CHASE: Component High Availability-Aware Scheduler in Cloud Computing Environment

Manar Jammal
ECE Department
Western University
London ON, Canada
mjammal@uwo.ca

Ali Kanso
Ericsson Research
Ericsson
Montreal Canada
ali.kanso@ericsson.com

Abdallah Shami
ECE Department
Western University
London ON, Canada
ashami2@uwo.ca

***Abstract*—Cloud computing promises flexible integration of the compute capabilities for on-demand access through the concept of virtualization. However, uncertainties are raised regarding the high availability of the cloud-hosted applications. High availability is a crucial requirement for multi-tier applications providing business services for a broad range of enterprises. This paper proposes a novel component high availability-aware scheduling technique, CHASE, which maximizes the availability of applications without violating service level agreements with the end-users. Using CHASE, prior criticality analysis is conducted on applications to schedule them based on their impact on their execution environment and business functionality. This paper presents the advantages and shortcomings of CHASE compared to an optimal solution, OpenStack Nova scheduler, high availability-agnostic, and redundancy-agnostic schedulers. The evaluation results demonstrate that the proposed solution improves the availability of the scheduled components compared to the latter schedulers. CHASE prototype is also defined for runtime scheduling in OpenStack environment.**

***Index Terms*—High availability, applications, components, virtual machines, outage tolerance, scheduling algorithms, recovery time, criticality, OpenStack, filters.**

I. Introduction

Cloud computing (CC) aims at transforming the data centers' (DCs) resources into virtual services, where tenants can access anytime and anywhere on a pay-per-use basis. CC promises flexible integration of the compute capabilities for on-demand access through the concept of virtualization [1] [2]. Using this concept, a cohesive coupling between the cloud provider's infrastructure and the cloud tenant's requirements is achieved using virtual machines (VMs) mappings [3]. VMs are used to manage software services and allocate resources for them while hiding the complexity from end-users. However, uncertainties are raised regarding the high availability (HA) of cloud-hosted applications.

HA is a crucial requirement for multi-tier applications providing services for a broad range of business enterprises. Planned and unplanned outages can cause failure of 80% of critical applications [4]. According to [5], outages in DCs have tremendous financial costs varying between $38,969 and $1,017,746 per organization. With these complexities, an HA-aware plan that leverages the risks of applications' or hardware's outage, upgrade, and maintenance is necessary. This plan should consider different factors that affect the application's deployment in a cloud environment and the business continuity. Therefore, it is important to develop an HA-aware scheduler for the cloud tenants' applications. This scheduler should implement different patterns and approaches that deploy redundancy models and failover solutions. Single points of failure caused at the level of VM, server, rack, or DC can be eliminated by distributing the deployment of the application's components across multiple availability zones. However, if this placement does not consider the other functional requirements constraining the interdependencies between different application's components, it can jeopardize the application's stability and availability.

In our previous work, a mixed integer linear programming (MILP) model is developed as an optimal solution for components' scheduling in small-scale network [6]. However, this paper follows a more pragmatic approach, where CHASE, component HA-aware scheduler, is proposed. Using CHASE, the availability of applications is attained while considering capacity and delay requirements, applications' criticality, interdependencies, and redundancies. Also, this paper considers different failure scopes and introduces the application's criticality concept to the proposed approach. To achieve this, an analysis is performed to give critical components higher scheduling priorities than standard ones. The HA-aware scheduler evaluates component's availability in terms of its mean time to failure (MTTF), mean time to repair (MTTR), and recovery time. The HA-aware scheduler is compared to the MILP model and OpenStack Nova scheduler in a small data center network [6] [7]. As for large networks, it is compared to greedy HA-agnostic and redundancy-agnostic schedulers. Evaluation results show that the proposed solution improves the component's availability while satisfying the delay and capacity requirements.

In our previous work, the cloud provider's resources and the user's applications were modelled as a unified modeling language (UML) class diagram [6]. This paper puts this model into practice as the basis for our model driven approach to automatically transform the model information into an HA-aware scheduling technique and design its prototype in an OpenStack environment.

This paper is organized as follows. Section II describes the

2159-6190/15 $31.00 © 2015 IEEE
DOI 10.1109/CLOUD.2015.70

cloud-application UML model. Section III defines the HA-aware deployment problem and the proposed solution. Section IV describes the simulation environment and the results of this work. CHASE-OpenStack implementation is discussed in Section V. Finally, the related work and conclusion are presented in Sections VI and VII.

II. HA-aware Deployment Modelling

At the infrastructure as a service (IaaS) level, the cloud provider may offer a certain level of availability for the VMs assigned to the tenants. However, this does not guarantee the HA of the applications deployed in these VMs. For instance, Amazon EC2 has offered recently 3 nines of availability for their infrastructure, which allows several hours of downtime per year [8]. Moreover, the cloud provider is not responsible for the monitory losses caused by the outage. Hence, ensuring the HA of the services becomes a joined responsibility between the cloud provider and user. The provider should offer the VM placement that accounts for the requirements of the tenants' application. As for the cloud tenants, they have to deploy their applications in an HA manner, where redundant standby components can take over the workload when a VM or a server fails. To illustrate this point, we consider the example of a multi-tier HA Web-server application consisting of three component types: the front-end has the HTTP servers, which handle static user requests and forward dynamic ones to the App servers that dynamically generate HTML content. The users' information is stored in the back-end databases (DBs). Fig. 1 illustrates a potential HA-aware deployment of our application example. At the front-end, multiple active (stateless) HTTP servers are deployed on VM_1 and VM_2. They share the requests' load in such a way that if one fails, the other would serve its workload. Most likely, this will incur a performance degradation. The (stateful) App server has a (2+1) redundancy model with one standby backing up the two active ones. At the back-end, one active database serves all the requests, and it is backed up by one standby. The functional dependency among the different component types is clearly visible.

The notion of a *computational path* is defined as the path that a user's request must follow through a chain of dependent components until its successful completion. For instance, in order to process a dynamic request, at least one active HTTP server, App server, and database must be healthy. The components of each type are deployed in a redundant manner forming a *redundancy group*. Upon failure, each component can have a different impact on the global service depending on how many active replicas it has.

The cloud can be modelled in terms of the cloud tenant's applications and the cloud provider's infrastructure deployed in geographically distributed DCs housing various physical servers. We believe that a HA-aware scheduling in the cloud should consider details of both the applications and the cloud infrastructure. Therefore, the configurations of the cloud infrastructure and applications are described in the UML

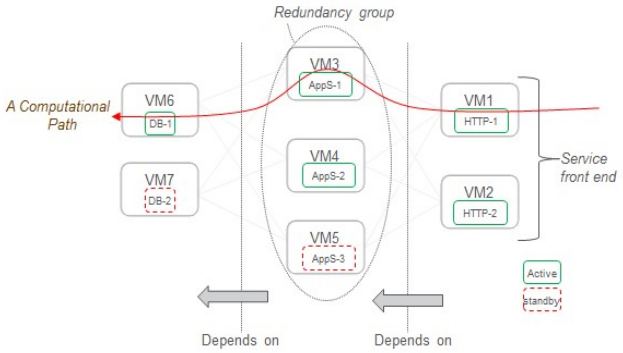

Fig. 1: Example of an application deployment in the cloud.

class diagram shown in Fig. 2 [6]. This diagram models the interactions among many classes working together and provides information required for scheduling the applications in cloud environment. Once the relationships are extracted from the existing diagram, they are translated into a java code. At runtime, the classes are instantiated to give the scheduler objects representing domain classes. Then CHASE performs the scheduling based on an instance of this UML model.

A. Cloud Architecture

The cloud can aggregate a set of geographically distributed DCs. Each DC has its own MTTF, MTTR, and recovery time. The failure of the entire DC is mainly attributed to four factors: a natural disaster causing a complete outage, a prolonged power outage that may cause a switch over to other sites, a human error causing a network interruption and consequently service outage such as Dynamic Host Configuration Protocol (DHCP) requests flooding, and finally a distributed denial of service (DDoS). In each DC, there exists a set of servers residing on different racks with different MTTF and MTTR. A rack's failure is typically caused by the failure of its power distribution unit or the failure of the top of the rack (ToP) switches that disrupt the rack connectivity. Similarly, each server $\{S\}$ has its own MTTF, MTTR, recovery time, and available resources such as CPU and memory.

This architecture can divide the inter-DCs into latency zones and the intra-DCs into latency and capacity (CPU and memory) zones. The inter-latency zone (D_4) can place the requested applications in any physical server in the cloud if the other constraints are satisfied. The intra-latency zone can place the applications either within a data center (D_3), within a rack (D_2), within a server (D_1), or within a VM (D_0).

B. Application Architecture

Each application is composed of at least one component $\{C\}$, which can be configured in one application. Each component belongs to one redundancy group that defines the number of active and standby components. Also, it belongs to a specific component type that defines component's resources' consumption, its functional dependencies, failure types, and scopes.

Two types of dependency are defined; the sponsoring dependency such as the one between the App server and the DB

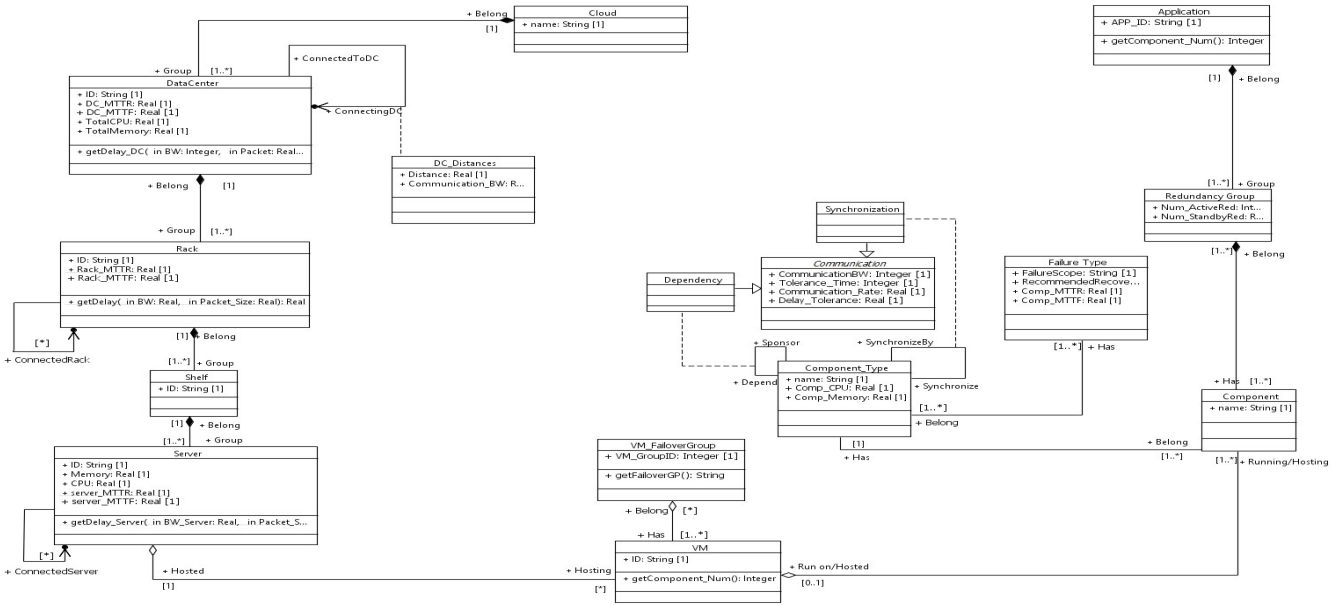

Fig. 2: CHASE UML Model [6].

and the synchronization dependency between components of the same type such as the one between the active and standby replicas of the DB. The delay tolerance, outage tolerance, and bandwidth define the interdependency and redundancy relations between different component types.

C. Cloud-Application Integration

The proposed HA-aware scheduler searches for hosts that maximize the availability of components and their corresponding application. Whenever the host is scheduled, a VM is mapped to it and to the corresponding component. Therefore, each component can reside on at most one VM. Also, each VM can be hosed on one server. A failover group is defined as the set of interdependent VMs (different VMs hosting dependent components). It defines a set of VMs that must failover together in case of unforeseen failure events [9].

III. CHASE: Design and Implementation

The tenant's application is specified as a partial instance of the UML class diagram, where the cloud tenant describes the components forming the application and their requirements/constraints. CHASE performs a criticality analysis to start scheduling components with the highest priority. Then it applies a sequence of filters that starts by sifting out the servers that do not satisfy the functional requirements and then selects the ones that maximize the availability constraints. A brief description of the functional and availability requirements is provided below.

1) Capacity Requirements: They ensure that the selected servers have enough resources (CPU in terms of number of cores and memory in terms of MB) to satisfy the computation demands of the components.

2) Network Delay Requirements: They ensure that the selected servers are at a distance that allows components to interact without interruptions.

3) High Availability Requirements: They are divided into the following constraints:

a) Availability Constraint: It is satisfied by finding the server with high MTTF and low MTTR.

b) Co-location Constraint: It is applied to dependent components that cannot tolerate the recovery time of their sponsor(s). Since the MTTF of a component is inversely proportional to its failure rate λ, the dependent components and their sponsors should be placed in the same server.

c) Anti-location Constraint: It ensures that the components should be placed on different servers. It is applied to redundant components and dependent ones that can tolerate the absence of their sponsors.

A. Criticality Analysis

Performing criticality analysis to applications is a significant step in any emergency or disaster recovery plan. For instance, the contingency plan in Health Insurance Portability and Accountability Act (HIPAA) requires to "assess the relative criticality of specific applications and data..." because they are not equally critical [10]. This is also applicable in HA-aware scheduling, where the highly critical components are given the priority to reside on more reliable servers. In the example shown in Fig. 1, there is only one active instance of the DB; therefore, its failure affects all the incoming requests. This gives the DB a higher impact where the failure of one instance of DB server affects half the requests.

Each component has its own MTTF and MTTR, and therefore

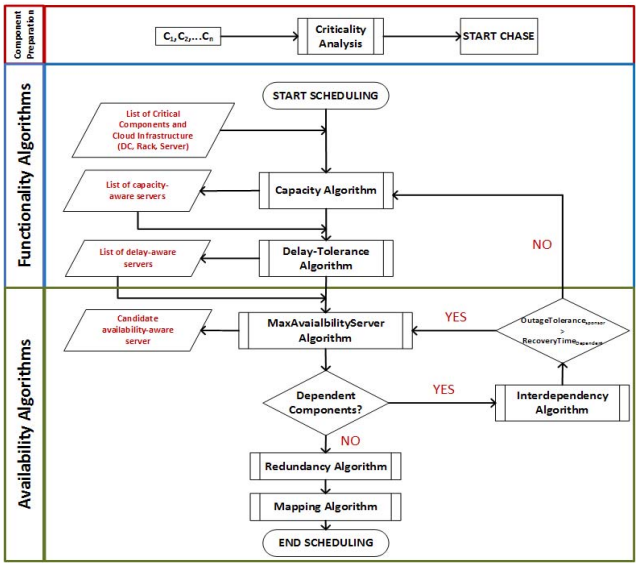

Fig. 3: CHASE Flowchart.

Algorithm 3 Availability Algorithm

INPUT: $A = (A_1, A_2, ..., A_p)$
$C = (C_1, C_2, ..., C_n)$
$S^{cap} = (S_1, S_2, ..., S_k), \quad S^{delay} = (S_1, S_2, ..., S_t)$
OUTPUT: S^{avail}
$where\ S^{avail} \subset S^{delay}$

begin:
$A = AppCriticalityRank(A)$
$C = ComponentCriticalityRank(A)$
for $a_i \in A$ **do**
  if $delay = D_0$ **OR** $delay = D_1$ **then**
    if $S^{cap}_{a_i}.size() isNotNULL$ **then**
      $S^{avail}_{a_i} = MaxAvailability(S^{cap}_{a_i})$
      $UpdateServerCapcity(S^{avail}_{a_i})$
    else
      for $cp_i \in M$ **do**
        $S^{avail}_{cp_i} = MaxAvailability(S^{cap}_{cp_i})$
        $UpdateComputationalPathServer(S^{avail}_{cp_i})$
        $UpdateServerCapcity(S^{avail}_{cp_i})$
      end for
    end if
  else
    for $c_i \in C$ **do**
      $Comp \leftarrow c_i$
      if $c_i.isScheduled = FALSE$ **then**
        $S^{avail}_{c_i} = MaxAvailability(S^{cap}_{c_i})$
        $AllocatedServer.add(S^{avail}_{c_i})$
        $UpdateServerCapcity(S^{avail}_{c_i})$
        $Comp_i.isScheduled = TRUE$
        if $delay = D_2$ **then**
          $RackId = S^{avail}_{c_i}.getRackId()$
          $DcId = NULL$
        else if $delay = D_3$ **then**
          $RackId = NULL$
          $DcId = S^{avail}_{c_i}.getDcId()$
        else
          $RackId = NULL$
          $DcId = NULL$
        end if
        $DependentScheduling(Comp, RackId, DcId, AllocatedServer)$
        $RedundantScheduling(Comp, RackId, DcId, AllocatedServer)$
      end if
    end for
  end if
end for
end

Fig. 4: Availability Algorithm.

its failure can cause either an outage *(o)* of the application or a degradation *(d)* of the service. The criticality value escalates when the failure scope of the component affects not only itself but also its execution environment and its dependent component(s). Generally, front-end *(FE)* components cause a service outage as expressed in (1). If a dependent component *(DeC)* can tolerate the outage of its sponsor *(SC)* until its recovery, then the failure of *SC* causes service degradation as shown in (2). Conversely, the failure of the sponsor causes not only a service degradation but also an outage as expressed in (3).

$$criticality_{FE} = (MTTF \times MTTR)_o \quad (1)$$

$$criticality_d = (MTTF \times MTTR)_d \quad (2)$$

$$criticality_{do} = \sum_{DeC} (Degradation + Outage) \quad (3)$$

$$where \begin{cases} Degradation = (MTTF_{SC} \times OT_{DeC})_d \\ Outage = (MTTF_{SC} \times (OT_{DeC} - MTTR_{SC}))_o \end{cases}$$

The redundancy relation influences the criticality calculation. It adds a weight parameter to the criticality value, which changes according to the number of active and standby instances of the used redundancy model. To finalize the criticality calculation, an impact equation is used to determine the relation between the outage, degradation, and weighted fallouts.

B. CHASE: Component HA-aware Scheduler

CHASE is based on a combination of greedy and pruning algorithms and aims to produce locally optimal results. It is divided into different sub-algorithms as shown in Fig. 3. Each sub-algorithm deals with a specific set of constraints such as capacity, delay, and availability constraints.

1) Capacity Algorithm: Once the most critical application's component is selected, CHASE executes the capacity sub-algorithm. This algorithm traverses the cloud and finds the servers that satisfy the computation resources needed by the requested components. In D_0/D_1 case, the application's components should reside on the same VM/server. Therefore, this algorithm searches for a server that can host them all. If no candidate host is found, the algorithm tries to divide the application into multiple computation paths (if allowed). Then it executes again the search for server(s) to host at least one computational path of the application. Similarly, the algorithm might repeat the above computational path analysis in case the co-location constraints are satisfied for the other delay zones.

2) Delay Tolerance Algorithm: The set of candidate servers satisfying the capacity constraints are fed into the delay sub-algorithm. In this algorithm, a pruning procedure is executed to discard the servers that violate the delay constraint. The delay and availability sub-algorithms are applied to each delay zone. For instance, in D_3 case, this algorithm searches for servers in the same DC to host the component's applications including the redundant ones. If there is not enough servers, the algorithm deals with separate computation paths instead of the whole application.

3) Availability Algorithm: After the delay pruning, communication performance is maintained between various components. At this point, an availability baseline must be achieved. This feature is captured by the availability sub-algorithm shown in Fig. 4. In this algorithm, the servers undergo another stage of pruning that tends to maximize the availability of each component while finding the locally optimal deployment.
Before searching for the server with the highest availability, this algorithm executes the co-location and anti-location algorithms depending on the relation between the tolerance time of a dependent component and the recovery time of its sponsor. If the co-location constraint is valid, the capacity algorithm must be executed again to find a set of servers that satisfies the computational demands of a group of components. Then this set is fed into the MaxAvailabilityServer algorithm to select

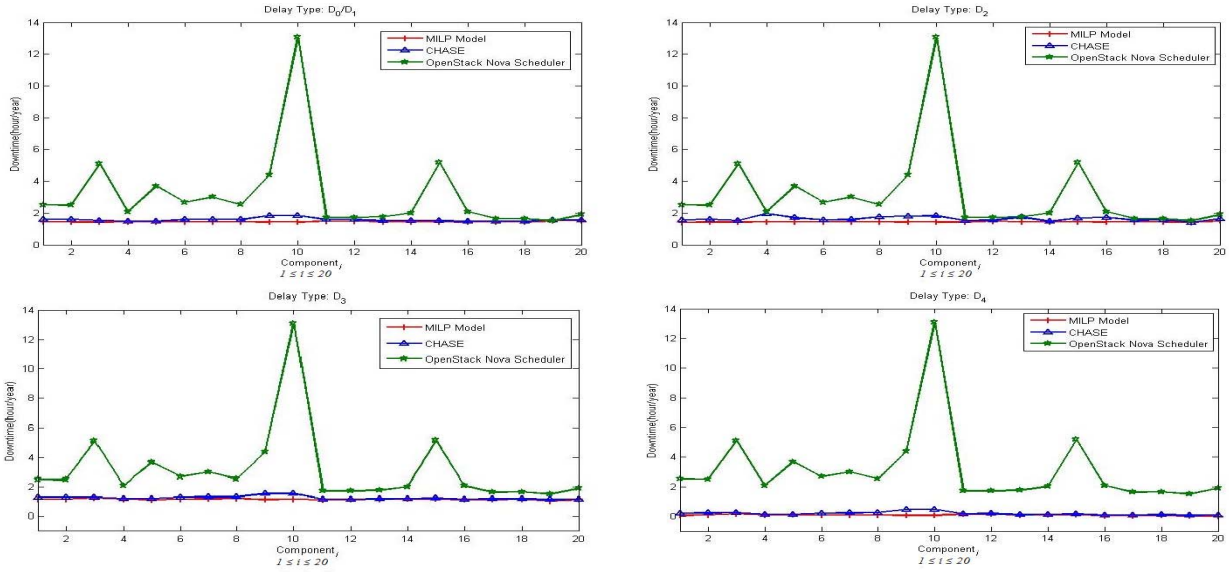

Fig. 5: Downtime of Components in Small-Scale Network.

the server with the highest availability (high MTTF and low MTTR). If the capacity, delay, and availability algorithms indicate that all components can be placed on servers satisfying all the above constraints, the redundancy algorithm is executed to generate placements for the redundant components based on the anti-location constraints.

At this stage, the algorithm has found a host for each component. However, a mapping should be generated among the selected server, the component, and a VM. CHASE executes a mapping sub-algorithm that creates VMs for the scheduled components and then maps them to the chosen hosts.

IV. CHASE Evaluation

To assess the proposed CHASE scheduler, small and large-scale evaluations are conducted using different tiered applications and infrastructure data sets. The MTTF, MTTR, and recovery time are the measures used to quantify the downtime and availability of components.

A. Small-Scale Network Setup

The MILP model developed in our previous work [6], OpenStack Nova scheduler [7], and CHASE are evaluated on a small-scale network. The same network setup in [6] is used to evaluate CHASE. This setup consists of 20 components, 2 DCs, 4 racks, and 50 servers. Each server is configured to have 20 to 30 GB of memory and 16 to 32 CPU cores. VMs are configured in small, medium, and large sizes using OpenStack options [11]. As for the availability measures, they are generated according to distributions shown in Table 1 [12] [13]. To evaluate the interdependencies and redundancies between components, the proposed approach is evaluated on two Web applications. Each application consists of three active databases, two active and two standby App servers, and three active HTTP servers. As for the interdependency relation, App server depends on a database server and sponsors an HTTP server. The results of the small-scale evaluation are shown below.

Attributes	Distribution	Characteristics (hours)
$MTTF_{S-C}$	Exponential	μ=2000
$MTTR_{S-C}$	Normal	μ=3,0.05; σ=1,0.016
$RecoveryTime_{S-C}$	Normal	μ=0.05,0.08; σ=0.016,0.002
$ToleranceTime_C$	Exponential	μ=10

TABLE I: Evaluation Parameters

1) CHASE vs MILP: Fig. 5 compares CHASE with the MILP model for different delay zones. There is a small gap between the MILP and CHASE for all the delay zones. This gap increases as the solution space expands, and it does not exceed 10%.

2) CHASE vs Nova Scheduler: Fig. 5 also compares CHASE to the core and RAM filters in OpenStack Nova scheduler for different delay zones. These types of filters select hosts that can satisfy the resources of components regardless of any other functionality or availability constraints. Therefore, Nova scheduler generates the same results for all delay zones. The Nova scheduler supports certain HA features, such as the notions of availability zones, affinity, and anti-affinity filters. However, it does not support the delay, criticality, and interdependency analysis.

Using CHASE, the downtime of component is reduced by 48%, 34%, and 31% for D_3, D_2, and D_0/D_1 zones respectively. Since D_4 zone distributes the components between DCs, the downtime is reduced by 94% using CHASE. In D_4 zone, if the host, its rack, or DC fails, the hosted component becomes inoperative until it is replaced by its redundant [6].

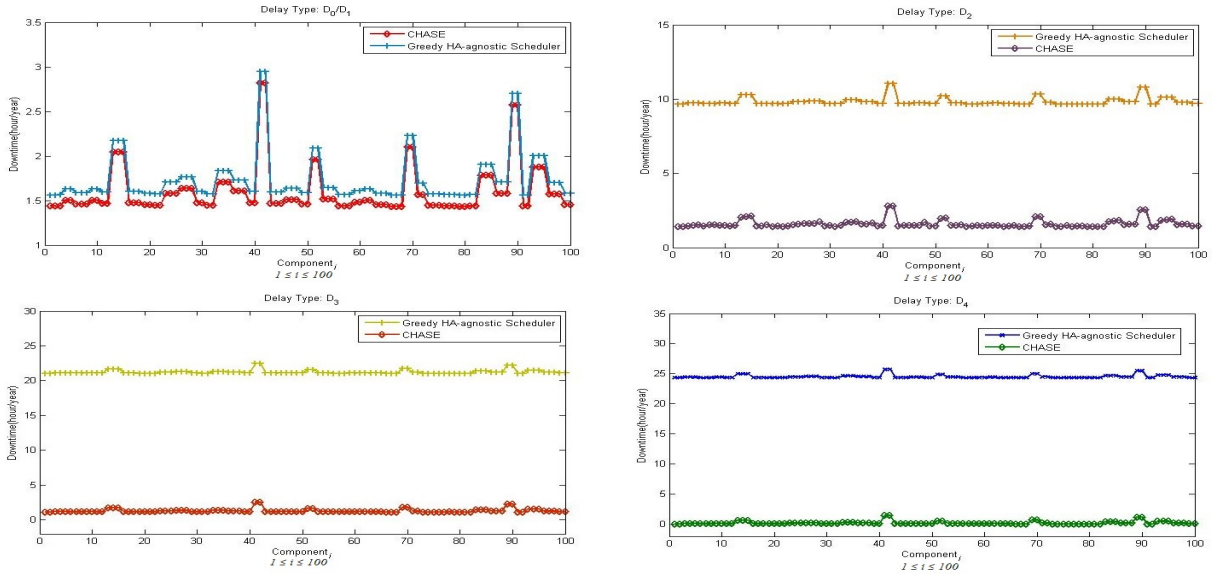

Fig. 6: Downtime of Components in Large-Scale Network.

However, in D_3 zone for instance, the failure of DC affects the hosted components and their redundant ones. Consequently, end-users should wait an execution of a repair policy for the DC or a migration plan for the components.

B. Large Scale Network

Since finding the optimal placement is an NP-hard problem, the MILP solution is only feasible for small networks [14]. Therefore, CHASE is proposed to remedy this issue and schedule cloud hosted applications with a more pragmatic approach. In order to evaluate its scalability, a large-scale network is conducted on CHASE for different delay zones. This network consists of 100 components, 4 DCs, 16 racks, and 1000 servers. The availability measures follow the same statistical distribution shown in Table 1. For the large network, CHASE is evaluated on ten Web applications.

For precision measurement, multiple data sets are generated with the same mean values for the MTTF and MTTR of the component and server shown in Table 1. The confidence level exceeds 95%, which reflects the stability of the results as the scheduling procedure is repeated for different delay zones.The results of the large-scale evaluation are shown below.

1) CHASE vs Greedy HA-Agnostic Scheduler: Fig. 6 compares the component's downtime between CHASE and the greedy HA-agnostic scheduler for different delay zones. The greedy algorithm searches for hosts that satisfy the resources and network delay constraints for components. It considers neither redundancy models, anti-location, co-location, nor availability constraints. Therefore, the gap between both algorithms is large and due to the difference in the placement criterias. CHASE filters the servers according to functionality and availability constraints whereas the greedy algorithm schedules a component on the first available server that satisfies its resources' demands. Although all components are hosted on the same server in D_0/D_1 zone, the availability curve fluctuates because each component type has different MTTF, MTTR, and recovery time. For the other delay zones, the solution space expands, and consequently the gap between CHASE and the greedy algorithm increases. By comparing the graphs, it can be concluded that the D_4 delay zone generates the lowest downtime per year compared to D_3, D_2, D_1, and D_0. The difference between D_4 and D_2 exceeds 85%. Therefore, expanding the solution space and minimizing the delay requirements maximize the application's availability.

Availability Improvement (%)	D_0-D_1	D_2	D_3	D_4
CHASE	99.981	99.981	99.984	99.99
RAS	99.27	99.21	99.1	99.07

TABLE II: Availability Improvement using CHASE

2) CHASE vs Redundancy-Agnostic Scheduler: To show the effect of redundancy on the availability analysis, CHASE is compared to a redundancy-agnostic scheduler (RAS) based on the distributions shown in Table 1. The latter searches for the host that satisfies functionality and interdependency constraints. However, it ignores redundancy models and their effect on the availability analysis. Using CHASE, up to four nines availability can be achieved whereas the redundancy-agnostic scheduler could not exceed two nines availability as shown in Table 2. Using RAS, when a failure occurs, the whole application might become inoperative until a repair plan is applied. Contrary, an inoperative component in CHASE fails over to its redundant component to serve its workload.

Although the component's availability is improved with the increase in the number of available servers, the time complexity of generating the scheduling plan also increases linearly with the number of components.

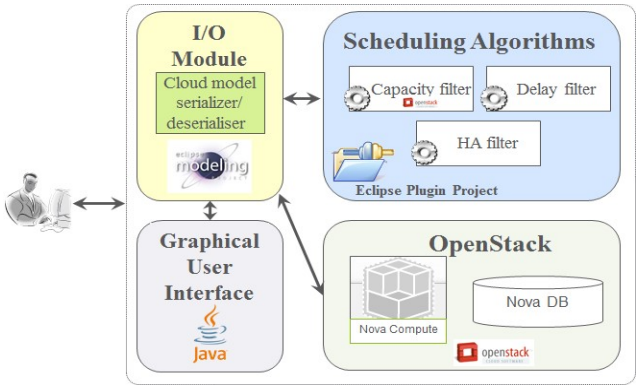

Fig. 7: Architecture of CHASE.

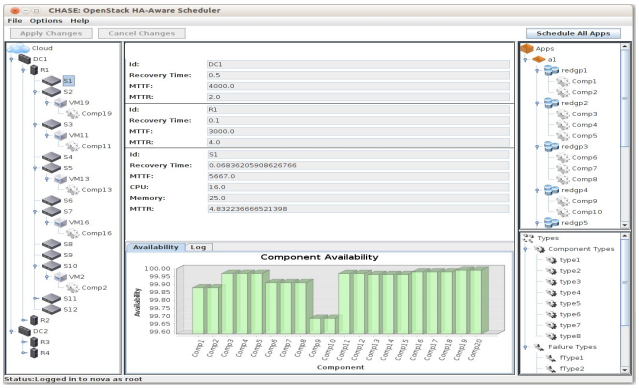

Fig. 8: A screenshot of the CHASE GUI.

V. Prototype Implementation

CHASE prototype is designed to perform scheduling in a real cloud setting. The scheduler communicates with the OpenStack cloud management system, where certain capabilities of the existing filters of OpenStack can be used to complement with CHASE HA filters [15]. The scheduling tool is composed of several complementary modules as shown in Fig. 7. The I/O module is responsible for the information exchange. It communicates with the graphical user interface (GUI) to collect the application information specified by the user. The GUI is used to populate an instance of the cloud-application UML model. It also communicates with the Nova DB of OpenStack, which has been extended to support the notions of DCs and racks. The existing DB table for the hosts is also extended to include the failure and recovery information. The I/O module is also responsible for triggering the CHASE algorithms, collecting the scheduling results, and applying them using the Nova command-line interface (CLI) commands.

Fig. 8 illustrates the CHASE GUI. The GUI contains multiple panels that provide different views of the application's components and the cloud infrastructure. On the right hand side, the user specifies the applications, their redundancy groups, their components as well as their component types and failure types. The user then schedules the applications. This triggers the scheduling algorithm to define the VM placement. Then the I/O module updates the Nova DB and the GUI's left hand side tree, which shows where the components are scheduled.

CHASE is implemented as an Eclipse plug-in project. We use Papyrus to define CHASE UML model. Papyrus is an EMF-based Eclipse plug-in, which offers advanced support of UML modeling [16]. Since Papyrus has limited support for the graphical modeling and Domain Specific Language (DSL) representation, the proposed implementation uses the Java Swing library to define the GUI. The scheduling algorithms are implemented in Java.

VI. Related Work

Application-aware VM scheduling is receiving a great attention in the recent research studies. In [17], the authors develop a placement technique to maximize the availability of applications while satisfying other quality of service (QoS) constraints. The authors divide their solution into search and fit algorithms. The search algorithm finds candidate placements that satisfy delay constraints while maintaining an acceptable reliability level for each application. As for the fit algorithm, it finds the actual placement of the application's component using CPU capacity. Despite the similarities with the objectives of this study, the authors do not consider the interdependencies between the VMs and their associated impact on the availability of the applications hosted by the scheduled VMs.

In [18], the authors address deployment of cloud applications that improves their availability and performance. Although the experimental evaluation shows good results, but the suggested availability analysis is based only on the failures between task executors. However, the authors do not consider the effect of the redundancy models, the dependency relations, and their associated attributes such as the tolerance and recovery times on the applications' availability. Other attempts that address availability of VM deployments are proposed in [19] [20]. While [19] shows the effect of redundancy on availability analysis, both papers have overlooked the effect of dependency models on that analysis.

In [21], the authors propose a VM placement technique that generates redundant configurations to avoid VM outages during host's failures. They aim to generate a minimum number of VMs that could maintain the service performance and quality. Despite the importance of redundancy model on the HA of applications, the authors ignore the effect of delay tolerance, interdependency models, MTTF, MTTR, recovery, and tolerance times on maintaining certain fault-tolerance level. Both [22] and [23] propose a failover plan during the placement problem using different approach schemes. While [22] exploits the co-location and anti-location constraints between interdependent applications, their capacity, and security constraints to provide a pseudo-optimal failover plan for an application, [23] assigns each VM a resiliency level that enables it to relocate to a new host if its current host fails. It also uses the anti-location and co-location constraints between VMs to create a backup for them against any failure.

In [24], the authors propose a load balancing-aware scheduling algorithm of VM resources. Using a scheduler controller and

a resource monitor, the algorithm collects historical data and system state. This data is loaded into the genetic algorithm to generate a mapping solution for each VM while minimizing the issues of imbalance load distribution and migration cost. Similarly, [25] develops a dynamic and integrated load balancing scheduling algorithm (DAIRS) for cloud DCs. The authors provide an integrated measurement for the imbalance level of a DC as well as its servers. Using the latter values, they propose load-balancing aware VM scheduling and migration algorithms. Although the authors maximize the resource utilization, they ignore the availability constraints and failure impact on the VM scheduling and service continuity.

Each of the previous literatures has considered different strategies to maximize applications' availability. Some approaches consider redundancy and failover solutions while others look at MTTF and recovery time of components. However, this paper proposes a novel scheduling technique that looks into the interdependencies and redundancies between application's components, their failure scopes, their communication delay tolerance, and resource utilization requirements. It examines not only MTTF to measure the component's downtime and consequently its availability, but the analysis is based on the MTTR, recovery, and outage tolerance times as well.

VII. Conclusion

With the broad range of business applications, high availability is gaining a great attention in enterprises and information technology (IT) sector. In order to deploy highly available applications, a solid HA-aware scheduling strategy that considers functionality requirements, real-time interdependencies, and redundancies between applications is needed. Attaining an always-on and always-available service was the main objective of the proposed HA-aware scheduler. The algorithm was evaluated for different delay zones and different communication relations between components. CHASE prototype was designed to schedule components in real cloud environment while communicating with OpenStack. In future work, the proposed approach will be extended to capture dynamic scaling and migration features to compensate any sudden changes in the application's behavior or infrastructure characteristics.

Acknowledgment

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC-STPGP 447230) and Ericsson Research. The authors would like to thank Philip Kurowski for his contribution in the prototype implementation.

References

[1] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "NFV: State of the Art, Challenges and Implementation in Next Generation Mobile Networks (vEPC)," *IEEE Network,* vol. 28, pp. 18-26, December 2014.

[2] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, "Software Defined Networking: State of the art and research challenges," *Elsevier Computer Networks*, vol. 72, pp. 74-98, October 2014.

[3] M. Abu Sharkh, M. Jammal, A. Shami, and A. Ouda, "Resource allocation in a network-based cloud computing environment: design challenges," *IEEE Communications Magazine,* vol. 51, pp. 46-52, November 2013.

[4] HP, "The High Availability challenge: 24x7 in a Microsoft environment," http://h71028.www7.hp.com/enterprise/downloads/4AA0-3147ENA.pdf, November 2010. [February 5, 2015]

[5] NetMagic, "Data center outages impact, causes, costs, and how to mitigate," http://www.netmagicsolutions.com/uploads/pdf/resources/whitepapers/WP_Datacenter-Outages.pdf, 2013. [February 10, 2015]

[6] M. Jammal, A. Kanso, and A. Shami, "High Availability-Aware Optimization Digest for Applications Deployment in Cloud," *IEEE International Conference on Communications (ICC)*, June 8-12, 2015.

[7] OpenStack, "Filter Scheduler," http://docs.openstack.org/developer/cinder/devref/filter_scheduler.html#costs-and-weights, Februray 2013. [January 18, 2015]

[8] Amazon EC2, "Amazon EC2 Service Level Agreement," http://aws.amazon.com/ec2/sla/, 2014. [January 18, 2015]

[9] P. Bodik, F. Armando, *et al.*, "Characterizing, modeling, and generating workload spikes for stateful services," *1st ACM symposium on Cloud computing,* pp. 241-252, June 10-11, 2010.

[10] HIPAA, "Contingency Plan: Applications and Data Criticality Analysis-What to Do and How to Do It," http://www.hipaa.com/2009/04/contingency-plan-applications-and-data-criticality-analysis-what-to-do-and-how-to-do-it/, 2014. [February 9, 2015]

[11] OpenStack, "OpenStack Operations Guide," http://docs.openstack.org/openstack-ops/openstack-ops-manual.pdf, 2015. [February 17, 2015]

[12] Reliability HotWire, "Availability and the Different Ways to Calculate It," http://www.weibull.com/hotwire/issue79/relbasics79.htm, September 2007. [December 20, 2014]

[13] EventHelix, "System Reliability and Availability," http://www.eventhelix.com/realtimemantra/faulthandling/system_reliability_availability.htm#.VL62B0fF83n, 2014. [January 16, 2015]

[14] O. Kone, C. Artigues, P. Lopez, and M. Mongeau, "Event-based MILP models for resource-constrained project scheduling problems," *Computers and Operations Research,* vol. 38, pp. 313, January 2011.

[15] OpenStack, "OpenStack Cloud Software," http://openstack.org. [February 10, 2015]

[16] Papyrus Eclipse Project, https://www.eclipse.org/papyrus/. [February 10, 2015]

[17] G, Jung, K.R. Joshi, M.A. Hiltunen, R.D. Schlichting, and C. Pu, "Performance and availability aware regeneration for cloud based multitier applications," *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN),* pp. 497-506, June 28-July 1 2010.

[18] J. Li; Q. Lu, *et al.*, "Improving Availability of Cloud-Based Applications through Deployment Choices," *IEEE Sixth International Conference on Cloud Computing (CLOUD),* pp. 43-50, June 28-July 3 2013.

[19] Q. Lu, X. Xu, *et al.*, "Incorporating Uncertainty into In-Cloud Application Deployment Decisions for Availability," *IEEE Sixth International Conference on Cloud Computing (CLOUD),* pp. 454-461, June 28-July 3 2013.

[20] W. Wenting, C. Haopeng, and C. Xi, "An Availability-Aware Virtual Machine Placement Approach for Dynamic Scaling of Cloud Applications," *9th International Conference on Ubiquitous Intelligence & Computing Conference on Autonomic & Trusted Computing (UIC/ATC),* pp. 509-516, September 4-7, 2012.

[21] F. Machida, M. Kawato, and Y. Maeno, "Redundant virtual machine placement for fault-tolerant consolidated server clusters," *IEEE Network Operations and Management Symposium (NOMS),* pp. 32-39, April 19-23, 2010.

[22] R.E. Harper, R. Kyung, *et al.*, "DynaPlan: Resource placement for application-level clustering," *IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W),* pp. 271-277, June 27-30, 2011.

[23] E. Bin, O. Biran, *et al.*, "Guaranteeing High Availability Goals for Virtual Machine Placement," *31st International Conference on Distributed Computing Systems (ICDCS),* pp. 700-709, June 20-24, 2011.

[24] H. Jinhua, G. Jianhua Gu, S. Guofei, and Z. Tianhai, "A scheduling strategy on load balancing of virtual machine resources in cloud computing environment," *IEEE third International Symposium on Parallel Architectures, Algorithms and Programming (PAAP),* pp. 89-96, December 18-20, 2010.

[25] T. Wenhong, Z. Yong, Z. Yuanliang, and X. Minxian, "A dynamic and integrated load-balancing scheduling algorithm for Cloud datacenters," *IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS),* pp. 311-315, September 15-17, 2011.