ReC^{2}S: Reliable Cloud Computing System

Alecsandru Patrascu, Catalin Leordeanu, Ciprian Dobre, Valentin Cristea
Faculty of Automatic Control and Computers,
University Politehnica of Bucharest
email: alecsandru.patrascu@cti.pub.ro,
{catalin.leordeanu, ciprian.dobre, valentin.cristea}@cs.pub.ro

ABSTRACT

In our research we focus on providing essential characteristics such as performance, availability, reliability and security for cloud computing systems, which are becoming more and more popular. In this document we accurately describe the capabilities that our project will provide to its end-users and also specify all the functional and non-functional requirements that the application will implement. We also describe in this document some of our design choices for our solution, along with how we intend to integrate our solution into other existing virtualization solutions and cloud computing software.

INTRODUCTION

Cloud Computing to put it simply, means Internet Computing. Cloud computing (Vaquero et al. 2009) is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

This cloud model (Keahey et al. 2009) promotes availability and is composed of five essential characteristics (on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service); three service models (cloud software as a service (SaaS), cloud platform as a service (PaaS) and cloud infrastructure as a service (IaaS)); and, four deployment models (private cloud, community cloud, public cloud and hybrid cloud). Key enabling technologies include: (1) fast wide-area networks, (2) powerful, inexpensive server computers, and (3) high-performance virtualization for commodity hardware.

The cloud computing model offers the promise of massive cost savings combined with increased IT agility. It is considered critical that government and industry begin adoption of this technology in response to difficult economic constraints. However, cloud computing technology challenges many traditional approaches to datacenter and enterprise application design and management. Cloud computing is currently being used. However, security, interoperability, and portability are cited as major barriers to broader adoption.

The Internet is commonly visualized as clouds; hence the term "cloud computing" for computation done through the Internet. With cloud computing users, for example, can access database resources via the Internet from anywhere, for as long as they need, without worrying about any maintenance or management of actual resources. Besides, databases in the cloud are very dynamic and scalable.

Cloud computing is unlike grid computing, utility computing, or autonomic computing. In fact, it is a very independent platform in terms of computing. The best example of cloud computing is Google Apps where any application can be accessed using a browser and it can be deployed on thousands of computers through the Internet.

We are going to talk in detail about the capabilities of the application that we are going to develop. The main goal of this paper is to present a unified and self-contained platform and framework on top of which end-users can develop custom application in a secure and reliable mode. The users will benefit from the full power of the cloud computing framework in order to reach their desired performance and security degree.

RELATED WORK

Many projects tackle the problem of dynamically overlaying virtual resources on top of physical resources by using virtualization technologies, and do so with different resource models. These models generally consider overhead as part of the virtual resource allocated to the user, or do not manage or attempt to reduce it. A common assumption in related projects is that all necessary images are already deployed on the worker nodes. Our requirements for dynamic deployment of advanced reservations (AR) and as soon as possible (ASAP) workspaces make it impossible to make this assumption.

Amazon Elastic Compute Cloud (EC2) (Ama 2011) is a central part of Amazon's cloud computing platform, Amazon Web Services (AWS). EC2 allows users to rent virtual computers on which to run their own computer applications and allows scalable deployment of applica-

tions by providing a web service through which a user can boot an Amazon Machine Image to create a virtual machine, which Amazon calls an "instance", containing any software desired. A user can create, launch, and terminate server instances as needed, paying by the hour for active servers, hence the term "elastic". EC2 provides users with control over the geographical location of instances which allows for latency optimization and high levels of redundancy. For example, to minimize downtime, a user can set up server instances in multiple zones which are insulated from each other for most causes of failure such that one backs up the other.

The XGE (Fallenbeck et al. 2006) project extends Sun Grid Engine so it will use different VMs for serial batch requests and for parallel job requests. The motivation for their work is to improve utilization of a university cluster shared by two user communities with different requirements. By using the suspend/resume capabilities of Xen virtual machines when combining serial and parallel jobs, the XGE project has achieved improved cluster utilization when compared against using backfilling and physical hardware. However, the XGE project assumes that two fixed VM images are predeployed on all cluster nodes.

The VIOLIN and VioCluster (Ruth et al. 2005) projects allow users to overlay a virtual cluster over more than one physical cluster, leveraging VM live migration to perform load balancing between the different clusters. The VioCluster model assumes that VM images are already deployed on potential hosts, and only a "binary diff" file (implemented as a small Copy-On-Write file), expressing the particular configuration of each instance, is transferred at deploy-time. This approach is less flexible than using image metadata, as COWs can be invalidated by changes in the VM images. Furthermore, our work focuses on use cases where multiple image templates might be used in a physical cluster, which makes it impractical to supply all the templates on all the nodes.

The Maestro-VC (Kiyanclar et al. 2006) system also explores the benefits of providing a scheduler with application-specific information that can optimize its decisions and, in fact, also leverages caches to reduce image transfers. However, Maestro-VC focuses on clusters with long lifetimes, and their model does not schedule image transfer overhead in a deadline-sensitive manner, and just assumes that any image staging overhead will be acceptable given the duration of the virtual cluster. Our work includes short-lived workspaces that must perform efficiently under our model.

The Shirako (Irwin et al. 2006) system developed within the Cluster-On-Demand project uses VMs to partition a physical cluster into several virtual clusters. Their interfaces focus on granting leases on resources to users, which can be redeemed at some point in the future. However, their overhead management model absorbs it into resources used for VM deployment and management. As we have shown, this model is not sufficient for AR-style cases.

The In-VIGO (Adabala et al. 2005) project proposes adding three layers of virtualization over grid resources to enable the creation of virtual grids. Our work, which relates to their first layer (creating virtual resources over physical resources), is concerned with finer-grained allocations and enforcements than in the In-VIGO project. Although some exploration of cache-based deployment has also been done with VMPlant, this project focuses on batch as opposed to deadline-sensitive cases.

CONTEXT

Cloud computing is cost-effective and the cost is greatly reduced as initial expense and recurring expenses are much lower than traditional computing. Maintenance cost is reduced as a third party maintains everything from running the cloud to storing data. The cloud is characterized by features such as platform, location and device independence, which make it easily adoptable for all sizes of businesses, in particular small and mid-sized. However, owing to redundancy of computer system networks and storage system cloud may not be reliable for data, but it scores well as far as security is concerned. In cloud computing, security is tremendously improved because of a superior technology security system, which is now easily available and affordable. Yet another important characteristic of cloud is scalability, which is achieved through server virtualization.

The main form of abstracting the hardware resources, and also the main method of providing the scheduler with information is the lease. The concept of leases is also used in other systems available. Basically, a lease is like a renting contract existing between the user that requests certain resources and the system that offers them. This contract specifies the duration of the renting and is based on the fact that the user will be responsible in that for the resources allocated.

The information that is going to be retained in these leases varies from system to system, but mostly they contain details regarding the processor, the memory that is going to be allocated. More exactly, our scheduler will accept leases that contain the following information, which will be joined together under a lease id: processor architecture, processor vendor, processor speed, processor number of cores, memory size, storage capacity, network bandwidth, network protocol, lease start time, lease end time, lease duration.

GENERAL SYSTEM ARCHITECTURE

The system presented in the following paper has a modular architecture. All modules are described in detail. It is easy to see that the whole ecosystem is actually pluggable and it can be extended with other modules or plugins.

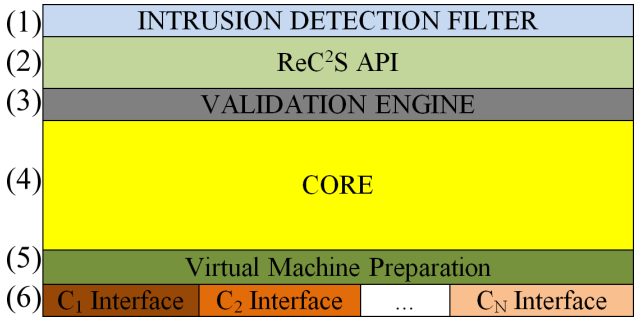

Figure 1: Components of the system

The entire system is composed from 6 layers. We will present them briefly to make a general idea of the components and how they fit in. In the following chapters I will present each of them in detail.

The first layer is responsible for filtering the requests that come from outside the cloud. It basically checks the legality of a certain request to the cloud system.

The second layer is the API layer. The API represents a set of primitives that are offered to the user and permits interaction with the cloud systems. It will be presented as a web front-end and a static API. For example an authenticated user can request to start/stop/restart a certain virtual machine, can push a job to the cloud system or a specific virtual machine, can retrieve answers from the job that he pushed, etc.

The third layer is responsible for checking that the actions that the user specified or requested are actually eligible for executing. The main goal is security - anomaly request detection.

The fourth layer and the most important one is the core of the system. It is responsible with interaction with the cloud systems that he manages. It will do a load balancing of the requests it receives, both in the same autonomous system or inter autonomous systems and it will run a lease based scheduler.

The fifth layer will be responsible with the preparation of the virtual machines that are going to be loaded. This includes finding the appropriate virtual image in the content repository, and, if necessary, installing different software stacks inside.

The sixth layer will be responsible for communication directly to a specific cloud system interface.

Intrusion detection filter

This module is the entry point in our system and it is responsible with its security. It is designed in such way that it can detect attacks that originate from outside. It detects malicious actions like flooding and DoS/DDoS attacks.

ReC²S API

This module offers a way to the user to interact with the system. In our current evolution state we provide means to add new leases. The requests are registered and sent to validation, to the proper module. In order to be more interactive, we provide a graphical user interface, in the form of a webpage.

This module is split in two separate layers. The first one represents a REST-full API implementation and the second one the proper implementation. In our implementation, the API wrapper must permit the use of the following actions:

- setting the lease details. In this section, the user must provide a series of information about the lease that he's creating, like:
  - processor architecture. The user can choose between a 32 or 64 bit architecture. This is important because in this way he can take advantage of different optimization and speedups available to certain processors;
  - processor vendor. The user can choose between Intel or AMD processor type. Also this is important for certain applications that are optimized for a specific vendor;
  - processor speed. This is the actually speed that the processor will do computing;
  - number of processor cores. This is the number of processor/cores that the virtual machine(s) from this lease will have available;
  - memory size. This is the amount of RAM available to the virtual machine(s);
  - storage capacity. This is the amount of disc space required for the virtual machine(s) to run;
  - network bandwidth. This is the speed of the virtual network card that the virtual machine(s) will have;
  - lease start time. This is important when adding a lease because it impacts the way in which the virtual machine(s) are added and started. This time is used in the different scheduling policies inside the core. The user can choose between a determinate or infinite time for the virtual machine(s). An infinite time is the same as having a persistent lease. The determinate time can be a certain timestamp in the future or even a certain event (ex: the user wants to start a lease when data is ready to be processed);
  - lease time end. This is important when adding a lease because it impacts the way in which the virtual machine(s) are ran, stopped and

preempted. This time is used in the different scheduling policies inside the core. Also the user can choose between a determinate or infinite (persistent) lease;

- lease duration. This, in conjuncture with the preemptible part can be used in help of the users that are not running very important tasks and they can permit the virtual machine to be paused and resumed for running more important leases. Also, this is a good way of reducing cost and achieving the desired elasticity. The user can choose between a timed or infinite (persistent) lease;
- preemtible. This offers the user the chance to make his lease prone to pausing and resuming if the scheduler decides to do that.

- setting the virtual machine(s) details. In this section, the user can customize settings for the virtual machine(s) that is (are) going to be run with this lease, like:
  - minimum instances. This is the minimum of virtual machines that the user wants at a certain period, and their number mustn't never get bellow this value;
  - maximum instances. This is the maximum number of the virtual machines that the user wants in heavy load periods. The scheduler must be careful when using this feature to start the machines when a heavy load is detected and stop the extra virtual machines when not needed. This is good for the user because it helps in achieving desired elasticity;
  - the type of the template. This is the operating system that every virtual machine involved in a lease must run. It's implemented as a lease in order to maintain an uniform view of the virtual machines for the scheduler. Currently, the only base template in use is Ubuntu Server 10.04. Later, these templates will be enriched with other Linux operating systems, or even let the user upload their own virtual machine template;
  - network configuration. This is the IP or IP pool of the virtual machine(s) that is going to be used by the user to connect to them.

- setting the packages that the users want to deploy on each virtual machine. This lets the user to auto-install some software on each of the running virtual machine(s). This can include a Java/Python/MPI/etc development framework, a database, etc;
- saving the request for future uses. This is useful for the user because he can save the settings made to the lease, and also the lease itself for future uses.

After adding a lease, the user can access its details and information like:

- lease administration. This permits the user to stop, pause or restart the lease at his desire;
- alerts. This permits the user to have him attention on events like: before starting the lease with T minutes, on lease start, on lease stop, when the scheduler decided when to start the lease.

API action validation

This module receives requests to add new leases in system. Every new request is checked for consistency and validated. If the request is legit and the user making the request is authenticated, the new lease is transformed in a job for our system and it's properly inserted in the job queue.

Core

This module is the most important from the entire system. The core is composed from 4 sub-modules. We will present each of them briefly in order to make a good idea of the core module.

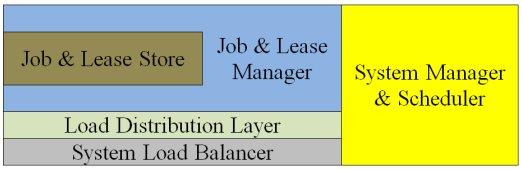

Figure 2: System core

The job and lease manager will mainly consist in a form of data replication and partition. It will provide real-time data aware routing, elastic data access and caching. The job and lease store is basically a submodule of the lease manager that is responsible for storing in a reliable way the things that the cloud system manager and scheduler will process.

The load distribution layer is a sub-module responsible with horizontal scaling the requests received from the scheduler. It will run an application framework in order to decouple the code from the existing underneath runtime.

The system load balancer is responsible for vertically scaling the requests received from the scheduler.

The system manager and scheduler is the most important sub-module. Its main purpose will be scheduling the jobs and leases efficiently between the virtual machines. It will also discover new instances of new services and virtual machines managers, load balancers,

load distributions. It's going to have a pluggable interface in order to be scalable.

Job and lease store

This is the sub-module that will store all lease requests from the outside. These requests are stored as objects in the queue. This sub-module can be implemented in a lot of ways, but at our current stage is implemented as a message queue. When choosing a particular queuing system we analyzed features like: the speed of the sub-system, support for different programming languages (Java, C, C++, etc), standard compliant, data security, data replication, support for different frameworks (Spring, etc), support for different communication protocols (TCP, SSL, UDP, multicast, etc). All these requirements were filled successfully by the open-source library, Apache ActiveMQ.

Job and lease manager

This is the sub-module that will connect the entire core part to the job and lease store. This will be implemented in a generic form and will have to provide access methods like getting an element from the store, putting an element to the store. Because the underneath store can be implemented in different ways and using different architectures, this sub-module should be implemented in a plug-and-play manner, using a pluggable system, so that every access to the store should be made using a specific plugin.

Every plugin must register to the manager. We recommend this approach in order to avoid having to redeploy and restart the manager every time a store is added, or the store API is changing. If this is not a problem for the environment, the whole sub-module can be stopped, modified and then started again.

As we have said above, the manager must provide an interface to the outside world. In our current implementation we used the following ones:

- addLease(L). This call will add to the specific store the lease L. This call must return to the caller a proper value that will reflect if the lease is added to the store and it's ready to be processed, or if the adding the lease to the store has failed. This return value is necessary in order to the caller to take proper actions: either inform the user that everything is good and to wait for the access to the virtual machine(s) that he requested or retry adding the lease to the store;
- getNextLease(). This call will return to the caller the next lease that is going to be processed, if it exists. If the store doesn't have any leases the caller must get an empty store alert in order to retry to call this until there are available leases to process.

In order for this manager to work properly it will have to provide real-time data aware routing. The only abstraction being the underneath store, starting from the upper layers, the system will have proper knowledge of the leases that are requested and offered to the caller.

This will be necessary for providing elastic data access, meaning that the sub-module itself must be capable of auto-balancing in case of high network traffic. Also this shall happen if the stores that it manages will be heavily loaded and/or the request are coming in faster for one manager to handle.

Caching must be another important feature that this module will have to handle. The leases requested from the underneath store will be kept in cache for a certain period of time. We have chosen this approach to fulfill the needs for the persistent leases. This case will rarely be found in practice because only few of the users will want the lease to be persistent.

Load distribution layer

This is be the layer responsible with horizontally scaling our work loads. This is done automatically and in the process of this analysis, the number of workstations is taken in account. If their number changes over the execution of our system, the entire algorithm is rerun to reflect the current situation.

System load balancer

This is the layer responsible with vertically scaling our work loads. This is done also automatically. There is a connection between this layer and the load distribution layer. To be clearer, we will give a simple example of how the two of them work together. Let's assume that we run our system over an infrastructure consisting in four servers, all the same. The system load balancer will detect that the hardware capabilities are the same and will report it to the load distribution layer. Then, the load distribution layer sees that it has four servers available, and after receiving updates from the system load balancer, it will split the entire work load in four parts and will submit them to each server. Now, let's assume that a server will be replace by one, twice as powerful. The load distribution layer will still see four servers, but it will get a notification from the system load balancer that one of the servers is twice more powerful than the remaining three. And so, it will split the work load into five parts and it will submit two parts to the newer and powerful server and to the rest of them, one part per each.

Scheduler

In order for the scheduler to function properly a scheduling scheme has been composed. Depending on the leases and the task, they are split in the following scheduling types and the scheduler will function in three ways:

- Using advanced reservations. This forms the base of the leasing strategies. It reflects fixed resources

that are going to be allocated.

- Using a best-effort strategy. This is formed from another four sub-strategies
  - *Preemptible.* The virtual machines can be started, stopped, paused and resumed at a given time. This process can be somehow compared with the preemption of processes made in the operating system, only that in our case the processes are replaced with virtual machines. To maintain a good system consistency an external clock must be used, to prevent messages loosing between the virtual machines when they are stopped.

    To be more exactly, in the following lines we will talk about this strategy and comparing it with a process. Let's assume that we have a virtual machine running, not exchanging data with any other virtual machine. When the scheduler decides to preempt it, for later reloading or to move it to another network node, it just calls the properly "Pause Virtual Machine" function. Then, the virtual machine is stopped and it's ready for restore.

    But if we have a machine that exchanges data with another virtual machine and it depends on it to function properly, when the scheduler decides to preempt it, all this chain of virtual machines must be stopped in the same time to avoid data loss. The same happens when they are restored - the scheduler must restore them in the same time, again to avoid data loss.
  - *Non-preemptible.* This type of scheduling is also known as "right now allocation". In order to achieve this, a series of information regarding the virtual machine and the place in which the virtual machine will run is required. The scheduler needs data like the actual size of the files that compose the virtual machine, the transfer speed to the destination network and the time needed for the virtual machine to start. For example, if the user wants a lease of one machine, starting in 2 minute, the scheduler must find the appropriate free server to start the virtual machine, and start to transfer the virtual machine template as soon as possible so that at the end of the 2 minute, a virtual machine will be ready for deployment. Also, if the template is copied in place sooner than the starting point, it is no problem because it can wait for the user to start using it. A problem appears if the starting point of the lease is sooner than the time needed to transfer the virtual machine image from the repository to the destination. This can be resolved by using virtual machines caches. This part will be resolved in the future by a different module of the system.
  - *Deadline.* This type of scheduling is also known as "you can allocate anytime, but no longer than T", where T is a fixed time. In order to achieve this, the scheduler must know how much time will the lease last so that he can allocate it before this.
  - *Negotiated.* Depending on the moment in time when you will allocate (Now: 100$, after 1h: 50$, after 2h: 20$, etc)
- Using an urgent lease. This is going to be used when instant resources must be allocated

In order to deploy efficiently through different virtual machines operating systems a process driver must be used. To make this task easier we will use Apache Ant because of its support for multiple architectures and operating systems.

The whole system is seen as in the picture. We will detail each of the modules after.

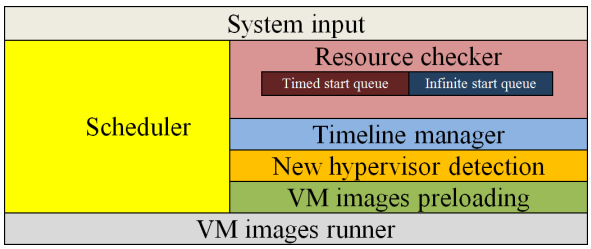

Figure 3: Scheduler architecture

We've mentioned the "System input" layer because in order to work, our system must have entries that contain leases to process. This is a connection layer between the core and the underneath lease storage.

When the input reaches the system it is analyzed and sent to the "Resource Checker" module. This is responsible with checking the availability of resources specified in the lease on the physical systems. It also checks if the system can sustain this requirements on the whole lease duration. This decision is influenced by the current state of the managed systems. It must also keep a safe guard for the required elastic expansion of the virtual machines involved in the allocation of the lease. If this module decides that the lease is safe to be allocated it saves the lease on a special queue, implemented as a priority queue, the "Timed start queue" from the above. If it decides that the physical system cannot hold the lease it saved the lease on another special queue, also implemented as a priority queue, the "Infinite start queue" from the above. The thing that differences the two queues is that in the moment the lease is inserted in the latest, the lease start time is modified to infinite, so that it can start as soon as the system has free resources. We will explain later how this thing is achieved.

Another thing that this module uses is the "Timeline manager", which acts as a global clock and it manages the two queues mentioned above. Mainly its functionality is that, at every defined moment in time, in our case at every minute, to check first for existing leases in the timed start queue. If it founds one, it is automatically deployed on a destination hypervisor chosen from the one attached to the virtual machine runner layer. It also checks if resources are available to run the lease from the infinite start queue.

The "New hypervisor detection" layer is responsible with detection of new hypervisor that are going to be attached and used when running the leases.

The "VM image preloading" layer is responsible with the initialization of the virtual machine image, or cloning an existing one. The initialization creates a new virtual machine from an existing configuration and installs the software stacks chosen by the user. The cloning function just duplicates an existing configuration.

The "VM image runner" is responsible with running the previously created and configured virtual image, on the destination hypervisor. The user will be granted access to the virtual machine at this step.

The core is composed from the "Scheduler" layer. Historically, scheduling was and it's going to be a hard and tricky problem in computer science. Scheduling is concerned with the allocation of scarce resources to activities with the objective of optimizing one or more performance measures. Depending on the situation, resources and activities can take on many different forms. Resources may be machines in an assembly plant, CPU, memory, I/O devices, etc. The form that we are going to study in deeper is called online scheduling.

In our online scheduling form, that we have entitled "ReC2Sched", the scheduler receives jobs during different periods of times. The most interesting part is that it must take decisions without knowing some of the details of the jobs or knowing what will happen in the future. This means that the decisions he is going to make will not be optimal, but with proper algorithms and heuristics this entire process will tend to that.

In our research for the most suitable scheduling algorithms we studied a couple of them. We will present them in the following section, together with our observations:

- randomized algorithms: these algorithms are based on a Monte-Carlo approach and take their decision based on a pseudo-random choice;
- semi-online algorithms:
  - Shortest Job First - is a scheduling policy that selects the waiting lease with the smallest execution time to execute next. This is advantageous because it's simple to use and implement and because it maximizes the average amount of time each lease has to wait until its execution is complete. SJF is rarely used outside of specialized environments because it requires accurate estimations of the time of all leases that are waiting to execute. Estimating the running time of queued leases is done using a technique called "aging";
  - Shortest Remaining Processing Time - is a scheduling method that is a preemptive version of the Shortest Job Next scheduling. In this algorithm, the lease with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progress, leases will always run until they complete or a new lease is added that requires a smaller amount of time. This policy is advantageous because short leases are handled very quickly. The system also requires very little overhead since it only makes a decision when a lease completes or a new lease is added, and when a new lease is added the algorithm only needs to compare the currently executing lease with the new lease, ignoring all other leases currently waiting to execute;
  - First In First Out - is basically an abstraction in ways of organizing and manipulation of data relative to time and prioritization. This expression describes the principle of a queue processing technique or servicing conflicting demands by ordering process by first-come, first-served (FCFS) behavior: what comes in first is handled first, what comes in next waits until the first is finished, etc;
  - High Density First - The algorithm Highest Density First always runs the job with the highest density, which is the weight of the job divided by the initial work of the job;
  - Round Robin - this is one of the simplest scheduling algorithms for leases, which assigns time slices to each lease in equal portions and in circular order, handling all leases without priority (also known as cyclic executive). Round-robin scheduling is both simple and easy to implement, and starvation-free. This kind of policy may not be desirable if the leases size are highly variable. A lease that produces large jobs would be favored over other leases. This problem may be solved by time-sharing, i.e. by giving each job a time slot or quantum (its allowance of CPU time), and interrupt the job if it is not completed by then. The job is resumed next time a time slot is assigned to that lease;

– Shortest Elapsed Time First - this algorithm devotes all the resources to the job that has been processed the least. In the case of ties, this amounts to round robin on the jobs that have been processed the least. While round robin perhaps most intuitively captures the notion of fairness, shortest elapsed time first can be seen as fair in an affirmative action sense of fairness.

Besides implementing our custom scheduling algorithm, for performance comparison we have chosen to implement custom FIFO and SJF algorithms. We will present them bellow.

The first engine is FIFO. This is a simplistic implementation and it processes leases as soon as it arrives. SJF additionally does a lease sort depending on the time of running. In our tests, the times obtained with these two engines were not sufficient to satisfy our need for speed. This lead us to the implementation of our custom algorithm, specially created for our problems. It is different because it does fast lookups into the lease storage unit and it analyses them in advance (we have used in our implementation a 2 minute processing time in advance). Also, our algorithm is a hybrid between the previous two and also is an adaptive algorithm. As a novelty, we are using a 0.5 second computation step. In each step a number of leases is choses to be scheduled.

Virtual machine preparation

This module is intended as a layer between the core and the actual cloud specific API. The most important job that it will do is the following:

- get the decision made by the scheduler;
- analyze it and create a deployment plan. The deployment plan must contain the virtual machine image that is going to be loaded, the software stack that is going to be installed and the appropriate installation script, if such option is chosen by the user;
- preload using the cloud API the image to the chosen location;
- run the installation script. When choosing a certain framework to run our script we analyzed features like: system independent implementation, the ability to run all kind of tasks (creating a folder, deleting a folder, calling certain specific commands). All these requirements were filled by the open-source project Apache Ant

Cloud specific interface

This module is intended as an abstraction layer between our system and the cloud platforms existing on the market. In order to be more scalable and also maintain a high degree of abstraction, an interface is provided and every implementation of the specific API must implement this. This module will act as a plugin manager, with the proper API being inserted at run-time in a plug-and-play way.

For our implementation, as we said at the beginning of the paper, we started with VMware ESXi. VMware ESXi has been designed to be easily adapted to any infrastructure and easily extended with new components. The result is a modular system that can implement a variety of Cloud architectures and can interface with multiple datacenter services.

RESULTS AND OPTIMIZATIONS

To optimize the speed of deployment of the virtual machines requested on a specific lease, we configured a list of virtual machines with predefined software stacks. For example we have in the repository virtual machines templates containing already-installed databases, and different other frameworks. This has a great impact on the entire system because deployment is made faster. This time is won from the time used by the virtual machines images to install the specific software stacks.

To test the scheduler and the system as a whole we have tested all three scheduling algorithms. The scenario was as follows. Suppose that we have 8 users logged in in our system. They each submit a lease, but all with the same time of start. This scenario is a common found in actual production environments.

The hardware platform used was composed from an AMD Phenom II X6, with 8GB RAM, RAID0 configured hard-disks running VirtualBox as hypervisor and an Intel DualCore, 4GB RAM as the scheduler. The network used is 10/100 MB.

In case of FIFO we obtained the following results:

#	Lease creation time (s)	Resource lookup time (s)	Lease lookup time (s)	Virtual machine clone time (s)	Create time (s)
1	9.939	0.100	0.560	9.029	0.250
2	9.940	0.098	0.560	9.029	0.253
3	9.948	0.098	0.560	9.029	0.261
4	9.927	0.097	0.560	9.029	0.241
5	9.926	0.101	0.560	9.029	0.236
6	9.939	0.095	0.560	9.029	0.255
7	9.931	0.098	0.560	9.029	0.244
8	9.951	0.098	0.560	9.029	0.264

In the first column we see the order in which the leases were ran. This is 1, 2, 3, 4, 5, 6, 7 and 8. On the second column we can see the total lease time spent in creation of a lease.

Thanks to our included Virtual Machine preloader, before the actual lease are created, the virtual machine template (800MB) is transferred to the destination. This took about 82 seconds in our test environment. The total time spent was $(\sum Leasecreationtime)+82 = 156.735seconds$

In case of SJF we obtained the following results:

#	Lease creation time (s)	Resource lookup time (s)	Lease lookup time (s)	Virtual machine clone time (s)	Create time (s)
2	9.940	0.098	0.560	9.029	0.253
4	9.927	0.097	0.560	9.029	0.241
7	9.931	0.098	0.560	9.029	0.244
1	9.939	0.100	0.560	9.029	0.250
3	9.948	0.098	0.560	9.029	0.261
8	9.951	0.098	0.560	9.029	0.264
5	9.926	0.101	0.560	9.029	0.236
6	9.939	0.095	0.560	9.029	0.255

We can see from the above table that the lease are picked to be run in a different order, 2, 4, 7, 1, 3, 8, 5 and 6. The total time spent in this case was $(\sum Leasecreationtime) + 82 = 155.947seconds$

In case of ReC2Sched we have obtained the following:

#	Lease creation time (s)	Resource lookup time (s)	Lease lookup time (s)	Virtual machine clone time (s)	Create time (s)
1	9.768	0.055	0.560	9.029	0.124
2	9.767	0.056	0.560	9.029	0.122
3	9.767	0.056	0.560	9.029	0.122
4	9.772	0.056	0.560	9.029	0.127
5	9.764	0.056	0.560	9.029	0.119
6	9.766	0.055	0.560	9.029	0.122
7	9.771	0.057	0.560	9.029	0.125
8	9.766	0.055	0.560	9.029	0.122

We can easily conclude that the times are lower than the previous two engines. Also, because the lease are running in parallel, the total time was $Max(startleasetime + delay) + 82 = 92.264seconds$ which is the lowest obtained in our test.

CONCLUSIONS AND FUTURE WORK

In this paper we presented a solution to provide reliability and security for Cloud users. Our approach takes the form of a complete framework on top of an existing Cloud infrastructure and we have described each of its layers and characteristics. Furthermore, the experimental results prove its efficiency and performance.

As future work we intend to further optimize the scheduling algorithms, as well as the other layers of the framework. We also intend to offer more complex security solutions which would greatly improve the usability of the system. Further testing using more complex scenarios with other existing Cloud infrastructures would also help us to further improve our solution.

Acknowledgments

The research presented in this paper is supported by national project: "TRANSYS - Models and Techniques for Traffic Optimizing in Urban Environments", Contract No. 4/28.07.2010, Project CNCSIS-PN-II-RU-PD ID: 238.

REFERENCES

2011. *Amazon Elastic Compute Cloud (EC2).* URL `http://aws.amazon.com/ec2/`.

Adabala S.; Chadha V.; Chawla P.; Figueiredo R.; Fortes J.; Krsul I.; Matsunaga A.; Tsugawa M.; Zhang J.; Zhao M.; Zhu L.; and Zhu X., 2005. *From virtualized resources to virtual computing grids: the In-VIGO system.* In *Future Generation Computer Systems.*

Fallenbeck N.; Pinch H.; Smith M.; and Freisleben B., 2006. *Xen and the art of cluster scheduling.* In *Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing.* IEEE.

Irwin D.; Chase J.; Grit L.; A.Yumerefendi; Becker D.; and Yocum K., 2006. *Sharing networked resources with brokered leases.* In *USENIX Technical Conference.*

Keahey K.; Tsugawa M.; Matsunaga A.; and Fortes J., 2009. *Sky Computing. IEEE Internet Computing*, 13, no. 5, 43–51. ISSN 1089-7801.

Kiyanclar N.; Koenig G.; and Yurcik W., 2006. *Maestro-VC: A paravirtualized execution environment for secure on-demand cluster computing.* In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and Grid.* IEEE.

Ruth P.; McGachey P.; and Xu D., 2005. *VioCluster: Virtualization for dynamic computational domains.* In *Proceedings of the IEEE International Conference on Cluster Computing.* IEEE.

Vaquero L.; Rodero-Merino L.; Caceres J.; et al., 2009. *A break in the clouds: towards a cloud definition. ACM SIGCOMM Computer Communication Review*, 39, no. 1, 50–55. ISSN 0146-4833. doi:http://doi.acm.org/10.1145/1496091.1496100.