extra resources assigned to an application to deal with a period of excessive load.
- `DO NOTHING`: Perform no action under the present circumstances. This action is selected usually when a server's capacity is considered enough to handle the load on its applications.

Note that almost all application actions deal with a source and a destination server. Therefore, the control system executes a combination of a server and an application base actions. So, a typical combined action is made up of a server action followed by an application action. For example, the combined action `START SERVER AND MOVE` adds a new server and then moves an application to it; `FIND SERVER AND DUPLICATE` identifies an existing lightly-loaded server and duplicates a heavily-loaded web application on it. To make the action space tractable, we have predefined 12 actions that cover all of the operations that the controller need to execute. These are listed in Table 3.1.

In case of server actions, the reward for `START SERVER` is negative as it incurs an extra cost. In contrast, `FIND SERVER` and `TERMINATE SERVER` earn positive rewards as they save on costs. Application actions earn positive rewards since `MOVE` and `DUPLICATE` improve an application's performance and `MERGE` paves the way for shutting down servers. However, the rewards for application actions are smaller than those for server actions.

For combined actions, the net reward is a sum of the rewards of the individual actions. For example, given an absolute value of 3.5 for server actions and 0.5 for application actions, the action `START SERVER AND MOVE` incurs a penalty of -3 while `MOVE AND TERMINATE SERVER` earns a reward of 4.

3.3 Updating Q-values in Knowledge Set

To choose an appropriate action in response to the current state, the controller first tries to locate the state in the KS. The existence of the state depends on whether a similar situation has been encountered before. If it is not found, the current state is classified into a general category (e.g. system in critical / warning state, application in critical / warning state) so that a group of candidate actions (with initialised Q-values) can be assigned to this state. This action clustering strategy significantly reduces the computational expense of the algorithm and accelerates the process of exploration. The environment states are used to identify inapplicable actions and the candidate action set is refined by eliminating these.

After the execution of action a_t drawn from the probability $p_t(s_t, a_t)$, the server enters the next state s_{t+1}. The reward of the state-action combination $R(s_t, a_t, s_{t+1})$ is calculated from Equation 2.1. The Q-value of (s_t, a_t) is updated with Equation 2.3.

Each server is a learning agent in our system, which aims to converge to an equilibrium. The controller publishes a copy of its Q-values to the peer-to-peer network among the servers. All the controllers later update their own Knowledge Set with the others' Q-values. With such a sharing strategy, the knowledge of each agent is populated very quickly and the learning process is accelerated. The update of Q-values marks the end of one complete episode of the learning, and the process is repeated as long as the server is not terminated.

4 Implementation

Figure 4.1 illustrates the software architecture on a server in our system. While all the servers share the same software base that is saved in a CentOS Linux Virtual Machine (VM) image, one of the servers takes on the role of the proxy server which acts as the first point of contact for all clients of the web applications. Only the proxy server has a public static IP address known to the clients. Applications are hosted on the backend servers that have private IP addresses. The backend servers are transparent to the client, so they are able to join and leave the system seamlessly.

The Load Monitor in the proxy server periodically performs a lookup via the peer network to determine the load on each server in the system. It enters this data into the Server DB. This database also stores the mapping between each web application and the servers on which they are hosted. When a request

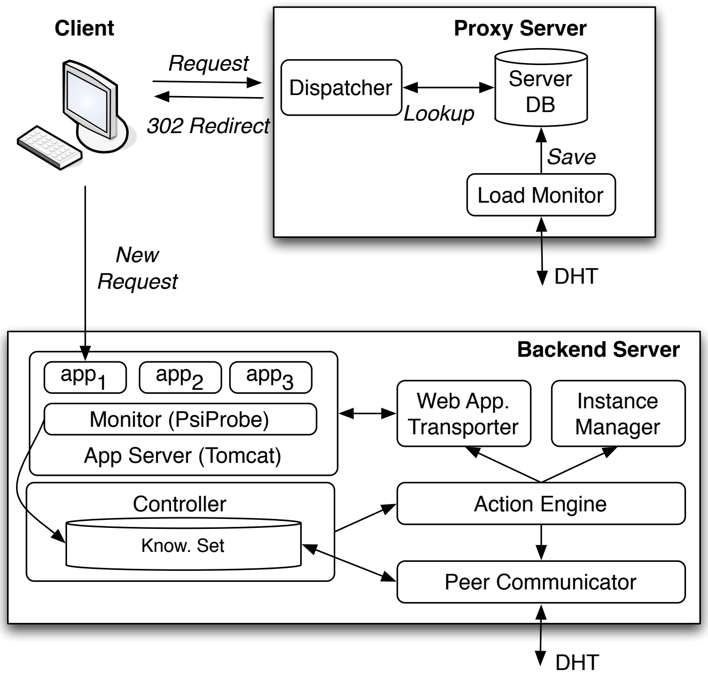


Figure 4.1: Server architecture

from a client arrives at the Dispatcher, it performs a lookup on the Server DB and selects the least loaded server on which the application is hosted. The Dispatcher then responds to the client with an `HTTP 302 Redirect` message along with the URL of the app on the selected backend server. For performance reasons, the Server DB is held entirely in memory.

4.1 Backend Server

The biggest component of the backend server is the application server that hosts the web applications. In this paper, we have targeted enterprise web applications that follow the J2EE standard. At present, we support Apache Tomcat[1] as the web application container/server. However, our mechanism can be extended to any J2EE-compliant server such as JBoss[2] and Glassfish[3].

Other components of the backend are associated with the reinforcement learning-based control mechanism described in the previous section. The controller is an implementation of the Q-learning based controller and interfaces with a knowledge set which stores the data about states, actions and rewards in a MySQL[4] database. CPU utilisation statistics are gathered using the `mpstat` utility and are used to determine local system state. The performance statistics for the web applications are collected by PsiProbe[5] which is a tool for real-time

[1]http://tomcat.apache.org/
[2]http://www.jboss.org/
[3]http://glassfish.java.net/
[4]http://www.mysql.com/
[5]http://code.google.com/p/psi-probe/

monitoring of Apache Tomcat instances. PsiProbe returns data on metrics such as the response time for each web application in the container, size of the application and the session size in memory. The response time is used to determine the local application states.

The Action Engine performs the actual execution of the actions in conjunction with the Instance Manager, Peer Communicator and Web Application Transporter (WAT). The Peer Communicator interfaces with other servers via the DHT. The Instance Manager carries out the task of managing the starting and termination of servers while the actual transfer of applications among server is executed by the WAT. We describe these components in detail as follows.

Action Engine

As described in Section 3.2, the action space consists of combine actions composed of 7 base actions. The Action Engine maintains a control process which decomposes the combined actions into base actions which are then executed in a specific order. For example, `FIND SERVER AND DUPLICATE` is executed in the following three steps:

1. Decide which application to duplicate
2. Negotiate with other servers to find out which server is willing to accept this application
3. Duplicate the application to the targeted server.

The criteria for selecting an application to be moved or duplicated is different for either action. For the `MOVE` action, the *heaviest* application (in terms of session and application size) will be chosen if the system is scaling up, whereas the *lightest* application is selected while scaling down. The reason for this strategy is that while scaling up, the overloaded source server should offload and reduce its workload to the largest extent, whereas while scaling down, we should avoid shifting large workloads between servers. In case of the `DUPLICATE` action, an application in the `CRITICAL` state is always chosen since it has the most urgent need for extra resources.

Negotiations with other servers is carried out via the Peer Communicator and is prompted by the `FIND SERVER` action. When load has to be shifted off the server, the selected web application and the action to be executed are broadcast to all the backend servers that are in the normal state. The controller at a receiving server computes a response (either `accept` or `reject`) based on the current local state s_t. If multiple servers are ready to accept the application, then the least loaded server is selected for the transfer. If an `accept` response is not received, then an exception is raised and the Action Engine defaults to the `START SERVER` action.

Since the reward for accepting applications is positive, a receiving server is incentivised to send an `accept` response unless the application is already hosted on it or if it has transitioned to a `WARNING` or `CRITICAL` state since the message was sent. Therefore, this means that the entire system will consistently aim to balance the load among existing servers.

Instance Manager

The Instance Manager interfaces with the IaaS Provider via the latter's APIs. In our work, we use Amazon Elastic Compute Cloud[6] as our IaaS provider. The Instance Manager manages the lifecycle of a server instance. The `START SERVER` action prompts the Instance Manager to acquire and start a new instance. Once ready, the Instance Manager on the new server takes control of its lifecycle.

When all servers are in the normal state, there is no transfer of applications. Therefore, one or more of the servers will eventually pick the action with the next highest reward, which is to terminate itself. When a server decides to shut itself down, it will aim to shift out all its applications via a combination of `MOVE` and `MERGE` actions. It will also stop accepting new applications unless one of its peers is in a critical state. This mechanism prevents the system from going into a situation where applications are forever moving around the servers.

While there is a chance that all servers may simultaneously decide to terminate, the probability of such an event is very low, given the different request arrival patterns for each application and the different times at which state changes occur in different servers. In practice, we have discovered that it is generally the least-loaded server with the smallest number of applications which is able to terminate itself first. This is a preferred outcome since a server in such a situation is utilised poorly.

Web Application Transporter

The WAT is responsible for transporting web applications among backend servers. After the web application is selected and the destination server is identified by either `START SERVER` or `FIND SERVER`, WAT starts the transportation. A J2EE-based enterprise web applications is required to be packaged as a standard WAR (Web Application Archive) file which contains the JSP (Java Server Pages) files, Java classes, XML files, tag libraries and static Web pages (HTML and related files) that together constitute the Web application. In response to a `MOVE` action, the WAT copies the WAR file from the source server to the destination server, deploys it in the Tomcat container at the destination, and undeploys it from the source server. In case of a `DUPLICATE` action, the application is retained on the source server. Afterwards, the WAT sends a message to the proxy server to update its mapping of the application locations in its Server DB accordingly.

4.2 Role of the DHT

Our system employs the Kademlia DHT [9] which plays an important role in ensuring reliability and scalability of the entire system. We have used Mojito[7], an implementation of Kademlia in Java, for communication between the servers. The servers exchange four types of messages as described below.

- Server event notification: When one server is added to or retrieved from the environment, it should announce this event to all the servers in DHT network, so that the other servers are aware of the scale and status of the current system.

[6]http://aws.amazon.com/ec2/

[7]http://wiki.limewire.org/index.php?title=Mojito

Table 5.1: Amazon EC2 Instance Types

Instance Type	EC2 Comp. Units	Mem. (GB)	Price (USD/hr.)
Small Instance	1	1.7	0.085
High-CPU Medium Instance	5	1.7	0.17

- Application list on each server: The backend servers report a list of web applications that are hosted by them to the proxy server to distribute the incoming requests.
- States of backend servers: The system state of each backend server is also pushed to the peer network periodically. The backend servers need this information for their learning, and the proxy server uses this information to distribute the requests in order to balance the load among the servers.
- Q-value updates: The Q-values acquired by each learning server are synchronised and shared by each agent. Therefore, whenever the Q-value for a state-action pair is updated on a server, it is pushed to the DHT and other agents will look it up when they update their Knowledge Set.

5 Evaluation

This section studies the performance of our reinforcement learning-based control scheme for hosting web applications on Amazon EC2. We evaluate our platform in terms of application performance in the face of varying and extreme workloads, and the elasticity exhibited by our platform.

5.1 Experimental Setup

For the sets of our experiments, we have used a High-CPU medium (c1.medium) EC2 instance type with the specifications shown in Table 5.1 as the proxy server which is acting as the load balancer. The load balancer is responsible for the redirection of all the requests from the clients.

We used Apache JMeter[1], a load testing tool, to emulate the HTTP clients and measure the performance of the applications. JMeter was run in headless (no GUI) mode and was deployed on a separate EC2 instance in the same region to reduce the effect of network latency. Multiple copies of JMeter were deployed on several EC2 instances to generate a very high load (thousands of requests per minute).

Backend servers were also deployed on EC2. Two types of EC2 instances (`Small` and `High-CPU medium`) were used in different sets of experiments. This attempted to discover the effect of increasing the capacity of a single server on the quality of service of the whole system. Each backend server hosted one web

[1]http://jakarta.apache.org/jmeter/

Table 5.2: Parameter settings for Q Learning

Parameters	Value
Learning rate α	0.38
Discount rate γ	0.98
Initial exploration temperature τ	4.0
Temperature annealing τ_d	0.99
Average move rate β	0.2

container (Tomcat 6.0) with multiple web applications in it. In addition, the Java Virtual Memory (JVM) was allocated 1GB of memory. The container was configured to a maximum number of 1000 threads so as to handle the large volume of requests.

Experiments were performed to test our system's capability of autoscaling in a multi-application environment. The system was initialised with one proxy server and one backend server that hosted 6 independent web applications. We employed a synthetic web application that simulated a hotel room reservation management system. The main actions were searching for an available room on a particular date range, booking the room, and then generating and emailing a unique confirmation code to the guest. We set a lower threshold of 200 milliseconds and an upper threshold of 500 milliseconds for executing the request.

Five of the six applications on the backend server were subjected to a background load drawn from a uniform random distribution. The remaining application was stressed using 3 different workload patterns - one that rises to a maximum and then gradually reduces (Peaking), one drawn from a power law distribution (Power law) and one drawn from a normal distribution - to test the elasticity of our system under different scenarios. The system was initialised with an empty database before starting every experiment. The following three metrics were chosen to measure the effect of the learning algorithm on the system and application performance:

- **Average Response Time**: This data was recorded by PsiProbe and was only the time spent in executing the request on the server. Therefore, network latency was removed from consideration.
- **Drop Rate**: This is the ratio of the number of requests that were not served (or "dropped") to the total number of requests sent to the system.
- **Number of Servers**: The count of backend servers that are active in the system.

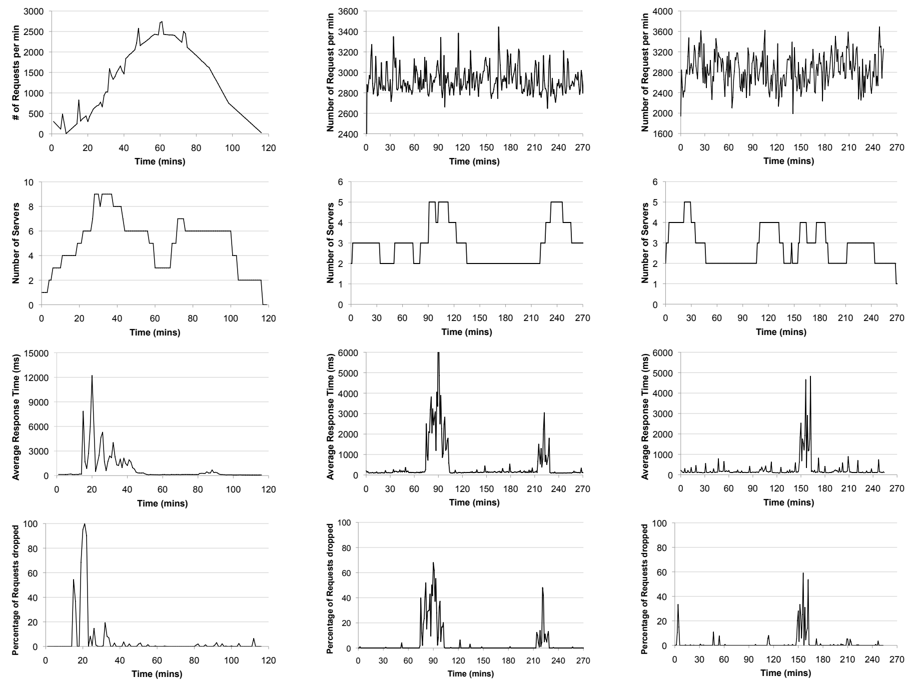
The Q-Learning controller for each server was initialised with the parameters shown in Table 5.2.

5.2 Results

Table 5.3 shows the overall results of our experiments. Overall, in each of the experiments, we were able to serve more than 96% of the requests under fluctuating workloads. The second column of Table 5.3 lists the amount of time a server spends under the upper threshold of CPU usage, that is, outside of the critical state. This value is expressed as a percentage of the total uptime of the server and averaged over all the servers started during the experiment. The

Table 5.3: Aggregate results

Experiment	Requests Accepted	Time Below Critical
Peaking	97.82%	98.72%
Power Law	96.19%	94.78%
Normal	98.27%	98.26%



(a) (b) (c)

Figure 5.1: Resource provisioning under: (a) peaking load with small instances (b) power law with high CPU instances and (c) normal with high CPU instances.

high values for this measure indicates that the controllers are able to respond quickly to excess demand and return the overall system to a steady state.

Figure 5.1 presents a more fine-grained view of the experiment results. The topmost or first row is a plot of the input load versus time, the second row plots the number of servers, the third row graphs the average response time and the fourth row plots the drop rate versus time. All the plots in a column belong to a single experiment and share the same timeline. Therefore, within a column, the bottom three rows are the response of the system to the input shown in the first row.

Peaking Load

Figure 5.1(a) depicts the performance of our system when tested with a load that peaks and then decreases. This experiment was performed with EC2 Small Instances. The background load in this test case was 60 req/min per application or 360 req/min for the whole system, since there are 6 web applications per server. As mentioned previously, only one web application was targeted at a time. In this test scenario, executed over 120 minutes, the number of requests to the target application was first increased from 60 req/min to 2400 req/min, then decreased back to 60 req/min. Occasional fluctuations of 300 req/min, represented by the spikes of the curve, were also emulated so that the scenario was more realistic.

The system quickly scaled up from 1 server to 9 servers in the first 30 minutes to meet the increasing load. However, each server takes around 3 - 5 minutes to come online and start serving requests. Therefore, the system's performance degraded in a transient period where the average response time reached 12 secs (24 times the threshold) and the drop rate increased to over 90%. However, the system stabilized well before the peak was reached. Consequently, the response time was reduced to well within the lower threshold, and the drop rate fell to 0.

While the requests are still increasing between 30 to 60 mins, the system determines that it is over-provisioned and starts shutting down servers gradually. After a minor adjustment, the system reached its balance at time 78 mins. At the end, the system scaled down in face of the declining workload. The key results here are that after an initial transient period, the system was able to maintain the response time and drop rate well within bounds even in the face of an increasing load. Minor perturbations in the load did not affect the learning algorithm as the number of server follows the long-term trends in the arrival rate.

Varying workloads

The next two experiments consider more realistic workloads drawn from distributions commonly seen for web applications. The requests for the second experiment follow a power-law probability distribution function. We have used EC2 High CPU instances for this experiment. The last experiment explored a workload whose average is constant over the long-term but fluctuates in short periods. We derived this request arrival pattern from a normal distribution with a mean (μ) of 2800 requests/min and the variance (σ) of 400 requests/min. We have used EC2 High CPU instances for this experiment.

For both the experiments, the system was able to maintain a consistent application performance well within the threshold for most of their durations. For the power law distribution, the number of servers required was either 2 or 3 for most of time, except for an increase to cover a degradation in response times between 60 and 90 minutes, and between 210 and 220 minutes into the experiment. A similar trend is seen in the last experiment where the number of servers oscillated between 2 and 4 for most of the duration of the experiment. Additionally, number of servers follows the long-term trend of the input arrival rates.

5.3 Discussion

The experiments illustrate the overall efficacy of the system in ensuring that the performance requirements of the applications are met while ensuring that unwanted servers are terminated to save on ongoing costs. However, there are a few caveats.

The first and second experiments demonstrate that initially the system may not be able to meet increasing demand by provisioning enough servers in time. One of the factors is the probability of the controller choosing the wrong actions for the state. As this probability of selecting an action is controlled by the annealing factor τ (Equation 2.4), the initial selection of actions is more random than learned. While the controller could be initialised with a set of initial state-action pairs, this could lead to the system being led towards a predefined provisioning plan rather than one learned purely from the environment. It could be argued that the transient spikes that occur during the initial period when the system is learning, are a trade-off to the longer term benefits of using reinforcement learning to control the elasticity of the system.

The other important factor is the time taken for a server from being provisioned to be online and serving requests. Here, an option is to tune the learning algorithm to consider the startup time either in the rewards or as part of the state transitions.

6 Related work

The problems of dynamic provisioning of resources for web applications have been extensively studied in the literature. A common approach to provisioning has been the use of models to approximate the behaviour of the system so as to predict the number of required resources [6, 18, 19, 20, 22, 12, 7]. Most of these efforts revolve around modeling web applications and the hosting environments as a system of queues. However, this requires knowledge about the long-range dependencies in the request arrivals as well as impact of these on the system performance. Moreover, these efforts have focused on scaling the infrastructure to meet the needs of a single, multi-tier web application. It has been hypothesised before that developing accurate models for complex, distributed systems may not be feasible [17]. Also, the model-based approach may not be adaptable to the appearance of new arrival patterns, as the queueing models being used are based on the assumption of certain arrival processes. Therefore, this approach may not be suitable for the scenario of hosting multiple web applications on an elastic platform as described in this paper.

Dynamic placement of web applications among different servers has also been studied extensively. Karve, et. al [8] model the dynamic placement problem as an optimisation problem and introduce an online heuristic algorithm to solve it. Tang, et. al [14] propose an faster algorithm that assigns applications to servers based on the solution of a maximum flow problem. However, both these efforts deal with placement of applications over a statically defined set of servers. Also, their algorithm does not explicitly model a cost of a placement change, so the impact of a new placement on the performance of applications is not well studied. Moreover, they only aim at equalising CPU utilisation across servers, whilst disregarding response times of an application on different nodes as a function of server utilisation. Al-Qudah, et. al [1] explore the problem of efficiently enacting placement decisions to improve the agility of a dynamic platform hosting multiple web applications in a server.

The research presented in this paper is closely related to the work performed by Tesauro, et. al [15] to develop a utility-function driven resource allocation mechanism using reinforcement learning in the Unity [4] system. Their architecture associates an Application Manager with each application hosted in a data center. The Application Manager computes a utility curve that estimates the value gained as a function of the servers allocated to that application. The utility curves are submitted to a Resource Arbiter that allocates servers so as to maximise the value over all applications. The learning is carried out within the Application Manager and was later extended to incorporate a hybrid approach wherein a queueing model was used to provide the initial policy (also called HybridRL) [16].

While there are similarities between the work presented in this paper and Unity, notably the use of decentralised agents, there are several differences. In Unity, each application is given exclusive control of a resource while in our system, the server is shared between multiple applications. Our system is elastic and incorporates actions to reduce ongoing cost whereas Unity was not designed for that goal. The decision making mechanism in our system is fully decentralised while the Arbiter makes the final decisions in Unity. Another important difference is the manner in which the coordination between the agents is carried out in both systems. Agents in Unity learn from the resources allocated in response to their submitted utility curves and the reward gained in return for meeting SLAs with the resources. In our system, the agents coordinate via negotiations and exchange of Q-values.

7 Conclusions and Future Work

To summarise, we have proposed and implemented a reinforcement learning-based mechanism for controlling an elastic web hosting platform underpinned by resources from an IaaS provider. The learning strategy uses a simple and restricted set of actions to control the elasticity of the platform in the face of complex and unpredictable request patterns. We have implemented this mechanism in a distributed software architecture where each server is an independent controller able to take any action to control its load. However, the servers are able to negotiate with each other to share excess load incentivised by an appropriately structured reward function.

We have experimented with various workloads on resources provisioned from Amazon EC2. The experiments show that, overall, the system is able to maintain performance in the face of rapid fluctuations in the workload by provisioning servers as necessary. However, there are periods in which the thresholds are violated due to insufficient number of resources. This is caused by the time required for the controllers to learn and for bringing new servers online.

At present, the web applications considered in our system are independent. Future work will involve extending the system to deal with a service-oriented architecture where each application invokes other applications hosted on the same platform. We also plan to extend the elastic capability to n-tier enterprise applications in the J2EE environment.

Finally, we plan to improve the controller to further reduce the amount of violations of application performance requirements. A method that we have identified is to have a set of servers on standby for scaling applications instantly. However, the controllers need to coordinate better to control the number of standby servers which would require further exploration of multi-agent reinforcement learning [3] techniques.

Bibliography

[1] Zakaria Al-Qudah, Hussein A. Alzoubi, Mark Allman, Michael Rabinovich, and Vincenzo Liberatore. Efficient application placement in a dynamic hosting platform. In *Proceedings of the 18th international conference on World wide web*, pages 281–290, Madrid, Spain, 2009. ACM.

[2] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. A reinforcement learning approach to online web systems auto-configuration. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems (ICDCS '09)*, pages 2–11, 2009.

[3] L. Busoniu, R. Babuska, and B. De Schutter. Multi-Agent reinforcement learning: A survey. In *Control, Automation, Robotics and Vision, 2006. ICARCV '06. 9th International Conference on*, pages 1–6, 2006.

[4] D.M. Chess, A. Segal, I. Whalley, and S.R. White. Unity: experiences with a prototype autonomic computing system. In *Proceedings of the First International Conference on Autonomic Computing (ICAC '04)*, pages 140–147, 2004.

[5] Rajarshi Das, Jeffrey O. Kephart, Charles Lefurgy, Gerald Tesauro, David W. Levine, and Hoi Chan. Autonomic multi-agent management of power and performance in data centers. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track*, AAMAS '08, pages 107–114, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[6] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin M. Vahdat. Model-based resource provisioning in a web service utility. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, pages 5–5, Seattle, WA, 2003. USENIX Association.

[7] Dejun Jiang, Guillaume Pierre, and Chi-Hung Chi. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th international conference on World wide web*, pages 471–480, Raleigh, North Carolina, USA, 2010. ACM.

[8] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *Proceedings of the 15th international conference on World Wide Web*, pages 595–604, Edinburgh, Scotland, 2006. ACM.

[9] Petar Maymounkov and David Mazires. Kademlia: A peer-to-peer information system based on the xor metric. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin / Heidelberg, 2002.

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

[11] R. Morris and Dong Lin. Variance of aggregated web traffic. In *Proceedings of Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings (INFOCOM 2000)*, volume 1, pages 360–366 vol.1, 2000.

[12] A. Quiroz, Hyunjoo Kim, M. Parashar, N. Gnanasambandam, and N. Sharma. Towards autonomic workload provisioning for enterprise grids and clouds. In *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing (GRID 09)*, pages 50–57, 2009.

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[14] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller for enterprise data centers. In *Proceedings of the 16th international conference on World Wide Web*, pages 331–340, Banff, Alberta, Canada, 2007. ACM.

[15] G. Tesauro, R. Das, W.E. Walsh, and J.O. Kephart. Utility-Function-Driven resource allocation in autonomic systems. In *Proceedings of Second International Conference on Autonomic Computing (ICAC '05)*, pages 342–343, 2005.

[16] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of IEEE International Conference on Autonomic Computing (ICAC '06)*, pages 65–73, 2006.

[17] Gerald Tesauro. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, 11(1):22–30, 2007.

[18] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC '05)*, pages 217–228, 2005.

[19] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 291–302, New York, NY, USA, 2005. ACM.

[20] Daniel Villela, Prashant Pradhan, and Dan Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Trans. Internet Technol.*, 7(1):7, 2007.

[21] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proceedings of First International Conference on Autonomic Computing (ICAC'04)*, pages 70–77, 2004.

[22] XiaoYing Wang, DongJun Lan, Gang Wang, Xing Fang, Meng Ye, Ying Chen, and QingBo Wang. Appliance-Based autonomic provisioning framework for virtualized outsourcing data center. In *Proceedings of the Fourth International Conference on Autonomic Computing (ICAC '07)*, page 29, 2007.

[23] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.