Cloud Simulation under Fault Constraints

Adrian Bosilca, Mihaela-Catalina Nita, Florin Pop*, Valentin Cristea
Faculty of Automatic Control and Computers, University *Politehnica* of Bucharest, Romania
emails: adrian.bosilca@@hpc.pub.ro, catalina.nita@hpc.pub.ro,
florin.pop@cs.pub.ro, valetin.cristea@cs.pub.ro
*Corresponding author

***Abstract*—As Cloud Computing offers support to more and more complex applications, the need to verify and validate computing models under fault constraints becomes more important, aiming to ensure applications performance. Doing this experimental validation in the early development phase and with small costs require a cloud simulation tool. An extensible framework for Cloud simulation and modeling is CloudSim. This paper proposes a new module for CloudSim consisting of a fault injector based on a specification language. The aim is to assist simulation to be more realistic and includes concrete conditions and constraints. The impact is on testing Cloud applications and help test fault tolerant applications by specifying defect patterns and failing components. The evaluation of the fault injection module is done by measuring the behavior and performance of a tool based on CloudSim, named CloudAnalyst. Several metrics are determined and measured for experimental validation, and conclusions are drawn.**

Keywords*—*Cloud Computing, fault injector, fault specification language, simulation, CloudSim, CloudAnalyst.

I. Introduction

Cloud Computing aims to provide the means for developing reliable, sustainable and highly scalable applications as services. Based on the level of abstraction that it provides, cloud computing can be viewed from different approaches: infrastructure as a service, platform as a service or software as a service. Regardless of the logical structure of the system, a cloud is based on a large number of interconnected physical machines, each hosting virtual instances of guest operating systems. Because of the large and distributed nature of clouds, failures in the underlying physical components are a common thing and if not dealt with accordingly, can cause failures in the software hosted on the Cloud. The initial faults that it can cause can vary and they can lead to loss of data or malfunctioning behavior in different areas of the application [15].

Depending on the type of service provided, the outcome of a scenario where faults are present can vary, from fault tolerant application that may not suffer at all because of the faults, to application experiencing performance falls and can even cause complete outages of the service if not dealt properly. To protect against this undesired effects, the software running on the cloud must be fault tolerant and take all the necessary measures to make sure that the system functions normally even in the presence of defects [2].

Before making an application available to the clients, it must first be tested to make sure that it provides all the required features and that it delivers the quality required in those circumstances. In doing so, using a real cloud in the first stages of testing can prove to be costly and might not fully verify all the aspects of the application. To reduce costs and make the software safer and more secure, a set of tests must be developed using a simulated environment. Beside reducing the costs and time required to test the software, a simulation framework has the added advantage that it can be configured to replicate edge scenarios that might not appear until the application has been deployed into production for a long period of time. If the application is already live and there are plans to expand and accommodate the user growth, it becomes more likely that a fault will occur and a simulated environment can be the perfect way to test how it behaves [6].

Fault injection is a technique used in testing complex software applications like cloud services in order to determine the robustness of the software under more realistic scenarios, similar to what it may encounter in a production environment. The advantages of performing this kind of testing is that certain code paths can be tested, that normally would not be executed in a controlled scenario and problems in the applications can be found and fixed early in the application lifetime cycle [9].

The solution proposed in this paper is a fault injector that integrates seamlessly with CloudSim [1] by extending its capabilities, making it able to specify faults either by explicitly specifying them in a script that can be interpreted to model the behavior of the cloud, or by assuming that faults can happen based on a certain probability curve which can be configured. The primary concern on this aspect is the ability to simulate the cloud environment while providing fault injection based on specified patterns and scenarios, that can stress the application and make sure that it behaves as expected in abnormal circumstances like failing machines, loss of connectivity or other faults. By running the simulation with and without faults, evaluation of the performance of the application can be done by comparing the results, considering correctness and speed in order to offer the means to improve the application. A similar solution, presented in [17] an extensible interface to help researchers implement new cloud service reliability enhancement mechanisms.

In the following sections the current capabilities of the cloud simulation framework, CloudSim, are first presented, as well as other related data like cloud computing and introduction to CloudAnalyst [16], followed by an analysis of the fault injection module that we add to it and how it can be used to improve the quality of the software that is tested. The design of the fault injection is presented followed by implementation details and describing the way it integrates with the current CloudSim architecture. In order to draw some relevant conclusions about the fault injector, a real production application will be evaluated: CloudAnalyst. Further on are presented some scenarios where fault tolerance is important and evaluate the impact of the added module in determining if the application is fit to deal with defects that might occur in a

978-1-4799-6569-4/14/$31.00 ©2014 IEEE

real cloud setup. We evaluate the results and present how the algorithm behaved in the presence of faults, comparing it to the results when running in a perfect simulated environment. In the end we draw conclusions and present the future work.

II. Background and Related Work

A. Cloud Computing and Simulation

The Cloud computing model uses a parallel and distributed architecture to offer computing resources to the customer based on a service level agreement. Based on the needs on the computing requests at a certain time, the cloud adapts and scales by allocating more resources and also scales down to save costs, when the demands are not high. To enable this highly versatile system, virtualization is used to achieve high availability and reliability regardless of the physical components on which the datacenter is built on. Clouds are usually based on one or more data centers, each having a very large number of servers organized in racks and interconnected through a wired network. There are several levels on which a Cloud can be modeled at: facility, network, hardware operating system, middleware, application, user

When developing a cloud simulator all of this levels must be considered and implemented accordingly in the simulation system. Depending on the service provided, the cloud can offer three different levels of abstraction: software as a service, platform as a service and infrastructure as a service. With software as a service the user does not need any additional specialized software or hardware to run the application, it just runs on servers in the cloud and the user cant start using with sometimes as little as a web browser or mobile app and an internet connection [7], [11].

Platform as a service and infrastructure as a service offer a lower level of abstraction, providing the user with more powerful means of leveraging the cloud resources, but also transferring a part of the associated complexity to the user. With PaaS, the user has the ability to develop and deploy his own application but based on the provided platform for managing the underlying infrastructure, while IaaS, offers access to the bare bone resources provided which have to be managed by the user. Apart from carefully considering all the components and layers that make up a cloud, we must also bear in mind that while most of the time the system will function as expected, there are some scenarios in which parts of the system will fail and actions must be taken in order not affect the overall functioning of the cloud and the provided services.

CloudSim is an extensible simulation framework that enables seamless modeling and simulation of cloud infrastructures and application services. With the help of this tool, entire cloud infrastructures can be easily simulated, creating the physical topology that mimics the one that could to be used in real datacenters. The infrastructure is specified by key infrastructure components like data centers or racks, which makes the simulation easy to create and maintain while still being relevant to the scenario that is being tested [1], [8].

On top of the simulated physical architecture, virtual instances can be spawned and tasks can be ran on those virtual machines. Among the features that enable good functioning of the system, CloudSim provides the option to use service brokers, scheduling, and allocations policies. The architecture of CloudSim framework is layered, with SimJava discrete event simulation engine at the lowest level. On top of that there is the GridSim layer [4], the CloudSim layer and the user code.

In order to help simulate accordingly this virtualization layer in the simulation, CloudSim provides both physical and virtual machine simulation and allows specification of the policies and scheduling algorithms of both types of instances.

An important aspect of any cloud system is the cost and CloudSim provides the ability to specify and extend different pricing policies and simulate a market by allowing simulation of the following associated properties: cost per processing unit, cost per unit of memory, cost per unit of storage, and cost per bandwidth used.

B. Faults, Errors and Failures

Because of the application complexity, most of the times faults in the system are unavoidable and must be considered a normal part of the environment. In order to properly deal with faults we must properly understand them and the chain of events that lead from defect to fault, error and failures [13].

A fault represents an anomaly in the system that causes it to behave in an unpredictable and unexpected matter. Faults can be classified according to the development phase they appear in, as development faults or operational faults. They can be external or internal based on the boundaries covered by the system, hardware or software, according to what causes them and they can be intentional or unintentional, whether they appear in the testing phase or in production [3], [10].

According to the way they manifest themselves, they can be persistent or transient, sometimes allowing the system to recover successfully from them. Based on the frequency with which they appear they can be reproducible or non reproducible. Reproducible faults allow programmers to more easily fix the fault because they can better observe it [5].

No matter what the fault is, the problems that it might cause are translated to an error in the system. This might not happen and even if it does it might have a certain latency before a fault manifest itself as an error. If an error is detected the system might still continue to work as expected but measures have to be taken to keep the system working as before.

If the error cannot be handled it will cause a failure which is a situation in which the results produced by the system are altered in some way. Only partial failures might occur, but it will still cause either a loss in performance or even complete shutdown of the system. In order to prevent failures in the application, three techniques can be used at different levels of the fault-error-failure chain: fault avoidance, fault removal, fault tolerance. Fault avoidance and fault removal both refer to the fact that faults should be dealt with as soon as possible and not allow them to develop into errors, as errors are harder to handle and can easily lead to failures [12], [14].

Because fault tolerance is a must for many systems, fault injection mechanisms, like the one proposed in this thesis are implemented to simplify the implementation of such features.

III. Fault Injection and Cloud Simuation

Fault injection is a an important technique in system testing that helps determine the dependability of a system. It is used to identify potential flaws in the system and make sure that the application respects the expected fault tolerance level. In order to use fault injection on a system, we must first understand the system under evaluation and determine the components that can fail and the likelihood of each failure. Fault injection can

be done using an experimental approach, by causing faults on the real physical system, or a more practical and cost efficient approach is to emulate the entire behaviour of the physical system through a software program.

As opposed to an experimental approach, is that we cannot use the failures encountered by the system to determine the mean time between failures and the mean time to repair, which are important metrics in determining a system availability. In our approach we are interested in supplying information about the system behaviour under fault constraints, meaning that we are using simulated fault injection, to verify the well functioning of the system in the case of component failures.

The high-level integration scheme of our fault injector, as shown in the Figure 1, contains the fault injector which is based on a library of faults, that can be obtained by statistically analyzing cloud failures in real deployments. Beside the fault injector, we have the workload generator and data collector, backed up by a data analyzer, all linked through a controller and they will all be provided by the CloudSim framework, which will host the fault injector module. Last but not least we have the system that needs to be evaluated, which is represented by the user code layer in CloudSim.

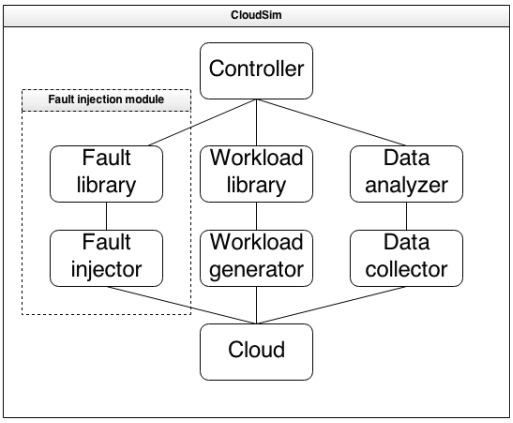

Fig. 1. Fault Injector Integration

A. Fault Injection Techniques

When developing a fault injector for a complex system like a cloud, there are two main problems that must be considered in order to have a clear and well designed architecture: a fault specification language for easy fault injection and the frequency at which fault will occur, which can be manually specified on execution with the help of the fault specification language or it can be deduced and estimated based on the observed distribution patterns in real systems.

1) Fault Specification Language: In order to input the faults into the system under test via the fault injection mechanism, first, a description of the faults that will be injected should be provided that make up the fault scenarios. This scenarios can then be parsed and interpreted in order to generate fault models that can be used by the fault manager to generate actual changes inside the system. The fault specification language can be represented through different forms and structures, ranging from simple list based or key-value based structures to complex branching languages that can be interpreted or compiled and can be used to define complex relationships between the elements of the language.

Because the scenarios for injecting faults are fairly straightforward the chosen option is a simple key-value based representation that can be used to define all the parameters of the fault injection scenarios by using simple key-value objects to build more complex objects that fully define the scenarios.

```
{
    "scenario_id": 12,
    "scenario_name": "Multiple failures",

    "datacenter_fault": {
        "id": 2,
        "minimum_vm_number": 10,
        "random_failures": 0,
        "vm_failures": {
            "vm_id": 1,
            "fault_timeout": 20
        }
    },

    "host_faults": [
        {
            "id": 0,
            "probability": 1.0
        },
        {
            "id": 1,
            "probability": 1.0
        },
        {
            "id": 2,
            "probability": 1.0
        },
    ]
}
```

This means that a lot of faults can be specified with little effort by using arrays or objects that can contain multiple attributes and properties that are converted into faults by the fault injector. In the above listing, an example of such an input file is given and several faults are specified. Beside the scenario name and identifier, a datacenter fault and several physical host faults are specified. Among the parameters specified there is the mandatory identifier field for each component, but also some optional fields like probability or timeout. Because this is a key value language, that is currently used to specify faults it can be easily extended in the future to specify more advanced scenarios and more complex setups.

2) Fault Rate: The fault rate represents the number of faults that appear in a given period of time and can be a measure of the underlying system reliability and stability. No mater how stable the physical system is, faults will still occur at some point, however a fault in the underlying system, does not necessarily translate to a failure in the application.

When a fault appears in a system, disregarding the cause of the fault which might be hardware related or just a human programming mistake, on its own it should not cause the system too much trouble if the application is designed to be fault tolerant. In order to make sure that the inherent faults that appear in a cloud do not affect the service provided by that cloud and the end user, the design of the application should consider faults a highly probable scenario that must be dealt with. The probability and frequency of a fault appearing in the system can be described with the help of mathematical formulas and modeled in to a simulation that can make the

fault injector behave as if the faults appeared in a production system at a naturally occurring rate.

B. Design

In order to achieve a efficient and maintainable solution for the injection module that will be included in CloudSim several design patterns were used: injector pattern, monitor pattern and observer pattern. By implementing the injector using does patterns the end product was a modular and maintainable system that can be easily extended and improved.

1) Observer Pattern: The observer design pattern describes an object, the subject, that maintains a list of other objects, that have a dependency to it, the observers. The subject automatically notifies the observers by sending them a notification, usually by calling a method of the observer. The advantage of using the observer pattern is having a good separation of concerns between objects. The observer pattern is also an important part of the model-view-controller pattern and is related to the publish-subscribe pattern.

2) Mediator Pattern: The mediator pattern encapsulates the interactions between different modules or components of the application. This helps in medium to large projects were the complexity of the application is quite high and the overhead of consistently keeping track of all the interactions can be better mitigated. This pattern is especially useful in maintenance and refactoring, as it helps to more easily keep track of the communication between different classes or components.

3) Monitor Pattern: The monitor pattern represents an object that prevents access to it's methods simultaneously, by using mutual exclusion to prevent other objects from erroneously using it at the same time.

C. CloudSim Injector

The objective of the CloudSim fault injector is to give the user the possibility to choose faults or sets of faults to add to the scenarios they create in CloudSim. There could be a list of faults and defects that the user could specify or he can choose to add multiple faults or make specific faults appear at specific stages of the scenarios based on predefined patterns.

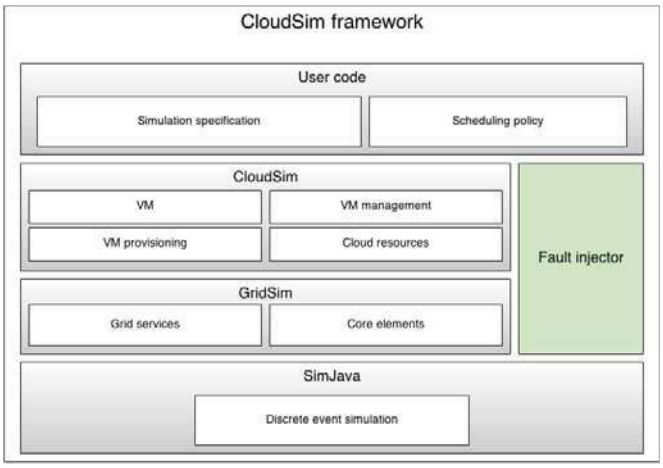

Fig. 2. CloudSim fault injector

D. Fault Tolerance

When injecting defects into an application the primary objective is to observe how the system behaves and if it complies to the requirements of fault tolerance. There are multiple ways in which a system can be made fault tolerant, but most of them involve replicating data on multiple servers, so even if the primary server goes down the data is still accessible.

The main methods in which the fault injection system is useful to the application developer is by providing the means to validate new technologies and models and by offering more realistic scenarios in a convenient simulated environment. By using statistic analysis and data gathered from real cloud usage, we can determine the mean time between failures and the mean time to repair, so we can simulate similar conditions to what the software program will face in a real cloud.

Depending on the requirements of the system, the replication can have strong consistency or eventual consistency. Strong consistency guarantees that when an entry is stored on the primary server it is also stored on the replicas, while eventual consistency relaxes this condition in favor of performance.

The main scenarios that can be tested using the fault injector are fault tolerant protocols for data replication, fault aware machine provisioning systems and fault aware task schedulers. Another scenario that can be tested using a fault injection system is of a distributed hash table that must guarantee persistency and durability to the stored key-value pairs. The performance of the tested program can be evaluated based on two main factors: the first one is the ability to withstand a high number of failures and the second one is to maintain good performance and quality of service in case of failures.

IV. Architecture and Implementation

The fault injection module for Cloudsim was designed to be highly decoupled and work without any changes to the simulations already developed in CloudSim. The implications of this property is that programs and applications based on CloudSim, that have already been implemented using the standard CloudSim framework, can now be ran in the presence of faults to observe the way they behave. The main gain from doing this type of testing and observation is confirming good availability for fault tolerant applications, measuring the drop in performance when different parts of a cloud fail to behave as expected and also make sure that the system will continue to function in the presence of failures and test code paths that normally would not be reached through normal usage. The main components are presented in Figure 3.

1) Simulation Input: The simulation input is composed of Java code that is normally run in CloudSim and serves as the baseline for the injection module and would function normally without the presence of faults. The module hooks into the resource classes of CloudSim and make them perform an additional check before processing of declaring the resource.

2) Fault Scenarios: The Fault Scenarios are implemented as external files that must be provided to the input simulation, either directly or through the injection module and consist in JSON encoded files that specify which component should fail at and the details of the failure. Components are referenced by there IDs from CloudSim and can be configured to fail either certainly or with a specified probability. Beside failure to start or process, timeouts can be specified to delay the execution of otherwise normal scenarios and observe if it causes any disruption in the overall functionality and quality of the service. The main resources that have been targeted so far are individual physical hosts and datacenters as a whole. When

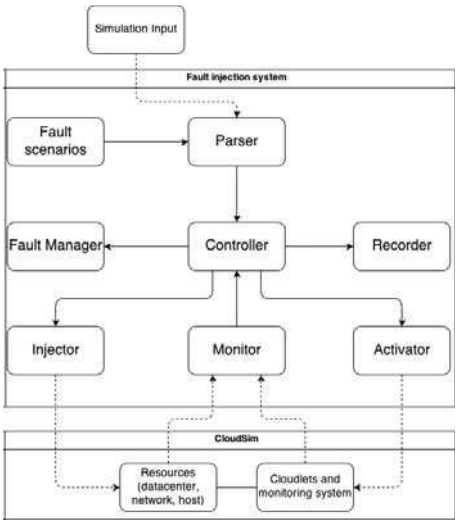

Fig. 3. Fault injector architecture

specifying a failure, the user can see the generated failures in the CloudSim logging system and will see a message even if the component did not fail because the probability of the failure was not 100%. When the datacenter is marked for failure, all hosts in that datacenter will fail regardless of the setting for individual hosts provided.

3) Parser: The parser is implemented with the simple JSON library. The parsed files are transformed into in memory arrays of fault objects that are used by the other components of the injector to generate the faults.

4) Controller: The controller manages interactions between different components of the fault injector and provides high decoupling and easy extensibility of the module features.

5) Fault Manager: The fault manager is used to interpret and process different characteristics of faults and the way they should interact with the simulated environment.

6) Recorder: The recorder stores the results of each simulation that uses the fault injection module from the perspective of generated faults. It can be used to validate the successful generation of faults in the system or track the evolution of scenarios that have a probabilistic outcome. For the moment it does not have a very broad capability but, it can be extended to host a large range of useful statistics and analytics that can lead to better simulations and in the end better applications.

7) Injector: The injector is used to communicate with the CloudSim core and is responsible for decision making from the standpoint of the CloudSim user. While providing just a thin layer between the core of CloudSim and the fault injection module, it is a required component that ensures standardized interface to the module.

8) Activator: The activator is a component inside the module that acts like a switch for the module enabling or disabling part of the features or the entire module as a whole.

9) Monitor: The monitor is a separate component responsible with subscribing for changes from CloudSim and receiving notifications based on different events and triggers. Based on this events the data is collected and communicated through the controller to the recorder to be stored and interpreted or aggregated to form a more meaningful representation.

A. CloudSim Integration

The fault injection module is designed to be as contained as possible and while it does not necesitate any changes in the fault scenarios, it did require a few changes in CloudSim in order to properly make it work with the rest of the existing components. In broad, the fault injection module is linked to CloudSim on two main levels: resources and tasks. Resources are represented in CloudSim by different components that have the ability to do some kind of processing and/or storage.

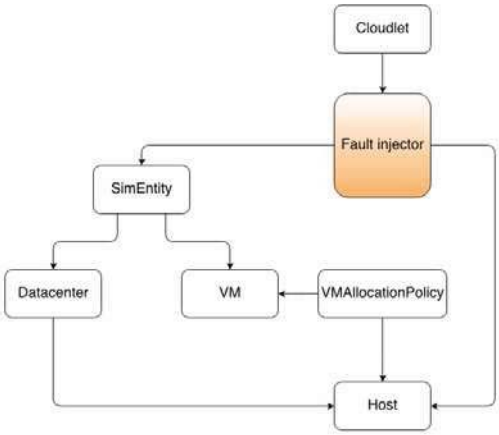

Fig. 4. Extended Simulation Flow

The main entities implemented in CloudSim that are considered resources are: Datacenters, Hosts and VMs (Virtual Machines). Most of this and many others are derived from the SimEntity and ensure uniform handling of events and can be easily extended.

B. Testing Strategy

The testing strategy was based on three phases that could test the fault injection module and also test the applications that would use it and integrate with it. The first phase consists in simple tests to prove that all the components of the fault injection module are working as expected and faults are generated according to specifications. The second phase represents testing it with a real production application that could also give information about the reliability and inner workings of different fault tolerant applications and the third phase has the role of checking the fault injection module in many environments in many real applications and is an ongoing process that necessitates a lot of time and provides data on many implementations of different algorithms that ensure proper operation of cloud environments.

1) Local Testing: The first phase of testing the fault injection module was local testing with the built in tools provided. Cloudsim comes by default with a set of test scenarios that simulate different clouds in increasing complexity order. By applying faults to this predefined set of tests and observing the results, a basic verification of the module was performed. A series of physical hosts were faulted in different datacenters and the logs together with the results examined and checked that they correspond to the expected behavior. This initial testing served as a quick way to validate the module before moving on to the next phase which involves using a real life application that is normally used to simulate clouds without the presence of faults.

2) CloudAnalyst Testing: In this phase of testing, CloudAnalyst was chosen to evaluate the impact of faults for different load balancing algorithms, by using the fault injection module to generate faults in the otherwise perfect environment in CloudSim. As mentioned in the related work section, CloudAnalyst is a tool for evaluating the distribution of datacenters compared to the geographical distribution of users. The application offers the possibility to replicate complex scenarios involving multiple clouds and users. All scenarios begin by configuring the geographical distribution of users and datacenters across the globe. The geographical location is specified by associating a datacenter or a user group to one of six region corresponding to the six continents. The configuration and setup are split into two main sections: Configuration Simulation and Internet Characteristics.

Configuration Simulation In the Configuration Simulation section, the user can specify details about the end points: the user base and the datacenters. Several user bases can be added in any region necessary and the following characteristics can be defined: name, region, requests per user per hour, data size per request in bytes, peak hours start time, peak hours end time, average peak users and average off-peak users. The application deployment can be configured, by specifying a server broker policy that can choose datacenters for user bases according to criterias like closest datacenter, optimize response time or it can be set to dynamically reconfigure. The duration of the simulation can also be specified in seconds.

Internet Characteristics. The internet characteristics section allows the configuration of two matrices: delay matrix and bandwidth matrix. The delay matrix contains the transmission delays from every region two every other region in milliseconds and the bandwidth matrix contains the bandwidth in Mbps for every combination of regions. Together with the configuration simulation section, defining internet characteristics fully specify the parameters needed.

3) Wide Range Testing: In this third phase of testing multiple applications and algorithms implemented based on CloudSim should be used. This is an ongoing process which starts with CloudAnalyst, but will be extended to other programs as well. The purpose of this phase is validation of already implemented applications and algorithms while also verifying that the fault injection module behaves as expected.

C. Evaluation

The evaluation of the fault injection engine is done by first determining the correct metrics that can be measured and are affected by the fault injector and then determining some appropriate scenarios that can be replicated in order to evaluate the experimental data. Because the role of the fault injection module is to insert defects into an otherwise working environment, the main metrics that we are interested in are related to the well functioning and performance of CloudAnalyst: data integrity, correctness, latency and cost.

Integrity. Data integrity refers to ensuring that the data under observation is maintained consistent and accurate at all phases of a given scenario and that the interference that might be caused by the fault injector do not compromise the data. This can be measured by ensuring that the system has the same view on the data with or without the presence of faults regardless of other differences related to performance.

Correctness. Correctness is a metric that can be measured by determining if the system still outputs correct results even if it is subject to partial component failures. As CloudAnalyst can make use of multiple datacenters composed of multiple physical machines linked through a network of routers and switches, some of the components may fail at some point. For determining if the system produces a correct result the focus will be on processing equipment like the physical machines that make up the cloud and not on network gear.

Latency. Latency is the most important metric that the evaluation is focused on as it reveals important aspects about the CloutAnalyst system event when data integrity and correctness is kept. As CloudAnalyst can simulate multiple user bases from different continents and multiple clouds with most of their complex components, it also allows for measuring the time it takes for requests to be processes and the latency of the responses, by measuring round trip times from user bases to cloud and back again. In order to have meaningful and consistent results, to aspects of the latency of responding to requests are measured: overall response time and datacenter processing time. We can isolate and evaluate the impact that failures in processing equipment have on the total processing time, that result in slower replies.

Cost. The cost is another interested metric that CloudAnalyst provides, by making some assumptions that are inputted into the scenario about the cost of each virtual machine and the cost of moving data around. The actual cost that is of interest in evaluating the fault injector is the cost of data transfer.

D. Test Scenarios

In order to test the behavior of CloudAnalyst in the presence of faults, several scenarios were considered that measure the metrics described above with different amounts of failures. In all of the considered scenarios, as long as the system remained operational, with at least one machine working in one cloud, data integrity and correctness was maintained even though the performance of the system was greatly degradated.

The main scenario considered one datacenter situated in North America that replies to requests from users from all of the continents. The average load generated by each user base is described in Table I. The datacenter was built homogeneous in order to gain measurable information when machines fail due to the fault injector. Each machine in the datacenter has the following configuration: x86 architecture, Linux operating system, 204800 MB of RAM, 100 TB of storage and 1GB available bandwidth. This scenario was ran several times with different amount of failures and the results were plotted in the graph from Figure 5. The Graph represents the variation of the overall response time recorded from all six continents, considering increasing number of datacenter failures.

Although they are closely matched, the one that is most interesting in this context is the overall processing time, because the faults that are created by the fault injector are physical host faults that cause a decrease in the total processing power of the datacenter resulting in higher processing times and higher response times.

A total of nine physical machines were simulated hosting 50 virtual machines and the simulation has been ran several times, each time increasing the number of faults by one, until reaching eight which is the maximum number of faults that can be generated in this scenarios without completely faulting the entire datacenter.

TABLE I. GENARATED USERBASE LOAD

Region	Request per user per hour	Data size per request (bytes)	Average number of users
North America	100	1000	1000
South America	100	1000	1000
Europe	100	1000	1000
Africa	100	1000	1000
Asia	100	1000	1000
Australia	100	1000	1000

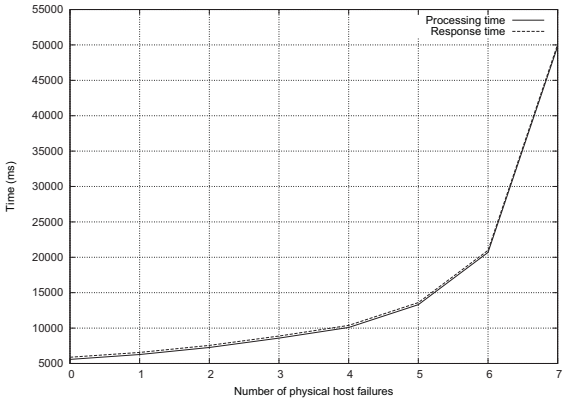

Fig. 5. Overall Response Time Evolution

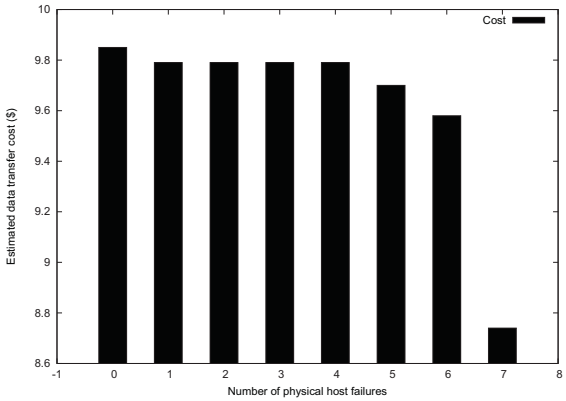

Fig. 6. Estimated Data Transfer Cost

The curve described by the graph has a higher that linear growth, as the number of working physical machines decreases, so does the overall processing power of the cloud and in this strenuous conditions the processing time increases asymptotically, reaching infinite when the number of failures equals the total number of machines in the datacenter.

The response time experiences a steeper growth as the number of faults increases. This is due to the fact that the physical machines decrease proportionally to the number of failures, but the number of virtual machines in the datacenter is attempted to be kept constant. This results in a slight accommodation of all the virtual machines on the remaining hosts at first, but as the number of physical machines decreases, so does their capacity to host additional VMs, which result a more sudden degradation in performance once each physical machine is fully used.

TABLE II. RESPONSE TIME BY REGION (NO FAILURES)

Userbase	Avg (ms)	Min (ms)	Max (ms)
Africa	6162.71	1053.35	12561.94
Asia	6102.25	931.19	12520.97
Australia	5706.25	678.90	12339.26
Europe	5848.11	791.95	12291.44
North_America	5805.47	652.70	12276.90
South_America	5641.79	679.01	12351.26

A more detailed view on the scenario latency response time results is presented in the following tables: Table II presents the response times for all user bases without the presence of faults, Table III presents response times with moderate failures: three failures out of a total of nine and Table IV presents results under heavy failures when only one physical machine is left running, but the system still outputs correct results. In this tables the average time per user base is presented, and it is easy to observe that the users closes to the datacenter have the lowest response times, but additionally the minimum and maximum response time for a response is presented.

TABLE III. RESPONSE TIME BY REGION (MODERATE FAILURES)

Userbase	Avg (ms)	Min (ms)	Max (ms)
Africa	9142.59	1423.35	18523.77
Asia	9074.02	1256.12	18240.94
Australia	8630.11	978.90	17930.84
Europe	9074.39	1100.86	15931.61
North_America	8913.31	852.70	17816.72
South_America	8419.39	979.01	16355.94

TABLE IV. RESPONSE TIME BY REGION (HEAVY FAILURES)

Userbase	Avg (ms)	Min (ms)	Max (ms)
Africa	57004.68	2948.51	160550.99
Asia	42755.88	2973.38	122716.50
Australia	53505.45	5200.85	163738.87
Europe	43410.07	2808.54	123167.93
North_America	48413.43	7552.05	156911.43
South_America	55401.90	2693.06	174219.40

As presented, this scenario focuses on high concurrency and a lot of processing power is needed in order to maintain quality of service which is greatly degraded as the number of failures increase.

The last metric considered is plotted in the graph from Figure 6 and presents an interesting perspective on the failures of the datacenter. Keeping in mind that this plot only represents the estimated cost of the transfer of data and not the cost of the failed physical machines, the graph shows a decrease in cost, caused by the reduced number of virtual machines that can be viably hosted on the working physical machines. On the first few failures, the cost does not significantly go down, but as the number of host goes down, the number of virtual

machines that can be kept up also goes down which results in a decrease in the transfer cost.

1) Fault Tolerance in CloudAnalyst: The summary of the results of running the main scenario on CloudAnalyst, under heavy failures, is presented in Figure 7 Analyzing this and all the results from executing this and other scenarios on CloudAnalyst, the result is that CloudAnalyst implements fault tolerant algorithms for provisioning VMs and can cope with loses in both physical machines and entire datacenters.

Fig. 7. CloudAnalyst Simulation Scenario under Heavy Failures

V. Conclusion

The basic building block for simulating and testing a cloud environment are available through the help of frameworks like CloudSim. What this thesis proposes is an enhancement to the existing software infrastructure, by enriching it with a fault injector that could help detect more elusive and hard problems. While there are still many enhancements that can be made, the proposed solution could help thoroughly test new technologies and models and follow throw the evolution of existing technologies as they become more and more fault tolerant and reliable. It is especially useful when the user does not have the necessary resources to test the application on a real cloud environment, but it is also useful to validate a proposed software solution much more cost efficient. By providing a simple fault specification language for the test scenarios and a modular and clean design that integrates easily with CloudSim, the fault injector is a good candidate to be used with many applications and algorithms that offer fault tolerance in the cloud. The results of the evaluation scenarios tested out with CloudAnalyst are very promising and they proved that the fault injector module can be used successfully to test and validate fault tolerance application. The exciting thing about using the fault injection module proposed in this thesis is that it can help you test the application in complex fault scenarios without requiring any changes or hooks inside the application.

In the future, the third phase of testing needs to be done, by testing the fault injection module with many production application already implemented and it can also be enhanced by a different layer that can suggest potential solutions to the identified problems. Possible future test scenarios are represented by user applications and services but can also be provided by the cloud internal programs like virtual machine provisioning systems and task scheduling algorithms. Meanwhile the fault injection module for CloudSim is a useful tool that can help simulate complex scenarios in cloud architectures under fault constraints, with the purpose of improving the services provided by the cloud.

Acknowledgment

The research presented in this paper is supported by projects: "*SideSTEP - Scheduling Methods for Dynamic Distributed Systems: a self-* approach*", ID: PN-II-CT-RO-FR-2012-1-0084; *CyberWater* grant of the Romanian National Authority for Scientific Research, CNDI-UEFISCDI, project number 47/2012; *MobiWay*: Mobility Beyond Individualism: an Integrated Platform for Intelligent Transportation Systems of Tomorrow - PN-II-PT-PCCA-2013-4-0321; *clueFarm*: Information system based on cloud services accessible through mobile devices, to increase product quality and business development farms - PN-II-PT-PCCA-2013-4-0870.

References

[1] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, Cesar A. F. De Rose, and Rajkumar Buyya. Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1):23–50, 2011.

[2] Changgui Chen, Weijia Jia, and Wanlei Zhou. A reactive system architecture for building fault-tolerant distributed applications. *J. Syst. Softw.*, 72(3):401–415, 2004.

[3] Valentin Cristea, Andrei Lavinia, Ciprian Dobre, and Florin Pop. A failure detection system for large scale distributed systems. *Int. J. Distrib. Syst. Technol.*, 2(3):64–87, 2011.

[4] Yongsheng Hao, Guanfeng Liu, and Na Wen. An enhanced load balancing mechanism based on deadline control on gridsim. *Future Gener. Comput. Syst.*, 28(4):657–665, 2012.

[5] Bahman Javadi, Jemal Abawajy, and Rajkumar Buyya. Failure-aware resource provisioning for hybrid cloud infrastructure. *J. Parallel Distrib. Comput.*, 72(10):1318–1331, 2012.

[6] Bo Hu Li, Xudong Chai, Baocun Hou, Chen Yang, Tan Li, Tingyu Lin, Zhihui Zhang, Yabin Zhang, Wenhai Zhu, and Zenghui Zhao. Research and application on cloud simulation, 2013.

[7] Yang Liu, Maozhen Li, Nasullah Khalid Alham, and Suhel Hammoud. Hsim: A mapreduce simulator in enabling cloud computing. *Future Gener. Comput. Syst.*, 29(1):300–308, 2013.

[8] Wang Long, Lan Yuqing, and Xia Qingxin. Using cloudsim to model and simulate cloud computing environment, 2013.

[9] Paul D. Marinescu and George Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 29(4):11:1–11:38, 2011.

[10] Marieta Nastase, Ciprian Dobre, Florin Pop, and Valentin Cristea. Fault tolerance using a front-end service for large scale distributed systems, 2009.

[11] Alberto Núñez, Jose L. Vázquez-Poletti, Agustin C. Caminero, Gabriel G. Castañé, Jesus Carretero, and Ignacio M. Llorente. icancloud: A flexible and scalable cloud infrastructure simulator. *J. Grid Comput.*, 10(1):185–209, 2012.

[12] Francesco Palmieri, Silvio Pardi, and Paolo Veronesi. A fault avoidance strategy improving the reliability of the egi production grid infrastructure, 2010.

[13] Brian Randell and Maciej Koutny. Failures: Their definition, modelling and analysis, 2007.

[14] Goutam Kumar Saha. Software fault avoidance issues. *Ubiquity*, 2006(November):5:1–5:15, 2006.

[15] Dawei Sun, Guiran Chang, Changsheng Miao, and Xingwei Wang. Analyzing, modeling and evaluating dynamic adaptive fault tolerance strategies in cloud computing environments. *J. Supercomput.*, 66(1):193–228, 2013.

[16] Bhathiya Wickremasinghe, Rodrigo N. Calheiros, and Rajkumar Buyya. Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications, 2010.

[17] Ao Zhou, Shangguang Wang, Qibo Sun, Hua Zou, and Fangchun Yang. Ftcloudsim: A simulation tool for cloud service reliability enhancement mechanisms, 2013.