Adaptive Resource Allocation for Back-end Mashup Applications on a Heterogeneous Private Cloud

Anonymous submission

Abstract—On-demand resource provision is one of the key features of cloud computing, allowing applications to grow or shrink on the basis of dynamic workloads. Thus far, most of the research on leveraging on-demand resource provision has focused on batch style applications for scientific computation or Web-centric applications, especially n-tier e-commerce applications. Very little research, however, has focused on leveraging the benefits of cloud computing for applications with alternative architectures. In this paper, we focus on Back-end Mashup applications that have resource-intensive back ends responsible for continuous collection and analysis of real-time data from external services or applications. We present a working prototype back-end mashup application, BuddyMonitor, that allows users of instant messaging services to monitor and analyze the online presence of their “buddies.” The prototype exploits adaptive allocation of cloud resources to scale gracefully in the presence of rapid increases in workload. We demonstrate the feasibility of the approach in an experimental evaluation with a testbed cloud and a realistic simulation of a large scale external XMPP chat service. We conclude that cloud computing with adaptive resource allocation has the potential to increase the usability, stability, and performance of large-scale back-end mashup applications.

Keywords—cloud computing, instant messaging, adaptive resource allocation, back-end mashups, private clouds.

I. INTRODUCTION

A cloud is a combination of physically and virtually connected resources. Virtualization allows us to instantiate virtual machines dynamically on physical machines and allocate resources to them as needed. Virtualization is thus one of the key technologies behind cloud computing infrastructures. There are several benefits expected from virtualization, such as high availability, ease of deployment, ease of migration, ease of maintenance, and low power consumption, that help establish a robust infrastructure for cloud computing. Many IT giants, such as IBM, Sun, Amazon, Google, and Microsoft, offer cloud-based computational and storage resource rental services to consumers. Consumers of these services host applications and store data for business or personal needs.

Cloud computing is rapidly growing as a research area spanning both academia and industry. The vast majority of the research has been oriented towards either batch-style applications for scientific computing or towards Web-centric applications, especially e-commerce applications. The typical scientific application uses single-instruction multiple data (SIMD) parallelization, without any dependencies on external services or between separate internal tiers. The typical Web application consists of a few tiers that are coupled, under the control of the application provider, and relatively easy to scale vertically or horizontally to meet performance goals. The needs of these types of applications are well understood, and cloud computing can fulfill them through on-demand resource provisioning, pay-per-use, and robust infrastructure.

Our research focuses on adaptive scaling of the virtual resources allocated to applications running on clouds. There is some existing work in this area. Amazon Auto Scaling [1] allows consumers to scale up or down according to criteria such as average CPU utilization across a group of compute instances. [2] uses log-based monitoring and adaptive resource allocation to identify service-level agreement (SLA) violations in single-tier Web applications and scale the applications to satisfy the SLA. [3] presents the design of an auto scaling solution based on incoming traffic analysis for Axis2 Web services running on Amazon EC2. [4] presents a statistical machine learning approach to predict system performance and minimize the number of resources required to maintain the performance of an application hosted on a cloud. Dubeyb et al. [5] present initial results from the use of dynamic regression and queuing modeling techniques to obtain an approximate system performance model for multi-tier Web application hosted in virtualized data centers. [6] monitors the CPU and bandwidth usage of virtual machines hosted on an Amazon EC2 cloud, identifies the resource requirements for Web-based applications, and dynamically switches between different virtual machine configurations to satisfy the changing workloads.

Very little research has been done, however, on how to leverage the benefits of cloud computing for applications with alternatives to these well-understood architectures. In this paper, we explore adaptive resource allocation for applications following an architectural pattern we call Back-end Mashup, in which the user-facing front end might be a relatively simple and lightweight Web interface, but the back end is a resource-intensive system that continuously collects and analyzes real-time data from external services or applications.

As an example back-end mashup, we have built a prototype application called BuddyMonitor that connects to any external chat server running XMPP (the eXtensible Messaging and Presence Protocol) and allows end users to monitor their chat “buddies” for specific durations of time and to examine their buddies’ presence patterns over time. We find that as the number of concurrent monitors increases, BuddyMonitor's resource demand on back-end virtual machines increases fairly rapidly, requiring allocation of additional virtual machines to accommodate additional requests without overloading the system.

To enable dynamic allocation of cloud resources for back-
end mashup applications such as BuddyMonitor, we have built a prototype system that adaptively allocates resources to the application, ensuring that it can always accommodate new requests despite a-priori undefined resource utilization requirements. We evaluate the prototype resource allocator on a heterogeneous testbed cloud running BuddyMonitor and demonstrate that it prevents the application from reaching an unstable state despite growing resource utilization requirements.

In the rest of this paper, we describe BuddyMonitor, our approach to adaptive resource allocation for back-end mashups, the prototype implementation, and an experimental evaluation of the prototype.

II. Buddy Monitor: An Example Back-end Mashup Application

Instant messaging (IM) provides real-time text communication between two or more people. Google, Microsoft, Yahoo, and Skype are some of the well-known IM providers. Most IM providers use XMPP to offer IM services. We developed a back-end mashup application named BuddyMonitor in Java using the Smack [7] API that is able to connect to any XMPP server using credentials provided by an end user. Users can elect to monitor the presence or availability of a list of their IM buddies in real time for a particular duration of time, and the system records that presence information for later visualization and analysis.

In BuddyMonitor, a user’s request to monitor a specific set of buddies for a certain duration of time is known as a user monitor request. Fig. 1 shows the availability and presence patterns for five of one of the authors’ real buddies on Google Chat on a specific day. The y axis shows the sampled presence code indicating whether the user is Unavailable (code = 0), Away/Busy (code = 1), or Available (code = 2).

![Fig. 1. Availability and presence patterns for five buddies of a real Google Chat user for a specific day. The y axis indicates sampled presence codes indicating whether the user is Unavailable (code = 0), Away/Busy (code = 1), or Available (code = 2).](image_url)

III. System Design and Implementation

To dynamically manage cloud resources utilized by virtual machines in private clouds, we developed two generic components: VLBCoordinator (Virtual Load Balancing Coordinator) and VMProfiler (Virtual Machine Profiler). VLBCoordinator interacts with a EUCALYPTUS cloud using Typica [8]. Typica is a simple API written in Java to access a variety of Amazon Web services such as EC2, SimpleDB, and DevPay. The core functions of VLBCoordinator are instantiateVirtualMachine and getVMIP, which are accessible through XML-RPC. VMProfiler is used to log the CPU utilization of each virtual machine. It exposes XML-RPC functions to obtain the CPU utilization of a specific virtual machine for the last n seconds. These components can be used with any cloud-based application running on EUCALYPTUS.

The BuddyMonitor back end is responsible for servicing user monitor requests and coordinates with the user-facing front end using a shared database. It obtains CPU usage statistics from VMProfiler and uses VLBCoordinator to spawn new virtual machines when necessary. Fig. 2 shows deployment and the interaction between the components in our system.

![Fig. 2. Component deployment and interaction in our prototype.](image_url)

IV. Experiments

In this section we describe the setup for an experimental evaluation of our prototype back-end mashup application running on a testbed cloud and interacting with a simulated large-scale XMPP service.

A. Testbed cloud

We built a small private heterogeneous compute cloud using five physical machines and EUCALYPTUS [9]. The cloud consists one Cloud Controller (CLC), one Cluster Controller (CC), three Node Controllers (NCs), and one database server.
threshold = 65.0 in the algorithm of Fig. 3.

Emulator Buddy Call U over last Get maximum CPU utilization

threshold?

No

Is

No

Fetch N user requests (R) from database and set I = 0

Yes

Yes

Is

No

Create a new thread for user monitor request R(I) ...

Is

Yes

Call VLBCoordinator to spawn a new VM hosting BuddyMonitor

No

Terminate N/2 threads.

Call VLBCoordinator to spawn a new VM

No

BuddyMonitor

VMProfiler

Node1

VM1

Buddy

Monitor

Buddy Emulator

XMPP Server

LAN

DB Server

Internet

VMProfiler

Node1

VM

Node2

VM

Node3

VM

DBServer

Fig. 3. Flow diagram for the algorithm our BuddyMonitor back end prototype uses to identify the maximum number of concurrent user monitor requests a virtual machine can process and to trigger cloud scaling events.

TABLE I

<table>
<thead>
<tr>
<th>Node</th>
<th>Type</th>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front-end</td>
<td>Intel Pentium IV</td>
<td>2.80 GHz</td>
<td>2 GB</td>
</tr>
<tr>
<td>DBServer</td>
<td>Intel Core 2 Duo</td>
<td>2.6 GHz</td>
<td>2 GB</td>
</tr>
<tr>
<td>Node1</td>
<td>Intel Pentium IV</td>
<td>2.66 GHz</td>
<td>1.5 GB</td>
</tr>
<tr>
<td>Node2</td>
<td>Intel Celeron</td>
<td>2.4 GHz</td>
<td>2 GB</td>
</tr>
<tr>
<td>Node3</td>
<td>Intel Pentium IV</td>
<td>2.66 GHz</td>
<td>0.5 GB</td>
</tr>
</tbody>
</table>

We installed the CLC and CC on a front-end node attached to both our main LAN and the cloud’s private network. We installed the NCs on three separate machines (Node1, Node2, and Node3) connected to the private network. We installed a MySQL server on a physical machine named DBServer. Table I shows the hardware configuration of the machines, and Fig. 5 shows the deployment of software components and network connections to the machines.

B. BuddyEmulator and workload generation

The BuddyMonitor back-end gets user monitor requests from a database shared with the user-facing front end and responds to those requests by opening monitoring connections to an XMPP server. To simulate the effects of user base growth on BuddyMonitor, we needed access to an XMPP server with a large number of potential buddies whose status changes over time, and we needed to simulate different user monitor request workloads over time.

For the XMPP server, we installed ejabberd [10], an open-source XMPP server providing basic IM services. We added 2,000 users and associated 50 roster items (buddies) with each user. To keep the simulation simple, we used the same 50 roster items for each user.

We then developed BuddyEmulator, a standalone Java program using the Smack API [7] that connects to the ejabberd server and simulates changes to the presence and availability status of the 50 buddies over time. It does so by connecting each of the 50 shared buddies (associated with all 2,000 users) to ejabbered and continuously changing the presence and availability status of each buddy randomly for a random duration between 0 to 60 seconds.

Finally, we simulated a large number of users making user monitor requests using a SQL script that executes directly on the BuddyMonitor database to insert the requests. We fixed the duration of each user monitor request to 30 minutes. The green line in Fig. 8 shows the number of active user monitor requests at each point of time in the simulation.

We performed two experiments based on this simulation of a real-world IM service, as described below.

C. Experiment 1: Static allocation

In this experiment, we established the experimental setup shown in Fig. 4 and profiled the system’s behavior against BuddyEmulator. We statically allocated only one virtual machine (VM1) to the BuddyMonitor back end. Here VMProfiler is only used to obtain the CPU usage of VM1 during the experiment.

D. Experiment 2: Adaptive allocation

In this experiment, we ported BuddyMonitor to our private testbed cloud and implemented the algorithm described in Fig. 3 to leverage the benefits of cloud computing and adaptive resource provisioning for this back-end mashup application. Fig. 5 shows the experimental setup we established. Initially, only one virtual machine (VM1) is alive and processing user monitor requests, while VM2 and VM3 are cached by EUCASTLYPTUS. As previously described, we used VMProfiler to monitor the CPU usage of each virtual machine, and we used VLBCoordinator to adaptively invoke additional virtual machines as required by the system. We used N = 10 and CPU_threshold = 65.0 in the algorithm of Fig. 3.

V. RESULTS

A. Experiment 1: Static allocation

This section describes the results we obtained in Experiment 1. Fig. 6 shows the number of concurrent user monitors...
requests being serviced at each point in time. After successfully reaching 490 concurrent monitors, the system stopped responding and stopped servicing user monitor requests completely.

Fig. 7 shows the CPU utilization of VM1 during the experiment. When the resource utilization of the system reaches some critical point, the system starts thrashing and the measured CPU utilization rapidly increases to over 90%. We halted the simulation after 20 minutes after finding that no user monitor requests were being handled at all.

With static resource allocation, even with a relatively small user base, to prevent catastrophic failure of the system, we would have to either limit the number of user requests allowed (not satisfy the needs of our users) or provision additional resources a priori (underutilizing those resources).

Since neither of these static solutions are suitable, in Experiment 2, we explore the use of adaptive allocation of cloud resources.

**B. Experiment 2: Adaptive allocation**

This section describes the results of Experiment 2. Fig. 8 shows the number of active user monitors during the experiment and compares it with the number of outstanding user monitor requests at each point in time. Initially, the system was configured with VM1 only. As the number of user monitors running on VM1 grows, the BuddyMonitor component on VM1 dynamically identifies the maximum number of concurrent monitor requests VM1 can handle and adaptively spawns VM2 to serve more user monitor requests. Whenever VM2 dynamically identifies the maximum number of concurrent user monitors it can handle, it then spawns VM3 to accommodate more requests. We observe linear growth in the number of active user monitors except during the time required to boot new VMs.

Fig. 9 shows the CPU utilization of the virtual machines during the experiment, and Fig. 10 shows the cumulative number of user presences logged by BuddyMonitor during the experiment. In contrast to the static allocation policy used in Experiment 1, the system is able to handle the rapid growth in user monitor requests without bringing any of the VMs to an unstable state.

One limitation of our current prototype is that it never
services all of the active user monitor requests because we terminate 2N monitors each time a VM reaches the saturation point. We found that this was necessary to ensure stable behavior of our BuddyMonitor VMs. We plan to fix this issue in the future, but since our current goal is merely to explore adaptive VM allocation, the limitation is not important.

Another limitation of our current prototype, evident from the bottom graph of Fig. 8, is that our current prototype is not capable of scaling down as the number of active user monitors decreases. Although this limitation is also unimportant for the current experiments, it is an interesting issue, since scaling down in this case would require migrating the monitors running on one underutilized VM to another underutilized VM. We plan to further explore this issue in future work.

VI. CONCLUSION

In this paper, we have described a working prototype back-end mashup application system that achieves scalability, stability, and good performance through adaptive dynamic resource allocation on a heterogeneous private cloud. Our experimental results, based on a realistic simulation of a large scale external service, demonstrate the feasibility of our approach.

Besides addressing the already-discussed limitations of the BuddyMonitor application, in future work we plan to explore techniques to reduce the effect of virtual machine boot up time by predicting future workload and allocating resources more proactively based on those predictions, and to further understand the performance requirements of back-end mashups through further case studies.

REFERENCES