AGARIC - a Hybrid Cloud based Testing Platform

Ji Wu Chunhui Wang Yang Liu Li Zhang
School of Computer Science and Engineering
Beihang Univeristy (BUAA)
Beijing, China
wuji@buaa.edu.cn, wangchunhui.ufo@gmail.com, crane.may@gmail.com, lily@buaa.edu.cn

***Abstract*—Cloud computing technology enables developers spend much more time on application quality without considering computing resource constraint, load balancing and performance tuning, etc. It raises challenges along with the benefits it offers to software testing. This paper is motivated with the concerns on how to test the online web applications in a scalable and diverse way. The resources occupied in the proposed cloud testing are diverse in computing ability, network performance, and software configuration (including operating system and browser, etc.). This paper at first identifies the eight unique features to define cloud testing. Then, this paper introduces the design of Agaric - hybrid cloud based testing platform. In Agaric we use both the diverse internet distributed and user owned computing resources, and the centered computing resources. The key to organize the test network is the proposed Test Flow Control Protocol (TFCP). TFCP is verified formally with Colored Petri Net and the platform is evaluated with both controlled and uncontrolled experiments.**

Keywords- software testing; cloud computing; cloud testing

I. INTRODUCTION

Cloud computing is one of the most hot topics both in computer science area and IT industry. Cloud computing is a model for enabling on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, application and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [1]. With the technologies such as virtualization, parallel computing and distributed storage, cloud computing can provide the following essential services:

- On-demand service: computing services such as servers, storage and virtual machines can be acquired automatically when needed without human interaction with the service providers.
- Open network access: services can be accessed over a network using different devices, e.g. servers, PCs, laptops and mobile phones.
- Resource pooling: services and resources are pooled and used through a multitenant approach usually.
- Rapid elasticity: service can be quickly scaled as required by its customers.

There are several service models designed: Software as a service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (Iaas). The resources in the three models are used and organized in a centered cloud in which distributed server clusters are deployed by the same provider usually. In addition to the three ones, another proposal is Human as a Service (HaaS) [2] in which a crowd of people can use the cloud technology from different geographical places and work together to complete a task. HaaS brings diversity of resources into cloud thus can better serve its customers. Testing as a Service (TaaS) now is usually called as a service model [3]. Strictly say, TaaS could be an instance of SaaS if it provides test tooling service, of PaaS or IaaS if it provides machine resources for testing, of HaaS if it provide manual tests. As a mixture of HaaS and SaaS, We propose the hybrid cloud based test here to take use of the diverse distributed user and centered cloud computing resources to provide online testing services.

Cloud computing raises challenges along with the benefits it offers to software testing. Applications in cloud will be maintained and evolve in an online way. The online testing of such application requires particular concerns on load and user experiences. The load of such application could change sharply from hundreds of parallel accesses to a scale of millions. It is very expensive to quickly produce the required large scale of load by using the existing load testing tools. The devices used to access such application may range from server, PC, laptop, to mobile phone with different hardware configuration (CPU and memory), different software configuration (operating system and browser), and different network situation (bandwidth and delay). Therefore, different user usually has different quality experiences. Therefore, the online testing needs to take into account the diversity of devices with different hardware configurations, software configurations and network situations. This is a big challenge and the main motivation of this paper. As a solution, we propose a hybrid cloud based test platform. Hybrid cloud means we use both centered cloud resources and diverse distributed user owned resources to organize the test network.

This paper is structured into six sections. Section 2 proposes the key features of cloud testing that provide guidelines for the design of Agaric. The framework and protocol design and verification is presented in section 3. Section 4 introduces the two experiments conducted for evaluating the effectiveness of Agaric. Related works are introduced in section 5 and the last section concludes this paper.

II. FEATURES OF CLOUD TESTING

Cloud testing denotes to the techniques involved in running a test and usually has two intuitive meanings now, testing in a

This work was supported by the National Basic Research 973 Program of China under Grant No. 2007CB310803.

978-1-4577-1637-9/11/$26.00 ©2011 IEEE

cloud, and testing toward a cloud application. Testing in a cloud usually deploys testing tools and test suites in a cloud like Amazon EC2 to get required test resources. Testing toward a cloud is the testing running test suites against a cloud application. In fact, the features of cloud testing has not been clearly defined yet. Because a cloud application does not show its differences than normal web applications from the view of user, we can treat such application as an online web application. Therefore, all the techniques invented for testing web application are still available for testing a cloud one. In this paper, we identify the cloud testing as a testing technique that cloud resources and technologies are involved to achieve the corresponding testing capabilities. Here we only focus on the black box testing.

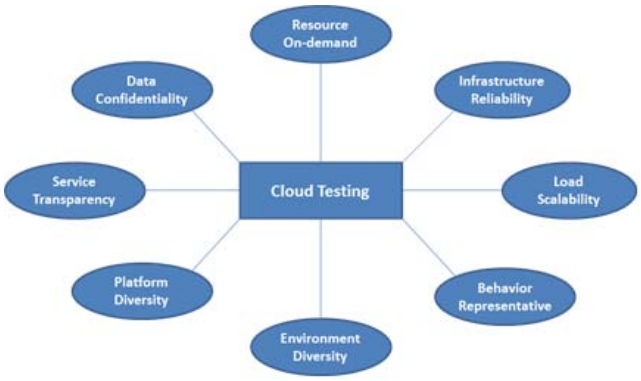

Figure 1 the eight features of cloud testing

Based on the survey, the key features of cloud testing are not clearly identified in the literatures partly because cloud testing is still in its infant phase. Features provide a classification scheme to manage and compare different cloud testing techniques and tools. In the above figure 1, eight unique features of cloud testing are identified.

1) Resource On-demand: the resources used for cloud testing should be On-demanded. That means cloud testing can be started whenever it is needed.

2) Infrastructure Reliability: the infrastructure in which a cloud testing deployed should be highly reliable to satisfy the 7*24 mode of testing.

3) Load Scalability: a cloud testing can produce scalable test load on SUT as required. The performance of test system itself won't decline significantly with the increasing test load.

4) Behavior Representative: the testing behaviors need to be representative of the real users to simulate the real use cases.

5) Environment Diversity: a cloud testing needs to provide the diversity of environment (network, hardware, etc) in which test cases are executed to reflect the varieties of real user client environment.

6) Platform Diversity: a cloud testing needs to provide the diversity of computing platform (OS, browser, etc) in which test cases are executed to reflect the varieties of real user client platform.

7) Service Transparency: services provided by a cloud testing is transparent to the computing resource, hardware and software configuration of testing nodes.

8) Data Confidentiality: data used in a cloud testing must be securely managed and accessed to avoid the leak of the confidential information.

The eight features identified form the scheme to manage and evaluate cloud testing techniques. It does not mean, however, a cloud testing should have all the features. Load Runner in Cloud is the offspring of Load Runner. It deploys load runner on public cloud infrastructure such as Amazon EC2 to produce test load on SUT. Load Runner in Cloud has the features of resource on demand, infrastructure reliability, load scalability, service transparency, and data confidentiality. Because it uses the centered cloud resources, it has no the features of environment diversity and platform diversity. Load runner uses a single script to control a number of virtual users' behavior. Therefore, it cannot generate representative test behaviors.

uTest is another example of cloud testing providing testing services through a community (i.e. resource pool) of professional testers [4]. It is in fact an instance of HaaS. The testers involved in uTest reflect the diversity of geographic locations, test experiences, devices, testing tools used, and so on. Customers of uTest can just specify testing requirements but leaves the testing to those pooled testers. It is obvious that uTest is strong in providing the diversity of environment and platform, and the representative behaviors; but weak in load scalability (now has totally 4000 testers), infrastructure reliability (e.g. human may be sick), and data security assurance because testers can read the data directly from the test script files. There are still some other mentioned cloud testing. According to the features proposed here, we can check which features are provided. Since the size limit, we will not enumerate more ones here.

III. DESIGN OF THE AGARIC

Internet is a huge pool to provide varieties of computing resources with different operating systems, browsers and internet environment. This is the start point to have the idea of developing Agaric.

Agaric does not reject the centered cloud resources. We emphasize much more on acquiring and managing the diverse distributed test resources into test network in this paper. Therefore in the prototype implemented, it does not include test nodes run in centered cloud. As shown in the below figure 2, there are three types of nodes designed in Agaric, test control node, test center node, and test daemon node. Test center node is the core. It is responsible for communicating with all the test daemon nodes distributed over the internet, organizing test environment, and providing services for test control node. Test control node is responsible for deploying test suites, scheduling test tasks, collecting test results and providing services to users. Test daemon node is responsible for receiving test script files from test center node and executing test cases according to the received commands.

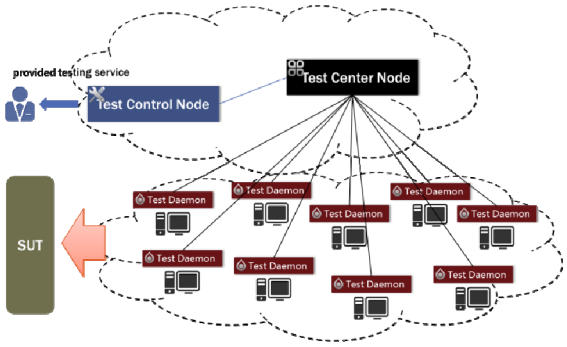

Figure 2 top level structure of Agaric

We deploy the test center node and test control node on Google App Engine to manage the test network and provide services to testers. So readers can find there are two clouds shown in the structure of Agaric. The top one denotes to Google App Engine, and the bottom one refers to the network of diverse distributed test daemon nodes.

There are three layers defined in Agaric, as shown in figure 3. The communication layer works on HTTP. We design a Test Flow Control Protocol (TFCP) to regulate the communications among the three types of nodes. The clock synchronization is designed to synchronize the clocks of test daemon nodes and test control node to make sure the test cases can be scheduled precisely. The message queuing interfaces for test nodes can assure reliable communications.

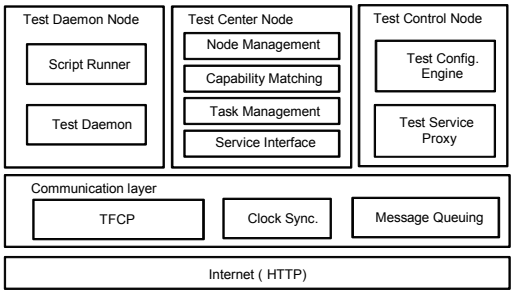

Figure 3 framework design of Agaric

A. *Test Daemon Node*

There are two main components in the daemon node, test daemon and script interpreter. Test daemon is a serving component to communicate with test center node to establish and maintain the test sessions. At first, it will register and tell its hardware and software configuration to the center node. Test center node will accept or reject it according to the reported configuration. If accepted, it will receive and execute test tasks according to the commands from the center node. During test execution, it will update its state periodically to the center node.

Script runner is an interpreter of test case script file. To support flexible test behaviors, we design three types of interpreters for TTCN-3 [9] and Java, python/perl, and Javascript/Actionscript. By using TTCN-3 or Java, tester can get full control of test. Python or perl has strong and efficient built-in data processing and communication facilities that are good for cloud testing. Javascript and Actionscript are part of web standards, usually used for enhancing UI experiences, improving data security and better transaction efficiency. Javascript or Actionscript is particularly useful for implementing test within a browser.

B. *Test Center Node*

Test center node manages test daemon nodes and organizes them into test network. Four main components are designed in the center node. Node management component deals with registration and quit of test daemon nodes, and maintains the status of registered daemon nodes. The assignment of test cases needs to take into account the ability of daemon node to fulfill a specified test task. In Agaric, the key capabilities checked for the assignment include the supporting script language, CPU and memory, network environment and browser, etc. The capability matching component can select the appropriate test nodes from the registered ones according to the properties of testing tasks.

A complete test session begins with test task deploying to qualified test daemon nodes, follows with test execution, and ends with returned test result. The task management component maintains the execution status of all testing tasks. Test center node provides the interfaces for test control node for submitting, scheduling and reviewing testing tasks.

C. *Test Control Node*

Test control node is a layer encapsulating test center node. It provides services to testers and requests services from the center node. Now we design two main test services, test task service and test configuration service, in Agaric to show how testing as a service works. Testers can submit test tasks with the executable test case files, and check the execution results via the test task service. With the test configuration service, testers can specify the size, configuration, and network requirements to request a test network, and deploy and schedule the test tasks.

There are two main concerns on the test service proxy component: one is for reducing the load of center node by filtering or preparing interaction messages, the other is for assuring the reliability of center node by checking and rejecting abnormal interaction requests.

The submitted test configuration script (in python) is interpreted by the configuration engine which will create a process for each configuration script to avoid the cross effects among different tests. Agaric allows simultaneous service access. It means there will be multiple instances of the engine processes running in parallel.

D. *TFCP (Test Flow Control Protocol)*

How to organize the diverse distributed test daemon nodes into a test resource pool and how to manage the communications among the three types of nodes to offer the test services are the key issues of cloud testing. This is why we design the test flow control protocol.

We select the connectionless and stateless HTTP as the base protocol of TFCP. It inherits the C/S communication mode and encodes transaction into <request, response> pair. In

general, test daemon node sends TFCP request and waits for response. The center node will check the sender and return TFCP response after receiving a request, as shown in the following figure 4.

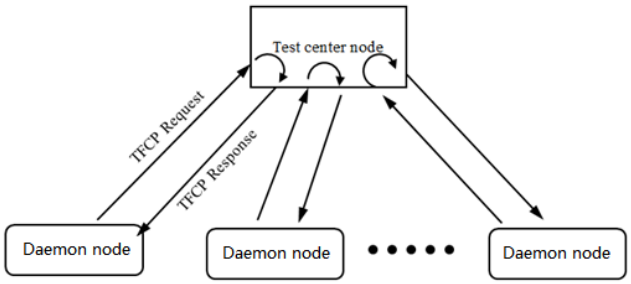

Figure 4 the C/S communication mode of TFCP

There are three main reasons to use the C/S communication mode: (1) there may be a large number of test daemon nodes. If all the communications were initiated by the center node, that would lead to a very large load on the center node; (2) test daemon node may run in a complex internet environment with a proxy or firewall behind. It is difficult for such test daemon node to open a public accessible service; (3) it is not acceptable to open a public accessible service in a user client for the sake of security concerns. Therefore, we design the TFCP from the daemon node whose state machine is presented in figure 5. There are seven states among which five ones are normal working states.

A daemon node will identify itself at first with the ***Initialization*** state. It will initialize standby time, send message queue and test task queue to register itself to the center node. Standby time is a parameter to decide the frequency of sending TFCP requests. If a daemon node successfully registers to the center node, its state will transfer into ***Standby*** in which it will do nothing but maintain the standby timer. Once the timeout event is thrown, it will transfer to the ***Request*** state. If there is a new test task arrived in the queue before timeout event thrown, it will go into ***Execution***. When a task is finished, it will collect the test result and send to the message queue and transfer to the ***Standby*** state.

Within in the ***Request*** state, test daemon node will send TFCP requests to the center node and wait for response. In order to reduce the load on the center node, daemon node will send request to the send message queue and pack all the requests in the queue to send shortly. When a daemon node receives configuration command from the center node, it transfers to ***Configuration*** state in which it will configure itself like clock synchronization, adjusting standby time, computing power assessment and test script downloading, etc. It will transfer to the ***Standby*** state when finishes the configuration.

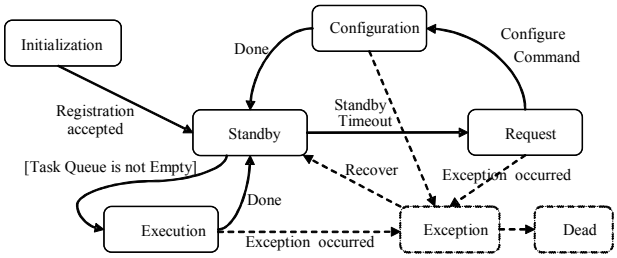

Figure 5 state machine of test daemon node

The ***Configuration***, ***Request*** and ***Execution*** states are protected ones. When an exception is captured, it will go into the ***Exception*** state to analyze the symptom and recover. If the exception is not serious, daemon node will notify the center node and go to the ***Standby*** state. The center node will eventually report the test result including the exceptions to testers. If the exception is serious like internet connection loss, it will be dead. When it connects to internet again, it will register itself as a new daemon node and go into the ***Initialization*** state.

There are two types of messages, request and response message in TFCP, as defined in (1). All messages defined in TFCP have the structure of "start-line CRLF Segment-List". It is line-sensitive and all message lines are delimited by the CRLF.

TFCP-Message = Request | Response (1)

TFCP stack ignores all blank lines before the start-line and empty lines inside a message. The request or response message starts with a request-line (2) or response-line (4). The Segment-List is actually a sequence of request segments (3) or response segments (5) delimited by CRLF. The Request-line carries the information of daemon node identity and TFCP version.

Request-line = UUID SP Nick-Name SP Domain SP TFCP-version CRLF (2)

UUID and Nick-Name are the identity and the display name of a daemon node. Domain is the hostname via which a daemon node connects to internet. The center node uses the domain to do access control. SP is the space character to separate fields. Message in the send message queue will be encoded into one Request-Segment and each Request-Segment includes an action field and data field. The semantics of the action is defined in table I.

Request-Segment = ReqAction SP data CRLF

ReqAction = DEPOLY| RUN| EVALUATE| CMD (3)

The response message has similar structure with the request message. It starts with a response line:

Response-line = standby-time (4)

Standby-time determines the frequency to send requests. To reduce communications, test center node may send response contains multiple pieces of data separated by CRLF.

Response-Segment = ResAction SP data CRLF

ResAction = DEPOLY | RUN@time |STOP | EVALUATE | CMD (5)

TABLE I. SEMANTICS OF TFCP ACTIONS

Action	Semantics	
	in a Response-Segment	*in a Request-Segment*
DEPOLY	Asks daemon node to download and deploy the test suite with given URL parameter.	Reports the result of last finished deployment.
RUN [@time]	Starts test task execution (optionally at the given time).	Returns the last test result and test log.
EVALUATE	Asks to assess the capacity with the provided assessment script in given URL.	Returns the result of last assessment.
STOP	Requires to cancel the execution of test task	
CMD	Used for possible action extensions in the future.	Used for possible action extensions in the future.

E. Formal Verification of TFCP

Colored Petri Net (CPNs) is a language for the modeling and verification of systems in which concurrency, communication, and synchronization play a major role [10]. It is common to use CPN to verify communication protocol [11]. We use the CPN Tools (http://cpntools.org) to model TFCP and verify its reachability and boundedness with its built-in simulation.

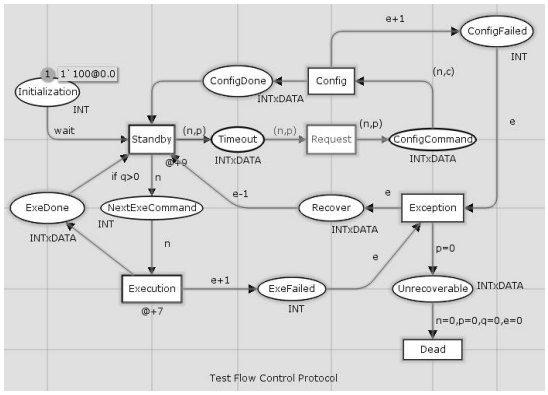

Figure 6 TFCP model in CPN Tools

CPN and state machine have similar model structure. The CPN model in figure 6 is transformed from the state machine of daemon node in figure 5. Where the variable wait is timer value, n refers to the last executed command, p denotes the next request, c is the next response from the center node, e is the number of exceptions captured, q is the number of messages in the send message queue. The simulation shows that every place had a bound, every transition is live and no dead lock was found. This formal verification assures the performance of TFCP and the reliability of Agaric. As we observed in the experiments, Agaric works in an efficient and scalable and way.

IV. EXPERIMENT ANALYSIS

In order to show the effectiveness of Agaric, we designed two experiments. The first one was set up in a controlled LAN environment, and the second one was set up in the real internet environment.

A. Controlled Experiment

In this experiment we have 10 computers to set a local network. In order to check if TFCP can handle the heterogeneity of involved nodes, we use 3 Apple computers, 3 Windows PCs, 2 laptops and two IPod Touches as the test daemon nodes. Laptops and IPods are connected to the test center through a wireless router, as shown in the figure 7. The test center node and test control node are deployed in a local server (Apache and Python CGI installed). To simplify the experiment, we don't use database to store and manage the data.

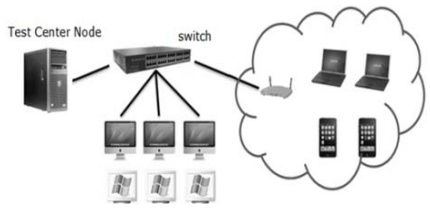

Figure 7 TFCP performance test environment

The SUT selected in this experiment is a simulated DNS server. The hardware and software configuration are listed in table II.

TABLE II. TABLE TYPE STYLES

Item	*Parameter*	Item	*Parameter*
CPU	AMD 3.0Ghz x64	OS	Linux (Cent OS 5.4)
Mem.	2GB	DNS Software	Bind 9.4.2

To test the performance of TFCP, we deploy 100 virtual test daemon nodes on every physical node. All the test daemon nodes continuously put 10000 queries (with a short pause of 100ms) on the SUT to implement a stress test. The number and type of test daemon nodes are specified in the test configuration, as shown in figure 8.

```
from rainbow import * #import the test service lib
from datetime import *
rb = Rainbow()
rb.start()
all_lst = []
for i in range(1000)
lst = rb.getNodes(10,10,'.*python.*')
all_lst += lst
job = rb.assignJob('http://path/dnstest.py',0,t,lst)
wait(all_lst)
print rb.getresult(job['jobid'])
```

Figure 8 test configuration defined in the experiment

The service *getNodes(min,max,rule)* gets the required number (between min and max) of daemon nodes satisfying the rule. The *assignJob(url,at,et,nodelist)* assigns test task (test suite is provided at the given url) to the nodes in *nodelist*, at denotes the time to start the test cases and et is the estimated

task execution length used for exception processing. In the above configuration script, we require 10 python nodes, and assign the *dnstest* task to all the acquired nodes.

The response performance of the DNS server is plotted in figure 9, where the x-axis is time (unit: second) and the y-axis is the scale factor. There are three curves plotted. Curve (a) shows the query number per second, the digit 1 at y-axis means 10,000 queries per second (10,000/qps); curve (b) represents CPU utilization, the digit 1 at y-axis means 100%; curve (c) shows the percentage of request timeouts (statistics once every 50 seconds).

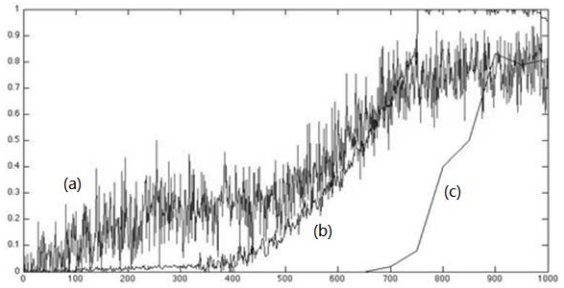

Figure 9 integrated result of the DNS test

We can find in the results that when the number of nodes increases from 700, the request timeouts percentage raises rapidly; CPU utilization increases up to 100%. We can infer that the biggest load of the DNS server is about 7000/qps. This controlled experiment shows the high efficiency of the load testing regardless of its simple configuration and small size of test network.

B. *Uncontrolled Experiment*

In the controlled experiment, we use only the resources deployed in LAN. The result may be biased. In this experiment, we will call for daemon nodes from internet without control. The SUT selected is GPL 2 licensed DokuWiki (*dokuwiki.org*) written in PHP. DokuWiki is similar to Wikipedia having a parser and rendering engine that consumes considerable computing resources. In this experiment, DokuWiki is deployed on the phpnet.us to simulate the test across WAN. The phpnet.us host offers free but limited quotas, i.e. 350MB disk space and 15GB traffic per month.

The test center node and test control node are deployed on Google AppEngine, as shown in figure 10. The guaranteed bandwidth and quality of service offered by AppEngine will ensure the smooth progress of the test.

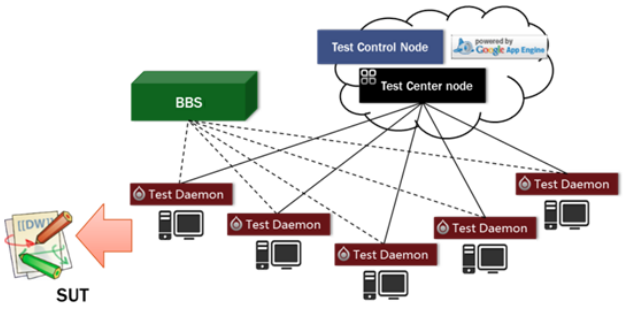

Figure 10 experiment set for DokuWiki testing

In this experiment, we expect to have the daemon nodes as much as possible. Because the daemon nodes called from internet are real user client devices, we cannot install any software in the nodes in this experiment, but use the built-in Actionscript and Javascript interpretation capabilities of the browsers. Test cases are embedded into web page. Once a daemon node accesses the provided web page, the browser will execute the test. If the web page was closed, test would be aborted. For this reason, we have to design the test cases with less test actions. In order to get the expected diversity, we choose a popular forum within China to post a funny flash in Actionscript with rich multimedia content for acquiring long stay time, i.e. long testing time.

The test case used has one POST request and two GET requests, as shown in figure 11. The center node schedules daemon nodes executing the case repeatedly with a pause of 30 seconds to avoid the rejection of SUT.

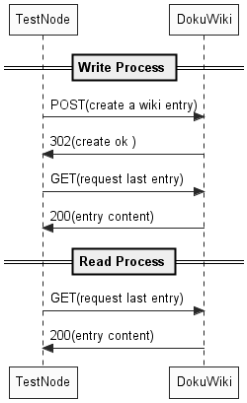

Figure 11 test case designed in this experiment

To assess the SUT performance, the test load increases gradually. The center node will call 50 daemon nodes in every test run. The experiment was started at 6:00 PM and terminated at 11:20 PM because the communications between the daemon nodes and the center node consumed the entire quota offered by AppEngine for that day, as shown in figure 12. We can find that the number began to grow from 7:00 PM and went to the peak at about 9:40 PM (around 800). The number of nodes kept over 500 between 7:50 PM to about 10:30 PM. It is consistent with the habit of most of network users within China. Because the time length of the posted flash movie is about 6 minutes, the center node schedule a test run at every 5 minutes within which all daemon nodes will execute the test case repeatedly.

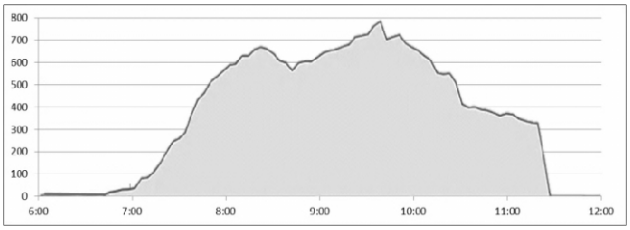

Figure 12 number of daemon nodes called

The total amount of requests sent to the SUT during the 4.5 hours of testing reached 202710. The load is quite promising and it is hard to achieve with the regular load generation tool and strategy. The overall statistics of the test run is shown in figure 13, where x-axis is time (unit: minute) and the unit of y-axis for different plot is marked in the chart.

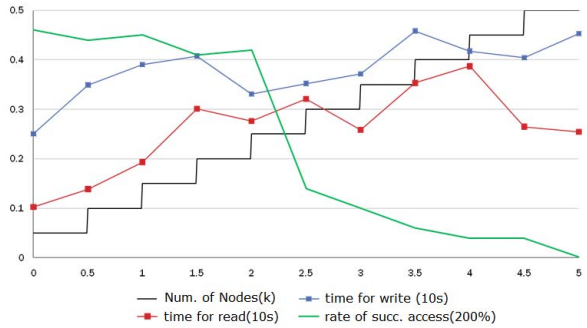

Figure 13 statistics of the test actions

As the number of test nodes increases (the black line), the proportion of successful accesses (the green line) gradually reduced. It rapidly declined to less than 30% at the time point of 2.5 minutes. From this observation, we can guess the maximal concurrent accesses accepted by the SUT are about 250. When the number reached to 500, there was only one daemon node can access successfully. The response time of read (the red line) was significantly less than the write operation (the blue line). That means the write operation is more time-consuming.

In this experiment, there are 1514 test nodes called in total. To show the diversity of the daemon nodes called, the following figure 14 presents the distribution of operating system and browsers.

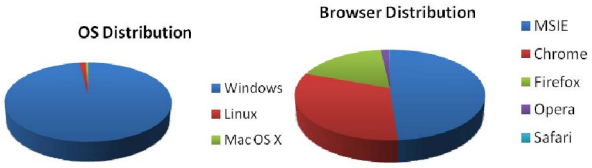

Figure 14 distribution of the OS and browser

The result shown in the left chart reflects roughly the real OS market shares in China. Among the 1514 nodes, 1484 nodes (over 98%) are installed with windows (1077 Windows XP nodes, 407 Windows 7 ones). There are 23 nodes installed with Linux and 7 nodes are installed with Mac OS X. They together share a ratio less than 2%.

To simplify the analysis here, we don't different the versions of browser. We can see in the right chart that there are 739 nodes installed with IE (nearly 49%), 481 ones with Chrome (more than 31%), 267 ones with Firefox (more than 17%), and 25 ones with Opera (less than 2%), only two nodes with Safari (less than 0.2%). The nodes randomly sampled from the hundreds of millions of computers connected to Internet within China reflect the shares of OS and browser in the market very well although the sampling ratio is quite small. The supports the effectiveness of the testing result very well.

Different users may have notable different performance experiences in the practice. We observed this in the experiment, as shown in the figure 15. The node A (OS: Windows 7, browser: Firefox/3.6.13) is from Beijing Unicom and B (OS: Windows XP, browser: IE/8.0) is from BUPT. According to the collected log files, the factor causing the difference is most likely the different browser cache policy set in the corresponding nodes.

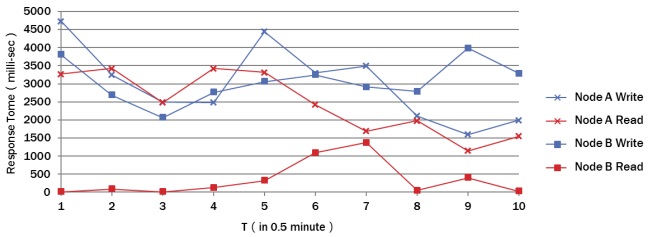

Figure 15 performance observed in different nodes

How does the user's network environment affect the performance user experienced? We chose randomly 30 independent nodes with the IP from Beijing Unicom, CERNET and Xi'an Telecom to check the response time of the write operation. From figure 16 we can see Beijing Unicom was the fastest with the average response time about 3 seconds, while CERNET was the slowest with an average response time of 5 seconds. That means CERNET is more crowded in bandwidth and throughput.

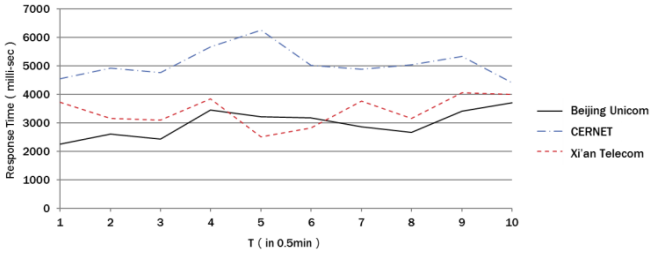

Figure 16 performance differences according to ISP

The statistics and analysis above really show quite effective testing result in this uncontrolled experiment which illustrates great potentials of cloud testing in the future. It is particularly valuable for testing online web applications with the diversely distributed nodes. This can help a service provider continuously get feed of how the users experience the application, catch notable performance loss quickly and automatically. The diversity gained in this experiment provides a strong indication that internet is really a unique and precious testing resource pool. The way of designing and running tests with the services of Agaric also shows the value of applying cloud computing to improve testing.

We still observe some pitfalls in this experiment. The daemon nodes are called in by chance. That increases the uncertainty of testing. The number of nodes called is not so large although the main reason is out of the quota. Therefore, the performance of Agaric may not be thoroughly tested. The SUT and the test case selected are small comparing with real web application. It means the observations may not conclude the situations of applying Agaric for testing large scale web applications.

RELATED WORKS

As stated before, most of the cloud testing tools in the market now focus on the testing in a cloud. For the size limit, we do not survey the testing techniques like test modeling, test design and so on here. Those are easily accessible from the published materials.

Eucalyptus [5], as an open-source software framework for cloud computing, is helpful the design and implementation of Agaric. The large resource pool in cloud computing can support large-scale performance testing [6]. Ganon presents a case study of testing a network management system by using cloud-based testing. Cloud computing imposes impacts on testing in multiple aspects [7]. Different practitioners may have different opinions on using clouding computing for testing. The interview with eleven industry practitioners shows that application, management, legal and financial issues are the most four ones concerned.

Takayuki et al. propose a software testing environment called D-Cloud using cloud computing technology with fault injection facility [8]. D-Cloud allows a user to easily set up and test a distributed system in the cloud. It could emulate hardware faults by using the Eucalyptus [5] and QEMU CPU emulator. Testing as a service is an alias for cloud testing. Lian et al develop a prototype of testing as a service over cloud platform and evaluate the scalability by increasing the test tasks [3].

With the surveys, we find cloud testing is not well touched and investigated in papers. The commercial test tool providers, however, react quickly to release the corresponding new versions that can be deployed in a cloud. This situation is similar as the development of cloud computing. The commercial tools released offer platforms for empirical researches. This is also the goal of the development of Agaric.

CONCLUSIONS

Cloud computing raises challenges along with the benefits it offers to software testing. Cloud testing recently begins to occur in literatures but usually with not clear meanings. There are two major understandings, testing in a cloud and testing toward a cloud. The authors think these cannot conclude the main features of cloud testing. Therefore, the eight main features of cloud testing are identified and discussed. Based on the survey of different models of cloud computing, we propose a hybrid cloud testing platform mixing the models of SaaS and HaaS to take use of the diverse distributed user and centered cloud computing resources to provide online testing services. The design of the Agaric and the TFCP as well as the formal verification is presented. To evaluate the effectiveness of the Agaric, two experiments are conducted and reported with sound results.

The technique used to get user clients from internet in the uncontrolled experiment is not part of Agaric. It only works for the experiment. In the future, we think there should have a business model like HaaS to get and manage the involved user clients. Moreover, how to have people experiences in finding application flaw to provide valuable testing services is a key issue of cloud testing.

Because cloud computing is under developing and cloud testing is still in its infant stage, the authors don't think Agaric can work well in all situations. From its beginning, Agaric is targeted for performance and compatibility testing of online web application. In the future, centered cloud resources will be taken into account and more testing services will be explored. Of course, TFCP needs to improve to fit in more complex test environment and test behavior.

REFERENCES

[1] P. Mell, and T. Glance, "The NIST definition of cloud computing," available at NIST web site.

[2] A. Lenk, M. Klems, et al, "What's in the Cloud: An Architectural Map of the Cloud Landscape," In Proc. of Cloud Computing Workshop of ICSE 2009, pp. 23-31.

[3] Lian Y., Wei-Tek T., et al, "Testing as a Service over Cloud", 5th IEEE International Symposium on Service Oriented System Engineering (SOSE), 2010, pp. 181-188.

[4] "What we test," http://www.utest.com/what-wetest.

[5] Graziano O., Sunil S., et al, "The Eucalyptus Open-source Cloud-computing System", in Proc. of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009, pp. 124~131.

[6] Z. Ganon, and I.E. Zilberstein, "Cloud-Based Performance Testing of Network Management Systems", IEEE 14th International Workshop on Computer Aided Modelling and Design of Communication Links and Networks, 2009. CAMAD '09, June 2009, pp. 1-6, doi: 10.1109/CAMAD.2009.5161466.

[7] Leah M. R., Ossi T., Kari S., "Research Issues for Software Testing in the Cloud", in Proc. of IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), 2010. pp. 557-564.

[8] Takayuki B., Hitoshi K., Ryo K., et al, "D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology", in Proc. of 10th IEEE/ACM International Conference on Cluster, 2010. Pp. 631 – 636.

[9] TTCN-3 language specification, www.ttcn-3.org

[10] Kurt J., Lars M. K., Lisa W., "Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems", Int. J. Software Tools and Technology Transfer (2007) 9:213–254

[11] Yang X., Xiao Y. X., Huan G. Z., "Modeling and Analysis of Electronic Commerce Protocols Using Colored Petri Nets", Journal of Software, vol. 6, no. 7 (2011), 1181-1187