Extending a workload injector for testing web service applications on Amazon EC2

Cloud Computing from a Software Engineering Perspective Project

Riccardo Desantis
Politecnico di Milano
MSc Student
riccardo.desantis@mail.polimi.it

Davide Molinari
Politecnico di Milano
MSc Student
davide2.molinari@mail.polimi.it

Ettore Trevisiol
Politecnico di Milano
MSc Student
ettore.trevisiol@mail.polimi.it

ABSTRACT

Cloud computing, a relatively recent term, builds on decades of research in virtualization, distributed computing, utility computing, and more recently networking, web and software services. It implies a service oriented architecture, reduced information technology overhead for the end-user, great flexibility, reduced total cost of ownership, on-demand services and many other **advantages**.

Our work is an attempt to analyse the current technologies of testing tools for traditional IT infrastructures and see how they work in a cloud computing environment. In particularly

In the first two chapters we present the current state of the art of the cloud computing, then in the third we summarize how a cloud benchmark should be and with which metrics we should evaluate it. The forth and the fifth chapters list respectively the cloud services and the tools we used in this work **to deploy the SPECweb workload in the cloud. Finally the sixth, seventh and eighth part explain how we extend, implement, interface and shape the load injector we choose to test the cloud.**

Categories and Subject Descriptors

A.1 [**General Literature**]: Introduction and Survey

General Terms

cloud, benchmark, Amazon, AWS, EC2, AMI, instance, EBS, SPECweb2005, web server, BeSim, Apache JMeter, CLIF, The Grinder, HTTP-QAT, test automation, API

Keywords

cloud computing, benchmarks, web services, auto scaling, load testing, performance metrics

1. INTRODUCTION

Cloud Computing **is frequently considered** a term that simply renames common technologies and techniques that **we have already seen** in IT. It may be interpreted to mean *data center hosting* and then subsequently dismissed without catching the improvements to hosting called utility computing that permit near realtime, policy-based control of computing resources. Or it may be interpreted to mean only data center hosting rather than understood to be the significant shift in Internet application architecture that it is.

Cloud computing represents a different way to architect and remotely manage computing resources. One has only to establish an account with Microsoft [13], Amazon [6], Google [8], IBM [9], or **some other provider** to begin building and deploying application systems into a cloud. These systems can be, but certainly are not restricted to being, **simple**. They can be web applications that require only http services. They might require a relational database. They might require web service infrastructure and message queues. There might be need to interoperate with CRM or e-commerce application services, **requiring** construction of a custom technology stack to deploy into the cloud if these services are not already provided there. They might require the use of new types of persistent storage, that might never have to be replicated because the new storage technologies build in required reliability. They might require the remote hosting and use of custom or 3rd party software systems. And they might require the capability to programmatically increase or decrease computing resources, as a function of business intelligence about resource demand using virtualization.

While not all of these capabilities exist in today's clouds, nor are all that do exist fully automated, a good portion of them can be provisioned. Metrics and tools to test all this new technologies are yet to be fully developed and implemented, so our work is an attempt to extend a workload injector to test the Amazon EC2 [4] service.

Politecnico di Milano 2012 Milano, Italy

2. CLOUD COMPUTING

Cloud computing [18][30][24][34][32] allows users to access **a wide set of** services from the Internet ("in the cloud") without owning the technology infrastructure that supports them. The range of services today varies from basic infrastructure services, such as providing storage space, to rather specialized services for payment, identity authentication and others. This section reiterates the advantages of using cloud services and provides an overview of the services offered today.

2.1 Why Cloud Computing?

Generally, three different types of cloud services are differentiated. At the core, *Infrastructure as a Service* (IaaS) **solutions** provision resources such as servers (often in form of virtual machines), network bandwidth, storage, and related tools necessary to build an application environment from scratch (see Figure 1).

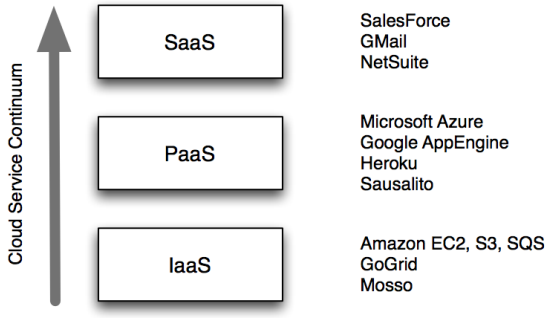

Figure 1: Cloud Services

The business model is pay-per-use. Thus the users do not need to make an investment upfront and pay for the hardware resources they consume. Furthermore, most providers guarantee virtually infinite resources. Hence, resources are provided on demand and there is no need to plan far ahead for provisioning. Although infrastructure services provide the highest flexibility, developers still have to deal with low level details such as maintaining virtual machines or load-balancing.

Platform as a Service (PaaS) often build on top of IaaS and provide a higher-level environment, a platform, on which developers write customized applications. Those platforms are managed by the service provider. The maintenance, load-balancing and scale-out of the platform are done by the service provider and the developer can concentrate on the main functionalities of his application. In exchange for the built-in scalability the developer **accepts** some restrictions on the type of software he can write. Again the business model is pay-per-use and the platform is virtually infinitely scalable.

Finally, *Software as a Service* (SaaS) refers to special-purpose software made available through the Internet. Well-known examples of SaaS are Google Apps, Salesforce[31] or DeskAway. However, those services are not suited for building individual applications and are excluded in the further discussion. The success of cloud computing for all service classes is based on economy of scale. The cloud provider can offer services to millions of users at a lower price than users accommodating these services themselves. In addition to the price, the quality of the service is a major incentive for using cloud services. In particular, the cloud provider is responsible for guaranteeing high availability and reliability. Users expect almost 100% availability and (virtually) constant response times independent of the number of concurrent users. Furthermore, users of cloud services do not need to worry about scalability because the offer is virtually infinite. The IT cost grows linearly with the business rather than step-wise as is the case for traditional computing, in which businesses need to buy hardware in the granularity of machines (rather than CPU cycles). In summary, the goal of cloud computing is to provide more for less.

2.2 Cloud Services Today

By today, a huge variety of services exists offering different infrastructure or platform services.

IaaS

The most flexible form of IaaS are server-hosting services like Amazon's Elastic Computing Cloud (EC2) or GoGRID. These services allow to rent virtual machines (VM) running the different distributions of Unix/Linux or Windows. Those virtual machines can be freely configured and enable to run almost every kind of software. Although the general concept of renting a virtual machine is similar for all providers, services differ in the way they are priced, how they persist data over failures, SLA, geographical location, and the **tool sets** they provide. Most critical for developing an application is how persistent data is stored and handled. Generally the virtual machines loose their data if they **are** shut down (e.g. by the user or by a hardware failure).

In the case of GoGRID, the developer himself has to take care of persistence. To help the developer, GoGRID offers a persistent (single data center replicated) network attached storage for backups.

Amazon on the other hand offers two additional services to store data persistently: Amazon's Simple Storage Service (S3)[5] and the Elastic Block Store (EBS)[3]. S3 can be used by several clients simultaneously, is highly reliable due to distributing the data across data centers, provides eventual consistency guarantees and is optimized for reads of **large** files. On the other hand, the Elastic Block Store is a special storage service which can only be accessed by a single virtual machine at a time, is optimized for writes, but only replicated in one data center. Using one or the other storage service results in completely different guarantees and limitations. With S3, the data is always readable - even in the case of network partitioning or data center failure, there is no restriction in the scale-out. However, the data itself is just eventual consistent. With EBS the data is highly consistent, but at the same time can only be accessed by **a single** VM. Thus, the scalability is limited.

The different solutions vary in the form of consistency guarantees they provide, the ability to replicate across data centers, performance, index support, ... There are also a variety of additional IaaS offerings, such as queue services or content delivery, which can help to build distributed, highly available applications. **The complete list** is out of scope of this paper.

PaaS

For PaaS, the most prominent examples are Google's App Engine and Microsoft's Azure platform.

Google's App Engine hosts python, **Java and Go** programs in a highly scalable manner. Similar to IaaS, data persistence is one of the critical differentiators between the different platforms, **which provide also different implementations of so called NoSQL systems.** Google provides a datastore API allowing to permanently persist data. With this API, the developer can take advantage of the notion of transactions and even a simple query language. However, data is required to be grouped into so-called entity groups. Transactions are only possible inside such entity group. Although there is no restrictions on the size of an en-

tity group, the scalability is limited as per entity group only one transaction can operate at a time. That is, all transactions are serialized.

The Azure Service platform is Microsoft's PaaS offering. Instead of Python, Azure is based on the .net language. Similar to Google's App Engine it has a dedicated API to store and retrieve data called SQL Services. The underlying system for these SQL Services is Microsoft SQL Server. Although not all functionalities of Microsoft SQL Server are exposed via the API, the user can run transactions and use a restricted SQL query language. Similar to Google's App Engine, the user has to partition the data manually into so-called containers. Transactions and queries are restricted to one container at a time. In contrast to Google's App Engine, inside one container several transactions can run simultaneously, but containers are restricted with regard to their size (i.e., 2GB).

There also exists a wide variety of startups offering different platforms for different languages. Again, next to the language and offered libraries, all the providers mainly vary in how they handle persistent data.

3. BENCHMARKING THE CLOUD

It was inevitable that with all the hype and marketing dollars directed at cloud computing **in** these days, someone would eventually start trying to use them for real work. Of course, this puts a nasty wrinkle into marketing plans because once people starting using them for real work, then there are actual performance results. The results themselves are too troubling because they are usually point cases, and negative messages are easily explained away by calling on the problems of a particular software stack. But then the results lead the engineering-minded to wonder whether all of the available cloud computing alternatives behave in the same way, and if not which of the them might be best suited for a particular task [19][29][15][28][23][20][16].

3.1 Requirements to a Cloud Benchmark

As indicated above, today's cloud services differ among others by cost, performance, consistency guarantees, load-balancing, caching, fault tolerance, SLA and programming language. System architects and developers are confronted with this variety of services and trade-offs. Hence, the purpose of a cloud benchmark should be to help the developer when choosing the right architecture and services for their applications [21].

Features and Metrics

Arguable, the main advantages of cloud computing are scalability, pay-per-use and fault tolerance. Despite the promises of cloud providers those features are often differently fulfilled. For example, most cloud providers claim to provide nearly infinite scalability for their services, but it is not self-evident that the combination or some of the limitations of one or more services does not yield a scalability limit (e.g., the limitations on the persistent storage).

Furthermore, price plans and the granularity of the pricing lead to different overall costs. For example, for PaaS pricing is typically based on CPU utilization. A web application which has to support a single transaction per hour, is priced exactly for this single transaction. In contrast, for IaaS virtual machine instances are often priced on a per hour basis. The costs for a VM instance, however, are the same regardless if one or 1000 transactions are performed per hour. In economics, this is often referred to as lot-size problem. The interested reader might notice, that it is possible to see the classical architecture with self-owned hardware as an extreme case having really high lot-sizes.

Finally, fault-tolerance differs significantly between providers. The number of faulty services the system can resist without user notice, or single data-center vs. multi data-center replication are just some examples. As mentioned before, the traditional benchmarks are mainly concerned with performance and cost of static systems. Those metrics still have relevance for the cloud applications but we need different ways for measuring them for scalable (i.e., dynamic) systems where resources come and go. Moreover, a benchmark for the cloud should additionally test the cloud-specific features (scalability, pay-per-use and fault-tolerance) and provide appropriate metrics for them [17][33].

4. AMAZON WEB SERVICES

Since 2006, Amazon Web Services (AWS) provides an infrastructure web services platform in the cloud. AWS allows access to a suite of elastic IT infrastructure services. Customers pay only for what they use and, with AWS, they can take advantage of Amazon.coms global computing infrastructure that is the backbone of Amazon.com's retail business and transactional enterprise. AWS main features are:

- Cost-effective. Pay only for what you use.
- Dependable. The Amazon Web Services cloud is distributed, secure and resilient.
- Flexible. You control the resources you consume.
- Comprehensive. Amazon Web Services gives you a number of services you can incorporate into your applications, from databases to payments.

In the following sections we describe Elastic Cloud Computing, Cloud Watch, Elastic Load Balancing and Auto Scaling, which are the main products offered by AWS as computing resource or related services.

4.1 Amazon Elastic Cloud Computing

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It provides complete control of the computing resources you buy and lets you run on Amazon's proven computing environment. Amazon EC2 allows you to pay only for capacity that you actually use. Amazon EC2 presents a true virtual computing environment, each user can launch instances with the preferred operating system, manage network access permission and run the image on the desired number of systems. The basic tasks to use EC2 are:

- Select a pre-configured image or create a custom Amazon Machine Image (AMI) containing your data.
- Configure security and network access.
- Start, terminate, and monitor as many instances of your AMI as needed.

- Determine whether you want to run in multiple locations, utilize static IP **end-points**, etc.

- Pay only for the resources that you actually consume.

4.2 Amazon Cloud Watch

Amazon CloudWatch[2] provides monitoring for AWS cloud resources. Developers and system administrators can use it to collect and track metrics and react immediately to keep their applications running smoothly. Amazon CloudWatch monitors AWS resources such as Amazon EC2 and Amazon RDS DB instances, and can also monitor custom metrics generated by a customer's applications and services. Metrics such as CPU utilization, latency, and request counts are provided automatically for AWS resources. But you can also supply your own custom application and system metrics, such as memory usage, transaction volumes, or error rates, and Amazon CloudWatch will monitor these too. With Amazon CloudWatch, you can set alarms for your metric data and use Auto Scaling to add or remove Amazon EC2 instances dynamically based on your Amazon CloudWatch metrics.

4.3 Amazon Elastic Load Balancing

Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances. Elastic Load Balancing detects unhealthy instances by pooling and automatically reroutes traffic to healthy instances until the unhealthy instances have been restored. Users can enable Elastic Load Balancing within a single Availability Zone or across multiple zones[1]. Elastic Load Balancing supports SSL termination at the Load Balancer, Internet Protocol version 6 (IPv6) and its metrics such as request count and request latency are reported by Amazon CloudWatch.

4.4 Amazon Auto Scaling

Auto Scaling[1] **(AS)** allows to scale Amazon EC2 capacity up or down automatically according to conditions defined by the user. With Auto Scaling, you can ensure that the number of Amazon EC2 instances you are using increases seamlessly during demand spikes to maintain performance, and decreases automatically during demand lulls to minimize costs. Auto Scaling is enabled together with Amazon CloudWatch. AS scales dynamically based on user defined Amazon Cloud- Watch metrics, or predictably according to a defined schedule.

5. TOOLS

In this section we present the software tools analysed for our work. **We evaluated these tools in order to decide which of them had the best features to emulate a SPECweb test.**

5.1 CLIF

CLIF[7] is OW2's project dedicated to load and performance testing, for any kind of system under test.

[1]AWS are geographically distributed in the world in 5 "regions", each region is a set of "availability zones" connected by low latency connections.

Customer Satisfaction and Operational Efficiency Tests are confronted with a double trade-off: testing time vs time-to-market, and testing costs vs return on investment. But, for a given amount of time and money spend, the more you test, the more **can be improved** your applications and services performance and robustness. CLIF's target is to create a test environment with fast response times, less breakdowns and a minimized run-time infrastructure.

It's possible to develop CLIF tests under Eclipse environment, or getting nightly performance reports through the Hudson/Jenkins plug-in.

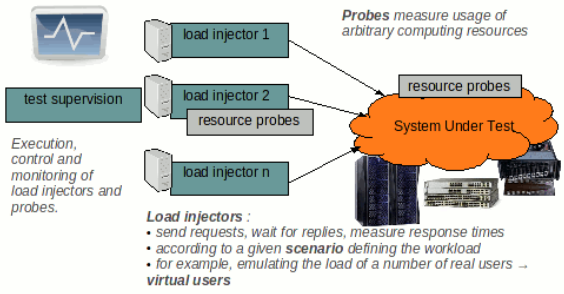

Figure 2: CLIF Framework

Clif comes in bundle with this pre-developed load injectors: **HTTP(S), FTP, TCP, UDP, LDAP and RTP.**

Or you can load any system command line (sh, ksh, csh, tcsh, bash, etc.) or any of your own Java programs. For every injection you can probes this kind of resources: **CPU, memory, JVM, disk, network, any of your own (in Java), JNX and SNMP samples.**

5.2 The Grinder

The Grinder[22] is an open-source, cross-platform testing framework, based on Java and distributed under a BSD style license.

It was originally presented in a J2EE book and then developed by a different group to versions 2 and 3, increasing its relevance among the load testing applications. The Grinder is capable of performing load testing on any application that exposes a Java API, including HTTP web servers, SOAP and REST web services, and application servers (CORBA, RMI, JMS, EJBs), as well as custom protocols.

Starting from version 3, with the introduction of the Jython engine, it allows to define tests as Jython scripts.

The goal of The Grinder is to **"minimize system resource requirements while maximizing the number of test contexts (virtual users)".** This is accomplished by means of a distributed, multi-threaded and multi-process architecture, which represents one of the most remarkable features.

Architecture

The framework is composed **by** three types of processes: worker processes, agent processes and the console. Each of these processes is a Java virtual machine.

- the **worker processes** interpret Jython[27] test scripts and perform tests using a number of worker threads

- the **agent processes** are in charge of manage the workers processes, generally following the directions from the console process

- the **console** represent the centralized control unit of the framework: it coordinates the other processes, collate and display statistics, allows to edit and distribute test scripts to the worker processes. It provides a helpful user interface to manage its functionalities.

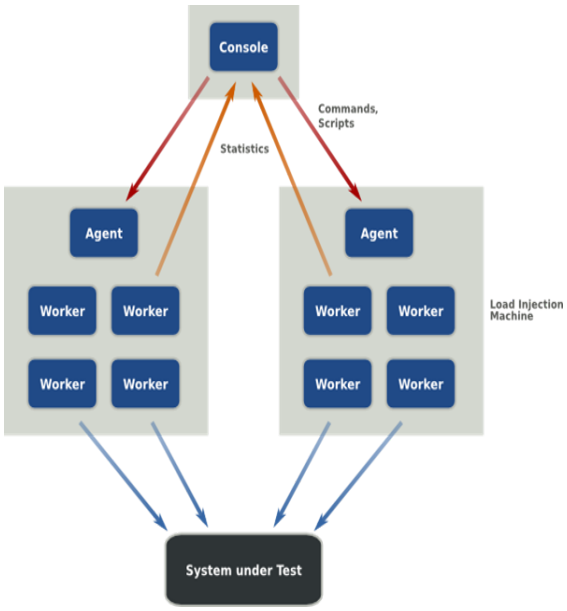

Figure 3: The Grinder processes and their interaction

Each worker process sets up a network connection to the console to report statistics. Each agent process sets up a connection to the console to receive commands, which it passes on to its worker processes. The console listens for both types of connection on a particular address and port.

It is possible to configure and execute tests without the presence of the console (useful options when the only command line control is available).

Another relevant feature (already observed in tools like Selenium IDE) is the possibility to use the *TCPProxy* functionality, which allows to register actual users navigational experiences (then translated into jython scripts) in order to intuitively recreate some testing sequences.

5.3 HTTP-QAT

HTTP-QAT[25] (Quality Assurance Tool) is a testing framework written in Jython and built on top of The Grinder framework (whereof it only involves the testing configuration layer).

It supports both functional and L&P (load and performance) tests and currently it only provides testing capabilities for HTTP and HTTPS services (GET, POST, PUT and DELETE methods).

It allows to easily define sets of test cases through a precise setup of configuration files. Following the guidelines of the documentation it is possible to configure a *testing application* as composed of several *chains* (sequences) of *test cases* (HTTP/HTTPS requests) and assign to each chain an execution probability.

Each request is highly customizable (several aspect such as parameters, headers, authentication can be managed) with apposite tags in the configuration files.

HTTP-QAT helps to exploit the functionalities of The Grinder Framework to set up complex HTTP testing applications without the need to go through Jython scripting.

5.4 SPECweb 2005

SPECweb2005[14] is the Standard Performance Evaluation Corporation (SPEC) benchmark for evaluating the performance of World Wide Web Servers. SPECweb2005 continues the SPEC tradition of giving Web users the most objective and representative benchmark for measuring a system ability to act as a web server. The SPECweb2005 benchmark includes many sophisticated features like:

- Measures simultaneous user sessions
- Relevant dynamic content: ASPX, JSP, and PHP implementations included
- Page requests through parallel HTTP connections
- Multiple, standardized workloads: Banking (HTTPS), E-commerce (HTTP and HTTPS), and Support (HTTP)
- File accesses closely matching today's real-world web server access patterns

The SPEC defined a set of rules that has to be fulfilled in order to obtain fair results that can be actually published. The SPECweb tool consists of 3 main components:

- Web applications
- Back-end simulator
- Load generator

A set of 3 web applications emulate different type of websites. These applications have to be deployed on the system that will be tested. Real world web applications usually make use of a back-end system in order to reply to user requests. The Back-end simulator (BeSim) is used to supply web applications with data requested by the user. This part of the tool has been introduced with the intent to better emulate the environment on which the tested system is going to work. BeSim has been developed in order to avoid bottleneck on this part of the system and efficiently test the web server. The load generator is used to produce the requests that the web server is going to serve. It is composed of 2 logical components:

- Prime Client
- Client

The Prime Client is a tool that coordinates many clients, that may be running on many machines, in order to produce the desired workload and gather statistics. The Client is a piece of software that can be run many times concurrently on the same machine or on remote systems, the main job of clients is to generate the workload as specified by the prime client.

The system that will actually be tested is the one on which the web server runs. In order to test entirely the system SPECweb provides 3 web applications called: banking, ecommerce and support

- **Banking** application **emulates** a web site of an online bank which let the user log into his account, transfer founds and make payments. The main characteristic of this application is that most of the connections are SSL based so the web server has to decode messages before processing them. This task generates heavy load on the CPU of the system.
- **E-commerce** application is used to emulate a big variety of web applications, it consists of an ecommerce web site that let the user search, customize and buy products. The requests to this applications are a mix of http and https. Ecommerce application is not designed to stress a single component of the system.
- **Support** application is used to stress the network interface of the system. It emulate a website that hosts files of different size (to a maximum of 40MB).

Clients generate random pattern of requests in order to emulate different users according to a Markov chain model which probabilities has been calculated using logs of real websites.

5.5 JMeter

Apache JMeter[26] is open source and cross platform software, a 100% pure Java application designed to load test functional behaviour and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions. It is part of the Apache Jakarta Project. It can be used to simulate a heavy load on a server to test its strength or to analyse overall performance under different load types. **JMeter can be used** to make a graphical analysis of performance or to test your server behaviour under heavy concurrent load. Apache JMeter can load and test the performance of many different server types: Web - HTTP, HTTPS, SOAP, Database via JDBC, LDAP, JMS, Mail - POP3(S) and IMAP(S). Apache JMeter main features are: full multithreaded framework, careful GUI design allows faster operation and more precise timings, caching and offline analysis/replaying of test results, highly extensible. JMeter is not a browser. As far as webservices and remote services are concerned, JMeter looks like a browser (or rather, multiple browsers); however JMeter does not perform all the actions supported by browsers. In particular, JMeter does not execute the Javascript found in HTML pages. Nor does it render the HTML pages as a browser does.

JMeter Tests

JMeter tests are specified in .jmx files, each one representing a Test Plan. The Test Plan object has a checkbox called "Functional Testing". If selected, it will cause JMeter to record the data returned from the server for each sample (it is off by default).

The main elements of a test plan are:

- A ThreadGroup, the **starting** point of any test plan. It sets: the number of threads, the ramp-up period, the number of times to execute the test.
- Samplers, tell JMeter to send requests to a server and wait for a response. They are processed in the order they appear in the tree. JMeter samplers include: FTP Request, HTTP Request, JDBC Request, Java object request, LDAP Request, SOAP/XML-RPC Request, WebService (SOAP) Request.
- Logic Controllers, let you customize the logic that JMeter uses to decide when to send requests. Logic Controllers can change the order of requests coming from their child elements.
- Listeners, provide access to the information JMeter gathers about the test cases while JMeter runs. The Graph Results listener plots the response times on a graph.
- Timers, by default, a JMeter thread sends requests without pausing between each request. The timer will cause JMeter to delay a certain amount of time before each sampler which is in its scope.

We decided to utilize JMeter for our work because of its usability and flexibility to create our own test plan which emulate workload and user behaviour of SPECweb test.

6. TESTS

6.1 SPECweb deployment

The main problem of using SPECweb to test the cloud is that this tool has been developed to target a different kind of platforms, so it requires some initializations steps that could not be easily performed in a dynamic environment such as the cloud. In order to understand what are the main difficulty of using this tool, one should first look at how it works, as shown in figure 4.

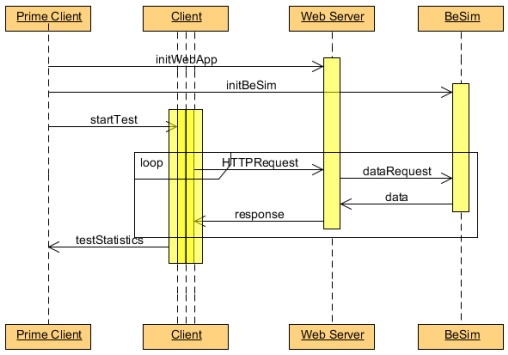

Figure 4: SPECweb 2005 Workflow

Using autoscaling feature, offered by the cloud, the number of tested web servers may change during the benchmark. This dynamic aspect had not been taken into account in developing SPECweb, in fact at the begin of the test the prime client initializes both the BeSim and the web server as can be seen in **Figure 4.** When the autoscaling engine spawns a new web server, the prime client is not capable of performing the initialization on the new machine. In order to make the new server responsive to requests of clients, it's necessary to automatically perform the initialization after boot. To correctly initialize a web server one should know how to reach the BeSim (IP address and port number). In SPECweb 2005 these parameters are written in the configuration file of the

prime client, who sends the information to the web server: this approach frees the user from the necessity to use a static IP for the BeSim. In a dynamic environment, in which each server has to auto initialize the web application immediately after boot, the IP address of the BeSim has to be known a priori, so a static IP is needed.

Another delicate point of an architecture with multiple web servers is the load balancer. This module is widely used in order to wisely distribute incoming requests among different web servers. There are two kind of problems associated with this part of the system. The first is about sessions: many web applications, like the banking one, use https persistent connections in order to reduce the overhead of opening a new connection for each request. A load balancer that operates at TCP level cannot distinguish between requests that use an already opened connection or a new one and usually fails to deliver the request to the right web server. The second problem with load balancing is about performance. One may think of solving the session problem of load balancing by using the balancer itself as SSL endpoint: this solution is feasible but if the number of SSL connections is very high, like in the banking application, some of the intense computation is moved from the web server to the balancer.

This situation is not ideal for two reasons: the results of the test would be inaccurate, because all of the computation it's supposed to be performed by the web server, and the load balancer could become the bottleneck of the system since it may not benefit of the autoscaling. Since both solutions (TCP layer load balancer or SSL termination load balancer) has pros and cons, we choose to try both of them. The final configuration that has been deployed on the cloud is shown in **Figure 5: the Prime Client acts as a coordinator of the test and it creates some Clients which send HTTP(S) requests to Web Servers. The incoming workload is distributed by the load balancer across several AMIs of the Autoscaling Group and, at the same time, Web Servers send requests to the BackEnd Simulator. Performances of Web Servers' AMIs are under control of CloudWatch, which collects measurements which help AutoScaling service to implements the policy configured: according to the level of performance, AS may decide to increase or decrease resources of the Web Server.**

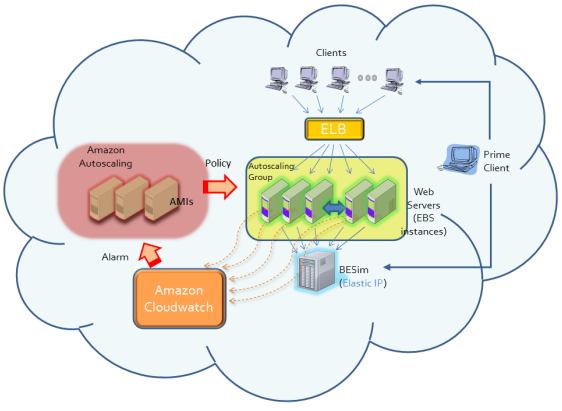

Figure 5: SPECweb 2005 Deployment

6.2 SPECweb tests

Behaviour

SPECweb load injectors work in a closed model environment. Simultaneous sessions are started with a fixed number of users and continuously send dynamic pages with a 10 seconds interval (THINK_TIME) between them, then they access to a new page, or leave the system.

Every test starts with his own configuration file, with fixed parameters, and stable behaviour during the workloads execution time. The diagram in **Figure** 6 reflects the different iteration/phases for the Banking, Ecommerce, and Support workloads.

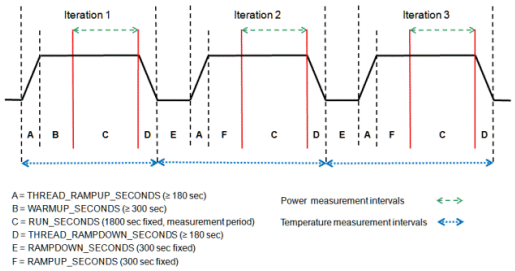

Figure 6: SPECweb 2005 Test Diagram

- **Phase A**: Ramp-up period is the time period across which the load generating threads are started. This phase is designed to ramp up user activity rather than beginning the benchmark run with an immediate and full-load spike in requests and sends at least one request per user thread.

- **Phase B**: The warm-up period is intended to be a time during which the server can **initialize** its cache prior to the actual measurement interval. At the end of the warm-up period, all results are cleared from the load generator, and recording starts a new one. Accordingly, any errors reported prior to the beginning of the run period will not be reflected in the final results for this benchmark run.

- **Phase C**: The run period is the interval during which benchmark results are recorded. The results of all HTTP requests sent and responses received during this interval will be recorded in the final benchmark results. During this phase the system power data is collected for later retrieval into the results files by the harness.

- **Phase D**: The thread ramp-down period is simply the inverse of A. It is the period during which all load-generating threads are stopped. Although load generating threads are still making requests to the server during this interval, all recording of results will have stopped at the end of the run period.

- **Phase E**: The ramp-down period is the time given to the client and server to return to their "unloaded" state. This is primarily intended to assure sufficient time for TCP connection clean-up before the start of the next test iteration.

- **Phase F**: The ramp-up period replaces the warm-up period, B, for the second and third benchmark run iterations. It is presumed at this point that the server's cache is already primed, so it requires a shorter period of time between the thread ramp-up period and the run period for these subsequent iterations in order to reach a steady-state condition.

Markov Chain

A Markov Chain[11] (Figure 7) is a random process characterized as memoryless: the next state depends only on the current state and not on the sequence of events that preceded it. This specific kind of "memorylessness" is called the Markov property.

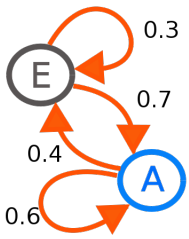

Figure 7: A simple two-state Markov chain

SPECweb adopts **this models of property for specifying** the path of the user in the system emulated. As illustrated in **Figure** 8, the execution flows through the chain according to different values of probability, obtained from the analysis of users behaviour in real systems. SPECweb implements the chain's edges with HTTP requests (GET, PUT, POST, ...) to the web server and each state is a single web page.

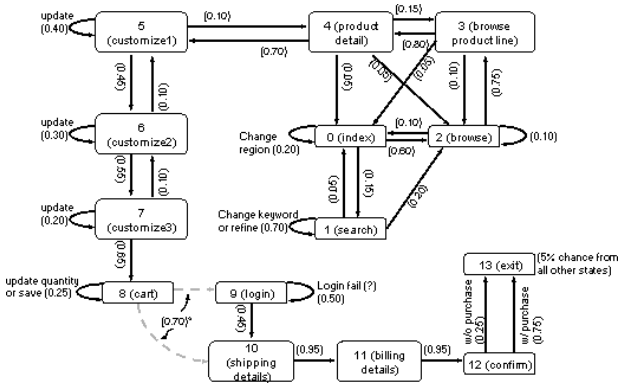

Figure 8: E-commerce test Markov chain

6.3 JMeter Extension

Initialization of the test environment

The first step we had to **implement** is an analysis of the startup phase of SPECweb. This phase includes the initialization of the web server, of the backend simulator and their interfacing. In order to do this we used the **.fcgi files of SPECweb to initialize and prepare instances** to the new JMeter test environment.

Behaviour adaptation

One important feature of SPECweb tests is the preconfigured number of SIMULTANEOUS_SESSIONS in the configuration file, that it **cannot** be dynamically changed at execution time. One of the goal of our work is to play on JMeter's features, in order to have a test with a varying number of simulated users. **To do this we adopted the tool of the project JMeter Plugins[10], that let us to defined a variable number of users for a single execution of a test.**

Markov chain translation

SPECweb tests is based on a Markov chain process. In our work we translate this type of test design, describing the more likely paths of a test execution and designing the correspondent configuration file of a thread group in JMeter application. At runtime, a particular path will be chosen according to its probability in the Markov chain of its workload in SPECweb.

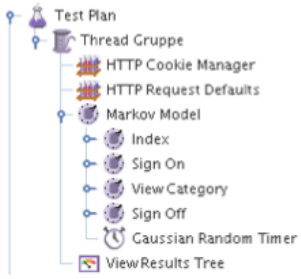

Figure 9: Markov chain example in JMeter

The original JMeter provided by Apache does not include any feature to implement a Markov chain, so we adopted a plug-in developed by the University of Kiel, called Markov 4 Jmeter[12], that allowed us to define the states of SPECweb's chain within the JMeter client (figure 9) and the behaviours with simple .csv files containing the transitions probabilities as in listing 1.

Listing 1: Sample file of behaviours

```
,"Index","Sign On","View","Sign Off","$"
"Index*",0,0.2,0.8,0,0
"Sign On",0.2,0,0.6,0.2,0
"View",0,0.2,0.6,0.2,1
"Sign Off",0,0,0,0,1
```

7. TEST AUTOMATION

We developed a tool in order to automatize and to make simpler the deployment of our injection tests on Amazon EC2 service, **and it's able to manage the architecture we create, as shown in Figure 10.**

Through the Amazon's API, the tool is able to start and configure the three instances we need to **run** the benchmark. As in **Figure 11**, the Prime Client launches one instance that hosts the JMeter Client, with proper configuration settings, and another two instances configured respectively as SPECweb web server and BeSim.

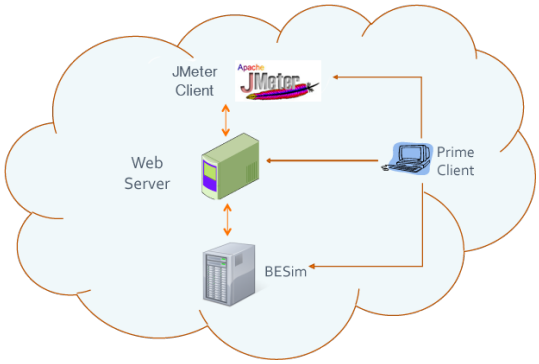

Figure 10: System test architecture

In addition to the ordinary configuration files of JMeter, our tool is able to manage two other file: one is for the simulation of the SPECweb Markov chain in the new client; the other one let to handle the variation of the simultaneous thread during the whole execution.

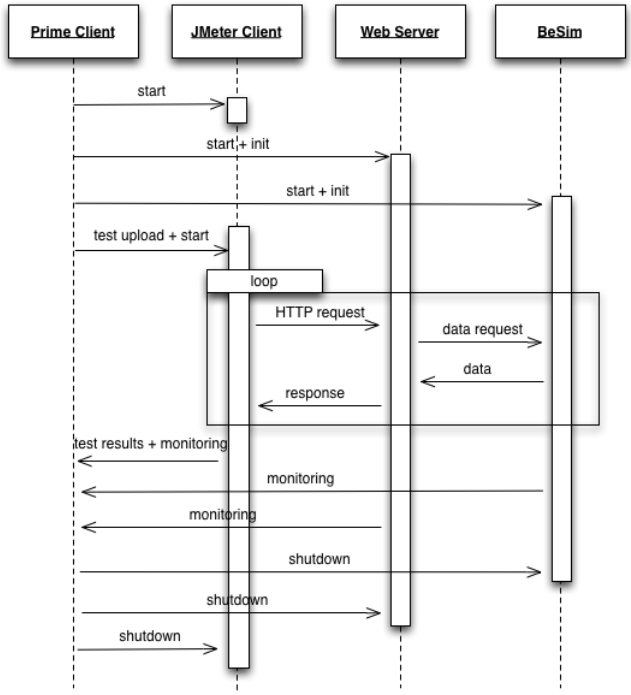

Figure 11: Test Automation Sequence Diagram

At the test completion, the Prime Client downloads and parses results from Amazon instances and stores them in a proper folder. Then, if there isn't more tests to run, it shutdowns all the instances used.

8. SPECMETER [NEW]

In order to run SPECweb workloads using JMeter client in a Amazon EC2 environment we developed a collection of Python scripts.

8.1 Libraries

Our tool uses a set of libraries to interface the host with Amazon and the servers we previously deployed in the IaaS. The first one is BOTO, a library able to use the API provided by Amazon. We used it start or shutdown the AMIs and to enable the CloudWatch service on them.

To manage the SSH connections with our AMIs we used the Paramiko (aka python-ssh) and the Crypto libraries. With these we are able to configure the servers, upload and download files and launch the tests.

8.2 Configuration files

Our program works with two configuration files. The first one is `ami.cfg` and you can use it to configure your AMIs so the tool will be able to manage and work with them.

Listing 2: ami.cfg

```
[client]
ami = ami-clientid
region = us-east-1
zone = us-east-1d
size = m1.large
keypair = your-keypair
user = pippo
password = psw
rootpassword = rootpsw

[webserver]
ami = ami-wsid
region = us-east-1
zone = us-east-1d
size = m1.large
keypair = your-keypair
user = pippo
password = psw
rootpassword = rootpsw

[besim]
ami = ami-8121c8e8
region = us-east-1
zone = us-east-1d
size = m1.large
keypair = your-keypair
user = pippo
password = psw
rootpassword = rootpsw
```

The other one is called `infrastructure.cfg`. With this file you can configure the test files you want to run, the configuration files and startup scripts of SPECweb webserver and BeSim and finally how and what logs you want to save.

Listing 3: infrastructure.cfg

```
[Credentials]
aws_access_key_id = your_access_key_id
aws_secret_access_key = your_secret_key

[webserver]
URI = /bank

START_SCRIPT =
```

```
sh /opt/apache-tomcat/bin/shutdown.sh
rm /opt/apache-tomcat/logs/*
export CATALINA_OPTS="-Xms512m -Xmx4096m"
sh /opt/apache-tomcat/bin/startup.sh
echo BESIM_POOL_SIZE = 100000 >
/opt/apache-tomcat/webapps/bank/bank.conf
ulimit -n 1000000 -u 1000000

STATISTICS =
mkdir /tmp/sar
sar -B 10 {0} > /tmp/sar/Paging_ws.log
sar -r 10 {0} > /tmp/sar/Memory_ws.log
sar -d -p 10 {0} > /tmp/sar/HD_ws.log
sar -I SUM 10 {0} > /tmp/sar/Interrupts_ws.log
sar -n ALL 10 {0} > /tmp/sar/Network_ws.log
sar -u 10 {0} > /tmp/sar/CPU_ws.log

CLOUDWATCH_METRICS =
CPUUtilization
DiskReadBytes
DiskReadOps
DiskWriteBytes
DiskWriteOps
NetworkIn
NetworkOut

[init_webserver]
BESIM_HOST = {0}
BESIM_PORT = 81
BESIM_URI = /besim/besim_fcgi.fcgi
BESIM_PERSISTENT = false
PADDING_DIR = dynamic_padding/
SMARTY_DIR = null
SMARTY_BANK_DIR = null
SEND_CONTENT_LENGTH = false

[besim]
URI = /besim/besim_fcgi.fcgi

START_SCRIPT =
sh /root/start.sh
ulimit -n 1000000 -u 1000000
mkdir /tmp/sar

STATISTICS =
sar -B 10 {0} > /tmp/sar/Paging_bes.log
sar -r 10 {0} > /tmp/sar/Memory_bes.log
sar -d -p 10 {0} > /tmp/sar/HD_bes.log
sar -I SUM 10 {0} > /tmp/sar/Interrupts_bes.log
sar -n ALL 10 {0} > /tmp/sar/Network_bes.log
sar -u 10 {0} > /tmp/sar/CPU_bes.log

CLOUDWATCH_METRICS =
CPUUtilization
DiskReadBytes
DiskReadOps
DiskWriteBytes
DiskWriteOps
NetworkIn
NetworkOut

[init_besim]
Time = {0}
Min_UID = 1
Max_UID = 50
Load = 1
ChkBase = images/subdir0000
Subdirs = 10

[client]
STATISTICS =
mkdir /tmp/sar
sar -B 10 {0} > /tmp/sar/Paging_cli.log
sar -r 10 {0} > /tmp/sar/Memory_cli.log
sar -d -p 10 {0} > /tmp/sar/HD_cli.log
sar -I SUM 10 {0} > /tmp/sar/Interrupts_cli.log
sar -n ALL 10 {0} > /tmp/sar/Network_cli.log
sar -u 10 {0} > /tmp/sar/CPU_client.log

CLOUDWATCH_METRICS =
CPUUtilization
DiskReadBytes
DiskReadOps
DiskWriteBytes
DiskWriteOps
NetworkIn
NetworkOut

[test]
basejmx = standard.jmx
behavior = behavior.csv
# valid values for protocol: http or https
protocol = https
jmeter_folder = /home/ubuntu/jmeter-2.7
test_folder = /home/ubuntu/jmeter-2.7/bin
```

8.3 Users files

We implement an easy way to configure the simultaneous users of the tests. To create your own users' behaviour you should edit `tests.txt`.

Listing 4: tests.txt

```
10 60
20 100
10 30
5 20
```

The first column represent the simultaneous users, while the second represents how many seconds they stay into the system. For example this tests.txt represent a shape as in Figure 12.

8.4 Execution

After configuring all the files required to do a correct test, you can start the automation tool just running the `test.sh` bash script. If you want to do more than one test, you can start the script with how many tests.txt files you want. The tool will do every test you specified for example you can run:

```
sh test.sh test1.txt test2.txt test3.txt
```

and you will have three logs folders, one for each ran test. The listing 5 is an example of an execution log.

Listing 5: Test run log

Figure 12: Simultaneous users' shape

```
pippo@ubuntu:~/SPECmeter$ sh test.sh
Launching BeSim...
Launching WebServer...
Launching JMeter client...

Webserver initialization...
 * Restarting web server apache2
   ...done.

BeSim initialization...
DONE ResetDate = 20120705,
Time=1452351190,Min_UID=1,Max_UID=50,
Load=1,ChkBase=images/subdir0000,Subdirs=10

Initialization phase finished.

Creating configuration files...
... configuration files created!

Sending files to the client...
... configuration files sent.

Starting the test...

Starting the test @ Thu Jul 05 09:57:09
UTC 2012 (1341482229974)
Waiting for possible shutdown message
on port 4445
Experiment start time (ms):1341482230375
Loading behavior file
/home/ubuntu/jmeter-2.7/bin/behavior.csv
for behavior banking

Experiment stop time (ms):1341482456850
Tidying up ...    @ Thu Jul 05 10:00:56
UTC 2012 (1341482456850)
... end of run

DEBUG   2012-07-05 10:01:04.344 [kg.apc.j]():
Creating jmeter env using JMeter home:
/home/ubuntu/jmeter-2.7

Test ended.
Saving logs files into logs folders...
Done.
End.

Shutting down BeSim...
Shutting down WebServer...
Shutting down JMeter client...
```

8.5 Results and logs

In each logs folder you can retrieve every logs you configured: the JMeter logs you defined in the default .jmx test file, the monitoring log files of the AMIs, included the CloudWatch logs and a useful PNG file that show the real shape of the users during the tests, as in Figure 13.

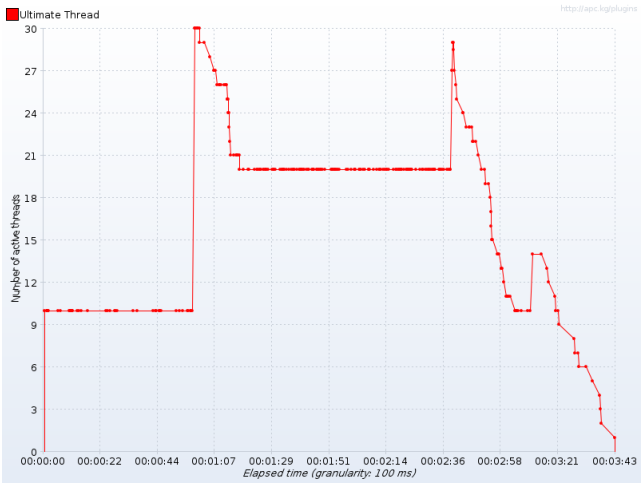

Figure 13: Real simultaneous users

9. CONCLUSIONS

In this work we addressed the problem of benchmarking the cloud. The lack of tools specialized in this kind of benchmark is probably due to the fact that the cloud is still a young and evolving technology. Although there exist many tools useful in generating customized workloads the lack of standard instruments and best practices makes benchmarking of cloud systems a very hard task.

In general, all the tools we used showed that they were designed to test web applications running on a fixed infrastructure through traditional measures of performance (like throughput or the response time). This makes them unsuitable for web applications deployed in a cloud infrastructure where aspects like "scalability" and "pay to use" are of primary importance.

After testing tools that have been presented in section 5 and noticed their lacks and weaknesses, we decided to develop our own tool to exploit cloud potentialities and simulate the typical behaviour of users in a web-based application.

APPENDIX

A. TEST EXECUTION FLOW [NEW]

Running a successful test require this easy steps:

1. Edit the `ami.cfg` with your AMIs details.

2. Edit the `infrastructure.cfg` if you want to change logs, behaviour and jmx files, protocol and initialization scripts.
3. Change the `tests.txt` file as you prefer, or create more than one if you want to run multiple tests.
4. Run the `start.sh` script.
5. Check logs files in the new log folder related to the test.

B. REFERENCES

[1] Amazon auto scaling. http://aws.amazon.com/autoscaling/.
[2] Amazon cloudwatch. http://aws.amazon.com/cloudwatch/.
[3] Amazon elastic block store. http://aws.amazon.com/ebs.
[4] Amazon elastic compute cloud. http://aws.amazon.com/elasticloadbalancing/.
[5] Amazon simple storage service. http://aws.amazon.com/s3/.
[6] Amazon web services. http://aws.amazon.com.
[7] Clif ow2. http://clif.ow2.org/.
[8] Google app engine. https://appengine.google.com/.
[9] Ibm smart cloud. http://www.ibm.com/cloud-computing/us/en/.
[10] Jmeter plugins. http://code.google.com/p/jmeter-plugins/.
[11] Markov chain. http://en.wikipedia.org/wiki/Markov_chain.
[12] Markov4jmeter plug-in. http://se.informatik.uni-kiel.de/markov4jmeter/.
[13] Microsoft azure. http://www.windowsazure.com/.
[14] Specweb 2005. http://www.spec.org/web2005/.
[15] J. Baliga, R. Ayre, K. Hinton, and R. Tucker. Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE*, 99(1):149–167, January 2011.
[16] A. Bei. *Performance Test, Laboratorio dei sistemi software*. Università di Roma Tor Vergata.
[17] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow? *Towards a Benchmark for the Cloud*, June 2009.
[18] A. R. Butt, D. D. Silva, A. Fox, L. A. Barroso, Y. Zhou, and P. Ranganthan. Frontiers of engineering, cloud computing. *The National Academy Press*, pages 3–40, 2010.
[19] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, January 2011.
[20] J. Ekanayake and G. Fox. High performance parallel computing with clouds and cloud technologies. *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, 34(1):20–38, 2010.
[21] G. Gibilisco, F. Marconi, M. Migliarina, J. Panerati, and S. Lombardo. Benchmarks for cloud deployed web applications. 2011.
[22] Grinder. http://grinder.sourceforge.net.
[23] R. Grossman, Y. Gu, M. Sabala, C. Bennet, J. Seidman, and J. Mambratti. The open cloud testbed: Supporting open source cloud computing systems based on large scale high performance, dynamic network services. *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, 25(4):89–97, 2010.
[24] B. Hayes. Cloud computing. *Magazine Communications of the ACM*, 51, July 2008.
[25] HTTP-QAT. http://sourceforge.net/projects/http-qat.
[26] JMeter. http://jmeter.apache.org/.
[27] Jython. http://www.jython.org/.
[28] K. M. Keith R. Jackson, Lavanya Ramakrishnan, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 159–168, 2010.
[29] G. Lee, N. Tolia, P. Ranganathan, and R. H. Katz. Topology-aware resource allocation for data-intensive workloads. *Newsletter ACM SIGCOMM Computer Communication Review*, 41(1), January 2011.
[30] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. *CCGRID '09 Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009.
[31] SalesForce. http://www.salesforce.com/it/.
[32] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: Towards a cloud definition. *Newsletter ACM SIGCOMM Computer Communication Review*, 39(1), January 2009.
[33] J. E. West. Benchmarking your cloud. http://www.hpcwire.com/hpcwire/2009-07-16/benchmarking_your_cloud.html.
[34] L. Youseff, M. Butrico, and D. D. Silva. Toward a unified ontology of cloud computing. *Grid Computing Environments Workshop*, 2008.