4-28-2017 12:00 AM

MACHS: Mitigating the Achilles Heel of the Cloud through High Availability and Performance-aware Solutions

Manar Jammal, *The University of Western Ontario*

Supervisor: Professor Abdallah Shami, *The University of Western Ontario*
A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Electrical and Computer Engineering

Abstract

Cloud computing is continuously growing as a business model for hosting information and communication technology applications. However, many concerns arise regarding the quality of service (QoS) offered by the cloud. One major challenge is the high availability (HA) of cloud-based applications. The key to achieving availability requirements is to develop an approach that is immune to cloud failures while minimizing the service level agreement (SLA) violations.

To this end, this thesis addresses the HA of cloud-based applications from different perspectives. First, the thesis proposes a component's HA-ware scheduler (CHASE) to manage the deployments of carrier-grade cloud applications while maximizing their HA and satisfying the QoS requirements. Second, a Stochastic Petri Net (SPN) model is proposed to capture the stochastic characteristics of cloud services and quantify the expected availability offered by an application deployment. The SPN model is then associated with an extensible policy-driven cloud scoring system that integrates other cloud challenges (i.e. green and cost concerns) with HA objectives. The proposed HA-aware solutions are extended to include a live virtual machine migration model that provides a trade-off between the migration time and the downtime while maintaining HA objective. Furthermore, the thesis proposes a generic input template for cloud simulators, GITS, to facilitate the creation of cloud scenarios while ensuring reusability, simplicity, and portability. Finally, an availability-aware CloudSim extension, ACE, is proposed. ACE extends CloudSim simulator with failure injection, computational paths, repair, failover, load balancing, and other availability-based modules.

Keywords: Cloud computing, High availability, Virtual machines, Dependability analysis, Petri Net, Load balancing, Component-based architecture, Scheduling, Live migration, Failover, CloudSim, Criticality, Redundancy, Interdependency, Computational path, OpenStack, JSON, Eclipse GMF.

Co-Authorship

This thesis contains the following manuscripts that have been submitted, accepted, and published.

1. M. Jammal, A. Kanso, and A. Shami, "High Availability-Aware Optimization Digest for Applications Deployment in Cloud," *IEEE International Conference on Communications (ICC)*, June 2015.
2. M. Jammal, A. Kanso, and A. Shami, "CHASE: Component High-Availability Scheduler in Cloud Computing Environment," *IEEE International Conference on Cloud Computing (CLOUD)*, June 2015.
3. A. Kanso, M. Jammal, and A. Shami, Component High Availability Scheduler, P44248 US1, October 2014.
4. M. Jammal, A. Kanso, P. Heidari, and A. Shami, "A Formal Model for the Availability Analysis of Cloud Deployed Multi-Tiered Applications," *IEEE International Symposium on Software Defined Systems*, April 2016.
5. M. Jammal, A. Kanso, P. Heidari, and A. Shami, "Availability Analysis of Cloud Deployed Applications," *IEEE International Conference on Cloud Engineering*, April 2016.
6. A. Kanso, P. Heidari, and M. Jammal, High availability multi-component cloud application placement using stochastic availability models, P48033US1, November 2015.
7. M. Jammal, A. Kanso, P. Heidari, and A. Shami, "Scrutinize High Availability-aware Deployments Using Stochastic Petri Net Model and Cloud Scoring Selection Tool," *Submitted to IEEE Transactions on Services Computing*, November 2016.
8. M. Jammal, H. Hawilo, A. Kanso, and A. Shami, "Mitigating the Risk of Cloud Services Downtime Using Live Migration and High Availability-Aware Placement," *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, December 2016.
9. M. Jammal, H.Hawilo, A. Kanso, and A. Shami, "GITS: Generic Input Template for CloudSim and Cloud Simulators," *Submitted to Elsevier Future Generation Computer Systems*, 2017.
10. M. Jammal, H.Hawilo, A. Kanso, and A. Shami, "ACE: Availability-aware CloudSim Extension," *Submitted to IEEE Transactions on Cloud Computing*, 2017.

The following co-authors provided experimental and technical support for the studies listed above:

- A. Shami supervised the development of the work and provided technical expertise, opinion, and perspective based on his experience as a professor at Western University. He supervised the work done in Chapter 2, Chapter 3, Chapter 4, Chapter 5, and Chapter 6.
- A. Kanso supervised the development of the work and provided technical expertise, opinion, and perspective based on his experience as a researcher at Ericsson, Montreal and as a senior software engineer at IBM Watson Center, NY, USA. He supervised to the work done in Chapter 2, Chapter 3, Chapter 4, Chapter 5, and Chapter 6.
- H. Hawilo provided technical opinion on the design of GITS and ACE and helped in the manuscripts preparation and review based on his experience as a Ph.D. student at Western University. He contributed to the work done in Chapter 4, Chapter 5, and Chapter 6.
- P. Heidari provided technical expertise on Petri Nets, helped in designing the cloud scoring tool, and helped in analyzing Petri Net results based on her expertise as a Postdoctoral researcher at Ericsson, Montreal and Western University. She contributed to the work done in Chapter 3.

This thesis work is dedicated to my beloved husband and great companion, Hassan, and to the memory of my grandfather, Ahmad.

Acknowledgements

Doing my Ph.D. has been a turning point for me. It is true that pursuing Ph.D. studies is a challenging step, but it is worth working for. During this journey, I have learned to hold on regardless of obstacles. This would not have been possible without the support and help of great people. I am grateful for all of them!

First, I would like to thank my supervisor Prof. Abdallah Shami. He has been very supportive since the days I began working with him in the OC2 group. I would like to express my sincere appreciation for the motivation and guidance he gave me during my Ph.D. studies. His immense knowledge and guidance helped me in all the research work. Prof. Shami has supported me not only by providing a research supervision, but he was there for me as a mentor and friend especially during the rough moments of this thesis. Thanks to him I also had the opportunity to work with Ericsson Research, Montreal where I had the chance to meet wonderful people there! Here, I would like to express my sincere gratitude to Dr. Ali Kanso for his encouragement to keep up the hard work. With his knowledge, he guided me through my thesis and made sure that I am always on the "right track". I am very grateful for the invaluable discussions, suggestions, and meetings. Above all, thanks for having him as a mentor whom I can always count on his support.

Special thanks to my thesis and advisory committee members Prof. Ben Liang, Prof. Hanan Lutfiyya, Prof. Jagath Samarabandu, Prof. Aleksander Essex, and Prof. Serguei Primak for taking the time to read the thesis and for their invaluable comments and feedback.

Special thanks to Prof. Abdelouahed Gherbi for his support and insightful advice on my research career. I would like to thank Prof. Armin Zimmermann for his technical insights on TimeNET. Many thanks to my previous supervisors, Prof. Ayman Youssef, Prof. Youssef Harkouss, and Mohamad Youssef, for their continual encouragement.

A very special word of gratitude to the administrative staff at Western Engineering, in particular, Stephanie Tigert and Chris Marriott.

A good support is a key to survive "Azkaban" (Grad school), and I was lucky to have wonderful friends along the way. Special thanks to my friend and labmate, Elena Uchiteleva. During Ph.D., there is always stressing moments and obstacles that get in the way, but that is where you always have that person as Elena who is there to support you, believe in you, and show you that it is all worth it. Many thanks also to Amani Dahab, Amna Zeid, Abdelrahmein Zeid, and Sami Fayoumi, my amazing friends and second family. Thanks Amani for your continual support, warmth words, and delicious food! My friends, Farah Kazan, Zeinab McHeimech, Sara Mantach, Anas Ibrahim, Tarek Menkad, Widad Hanin, Jad Atwi, Fuad Shamieh, Ali El Takch, and Hoda Sbeity, thanks for being very supportive and caring.

I would like to extend my sincerest thanks to the wonderful IEEE London team and particularly the IEEE London Women In Engineering Group. Volunteering with you is a life-changing experience. I am very thankful to have the opportunity to meet and work with remarkable people like you. Special thanks to the outstanding volunteer, Murray MacDonald. I have never met someone who is so much dedicated to his volunteer work as Murray. He is an amazing and supportive person. Elena Uchiteleva, Maike Luiken, Ana Luisa Trejos, Rebecca Jevnikar, Hassan Hawilo, Joanne Moniz, Iram Raza, Anas Ibrahim, Wafaa Anani, and Lotfieh Albarazi: you are the best team! I have learned from you how giving back to the community is so much rewarding and fun.

I would like to extend special thanks to my Western friends and labmates for their motivating and thoughtful words they gave throughout this journey. I am grateful to meet supportive peers like you; Alexandra L'Heureux, Philip Kurowski, Mohamed Abu Sharkh, Karim Hammad, Brad de Vlugt, Mohamed Kalil, Anas Saci, Mohamed Noor, Trevor Crawford, Khalim Amjad Meerja, Mohamed Hussein, Khaled Al Hazmi, Abdallah Moubayed, Emad Aqili, Sara Zimmo, Maysam Mirahmdi, Aidin Reyhani-Masouleh, Taranpreet Singh, and Abdulfattah Noorwali.

I would never have completed this thesis without the assistance of my amazing family and in-laws; in particular, Aunt Leila, Maryam, Gaby, and Aunt Randa. Thank you for your love, encouragement, and support in all my pursuits. The big thank is to my cousin, sister, best friend, and inner voice, Mira Kamal. Words cannot express how grateful I am for having you. You have been with me throughout every step of my life. Thanks for always believing in me, cheering me up, and being there for me when I most needed you. I would have never made it through my thesis without your energetic messages, inspirational quotes, funny jokes, and best recipes (although I did not have the chance to try them all)! "You are my Nemo. If you ever get lost in the big ocean, I will find you".

I finish my acknowledgment with my basic source of energy: my beloved husband, Hassan Hawilo. I have spent some time to figure out how to extend my heartfelt thanks to you. Hassan has been living every single minute of my Ph.D., and without whom, I would not have had the strength to decide on undertaking this journey. It is your support and encouragement to follow my dreams that made me whom I am today and inspired me when I got weary. Thank you for supporting in the first place my decision to travel and for everything you have done to make my dream comes true. Your unconditional love, care, and patience throughout my lengthy working hours have allowed me to complete this thesis. Yet, during this period; we had the chance to spend every moment of every day for the last four years not only in the same apartment but also in the same lab insomuch that now we had our own language. Thank you for the sleepless nights and for all the times we spent discussing universe wonders and arguing about different existence and behavioral theories. Above all, thanks for being my best friend, gut-knowing, listener, financial manager, best comedian, for keeping things going smoothly, and always showing how proud you are of me! I undoubtedly would not have done this without you.

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC-STPGP 447230) and Ericsson Research.

On a final note, "Happiness can be found even in the darkest of times if only one remembers to turn on the light" Albus Dumbledore.

I am grateful to all of you for making this journey memorable!

Table of Contents

List of Tables

List of Figures

Acronyms

AaaS	*Application as a Service*
AAD	*Adaptive Anomaly Detection*
ACE	*Availability-aware CloudSim Extension*
AFT	*Adaptive Fault Tolerance*
AL	*Allowed Workload*
API	*Application Programming Interface*
App	*Application*
AWS	*Amazon Web Services*
AZ	*Availability Zone*
BPEL	*Business Process Execution Language*
BPMN	*Business Process Model and Notation*
BSP	*Bulk Synchronous Parallel*
CAMP	*Cloud Application Management for Platforms*
CAPEX	*Capital Expenditure*
CBA	*Component-based Architecture*
CC	*Cloud Computing*
CDN	*Content Delivery Network*
CDO	*Cloud Deployment Option*
CHASE	*Component High Availability Scheduler*
CL	*Current Load*
CLI	*Command-Line Interface*
CMS	*Cloud Management System*
COP21	*21st Conference of the Parties*
CORBA	*Common Object Request Broker Architecture*
COTS	*Commercial off The Shelf*
CoW-B	*Copy-on-Write-Basic*

CRM	*Customer Relationship Management*
DB	*Database*
DC	*Data Center*
DCN	*Data Center Network*
DCOM	*Distributed Component Object Model*
DeC	*Dependent Component*
DES	*Discrete Event Simulator*
DHCP	*Dynamic Host Configuration Protocol*
DOM	*Document Object Model*
DSL	*Domain Specific Language*
DSPN	*Deterministic Stochastic Petri Net*
EC2	*Elastic Compute Cloud*
ECE	*Energy and Carbon-Efficient*
EMF	*Eclipse Modeling Framework*
FCFS	*First Come First Serve*
FE	*Front End*
FFA	*Functional Failure Analysis*
GDM	*Graphical Definition Model*
GEF	*Graphical Editing Framework*
GHG	*Greenhouse Gas*
GITS	*Generic Input Template for Cloud Simulators*
GMF	*Graphical Modeling Framework*
GSPN	*Generalized Stochastic Petri Net*
GUI	*Graphical User Interface*
HA	*High Availability*
HIPAA	*Health Insurance Portability and Accountability Act*
HOT	*Heat Orchestration Template*
HTTPS	*Hypertext Transfer Protocol Secure*
IaaS	*Infrastructure as a Service*
ICT	*Information and Communications Technology*
IFT	*Intermediate Data Fault Tolerant*
IoT	*Internet of Things*

IT	*Information Technology*
JAR	*Java Archive*
JSON	*JavaScript Object Notation*
LoB	*Lines Of Business*
M2M	*Model-to-Model Transformation*
M2T	*Model-to-Text Transformation*
MILP	*Mixed Integer Linear Programming*
MTBF	*Mean Time Between Failure*
MTTF	*Mean Time To Failure*
MTTR	*Mean Time To Repair*
MVC	*Model-View-Controller*
NFV	*Network Function Virtualization*
NP	*Non-deterministic Polynomial Time*
NRDC	*Natural Resources Defense Council*
OASIS	*Organization for the Advancement of Structured Information Standards*
OL	*Overload Factor*
OLB	*Opportunistic Load Balancing*
OPEX	*Operational Expenditure*
OS	*Operating System*
PaaS	*Platform as a Service*
PHA	*Preliminary Hazard Analysis*
PM	*Physical Machine*
PNs	*Petri Net*
PUE	*Power Usage Effectiveness*
QoE	*Quality of Service Experience*
QoS	*Quality of Service*
RAM	*Random-access memory*
RAS	*Redundancy-Agnostic Scheduler*
RBD	*Reliability Block Diagram*
RDS	*Relational Database Service*
ROI	*Return On Investment*
RU	*Relative Average Utilization*

SaaS	*Software as a Service*
SC	*Sponsor Component*
SCPN	*Stochastic Colored Petri Net*
SDN	*Software Defined Networking*
SLA	*Service Level Agreement*
SMI	*Service Management Index*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access Protocol*
SPN	*Stochastic Petri Net*
SPOF	*Single Point Of Failure*
SRN	*Stochastic Reward Net*
SSaaS	*Simulating a Software as a Service*
SUMO	*Simulation of Urban Mobility*
TCO	*Total Cost of Ownership*
TDM	*Tooling Definition Model*
TOSCA	*Topology and Orchestration Specification for Cloud Applications*
UML	*Unified Modelling Language*
VE	*Virtual Environment*
VM	*Virtual Machine*
WSDL	*Web Services Description Language*
XaaS	*Everything as a Service*
XML	*Extensible Markup Language*
YAML	*Yet Another Markup Language*

Chapter 1
Introduction

Today, cloud computing is one of the groundbreaking technologies that have transformed the landscape to support new businesses with dynamic and changing workforce and demands. It is becoming the lifeblood of most telecommunication network services and information technology (IT) software applications [1] [2]. With the development of the cloud market, it can be seen as an opportunity for information and communications technology (ICT) companies to deliver communication and IT services over any fixed or mobile network with high performance and secure end-to-end quality of service (QoS) for end users [3]. Although cloud computing provides benefits to different players in its ecosystem and makes services available anytime, anywhere and in any context, many concerns arise regarding the performance and quality of the services offered by the cloud. One major concern for the enterprises is the high availability (HA) of cloud-based applications where business is expected to be running in the occurrence of any disruptive incident or sudden failure. Since these applications are hosted by a virtual environment (virtual machines (VMs) or containers) residing on servers, their availability depends on that of the hosts [4] [5] [6]. When a hosting server fails, its VMs/containers and their applications become inoperative. The absence of application protection plan has a tremendous effect on business continuity and IT enterprises. Outages can happen even on the well-managed cloud platforms. For example, Amazon has faced an outage due to a "human typo" on Feb. 28, 2017, which has affected many services and websites including Quora, Sailthru newsletter provider, Business Insider, Slack filesharing, and various connected Internet of things (IoT) hardware [7]. Also, on Feb. 1, 2017, GitLab has reported data loss due to accident deletion, which has caused the permanent loss of "six hours' worth" of data [8]. With the growing reliance on the cloud services and data centers (DCs), it is necessary to understand the direct and indirect impact of outages on the enterprises. According to Aberdeen Group, the cost of one hour of downtime is $74,000 for small organizations and $1.1 million for larger ones [9]; excluding reputation damage that can be significantly greater in the longer term. At the same time, a Disaster Recovery Preparedness Council survey has released that 27% of the enterprises have obtained a disaster readiness passing state [10]. In addition, the Ponemon Institute study shows that the average cost of a DC outage has increased 38% since 2010 (from $505,502 in 2010 to $740,357 in 2016) [11]. It is not always easy to place a direct cost on downtime. Angry customers and bad publicity are all costly, but not directly measured in currency. This also includes:

- Damage to mission-critical applications, marketplace reputation, and other assets
- Loss of trustworthiness among stakeholders and customers
- Cost to repair affected operational processes that handle the "core business and value chain"
- Regulatory and legal effect, such as litigation cost

The number of active Facebook users has grown to 1.5 billion, which requires an immune system to ensure data delivery anytime and anywhere [11]. Besides, Business Insider (BI) Intelligence research determines that the number of globally shipped smartphones is expected to be above 1.5 billion [12]. Additionally, International Data Corporation (IDC) expects that the IoT market will reach $1.7 trillion in 2020 [13]. These developments and new technologies means more data and business opportunities are emerging. This will cause a spurt growth in the cloud, which is expected to play a critical role to handle these high data undertow, emerging technologies, and new market demands. However, the Ponemon study indicates that the downtime costs is still rising, and the outages reasons are similar to what they were few years ago. At the same time, many enterprises require continuous availability of their resources and services. The key solution to these issues is to develop a highly available system that protects services, avoids downtime, and maintains business continuity. Achieving this mandate involves designing and development of multiple management systems for high availability including redundancy, failover, and other HA practices. This thesis provides an approach that is immune to failure while satisfying other quality of service (QoS) requirements. It proposes a pragmatic methodology to address the high availability in the cloud. This methodology consists of deployment and design phases. Fig. 1.1 summarizes the thesis contributions. It starts with designing an HA-aware scheduler for cloud applications and associates it with a stochastic availability

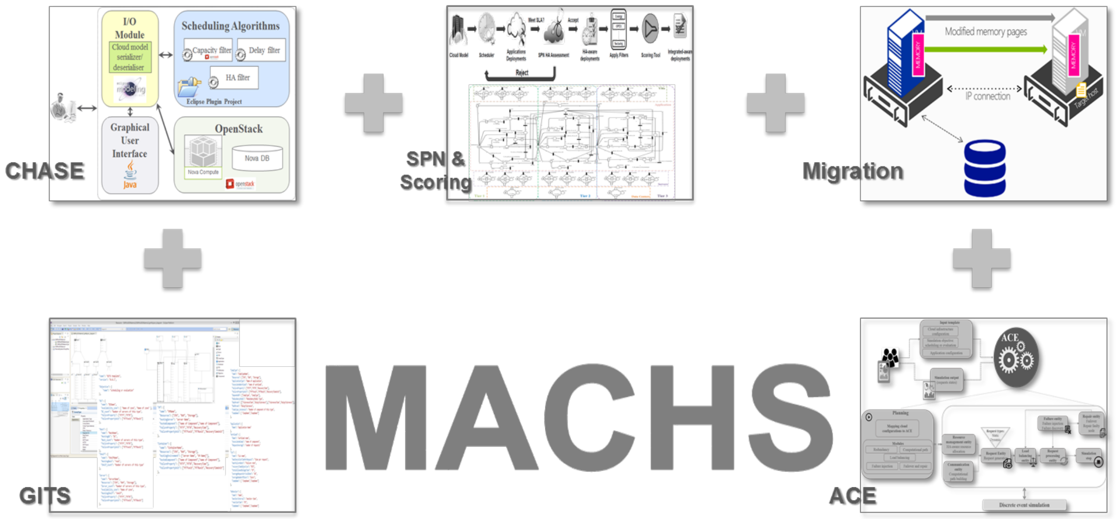


Figure 1.1: Overview of the thesis contributions.

model to assess different HA-aware deployments of applications' components. It then provides a cloud scoring selection solution that integrates the HA-aware approach with other challenges facing the cloud including energy and various performance measures. The thesis also provides a live VM migration as another HA-aware approach that can be triggered for load balancing, overload, or pre-disaster recovery objectives. At the design phase, this thesis alleviates the portability and orchestration challenges of cloud-based applications and provides a generic input template for cloud systems. It also proposes availability aware extension to CloudSim, well-defined cloud simulator, to mitigate the configuration settings challenges of modeling and simulating HA-aware mechanisms in real-cloud platforms.

1.1 Thesis Outline

This thesis is organized as follows. Chapter 2 provides a novel HA-aware scheduling technique, CHASE, that maximizes applications availability while satisfying different functionality constraints. This problem is formulated as a mixed integer linear programming (MILP) optimization model, which is associated with a heuristic solution, CHASE. The proposed approach envisions the cloud model as a cloud provider and user and integrates them through a virtualization layer. It also performs a criticality analysis to prioritize the deployment of mission-critical applications. To ensure HA-aware applications deployments, CHASE captures different redundancy and interdependency constraints and other HA measures. It is evaluated on a 3-tier web application and is designed to communication with the cloud management system, OpenStack. Chapter 3 defines the different types of cloud failures and provides an availability-centric analysis model using Stochastic Petri Net (SPN). The SPN model captures the different elements of the cloud model and their workflow. It then assesses different applications deployment in terms of HA while modeling the requests, load balancing, failures, failover, and interdependency practices. The proposed SPN model provides HA-aware guidelines for cloud scheduling solution. It is also associated with an extensible scoring selection system that integrates HA objectives with other performance and QoS norms in the cloud. In this chapter, the scoring system selects the optimal HA-aware deployment while satisfying green and cost objectives. Chapter 4 provides design considerations to implement an HA-aware deployment solution. It also provides a comparative study to select best availability assessment model. It then proposes a live VM migration approach to ensure data delivery upon unforeseen failures (i.e., natural disasters). The proposed migration approach is formulated as a MILP model that minimizes migration time and downtime. Chapter 5 provides GITS, a generic input template for different cloud simulators. GITS alleviates the portability and orchestration issues facing the cloud. For this purpose, the chapter models the cloud in terms of functionality measures and HA features and provides a JavaScript Object Notation (JSON) schema to facilitate the creation of cloud scenarios and experiments reusability. Chapter 6 provides ACE, availability-aware CloudSim extension. ACE extends CloudSim simulator to support HA measures and allows the modeling and evaluation of multiple HA mechanisms in a cloud-based environment. To that end, ACE provides a mapping of the JSON template to the CloudSim environment, supports dynamic request generations, injects failure, recovers/repairs them, and provides other HA-aware features.

1.2 Thesis Contributions

The major contributions of the thesis are summarized as follows.

1.2.1 Chapter 2 contributions

Cloud schedulers that are agnostic of the intricacies of the tenant's application may result in suboptimal placements. In these placements, redundant components may be placed too close to each other rendering their existence obsolete because a single failure can affect them all, or the delay constraints can be violated, hindering the application functionality. With this in mind, the main contributions of this chapter are the following:

1. This chapter proposes CHASE, a novel component's HA-aware scheduling technique, which maximizes the availability of applications without violating service level agreements (SLAs) with the end-users. For this purpose, the scheduling is formulated as a MILP model associated with a heuristic solution.
2. CHASE captures at an abstract level details of both the applications and the cloud infrastructure. When analyzing availability, the cloud topology should be defined to pinpoint the single points of failure (SPOF) or possible bottlenecks that might affect the data processing and availability. Using a unified modelling language (UML), CHASE starts by modeling the applications with their functional and non-functional requirements. Then it considers the cloud infrastructure model to be a constrained solution space where a mapping between applications, VMs, and servers are generated to maximize the availability. With this model, we start with a specific cloud infrastructure and a set of requested applications, and we end up by generating an interface between the provider and user side using VM mappings.
3. Using CHASE, prior criticality analysis is conducted on applications to differentiate between mission-critical and standard applications and consequently, schedule them based on their impact on the execution environment and business functionality.
4. CHASE implements different approaches that deploy redundant models and failover solutions. These practices are achieved through geographically distributed redundant applications' deployments without violating the interdependency requirements. This allows the elimination of single point of failure caused at the level of VM, cluster, or cloud.
5. CHASE overcomes the challenges of maintaining HA-aware application's deployment and compromises between different functionality, failover, affinity, and anti-affinity constraints affecting it.

6. 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 the OpenStack Nova scheduler complement with CHASE HA filters. The scheduling tool is composed of several complementary modules: I/O module, the graphical user interface (GUI) that is populated from an instance of the designed cloud application UML model, OpenStack Nova database, and the scheduler, CHASE.

1.2.2 Chapter 3 contributions

This chapter is divided to two sub-problems:

Dependability Analysis: It is not enough to provide an HA-aware solution that can mitigate failures and maintain certain availability baseline, but it is necessary to assess such solution and its resiliency to any failure modes. A formal and analytical stochastic model is needed for both the tenants and providers to quantify the expected availability offered by an application deployment. Therefore, the main contributions of this chapter are the following:

1. This chapter proposes a Stochastic Petri Net model (SPN) that captures the stochastic characteristics of the cloud services and translates them into elements of an availability model.
2. Using linear temporal logic, the SPN model captures the stochastic nature of failures according to different probability distribution functions.
3. The SPN model also captures the cloud elements (DCs, servers, and VMs/containers) and the correlation aspect of their failures.
4. The SPN model envisions the functional workflow between the components of multi-tiered applications (queuing and request forwarding) as well as the high availability mechanisms they employ (load balancing and redundancy schemes).
5. Finally, the model assesses and quantifies the expected availability of the cloud services and their deployments in geographically distributed DCs.

Performance-aware cloud deployments: Although the SPN model provides generic guidelines to maintain HA of cloud applications, there are still a few concerns with respect to the energy, cost, and other challenges associated with cloud. The environmental and cost impacts of running the applications in the cloud are an integral part of incorporated responsibility, where both the cloud providers and tenants intend to reduce. If multiple deployment options can satisfy the HA requirement, the question remains, how can we choose the deployment that satisfies the other providers and tenants requirements? For instance, choosing DCs with low carbon emissions can both reduce the environmental footprint and potentially earn carbon tax credits that lessen the operational cost. Therefore, the main contributions of this chapter are the following:

1. This chapter provides a solution that integrates the HA constraints with the other cloud challenges. It couples the above SPN model with a cloud scoring system that selects the optimal deployment according to predefined policies, such as lower operational expenditure (OPEX), low carbon footprint, and/or other norms.
2. The scoring policies requirements are integrated with functionality and availability constraints to select best placements of application components.
3. The scoring selection system envisions user needs and assesses DCs capabilities to weight the best HA-aware deployments and select the optimal ones accordingly.
4. The proposed scoring system is extensible and depends on the capabilities and preferences offered by the cloud providers such as green and cost criteria to evaluate cloud DC.

1.2.3 Chapter 4 contributions

The workload of the cloud-based might vacillate due to growth of its applications or variation in its resources' demands. This might generate hotspots that downgrade the QoS of the applications and affect the service level agreements with the clients. Therefore, the main contributions of this chapter are the following:

1. This chapter provides various guidelines to design an HA-aware solution for a cloud system starting with system modeling, followed with a deployment solution up to a dependability analysis model.
2. This chapter also proposes live migration approach, a different fault tolerant technique to ensure the delivery of services upon a sudden failure, a virtual machine (VM)/infrastructure overload, or maintenance.

3. The live migration approach is formulated as a MILP optimization model that provides a trade-off between the migration time and the downtime. It employs the iterative pre-copy mechanism to perform the VMs migration.
4. The migration approach minimizes the migration and downtime not only based on the number of dirtied pages in the iterative stage but also depending on an optimal placement of the virtual machines.

1.2.4 Chapter 5 contributions

The creation of cloud scenarios is time-consuming and requires error-prone programming efforts, and consequently, the scenarios will be non-reusable and only feasible for experienced programmers. Additionally, it is necessary to ensure that the applications, core of the cloud model, are well orchestrated to fully realize the cloud capabilities. With this in mind, the main contributions of this chapter are the following:

1. This chapter provides a generic input template for cloud simulators, GITS. The template captures the specification of complex application behavior, cloud infrastructure, HA measures, and other SLA requirements.
2. GITS facilitates the creation of cloud scenarios in different cloud simulators in general and CloudSim in particular.
3. GITS supports HA features including HA measures (failure, recovery, and repair times), redundancy models, failover practices, and different failure types.
4. GITS ensures simplicity, reusability, and portability through defining different modules of the template including a UML module that captures system requirements, a user Eclipse Graphical Modeling Framework (GMF) module for scenarios visualization, Parser module, and a human readable module that uses JavaScript Object Notation (JSON) templating.
5. GITS is an extensible template that can be easily modified to meet OpenStack Heat template, Extensible Markup Language (XML), or other cloud schema.

1.2.5 Chapter 6 contributions

In the interconnected globe where service delivery is the measure of a success, high availability is an indispensable area that cannot be negotiated for enterprises migrating to the cloud. HA is about building a resilient cloud system that can deliver continuous services and applications. To design an HA-aware solution, cloud systems must be designed to handle planned and unplanned outages of the cloud infrastructure and applications, whether it is a failure at the granularity of a single software instance or an entire data center. However, the platform configuration settings can have a significant impact on the effectiveness of the HA-aware approaches used to mitigate the impact of failures. To this end, simulation tools can be used to evaluate availability solutions and assess a cloud resiliency against failures. CloudSim is a well-known and highly utilized cloud simulator that enables seamless modeling, simulation, and experimenting of scheduling and allocation policies of large-scale cloud platforms. It allows designing and evaluating of new scheduling and allocation policies. However, CloudSim does not support HA properties, such as redundancy, failure/recovery rates, and HA-aware scheduling. With this in mind, the main contributions of this chapter are the following:

1. This chapter provides a modular availability-aware solution CloudSim extension, ACE.
2. ACE defines an HA-aware cloud architecture to capture the abstract level of the cloud model and its HA measures.
3. ACE implements a graphical modeling interface and a JavaScript Object Notation (JSON) template to ensure simplicity, repeatability, and reusability of cloud configurations and applications deployments.
4. ACE provides dynamic and static generation of requests and can determine the applications computational path and their protection group to ensure successful completion of request.
5. ACE embeds an HA-aware deployment solution that generates placement for cloud-based applications cloud applications while maximizing their availability and minimizing SLA violations.
6. ACE provides a fair load-balancing approach for requests distribution among the different applications tiers.
7. For availability assessment, ACE injects failure, supports failover to redundant components, and repairs faulty nodes to evaluate cloud system resiliency.
8. ACE provides the extensions needed to simulate the expected availability of an HA-aware deployment.

References

[1] M.A. Sharkh, M. Jammal, A. Shami, and A. Ouda, "Resource allocation in a network-based cloud computing environment: design challenges," *IEEE Communications Magazine,* vol. 51, no.11, pp. 46-52, November 2013.

[2] Microsoft, "The Economics of the Cloud," `http://www.microsoft.com/en-us/news/presskits/cloud/docs/the-economics-of-the-cloud.pdf`, November 2010. [September 14, 2014]

[3] ITU, "Cloud Computing Benefits from Telecommunication and ICT Perspectives," `http://www.itu.int/dms_pub/itu-t/opb/fg/T-FG-CLOUD-2012-P7-PDF-E.pdf`, February 2012. [September 14, 2014]

[4] L. Grit, D. Irwin, A. Yumerefendi, and A. Chase, "Virtual Machine Hosting for Networked Clusters: Building the Foundations for Autonomic Orchestration," *in 2nd International Workshop on Virtualization Technology in Distributed Computing,* 2006.

[5] D. Jayasinghe, C. Pu, *et al.*, "Improving Performance and Availability of Services Hosted on IaaS Clouds with Structural Constraint-Aware Virtual Machine Placement," *in IEEE International Conference on Services Computing (SCC),* pp. 72-79, July 4-9, 2011.

[6] J. Dean and L. Barroso, "The Tail at Scale," *Communications of the ACM,* vol. 56, no. 2, pp. 74-80, February 2013.

[7] Tech Crunch, "Amazon AWS S3 outage is breaking things for a lot of websites and apps," `https://techcrunch.com/2017/02/28/amazon-aws-s3-outage-is-breaking-things-for-a-lot-of-websites-and-apps/`, February 28, 2017. [March 4, 2017]

[8] WHIR Hosting Cloud, "GitLab's Not Alone: AWS, Google, and Other Clouds Can Lose Data, Too," `http://www.thewhir.com/web-hosting-news/gitlabs-`

not-alone-aws-google-and-other-clouds-can-lose-data-too?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+thewhir+%28theWHIR.com+-+Daily+Web+Hosting+News%2C+Features%2C+Blogs+and+more%29, February 8, 2017. [February 15, 2017]

[9] Aberdeen Group, "Why Mid-Sized Enterprises Should Consider Using Disaster Recovery-as-a-Service," http://www.aberdeen.com/Aberdeen-Library/7873/AI-disaster-recovery-downtime.aspx, April 2012. [August 25, 2016]

[10] Disaster Recovery Preparedness Council, "The State of Global Disaster Recovery Preparedness," Annual Report, https://drbenchmark.org/wp-content/uploads/2014/02/ANNUAL_REPORT-DRPBenchmark_Survey_Results_2014_report.pdf, 2014.

[11] Ponemon Institute, "Cost of Data Center Outages," http://files.server-rack-online.com/2016-Cost-of-Data-Center-Outages.pdf, January 2016. [September 9, 2016]

[12] Business Insider, "This year's smartphone shipments might be worse than we previously thought," http://www.businessinsider.com/idc-lowers-smartphone-shipment-predictions-again-2016-9, September 2016. [January 8, 2017]

[13] The Wall Street Journal, "Internet of Things Market to Reach $1.7 Trillion by 2020: IDC," http://blogs.wsj.com/cio/2015/06/02/internet-of-things-market-to-reach-1-7-trillion-by-2020-idc/, June 2015. [December 15, 2016

Chapter 2
CHASE: Component High Availability-Aware Scheduler in Cloud Computing Environment

2.1 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 [3] [4]. 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 [5]. 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 [6]. According to [7], 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

The content of this chapter have been published [1] and [2].

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC-STPGP 447230) and Ericsson Research.

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.

This work aims to demonstrate the effect of application's placement strategy on the HA of the services provided by the virtualized cloud to its end users. To attain this objective, the cloud provider and user are modelled as a unified modeling language (UML) class diagram [1] [2]. This work puts the cloud UML 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. Also, we propose a novel scheduling technique that looks into the applications' criticality, interdependencies, and redundancies between application's components, their failure scopes, their communication delay tolerance, and resource utilization requirements. The technique examines not only mean time to failure (MTTF) to measure the component downtime and consequently its availability, but the analysis is based on the mean time to repair (MTTR), recovery, and outage tolerance times as well. To this end, a mixed integer linear programming (MILP) model is developed as an optimal solution for components' scheduling in small-scale network [1]. For large-scale systems, the MILP model is associated with an HA-aware scheduler for applications' components, CHASE, as the heuristic solution [2].

The HA-aware scheduler is compared to the MILP model and OpenStack Nova scheduler in a small data center network [1] [2]. 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.

The main contributions of this work are to:

- Capture all the functionality and availability constraints that affect application's placement.
- Reflect availability constraints not only by the failure rates of application's components and scheduled servers, but also by functionality requirements, which generate anti-location and co-location constraints.

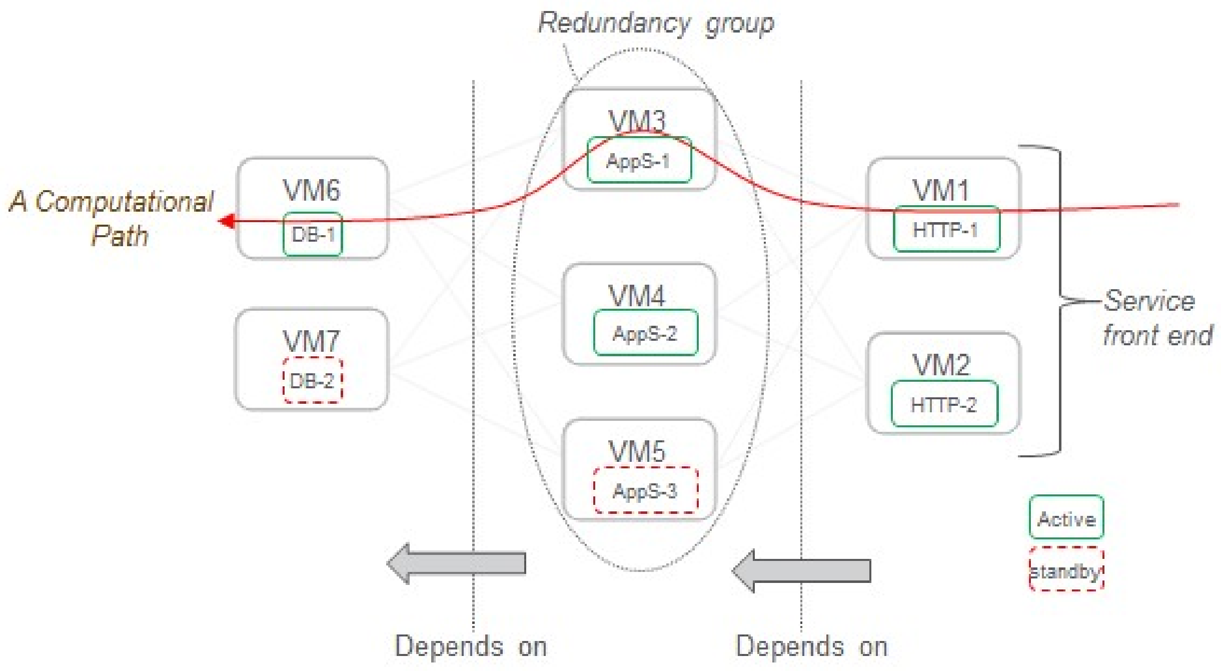


Figure 2.1: Example of an application deployment in the cloud.

- Consider various interdependencies and redundancies among application's components.
- Examine multiple failure scopes that might affect the component itself, its execution environment, and its dependent components.

This chapter is organized as follows. Section 2.2 describes the cloud UML model. Section 2.3 defines the HA-aware deployment problem and the proposed solution. Section 2.4 and Section 2.5 describe the simulation environment and the evaluation results of the MILP model and CHASE. CHASE-OpenStack implementation is discussed in Section 2.6. Finally, the related work and conclusion are presented in Section 2.7 and Section 2.8.

2.2 System Modelling and Schematization

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

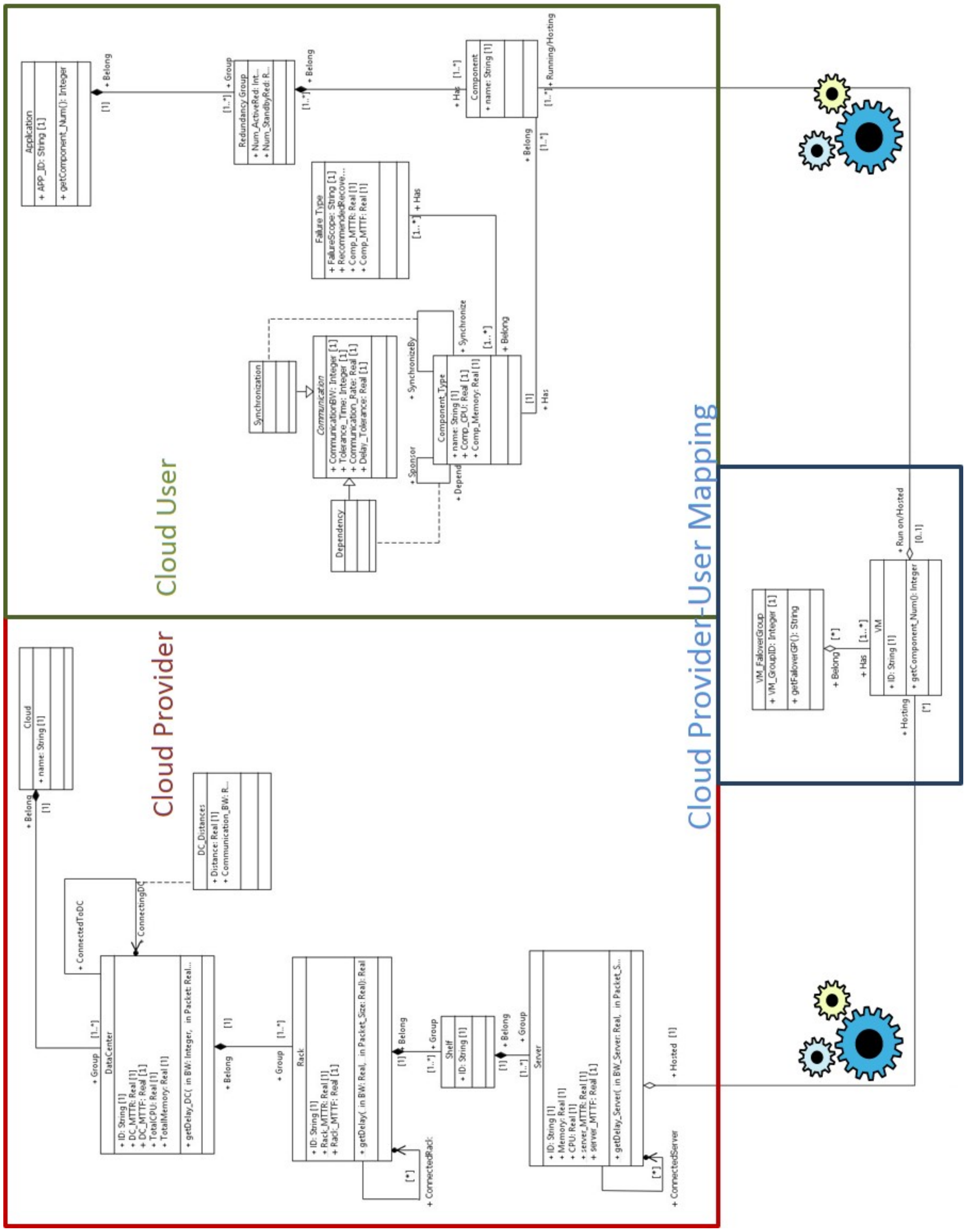


Figure 2.2: UML class diagram of CHASE cloud model.

year [8]. Moreover, the cloud provider is not responsible for the monetary 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 HTTPS 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. 2.1 illustrates a potential HA-aware deployment of our application example. At the front-end, multiple active (stateless) HTTPS 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 HTTPS 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. It is necessary to note that the architecture of the web-application's components (i.e. the number of tiers and the inter-dependency between the application's components) is defined before triggering CHASE.

The cloud can be modeled in terms of the cloud tenant's applications and the cloud provider 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 class diagram shown in Fig. 2.2. 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. It is necessary to note that different Eclipse integration plugins can be used to integrate the UML model with the generated code and enable round-tripping engineering [9] [10].

2.2.1 Cloud infrastructure model

The proposed cloud architecture is captured in the UML class diagram. At the root level, the cloud consists of data center networks distributed across various geographical areas. Each data center consists of multiple racks communicating through aggregated switches. Each rack has a set of shelves housing a large number of servers, which can have different capacities and failure rates. Servers residing on the same rack are connected with each other through the same network device (the top of the rack switch). Finally, the VMs are hosted on the servers. This tree structure determines the network delay constraints, and consequently, the delay between the communicating applications. This architecture divides the cloud into five different latency zones, which will be further discussed in Section 2.3.

In the proposed tree structure, each node *i* has its own failure rate (λ) and MTTR. The MTTF and MTTR parameters divide the intra- and inter- data center networks into availability zones. We are assuming that the availability *avail* of the host *h* depends not only on its λ and MTTR, but on that of corresponding DC and rack *R* as well. Thus, a request is successfully processed if the corresponding component, its host, and its parent rack and DC are all healthy. With this in mind, the cloud infrastructure is considered a series system [11]. In the case of failure (i.e. natural disaster as the worst case scenario) and to minimize the number of false negatives, it is assumed that the cloud infrastructure can resume its normal activity after the execution of the repair mechanism(s) of the faulty node(s) (i.e. after passing of the MTTR(s)). Thus, each host *h* can be seen as *(DC, R, S)* and is associated with a weight parameter, *avail*. In this scenario, it is assumed that a DC repair mechanism happens gradually; first, the DC is repaired followed by the recovery of its racks and then its corresponding servers. With this assumption, the false negatives (i.e. a node is considered healthy when it is actually a faulty one) are avoided. To this end, the host with the highest *avail* is selected as a candidate placement for application's component(s). The *avail* is calculated as follows:

$$avail_h = \frac{MTTF_h}{MTTF_h + MTTR_h} \tag{1}$$

$$where \begin{cases} MTTF_h = \frac{1}{\lambda_{DC}+\lambda_R+\lambda_s} \\ \\ MTTR_h = MTTR_{DC} + MTTR_R + MTTR_S \end{cases}$$

It is necessary to note that the *avail* parameter is only used to differentiate between different hosts and prune the ones with the low MTTF and high MTTR values.

2.2.2 Cloud application model

Applications are typically developed using a component based architecture where each application is made up of one or more components. The application combines its components' functionalities to provide a higher level of service [12] [13]. To maintain availability requirements, each component can have one or more redundant components. The primary component and its redundant ones are grouped into a dynamic redundancy group. In this group, each component is assigned a specific number of active and standby redundant components. As shown in the UML model, each redundancy group is assigned to at most one application, which consists of at least one redundancy group.

As for the component, it belongs to one component type. A component type represents an executable software deployment. From this perspective, the component represents a running instance of the component type. Components of the same type have the same attributes defined in the component type class, such as computational resources (CPU and memory) attributes.

Each component can be configured to depend on other components. The dependency relation is captured at the type level and can be configured using the delay tolerance, outage tolerance, and communication bandwidth attributes. The delay tolerance determines the minimum required latency to maintain communication between sponsor and dependent components. As for the outage tolerance or tolerance time, it is the time that the dependent component can tolerate without the sponsor one. The same association is used to describe

the relation between redundant components that need to synchronize their states.
Finally, each component type is associated with at least one failure type. The list of failure types determines the failure scope of each component type, its MTTF, MTTR, and recommended recovery.

2.2.3 Cloud-Application integration

Each component of the application model is scheduled on a server in the cloud provider model using VM mappings. Each VM can be hosted on one server and can have at least one component instance running in it. Sudden failure events can occur to cloud-application such as natural disaster, network or runtime failures [14]. In order to deal with these events, the inoperative VMs are switched off, and a failover group takes over the control. The failover group consists of at least one VM, which is a redundant VM of the inoperative one.
As mentioned earlier, the proposed HA-aware scheduling technique searches for the optimum physical server to host the requested component. Whenever a server is scheduled, a VM is mapped to the corresponding component and to the chosen server. Therefore, a component can reside on that VM.

2.3 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. The HA-aware scheduling technique performs a criticality analysis to start scheduling the 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.
The proposed technique provides an efficient and highly available allocation by satisfying the following constraints:

1) *Capacity Constraints:* These are functional constraints, which are satisfied by searching for servers that meet the resource needs of each application. In the proposed model, the computational resources consist of the CPU and memory.

2) Network Delay Constraints: Using these constraints, another list of servers is generated. These servers satisfy the latency requirements to avoid service degradation between communicating applications. It is assumed that the delay requirements are divided into five delay types (i.e., latency zones) as follows:

a) D_0 *Type:* Requires that all communicating components should be hosted on the same VM and consequently on the same server.

b) D_1 *Type:* Requires that all the communicating components should be hosted on the same server.

c) D_2 *Type:* Requires that all the communicating components should be hosted on the same rack.

d) D_3 *Type:* Requires that all the communicating components should be hosted on the same DC.

e) D_4 *Type:* Requires that all the communicating components can be hosted across different data centers but must be within the same cloud.

3) Availability Constraints: These constraints prune the candidate servers generated by the capacity and delay constraints to select the ones that maintain a high level of application's availability. In order to maximize the HA of an application, three sub-constraints should be satisfied:

a) Failure Rate Constraint: It determines that the selected server should maximize component's availability. In order to satisfy this constraint, the model searches for the server with maximum MTTF and minimum MTTR. Whenever it is found, then the $MTTF_c^A$ of the component *C* after being hosted can be calculated as follows:

$$MTTF_c^A = \frac{1}{\lambda_c + \lambda_h} \tag{2}$$

b) Dependency Constraint: This constraint is divided into two sub-constraints: *Co-location Constraint:* This is valid whenever the tolerance time of the dependent component is lower than the recovery time of its sponsor. When the dependent component cannot tolerate the absence of its sponsor, then the failure of its server, its sponsor or sponsor's server affects it. In order to minimize its failure rate, both dependent and sponsor should share same server.

Anti-location Constraint:: It requires that the dependent component and its sponsor should be placed on different servers. This is valid whenever the tolerance time of the dependent component is greater than the recovery time of its sponsor. By considering this case, the MTTF of the application will be maximized because of its inverse proportionality relation with λ.

c) Redundancy Constraint: It basically prevents redundant components of a primary one from residing on the same server and requires that they should be placed far away from each other as the delay constraints allow.

With these constraints, we develop a MILP model and CHASE that minimize components' downtime while finding the optimal physical server to host them.

2.3.1 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 [15]. 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. 2.1, 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 (failure rate) and MTTR, and therefore its failure can cause either an outage *(o)* of the application or a degradation *(d)* of the service. Let N_{fail} be the failure occurrence. The criticality value of a component is the product of the component's unreliability and the number of occurrence of component's failure [16]. The criticality 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 (3). If a dependent component *(DeC)* can tolerate the outage, *(OT)*, of its sponsor *(SC)* until its recovery, then the failure of *SC* causes service degradation as shown in (4). Conversely, the failure of the sponsor causes not only

Notation	Significance	Representation
R	Resource Type: CPU or memory	Number of Cores and MB of RAM
$RED_{cc'}$	Redundancy matrix of C and C'	{0,1}
$DEP_{cc'}$	Dependency matrix of C and C'	{0,1}
$DEL_{ss'}$	Delay between S and S'	second
OT_c	Outage tolerance of C	hour
RT_c	Recovery time of C	hour
DT_c	Delay tolerance of C	hour

Table 2.1: Variable notations.

a service degradation but also an outage as expressed in (5).

$$criticality_{FE} = (N_{fail} \times MTTR)_o \tag{3}$$

$$criticality_d = (N_{fail} \times MTTR)_d \tag{4}$$

$$criticality_{do} = \sum_{DeC} (Degradation + Outage) \tag{5}$$

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

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.

2.3.2 Mathematical formulation

This section introduces a MILP model to solve the HA-aware placement problem. The proposed MILP model was solved using the IBM ILOG CPLEX optimization solver.

2.3.2.1 Notations

Various parameters were used to solve the placement problem and develop the MILP model.

a) Input Parameters

Let a virtual machine be denoted as *V* and a server as *S*. Each VM consists of an application $\{A\}$, which consists of a specific number of components $\{C\}$, which are of component types $\{CT\}$. Therefore, each application is a set of *C* and *CT* and can be denoted as *A* =$\{$*C*, *CT*$\}$. This notation ensures that whenever a set of components *C* of types *CT* is scheduled, its corresponding application is considered hosted. As for the computational resources, L_{cr} and L^T_{sr} denote the set of resources, which can be memory or CPU of component and server respectively. Table 2.3.2.1 shows the various parameters notations used in the MILP model.

b) Decision Variables

The decision variables are defined as follow:

$$X_{cs} = \begin{cases} 1 & if\ S\ host\ C \\ 0 & otherwise \end{cases} \tag{6}$$

$$z_c \;=\; \begin{cases} 1 & if\ DEL_{ss'} \leq DT_c \\ 0 & otherwise \end{cases} \tag{7}$$

2.3.2.2 MILP model

The downtime represents a duration during which a system is unavailable or fails to function. In this chapter, the system can be a component or a host. However, the downtime of *C* does not only depend on the component itself *(*$Downtime_c$*)*, but on its hosting server *(*$Downtime_s$*)* as well. In order to minimize the overall downtime of *C*, the objective function of the formulated MILP model should minimize $Downtime_c$ and $Downtime_s$. The objective function and its constraints are formulated as follows:

Objective Function:

$$\min \quad \sum_c \sum_s (Downtime_c + Downtime_s) \times X_{cs}$$

Subject to:

Capacity Constraints:

$$\sum_c (X_{cs} \times L_{cr}) \leq L^T_{sr} \qquad \forall\, s,\ r \tag{8}$$

$$\sum_s X_{cs} = \quad 1 \qquad\qquad \forall\, c \qquad\qquad (9)$$

$$X_{cs} \quad \in \quad \{0,1\} \qquad\qquad \forall\, c, s \qquad\qquad (10)$$

Network Delay Constraints:

$$(X_{c's'} \times DEL_{ss'} - DT_c) \leq M \times z_{c'} \quad \forall\, c, c',\ s, s' \qquad (11)$$

$$X_{cs} - 1 \leq M \times (1 - z_{c'}) \qquad \forall\, c, c',\ s \qquad (12)$$

$$z_{c'} \quad \in \quad \{0,1\} \qquad \forall\, c' \qquad (13)$$

Redundancy Constraint:

$$X_{cs} + X_{c's} \leq 1 \qquad \forall\, c, c', s,\ RED_{cc'} \qquad (14)$$

Dependency Co-location Constraint:

$$X_{cs} + X_{c's} \leq 2 \qquad \forall\, c, c', s,\ DEP_{cc'} \qquad (15)$$

Dependency Anti-location Constraint:

$$X_{cs} + X_{c's} \leq 1 \qquad \forall\, c, c', s,\ DEP_{cc'} \qquad (16)$$

Boundary Constraint:

$$Downtime_c, Downtime_s \geq 0 \quad \forall\, c, s \qquad (17)$$

As discussed earlier, the HA-aware placement of the application is affected by capacity, delay and availability constraints. Regarding capacity constraints, constraint (8) ensures that the requested component's resources must not exceed the available resources of the selected destination server. Constraint (9) determines that the component can be placed on at most one physical server. Constraint (10) ensures that the decision variable *(X_{cs})* is a binary integer. The delay constraints (11), (12), and (13) ensure that communicating components will be placed on a server that satisfies the required latency. These constraints are applied on the dependency and redundancy communication relations between scheduled components.

The availability constraint (14) reflects the anti-location constraint between a component and its redundant ones. Using constraint (15), the dependent components should share the same server in case their outage tolerance is smaller than the recovery time of their sponsor component. The anti-location constraint between dependent and sponsor components is

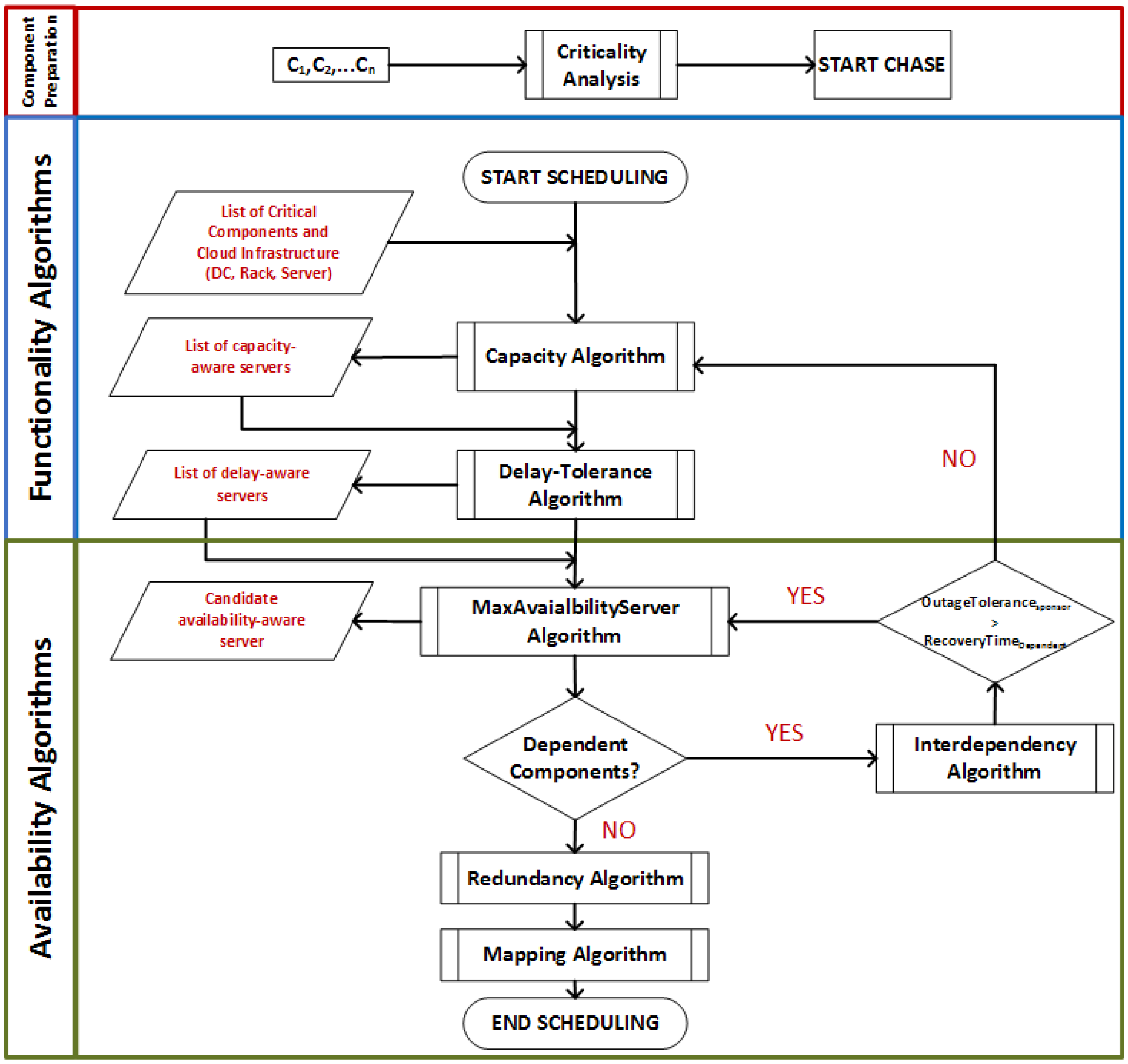

Component Preparation
$C_1, C_2, \ldots C_n$
Criticality Analysis
START CHASE
Functionality Algorithms
START SCHEDULING
List of Critical Components and Cloud Infrastructure (DC, Rack, Server)
Capacity Algorithm
List of capacity-aware servers
Delay-Tolerance Algorithm
List of delay-aware servers
NO
Availability Algorithms
MaxAvailabilityServer Algorithm
Candidate availability-aware server
YES
$OutageTolerance_{sponsor} > RecoveryTime_{Dependent}$
Dependent Components?
YES
Interdependency Algorithm
NO
Redundancy Algorithm
Mapping Algorithm
END SCHEDULING

Figure 2.3: Flowchart of CHASE approach.

active in the contrary case as shown in (16). The boundary constraint (17) specifies real positive values for downtimes of C and S.

2.3.3 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. 2.3. Each sub-algorithm deals with a specific set of constraints such as capacity, delay, and availability constraints.

```
Algorithm 1 Capacity Algorithm
INPUT: $A = (A_1, A_2, ..., A_p)$
       $C = (C_1, C_2, ..., C_n)$
       $S = (S_1, S_2, ..., S_m)$
OUTPUT: $S^{cap} = (S_1, S_2, ..., S_k)$
       where $S^{cap} \subset S$
1: $delay = GenerateDelayType()$
2: begin:
3: if $delay = D_0$ OR $delay = D_1$ then
4:     for $a_i \in A$ do
5:         $find = FALSE$
6:         $AppCPU_{a_i} = \sum_{c_{a_i}} CPU_{c_{a_i}}$
7:         $AppMemory_{a_i} = \sum_{c_{a_i}} Memory_{c_{a_i}}$
8:         for $s_i \in S$ do
9:             if $AppCPU_{a_i} < CPU_{s_i}$ AND $AppMemory_{a_i} < Memory_{s_i}$ then
10:                $S^{cap}.add(s_i)$
11:                $find = TRUE$
12:            end if
13:        end for
14:        if $find = FALSE$ then
15:            Let M be the mapping between C and CP
16:            $M = doComputationalPathAnalysis(a_i)$
17:            for $CP_i \in M$ do
18:                $S^{cap} \leftarrow GenerateCandidate_{CP_i}$
19:            end for
20:            $find = TRUE$
21:        end if
22:    end for
23: else if $delay = D_2$ OR $delay = D_3$ OR $delay = D_4$ then
24:    for $a_i \in A$ do
25:        for $c_i \in C$ do
26:            for $s_i \in S$ do
27:                if $CPU_{c_i} < CPU_{s_i}$ AND $AppMemory_{c_i} < Memory_{s_i}$ then
28:                    $S^{cap}.add(s_i)$
29:                end if
30:            end for
31:        end for
32:    end for
33: end if
34: end
```

Figure 2.4: Capacity algorithm of CHASE.

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. Fig. 2.4 describes the capacity algorithm.

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

```
Algorithm 2 Delay Tolerance Algorithm
INPUT: $S^{cap} = (S_1, S_2, ..., S_k)$
       $R = (R_1, R_2, ..., R_q)$
       $DC = (DC_1, DC_2, ..., DC_b)$
OUTPUT: $S^{delay} = (S_1, S_2, ..., S_t)$
       where $S^{delay} \subset S^{cap}$
begin:
if $delay = D_2$ then
    for $r_i \in R$ do
        for $s_i \in S^{cap}$ do
            Let $RG$ be the mapping between $S$ and $Rack$
            $S^{delay} = RackGroup(s_i)$
        end for
        $RG_r \leftarrow (S^{delay}, r)$
    end for
else if $delay = D_3$ then
    for $dc_i \in DC$ do
        for $s_i \in S^{cap}$ do
            Let $DG$ be the mapping between $S$ and $DC$
            $S^{delay} = DataCenterGroup(s_i)$
        end for
        $DG_{dc} \leftarrow (S^{delay}, dc)$
    end for
end if
end
```

Figure 2.5: Delay Tolerance algorithm of CHASE.

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. Fig. 2.5 depicts the delay tolerance algorithm.

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. 2.6. 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. Fig. 2.7 depicts the inter-dependency algorithm. 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

Algorithm 3 Availability Algorithm

```
INPUT: $A = (A_1, A_2, ..., A_p)$
       $C = (C_1, C_2, ..., C_n)$
       $S^{cap} = (S_1, S_2, ..., S_k)$,   $S^{delay} = (S_1, S_2, ..., S_t)$
OUTPUT: $S^{avail}$
       $where\ S^{avail} \subset S^{delay}$
1: begin:
2: $A = AppCriticalityRank(A)$
3: $C = ComponentCriticalityRank(A)$
4: for $a_i \in A$ do
5:     if $delay = D_0$ OR $delay = D_1$ then
6:         if $S^{cap}_{a_i}.size()isNotNULL$ then
7:             $S^{avail}_{a_i} = MaxAvailability(S^{cap}_{a_i})$
8:             $UpdateServerCapcity(S^{avail}_{a_i})$
9:         else
10:            for $cp_i \in M$ do
11:                $S^{avail}_{cp_i} = MaxAvailability(S^{cap}_{cp_i})$
12:                $UpdateComputationalPathServer(S^{avail}_{cp_i})$
13:                $UpdateServerCapcity(S^{avail}_{cp_i})$
14:            end for
15:        end if
16:    else
17:        for $c_i \in C$ do
18:            $Comp \leftarrow c_i$
19:            if $c_i.isScheduled = FALSE$ then
20:                $S^{avail}_{c_i} = MaxAvailability(S^{cap}_{c_i})$
21:                $AllocatedServer.add(S^{avail}_{c_i})$
22:                $UpdateServerCapcity(S^{avail}_{c_i})$
23:                $Comp_i.isScheduled = TRUE$
24:                if $delay = D_2$ then
25:                    $RackId = S^{avail}_{c_i}.getRackId()$
26:                    $DcId = NULL$
27:                else if $delay = D_3$ then
28:                    $RackId = NULL$
29:                    $DcId = S^{avail}_{c_i}.getDcId()$
30:                else
31:                    $RackId = NULL$
32:                    $DcId = NULL$
33:                end if
34:                $DependentScheduling(Comp, RackId, DcId, AllocatedServer)$
35:                $RedundantScheduling(Comp, RackId, DcId, AllocatedServer)$
36:            end if
37:        end for
38:    end if
39: end for
40: end
```

Figure 2.6: Availability algorithm of CHASE.

of components. Then this set is fed into the MaxAvailabilityServer algorithm to select 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. Fig. 2.8 describes the redundancy algorithm.

At this stage, the algorithm has found a host for each component. However, a mapping

```
Algorithm 4 Inter-dependecy Scheduling Algorithm
INPUT: $Comp, AllocatedServer$
OUTPUT: $S^{avail}$
        $where\ S^{avail} \subset S^{delay}$
1:  begin:
2:  for $d \in Comp.Dependent$ do
3:      if $d.isScheduled = FALSE$ then
4:          if $OT_d > RT_comp$ then
5:              $S_d^{delay} = UpdateComponentServer(AllocatedServer)$
6:              if $delay = D_2$ then
7:                  $S_d^{avail} = D_2.MaxMTTF(S_d^{delay}, RackId)$
8:              else if $delay = D_3$ then
9:                  $S_d^{avail} = D_3.MaxMTTF(S_d^{delay}, DcId)$
10:             else if $delay = D_4$ then
11:                 $S_d^{avail} = MaxMTTF(S_d^{delay})$
12:                 $AllocatedServer.add(S_d^{avail})$
13:                 $d.isScheduled = TRUE$
14:             end if
15:         else
16:             if $CPU_d < CPU_{S_{comp}^{avail}}$ then
17:                 $S_d^{avail} \leftarrow S_{comp}^{avail}$
18:                 $d.isScheduled = TRUE$
19:             else
20:                 $S_{comp}^{delay} = UpdateComponentServer(S_{comp}^{avail})$
21:                 $RepeatAvailability()$
22:             end if
23:         end if
24:         $Comp \leftarrow d$
25:         $UpdateServerCapcity(S_{d_i}^{avail})$
26:         if $delay = D_2$ then
27:             $RackId = S_{d_i}^{avail}.getRackId()$
28:             $DcId = NULL$
29:         else if $delay = D_3$ then
30:             $RackId = NULL$
31:             $DcId = S_{d_i}^{avail}.getDcId()$
32:         else
33:             $RackId = NULL$
34:             $DcId = NULL$
35:         end if
36:         $DependentScheduling(Comp, RackId, DcId, AllocatedServer)$
37:         $RedundantScheduling(Comp, RackId, DcId, AllocatedServer)$
38:     end if
39: end for
40: end
```

Figure 2.7: Interdependency algorithm of CHASE.

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. Fig. 2.9 shows the mapping algorithm.

2.4 MILP Evaluation

To assess the MILP model, different simulations are conducted using different data sets. The MTTF, MTTR, and recovery time are used as measures of the downtime and avail-

Algorithm 5 Redundancy Scheduling Algorithm

```
INPUT: $Comp, AllocatedServer$
OUTPUT: $S^{avail}$
        where $S^{avail} \subset S^{delay}$
begin:
for $rd \in Comp.Redundant$ do
    if $rd.isScheduled = FALSE$ then
        $S_{rd}^{delay} = UpdateComponentServer(AllocatedServer)$
        if $delay = D_2$ then
            $S_{rd}^{avail} = D_2.MaxMTTF(S_{rd}^{delay}, RackId)$
        else if $delay = D_3$ then
            $S_{rd}^{avail} = D_3.MaxMTTF(S_{rd}^{delay}, DcId)$
        else if $delay = D_4$ then
            $S_{rd}^{avail} = MaxMTTF(S_{rd}^{delay})$
        end if
    end if
    if $S_{rd}^{avail} isNotNULL$ then
        $UpdateServerCapcity(S_{rd_i}^{avail})$
        $AllocatedServer.add(S_{rd_i}^{avail})$
        $rd.isScheduled = TRUE$
        if $delay = D_2$ then
            $RackId = S_{rd_i}^{avail}.getRackId()$
            $DcId = NULL$
        else if $delay = D_3$ then
            $RackId = NULL$
            $DcId = S_{rd_i}^{avail}.getDcId()$
        else
            $RackId = NULL$
            $DcId = NULL$
        end if
        $DependentScheduling(Comp, RackId, DcId, AllocatedServer)$
    else
        Let Q be the mapping between C and CP
        $Q = doComputationalPathAnalysis(comp)$
        for $CP_i \in Q$ do
            for $c \in CP_i$ do
                $Comp \leftarrow c_{CP_i}$
                if $Comp_{c_{CP_i}}.isScheduled = FALSE$ then
                    $S_{c_{CP_i}}^{avail} = MaxMTTF(S_{c_{CP_i}}^{cap})$
                    $AllocatedServer.add(S_{c_{CP_i}}^{avail})$
                    $UpdateServerCapcity(S_{c_{CP_i}}^{avail})$
                    $Comp_{c_{CP_i}}.isScheduled = TRUE$
                    if $delay = D_2$ then
                        $RackId = S_{c_{CP_i}}^{avail}.getRackId()$
                        $DcId = NULL$
                    else if $delay = D_3$ then
                        $RackId = NULL$
                        $DcId = S_{c_{CP_i}}^{avail}.getDcId()$
                    else
                        $RackId = NULL$
                        $DcId = NULL$
                    end if
                    $DependentScheduling(Comp, RackId, DcId, AllocatedServer)$
                end if
            end for
        end for
    end if
end for
end
```

Figure 2.8: Redundancy algorithm of CHASE.

Algorithm 6 Mapping Algorithm

```
INPUT: $A = (A_1, A_2, ..., A_p)$
       $C = (C_1, C_2, ..., C_n)$
OUTPUT: $VM = (VM_1, VM_2, ..., VM_j)$
       $where\ j \leq n$
begin:
for $a_i \in A$ do
    if $delay = D_0$ then
        if $S_{a_i}^{cap} isNotNULL$ then
            $VM_1 = createVM()$
            for $c_i \in C$ do
                $Schedule\ VM_1\ on\ S_{ci}{}^{avail}$
                $Host\ c_i\ on\ VM_1$
            end for
        else
            for $CP_i \in M$ do
                $VM_i = createVM()$
                for $c_i \in Cp_i$ do
                    $Schedule\ VM_i\ on\ S_{ci}{}^{avail}$
                    $Host\ c_i on\ VM_i$
                end for
            end for
        end if
    else
        for $c_i \in C$ do
            $VM_i = createVM()$
            $Schedule\ VM_i\ on\ S_{ci}{}^{avail}$
            $Host\ c_i\ on\ VM_i$
        end for
    end if
end for
end
```

Figure 2.9: Mapping algorithm between cloud infrastructure and cloud applications.

ability of various components. To clarify the importance of the proposed model, different simulations are conducted to compare the model to the OpenStack Nova scheduler algorithm for different delay zones [17].

2.4.1 Availability analysis

As mentioned earlier, the availability $avail_c$ of a component *C* is inversely proportional to its downtime. Since the downtime was generated in terms of hours per year, then the availability is calculated as follows:

$$avail_c = (\frac{8760 - downtime_c}{8760} \times 100) \tag{18}$$

As for the downtime, it changes with delay types and can be calculated in terms of λ, MTTR, and/or *RT* of a component, its corresponding *DC*, rack *R*, and server *S*. Since our work considers redundancy and failover solutions, then downtime depends on λ and *RT*. For instance, the downtime in D_4 and D_2 type is calculated as (19) and (20) respectively:

$$downtime_c^{D_4} = (\lambda_c + \lambda_s + \lambda_{DC} + \lambda_R) \times RT_c \tag{19}$$

$$\begin{aligned} downtime_c^{D_2} &= ((\lambda_c + \lambda_s) \times RT_c) + (\lambda_R \times RT_R) \\ &+ (\lambda_{DC} \times RT_{DC}) \end{aligned} \tag{20}$$

2.4.2 Computational complexity

Any scheduling problem can be defined as a triplet $\alpha \mid \beta \mid \gamma$. Starting with α, it represents the problem environment. As for β and γ, they represent the problem constraints and the objective to be optimized respectively [18]. This triplet can take various fields depending on the scheduling type. Since the proposed scheduling work has *n* components to be assigned to *m* servers while minimizing downtime, it can be formulated as special case of the transportation problem. It can be represented as $Q_m \mid p_j \mid \sum h_j\ (C_j)$, where Q_m indicates that the problem environment consists of *m* different parallel machines, p_j determines that a job *j* can be processed using one machine *m* and finally *h(k)* represents the cost function to be optimized. This special case is known as the bipartite matching or assignment problem. It is represented as bipartite graph *G = (*n_1*,* n_2*, a)*. This graph consists of two sets of nodes n_1 and n_2 connected using arc *a*. This arc *a* = $\{$*j, k*$\}$ assigns node *j* of set n_1 to node *k* of set n_2 and is represented by the decision variable X_{cs} defined in Section 2.3.2. This type of scheduling problems is formulated using linear programming models, but it is characterized by NP-hard complexity hierarchy. Because of the NP-hardness of the proposed optimization model, it is only feasible for small DC networks [19]. In the evaluated small network, the number of variables generated in the optimization solver is approximately 4000.

2.4.3 OpenStack filter scheduler

OpenStack is an open source cloud management system that satisfies the needs of private and public clouds using computing, networking and storage services [17]. Nova is one of the computing components of OpenStack that schedules VM based on a predefined instance type such as CPU or RAM. Nova compute service provides different types of filters

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

Table 2.2: Evaluation parameters.

and weights to host an instance. During the filtering stage, the scheduler generates list of servers that are capable of hosting the instance. Then weight and cost functions are applied to this list to determine the best compute node for it [20]. Nova supports other filters that can be used to support HA placement such as availability zone, affinity, and anti-affinity filters, however, these existing filters are agnostic of the delay tolerance and inter-VM dependencies, which means they need to be extended. Also, the users have to manually define the needed filter based on the instance properties [20]. Alternatively, our proposed work eliminates the user interference. It provides an automated process for deploying VMs by considering functionality (resources and delay) and availability (different types of dependencies) requirements.

2.4.4 Results

Both the optimization model and the OpenStack scheduler are evaluated on a network consisting of 20 components, 2 DCs, 4 racks, and 50 servers. The server's and component's MTTFs are generated using an exponential distribution with mean = 2000 hours for both [21]. As for the server's and component's MTTR, a truncated normal distribution is used with mean = 3 and 0.05 hours and standard deviation = 1 and 0.016 hours respectively [21] [22]. For server's and component's recovery times, a truncated normal distribution is used with mean = 0.05 and 0.008 hours and standard deviation = 0.016 and 0.002 hours respectively. Table 2.2 shows the distributions for the availability measures. Each physical server has 30 GB and 32 CPU cores. VMs' instances are configured in small, medium, or large sizes [23] [24]. To evaluate the interdependencies and redundancies between components,

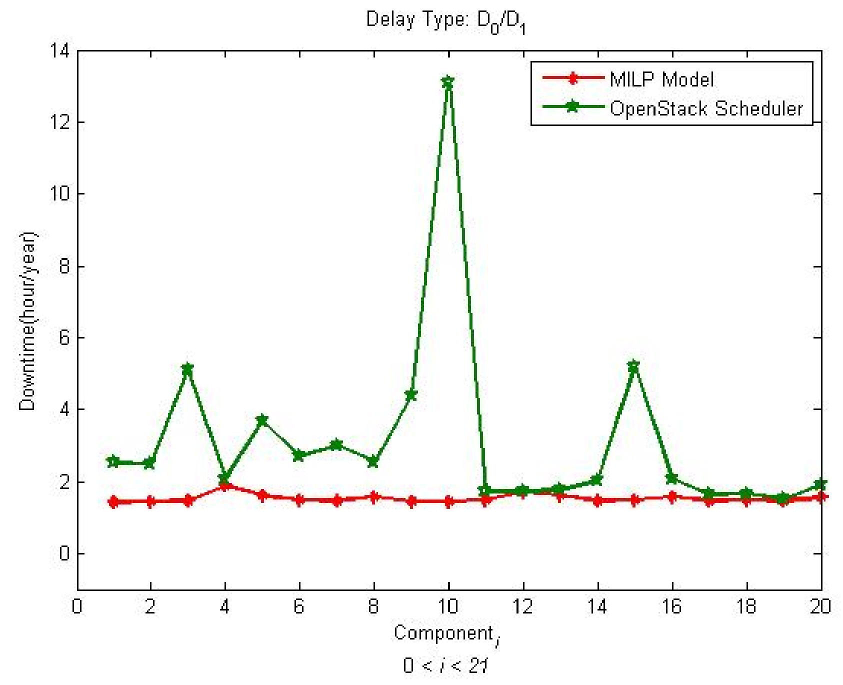


Figure 2.10: Downtime of each application's component using MILP and OpenStack scheduler for D_0/D_1 delay type.

the proposed MILP is evaluated on two real-time Web applications. The Web applications include two types of dependencies among the components: (1) the synchronization dependency between the active component and its replicas, and (2) the functional dependencies where the App server depends on database server and sponsors the HTTPS server.

2.4.4.1 MILP vs OpenStack Nova scheduler

The MILP model is compared to the core and RAM filters in Nova scheduler in order to show the impact of availability constraints on the downtime of each application's component per year. Using these filters, only servers with sufficient RAM and CPU cores are eligible for hosting VMs. Fig. 2.10, Fig. 2.11, Fig. 2.12, and Fig. 2.13 show the downtime difference for each application's component using MILP model and OpenStack scheduler for different delay zones. Since the Nova filters do not consider delay requirements, they generate similar results for each component for all delay types. As the delay zones widen the solution space for the components' placement, the difference between OpenStack scheduler and MILP increases gradually. Using the HA-aware MILP model, the downtime of each component is reduced by 35%, 39%, and 52% for D_0/D_1, D_2, and D_3

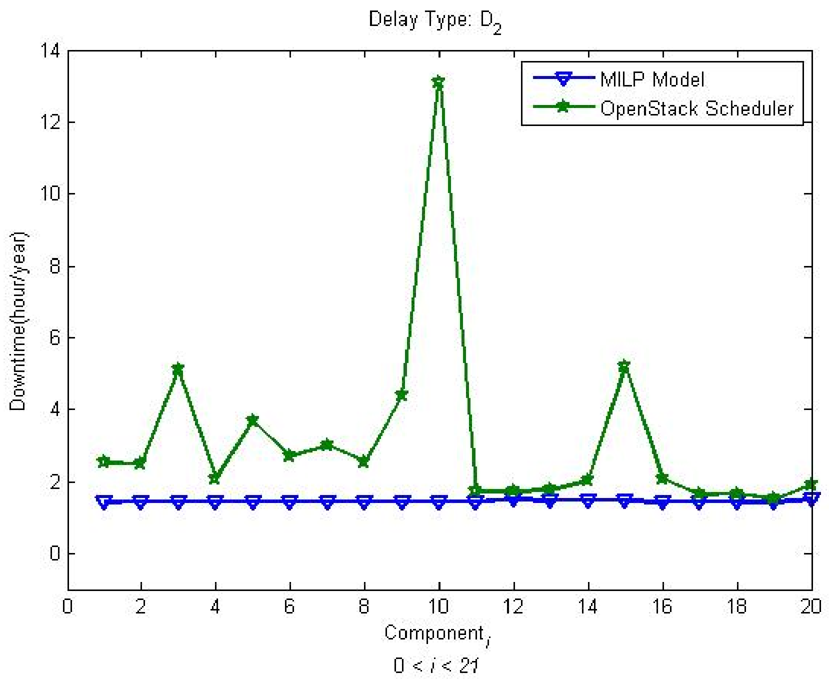


Figure 2.11: Downtime of each application's component using MILP and OpenStack scheduler for D_2 delay type.

types respectively. As for D_4 type, it allows scheduling component on any server in the cloud and allows placing primary and redundant components on two different DCs. In this case, if the server, rack, or DC of component C fails, C will be down until it fails over to its redundant. Therefore, the downtime of a C is calculated using (19). But sometimes a sponsor component can affect the downtime of its dependents if the latter cannot tolerate its failure. In any case, the component's downtime is reduced significantly by 97% using the HA-aware technique in D_4 type.

Note that the downtime varies among the components because each one is characterized by its availability metrics (MTTF, MTTR, and recovery time) and is deployed on a specific host that has its own MTTF and MTTR. In other words, Nova filters discard HA constraints, and consequently, they can host some components on servers with acceptable availability (high MTTF and low MTTR) and can also choose servers with low MTTF or high MTTR to deploy other components.

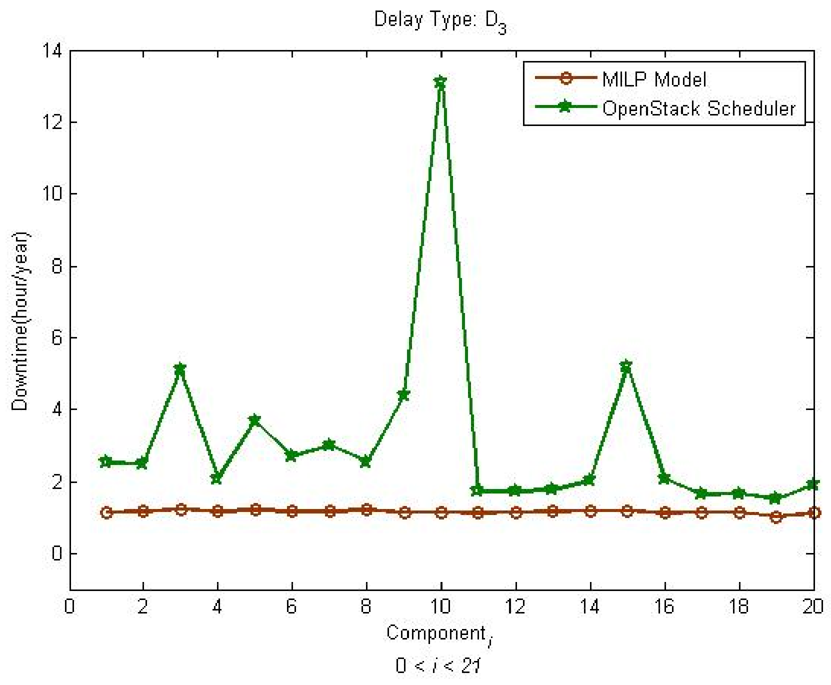


Figure 2.12: Downtime of each application's component using MILP and OpenStack scheduler for D_3 delay type.

2.4.4.2 Availability improvement within MILP model

Although the MILP model maximizes the availability of components, its results change among different delay zones. Since D_0 and D_1 types require placement of components in the same server, then the solution space of the selection process is limited. Consequently, the availability is affected and depends on the recovery time of the hosting server, rack, and DC. However, the availability is highly improved in D_4 because it depends only on the recovery time of the hosted component. Additionally, this delay zone eliminates the restrictions on the locations of the components. Fig. 2.14 shows the difference between downtime among the 5 delay zones. Compared to D_4 results, the availability of each component is reduced as more restrictions are added to the servers' location.

It is necessary to note that the confidence level of the deployments' results of the MILP model exceeds 95% for the same configuration settings of the cloud applications and infrastructure (i.e. same mean values for the MTTF and MTTR of the component and server as shown in Table 2.2). In other words, the MILP model generates the same placements' results for the application's components using the same cloud scenario and metrics.

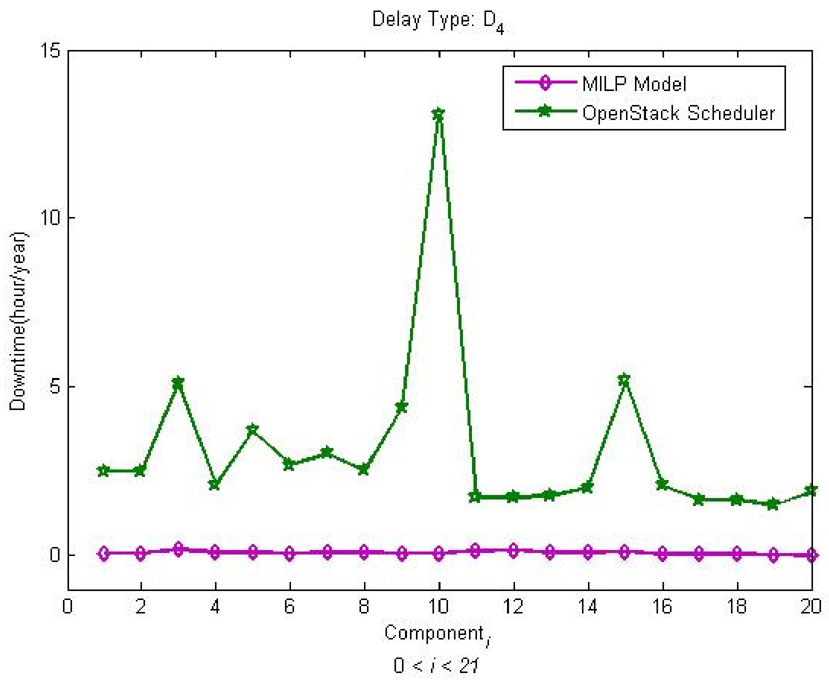


Figure 2.13: Downtime of each application's component using MILP and OpenStack scheduler for D_4 delay type.

2.5 CHASE Evaluation

To assess the proposed CHASE scheduler, small and large-scale simulations 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 the application's components.

2.5.1 Small-Scale network setup

The MILP model, OpenStack Nova scheduler [25], and CHASE are evaluated on a small-scale network. The above network setup is used to evaluate CHASE. This setup consists of 20 components, 2 DCs, 4 racks, and 50 servers. VMs are configured in small, medium, and large sizes using OpenStack options [24]. As for the availability measures, they are shown in Table 2.2.

To evaluate the interdependencies and redundancies between the 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 HTTPS servers. As

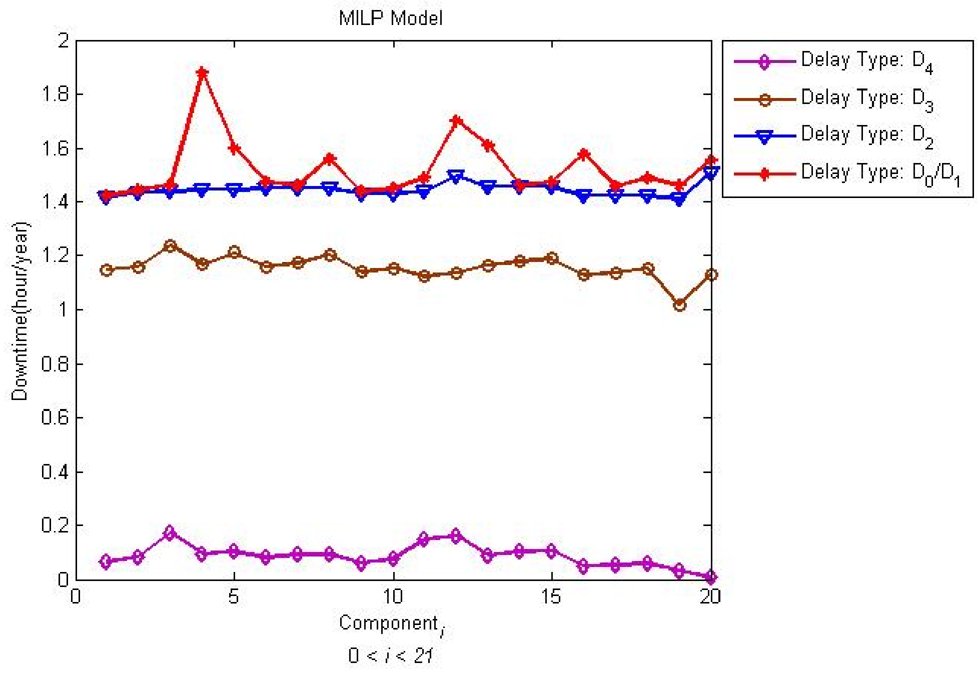


Figure 2.14: Availability improvement for each application's components among delay zones using MILP model.

for the interdependency relation, App server depends on a database server and sponsors an HTTPS server. The results of the small-scale evaluation are shown below.

1) CHASE vs MILP: Fig. 2.15, Fig. 2.16, Fig. 2.17, and Fig. 2.18 compare the downtime of each component using CHASE and the MILP model for different delay zones. There is a small gap between the MILP and CHASE for each component 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. 2.15, Fig. 2.16, Fig. 2.17, and Fig. 2.18 also compare the downtime of each component using CHASE and the core/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 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 each component is reduced by 48%, 34%, and 31% for

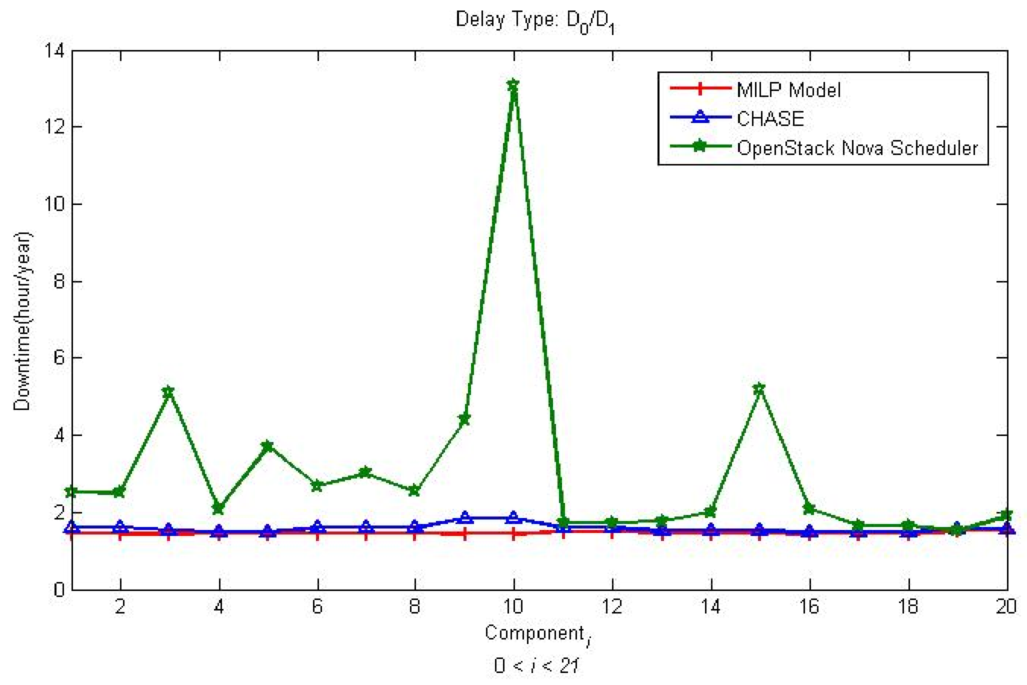


Figure 2.15: Downtime of each application's component in small-scale network for D_0/D_1 delay type.

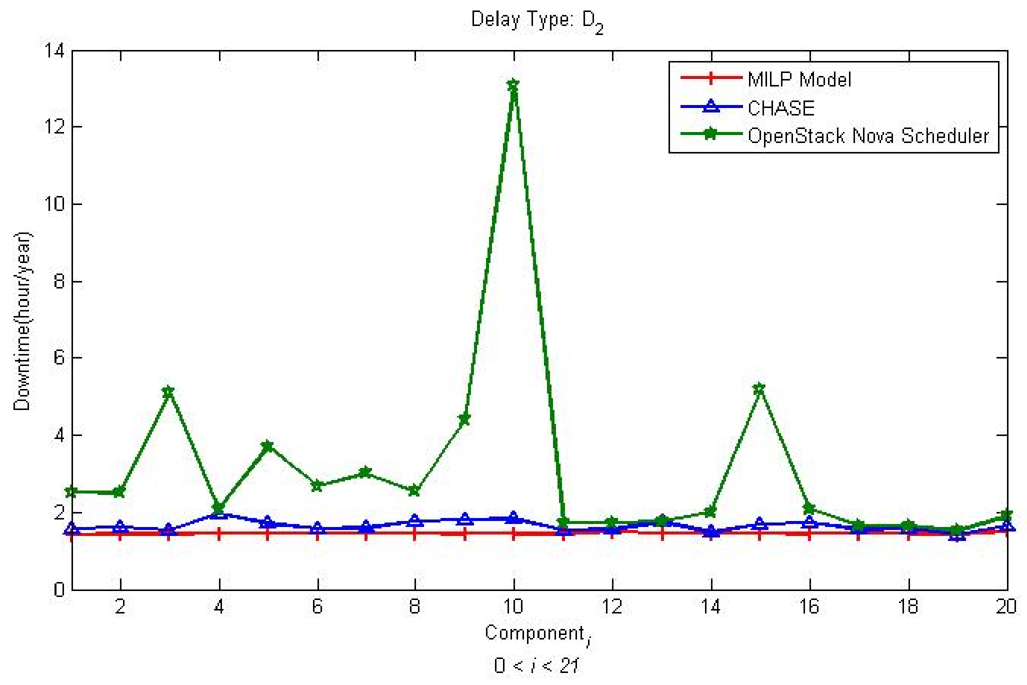


Figure 2.16: Downtime of each application's component in small-scale network for D_2 delay type.

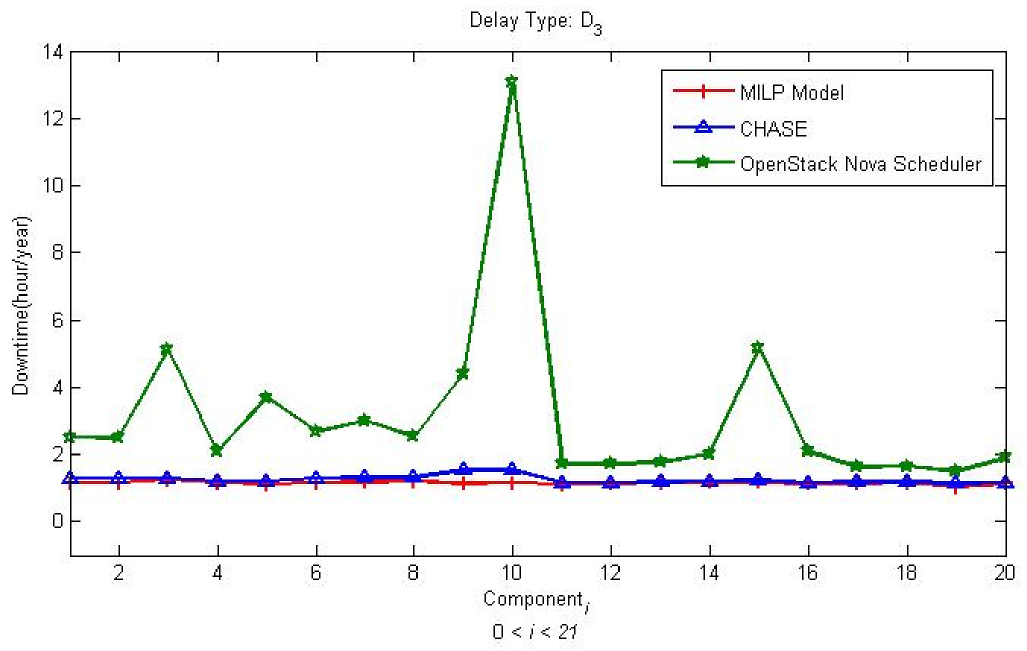


Figure 2.17: Downtime of each application's component in small-scale network for D_3 delay type.

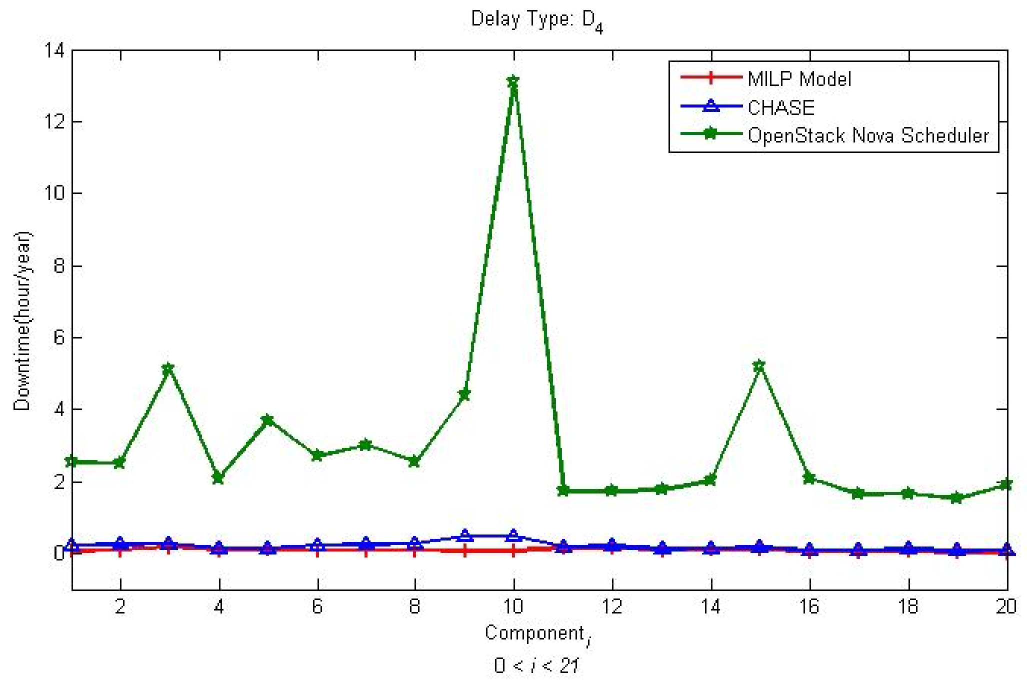


Figure 2.18: Downtime of each application's component in small-scale network for D_4 delay type.

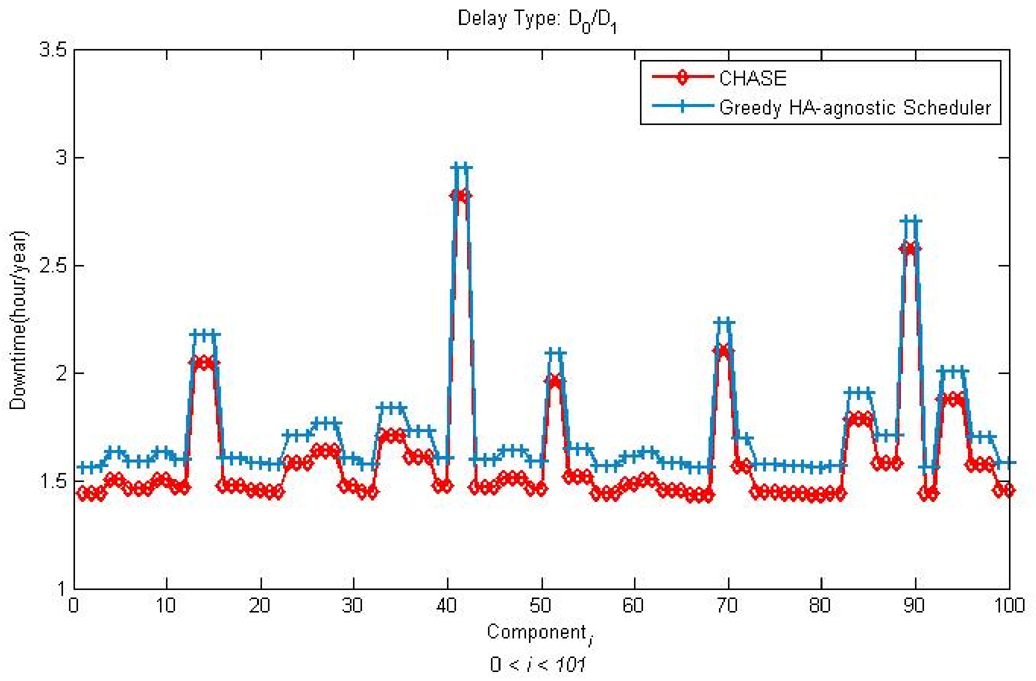


Figure 2.19: Downtime of each application's component in large-scale network for D_0/D_1 delay type.

D_3, D_2, and D_0/D_1 zones respectively. Since D_4 zone distributes the components between DCs, the component's 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 [1]. However, in D_3 zone for instance, the failure of DC affects the hosted components and their redundant ones. Consequently, end-users should wait for an execution of a repair policy for the DC or a migration plan for the components.

2.5.2 Large-Scale network

Since finding the optimal placement is an NP-hard problem, the MILP solution is only feasible for small networks [19]. Therefore, CHASE is proposed to remedy this issue and schedule cloud-based 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 2.2. 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

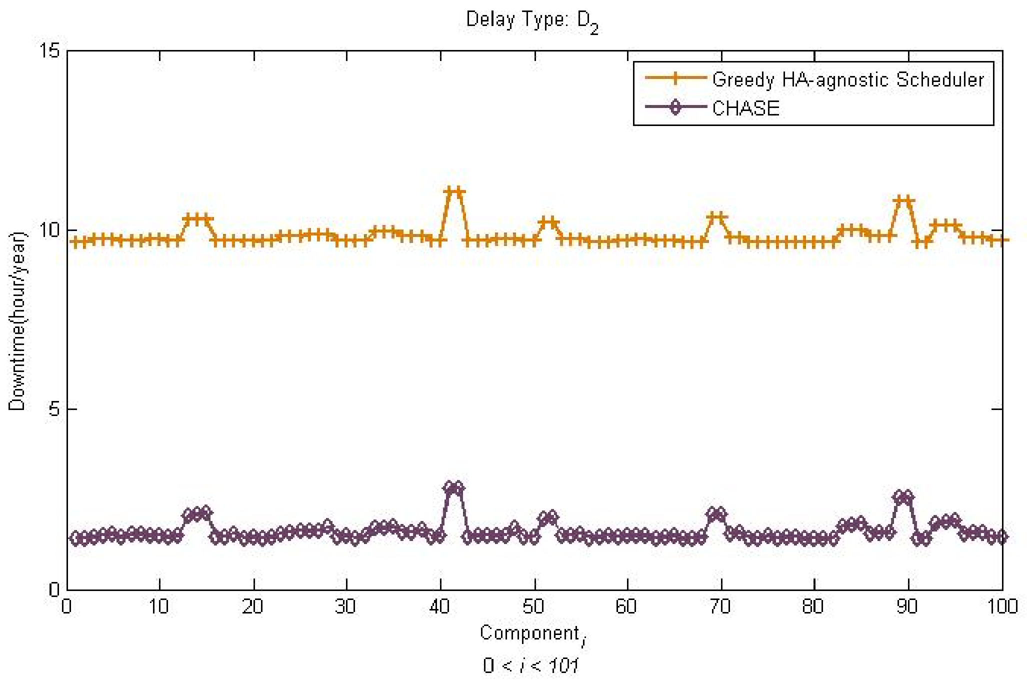


Figure 2.20: Downtime of each application's component in large-scale network for D_2 delay type.

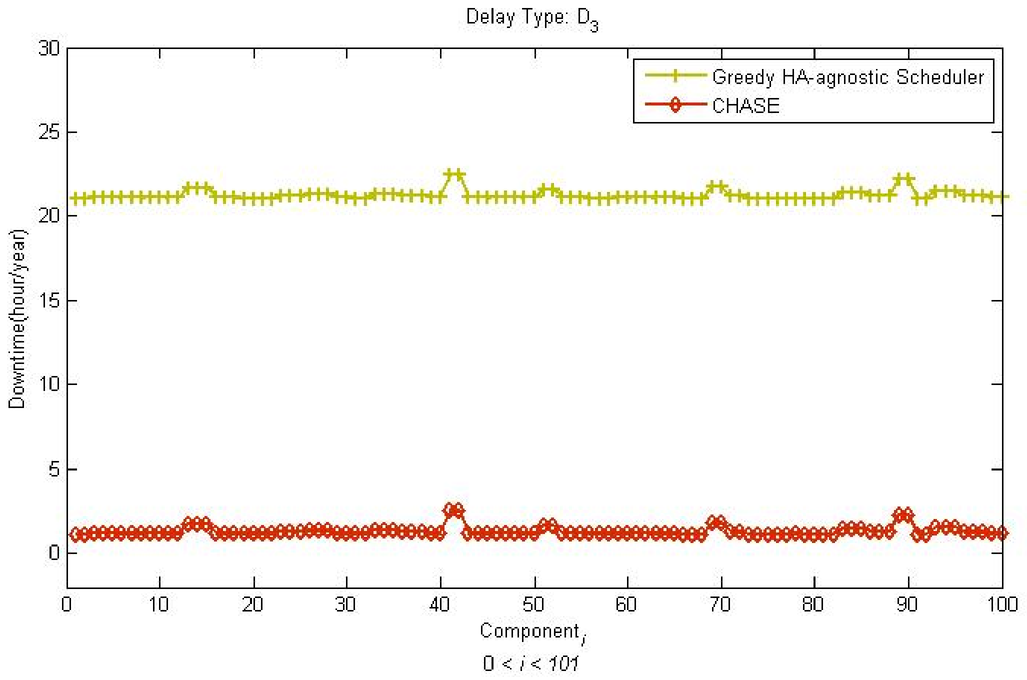


Figure 2.21: Downtime of each application's component in large-scale network for D_3 delay type.

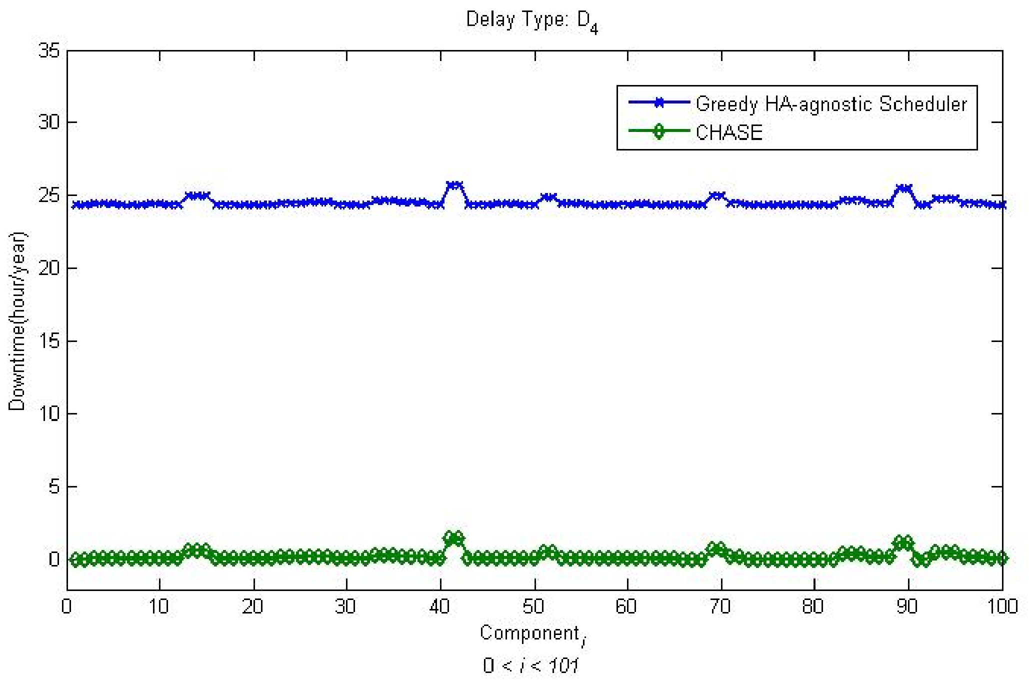


Figure 2.22: Downtime of each application's component in large-scale network for D_4 delay type.

the MTTF and MTTR of the component and server shown in Table 2.2. 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. 2.19, Fig. 2.20, Fig. 2.21, and Fig. 2.22 compare 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 criterion. 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

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 2.3: Availability improvement among different delay types using CHASE and RAS.

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.

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 2.2. 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 2.3. 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.

2.6 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 [17]. The scheduling tool is composed of several complementary modules as shown in Fig. 2.23. 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

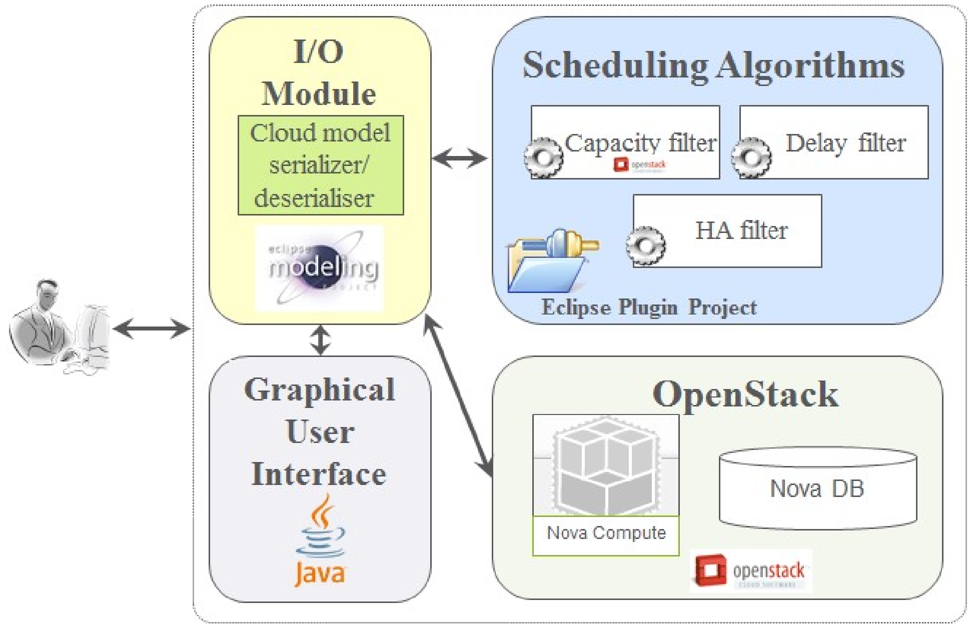
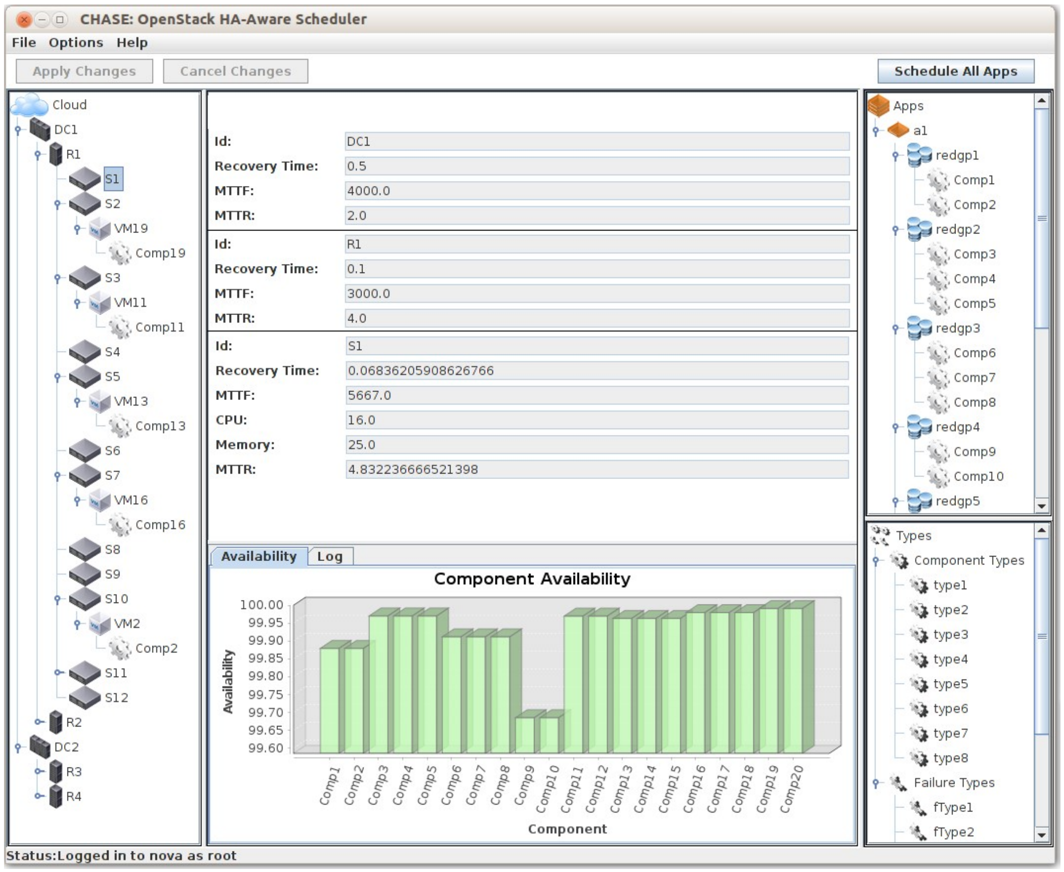


Figure 2.23: Architecture of CHASE prototype.

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. 2.24 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 [26]. Since Papyrus has limited support for the graphical modeling and Domain Specific Language (DSL) representation, the proposed implementation uses the



Id:	DC1
Recovery Time:	0.5
MTTF:	4000.0
MTTR:	2.0

Id:	R1
Recovery Time:	0.1
MTTF:	3000.0
MTTR:	4.0

Id:	S1
Recovery Time:	0.06836205908626766
MTTF:	5667.0
CPU:	16.0
Memory:	25.0
MTTR:	4.832236666521398

Status:Logged in to nova as root

Figure 2.24: A screenshot of CHASE GUI.

Java Swing library to define the GUI. The scheduling algorithms are implemented in Java.

2.7 Related Work

High availability is an interesting concept that has attracted several recent research studies. However, the way to attain a certain availability baseline when scheduling VMs or applications changes from one research study to another.

2.7.1 Replication approaches

Jung et al. propose a placement approach to generate VM configurations while maintaining high availability for multi-tier applications and improving their performance [27]. They develop a replication strategy to maintain HA constraints based on the mean time between failures (MTBF) while satisfying latency demands to minimize performance degradation for each application. 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.

Addis et al. address the resource allocation problem for deployment of multi-tier applications [28]. They aim to maximize total service level agreement (SLA) profit while maintaining a certain level of availability. They develop a non-linear programming model to achieve their objective while guaranteeing a level of availability based on a load-sharing fault-tolerance arrangement.

Other attempts that address the availability of VM deployments are proposed by Lu et al. and Wenting et al. [29] [30]. While Lu et al. show the effect of redundancy on availability analysis [29], both chapters have overlooked the effect of dependency models on that analysis.

Machida et al. propose a VM placement technique that generates redundant configurations to avoid VM outages during host's failures [31]. 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.

2.7.2 Diversified geographical sites and failover approaches

Li et al. address deployment of cloud applications that improves their availability and performance [32]. 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.

Both Harper et al. and Bin et al. propose a failover plan during the placement problem using different approach schemes [33] [34]. While Harper et al. exploit the co-location and anti-location constraints between interdependent applications [33], their capacity, and security constraints to provide a pseudo-optimal failover plan for an application, Bin et al. assign each VM a resiliency level that enables it to relocate to a new host if its current host fails [34]. It also uses the anti-location and co-location constraints between VMs to create a backup for them against any failure. Another attempt that maximizes service availability by providing a failure-resiliency plan is proposed by Abouzamazem et al. and Frincu et al. [35] [36]. Frincu et al. address the scheduling of application's components on the cloud infrastructure [36]. They propose a multi-objective scheduling approach that tends to maximize resource utilization, minimize the cost of application runtime, and maximize application's availability through a component replication approach.
Jinhua et al. propose a load balancing-aware scheduling algorithm of VM resources [37]. 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, Wenhong et al. develop a dynamic and integrated load balancing scheduling algorithm (DAIRS) for cloud DCs [38]. 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 literature studies 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 chapter 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.

2.8 Conclusion

Unexpected cloud-services outages can have a profound impact on business continuity and IT enterprises. The key to achieving availability requirements is to develop an approach that is immune to failure while considering real-time interdependencies and redundancies between applications. This chapter has addressed the problem environment from different vantage points to generate highly available optimal placement for the requested applications. The proposed MILP model minimizes the downtime of applications, but its computational complexity limits its evaluation on large networks. Therefore, the optimization model was associated with a heuristic solution. CHASE solves the scheduling problem in polynomial time while satisfying all QoS and SLA requirements and differentiating between mission-critical and standard applications. The MILP and CHASE were evaluated for different delay zones and different communication relations between components. CHASE prototype was designed to schedule components in a real cloud environment while communicating with OpenStack.

References

[1] 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.

[2] M. Jammal, A. Kanso and A. Shami, "CHASE: Component High Availability-Aware Scheduler in Cloud Computing Environment," *IEEE 8th International Conference on Cloud Computing*, pp. 477-484, 2015.

[3] 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.

[4] 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.

[5] 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.

[6] 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]

[7] 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]

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

[9] Eclipse MarketPlace, "UML Lab Modeling IDE," `https://marketplace.eclipse.org/category/free-tagging/round-trip-engineering-0`, March 23, 2017. [April 29, 2017]

[10] Visual Paradigm, "Round-trip engineering with Eclipse Integration," `https://knowhow.visual-paradigm.com/uml/round-trip-eclipse/`, August 2010. [May 1, 2017]

[11] Warwick Manufacturing Group, "Introduction to Reliability," `http://www2.warwick.ac.uk/fac/sci/wmg/ftmsc/modules/modulelist/peuss/slides/section_7a_reliability_notes.pdf`, 2007. [April 30, 2017]

[12] SA Forum, "Service AvailabilityTM Forum Application Interface Specification," `http://www.saforum.org/ResourceCenter/Download/16627~333319?view=1`, June 2014. [August 14, 2014]

[13] P3 InfoTech Solutions, "Web Application Deployment in the Cloud Using Amazon Web Services From Infancy to Maturity," `http://www.slideshare.net/p3infotech_solutions/web-application-deploymentaws#`, July 2013. [September 20, 2014]

[14] P. Bodik, F. Armando, *et al.*, "Characterizing, modeling, and generating workload spikes for stateful services," *in ACM Symposium Cloud Computing,* pp. 241-252, June 10-11, 2010.

[15] 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]

[16] Reliability HotWire, "Basic Concepts of FMEA and FMECA," `http://www.weibull.com/hotwire/issue46/relbasics46.htm`, 2017. [May 1, 2017]

[17] OpenStack, "OpenStack Cloud Software," `http://openstack.org`. [September 20, 2014]

[18] M. Pinedo, *Deterministic Models: Preliminaries, Scheduling Theory, Algorithms and Systems,* Spring, New York, pp. 13-33, 2008.

[19] 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. 3-13, January 2011.

[20] OpenStack, "Filter Scheduler," `http://docs.openstack.org/developer/cinder/devref/filter_scheduler.html#costs-and-weights`, February 2013. [September 20, 2014]

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

[22] EventHelix, "System Reliability and Availability," `http://www.eventhelix.com/realtimemantra/faulthandling/system_reliability_availability.htm#.U-AcuPldXCf`, 2014. [September 20, 2014]

[23] Microsoft, "How to: Change the Size of a Windows Azure Virtual Machine," `http://msdn.microsoft.com/en-us/library/dn168976(v=nav.70).aspx`, 2013. [September 20, 2014]

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

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

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

[27] 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 Conference Dependable Systems and Networks,* pp. 497-506, June 28-July 1 2010.

[28] B. Addis, D. Ardagna, B. Panicucci, and L. Zhang, "Autonomic Management of Cloud Service Centers with Availability Guarantees," *IEEE International Conference Cloud Computing (CLOUD),* pp. 220-227, July 5-10, 2010.

[29] Q. Lu *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.

[30] 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.

[31] 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.

[32] J. Li *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.

[33] 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.

[34] E. Bin *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.

[35] A. Abouzamazem and P. Ezhilchelvan, "Efficient Inter-cloud Replication for High-Availability Services," *IEEE International Conference Cloud Engineering (IC2E),* pp. 132-139, March 25-27, 2013.

[36] M.E. Frincu and C. Craciun, "Multi-objective Meta-heuristics for Scheduling Applications with High Availability Requirements and Cost Constraints in Multi-Cloud Environments," *IEEE Conference Utility and Cloud Computing,* pp. 267-274, December 5-8, 2011.

[37] 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.

[38] 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.

Chapter 3
Scrutinize High Availability-aware Deployments Using Stochastic Petri Net Model and Cloud Scoring Selection Tool

3.1 Introduction

With the cloud computing era, many business applications are offered as cloud services where they can be accessed anytime and anywhere [4]. Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) are essential forms of cloud services provided for many enterprises, such as Microsoft Azure and Amazon Elastic Compute Cloud (EC2) [5] [6]. Depending on the cloud user's needs, PaaS and IaaS provide the required web applications and computational resources in the form of virtual machines (VMs). With the widespread of on-demand cloud services/VMs, their availability becomes a paramount aspect for cloud providers and users [7]. It is important to note that availability is the percentage of time where these services are available in a given duration. Cloud services encounter different types of hardware and software failures and consequently become unavailable [8]. As for the cloud users, they cannot prevent or mitigate the service downtime unless they have their proprietary high availability (HA) solutions, such as the Netflix HA approach [9]. Therefore, cloud users and providers should handle the different hardware and software failures, and other sporadic uncertainties by selecting and analyzing the best application

Part of this chapter has been published [1] and [2]. The work is then extended to a journal and is submitted for publication [3].

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC-STPGP 447230) and Ericsson Research.

deployments in a given cloud environment. Nowadays, cloud users and providers depend on affinity/anti-affinity policies, overprovisioning practices, and multi-zone/region deployments to achieve high availability rather than defining a comprehensive and analytical model to analyze the HA of a cloud application. For instance, OpenStack Nova schedulers use anti-affinity/affinity filters and availability zones notions to deploy applications in geographically distributed data centers (DCs) in a given cloud to maintain high availability [10]. Although these notions minimize outage of cloud applications, they are still missing a quantitative model to analyze the availability of these applications and provide generic guidelines for HA-aware scheduling solutions. The deployment of cloud applications to maintain the preferred availability is not a straightforward process. Therefore, cloud providers should offer and evaluate HA solutions that mitigate any encountered downtime and recover any data loss.

With the cloud being the lifeblood of many information technology (IT) applications and telecommunication services, its DCs can be seen as an opportunity for flexible integration of multiple compute capabilities and virtual services for performance-aware and on-demand access [11] [12]. Therefore, the comprehensive and analytical HA-aware model should be associated with a performance-aware cloud scoring tool to select the best HA-aware, energy efficient, and cost-aware applications deployments. The objective of this chapter is to define an availability analysis approach that considers the effects of hardware and software failure types, recovery duration, load balancing delay, and user request processing time and accordingly assesses whether the given cloud deployment would be able to satisfy the availability and performance requirements of the service level agreement (SLA).

Our approach is based on a Stochastic Petri Net model (SPN) and a policy-driven cloud scoring system to evaluate the availability of cloud services deployed in geographically distributed data centers and select the optimal one [13]. Fig. 3.1 summarizes this approach. First, the proposed Stochastic Petri Net model captures the characteristics of the cloud provider and user. It translates them into elements of an availability model that can be solved to calculate the expected availability and subsequently be used to guide the cloud scheduling solution. These elements are then synchronized according to their interdependencies in order to form a stochastic availability model. The model generates HA-aware de-

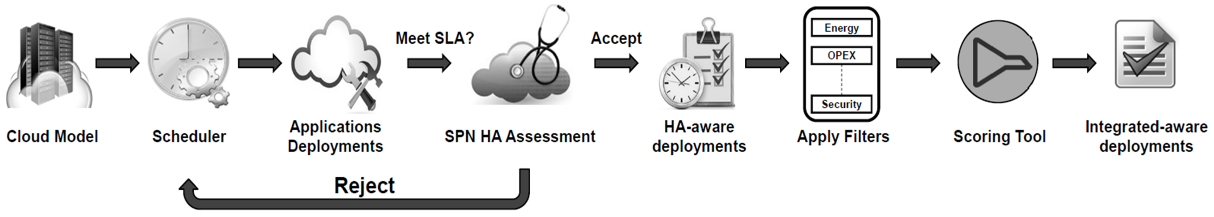


Figure 3.1: SPN model and scoring selection approach.

ployments for a given application. These deployments are then inputted to the cloud scoring tool to select the optimal one according to predefined policies, such as lower operational expenditure (OPEX) and carbon footprint. Initially, our approach is used to evaluate whether a given cloud deployment satisfies the availability requirements of a cloud-deployed application. Then it provides a policy-driven ranking system to weight the best HA-aware deployments and select the optimal ones among them. The scoring solution introduced in this chapter is a generic approach where the evaluation criteria is determined based on the cloud providers preferences, and the selection process is modified accordingly.

In this chapter, we discuss the challenges of availability analysis of cloud deployed applications. Additionally, we propose a comprehensive HA-aware analysis approach of cloud-deployed applications using SPN [1] [2]. The latter uses SPN model to evaluate application scheduling in a cloud environment while considering the HA objective and impact of functionality constraints on their performance. Elaborate simulation results are generated to provide guidelines for cloud deployments approaches. Although this approach models the application behavior in terms of HA, it discards other challenges associated with cloud applications deployments, such as energy and cost efficiency. It is necessary to design a system that integrates HA-aware cloud applications deployments with other cloud applications issues. Therefore, we escalate that work to the following:

- Associate the HA-aware SPN model with policy-driven cloud scoring system.
- Capture energy/OPEX as scoring policies to provide HA and performance-aware scheduling of cloud applications.
- Integrate the scoring policies with the functionality and availability constraints to select best placements of application components to maximize HA and maintain energy/cost needs.

- Envision user needs and assess DCs capabilities to filter out best HA-aware deployments to minimize Greenhouse Gas (GHG) emissions and OPEX in DCs.
- Scrutinize the deployments results of the SPN model using the scoring tool and comprehensive analysis.
- Provide an extensible scoring system that depends on the generic cloud environment.
- Modify the evaluation criterion based on the capabilities and preferences offered by the cloud providers such as green and cost criteria to evaluate cloud DCs.

The rest of this chapter is organized as follows. Section 3.2 and 3.3 define the problem background where it presents the different challenges of placement of applications components, the need for SPN models, and scoring selection system for deployments of cloud applications. Section 3.4 describes the cloud model, the cloud deployments, the proposed SPN model, and the scoring selection system. Section 3.5 describes the evaluation and results of the SPN model and the scoring selection tool. Section 3.6 presents some related works for availability analysis as well as green- and cost-aware scheduling. Finally, Section 3.7 concludes the chapter.

3.2 Modeling High Availability in Cloud

Many HA solutions have been proposed to mitigate the software or hardware failures of a virtualized system [14], [15], and [16]. However, these approaches do not associate their solution with an availability assessment model to evaluate the impact of the above requirements on that solution. Different types of failures affect the cloud infrastructure and applications. Besides, some challenges are raised when choosing best deployment of cloud applications while satisfying the HA, functionality, green, and cost requirements. Therefore, it is necessary to understand the various failure forms affecting the cloud model, the HA and performance-aware scheduling challenges, and the need for a SPN model and cloud scoring selection tool to handle them.

3.2.1 Failure types and distributions

The cloud model typically consists of multiple data centers, each having a set of servers and a set of applications with multiple components. Using the appropriate scheduling solution, the applications are hosted on the servers that best fit the application requirements using VM (or containers) mapping. Consequently, any DC/server failure mode can bring the hosted application down whether it is a planned or unplanned outage. Unplanned downtime can be defined as the time where a system enters a failure mode and becomes unavailable. Such downtime is a result of an unexpected failure event, and consequently, neither the cloud provider nor the users are notified of it in advance. Therefore, it is necessary to have a model that takes into account the actual effect of failures on the system availability. There are different forms of failures:

1) Hardware/Infrastructure failures [17] [18]: Such failures happen at the data center and server layers. They can be the results of faulty elements of the server, storage, and network, such as faults in memory chips, disk drivers/arrays, switches, routers, or cabling. Such failures can be captured by the failure rates of the servers as well as the entire DC.

2) Application failures [19]: Such defects occur at the application and VM/container levels. They might be generated from the hypervisor malfunctioning, unresponsiveness of the operating system, files corruption or viruses and software bugs, such as Heisenbugs, Bohrbugs, Schroedinbugs, or Mandelbugs [20]. We capture such failures by the failure rates of the components and VMs/containers.

3) Force majeure failures [21]: These failure events affect both the cloud provider infrastructure and the cloud applications. They are generated from power loss, storms, fires, earthquakes, floods, and other natural disasters. Due to their scale, we capture such failures by the failure rate of the DC.

4) Cascading failures: These failures are the results of an accumulated impact of hardware or software failure. For example, a malfunctioning dynamic host configuration protocol (DHCP) server can flood the network with DHCP requests causing a DC failure. Consequently, its corresponding servers, their hosted applications, and VMs/containers will become inaccessible. The functionality of the corresponding application or VM/container is ceased, which associate its recovery with the repair or recovery policy of its host. Due to their propagation impact, we capture such failures by the failure rate of the DC.

Each of the previous failure states is associated with a failure rate or mean time to failure (MTTF) and mean time to repair or recover (MTTR) determined by the used repair or recovery policy. Due to the stochastic nature of the corresponding failure events, it is assumed that they are generated using certain probabilistic distribution functions. However, there is no restriction or specific consent on the distribution type of every failure event. It can follow exponential, Weibull, normal, or any other stochastic model. Regarding the recovery or repair policy, it is assumed to have a deterministic or a stochastic nature depending on the used recovery behavior [22].

The exponential failure distribution has been used in many previous failure analysis and availability related works [23], [24], [25], [26], [27], and [28]. Therefore, in this chapter, the exponential failure distribution is used to reflect failure rate or MTTF of DC, server, application, and VM/container. Such distribution is applied on all the stochastic failure transitions of the proposed Stochastic Petri Net model. As for the repair/recovery timed transitions, a deterministic distribution is applied on them to trigger any repair or recovery behavior for the DC, server, application, and VM/container [29] [30]. It should be noted that our approach also supports other failure and repair rates, as our model does not depend on a specific probability distribution.

3.2.2 Multi-tier applications in the cloud:

When it comes to HA-aware scheduling of applications in a cloud environment, various HA approaches can be adopted to mitigate the outage impact. Some scheduling solutions are associated with load balancing mechanism for HA purposes while other schedulers incorporate their approach with replication or failover techniques to maintain certain HA baseline. The challenge here lies in selecting the best deployment model while analyzing the impact of the adopted HA mechanism, different failure types, functionality constraints, the redundancy, and interdependency models between different components. For instance, multi-tier application uses redundancy models and load balancing to maintain certain HA baseline. Each layer consists of a primary component backed up with multiple active components depending on the used redundancy model. Upon arrival of requests, the used load balancer distributes them between different servers. Through constant monitoring, it ensures that these requests are served by healthy VMs. Upon failure detection, the load balancer removes the faulty machine from the load balancing group and redirects the request to a healthy one.

Typical web applications consist of three-tiers with a frontend (e.g. multiple Hypertext

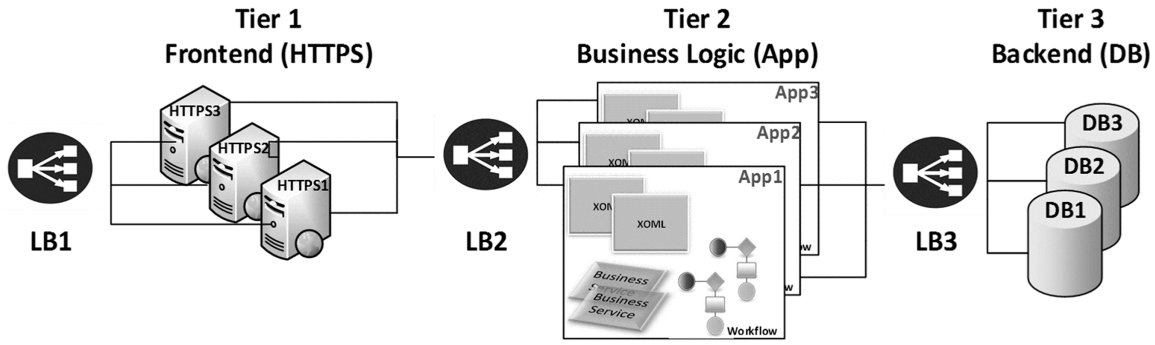


Figure 3.2: Example of three-tier web application.

Transfer Protocol Secure (HTTPS) servers), a business logic application (App) on the middle tier, and a database (DB) storing the system state at the backend. The HTTPS servers depend on the App, which in turn, is sponsored by the DB. Each component type (HTTPS, App, and DB) consists of a primary component and multiple active replicas as shown in Fig. 3.2. Each type is associated with certain failure types. When it comes to deploying such application in a single cloud with geographically distributed DCs, multiple options are to be considered on whether inter- or intra- DC deployment should be selected. It is not always the case that maximum inter-DC distribution is preferable because this decision depends on many factors, such as the failure distributions, recovery behaviors, and the used HA mechanisms as we will demonstrate in Subsection 3.5.1.

3.2.3 Stochastic Petri Nets in the cloud:

The stochastic nature of service failures and the urgent need for availability solutions require an availability evaluation model that identifies failures, their underlying causes, and mitigates the associated risks and service outages. It has been shown that analytical models, such as SPNs and Markov chains have been used to analyze the reliability/availability of many complicated IT systems [17] [18] [31]. However, the complicated nature of cloud infrastructure configurations and dynamic state changes require a comprehensive and analytical availability-centric model [32]. Such model should satisfy essential requirements consisting of:

- Capture the stochastic nature of failures according to different probability distribution functions.

- Capture the cloud stack (DCs, servers, and virtual environment (VE)) and the correlation aspect of their failures.
- Capture the functional workflow between the components of multi-tier applications (queuing and request forwarding) as well as the HA mechanisms they employ (load balancing and redundancy schemes).
- Capture different deployments of the application components in the cloud (inter- vs. intra-DC deployment).
- Assess and quantify the expected availability of the application according to its cloud deployment.

Petri Nets (PNs) are widely used to model the behavior of different Discrete Event Systems (DES) [33]. They are graphically presented as directed graphs with two types of nodes: places and transitions. Different extensions of PNs are introduced in the literature to make them more expressive. Deterministic Stochastic Petri Nets (DSPN) are one of Petri Nets extensions for modeling the systems with stochastic and deterministic behaviors [34]. Three transition types are defined in DSPN: immediate transitions model the actions that happen without any delay under a condition, timed transitions model the actions that happen after a deterministic delay, and stochastic transitions model the actions that happen after an exponentially distributed delay.

DSPN is formally presented as a tuple of $(P, T, I, O, H, G, M_0, \tau, W, \Pi)$ where *P* and *T* are the non-empty disjoint finite sets of places and transitions, respectively. *I* and *O* are the forward and backward incidence functions such that *I, O:* $(P \times T) \cup (T \times P) \longrightarrow \mathbb{N}$ where $\mathbb{N}$ is the set of non-negative integers. *H* describes the inhibition conditions. *G* is an enabling function that given a transition and a model state determines whether the transition is enabled. M_0 is the initial marking. The function τ associates timed transitions with a non-negative rational number ($\tau : T \longrightarrow \mathbb{Q}^+$, where $\mathbb{Q}^+$ stands for the set of non-negative rational numbers). The function *W* associates an immediate transition with a weight (relative firing probability). Finally, Π associates an immediate transition with a priority to determine a precedence among some simultaneously firable immediate transitions. Note that the priority of timed transitions (either deterministic or stochastic) against immediate ones is zero.

To model the behavior of an application running on the cloud with stochastic failures and deterministic recovery events, we have used Stochastic Colored Petri Net (SCPN). SCPN supports both stochastic and deterministic events, and it is a class of DSPN models where the tokens can have different colors (types) [35]. The model is simulated and analyzed using TimeNET [36]. Although DSPN imposes the restriction of only one enabled deterministic transition in each marking, TimeNET provides transient and stationary analysis of SCPN without any restriction on the number of concurrently enabled transitions. In Subsection 3.4.2, we explain the SCPN model proposed for a multi-tier application deployed in the cloud.

Although the SCPN model captures the cloud characteristics and translates them into elements of an availability model, it overlooks the other challenges associated with the cloud. In the following, we explain the policy-driven scoring system that weighs the HA-aware deployments and selects the optimal ones according to a predefined policy (i.e. green and/or cost).

3.3 Cloud Scoring System

Energy efficiency, carbon footprint, and OPEX are gaining a lot of interest in information and communication technology (ICT) sector and cloud market. DCs and server farm spaces can devour up to 100 times as much power as typical offices [37]. With this high energy consumption, DCs are supposed to have performance- and energy-aware configuration measures that can lessen the power use and save OPEX, all aimed at having HA, green, and cost-aware solutions. This section explains the need to associate the analytical SCPN model with a cloud scoring tool. The latter tool selects the optimal green- and/or cost-aware deployment based on functionality features, such as lower carbon footprint and OPEX.

3.3.1 Motivation:

Nowadays, the size of DCs has increased significantly to satisfy the migration to the cloud and the growth in the usage of internet services [38]. Besides, many telcos are selling their DCs and moving to the cloud, such as Verizon and AT&T [39]. With more DCs being built, more services will be provided to the cloud users, and additional investments and incentives will be brought to the market. This increase in the rate of DCs construction is accompanied

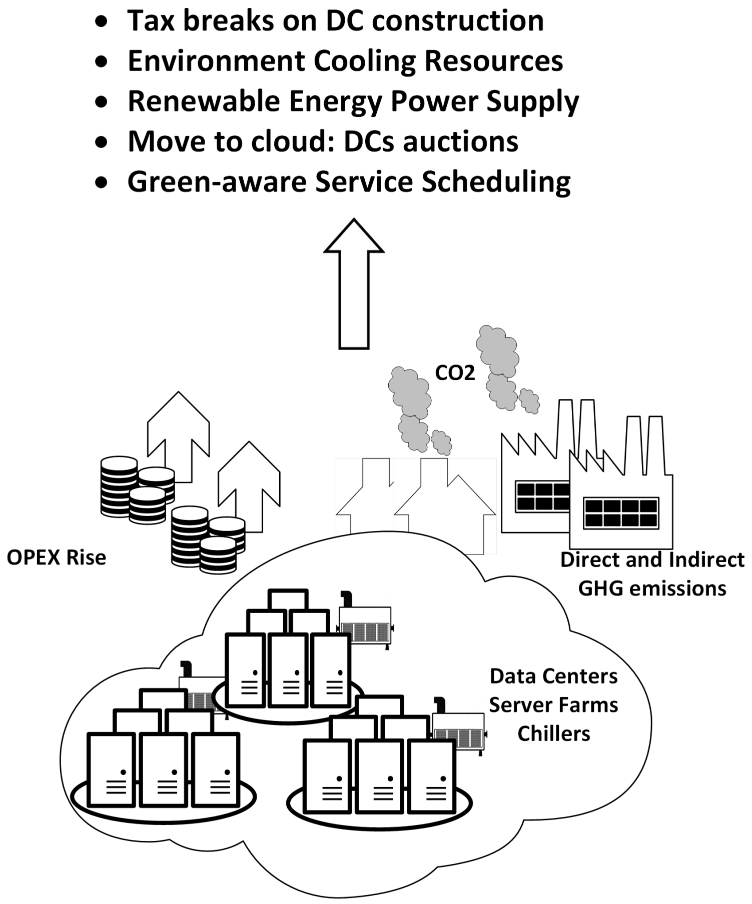

• Tax breaks on DC construction
• Environment Cooling Resources
• Renewable Energy Power Supply
• Move to cloud: DCs auctions
• Green-aware Service Scheduling
CO2
OPEX Rise
Direct and Indirect GHG emissions
Data Centers
Server Farms
Chillers

Figure 3.3: Different energy challenges and solutions in cloud DCs.

by a significant growth in energy consumption that might exceed in some scenarios the thresholds introduced by the power delivery and cooling systems. In a DC, a large amount of energy is wasted due to underutilized servers, cooling solutions, and heat dissipation of electronic or network equipment. It was estimated that electricity usage in DCs is around 61 billion kilowatt-hours [40]. It is equivalent to the consumption of 5.8 million United States (U.S.) households, and it is equal to the electricity usage of the U.S transportation manufacturing industry.

DCs are also going to face an increase in operational costs due to the high energy consumption. Running a large DC can cost 10 to 25 million dollars per year where 42% of the cost is allocated to OPEX [41].

Regarding GHG emissions, they depend on the power consumption of DCs, the grid electricity, power supplies, and the materials used for power delivery. Based on Natural Resources Defense Council (NRDC) nationwide study, DCs consume around 3% of the energy produced globally, but they generate 200 million metric tons of carbon [38].

GHG emissions also depend on the geographical location of a DC. For instance, a DC located in a region where renewable energy resources can be accessed produces lower GHG emissions compared to a DC located in an area using coal or natural gas as energy resources. However, it is not always the case where it is allowed to place DC in environment friendly areas due to some governmental restrictions. Additionally, delay constraints between interacting cloud applications can impose an obstacle in designing green-aware solutions.

To mitigate the above challenges, energy efficiency directive proposes a "20-20-20"% energy improvement, renewable energy consumption, and carbon footprints reduction framework to be issued by 2020. ICT sectors are integrated with this framework, and they introduce strategies to achieve the designed target [42] [43]. They are incorporating in this framework in direct, indirect, and systematic ways to reduce its energy demands and carbon emissions. Greenpeace is also pushing IT and telecommunication companies to have carbon-free DCs [44]. One solution could be a migration to the cloud and adoption of virtualization concept. The VMs, containers, and consolidation concepts can eliminate idle servers and reduce OPEX while providing 75% increase in server efficiency [41] [45].

With the migration to the cloud, its providers are searching for alternative solutions to reduce the high energy consumption and expenditures. They adopt multiple approaches, such as using renewable energy and building DCs in cooler areas to reduce cooling cost and earn carbon tax credits. For example, many local governments are promising faster adoption of green DC such as Paris (United Nations Climate Change Conference (COP21) agreement), San Diego, and Las Vegas [46]. Norway as well is trying to minimize cooling costs by using the environment resources. It is planning to use fjord to get cold water and cool halls for its DCs [47]. Lately, Facebook has announced the construction of one of the most sustainable, reliable, and green DC, Lulea [48].

Other cloud providers are benefiting from the tax breaks on building DCs in some states to reduce costs. For instance, Arizona is providing tax breaks for big and medium-size multi- and single-tenants' operators in order to motivate DCs constructions [49].

Overall, the construction of more DCs and server farms is associated with high energy consumption, a rise in expenditures, and GHG/carbon emissions. Many solutions are designed to address these challenges and provide green and cost-aware cloud environment, such as renewable energy, cloud solutions/scheduling, and government tax breaks. Fig. 3.3 represents the above challenges and solutions in DCs.

It has been shown that power and cooling solutions in DCs can reduce power bills, capital investments for power plants, and GHG emissions, but one major impediment is raised regarding the reliability and performance. It is necessary to delineate an approach that will compromise between the availability, cost, and green requirements. To ensure redundancy and workload proximity, cloud providers should have multiple geographically distributed DCs, each with a different OPEX. Having a profitable cloud necessitates a scoring mechanism that distributes the workload while satisfying the HA requirements (different availability zones, SLA level) and minimizing DCs energy consumption and OPEX. Note that the scoring selection tool can use objectives other than green and cost efficiency depending on the predefined options of the cloud providers.

3.3.2 Cloud scoring approach:

This chapter proposes a SCPN model that analyzes the availability of cloud applications and chooses their best deployment nodes that satisfy HA requirements and functionality constraints, such as latency and computational resources. In some cases, multiple HA-aware deployments might be eligible for the application components with certain MTTF and MTTR values of examined DCs. For instance, if the cloud user is looking for HA-baseline greater than 90%, SCPN evaluation can end up with more than one satisfactory solutions. Therefore, a scoring selection tool is needed to add weights to the selected deployments and select optimal ones among them. The scoring selection tool is extensible and can address different preferences of cloud providers. It has an evaluation criterion with multiple options to allow scoring the deployments. In order to determine a pragmatic evaluation methodology, some afore steps are considered:

1) User Requirements Envisioning: The scoring approach envisions the user requirements and usage patterns to generate certain groupings of the application components. For instance, if the deployment of a 3-tier web application is scrutinized using the scoring tool, the envisioning process should consider the interdependencies between components and examine tolerance time of the dependent ones to generate the possible groupings. In the 3-tier web application case, HTTPS depends on the App that is sponsored by the DB. If the HTTPS cannot tolerate the failure of App, HTTPS and App should be deployed in the same DC. Consequently, a co-location group is generated, and the evaluation criterion selects the best green and cost-aware DC accordingly. In this work, we focus on green and cost objectives as the evaluation criterion to select optimal placement of the applications components.

2) Cloud Infrastructure Assessment: It is necessary to measure the DCs capabilities in terms of OPEX, carbon footprint, governmental regulations, usage patterns, etc [50]. With these measures, DC workloads can be evaluated, and consequently, the overload factor can be calculated for each DC. Overload represents the increased load that a DC can handle upon a sudden failure, slashdot effect, or any other growth in workload. Generally, overloads require load balancing to distribute them to the DC that satisfies certain HA, green or any other objective. Therefore, each DC is associated with its overload factor to help select best DC upon load distribution or redirection process. In order to determine the overload factor, it is necessary to select a baseline DC. The baseline DC, DC_b, is the DC that has the highest GHG emissions and OPEX. Therefore, we have assumed that DC_b does not improve OPEX, GHG emissions, or other metrics preference compared to other DCs with higher metrics. Once the baseline is determined, it is assigned an overload factor OL_b of 1. Then the overload factors of remaining DCs, DC_{r_i}, are calculated accordingly. For example, if DC_{r_1} has low carbon footprint, it is assigned up to $x\%$ overload. Subsequently, its overload factor is calculated as follows:

$$OL_{r_i} = OL_b + \frac{x}{100} \qquad (1)$$

Generally, the $x\%$ overload is determined by the cloud provider during the DC planning strategy. This overload percentage is affected by DC type (server room, small, mid-size, large, etc), CPU, network, storage, memory, and power modeling in the corresponding DC [51], [52], and [53].

Due to the above energy challenges and motivations, this chapter uses carbon footprint and

OPEX as assessment metrics of DCs. However, the assessment phase is not only bounded to green and cost metrics, it can be extended to other objectives based on the capabilities and choices of the cloud providers.

3) Evaluation Criteria Extraction: The envisioning process is integrated with the assessment phase, and the suitable criterion is generated accordingly. For instance, if the cloud user requires HA-aware deployments for interdependent application components while taking into consideration energy efficiency, the evaluation criterion will have low, medium, and high carbon footprint options. Then the overload factors of the DCs are evaluated. Also, an evaluation criterion can be a combination of multiple features/preferences where each feature/preference is scrutinized in the cloud infrastructure.

While the proposed SCPN model handles the availability requirements, the scoring selection tool uses OPEX and carbon footprint as criteria for its evaluation/selection process. The tool can also use other evaluation criteria depending on the preferences defined by the cloud providers and users.

3.4 Approach

To address the challenges of HA, cost, and green-aware scheduling discussed in the previous sections, we need first to elaborate a behavioral model that can capture the stochastic nature of different failures in a system and then associate it with an energy- and cost-aware scoring selection tool.

The Unified Modeling Language (UML) can reflect the service availability features, but as a semi-formal model, it cannot simulate the behavior of the system or measure the availability of a service while different stochastic failures are happening. On the other hand, Stochastic Petri Nets are behavioral models that have proven to be suitable to model and simulate the cloud system with stochastic and deterministic behaviors. Creating the SPN model manually can be a tedious, time-consuming, and error prone task. To mitigate this complexity, our approach is based on mapping an instance of the UML model describing a given deployment of the cloud application to the corresponding Stochastic Colored Petri Net model. Then this approach analyzes this model using TimeNET, a SCPN simulation tool, to quantify the expected availability of the application.

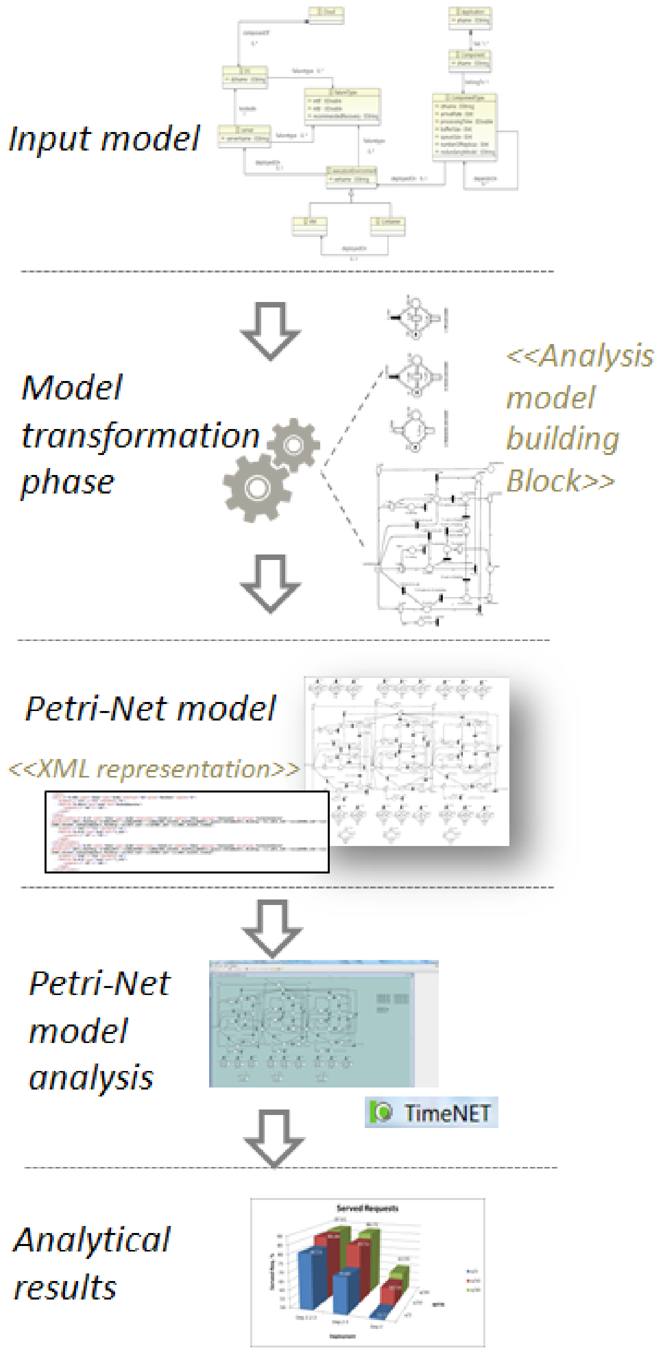


Figure 3.4: The overall SCPN approach.

Fig. 3.4 summarizes this approach.

The SCPN model is used to select best HA-aware placements while overlooking energy and cost objectives. Therefore, its results are inputted to a scoring tool to filter out deploy-

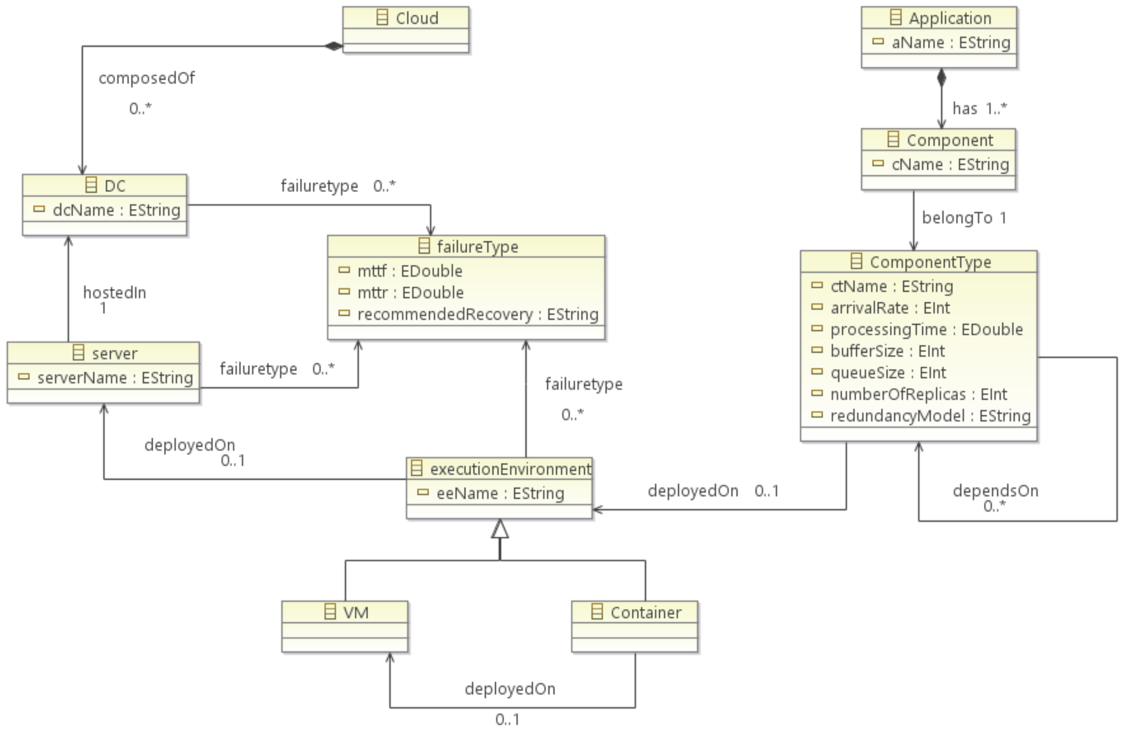


Figure 3.5: The UML model for a cloud deployment.

ments according to green and cost constraints. Once DCs are winnowed, the scoring tool selects the optimal one.

In the following, we explain the transformation from a cloud system to the corresponding SCPN model. Then we describe the evaluation criteria of the scoring selection system.

3.4.1 Cloud model

Many modeling approaches are developed to describe the heterogeneity of cloud architectures. They use different modeling frameworks in terms of general-purpose languages or domain-specific languages. For instance, OpenStack proposes Heat as an orchestration project that describes the cloud application and infrastructure, known as the stack, in Yet Another Markup Language (YAML) file called Heat Orchestration Template (HOT) [54]. With this template, Heat allows some application programming interface (API) to be used by clients to import templates to its engine. The latter parses the templates and then communicates with the necessary OpenStack services to create the specified stack and deploy its associated resources [54]. Also, HOT provides HA, auto-scaling, and failover capabilities within the created stack to allow automatic addition and destruction of VMs based on the monitored workload. Similarly, Amazon Web Services (AWS) CloudFormation uses JavaScript Object Notation (JSON) file to describe cloud architectures [55] [56] [57]. The proposed template is used to address the AWS cloud infrastructure. It consists of resource section that defines the stack properties such as Amazon Elastic Compute Cloud instance and conditions to control creation and release of stack resources. With such templates, AWS CloudFormation allows modeling and setting up of certain AWS resources while minimizing service management time and keeping track of used services for repeatability purposes.

Besides, general-purpose languages are widely used to describe cloud environment. For instance, UML can describe platform, infrastructure, and software artifacts to reflect the characteristics of different cloud component [58]. It can also specify the mapping of an application on the best cloud host according to predefined policies (HA, green, performance...).

With the generic property of UML, it is used as a template to capture a given cloud environment, and it can be easily translated to a YAML or a JSON file. A typical cloud deployment is composed of multiple software components running on an execution environment. The latter can be a VM hosted on a server or a container hosted either on a VM or on a server. In any case, the server is deployed on a data center. In the previous chapters [27] [28], we proposed a cloud-based UML model that captures a detailed description of a typical cloud system. We have modified the previous UML model to meet the requirements of the HA analysis of a given cloud deployment. Fig. 3.5 illustrates our modified UML model that captures such cloud deployment. Each application consists of multiple software components of different types. Each software component has some attributes to capture the incoming workload distribution (*arrivalRate*), the time duration required to process a request (*processingTime*), the number of requests the component can process in parallel (*bufferSize*), the maximum capacity of the requests waiting to be processed (*queueSize*), the number of redundant replicas considered for each component (*numberOfReplicas*), and the redundancy schema of the component (*redundancyModel*) to show which redundancy type a component is capable of accepting. Execution environment (VM or container), server, and DC may fail because of different failure types. Each failure type has a failure rate, a

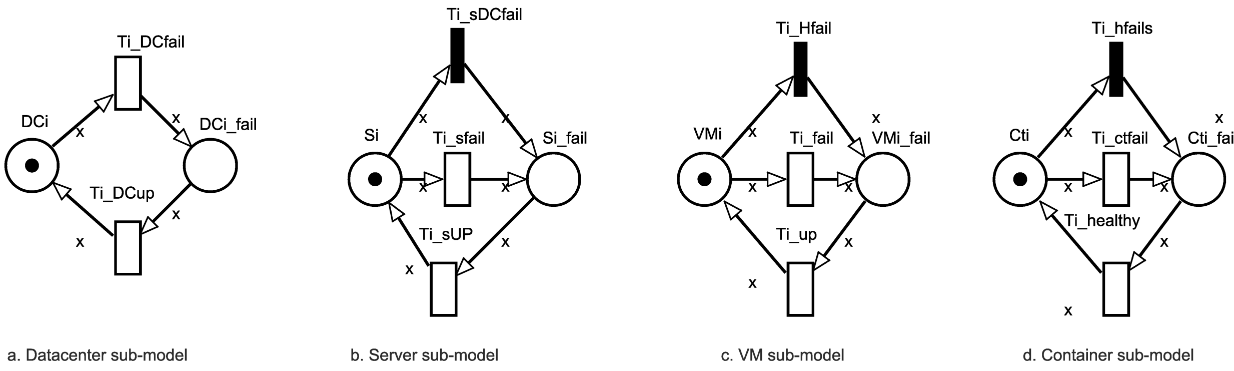


Figure 3.6: Data center, server, VM, and container sub-SCPN models.

recommended recovery action, and recovery duration based on the recommended recovery. With the transformable property of the UML model, multiple cloud deployments and profiles are generated as reusable templates to identify the mapping between cloud infrastructure and applications. Then these deployments are imported to the SCPN model and scoring system to analyze and select best HA, energy, and cost-aware deployments accordingly.

3.4.2 SCPN model building blocks

This section explains SCPN model used to evaluate various HA application deployments in a cloud environment. We define various building blocks of SCPN, which when combined form a complete SCPN model that can be analyzed to assess the expected availability. We propose six different building blocks that we use in our model transformation phase.

In our model, each of the software components can run on a virtual machine or a container. The VM is hosted on a server while the container can be hosted either on a VM or on a server. The server, in turn, is hosted on a DC. Each execution environment, server, and DC have its own recovery time (MTTR) and failure rate/MTTF. Figures provided in this chapter follow the representation of TimeNET. In TimeNET, the immediate transitions are shown as black bars while deterministic and exponential timed transitions are shown as thick white-filled bars. Note should be taken that this representation is slightly different from the standard DSPN presentation where immediate transitions are modeled with narrow bars, timed transitions are modeled with thick black-filled bars, and exponential transitions are modeled with thick white-filled bars.

1) Data center model: Fig. 3.6a shows the data center model. A data center has two states: healthy (the place DC_i) and failed (the place DC_i_fail). Failure is modeled using an exponential timed transition (T_i_DCfail) whereas the recovery is a deterministic one (T_i_DCup) [22] [29] [30]. Table 3.1 lists these transitions, their types, and time functions.

Table 3.1: Time function of DC model transitions.

Transition Name	Type	Time Function
T_i_DCfail	Exponential	EXP(dc.mttf)
T_i_DCup	Deterministic	DET(dc.mttr)

2) Server model: Fig. 3.6b presents the server model. The server also has two states: healthy (S_i) and failed (S_i_fail). The server can fail, and the failure is an exponential transition (T_i_sfail). It can also fail immediately due to the failure of its hosting data center ($T_i\_sDCfail$). We represent the data center hosting S_i with $S_{(i)DC}$. In the following, we use the place name in the formulas to show the number of the tokens available in that place. The immediate transition $T_i\_sDCfail$ is guarded with:

$$G_{T_i\_sDCfail} = (S_{(i)DC} == 0) \qquad (2)$$

The recovery occurs according to a deterministic transition (T_i_sUP). A server cannot be recovered unless its host data center is healthy. Thus, T_i_sUP is guarded with:

$$G_{T_i\_sUP} = (S_{(i)DC} == 1) \qquad (3)$$

Table 3.2 provides the information about the timed transitions of server sub-model.

Table 3.2: Time function of server model transitions.

Transition Name	Type	Time Function
T_i_sfail	Exponential	EXP(server.mttf)
T_i_sUP	Deterministic	DET(server.mttr)

3) VM model: A VM (Fig. 3.6c) can fail through an exponential transition ($T_i\text{-}fail$) or can fail immediately due to the failure of its hosting server or data center (T_i_Hfail). We refer to the server and DC hosting the VM with $VM_{(i)Server}$ and $VM_{(i)DC}$, respectively. T_i_Hfail is guarded with:

$$G_{T_i\text{-}fail} = (VM_{(i)DC} == 0 \vee VM_{(i)Server} == 0) \qquad (4)$$

The recovery happens after a deterministic delay (T_i_up). Note that in this case, also a VM cannot be recovered unless its hosting data center and server are healthy. Thus, T_i_up is guarded with:

$$G_{T_i\_up} = (VM_{(i)DC} == 1 \wedge VM_{(i)Server} == 1) \qquad (5)$$

Table 3.3 provides the information of the timed transitions of the VM sub-model.

Table 3.3: Time function of VM model transitions.

Transition Name	Type	Time Function
$T_i\text{-}fail$	Exponential	EXP(vm.mttf)
T_i_up	Deterministic	DET(vm.mttr)

4) Container model: A container (Fig. 3.6d) can fail through an exponential transition (T_i_ctfail) or can fail immediately due to the failure of its host and data center (T_i_hfails). Note that the host can be a VM or server. In any case, the failure of server causes an immediate outage of the container. We refer to the host and DC of the container with $Ct_{(i)H}$ and $Ct_{(i)DC}$, respectively where $Ct_{(i)H}$ can be $Ct_{(i)VM}$ or $Ct_{(i)Server}$. If the container is hosted on a VM then T_i_hfails is guarded with:

$$G_{T_i\_ctfail} = (Ct_{(i)DC} == 0 \vee Ct_{(i)Server} == 0 \vee Ct_{(i)VM} == 0) \qquad (6)$$

If the container is hosted on a server then T_i_hfails is guarded with:

$$G_{T_i\_ctfail} = (Ct_{(i)DC} == 0 \vee Ct_{(i)Server}) \qquad (7)$$

The recovery happens after a deterministic delay ($T_i\_healthy$). Note that in this case, also a container cannot be recovered unless its host and data center are healthy. If the container is hosted on a VM then $T_i\_healthy$ is guarded with:

$$G_{T_i\_healthy} = (Ct_{(i)DC} == 1 \wedge Ct_{(i)Server} == 1 \wedge Ct_{(i)VM} == 1) \qquad (8)$$

If the container is hosted on a server then $T_i\_healthy$ is guarded with:

$$G_{T_i\_healthy} = (Ct_{(i)DC} == 1 \wedge Ct_{(i)Server} == 1) \qquad (9)$$

Table 3.4 provides the information of the timed transitions of the container sub-model.

Table 3.4: Time function of container model transitions.

Transition Name	Type	Time Function
T_i_ctfail	Exponential	EXP(container.mttf)
$T_i\_healthy$	Deterministic	DET(container.mttr)

5) Load Balancer model: The load balancer distributes traffic among multiple compute instances. It is an effective way to maintain the availability of a given cloud system. It provides fault tolerance policy in a given application deployment [59] [60]. Upon failure of some instances, load balancer seamlessly replaces them while maintaining the normal operation of other nodes/instances. Amazon EC2 uses the concept of elastic load balancing to provide HA among its Availability Zones (AZs) [59] [61].

Fig. 3.7 illustrates the load distributor and round robin load balancer sub-model. The place *LoadDistributor* has a fixed number of tokens, and the load balancer transitions (T_LB_i and T_LB_0) distribute the workload among the active replicas of the same component.

Each component has a queue place (C_i_queue) to represent the number of requests it can queue for processing and a flushing place ($C_i\_flushing$). The transitions T_LB_i and

The flushing place is a place holder for the load balancing mechanism to ensure a round robin distribution, and thus it is not used to capture a specific component behavior.

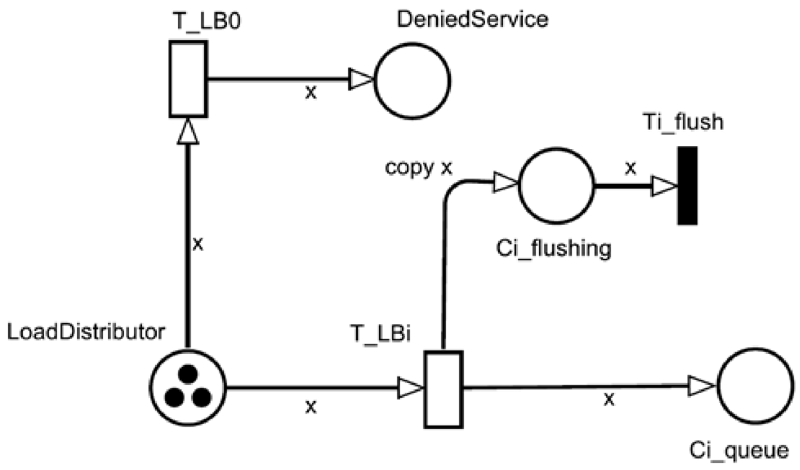


Figure 3.7: Load balancer SCPN model.

T_i_flush are guarded such that they model a round robin policy. When a component C_i receives a token in its queue, its flushing place is marked, and the component will not receive another token until its flushing place is unmarked. Let the round robin order be $C_1, C_2, C_3, ...C_M$ where *M* is the number of replicas $(numberOfReplicas)$, and then the same order repeats. The transition T_LB_1 is the first one that becomes enabled, and its clock starts elapsing. Once it is fired, one token is produced in C_1_queue, and one token is produced in $C_1\_flushing$. As long as $C_1\_flushing$ is marked, C_1 cannot receive another token. On the other hand, T_1_flush cannot be fired until all other components have received their share. As soon as C_1 receives a token, the transition T_LB_2 becomes enabled, and its clock starts elapsing. Then, T_LB_2 fires, and C_2_queue and $C_2\_flushing$ receive a token. The same way other components receive their share until C_M receives a token. At this time, T_1_flush is enabled, and $C_1\_flushing$ is unmarked. Subsequently, $T_2\_flush, T_3\_flush, ...T_M\_flush$ also fire. According to the nature of workload arrival of the system, T_LB_i can have different distributions (e.g. deterministic, exponential, ...). Table 3.5 lists different timed transitions of the load balancer sub-model, their type, and time functions.

Table 3.5: Time function of Load balancer model transitions.

Transition Name	Type	Time Function
T_LB_0	Deterministic*	DET(comp.arrivalRate)
T_LB_i	Deterministic*	DET(comp.arrivalRate)

*: Depending on the nature of the workload arrival, these transitions can have other time functions.

Note that if a component is not available due to a full queue or a component failure, VM/container failure, server failure, or data center failure, it should give its turn to the next available component. If the execution environment is VM, then for M being the number of replicas, L being the maximum capacity of a component queue $(queueSize)$, $VM_{(i)Server}$ and $VM_{(i)DC}$ being the host server and DC of VM_i, we define $VSD_{H(i)}$ and $VSD_{F(i)}$ as follows:

$$VSD_{H(i)} = [VM_i == 1 \wedge VM_{(i)Server} == 1 \wedge VM_{(i)DC} == 1] \tag{10}$$

$$VSD_{F(i)} = [VM_i == 0 \vee VM_{(i)Server} == 0 \vee VM_{(i)DC} == 0] \tag{11}$$

If the execution environment is container, then for M being the number of replicas, L being the maximum capacity of a component queue $(queueSize)$, $Ct_{(i)H}$ and $Ct_{(i)DC}$ being the host VM/server and DC of Ct_i, we define $VSD_{H(i)}$ and $VSD_{F(i)}$ becomes as follows:

$$VSD_{H(i)} = [Ct_i == 1 \wedge Ct_{(i)H} == 1 \wedge Ct_{(i)DC} == 1] \tag{12}$$

$$VSD_{F(i)} = [Ct_i == 0 \vee Ct(i)H == 0 \vee Ct(i)DC == 0] \tag{13}$$

T_LB_i is guarded with $G_{T\_LB_i}$:

$$\forall_{i \in 1:M} G_{T\_LB_i} = (C_i\_flushing == 0 \wedge VSD_{H(i)} \wedge C_i\_queue < L)$$

$$\bigwedge_{k=1:i-1} (C_k\_flushing == 1 \vee VSD_{F(k)}) \bigwedge_{j=i+1:M} (C_j\_flushing == 0 \vee VSD_{F(j)}) \tag{14}$$

And T_i_flush is guarded with G_{Ti_flush}:

$$\forall_{i\in 1:M} G_{T_i\_flush} = \bigwedge_{j=1:i-1} (C_j\_flushing == 0 \vee VSD_{F(j)})$$

$$\bigwedge_{k=i+1:M} (C_k\_flushing == 1 \vee VSD_{F(k)}) \tag{15}$$

If all the components fail or their queues are full, the requests are dropped and sent to the place *DeniedService*. If the execution environment is VM, the Transition T_LB_0 is guarded with:

$$G_{T\_LB_0} = \bigwedge_{i=1:M} (VM_i == 0 \vee VM_{(i)Server} == 0 \vee VM_{(i)DC} == 0 \vee C_i\_queue \geq L) \tag{16}$$

An alternative solution to model the load distribution is to use the loop back arcs from T_LB_i and T_LB_0 to the place *LoadDistributor* to continuously re-enable the load balancer transitions and regenerate the workload infinitely. Note should be taken that with this alternative approach of load distributing, we can run into the issue of over-flooding the model with tokens if the generation rate of the tokens (representing the arrival rate of requests) is faster than the consumption rate of the tokens (representing the processing rate of the requests).

To avoid this issue, we fix the number of tokens in the place *LoadDistributor* and do not consider the feedback input arcs. The transitions and their guards remain the same to model the round robin policy. We include both techniques in the chapter so that the reader can select the one that best fits their simulation needs.

6) Component model: Fig. 3.8 illustrates the model of a component including partially the load balancer delivering the workload to the component. In the following, the execution environment is VM. Each component has a queue (C_i_queue) to model the maximum capacity of the requests waiting to be processed and also a buffer to model the maximum number of requests a component can process in parallel ($C_i\_processing$), such as multi-threaded components. The requests stored in the queue can enter the buffer only if the component, its corresponding server, and VM are healthy, and the number of tokens already in the buffer is below the maximum. When a component fails, all the requests in its

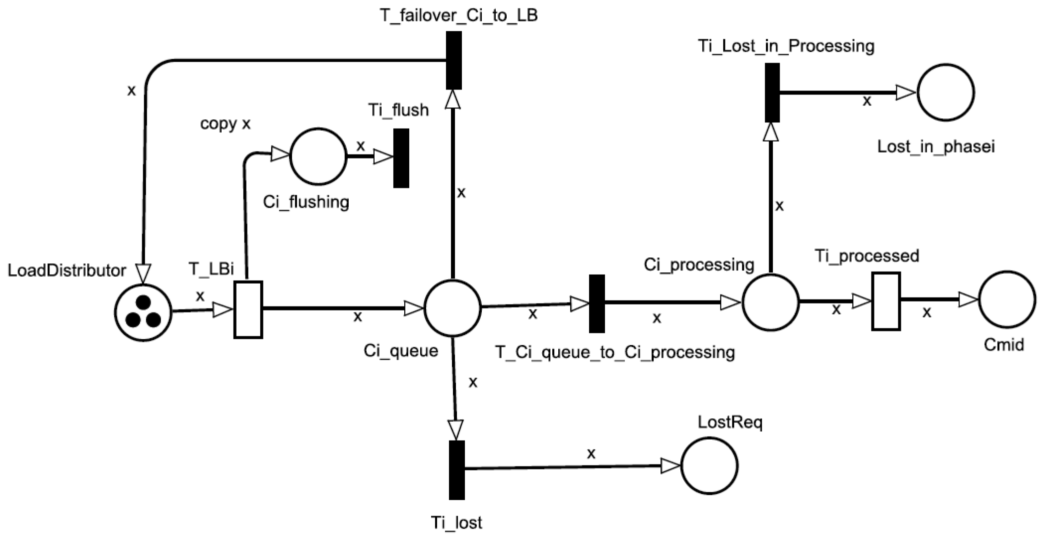


Figure 3.8: Component SCPN model.

buffer are lost and transferred to the place $Lost\_in\_phase_i$ where *'i'* is the tier number. The transition $T_i\_Lost\_in\_Processing$ is guarded with:

$$G_{T_i\_Lost\_in\_Processing} = ((VM_{(i)} == 0) \vee (VM_{(i)Server} == 0) \vee (VM_{(i)DC} == 0)) \quad (17)$$

In addition, in each tier, if all the replicas fail at the same time, all the tokens stored in the component queue are transferred to the place *LostReq*. The transition T_i_Lost is guarded with:

$$G_{T_i\_Lost} = \bigwedge_{i=1:M} (VM_{(i)} == 0 \vee VM_{(i)Server} == 0 \vee VM_{(i)DC} == 0) \qquad (18)$$

When a component fails, the requests already stored in its queue are transferred again to the load distributor to be failed over to the other healthy components. This behavior simulates a multi-active stateful redundancy where each component is equally backed up by the other components. The transition $T\_failover\_C_i\_to\_LB$ is guarded with:

$$\forall_{i \in 1:M} G_{T\_failover\_C_i\_to\_LB} =$$

$$(VM_{(i)} == 0 \vee VM_{(i)Server} == 0 \vee VM_{(i)DC} == 0) \wedge$$

$$\bigvee_{j=1:i-1} (VM_{(j)} == 1 \wedge VM_{(j)Server} == 1 \wedge VM_{(j)DC} == 1)$$

$$\bigvee_{k=i+1:M} (VM_{(k)} == 1 \wedge VM_{(k)Server} == 1 \wedge VM_{(k)DC} == 1) \qquad (19)$$

The tokens successfully processed are stored in the place *Cmid*. Note that in a multi-tier system, the tokens successfully processed in one tier are carried to the next tier where they are load balanced among the replicas of the next tier. The tokens successfully processed in all the tiers are stored in a final place. The availability of the system is only determined by those tokens that reach this final place. Table 3.6 presents the list of timed transitions and their information.

Table 3.6: Time function of component model transitions.

Transition Name	Type	Time Function
$T_i\_processed$	Deterministic	DET(comp.processingTime)
T_LB_i	Deterministic*	DET(comp.arrivalRate)

*: Depending on the nature of the workload arrival, this transition can have other time functions.

3.4.3 Transformation of UML object diagram to SCPN model

Our approach is based on transforming an instance of the UML model (i.e. an object model) into a solvable SCPN model. For example, an instance of the UML model can be an AWS cloud web application described in [59] [60]. The AWS web application is a three-tier model with a web server at the frontend forwarding traffic to an App server. At the backend, Structured Query Language (SQL) databases store user information and act as a common repository for content discovery. Each software component is backed up by a redundant component to enable fault tolerant and HA policies [59] [60]. Then the object UML model consists of three-tier component blocks, six VM blocks (two for each tier), six server blocks (one for each VM) and two DC blocks (one for active components and one for redundant ones). Each of cloud application and infrastructure is associated with its availability metrics (MTTF, MTTR...). Although we explain the approach based on a UML input model, the solution is extensible to other object-oriented models or any other configuration snippets/templates such as OpenStack HOT, AWS CloudFormation application template, or OpenNebula VM template [62] [63]. The given input model/template captures the availability attributes of a cloud deployment such as MTTR, MTTF, and other attributes described in the UML model. Having the required attributes in a given configuration script, the transformation algorithm is applied to generate the SCPN building blocks.

The overall transformation algorithm is described in the flowchart shown in Fig. 3.9. The algorithm starts by building a dependency graph based on the component types interdependencies. At this stage, we identify the number of tiers and their orders. Next, the algorithm creates the places and transitions that are common in all SCPN models, such as the *Load-Distributor*, *LostReq*, and *DeniedService* places. Then, the algorithm iterates over each tier creating the load balancer, all the component replicas, their VMs/containers, and their corresponding servers. This is based on the building blocks defined in the previous subsection. For instance, if the model includes five VMs, the VM building block is replicated five times. However, the transition and guards of each building blocks may be different. Then, in the final stage, the DCs are created, the transitions are annotated with the proper rates, and the guards are annotated with the corresponding conditions. It is the annotation phase that glues the model together reflecting the actual deployment and the failure cascading effects.

3.4.4 Deployment scoring selection system

Once the solvable SCPN model is inputted into TimeNET, multiple deployments of application components are evaluated in terms of HA and functionality constraints. In some cases, multiple HA-aware deployments are eligible for certain HA baseline. Therefore, a scoring selection system is required to choose the optimal deployment according to a given policy (green-aware and/or cost-efficiency). The proposed scoring selection system consists of evaluation criteria with multiple options and a scoring methodology.

1) Evaluation Criteria: Multiple measures can be used as evaluation criteria of the scoring system [64]. The SCPN model evaluates the deployments of application components in order to maximize the availability while taking into account functionality constraints, interdependency and redundancy of application components. Therefore, the above measures can be eliminated from the evaluation criteria.

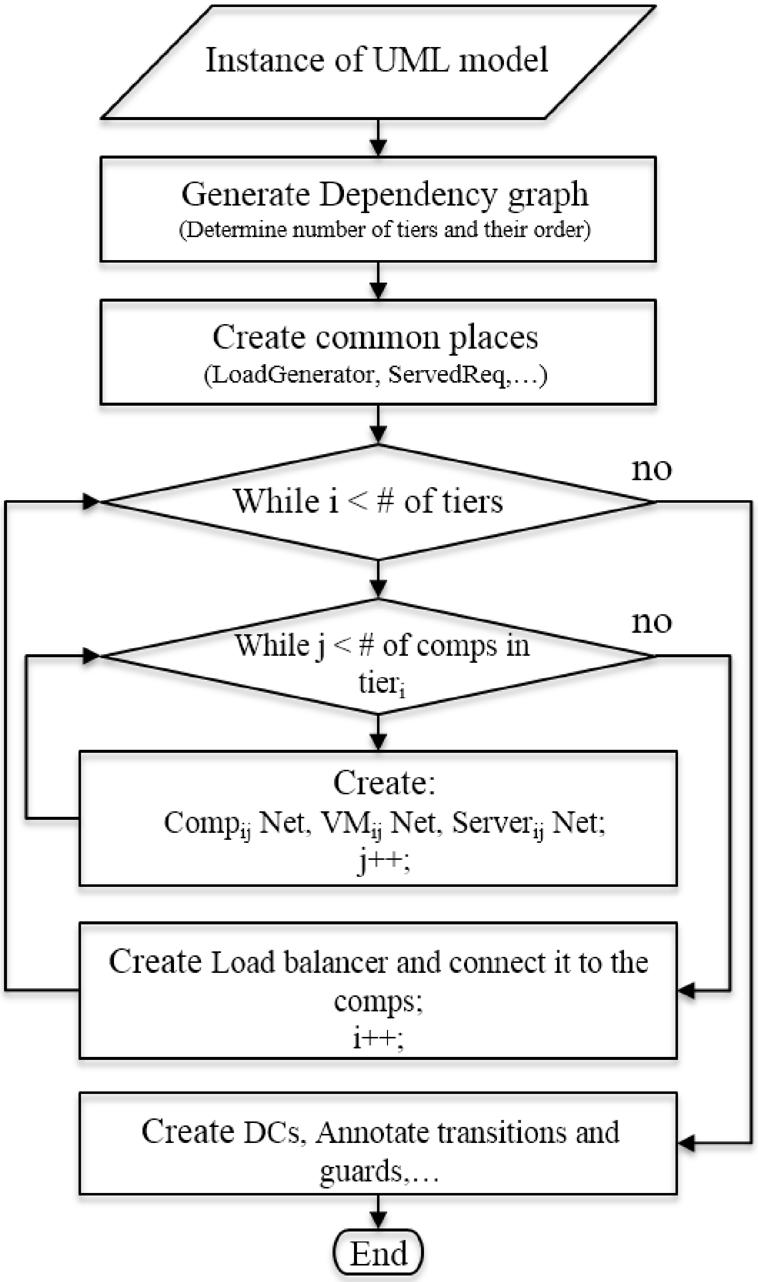

Instance of UML model
Generate Dependency graph
(Determine number of tiers and their order)
Create common places
(LoadGenerator, ServedReq,…)
While i < # of tiers
no
While j < # of comps in tier$_i$
no
Create:
Comp$_{ij}$ Net, VM$_{ij}$ Net, Server$_{ij}$ Net;
j++;
Create Load balancer and connect it to the comps;
i++;
Create DCs, Annotate transitions and guards,…
End

Figure 3.9: Transformation algorithm to generate the SCPN model.

In order to inject cost and green objectives into the proposed approach, the evaluation criterion assesses the cloud infrastructure in terms of OPEX and carbon footprint. During the assessment process, each DC is examined, and its overload factor is calculated subsequently. For a given OPEX or carbon footprint baseline, or a combination of both, the examined DC

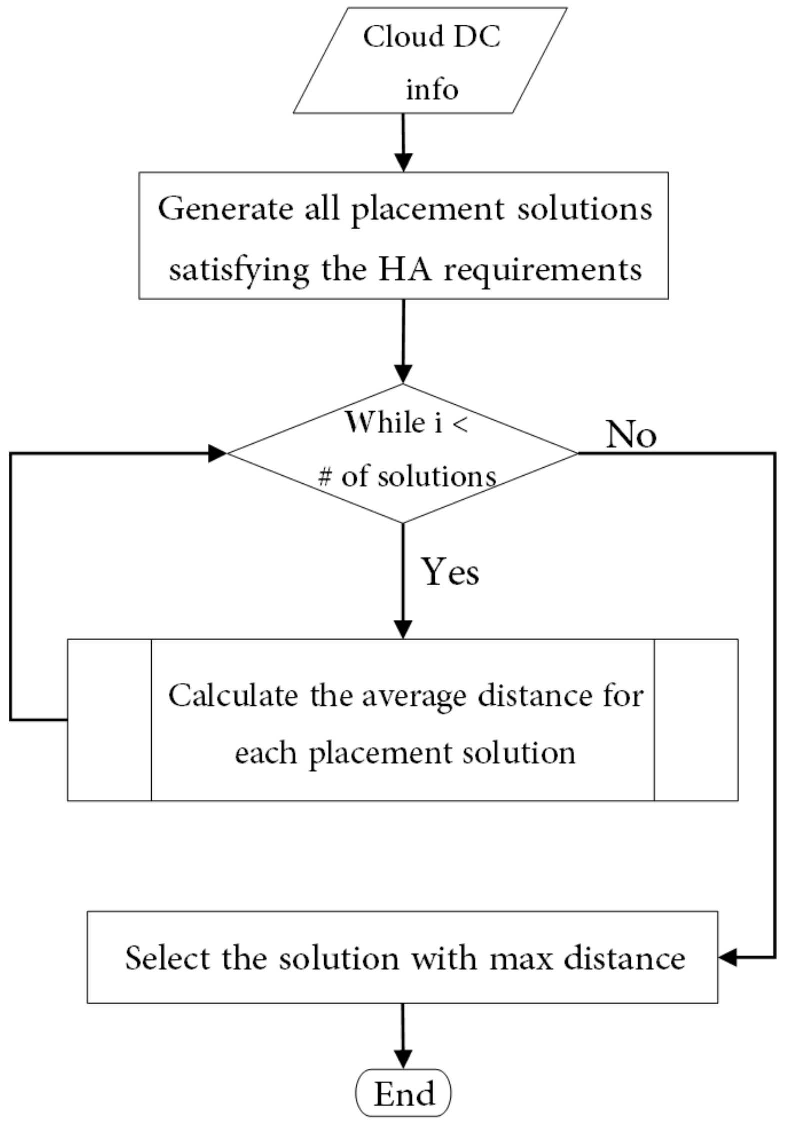

Cloud DC info
Generate all placement solutions satisfying the HA requirements
While i < # of solutions
No
Yes
Calculate the average distance for each placement solution
Select the solution with max distance
End

Figure 3.10: Scoring selection algorithm.

operates at a higher load factor, the overload factor, compared to default/baseline DC [37] [65]. This increase in the load factor gives preference for one DC over the others.

2) Scoring Methodology: Once the evaluation criterion is determined, and the SCPN model evaluates the components deployments, the scoring methodology is used to select the optimal one. The scoring selection algorithm is depicted in Fig. 3.10. Each DC is characterized by a distance metric that represents its available capacity before reaching the allowed load. Also, each deployment is characterized by a distance attribute that refers to its corresponding DCs' distances. The algorithm defines a scoring system that allows the selection of the optimal deployment. For the initial deployment, a default preference is defined as the baseline. For subsequent deployments, the algorithm evaluates each eligible deployment distance and selects the one offering the largest distance.

Let *NumDC* be the total number of available DCs and CL_i the current load of corresponding DC_i, then the relative average utilization (*RU*) of DC_i is calculated as follows:

$$\underset{i \in 1:NumDC}{\forall} DC_i.RU_i = \frac{(\sum_{j=1:NumDC|j \neq i} DC_j.CL_j)}{(NumDC - 1)} \tag{20}$$

Let *OL* be the overload factor of each DC, the maximum allowed workload (*AL*) is calculated as follows:

$$\underset{i \in 1:NumDC}{\forall} DC_i.AL_i = DC_i.RU_i \times DC_i.OL_i \tag{21}$$

Then the distance (*dist*) for each DC is calculated as follows:

$$\underset{i \in 1:NumDC}{\forall} DC_i.dist_i = DC_i.AL_i - DC_i.CL_i \tag{22}$$

Suppose *Dep* is the set of DCs used in a deployment, and *DepN* is the number of elements in the set *Dep*. Then for every eligible deployment, its distance (*Deployment.dist*) is calculated as follows:

$$Deployment.dist = \frac{(\sum_{\forall i \in Dep} DC_i.dist_i)}{DepN} \tag{23}$$

Then the eligible deployment that corresponds to the maximum deployment distance is chosen as the optimal solution. The maximum distance is the measure used to capture the imbalance between the examined DCs and the preferences of cloud providers (e.g., low OPEX or carbon footprint).

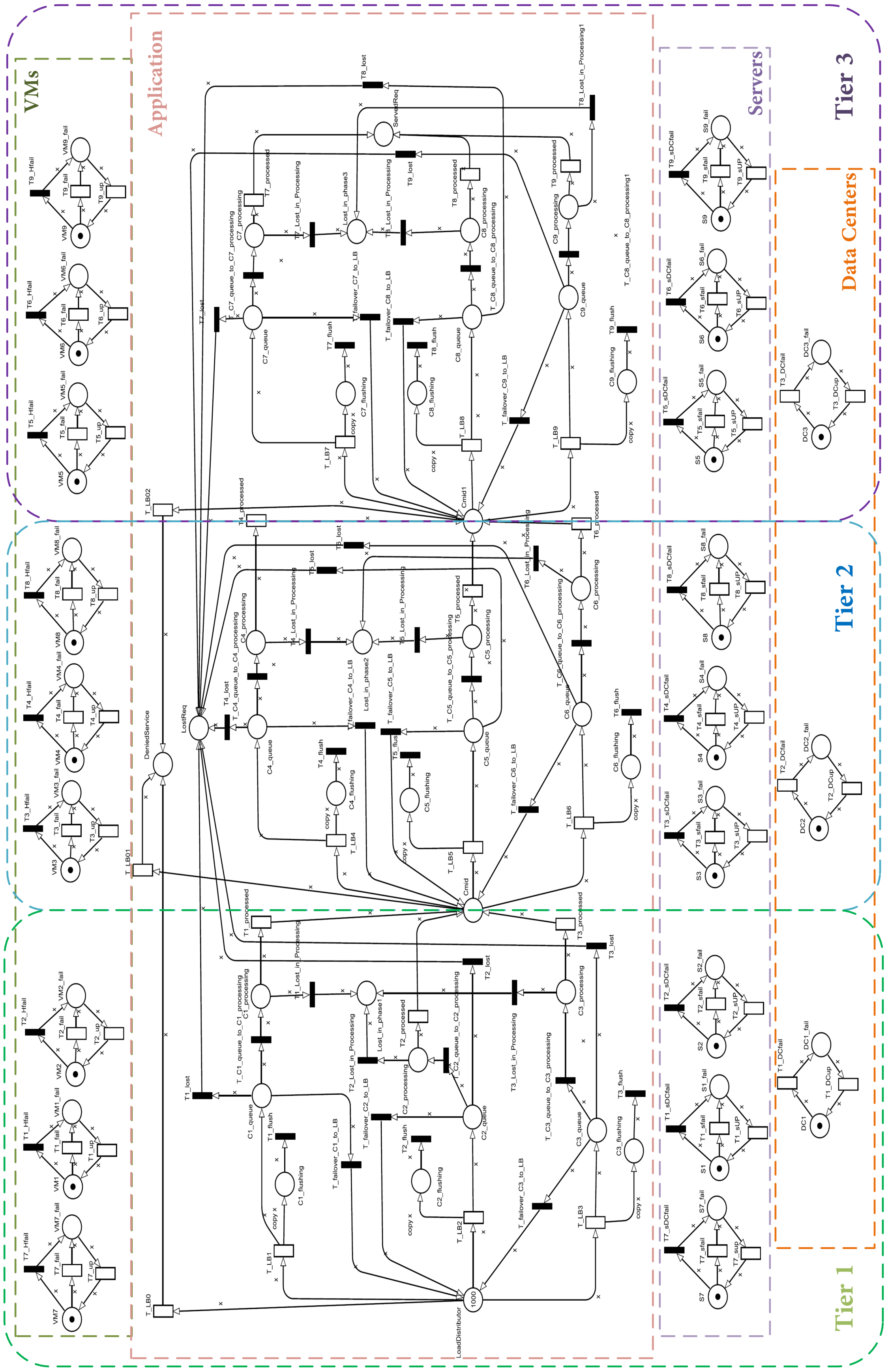


Figure 3.11: SCPN model of a three-tier Amazon web application running in a cloud environment.

3.5 Case Study

This section provides an example of a cloud deployment modeled by SCPN, and then the model is used to evaluate different deployments from HA perspective. The generated HA-aware deployments undergo a green and cost-aware filtering phase where the optimal one is selected using the proposed scoring tool. In this case study, we are particularly interested to compare inter- and intra-DC scheduling, and we change alternatively the data center hosting the servers and VMs. We assume that each VM_i is hosted on the server S_i. The data center hosting S_i is not fixed and depending on our deployment, the server can be hosted on any of the available DCs. We refer to the data center hosting VM_i and S_i using $VM_{(i)DC}$.

The system under study is a three-tier web application. At the frontend, the load balancer distributes the requests to the *Web Servers* that handle these requests and forward them to the *App Servers*. The latter handles the application operations between cloud user and backend *DBs* that store user content. In each tier, the software component is running on a virtual machine, and the VM is hosted on a server. The server, in turn, is hosted on a DC. Each tier is replicated three times using an active redundancy model. In each tier, an elastic load balancer distributes the workload among the replicas based on a round robin policy. Fig. 3.11 illustrates a snapshot of the SCPN model of this system. The depicted model is using only VMs as an execution environment, but it can be easily modified to include containers. In the latter case, the container sub-model and guards defined in Subsection 3.4.2 can be added to the SCPN model to perform availability analysis and quantification. Note that if the HA metrics of a given cloud application deployments are available, the SCPN model can be used to analyze their corresponding availability. Amazon Web application deployed using AWS Elastic Beanstalk [56] [66] can be a case study example for the proposed SCPN model.

Analyzing the service availability can be done either by (1) quantifying the percentage of time a given service is in a healthy state, or (2) by analyzing the percentage of served requests in comparison to the total number of received requests. We used the latter technique; therefore, we have fixed the number of tokens in the initial *LoadDistributor* place. Note that when we create the model from the blocks already mentioned in Subsection 3.4.2, some places may overlap. For example, the place '$Lost\_in\_phase_i$' is shared in each tier among the replicas whereas the place *'LostReq'* is unique per model. In each tier, served requests are stored in a place, which serves as the load distributor of the next tier (e.g. $Cmid$ and $Cmid_1$ places in Fig. 3.11). The tokens successfully processed in all of the three tiers are stored in the place *ServedReq* in the 3^{rd} tier. The percentage of the requests that are successfully processed through the three tiers (*ServedReq*) indicates the service availability of the cloud application. If all the components fail, or their queues are full, the requests are dropped and sent to the place *DeniedService*. When a component fails, the requests already stored in its queue are resent to the load distributor to be failed over to the other healthy components. $Lost\_in\_phase_1$, $Lost\_in\_phase_2$, and $Lost\_in\_phase_3$ collect in each phase the lost requests from the components buffers. If all the replicas of a tier fail at the same time, all the tokens waiting in the components queues are transferred to the place *LostReq*.

Table 3.7: Different MTTF, MTTR, and processing time.

MTTF(DC_1; DC_2; DC_3)	x*;x;x	x;1.5x;2x	x;2x;3x
MTTR	x/3	x/10	x/30
Load Processing Time	a*	5a	10a

*: 'x' is the failure rate of DC_1 and 'a' is the request arrival rate in each tier.

3.5.1 SCPN evaluation and results

To investigate different DC scheduling, we have considered multiple scenarios and conducted some experiments with the SCPN model. The VMs and servers can fail due to DC failure through immediate transitions $T_i\_sDCfail$ and T_i_Hfail. The failure rates of VMs and servers (used in T_i_fail and T_i_sfail) are fixed throughout these experiments. We consider that DCs can have similar or different failure rates. As a baseline, they all have the same MTTF (x; x; x). Then we modify the failure rate of the DCs assuming that DC_1 fails more frequently, DC_3 is always the most reliable one, and DC_2 has a failure rate between the two others. Then, we consider different MTTR for each variation of the MTTF. However, recovery time is always the same among the DCs. Table 3.7 shows different parameters altered in our experiments.

We have considered three deployments: the first deployment maximizes the distribution

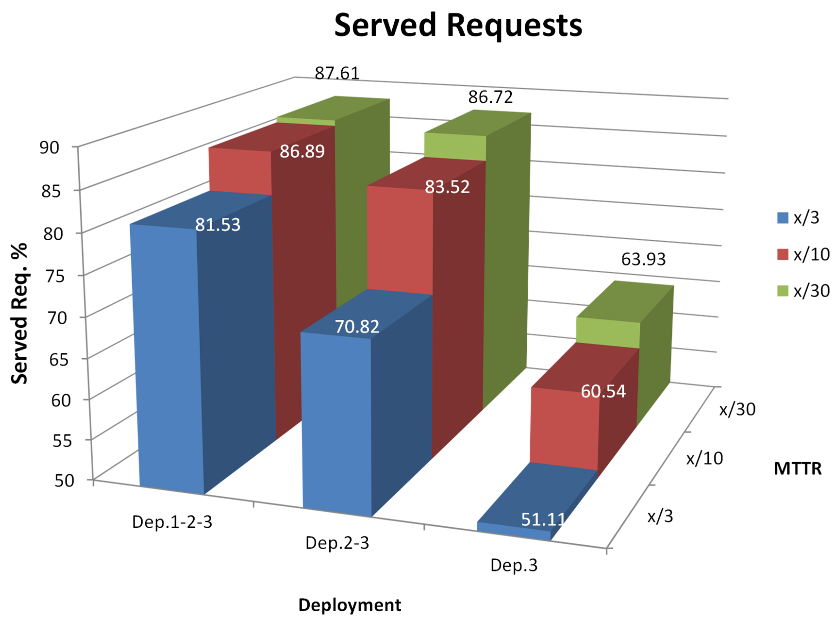


Figure 3.12: Service availability of different deployments and different MTTRs. DCs have similar MTTF.

among the DCs, such that in each tier at least one of the replicas is on DC_1, one is on DC_2, and one is on DC_3 (named Dep.1-2-3). The DCs are distributed using Amazon DCs distribution [67]. The latter are geographically distributed in the AWS Cloud that has 33 availability zones in 12 worldwide geographic regions [56] [67]. In our case, we have assumed that DC_1, DC_2, and DC_3 are located in Virginia, Oregon, and California respectively [67]. In the second deployment, we put one replica of each tier on DC_2 and two other replicas of each tier on DC_3 (called Dep. 2-3). In the third deployment, all the replicas are hosted by the most reliable DC, which is DC_3 (Dep.3 afterward). We aim to evaluate which of the three deployments would maximize the availability of the application. If DC_3 is the most reliable one, is it better to choose the third deployment and put all of the replicas on the most reliable DC or is it better to maximize the distribution among the DCs? The model presented in Fig. 3.11 is analyzed with a transient simulation of the TimeNET4.2 running on a Linux VM with 225GB of RAM and 20 vCPUs running Ubuntu12.04. The results presented in this chapter are the outcome of multiple repetitions of the simulation. First, we consider the case where all of the DCs have the same MTTF (x; x; x), and we vary the MTTR among DCs as presented in Table 3.7. Since we choose the values of MTTR as

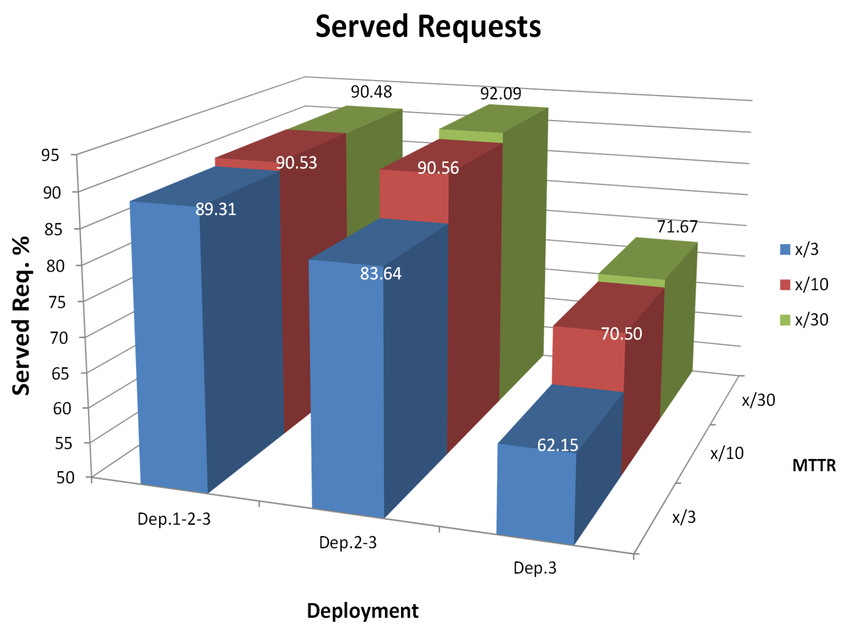


Figure 3.13: Service availability of different deployments and different MTTRs. DCs have different MTTF (x; 1.5x; 2x).

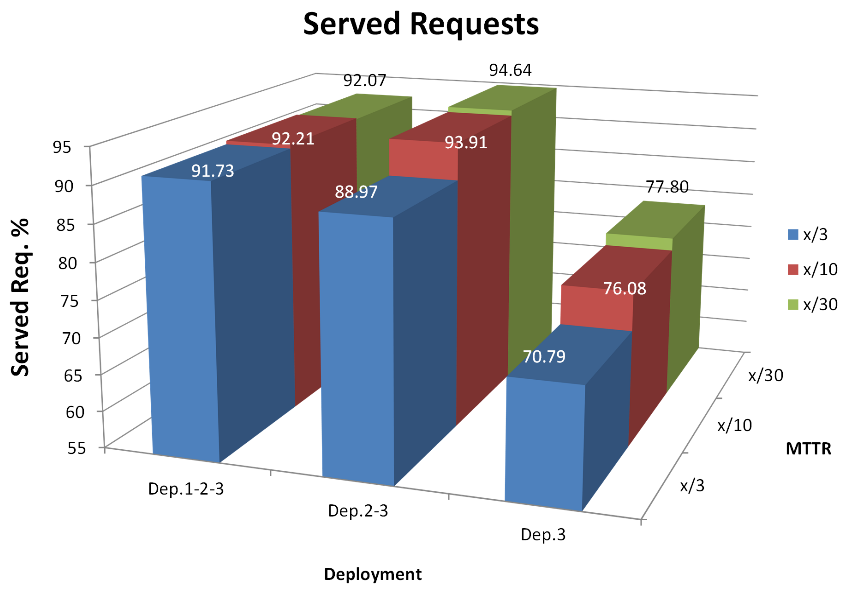


Figure 3.14: Service availability of different deployments and different MTTRs. DCs have different MTTF (x; 2x; 3x).

a ratio of the MTTF, ‘x’ is instantiated to maintain the MTTR within the allowed downtime for cloud providers [66] [68]. Fig. 3.12 depicts the corresponding results for the three de-

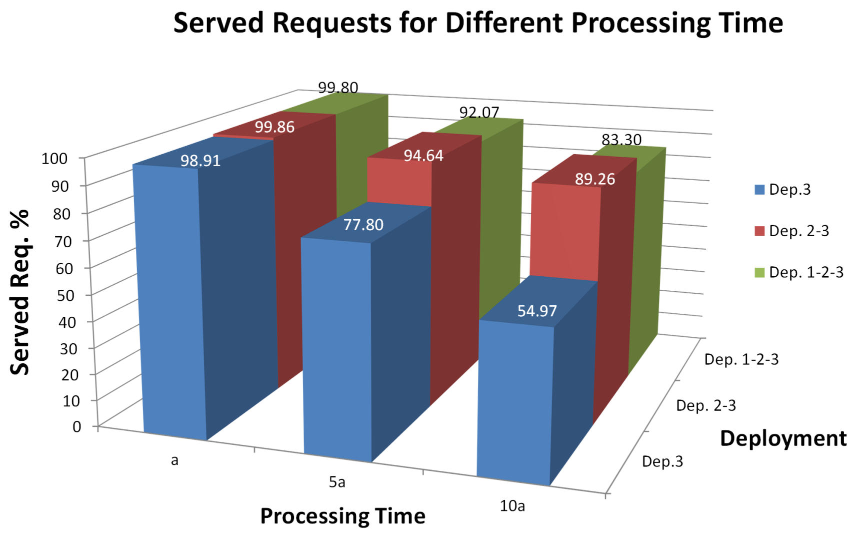


Figure 3.15: Served requests for different processing time. Request arrival rate is 'a'.

ployments mentioned above. When the DCs have the same failure rates, we should go for a maximum distribution as it reduces the probability of the service outage due to multi-DC failures.

In the second step, we change the failure rates of DC_1, DC_2, and DC_3 to x, 1.5x, and 2x, respectively and change the recovery time as listed in Table 3.7. Fig. 3.13 presents the results. Finally, we consider the case where DCs have different MTTF of x, 2x, and 3x, respectively. Again, we vary the MTTR according to Table 3.7. The results are presented in Fig. 3.14. Based on the results of Fig. 3.13 and Fig. 3.14, when the reliability of DCs differs, we can opt for the most reliable ones instead of maximum distribution. A single DC deployment is not the optimal choice.

For the last set of experiments, we investigate the impact of changing the load processing time. We assume that DC_1, DC_2 and DC_3 have different MTTF of (x; 2x; 3x), respectively. Let 'a' be the request arrival rate, we have experimented with three load processing times of 'a', '5a' and '10a'. The results are given in Fig. 3.15. The processing time affects the length of the processing queue. An increased processing time reduces the system availability due to the requests failed during processing. On the other hand, by decreasing the processing time, we reduce the impact of failures and therefore reduce the difference in HA between the intra- and inter-DC deployments.

The proposed SCPN approach is a framework verifying which scheduling options among different placement possibilities can meet the required level of availability. It provides HA-aware scheduling guidelines. The inferred keys can be applied to small- or large-scale scheduling scenarios. Note that solving a model may take time some hours due to the complicated stochastic analysis.

3.5.2 Scoring selection system evaluation and results

To select the optimal deployment, the scoring selection algorithm is applied to the above SCPN evaluation results. Since we focus in this chapter on the DC failures impact on HA, the evaluation criterion is applied to DCs. Two cases are presented to evaluate the selected deployments against different policies. In the first case, the criterion is OPEX and carbon footprint while in the second case only carbon footprint is considered.

The scoring selection algorithm is applied to the above SCPN evaluation cases: (same MTTF, different MTTR), (different MTTF and MTTR), and (different ‘a’) using Dep.1-2-3, Dep.2-3, and Dep.3 deployments. We aim to select the best deployment if multiple eligible ones are chosen by the SCPN model.

Table 3.8: DC evaluation metrics of the first case.

DC	OPEX option (%)	Carbon footprint option (%)	OL (%)	OL factor	CL (%)
DC_1	medium	none	20	1.2	42
DC_2	none	low	10	1.1	41
DC_3	none	none	0	1.0	40

1) First scoring case: In this case, each DC is examined in terms of OPEX and carbon footprint, and its corresponding overload factor is generated. Table 3.8 shows an example of metrics that characterize each DC, such as current load (CL), overload factor (OL), OPEX, and carbon footprint improvement options. The option can be either high, medium, low, or none where “high” represents a high improvement in OPEX or carbon footprint reduction, and “none” reflects the opposite state.

Table 3.9: DC distances of the first case.

DC	RU(%)	AL(%)	dist(%)
DC_1	40.5	48.6	6.6
DC_2	41	45.1	4.1
DC_3	41.5	41.5	1.5

Table 3.9 shows the calculated relative utilization (RU), allowed load (AL), and distance $(dist)$ for each DC using (18)-(20). Using values of Table 3.9 and (21), the deployment distances are calculated for each of evaluated placements as shown in Table 3.10.

Table 3.10: Deployment distances of the first case.

Dep	Deployment Distances
Dep.1-2-3.dist	4.06
Dep.2-3.dist	2.8
Dep.3.dist	1.5

The scoring selection algorithm is applied to the three cases introduced in Subsection 3.5.1. The results are shown in Table 3.11. In the first case (same DCs MTTF, different DCs MTTR), Dep.1-2-3 and Dep.2-3 are the eligible solutions for MTTF of (x) and MTTR of (x/10 and x/30) if the desired HA-baseline is greater than 80%. In the second case, Dep.1-2-3 and Dep.2-3 are the eligible solutions for MTTF of (x, 2x, and 3x) and MTTR of (x/3, x/10, and x/30) if the desired HA baseline is greater than 80%. In the third case, if the desired HA baseline is greater than 80%, Dep.1-2-3 and Dep.2-3 are the eligible solutions for ('5a', and '10a'), and Dep.1-2-3, Dep.2-3, and Dep.3 are eligible for 'a'. Since three deployments are eligible, the algorithm is applied to: Dep.1-2-3, Dep.2-3, and Dep.3 in the three cases. Once the eligible solutions are selected, the scoring algorithm calculates the (RU), (AL), and, $(dist)$ for each DC. With these parameters, the $Deployment.dist$ is calculated, and consequently, Dep.1-2-3 is the optimal deployment since it has maximum distance compared to the others.

Table 3.11: Optimal deployments of the first case.

Dep	HA-baseline $\geq 80\%$
Eligible *Dep(s)*	Dep.1-2-3, Dep.2-3, and Dep.3
Optimal *Dep*	Dep.1-2-3

If the desired HA baseline is greater than 87%, the first case generates one eligible solution, Dep.1-2-3 for MTTF of (x) and MTTR of (x/3). With the same HA-baseline applied to the second case, Dep.1-2-3 and Dep.2-3 are the best placements for MTTF of (x, 2x, 3x) and MTTR of (x/10 and x/30). As for the last case, Dep.1-2-3, Dep.2-3, and Dep.3 are the eligible placements for ‘a’, Dep.1-2-3 and Dep.2-3 are the eligible solutions for ‘5a’, and Dep.2-3 is the only eligible deployment for ‘10a’. Therefore, the scoring algorithm is only applied to the second and the third cases where DCs have different MTTF of (x, 2x, 3x) with (‘a’, ‘5a’) request arrival rates.

Table 3.12: DC carbon metrics in 2013 used in the second case.

DC	Carbon Emission (kg/million Btu)	Carbon footprint option (%)	OL (%)	OL factor	CL (%)
DC_1	52.5	none	0	1.0	55
DC_2	35.6	medium	39	1.39	10
DC_3	51.4	low	2	1.02	25

Table 3.13: DC distances of the second case.

DC	RU(%)	AL(%)	dist(%)
DC_1	17.5	17.5	-37.5
DC_2	40	55.6	45.6
DC_3	32.5	33.15	-8.15

Table 3.14: Deployment distances of the second case.

Dep	Deployment Distances
Dep.1-2-3.dist	-0.016
Dep.2-3.dist	18.725
Dep.3.dist	-8.15

2) Second scoring case: In this case, each DC is examined in terms of carbon emission based on the U.S. energy report [69]. Table 3.12 shows the carbon emissions of industrial sectors in California, Oregon, and Virginia where the above three DCs are located [69]. Since Virginia has highest carbon emissions, its DC, DC_1, is considered the baseline one, and consequently, its overload factor (OL) is one. The deployments evaluation is based only on the carbon emission factor. Similarly, the option can be either high, medium, low, or none.
Table 3.13 and Table 3.14 show the calculated relative utilization (RU), allowed load (AL), distance $(dist)$, and deployment distances for each DC and evaluated placements using (18)-(21).

Table 3.15: Optimal deployments of the second case.

Dep	HA-baseline $\geq$ 80%
Eligible *Dep(s)*	Dep.1-2-3, Dep.2-3, and Dep.3
Optimal *Dep*	Dep.2-3

The scoring selection algorithm is applied to the three cases introduced in Subsection 3.5.1. The results are shown in Table 3.15. In the first case (same DCs MTTF, different DCs MTTR), Dep.1-2-3 and Dep.2-3 are the eligible solutions for MTTF of (x) and MTTR of (x/10 and x/30) if the desired HA-baseline is greater than 80%. In the second case, Dep.1-2-3 and Dep.2-3 are the eligible solutions for MTTF of (x, 2x, and 3x) and MTTR of (x/3, x/10, and x/30) if the desired HA baseline is greater than 80%. In the third case, if the desired HA baseline is greater than 80%, Dep.1-2-3 and Dep.2-3 are the eligible solutions for ('5a', and '10a'), and Dep.1-2-3, Dep.2-3, and Dep.3 are eligible for 'a'. Since three deployments are eligible, the algorithm is applied to: Dep.1-2-3, Dep.2-3, and Dep.3 in the three cases. The scoring algorithm calculates the (RU), (AL), and, $(dist)$ for each DC of the eligible deployments. Then, the $Deployment.dist$ is calculated, and consequently, Dep.2-3 is the optimal deployment since it has maximum distance compared to the others. Note that a change in the DC workload, its OPEX, or carbon footprint option affects the (RU), (AL), and, $(dist)$ calculation. Consequently, different deployment might win the scoring test since $Deployment.dist$ of the eligible solutions will be modified.

3.6 Related Work

Although organizations are facing a challenge in selecting the best HA solution to meet the business requirements, a few literature studies address the scheduling of cloud services and their availability and green analysis using different extensions of Petri Net models and scoring selection system.

3.6.1 Availability analysis using Petri Net models

Many cloud providers analyze the services availability using empirical data. Service dashboard provides a summary of the existing availability solutions and the status histories [68] [70]. Longo et al. propose an availability analysis approach for cloud computing systems using Stochastic Reward Net (SRN) and Markov chain models [71]. They develop multiple equations to analyze the impact of changing the number of physical machines, their MTTF, and MTTR on the services availability. Although their approach minimizes the problem-solving time and analyzes service availability in large-scale networks, the authors only focus on the MTTF and MTTR of the servers discarding the impact of those of VMs or software components. Also, the approach does not consider any redundancy or interdependency models, stochastic nature of failures, functional workflow between different components, and their impacts on the availability analysis.

Javadi et al. propose statistical models to predict the availability of a distributed system [72]. Their main objective behind this prediction is to find host subsets with related statistical characteristics and availability models. They use randomness test to determine the hosts having independent and identically distributed availability. When such hosts are identified, they are clustered into subsets with comparable availability models. Although this work tries to predict availability models of hosts in a distributed system, it ignores the impact of other factors on the availability of the system, such as the VM failure, repairing plans, redundant hosts or VMs, stochastic nature of failures, requests processing, and forwarding. Ghosh et al. develop a performance analysis model for services deployment in a cloud system using continuous time Markov chain model [73]. Their keys of interest are the service availability and response delays. In order to evaluate the service performance, the authors analyze the impact of failure rates, recovery modes, workload variation, and the available resources on the quality of service. The proposed approach shows good results in terms of the studied key metrics, but it focuses only on the infrastructure-side impact on the service availability. It neglects the effect of services failures, interdependency between them, and redundant models. Besides, it discards the impact of request processing time and load balancing on the availability of deployed services.

Paing et al. propose an approach that integrates virtualization, clustering methods, and software rejuvenation mechanisms to analyze cloud applications availability [74]. The authors use Stochastic Petri Net model where availability is expressed in terms of the stochastic failure and recovery time transitions of the model. Similarly, Nguyen et al. propose a SRN for availability analysis [75]. The used SRN considers various VM failures types, recovery methods, and interdependency among VMs, hosts, and hypervisors. The availability analysis is based on the number of lost transactions and impact of software rejuvenation. While Salfner et al. propose queuing and Stochastic Petri Net service availability models through software rejuvenation and failure prevention [76], [77], and [78], Salfner et al. propose another model that describes the impact of adding servers on service availability using Stochastic Colored Petri Net [79]. Although the proposed models show performance improvements, they only focus on few aspects of availability analysis. When it comes to applications scheduled in a cloud model, various factors affects their availability. These factors are not only associated with the existing infrastructure or the cloud user side, but they are a combination of both. Additionally, these factors are affected by failover, request processing time, and interaction between different components and their hosts.

Jiang et al. have modeled the behavior of software and hardware using Generalized Stochastic Petri Net (GSPN) while considering failure dependency between software and hardware [80]. They have shown the impact of disc redundancy on improving the system availability.

Also, they have analyzed the model to extract the suitable failure rate and recovery time to achieve the required level of availability.
Melo et al. have used SPN and Reliability Block Diagram (RBD) to show the impact of software rejuvenation as a solution to increase the high availability of the cloud systems [81]. They have considered software aging and different rejuvenation policies and determined the positive impact of live migration on the availability of the system.

3.6.2 Scoring and selection of cloud deployments

Green DC and cloud solutions are interesting concepts that have attracted several research studies. Subramanian et al. propose a cloud brokering approach that generates optimal deployments for VMs placements in multiple heterogeneous clouds [82]. According to the constraints defined in the service management index (SMI), the approach selects the best deployments with optimal cost using mixed integer linear programming model. The cloud user sends the corresponding request information and defines the requirements and their weights in the SMI. Then the approach provides a scoring system to obtain optimal deployment. Although the authors provide optimal solutions, they overlooked HA and energy requirements in their approach.
Joo et al. use Colored Petri Net model to provide workflow scheduling approach [83]. The latter uses phased scheduling scheme that separates the scheduling and the execution phase while minimizing processing cost and satisfying computational resources constraints. The authors focus on performance requirements while discarding availability metrics. Energy objective is discarded in the evaluation criteria of their scoring system.
Qian et al. propose a cloud service selection approach characterized by its automatic selection of existing infrastructures [84]. The approach focuses only on deployment costs, but it also considers interdependency between multiple applications. Their approach is also associated with a step-wise placement algorithm to consider scalability solutions. Although the proposed algorithm finds sub-optimal deployments, it does not take into account any availability requirements including redundancy and load balancing solutions. While Fan et al. describe a clustering deployment model that maximizes performance [85], Nguyen et al. provide a comprehensive availability model using stochastic reward nets (SRN) [86]. Fan et al. generate communication performance of cloud nodes, selects the initial centroid

using a density-based algorithm, and then uses a greedy algorithm to select the optimal placement [85]. In this work, the authors do not consider any HA guidelines neither applications interactions. However, Nguyen et al. use high availability configuration between sites, existing fault and disaster tolerant mechanisms and considers the interaction between different elements of cloud systems [86]. The latter proposes the SRN model to analyze downtime cost.

Ranganathan et al. propose a technique that manages the server power in an ensemble form [87]. The approach monitors the resource usage and allows active servers to hook the power from the inactive ones. The proposed technique reduces power consumption and cooling cost in data centers. Rusu et al. present a technique that determines which servers should be turned on or off in order to minimize the overall power consumption [88]. Liu et al. use VM migration to allow server consolidation and turn off underutilized servers [89]. Their approach aims to minimize the migration time and energy consumption. Therefore, it iterates over the existing servers and selects the one with low energy consumption and minimal migration overhead.

Bradley et al. propose a power management approach that minimizes the power consumption while satisfying the workload demands [90]. They use CPU utilization to predict these demands. When the utilization exceeds a certain threshold, extra servers are turned on to minimize the CPU usage of all severs. On the other hand, when CPU usage in the servers is below the given threshold, some servers are turned off. Khosravi et al. propose a VM deployment solution that minimizes carbon footprint while distributing VMs across data centers [91]. Each DC is associated with its carbon footprint rate and power usage effectiveness (PUE). Based on these parameters, the proposed energy and carbon-efficient (ECE) cloud model places VMs in the suitable DC and server.

We distinguish ourselves from the related work by proposing a Petri Net model that takes into account not only different stochastic failure types and deterministic recovery and repair plans, but it captures the impact of service load balancing and processing, application/VM failover, different redundancy models, and interdependency relations. In the proposed approach, HA-aware guidelines are provided that allow evaluating any deployment solution and select the optimal one using the scoring selection sub-approach. It considers not only

HA requirements and functionality constraints, but also adds energy and cost objectives to the scoring evaluation criteria. Starting with a certain HA-baseline, the user can end up with HA, green, and cost-aware deployments.

3.7 Conclusion

Cloud services experience various stochastic failures and consequently become unavailable. With the always on and always available trend, inoperative services halt the business continuity. It is not enough to provide HA solution that can mitigate failures and maintain certain availability baseline, but it is necessary to assess such solution and its resiliency to any failure modes. Additionally, it is essential to integrate such assessment with green and cost requirements to uphold the quality of service (QoS) with lower carbon footprints and OPEX. With these objectives, this chapter proposed a SCPN model that evaluates the availability of cloud services and their deployments in inter- or intra-DCs. This model considers different stochastic failures, deterministic repairs, functionality constraints, redundancy, and interdependencies between different applications components. Consequently, different decisions had been extracted from this model that aid in designing the best HA solution of an existing cloud model. The SCPN model inputted the HA-aware deployments into a scoring selection tool. Using the latter algorithm, HA-aware placements are filtered in terms of energy and cost metrics to select the optimal deployment. The scoring selection tool is extensible to different criteria and is not limited to the aforementioned measures.

References

[1] M. Jammal, A. Kanso, P. Heidari, and A. Shami, "Availability Analysis of Cloud Deployed Applications," *IEEE International Conference on Cloud Engineering (IC2E)*, April 2016.

[2] M. Jammal, A. Kanso, P. Heidari, and A. Shami, "A Formal Model for the Availability Analysis of Cloud Deployed Multi-Tiered Applications," *Third IEEE International Symposium on Software Defined Systems*, April 2016.

[3] M. Jammal, A. Kanso, P. Heidari, and A. Shami, "Scrutinize High Availability-aware Deployments Using Stochastic Petri Net Model and Cloud Scoring Selection Tool," *Submitted to IEEE Transactions on Services Computing*, November 2016.

[4] M. Armbrust, A. Fox, et al., "A view of cloud computing," *IEEE Communications Magazine,* vol. 53, pp. 50-58, April 2010.

[5] Microsoft Azure, `http://azure.microsoft.com/en-us/overview/what-is-azure/`. [July 22, 2015]

[6] Amazon ec2, `http://aws.amazon.com/ec2`, 2016. [April 2016]

[7] H. Hawilo, A. Kanso, and A. Shami, "Towards an Elasticity Framework for Legacy Highly Available Applications in the Cloud," *IEEE World Congress on Services (SERVICES)*, pp. 253-260, July 2015.

[8] P. Yong and H. Ning,"Research on dependability of cloud computing systems," *International Conference on Reliability, Maintainability, and Safety (ICRMS),* pp. 435-439, August 2014.

[9] NETFLIX, "AWS Re:Invent - High Availability Architecture at Netflix," `http://www.slideshare.net/adrianco/high-availability-architecture-at-netflix`, December 2012. [August 20, 2015]

[10] OpenStack, "Filter Scheduler," `http://docs.openstack.org/developer/nova/filter_scheduler.html`, 2010. [June 17, 2016]

[11] M. A. Sharkh, M. Jammal, A. Shami, A. Ouda, "Resource allocation in a network based cloud computing environment: design challenges," *IEEE Communications Magazine*, vol. 51, no. 11, pp. 46-52, November 2013.

[12] Microsoft, "The Economics of the Cloud," `http://www.microsoft.com/en-us/news/presskits/cloud/docs/theeconomics-of-the-cloud.pdf`, November 2010. [October 2015]

[13] International Standard ISO/IEC, "High-Level Petri Nets - Concepts, Definitions and Graphical Notation," *Final Committee Draft ISO/IEC 15909-1,* May 2002, `http://www.petrinets.info/docs/pnstd-4.7.1.pdf`.

[14] S. Loveland, E. M. Dow, F. LeFevre, D. Beyer, and P. F. Chan, "Leveraging virtualization to optimize high-availability system configurations," *IBM Systems Journal,* vol. 47, no. 4, pp. 591-604, 2008.

[15] B. Cully, G. Lefebvre, et al., "Remus: high availability via asynchronous virtual machine replication," *5th USENIX Symposium on Networked Systems Design and Implementation,* pp. 161-174, 2008.

[16] E. M. Farr, R. E. Harper, L. F. Spainhower, and J. Xenidis, "A case for High Availability in a virtualized environment (HAVEN)," *Third International Conference on Availability, Reliability and Security,* pp. 675-682, March 2008.

[17] D. S. Kim, F. Machida, and K. S. Trivedi, "Availability modeling and analysis of a virtualized system," *15th IEEE Pacific Rim International Symposium on Dependable Computing,* pp. 365-371, November 2009.

[18] W. E. Smith, K. S. Trivedi, L. A. Tomek, and J. Ackaret, "Availability analysis of blade server systems," *IBM Systems Journal,* vol. 47, no. 4, pp. 621-640, 2008.

[19] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN),* pp. 447-456, 2010.

[20] K. Ramo, "Eliminating Software Failures-A Literature Survey," *Licentiate Thesis,* 2009, `http://www.doria.fi/bitstream/handle/10024/61561/nbnfi-fe201005051790.pdf?sequence=3`.

[21] P. Bodik, F. Armando, M. J. Franklin, M. I. Jordan, and D.A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," *ACM Symposium on Cloud Computing,* pp. 241-252, June 2010.

[22] Q. Anderson, "Storm real-time processing cookbook: Efficiently process unbounded streams of data in real time," *Packt Publishing,* 2013.

[23] F. Machida, D. S. Kim, and K. S. Trivedi, "Modeling and analysis of software rejuvenation in a server virtualized system," *IEEE Second International Workshop on Software Aging and Rejuvenation,* pp. 1-6, November 2010.

[24] J. Xu, X. Li, Y. Zhong, and H. Zhang, "Availability modeling and analysis of a single-server virtualized system with rejuvenation," *Journal of Software,* vol. 9, no. 1, pp. 129-139, January 2014.

[25] M. T. Hla Myint and T. Thein, "Availability improvement in virtualized multiple servers with software rejuvenation and virtualization," *Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI),* pp. 156-162, June 2010.

[26] T. Thein and J. S. Park, "Availability analysis of application servers using software rejuvenation and virtualization," *Journal of Computer Science and Technology,* vol. 24, no. 2, pp. 339-346, April 2009.

[27] M. Jammal, A. Kanso, and A. Shami, "High Availability-Aware Optimization Digest for Applications Deployment in Cloud," *2015 IEEE International Conference on Communications (ICC),* pp. 6822-6828, June 2015. Available: `http://vixra.org/pdf/1410.0193v1.pdf`

[28] M. Jammal, A. Kanso, and A. Shami, "CHASE: Component High-Availability Scheduler in Cloud Computing Environment," *IEEE International Conference on Cloud Computing (CLOUD),* pp. 477-484, 2015.

[29] J. O. Grady, "System Requirements Analysis," *Elsevier,* December 2013.

[30] P. L. Gonzalez-R, J. M. Framinan, A. Dopfer, and R. Ruiz-Usano, "Optimization Customized Token-Based Production Control Systems Using Cross-Entropy," *Digital Enterprise Technology,* pp. 123-131, 2007.

[31] K. S. Trivedi, D. Kim, and R. Ghosh, "System availability assessment using stochastic models," *Applied Stochastic Models in Business and Industry,* vol. 29, no. 2, pp. 94-109, 2013.

[32] R. Ghosh, D. Kim, and K. S. Trivedi, "System resiliency quantification using non-state-space and state-space analytic models," *Reliability Engineering & System Safety,* vol. 116, pp. 109-125, 2013.

[33] C. Petri, "Kommunication mit Automaten," *University of Bonn*, 1962.

[34] G. Ciardo and C. Lindemann, "Analysis of deterministic and stochastic Petri nets," *5th International Workshop on Petri Nets and Performance Models,* pp. 160-169, 1993.

[35] N. Gharbia, C. Dutheilletb, and M. Ioualalen, "Colored Stochastic Petri Nets for modelling and analysis of multiclass retrial systems," *Math. Comput. Model.,* vol. 49, pp. 1436-1448, 2009.

[36] A. Zimmermann, "Modeling and Evaluation of Stochastic Petri Nets with TimeNET 4.1," *6th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS),* pp. 54-63, 2012.

[37] U.S. Department of Energy, "Best Practices Guide for Energy-Efficient Data Center Design," `http://energy.gov/sites/prod/files/2013/10/f3/eedatacenterbestpractices.pdf`, March 2011. [October 2015]

[38] Data Center Knowledge, "Undertaking the Challenge to Reduce the Data Center Carbon Footprint," `http://www.datacenterknowledge.com/archives/2014/12/17/undertaking-challenge-reduce-data-center-carbon-footprint/`, December 2014. [November 2015]

[39] Data Center Dynamics, "Verizon to auction its data centers report," `http://www.datacenterdynamics.com/design-strategy/verizon-to-auction-its-data-centers-report/95445.article`, January 2016. [January 2016]

[40] U.S. Environmental Protection Agency, "Report to Congress on Server and Data Center Energy Efficiency," `http://hightech.lbl.gov/documents/data_centers/epa-datacenters.pdf`, August 2007. [August 2011]

[41] Ingram Micro Advisor, "How Data Center Design Impacts Efficiency and Profitability," `http://www.ingrammicroadvisor.com/data-center/how-data-center-design-impacts-efficiency-and-profitability`, July 2015. [January 2016]

[42] International Telecommunication Union (ITU), "ITU and Climate Change," `http://www.itu.int/dms_pub/itu-s/opb/gen/S-GEN-CLIM-2008-11-PDF-E.pdf`, 2006. [November 2008]

[43] G. Koutitasa, and P. Demestichas, "A review of energy efficiency in telecommunication networks," *17th Telecommunications Forum*, November 2009.

[44] Data Center Dynamics, "Reducing data center carbon: IT efficiency is king," `http://www.datacenterdynamics.com/critical-environment/reducing-data-center-carbon-it-efficiency-is-king/80782.fullarticle`, July 2013. [December 2015]

[45] 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, no. 6, pp. 18-26, December 2014.

[46] Green Data Center News, "Local governments may force faster green data center adoption," `http://www.greendatacenternews.org/articles/840122/local-governments-may-force-faster-green-data-cent/`, January 2016. [January 2016]

[47] Data Center Knowledge, "Norway's Fjord-Cooled Data Center," `http://www.datacenterknowledge.com/archives/2011/12/20/norways-fjord-cooled-data-center/`, December 2011. [November 2015]

[48] EuroNews, "Facebook boasts green data centre in Lule, Sweden," `http://www.bloomberg.com/bw/articles/2013-10-03/facebooks-new-data-center-in-sweden-puts-the-heat-on-hardware-makers`, October 2015. [25 October 2015]

[49] Data Center Dynamics, "Arizona passes data center tax breaks," `http://www.datacenterdynamics.com/design-strategy/arizona-passes-data-center-tax-breaks/80453.fullarticle`, June 2013. [December 2015]

[50] Oracle, "Oracle's Approach To Cloud," `http://www.oracle.com/technetwork/topics/entarch/oracle-ds-cloud-approach-r3-0-1556829.pdf`, 2012. [December, 2015]

[51] S. Polfliet, F. Ryckbosch, and L. Eeckhout, "Optimizing the data center for data-centric workloads," *The international conference on Supercomputing*, pp. 182-191, 2011.

[52] C. Delimitrou and C. Kozyrakis, "Cross-Examination of Datacenter Workload Modeling Techniques," *31st International Conference on Distributed Computing Systems Workshops*, pp. 72-79, June 2011.

[53] S. Shen, V. Beek, and A. Iosup, "Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters," *5th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, pp. 465-474, May 2015.

[54] D. Michelino, "Implementation and testing of OpenStack Heat," *CERN openlab Summer Student Report*, September 2013, `https://zenodo.org/record/7571/files/CERN_openlab_report_Michelino.pdf`.

[55] Amazon Web Services, "Template Anatomy," `http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-anatomy.html`, May 2015. [June 2016]

[56] Amazon Web Services, "AWS Template Format," `https://s3-us-west-2.amazonaws.com/cloudformation-templates-us-west-2/AutoScalingMultiAZWithNotifications.template`, September 2010. [March 2016]

[57] Amazon Web Services, "AWS CloudFormation: User Guide," `http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-ug.pdf`, May 2010. [March 2016]

[58] S. Bernardi, J. Merseguer, and D. Petriu, "An UML profile for dependability analysis and modeling of software systems," *Technical Report*, May 2008, `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.205.4357&rep=rep1&type=pdf`.

[59] M. Tavis and P. Fitzsimons, “Web Application Hosting in the AWS Cloud Best Practices,” https://media.amazonwebservices.com/AWS_Web_Hosting_Best_Practices.pdf, May 2010. [September 2012]

[60] Amazon Web Services, “Web Application Hosting,” http://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_web_01.pdf, 2016. [May 2016]

[61] Amazon Web Services, “Fault Tolerance and High Avaialbility,” http://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_ftha_04.pdf, 2016. [May 2016]

[62] OpenNebula, “Virtual Machine Definition File,” http://docs.opennebula.org/4.12/user/references/template.html, 2015. [May 2016]

[63] OpenNebula, “Virtual Machine High Availability,” http://docs.opennebula.org/4.6/advanced_administration/high_availability/ftguide.html#virtual-machine-failures, 2014. [December 2015]

[64] Oracle, “Cloud Candidate Selection Tool: Guiding Cloud Adoption,” http://www.oracle.com/technetwork/topics/entarch/oracle-wp-cloud-candidate-tool-r3-0-1434931.pdf, December 2011. [November 2015]

[65] PG&E, “Data Center Best Practices Guide: Energy efficiency solutions for high-performance data centers,” http://www.pge.com/includes/docs/pdfs/mybusiness/energysavingsrebates/incentivesbyindustry/DataCenters_BestPractices.pdf, October 2012. [December 2015]

[66] A. Adegoke and E. Osimosu, “Service Availability in Cloud Computing-Threats and Best Practices,” *Bachelor Thesis*, http://www.diva-portal.se/smash/get/diva2:646329/FULLTEXT01.pdf, June 2013.

[67] Amazon Web Services, “AWS Global Infrastructure,” https://aws.amazon.com/about-aws/global-infrastructure/, 2016. [May 2016]

[68] Amazon EC2, “Amazon EC2 Service Level Agreement,” http://aws.amazon.com/ec2-sla/, June 1 2013. [July 20, 2015]

[69] U.S. Energy Information Administration, "Energy-Related Carbon Dioxide Emissions at the State Level, 2000-2013," `http://www.eia.gov/environment/emissions/state/analysis/pdf/stateanalysis.pdf`, October 2015. [April 2016]

[70] Google Cloud Platform, "Google Cloud Status," `https://status.cloud.google.com/`, July 23 2015. [August 10, 2015]

[71] F. Longo, R. Ghosh, V. Naik, and K. Trivedi, "A scalable availability model for infrastructure-as-a-service cloud," *41st IEEE/IFIP International Conference on Dependable Systems & Networks (DSN),* pp. 335-346, June 2011.

[72] B. Javadi, D. Kondo, J. Vincent, and D. Anderson, "Discovering statistical models of availability in large distributed systems: An empirical study of seti@home," *IEEE Transactions on Parallel and Distributed Systems,* vol. 22, no. 11, pp. 1896-1903, November 2011.

[73] R. Ghosh, K. S. Trivedi, V. K. Naik, and D. S. Kim, "End-to-end performability analysis for infrastructure-as-a-service cloud: An interacting stochastic models approach," *16th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC),* pp. 125-132, December 2010.

[74] A. M. Paing and N. L. Thein, "A Petri Net Model for High Availability in Virtualized Local Disaster Recovery," *International Conference on Information Communication and Management,* vol. 16, pp. 158-164, 2011.

[75] T. A. Nguyen, D. S. Kim, and J. S. Park, "A Comprehensive Availability Modeling and Analysis of a Virtualized Servers System Using Stochastic Reward Nets," *The Scientific World Journal,* August 2014.

[76] F. Salfner and K. Wolter, "Analysis of service availability for time-triggered rejuvenation policies," *Journal of Systems and Software,* vol. 83, no. 9, pp. 1579-1590, May 2010.

[77] F. Salfner and K. Wolter, "Service Availability of Systems with Failure Prevention," *IEEE Asia-Pacific Services Computing Conference,* pp. 1219-1224, December 2008.

[78] F. Salfner and K. Wolter, "A queuing model for service availability of systems with rejuvenation," *IEEE International Conference on Software Reliability Engineering Workshops (ISSRE),* pp. 1-5, 11-14 November 2008.

[79] F. Salfner and K. Wolter, "A Petri Net model for Service Availability in Redundant Computing Systems," *Winter Simulation Conference (WSC),* pp. 819-826, December 2009.

[80] S. Jian, W. Shaoping, and S. Yaoxing "Petri-Nets Based Availability Model of Fault-Tolerant Server System," *IEEE Conference on Robotics, Automation, and Mechatronics,* pp. 444-449, 2008.

[81] M. Melo, P. Maciel, J. Araujo, R. Matos, and C. Araujo, "Availability study on cloud computing environments: Live migration as a rejuvenation mechanism," *43rd IEEE/IFIP International Conference on Dependable Systems and Networks,* pp. 1-6, 2013.

[82] T. Subramanian and N. Savarimuthu, "Application based brokering algorithm for optimal resource provisioning in multiple heterogeneous clouds," *Vietnam Journal of Computer Science*, pp. 1-14, December 2015.

[83] K. Joo, S.H. Kim, D. Kim, and C.H. Youn, "Cost-Aware Workflow Scheduling Scheme Based on Colored Petri-net Model in Cloud," *International Conference on Future Web*, November 2014.

[84] H. Qian, H. Zu, C. Cao, and Q. Wang, "CSS: Facilitate the cloud service selection in IaaS platforms," *International Conference on Collaboration Technologies and Systems (CTS)*, pp. 347-354, May 2013.

[85] P. Fan, J. Wang, Z. Chen, Z. Zheng, and M. R. Lyu, "A spectral clustering-based optimal deployment method for scientific application in cloud computing," *International Journal of Web and Grid Services*, vol. 8, pp. 31-55, 2012.

[86] T. A. Nguyen, D. S. Kim, and J. S. Park, "Availability modeling and analysis of a data center for disaster tolerance," *Future Generation Computer Systems*, vol. 56, pp. 27-50, October 2015.

[87] P. Ranganathan, P. Leech, D. Irwin, and J. Chase, "Ensemble-level power management for dense blade servers," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 66-77, 2006.

[88] C. Rusu, A. Ferreira, C. Scordino, and A. Watson, "Energy-efficient real-time heterogeneous server clusters," *12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 418-428, 2006.

[89] L. Liu, H. Wang, et al., "GreenCloud: a new architecture for green data center," *6th international ACM on Autonomic computing and communications industry session*, pp. 29-38, 2009.

[90] D. J. Bradley, R. E. Harper, and S. W. Hunter, "Workload-based power management for parallel computer systems," *IBM Journal of Research and Development*, vol. 47, pp. 703-718, 2003.

[91] A. Khosravi, S. K. Garg, and R. Buyya, "Energy and Carbon-Efficient Placement of Virtual Machines in Distributed Cloud Data Centers," *19th International Conference on Parallel Processing*, pp. 317-328, 2013.

Chapter 4
Mitigating the Risk of Cloud Services Downtime Using Live Migration and High Availability-Aware Placement

4.1 Introduction

Cloud computing emerges as a distributed platform that provides on-demand compute, storage, software systems, or applications as a service [2]. Using virtualization and elastic computing resources, the cloud aims to transform everything from file management, desktop replacement to the information technology (IT) infrastructure into a service-driven platform [3]. The cloud hosts different enterprise and real-time applications that are deployed on virtual machines (VMs) or containers. However, planned and unplanned outages are bound to occur and thus shutting down data-sensitive, critical or, any cloud application [4] [5]. This emanates availability concerns regarding the adoption of the cloud application. A highly available system should adopt design principles that prevent service loss by managing failures and minimizing/avoiding downtime especially for critical applications such as health, life, or economical-based services. The service level agreement (SLA) offered by the cloud provider considers only the availability of the resources while discarding that of the tenant's applications using these resources. Therefore, it is crucial from a tenant perspective to have the ability to assess the expected availability of its business critical applications [6]. Using the proper assessment, a better high availability (HA) plan can be associated with the cloud services to handle any unplanned outage and system interruptions

The content of this chapter have been published [1].

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC-STPGP 447230) and Ericsson Research.

of a single instance all the way to the servers and data centers (DCs).

Different strategies can be used to maintain certain HA baseline and gear the "always-on" applications. The objective of this chapter is to propose a HA-aware solution that embraces HA-aware placement and live migration strategies to achieve HA in the cloud and then evaluate these approaches not only from HA perspective but also to assess the performance metrics associated with the proposed solution. This chapter provides guidelines to design and evaluate HA-aware placements of cloud applications. The placements are evaluated using the Stochastic Colored Petri Net (SCPN) model designed earlier [7] [8]. Our evaluation metric is based on the number of served, lost, and delayed requests. The assessment stage is necessary to ensure the effectiveness of the deployments, their resiliency to failures or sudden changes in the workload and consequently, maximizing the HA of the VM and its hosted applications. Also, the chapter proposes a live VM migration to handle the cloud infrastructure or application/VM failure, overload, or maintenance. It uses the HA-aware placement approach to find new hosts for the migrated VMs while satisfying the HA, performance, and other SLA requirements. For this purpose, the chapter provides a mixed integer linear programming model that minimizes the migration downtime based on the VM memory pages and the optimal HA-aware placement of the VM.

The rest of this chapter is organized as follows. Section 4.2 presents related work for live migration and HA-aware placement solutions. Section 4.3 describes the proposed framework where HA-aware placement and live migration are discussed. The evaluation and results of the proposed framework are defined in Section 4.4. Finally, the conclusion is presented in Section 4.5.

4.2 Related Work

Building highly available and resilient cloud system has attracted many studies. Many prior research works address the availability of the cloud applications, each from a different perspective. Some focus on live migration approaches while other tackle the HA-aware applications placement, fault-tolerant, and redundancy techniques.

Liu et al. propose a technique in order to minimize the downtime of the migration [9]. This technique performs the logging and replay strategy before the stop and copy stage. It copies the execution log of the VM instead of the modified pages. It is an iterative technique since the logged file is transferred during n rounds. The last log file is reduced to small size and copied in the stop and copy stage thus generating minimized downtime. Riteau et al. propose Shrinker, a new scheme that searches for common data (same versions of programs, shared libraries or kernels) between the migrated VM and the VMs residing on the new location [10]. They use cryptographic hash table and digest algorithms that map the data into a hash value. Whenever a common hash value is found then the pages are copied through the local network instead of the WAN. They implement two subsystems to improve the migration efficiency; site-wide distributed hash table that locates nodes having a copy of a given page using its hash value and periodic memory indexer added to the hypervisors that populate the distributed hash table.

Wood et al. propose Sandpiper, a technique that provides automated strategies for VM migration in DC [11]. It consists of a black box, an application agnostic, that migrates the VM, and a gray box that gathers statistics of the operating system and the application in order to have better migration. Sandpiper uses a hotspot algorithm that determines the suitable time to migrate, the best position for migration, and the best VM to be migrated. It implements a nucleus in each server to gain statistics about the usage profile and determine the hotspot that simulates the migration occurrence. Also, it implements a hotspot heuristic that determines which overloaded VM to migrate and its placement while minimizing the migration overhead (transferred data) and the migration time.

Keller et al. formulate the VM migration problem as a relocation problem [12]. It consists of choosing the VM to be migrated and the server where it should reside. Because the relocation problem minimizes the SLA violations, the order of migrated VMs and their servers affects the migration performance. Therefore, the authors propose different relocation policies that manipulate the orders of the candidate VMs and the target servers according to their CPU utilization.

Shrivastava et al. propose a virtual machine migration method that takes into consideration the real-time communication among VMs, the data center network (DCN) topology, and the servers' capacity (resources) [13]. This method aims to minimize the DCN traffic while satisfying all of the server-side constraints. They develop an optimization model that minimizes the overhead of the VM migration by placing the dependent couple close to each other in the data center topology. Because the proposed model is an NP-complete problem, the authors develop an approximate solution (AppAware) that places the VM one at a time on the suitable server while minimizing the mapping cost.

Bose et al. aim to maximize the number of in-migrations and minimize the number of out-migrations [14]. In-migration means that active servers are the candidates for new migration. While the out-migration switches to new servers to accommodate the migrated VMs. They develop an optimization model that minimizes the migration cost taking into account the CPU and memory resources of the new servers and the overloaded VMs. However, the authors use heuristics in order to approximate the solution in a reasonable time since real DCs contain hundreds of servers and thousands of VMs. Finally, Wei et al. develop LVCMI model; live VM migration with less cost and application interference [15]. They propose a cost migration model that chooses the best VM to be migrated. The cost model depends on the performance degradation that faces the user. Also, they propose an interference model that generates an optimal placement of the migrated VM and minimizes the relocation interference. Their work is implemented in Xen, and they implement a VM monitor in each VM and PM monitor in each server to collect information about the state of each VM and physical machine (PM).

Wu et al. estimate the migration time based on the resource allocation and management strategies for a certain virtualized data center [16]. They control the CPU usage during migration by providing the Xen with the ability to assign a specific amount of CPU to the Dom0. Controlling the resources of Dom0 limits the CPU usage of each VM. They conclude that the availability of the CPU cycles and network bandwidth are important factors for live VM migration. These two parameters are highly correlated during VM migration. In other words, it is unnecessary to take into consideration the CPU and network bandwidth availability to implement performance model for live VM migration.

Machida et al. propose a redundancy-aware approach where a minimum number of VMs is generated to maintain the service availability [17]. Al-Omari et al. and Zhu et al. achieve a fault tolerant design using a backup technique [18] [19]. The latter schedules the same task in different processors to tackle failure and task execution.

Feller et al. consolidate the workload of different VMs and infrastructure clusters to provide a fault-tolerant system [20]. Wang and Xu et al. propose an approach that adds fault

handlers into the fault-tolerant system [21] [22]. The approach examines faults and randomly generates others to update the system design.
While Jung et al. allocate multi-tier applications to maintain HA and service performance [23], Zhong et al. use non-linear modeling to achieve application availability [24]. Zhong et al. find the best placements for the replicas [24]. The placement approach takes into consideration the impact of task failures on the execution of other correlated tasks.

4.3 Approach

High availability is a challenge that many enterprises are endeavoring to achieve. It can be attained by designing a system that can handle different workload and cloud failures while maximizing the service uptime. The chapter proposes a HA-aware approach that is a function of HA-aware placement and live migration while considering the component frequency of failure and its associated impact, the interactions of the applications components, and the required computational resources and latency. In the following, different design considerations are proposed to achieve and assess HA-aware placements. Then the proposed placement is used in the live migration approach to find hosts for the migrated VMs.

4.3.1 HA-aware placement

Cloud users are considered an important entity of any cloud application. They are the main drivers of quality of service and policy makers where the performance of any cloud solution depends on the impact it has on its users. Cloud users can be classified into four categories: developers, authors, experts, and end users [25]. The developers are responsible for the development, administration, and maintenance of the cloud applications. The authors provide services to be integrated into the workflows. The experts provision services resources and allow interfacing with end-users. Finally, the end users require service provisioning in a highly available mode. Therefore, it is the responsibility of cloud users to provide a HA-aware placement for their cloud applications. The cloud provider offers different SPI (Software, Platform, and Infrastructure as a Service) models and simplifies their complexity as a foundation to help the cloud users choose and design their HA approach. For instance, Netflix is one of the cloud users that maintains its application availability (e.g.

Eureka: elastic load balancing and failover tool) while using the Amazon infrastructure as a service [26].

In this section, we explain the design considerations to architect a HA-aware placement solution.

4.3.1.1 System modeling

The first step toward an HA architecture is conceptualizing and building the cloud using system modeling such as Unified Modeling Language (UML) class diagrams. The modeling step aims at defining the point of failures, faulty nodes, and different HA and performance metrics. Furthermore, it allows a better understanding of the cloud applications interactions such as dependency and redundancy relations.

The cloud consists of a cloud provider side that has multiple data centers (DCs) hosting many servers and a cloud user side consisting of multiple applications components. Every module of the cloud is assumed to fail at some point, and consequently, each is associated with its mean time to failure (MTTF) and mean time to repair (MTTR). Note that MTTF is the expected time until the first failure and MTTR is the time to repair a failed module. Each application component has one or many redundant instances to avoid a single point of failures. These components can be deployed with different redundancy models. If the components are deployed in an active-active manner, the primary and redundant components process requests/data in parallel. If the components are deployed in an active-hot-standby, both primary and redundant are up where only the primary can process requests. Also, the components can be deployed in an active-cold-standby manner. In this case, the redundant components are powered off until the failure of the primary component.

In this scope, we address the placement of 3-tier web application that handles HTTPS requests at the frontend and forwards them to the business logic or App server, which accesses the database at the backend and returns an HTML response. Dividing the applications into multiple separate tiers results in cloud applications that meet the economy of scale. The 3-tier web application allows the generation of different application layers where each is responsible for part of the application processing. It is necessary to take into consideration the interdependency relation between the application components during the placement process to ensure the successful completion of requests and consequently, to maximize the availability of multiple computational paths. Such relation requires placements that minimize both the component-interaction latency and the impact of failures of the sponsor components.

4.3.1.2 Placement algorithm

Amazon provides certain availability baseline using the multiple availability zones. It implements redundancy approach to minimize downtime and provides three nines of HA. Similarly, Microsoft azure service provides three nines of HA by defining an availability set for a VM where each set has an update domain and a fault domain to determine which VMs and hosts can be restarted and which VMs share same network switch and power source respectively. Although these cloud platforms provide certain HA baselines, it is not always the case that the inter-applications distribution can achieve a certain HA baseline [7]. These platforms discard the impact of applications placement, interaction between redundant and interdependent components, and other availability and performance requirements on the HA plan.

The HA-aware placement aims to deliver cloud services while minimizing their downtime [27] [28]. The cloud applications and infrastructure model are used to extract the placement objective and associated constraints. For this purpose, a mixed integer linear programming optimization model can be used to generate optimal HA-aware deployments. This model maximizes the applications availability while satisfying the computational resources of each component and the latency requirements between redundant and interdependent components. The computational complexity of such mathematical models hinders their applicability on large scale networks. Therefore, it is necessary to extend this mathematical modeling to an approximate solution with the same objective and constraints and apply it to real cloud settings.

Once the information is collected, the placement algorithm is triggered to find the best HA and performance-aware hosts for the applications. The proposed HA-aware algorithms consist of different filters to achieve its objective. The resources and latency filters find a set of hosts that satisfy the functionality requirements. The availability filter aims at minimizing both the frequency and impact of failures [6] [27] [28]. The availability filter does not only select servers with high MTTF and low MTTR, but it selects the servers that satisfy redundancy and dependency constraints as well. To avoid single point of failures, redundant components should not be hosted on the same server if such policy does not vio-

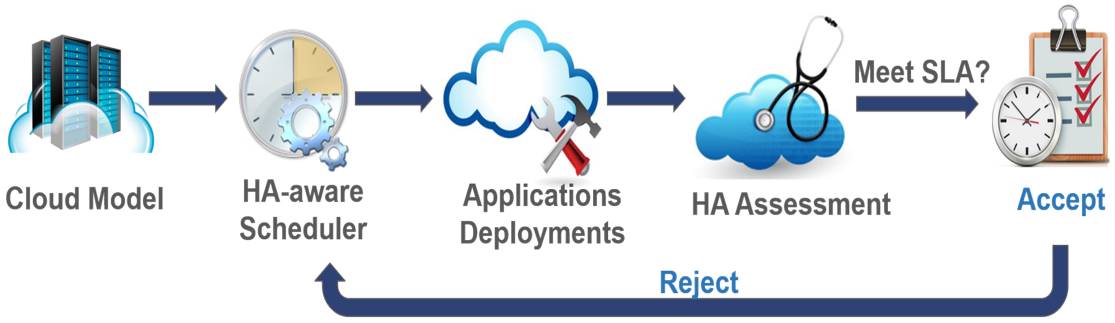


Figure 4.1: HA-aware deployments analysis approach.

late functionality constraints. Similarly, the dependent components do not share its sponsor host unless they cannot tolerate its failure.

Generally, cloud applications should expect the failure of some cloud entities. This outage should not hinder the functionality and service delivery in the cloud. It is necessary to inject failures and examine the applications resiliency to them. Therefore, SCPN model is used to evaluate the above HA-aware deployments.

4.3.1.3 Deployments dependability analysis

Unplanned outages are the result of stochastic failure events, and consequently, the cloud providers and users are not aware of it. Stochastic failures come in different forms and occur at different cloud levels: infrastructure failures, applications errors, or cascaded failures [7] [8]. Although the HA-aware placement satisfies the availability and functionality requirements, it is necessary to model the different stochastic failures of the cloud entities and evaluate the effectiveness of the HA-aware placement approach and its resiliency to failures. Once the HA-aware deployments are generated, it is necessary to perform a dependability analysis to assess the system availability. Dependability analysis aims at answering the questions related to the availability of a given system, such as cloud applications availability. Using this analysis, we can determine if a component can tolerate failures, if the cloud service can be provided during a specific time period, and the time needed to recover the faulty entity upon a failure. Different analysis techniques can be used to assess a system availability.

In order to find the technique that satisfies the system analysis objective, a comparative analysis between multiple approaches is needed. The comparison is done in terms of the techniques types, aims, and supported characteristics. Regarding the types, some techniques are performed at the beginning of the design stage to identify the potential single point of failures and their impacts at the next system levels, such as Functional Failure Analysis (FFA) and Preliminary Hazard Analysis (PHA). Alternatively, other techniques are performed after the system designing process to evaluate its reliability/availability and provide guidelines for design alternatives. They aim at analyzing the effect of multiple concurrent failures, such as Markov analysis, Petri Nets, and Truth Tables. Regarding the analysis objective, it can be quantitative, qualitative, or both. Besides, it is necessary to determine the capability of modeling a dependent behavior where a failure of an entity might affect other dependent ones.

In the case of HA-aware deployments, the availability evaluation should model the components deployments, interactions between redundant and dependent components, assess their deployments, and improve them in terms of availability constraints if applicable. Also, the analysis is not only performed to identify failures, their impacts, and repair/recovery policies, but it aims at estimating a numerical data as an availability metric under stochastic or probabilistic failure assumptions as well. For this purpose, Petri Nets can be used to evaluate HA-aware deployments and assess their resiliency to failures. They are expressive and flexible models that represent the interactions between different entities, including the firing of events. They support qualitative, quantitative, and dependency modeling analysis. In our previous work, we developed a SCPN model to serve the above objectives [7] [8]. Such model assesses the HA-aware deployments. Briefly, upon the arrival of requests, the load balancing event is triggered to distribute them across multiple components: the primary and its redundant ones. When the failure event is fired, failover is triggered and the load balancer redistributes the workload among the redundant component(s). Once a given deployments scenario is analyzed, the SCPN model generates the availability of that scenario in terms of the successful requests number.

The analysis model does not only evaluate cloud system availability, but it provides guidelines to improve the deployments as well. In other words, it allows graceful degradation to be considered in the cloud where a failure can still occur, but the cloud system is functional and maintains its service delivery. The assessment approach is shown in Fig. 4.1.

The proposed HA-aware placement is used in the following live migration approach to find hosts for the migrated VMs while satisfying availability and performance requirements.

4.3.2 Live VM migration

Live migration is the movement of VM from its original server to another one without any disconnection in its activity. During live VM migration, the whole VM, its memory, registers, storage, and operating system, are transferred from one host to another. It finds a new host for the faulty or overloaded virtual machine while minimizing its downtime. It can be applied in one DC or across DCs to maintain certain availability and performance baselines. Live migration can be performed during different situations, such as:

- Migrating a VM host system for maintenance reasons such as, hardware maintenance, software upgrade, or firmware update.
- Exploiting specific server resources or characteristics without violating the availability and functionality requirements of the application.
- Balancing server workloads due to a sudden change in their VMs' load, an abrupt grouping of VMs with common resources demands or certain dependency relation.
- Optimizing resource usage in order to have a green cloud network.
- Pre-disaster recovery process that requires evacuation or shutting down certain DC and thus migrating VMs before the disaster happens.

4.3.2.1 Live migration mechanism

The evaluation of a live migration mechanism depends on two metrics: the migration time and the downtime. The migration time is the required time to migrate the VM from one server to another. On the other hand, the downtime is a fraction of the migration time in which the VM is halted [29]. There are different mechanisms for live migration that conciliate the above two factors.

1. Post-copy migration mechanism: It transfers the content of the memory after migrating its process state to new destination [30] [31]. The pages of the memory that are modified in the original server are known as the demand-paged over the network. However, in post copy technique, the transfer of the memory pages happens at most once. It uses dynamic self-ballooning mechanism to handle the migration of the pages of the VM. It gets the pages from the original host using following steps:

- Demand-Paging: It ensures that the pages are transferred once over the network.

- Active Push: It removes the dependency of the VM from the source server.
- Pre-Paging: It ensures that pages are not duplicated during the migration process
- Dynamic Self-Ballooning: It handles free memory pages.

Although this technique is easily implemented and minimizes the total migration time, but it is characterized by high downtime.

2. Pure-on-demand migration mechanism: The VM is paused only to copy the essential data, and its remaining address space is transferred when it is activated on the new server [30]. This technique has low downtime, but it ends up with a high migration time.

3. Pre-copy migration mechanism: This mechanism provides a compromise between the above techniques. It addresses their shortcomings by adding an iterative pre-copy stage before starting the stop and copy step [30]. In general, the pre-copy technique selects a target VM to be migrated, finds a new host for it, performs an iterative transfer of the VM memory pages, and finally stops the VM for a final transfer iteration. The following steps summarize the pre-copy technique:

- Pre-Migration: This step selects a target VM to be migrated. It then executes the placement algorithm to select the destination host for the VM or to find candidate destination hosts that satisfy the VMs performance and other QoS requirements.
- Reservation: This step reserves the resources of the VM at the destination host.
- Iterative Pre-Copy: This performs an iterative transferring of the dirtied pages to the destination.
- Stop-and-Copy: The running operating system (OS) is suspended at the source host, and its network traffic are transferred to the target host. The CPU state and the remaining memory pages are transferred. At the end of this step, an image of the VM resides on both the source and target hosts.
- Commitment: It provides an acknowledgment from the destination indicating a successful migration of the VM.
- Activation: It activates the VM on the target destination.

4.3.2.2 Live migration approach

Due to stochastic failures and sudden increase in the computational demands of some applications, migration scheme is an inevitable step that mitigates the computational resources and outage burdens.

Live migration aims at minimizing the downtime of the migrated VM. The downtime depends on the number of pages transferred during the iterative and stop-copy stages, applications placements, and transmission delay. Once a dynamic pool of VMs requires migration, the above HA-aware placement is triggered to generate new deployments while satisfying the availability and functionality constraints. Then it is evaluated using the SCPN model. If the deployments do not meet the SLA, the placement is re-executed. Note that if the new destination does not satisfy the latency requirements between the interdependent and redundant VMs, a new set should be migrated. In other words, the placement algorithm generates new migration unit.

When the new hosts are selected, the iterative copy stage is activated. It is considered iterative because any application can cause a modification of its memory pages in a regular manner. Once a page is modified, it should be recopied to the new server. It is noticeable that the migration time is affected not only by the HA-aware deployments overhead but by the pages dirty rate as well. Consequently, the migration approach minimizes the migration downtime by moving the pages with high dirty rate during the iterative stage and those with low rates during the stop-and-copy stage.

Ultimately, this objective is achieved using a mixed integer linear programming (MILP) optimization model that embraces the above requirements and considerations to minimize migration downtime.

4.3.2.3 Optimization model

The MILP model is solved using ILOG CPLEX optimization tool. Its objective function minimizes the downtime of the migrated VM $(Downtime_v)$ while satisfying HA, functionality, and other migration constraints.

a) Notations and decision variables: In this model, VM is denoted as V, a server is denoted as S, computational resources are denoted as Res, redundant and dependent VMs are denoted as Red and Dep respectively. Also, the tolerance time, recovery time, and delay tolerance of a VM are denoted as TT, RT, and DT respectively. P and $P^{threshold}$ represent the probability and its threshold to modify the VM memory pages. Memory pages with high dirty rates are denoted Pg^h. The iterations during the pre-copy iterative stage are denoted as $iter$. Finally, the page size, transmission link bandwidth, and speed of light are denoted as $size_{pg}$, BW, and c respectively.

As for decision variables, they are described as follows:

$$W_{vs} = \begin{cases} 1 & if\ S\ hosts\ V \\ 0 & otherwise \end{cases}$$

$$X_{ipv} = \begin{cases} 1 & if\ pg\ is\ transferred\ during\ iter\ i \\ 0 & otherwise \end{cases}$$

$$MU_v = \begin{cases} 1 & if\ V\ \in\ migration\ unit\ MU \\ 0 & otherwise \end{cases}$$

b) Mathematical Formulation: The MILP objective function is defined as follows:

$$\min \quad \sum_v \text{Downtime}_v$$

It is subjected to the following constraints:

Boundary and Page Constraints:

$$W_{vs}, MU_v,\ X_{pv} \in \{0,1\}\, \forall\, v \in\ V, s \in\ S, p \in\ P \tag{1}$$

$$X_{ipv} * p_{pv} = P_v^{threshold} \quad \forall\, v \in\ VM,\ \ p \in\ P,\ \ i \in\ iter \tag{2}$$

$$Downtime_v \geq 0 \qquad \forall\, v \in\ V \tag{3}$$

Placement Constraints:

$$\sum_v (W_{vs} * \text{Res}_{vr}) \leq \text{Res}_{sr} \quad \forall\, s \in\ S,\ \forall r \in\ \text{Res} \tag{4}$$

$$\sum_s W_{vs} = 1 \qquad \forall\, v \in\ VM \tag{5}$$

$$\begin{cases} (W_{v's'} * delay_{ss'} - DT_v) \leq M \times y_{v'} \\ W_{vs} - 1 \leq M \times (1 - y_{v'}) \\ \qquad y_{v'} \quad \in \quad \{0,1\} \\ \qquad \forall\, s, s' \in\, S,\; v, v' \in \text{ Dep, Red} \end{cases} \qquad (6)$$

$$W_{vs} + W_{v's} \leq 1 \quad \forall\, s \in\, S,\; v, v' \in \text{ Red} \qquad (7)$$

$$W_{vs} + W_{v's} \leq 2 \quad \forall\, s \in\, S,\; v, v' \in \text{Dep} \quad TT_{v'} < RT_v \quad (8)$$

$$W_{vs} + W_{v's} \leq 1 \quad \forall\, s \in\, S,\; v, v' \in \text{Dep} \quad TT_{v'} > RT_v \quad (9)$$

$$\begin{cases} (1 - MU_v) \leq H * z_{v'} \\ \qquad (DT_v - delay_{vv'}) \leq H * (1 - z_{v'}) \\ \qquad z_v \in \{0,1\} \quad \forall\, v, v' \in \text{ Red, Dep} \end{cases} \qquad (10)$$

Downtime Constraint:

$$DT_v \leq (\frac{\sum_{v} Pg_v^h}{BW}) * size_{pg} + \frac{delay_{ss_{old}}}{c} \qquad (11)$$

$$\forall s, s_{old} \in\, S,\;\; v \in\, V$$

Constraints (1) and (2) ensure that the downtime is a positive number, and the other decision variables are binary. Constraint (3) determines that the page sets with low dirty rate are transferred during the iterative stage. Constraint (4) ensures that the VM computation resources should not exceed those of the selected host. Constraint (5) shows that a VM can be placed on only one server. Constraint (6) ensures that a migrated VM should be placed on the server while satisfying latency requirements with is dependents and redundant one. Constraint (7) shows that a VM cannot share the same host with its redundant whether the latter is migrated or not. Similarly, constraints (8) and (9) determine that a migrated VM cannot share the same host with its dependent(s) unless the latter cannot tolerate its absence. As for constraint (10), it shows that a VM belongs to migration unit if the placement algorithm cannot find a server satisfying the interaction constraints with its sponsor or redundant VMs. Finally, constraint (11) shows that the downtime is calculated

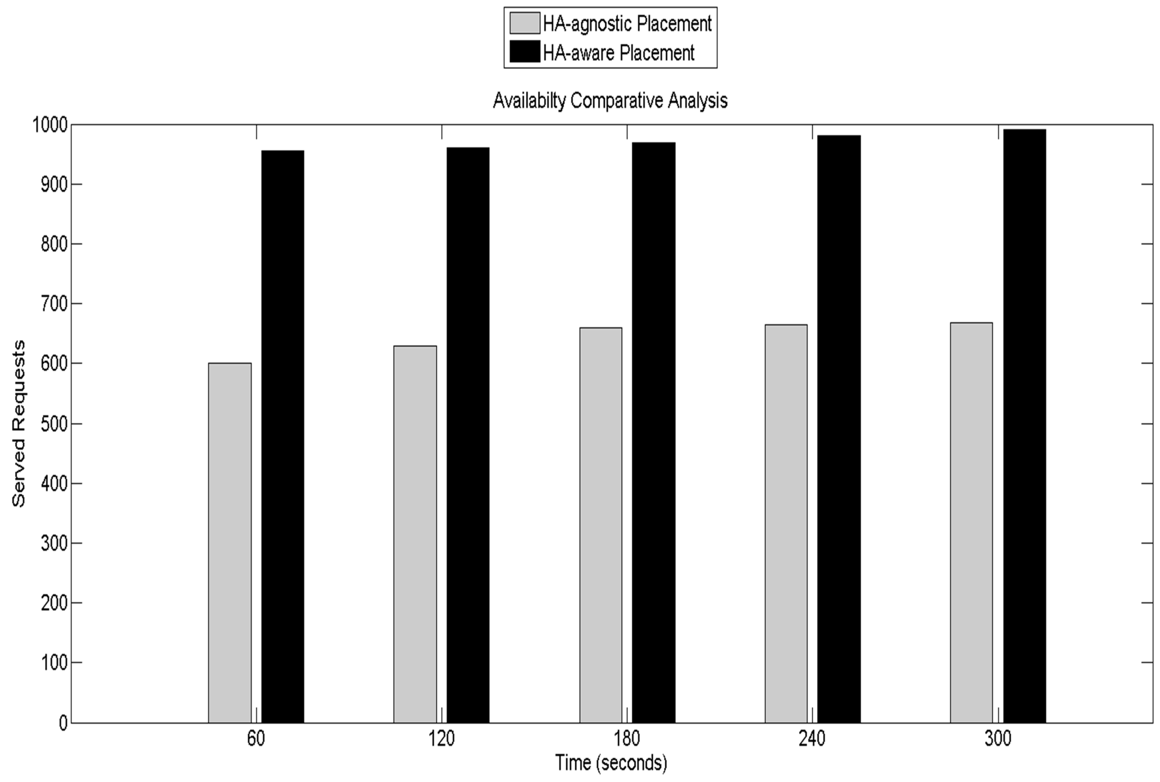


Figure 4.2: HA-based comparative analysis of placement algorithms in terms of the number of served requests.

in terms of memory pages with high dirty rate and delay between the new and old host of the corresponding VM.

4.4 Case Study

In this section, we perform a comparative analysis between different migration and placement approaches.

4.4.1 HA-aware deployments analysis

The HA-aware placement is evaluated on a 3-tier web application using SCPN model [7] [8]. At each tier, a component is backed up by 2 active redundant components. The infrastructure has 3 DCs and 50 servers. Once the placement algorithm generates the components deployments, they are inputted to the SCPN model. The model analyzes the deployments in terms of the number of served requests. During the analysis process, multiple components, servers, or DC failures, repair, and failover events can happen.

The HA-aware placement is compared to an HA-agnostic algorithm. The latter overlooks availability, redundancy, and interdependency constraints. It places the component on the server that satisfies computational resources and latency constraints. Both algorithms' deployments are evaluated using the SCPN. The model performs the evaluation with of TimeNet4.2 using transient simulation [32]. Note that the HA-agnostic approach does not support redundancy technique, and consequently, a component is not backed up when a failure event is fired.

Fig. 4.2 shows the results of the comparative analysis between both placement approaches. It is noticeable that the number of served requests is higher in the case of HA-aware deployments. The latter implements a failover technique that is triggered by a failure of the component, its host, or DC. Once a failure event is fired, the component is down, and its requests failover to its redundant components. Simultaneously, the repair policy is triggered, and the component is healthy again after the MTTR of the failed entity (component, its server, or DC). As for the HA-agnostic approach, it discards a redundancy technique and chooses the servers that satisfy only the functionality constraints. Consequently, the chosen server do not necessary have high MTTF or low MTTR and consequently, additional failures events might be fired. The failure of a component hinders the request processing, and thus the request is lost. Once the HA-aware placement approach is proven to meet the SLA, it is used to reserve new hosts for the migrated VMs.

4.4.2 Live migration preliminary results

The migration MILP model is also applied to a 3-tier web application and pool of 3 DCs and 50 servers. A paging analysis is performed on the migrated VMs [30]. This technique generates the writable working set of 4KB pages of the VM memory. This set determines the dirty rate of memory pages. The guest VMs sizes range from 256 MB to 1024 MB with BW = 256Mb/s. The MILP model is evaluated using CPLEX tool.

The downtime of each migrated VM is the metric used to compare the proposed HA-aware migration model to two different migration mechanisms, HA-agnostic pre-copy and stop-and-copy techniques. The results are shown in Fig. 4.3. The proposed migration model has lower downtime compared to the other techniques. It does not only perform iterative pre-

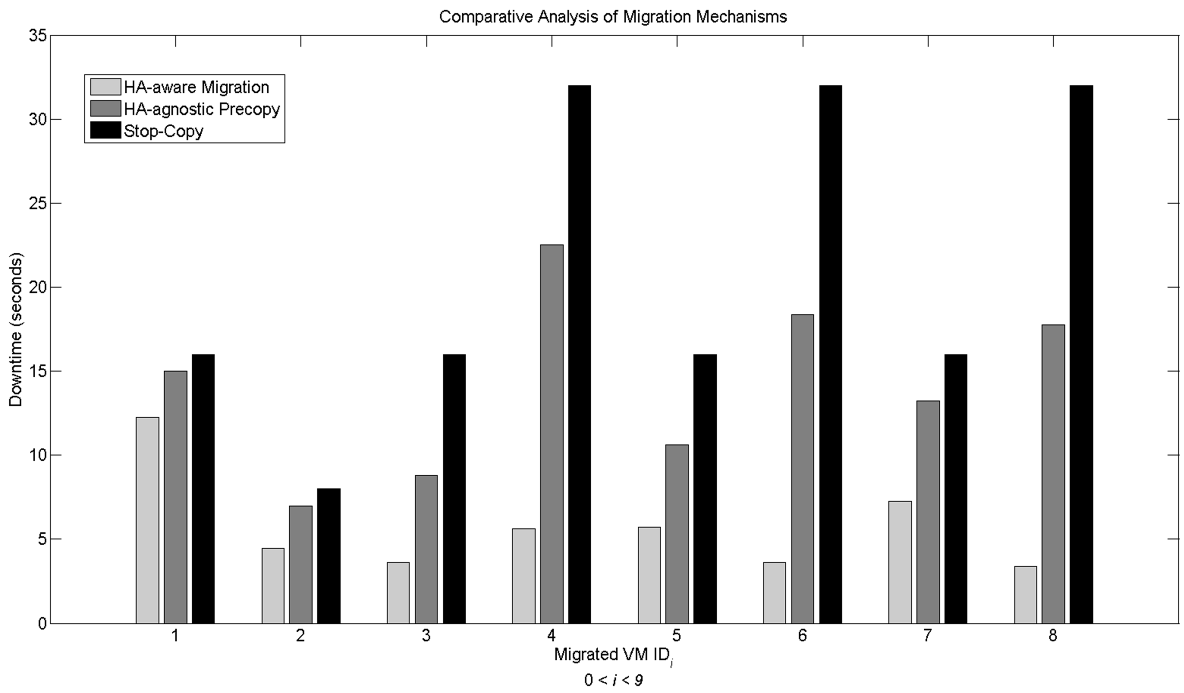


Figure 4.3: The downtime of each migrated VM for different migration mechanisms.

copy migration but searches for a new server that satisfies the availability and functionality constraints mentioned earlier. In addition to the high MTTF and low MTTR, the chosen server meets the latency requirements between the migrated component and its dependents such that the delay is reduced as much as possible to minimize the downtime. As for the HA-agnostic pre-copy, it has higher downtime values compared to the proposed migration model. It selects servers that satisfy computational resources requirements and perform iterative copying of the VM pages. Finally, the stop-and-copy technique has the highest downtime because it stops the VM and copies its memory to the selected destination. In this case, the migration time is the downtime, and the VM memory pages are copied during the pausing period whether they have low or high dirty rates.

4.5 Conclusion

The absence of HA policy can hinder the functionality of the business continuity. When cloud outage is not an option, it is necessary then to implement proactive HA-aware solutions to maintain service delivery and mitigate failures impacts. This chapter implemented different HA techniques to ensure the delivery of “always-on” and “always-available” services. The chapter provided design considerations to implement and evaluate a HA-aware placement technique. It also developed a MILP model to achieve live VM migration in the cloud while minimizing the downtime.

References

[1] M. Jammal, H. Hawilo, A. Kanso, and A. Shami, “Mitigating the Risk of Cloud Services Downtime Using Live Migration and High Availability-Aware Placement,” *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, December 2016.

[2] M. Armbrust, A. Fox, et al., “A view of cloud computing,” *IEEE Communications Magazine,* vol. 53, pp. 50-58, April 2010.

[3] R. M. Sharma, “The Impact of Virtualization on Cloud Computing,” *International Journal of Recent Development in Engineering and Technology,* July 2014.

[4] Ponemon Institute, “Study on Data Center Outages,” `http://www.emersonnetworkpower.com/en-US/Resources/Market/Data-Center/Latest-Thinking/Ponemon/Documents/2016-Cost-of-Data-Center-Outages-FINAL-2.pdf`, September 2013.

[5] Century Link, “The Low Down on High Availability in the Cloud,” `https://www.ctl.io/assets/pdf/CenturyLink-High-Availability-Whitepaper.pdf`, 2016.

[6] H. Hawilo, A. Kanso and A. Shami, “Towards an Elasticity Framework for Legacy Highly Available Applications in the Cloud,” *IEEE World Congress on Services*, pp. 253-260, 2015.

[7] M. Jammal, A. Kanso, P. Heidari, and A. Shami, “A Formal Model for the Availability Analysis of Cloud Deployed Multi-Tiered Applications,” *Third IEEE International Symposium on Software Defined Systems*, April 2016.

[8] M. Jammal, A. Kanso, P. Heidari, and A. Shami, “Availability Analysis of Cloud Deployed Applications,” *IEEE International Conference on Cloud Engineering (IC2E)*, April 2016.

[9] H. Liu, H. Jin, X. Liao, C. Yu, and C. Z. Xu, “Live Virtual Machine Migration via Asynchronous Replication and State Synchronization,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no.12, pp. 1986-1999, 2011.

[10] P. Riteau, C. Morin, and T. Priol, "Shrinker: Efficient Wide-Area Live Virtual Machine Migration using Distributed Content-Based Addressing," *Springer Euro-Par parallel processing*, pp 431-442, 2011.

[11] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," *4th USENIX conference on Networked Systems Design And Implementation*, 2007.

[12] G. Keller, M. Tighe, H. Lutfiyya, and M. Bauer, "An analysis of first fit heuristics for the virtual machine relocation problem," *8th international conference and workshop on systems virtualiztion management*, pp. 406-413, October 2012.

[13] V. Shrivastava, et al., "Application-aware virtual machine migration in data centers," *IEEE INFOCOM*, pp. 66-70, April 2011.

[14] S.K. Bose and S. Sundarrajan, "Optimizing Migration of Virtual Machines across Data-Centers," *International Conference on Parallel Processing Workshops*, pp. 306-313, September 2009.

[15] Z. Wei et al., "LVMCI: Efficient and Effective VM Live Migration Selection Scheme in Virtualized Data Centers," *IEEE 18th International Conference on Parallel and Distributed Systems*, pp. 368-375, December 2012.

[16] Y. Wu and M. Zhao, "Performance Modeling of Virtual Machine Live Migration," *IEEE 4th International Conference on Cloud Computing*, pp. 492-499, 2011.

[17] F. Machida, M. Kawato, and Y. Maeno, "Redundant Virtual Machine Placement for Fault-tolerant Consolidated Server Clusters," *IEEE/IFIP Network Operations and Management Symposium*, pp. 2-39, 2010.

[18] R. Al-Omari, A.K. Somani, and G. Manimaran, "An adaptive scheme for fault-tolerant scheduling of soft real-time tasks in multiprocessor systems," *Journal of Parallel Distributed Computing*, vol. 65, pp. 595-608, 2005.

[19] X. Zhu, X. Qin, and M. Qiu, "QoS-aware fault-tolerant scheduling for real-time tasks on heterogeneous clusters," *IEEE Transaction on Parallel Distributed System*, vol. 60, pp. 800-812, 2011.

[20] E. Feller, L. Rilling, C. Morin, R. Lottiaux, D. Leprince, "Snooze: a scalable, fault-tolerant and distributed consolidation manager for large-scale clusters," *IEEE/ACM Int'l Conference on Green Computing and Communications and Int'l Conference on Cyber, Physical and Social Computing*, pp. 125-132, 2010.

[21] R. T. Wang, "A dependent model for fault tolerant software systems during debugging," *IEEE Transactions on Reliability*, vol. 61, no. 2, pp. 504-515, 2012.

[22] W. Xu and T. Zhang, "A time-aware fault tolerance scheme to improve reliability of multilevel phase-change memory in the presence of significant resistance drift," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1357-1367, 2011.

[23] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu, "Performance and availability aware regeneration for cloud based multitier applications," *Dependable Systems and Networks Conference*, pp. 497-506, 2010.

[24] M. Zhong, K. Shen, and J.I. Seiferas, "Replication degree customization for high availability," *3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pp. 55-68, 2008.

[25] M. Alam and K. A. Shakil, "Recent Developments in Cloud Based Systems: State of Art," `https://arxiv.org/abs/1501.01323`, 2015.

[26] Netflix, "Netflix Shares Cloud Load Balancing And Failover Tool: Eureka!," `http://techblog.netflix.com/2012/09/eureka.html`, September 2012. [July, 2016]

[27] M. Jammal, A. Kanso, and A. Shami, "High Availability-Aware Optimization Digest for Applications Deployment in Cloud," *IEEE International Conference on Communications (ICC)*, pp. 6822-6828, 2015.

[28] M. Jammal, A. Kanso, and A. Shami, "CHASE: Component High- Availability Scheduler in Cloud Computing Environment," *IEEE International Conference on Cloud Computing (CLOUD)*, pp. 477-484, 2015.

[29] S. Akoush, R. Sohan, A. Rice, A. W. Moore, and A. Hopper, "Predicting the Performance of Virtual Machine Migration," *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pp. 37-46, August 2010.

[30] C. Clark, et al., "Live migration of virtual machines," *2nd conference on Symposium on Networked Systems Design & Implementation*, p. 273-286, May 2005.

[31] D. Kapil, E. S. Pilli and R. C. Joshi, "Live virtual machine migration techniques: Survey and research challenges," *3rd IEEE International Advance Computing Conference (IACC)*, pp. 963-969, 2013.

[32] A. Zimmermann, "Modeling and Evaluation of Stochastic Petri Nets with TimeNET 4.1," *6th International Conference on Performance Evaluation Methodologies and Tools*, pp. 54-63, 2012.

Chapter 5
GITS: Generic Input Template for CloudSim and Cloud Simulators

5.1 Introduction

Nowadays, many enterprises are moving to the cloud to benefit from the cloud applications availability, elasticity, pay-per-use basis, multi-tenancy, and resource provisioning. With the lightweight virtualization, the cloud infrastructure provides virtual machines (VMs) and containers to run multiple applications in private, public, or hybrid cloud environments [2] [3]. With this cloud migration movement, cloud-based applications are handling many users' demands while leveraging the economy of scale [4]. The application flexibility, management, and appearance over virtualized cloud infrastructure are the key points to measure the competitiveness between different cloud providers and users. Therefore, it is necessary to have an efficient cloud orchestration that handles the management, configuration, and coordination of cloud-based applications. Cloud orchestration automates the management of interacting applications and facilitates their portability across different cloud systems. In order to ensure an automated cloud management, it is important to develop application-infrastructure models and specifications that capture both cloud provider, cloud user, hypervisor, and service level agreement (SLA) requirements including high availability, computational performance, and latency.

Cloud offers different types of services including the Software, Infrastructure, and Platform as a Service. Platform as a Service (PaaS), such as Google AppEngine [5] or Microsoft Azure [6] and Infrastructure as a Service (IaaS), such as GoGrid [7] or Amazon

The content of this chapter have been submitted for publication [1].

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC-STPGP 447230) and Ericsson Research.

Elastic Compute Cloud (EC2) [8] are the base of multiple cloud applications. Although the deployment and optimization policies of these services are widely covered, there are few approaches regarding the generic design of cloud-based applications and infrastructure while satisfying the high availability (HA) requirements. The dependability on different cloud offerings including Application as a Service (AaaS) requires a generic design for the cloud-based application model that can be easily customized based on a given cloud properties. Besides, applications deployments are user-specific and change the cloud properties and constraints where different clouds can allow different dependency and sharing between applications components. This requires a user-customized cloud template that can reach more customers while ensuring that elasticity, HA, and workload management are maintained.

Designing generic cloud-based models should capture the cloud infrastructure, applications architecture, and their interactions. Cloud-based applications consist of multiple components running on different VMs or containers. The latter maps the application to the cloud infrastructure consisting of data center (DC) networks. Besides, different interactions relations take place between cloud-based applications including dependency and redundancy. Therefore, using a swivel chair interface to create cloud scenarios that capture a given cloud system is a tedious and error-prone task [9] [10].

Cloud simulations are used to imitate cloud system behaviors. They give insights into how cloud performs under certain conditions and constraints. CloudSim simulator allows modeling and simulation of cloud infrastructure, cloud broker, and scheduling policies [9]. It provides virtualization engine that creates multiple services on a DC while considering time- and space-shared allocation policies. However, a generic input template that model cloud scenarios is still missing. In this chapter, we aim at providing a generic input template for cloud simulators (GITS) in general and CloudSim specifically. The proposed template models cloud infrastructure consisting of multiple DCs, cloud user consisting of multiple interacting applications components, workload models, high availability (HA) metrics and redundancy models, and other SLA requirements. In other words, GITS defines the cloud infrastructure configurations and applications interoperable descriptions including components, relationships, interdependencies, computational and latency requirements. GITS allows applications interoperability and automated orchestration across mul-

tiple cloud providers thus facilitating cloud scenarios creation, ensuring configurations and models reusability, improving availability, and minimizing error, time, and cost-to-value.
This chapter provides a multi-layer input template. At the frontend layer, a graphic modeling framework (GMF) project is designed to capture the above cloud model. GMF generates an Extensible Markup Language (XML) file. In order to ensure data reusability, the XML file is inputted to the middle layer where a JavaScript Object Notation (JSON) template is generated. This template provides the specifications of cloud model in a human readable format. In the backend layer, the JSON template is mapped to a Unified Modeling Language (UML) class diagram. With this diagram, CloudSim specification is extended to include the cloud infrastructure, cloud application components, and HA requirements. It is important to note that the proposed template can be easily mapped to any other cloud simulator while applying few tuning to the transformation step.
The rest of this chapter is organized as follows. Section 5.2 presents related work for industry and research-based cloud orchestration and specifications. Section 5.3 defines the motivation behind GITS where it describes CloudSim simulator, need for component-based architecture to model cloud scenarios, cloud challenges, and GITS contributions that address these issues. Section 5.4 describes the GITS framework, where it presents the GITS graphical and textual models, and the transformation algorithm to translate them to a data format that is understandable by CloudSim. The evaluation of GITS and other encoding methods of proposed template are defined in Section 5.5. Finally, the conclusion is presented in Section 5.6.

5.2 Related Work

Several literature studies address the cloud applications orchestration and define the cloud specifications models for simulation tools [11]-[17].

5.2.1 Industry-based cloud orchestration and specifications

Amazon Webservices (AWS) provide AWS CloudFormation as its proprietary orchestration approach [18]. It provides a stack, JSON template, that specifies AWS resources (Elastic Load Balancer instances, Amazon Relational Database Service (RDS) instances). The template describes the resources needed to process certain applications, and it is managed as an entity. The AWS CloudFormation provisions resources, manages their creation, deletions, and dependencies. However, the template ignores HA attributes and redundancy models.

OpenStack proposes Heat as their orchestration platform. Heat uses templating approach, Heat Orchestration Template (HOT) to manage resources creation and management and facilitate portability between multiple clouds environments [19]. The proposed templates have a similar structure as the AWS CloudFormation templates and can be integrated with Puppet and Chef. Heat uses YAML files to define its template that supports auto-scaling and some HA features including instances logical grouping, services running in an instance, VMs or individual instances. Although HOT supports some HA features, but it discards the mean time to failure (MTTF), mean time to repair (MTTR), recovery time, and tolerance time as basic HA metrics. Also, it does not describe redundancy models and interdependency between multiple applications components.

Organization for the Advancement of Structured Information Standards (OASIS) proposes Topology and Orchestration Specification for Cloud Applications (TOSCA) [20]. The latter is an industry-based standard developed to define and manage applications and facilitate their portability between different multiple cloud providers and thus minimizing vendor lock-in. Using XML-based template, TOSCA describes the topology of applications components and their interactions, provides orchestration platform to manage applications deployment, and supports interoperability. TOSCA defines topology as a graph with a set of nodes and relationships, each assigned a type (inheritance, requirements, properties, implementations). Business Process Model and Notation (BPMN) and Business Process Execution Language (BPEL) are used in TOSCA management plans, which depend on standard workflow settings. Based on OpenTOSCA [21] [22], a cloud-based modeling tool, Winery, is designed to define cloud-based application topologies [23] [24] [25]. Winery is a web-based graphical model that describes TOSCA cloud services topologies and management plans. Winery consists of a type and template management modules that create and modify components defined in TOSCA. It stores the information in a repository and uses TOSCA packaging format to import and export them [24]. Similarly, both TOSCA and Winery do not support HA features and redundancy relationships.

Carlson et al. propose Cloud Application Management for Platforms (CAMP) [21]. CAMP is an approach that improves the cloud interoperability over different cloud infrastructures. With CAMP, authors are aiming at defining specifications that facilitate the cloud applications management. CAMP provides a self-service management Application Program Interface (API) that allows a platform implementation layer to control applications deployment and platform usage. Through its specifications, CAMP allows the creation of different services that can interact with other platforms thus facilitating interoperability. Although CAMP aims at facilitating cloud portability, it discards impact of interoperability on cloud applications HA. It does not describe services interactions, redundancy models, recovery, and repair policies.

5.2.2 Research-Based cloud orchestration and specifications

Tian et al. propose CloudSched, Java-based simulation tool to evaluate the resource allocation for cloud-based applications [26]. The tool consists of a graphical user interface (GUI) where the simulation setup and cloud scenarios are created. The specification of cloud scenarios does not follow any standardized format, which hinders the reusability of certain scenarios. The proposed GUI is simple and has restrictions on the number of physical machines and VMs, which limits its functionality in simulating real cloud systems. Although CloudSched provides a GUI to facilitate setup creations, it discards the need for data reusability. Additionally, it does not capture the cloud provider and user models and overlooks any HA attributes or applications interactions. Filho et al. propose a YAML document as a template to create scenarios to CloudSim [27]. Although YAML schema ensures simplicity and reusability, but the proposed template is proprietary to CloudSim and does not capture any HA metrics. This proprietary hinders the applications portability across different cloud simulators.

Wickremasinghe et al. provide CloudAnalyst, an extension to CloudSim [28]. CloudAnalyst describes the cloud applications workloads, DCs resources, and traffic/DCs geographical locations. With these features, request response time, processing time, and other related measures are determined. CloudAnalyst consists of a Java-based GUI to create different cloud setups and a graphical output representation in table and charts forms. It also supports XML-based files to save simulations input data and results. Although CloudAnalyst supports XML files, but it does not provide an input model template to minimize the error and time consumed by cloud users.

Jakovits et al. and Srirama et al. develop Stratus cloud simulation framework [29] [30]. Stratus provides simulations of distributed cloud applications using bulk synchronous parallel (BSP) models. BSP consists of an iterative algorithm that ensures the tasks parallelism. Stratus also provisions resources to allow scaling up and down of cloud scientific applications. However, Stratus does not provide an input template to facilitate creation of cloud scenarios. It also overlooks HA features of the cloud model. Guo et al. provide a service specification for simulating a software as a service (SSaaS) [31]. The specification describes service-oriented and SSaaS setups, which generates a meta-model for simulated scenarios. Although the authors propose a web-based GUI, they do not capture HA metrics of different cloud elements. Also, the GUI does not support a reusability and repeatability features of the cloud scenarios.

While Balmer et al. propose MATSim, a traffic simulator [32], Behrisch et al. provide Simulation of Urban mobility (SUMO) tool that can be modified to simulate cloud-based scenarios [33]. In contrary to MATSim, SUMO has a GUI, but both simulators use configuration files to import its input parameters and export its simulation results. In both tools, the cloud scenario discard HA characteristics, and its creation requires a deep knowledge from the user to the configuration settings of the cloud system. In contrary, GITS propose a human-readable and HA-aware template.

In this chapter, we distinguish our work from the previous initiatives by developing a generic input template for any cloud simulator, mainly CloudSim. GITS has a graphical modeler where Eclipse GMF model is created. In the middle layer, a Parser engine is developed to ensure the generation of a JSON-based template from an XML-based files. The JSON template ensures the input models' repeatability and reusability. It also captures the cloud environment from both provider and user sides. It maps the cloud infrastructure with cloud applications using lightweight virtualization technology, VMs or containers. One of the main features of GITS is the HA-aware specifications. It describes the redundancy models of cloud applications, recovery/repair policies, interdependency between applications, MTTF, MTTR, and other HA metrics. With GITS, any cloud scenario can be created, which includes the needed infrastructure configuration files (resources and HA metrics) and the cloud application structure information.

5.3 Motivation

Cloud computing is considered a transitioning technology for Information and Communications Technology (ICT) area where different service models are provided over broadband networks [34]. Given an SLA, a cloud model should satisfy different functional and non-functional properties such as HA, performance, energy efficiency, and other attributes [35]. In order to meet these requirements, a simulation environment is needed to evaluate any cloud design and associated scenarios given a set of metrics (computational resources, SLA requirements, HA parameters). Simulation is an important way for the cloud to allow repeatable scenarios in a controlled environment. They can evaluate different cloud parameters such as outage or security issues. In turn, the evaluation strategies can be used as filters for the failed approaches compared to other ones or for the approaches that do not meet the quality of service (QoS). However, a simulation tool requires an input modeling or templating of a cloud system to ensure scenarios reusability, system modularity, and minimizes error associated during manual input model generation. In this section, we discuss CloudSim simulator, need for component-based architecture, challenges of the cloud solutions, and GITS contributions.

5.3.1 CloudSim simulator

CloudSim is a Java-based open source framework for modeling, simulating, and evaluating cloud environments [9] [10]. It is proposed by GRIDS laboratory and developed on the top of SimJava, a discrete event simulator [36]. CloudSim is used to model cloud infrastructure (data centers (DCs) and servers), cloud broker, cloud information system, and multiple time and space-based allocation and scheduling policies. Besides, it models containers/VMs processing including instantiation, provisioning, and destruction. With CloudSim, large cloud systems can be processed using different scheduling approaches where VMs/containers are mapped to the hosts that satisfy their computational resources. Although CloudSim is a self-contained tool that models cloud architecture, it does not support the simpler creation of cloud scenarios. The latter is manually created or hard-coded, which can be a tedious and error-prone job. Therefore, in this chapter, a generic input template is proposed that models cloud infrastructure and applications. This template can be easily tuned to model scenarios for different cloud simulators.

5.3.2 Component-Based architecture

Component-based architecture (CBA) provides well-designed complex systems with different methods, properties, and events. CBA decomposes the system design into logical or functional sub-systems where each compromises a specific partition of the whole communicating system. With this system abstraction, CBA allows designing the cloud models using software entities of Commercial off The Shelf (COTS) offered by multiple cloud providers. Adapting CBA in building cloud templates ensures components reusability, replaceability, extensibility, and modularity. Each entity, known as a component, of the CBA encapsulates certain properties, methods, and behaviors that represent for example an element of the cloud-based model (application and infrastructure). For example, SaaS offers services as interned-based applications including multiple component-based applications [37]. These applications communicate using a message exchange engine that is based on protocols, such as web services description language (WSDL) or simple object access protocol (SOAP). Different platforms can be used to build CBA including JavaBeans [38], Common Object Request Broker Architecture (CORBA) [39], Distributed Component Object Model (DCOM) [40], and Microsoft.NET. With component-based applications, the cloud can offer scalable, interoperable, and highly available services.

To this end, CBA can be adopted to design a generic template for cloud simulators, which aims at enhancing system quality, evaluation, and building scenarios from standard elements instead of redesigning the wheel. Additionally, it replaces the manual design of cloud entities by an automated generation of scenarios that can be parameterized according to a given cloud environment (provider and/or simulator objective) [41]. Thus, entities needed for particular application and infrastructure are instantiated from the proposed template. This describes a framework that includes the structural entities, methods, and parameters of the cloud system and the interconnection behaviors between its different elements. The main idea is to design interoperable cloud-based template through well-defined connections that can be easily customized to meet a given cloud management system, such as OpenStack (HOT) and AWS EC2 (CloudFormation template) and the associated SLA requirements.

5.3.3 Issues and contributions

This section addresses the different cloud issues and the corresponding contributions proposed by GITS framework.

1. Issue 1: Cloud-based Application Interactions

Cloud-based applications consist of multiple components that interact with each other due to dependency-redundancy relations. When modeling these applications, it is necessary to describe the interactions relations between their components and their dependency on the cloud infrastructure.

GITS Contribution 1: In GITS, a component-based application is modeled to capture different redundancy and dependency relations among different components and their dependencies on given cloud infrastructure.

2. Issue 2: Cloud Provider-User Partition

Vendor lock-in is one of the challenges faced by users when choosing the corresponding cloud providers [42]. Many cloud-based applications are provided and offered by the same vendors, such as the Google office suite application [43] or Salesforce Customer Relationship Management (CRM) application [44]. This creates users dependency on certain providers. Therefore, it is necessary to separate the cloud providers and cloud applications, which allows users to select the appropriate provider and application vendors. Besides, this portioning allows the provider and user to enlarge their customer numbers and share a standardized cloud provider-user template.

GITS Contribution 2: GITS provides a well-defined cloud template that ensures a separation between the cloud provider and cloud user while mapping them through a virtualization engine, where VMs/containers are instantiated according to a given allocation objective. The proposed template does not depend on certain technologies or cloud simulators.

3. Issue 3: High Availability for Cloud Applications

The dependency on multiple cloud services does not only increase the number of customers but requires systems that are always available anytime and anywhere. Building highly available cloud environment becomes inevitable.

Although different HA-aware policies for cloud applications have been proposed [45], they are proprietary approaches, which hamper the applications interoperability across different providers. Also, the literature has many cloud simulations with predefined input models, but up to our knowledge, the literature does not propose an input template that captures availability metrics. These metrics include MTTF, MTTR, redundancy models, failover policies, and other HA attributes.

GITS Contribution 3: High availability modeling is one of the main contributions of GITS. Different redundancy models, availability parameters, failure types, and HA middleware attributes are captured in GITS template.

4. Issue 4: Customization of Applications

Cloud users have different functional and non-functional requirements on applications. In order to satisfy their needs, these applications should be well-defined, modeled, and evaluated while considering modularity and portability. Building a modular and portable template allows the cloud users to adjust the application to meet certain objectives and a given cloud infrastructure. Template customization requires prompt variability adaptation without violating applications interactions. To ensure customization and modularity, generic cloud templates should be designed, which can run on any cloud environment.

GITS Contribution 4: GITS template is generically designed to model cloud providers and applications. With appropriate transformation algorithm, GITS can capture the input model of CloudSim and any other cloud simulator or environment.

5. Issue 5: Unique Template for Cloud Management

Each cloud provider has its own application programming interfaces (APIs) and tools to set up applications and infrastructures and provision them. It is necessary to define a rubric that describes the necessary attributes for a cloud provider-user management module. An architecture of a unification layer that syntactically and functionally unifies the APIs of different providers and their provisioning infrastructure is needed. This unification layer is the basis for describing the provider-independent provisioning scripts for applications.

GITS Contribution 5: GITS provides a generic specification that provides a unified view of requirements, parameters, and interactions between different entities of a cloud. This specification facilitates the description of management and provisioning unit for cloud infrastructure and applications.

In the following, we explain the GITS framework design and its mapping to CloudSim.

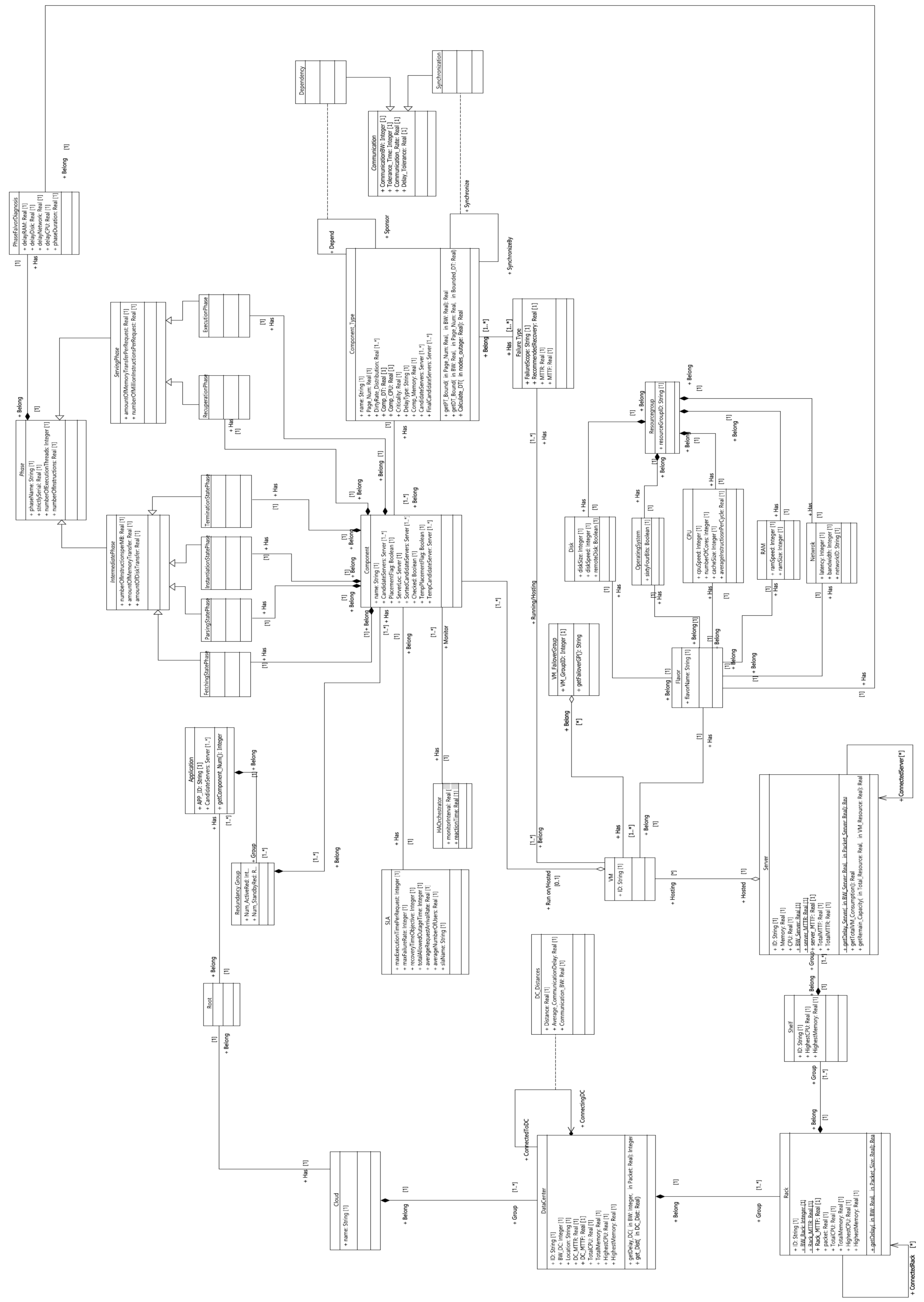


Figure 5.1: UML class diagram of GITS cloud model.

5.4 GITS Framework

Generating scenarios for CloudSim and other cloud simulators requires knowledge in development and simulator environment. It is not always the case that the cloud users are aware of the code needed to generate these scenarios. This can hinder the process of applications deployment and cloud systems evaluations because of the complexity challenge associated with cloud scenarios creation. For instance, if cloud-based applications are deployed while considering the impact of dependency relation among them, a new code should be added that reflects this interaction. This can be an exhausting task, which is not only an erroneous process but requires prerequisite knowledge in the tool used as well.

In order to have a generic input template for cloud simulators where any user can create it, we propose GITS, a user-friendly approach that solves the above challenges. GITS minimizes the exposure of cloud users to the development process and provides an intuitive way to generate any cloud scenario that depicts applications, infrastructure, and their interfaces. The objective of this template is to separate the cloud providers from the cloud users and ensure their mapping through a virtualization layer. Besides, the proposed template captures the high availability features, such as redundancy models, availability metrics, failure types, and repair policies. In addition to performance metrics (computational resources and latency), modeling HA parameters allows considering the availability as an objective during cloud applications deployment where applications are mapped to the providers that protect them according to a given redundancy model and other related metrics.

Creating such template is achieved using a UML model to build cloud model, JSON file to have a human readable template, and finally a graphical interface to maintain GITS user-friendly feature. In the following, we describe the details of GITS design.

5.4.1 GITS UML model

The initial phase in designing a generic template for cloud simulators is abstracting the cloud through system modeling. The objective of the abstract modeling is to define different cloud characteristics, faulty points, redundancy model, recuperation phases, and other availability and performance attributes. In this chapter, UML class diagram is used to provide an abstract view of the cloud model. The diagram can be partitioned into cloud infrastructure, cloud application, and virtualization layers.

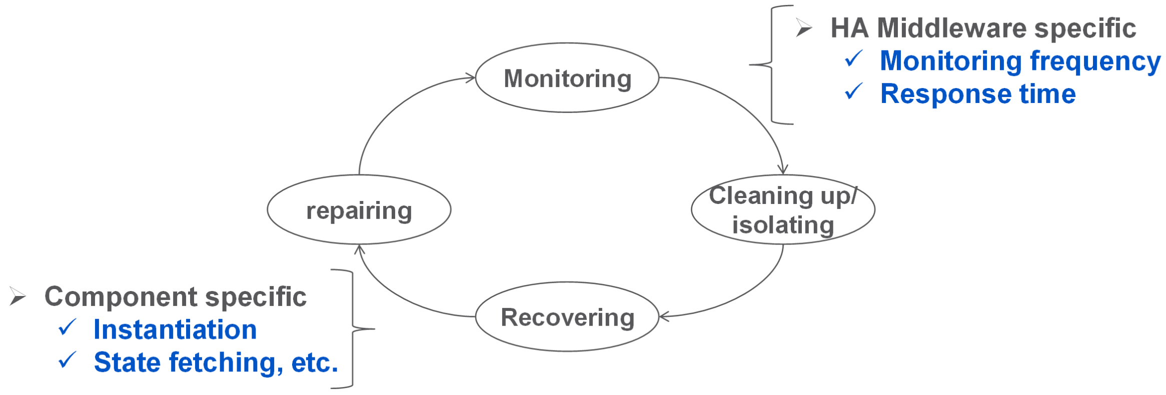


Figure 5.2: HA solution state model.

Fig. 5.1 shows the cloud UML diagram.

1) Cloud Infrastructure:

The cloud infrastructure consists of data center network where each DC has its computational resources (CPU and memory) and associated HA metrics, such as MTTF and MTTR. Each DC has one or many rack(s). Similarly, each rack is characterized by its available resources and HA attributes. Multiple shelves can be grouped in one rack where each shelf has one or more server(s). Each server is defined through its resources and HA parameters. Each of these elements has other attributes defined in the class diagram to be used for designing and scheduling purposes.

2) Cloud Application:

The cloud-based application consists of multiple component types. Each type has multiple components where one component is considered active while the others represent the redundant ones. Each component type requires specific computational resources to be properly processed. These resources are captured as flavors in the proposed diagram. Flavors can be repressed as disk resources (disk size or speed), operating system specification, CPU requirements (core, cache size, speed), memory (RAM size or speed), and/or network (bandwidth or latency). Each component type has its own failure types that define failure scope (impact of component type failure), MTTF, MTTR, and recovery time.

Each component type consists of multiple redundant components, which forms protection or redundancy group. Each group determines the number of active, standby, and/or spare components depending on the used redundancy model. In order to maintain HA, each

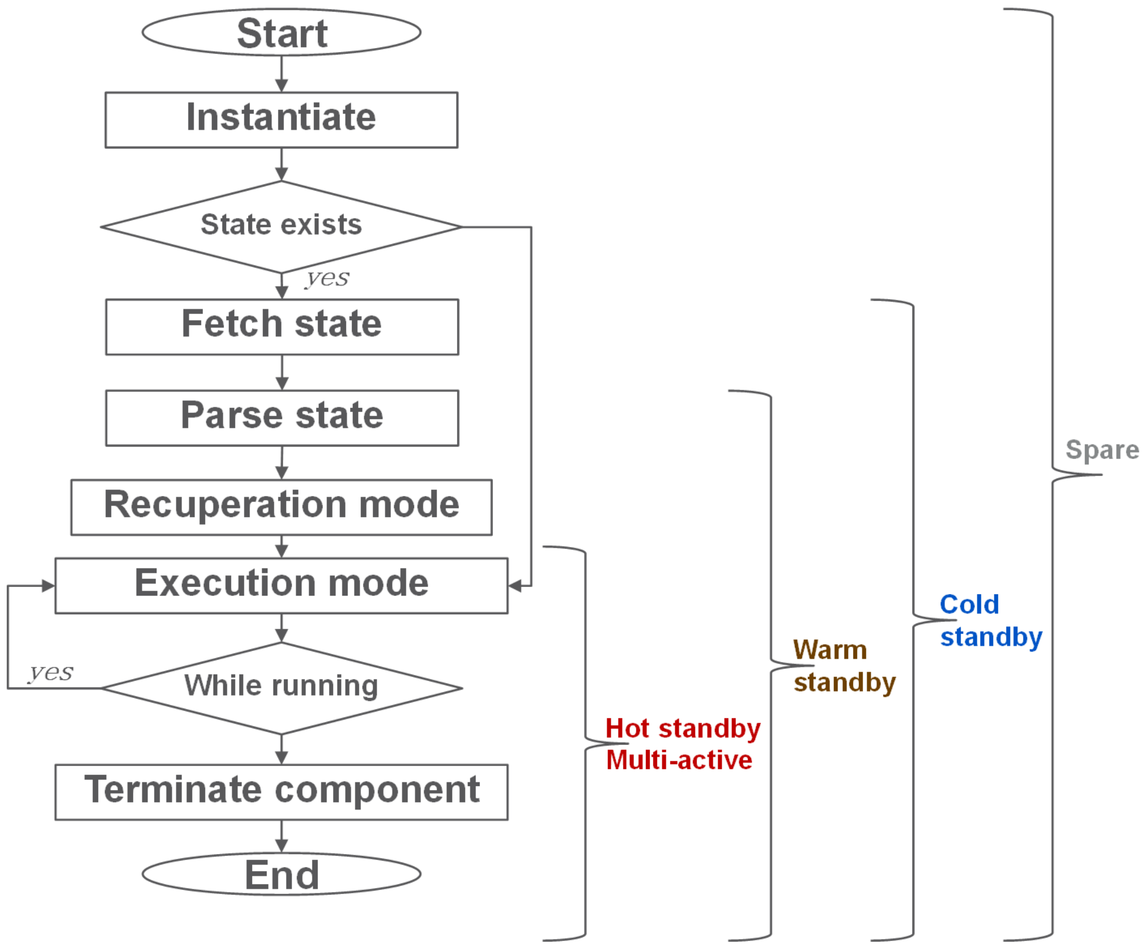


Figure 5.3: Effect of redundancy model on failover time.

component follows a sequence of operations during its life cycle. The state model of an HA solution is captured in the UML diagram. Fig. 5.2 depicts the above component state model. During this life cycle, an HA orchestrator or middleware monitors the health of one or more components. Each orchestrator is characterized by a monitoring frequency and a response time. Upon failure detection, the faulty component is isolated and fails over to its redundant component(s). The failover time depends on the redundancy model (active/active, active/standby, or active/spare). Fig. 5.3 depicts the impact of the redundancy model type on the failover time calculation. For instance, if the redundancy model is active/spare, the failover time is the summation of the instantiation time, fetch state delay, parsing state delay, recuperation duration, execution time, and termination duration. Each of these durations depends on the flavors of the component's host.

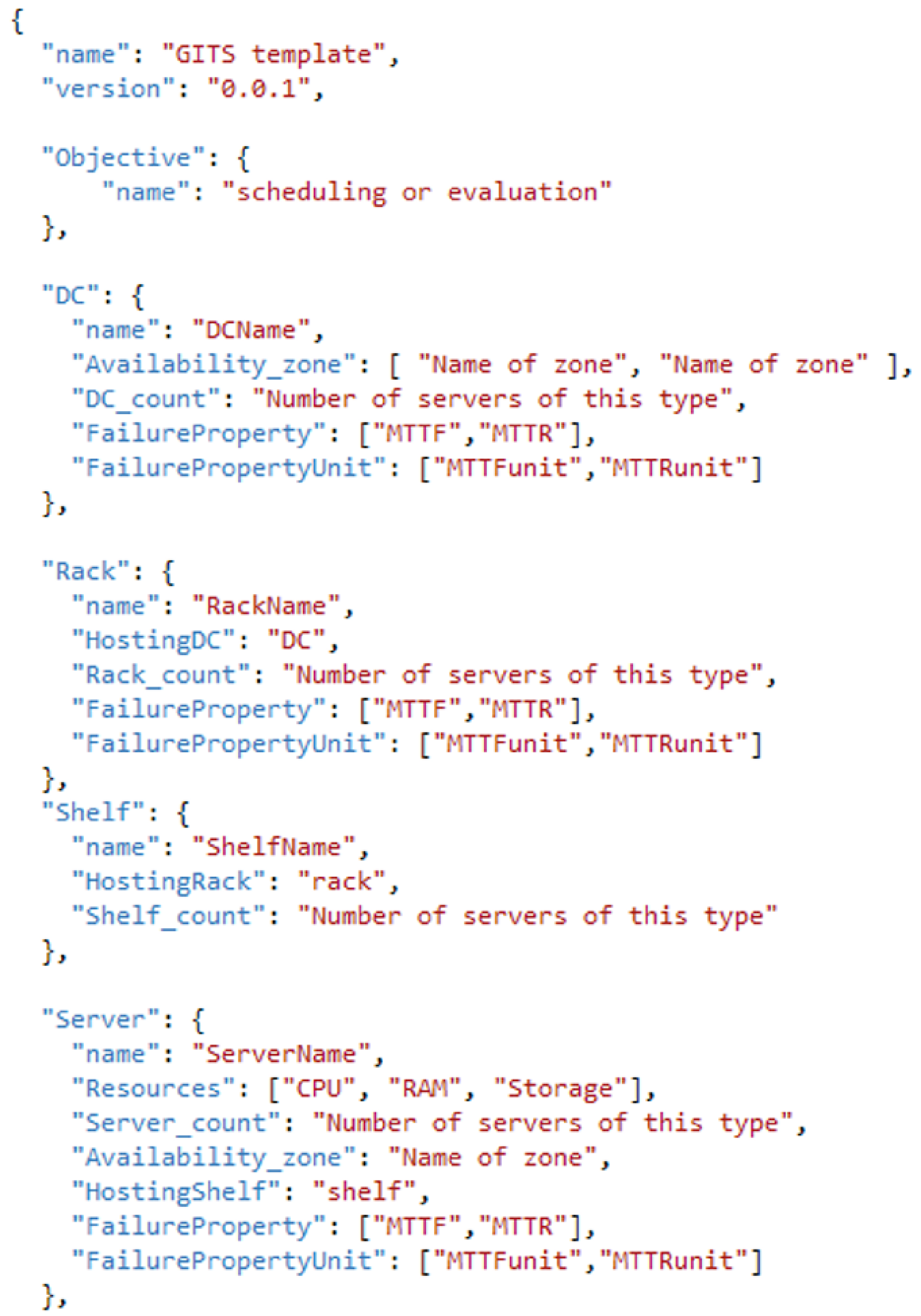
Each component type can interact with other types through dependency relation. A type can sponsor or depend on another type. A 3-tier web application can be an example of cloud-based applications. A web application consists of Hypertext Transfer Protocol Secure (HTTPS) server at the front-end, which processes user requests and forwards them to an App server. The latter generates the required content and in turns depends on the back-end database (DB) server that stores the users' data. The communication between these different component types forms the functional path that a request should follow to be successfully processed. The dependency between different types is characterized by a delay tolerance that represents the allowed delay between them and a tolerance time that determines how much a dependent component can tolerate the absence of its sponsor(s). These metrics have an important role when selecting dependent and sponsor placements. Each component is associated with an SLA that determines the allowed outage time, average request arrival rate, recovery time objective, and other HA, performance, and scheduling attributes.

3) Virtual Mapping:

The applications components are mapped to the servers that can satisfy their computation needs, HA, and other performance objectives. Once the allocator finds the best server that can host a given application components, a virtual mapping is generated between the server and the component. The virtual mapping can be a virtual machine or a container. This mapping forms the glue between the cloud provider and user.

5.4.2 GITS JSON file

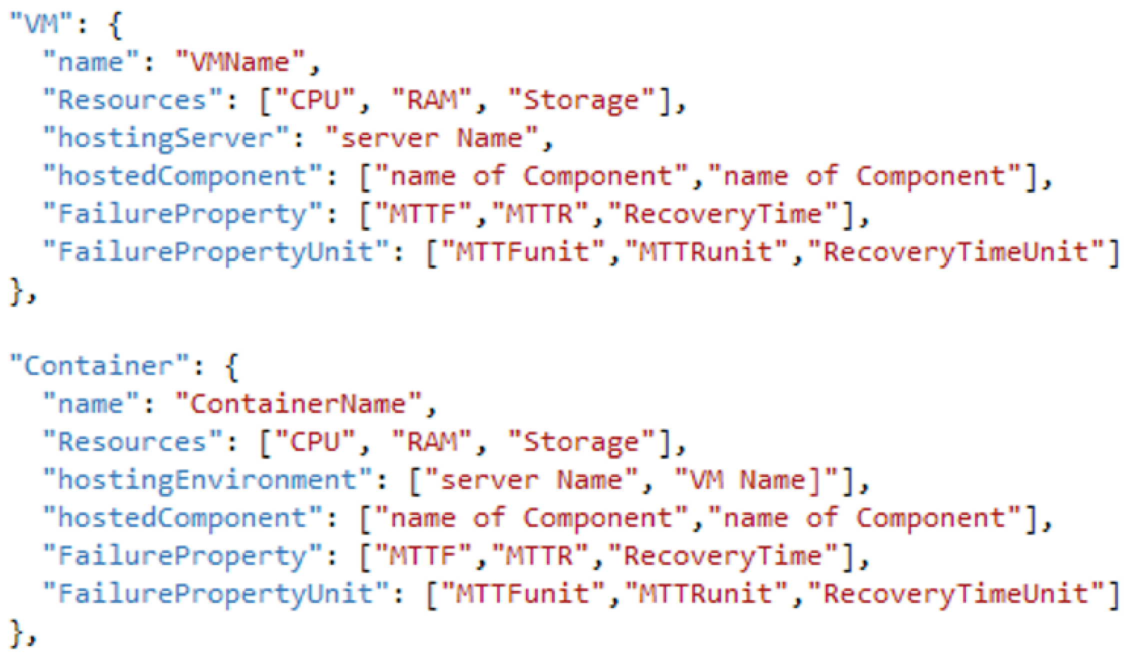
UML class diagram is a general-purpose language that provides an abstract view of the cloud model. However, it does not ensure simplicity in scenarios creation and repeatability. For this purpose, we use JSON to represent a cloud template. JSON is a simple, human-readable, and universal language. It is considered a lightweight format for data exchanging [46]. JSON does not depend on the programming language type and can support multiple data types (numbers, arrays, objects, strings, Boolean, and null) and deep level hierarchal data. Also, JSON has many extensions that enable cyclic relations implementation, such as dojox in Dojo toolkit used in Google Content Delivery Network (CDN) [47]. Besides, JSON can be parsed to any other data schemas, such as XML and YAML documents. To



```
{
  "name": "GITS template",
  "version": "0.0.1",

  "Objective": {
      "name": "scheduling or evaluation"
  },

  "DC": {
    "name": "DCName",
    "Availability_zone": [ "Name of zone", "Name of zone" ],
    "DC_count": "Number of servers of this type",
    "FailureProperty": ["MTTF","MTTR"],
    "FailurePropertyUnit": ["MTTFunit","MTTRunit"]
  },

  "Rack": {
    "name": "RackName",
    "HostingDC": "DC",
    "Rack_count": "Number of servers of this type",
    "FailureProperty": ["MTTF","MTTR"],
    "FailurePropertyUnit": ["MTTFunit","MTTRunit"]
  },
  "Shelf": {
    "name": "ShelfName",
    "HostingRack": "rack",
    "Shelf_count": "Number of servers of this type"
  },

  "Server": {
    "name": "ServerName",
    "Resources": ["CPU", "RAM", "Storage"],
    "Server_count": "Number of servers of this type",
    "Availability_zone": "Name of zone",
    "HostingShelf": "shelf",
    "FailureProperty": ["MTTF","MTTR"],
    "FailurePropertyUnit": ["MTTFunit","MTTRunit"]
  },
```

Figure 5.4: JSON-based cloud infrastructure template.

this end, we use JSON data format to represent a readable and reusable cloud settings. Fig. 5.4, Fig. 5.5, and Fig. 5.6 show the JSON files for the cloud infrastructure, application,



```
"VM": {
  "name": "VMName",
  "Resources": ["CPU", "RAM", "Storage"],
  "hostingServer": "server Name",
  "hostedComponent": ["name of Component","name of Component"],
  "FailureProperty": ["MTTF","MTTR","RecoveryTime"],
  "FailurePropertyUnit": ["MTTFunit","MTTRunit","RecoveryTimeUnit"]
},

"Container": {
  "name": "ContainerName",
  "Resources": ["CPU", "RAM", "Storage"],
  "hostingEnvironment": ["server Name", "VM Name]"],
  "hostedComponent": ["name of Component","name of Component"],
  "FailureProperty": ["MTTF","MTTR","RecoveryTime"],
  "FailurePropertyUnit": ["MTTFunit","MTTRunit","RecoveryTimeUnit"]
},
```

Figure 5.5: JSON-based provider-user mapping template.

and virtualization layer.

The GITS JSON template consists of the following:

1) Objective:

It is a string data type that represents the goal behind using the template. It can be either evaluation or scheduling. In the case of “evaluation” objective, the template is inputted to a cloud simulator, such as CloudSim to evaluate certain applications deployment in terms of availability or other performance. For this purpose, the template is populated with deployment information of a certain application, such as the hosts of the application components. In the case of “scheduling” evaluation, the template is inputted to a cloud simulator to schedule the application components.

2) Cloud Infrastructure Information:

DC: It represents the DC details of a given cloud infrastructure. It includes the DC *name* (string), *Availability_zone* (array of strings) that hosts its servers, *FailureProperty* (array of numbers) that represents its MTTF and MTTR, and *FailurePropertyUnit* (array of strings) that reflects units of MTTF and MTTR. If multiple DCs hold similar characteristics, a *DC_count* can be defined to automate their generation and avoid repetition.

Rack: It represents the rack details of a given cloud infrastructure. It includes the rack *name* (string), *HostingDC* (DC object) that represents the DC hosting the corresponding rack, *FailureProperty* (array of numbers) that represents its MTTF and MTTR, and *FailurePropertyUnit* (array of strings) that reflects units of MTTF and MTTR. If multiple racks hold similar characteristics and reside on same DC, a *Rack_count* can be defined to automate their generation and avoid repetition.

Shelf: It represents the shelf details of a given cloud infrastructure. It includes the shelf *name* (string) and *HostingRack* (rack object) that represents the rack hosting the corresponding shelf. If multiple shelves hold similar characteristics and reside on the same rack, a *Shelf_count* can be defined to automate their generation and avoid repetition.

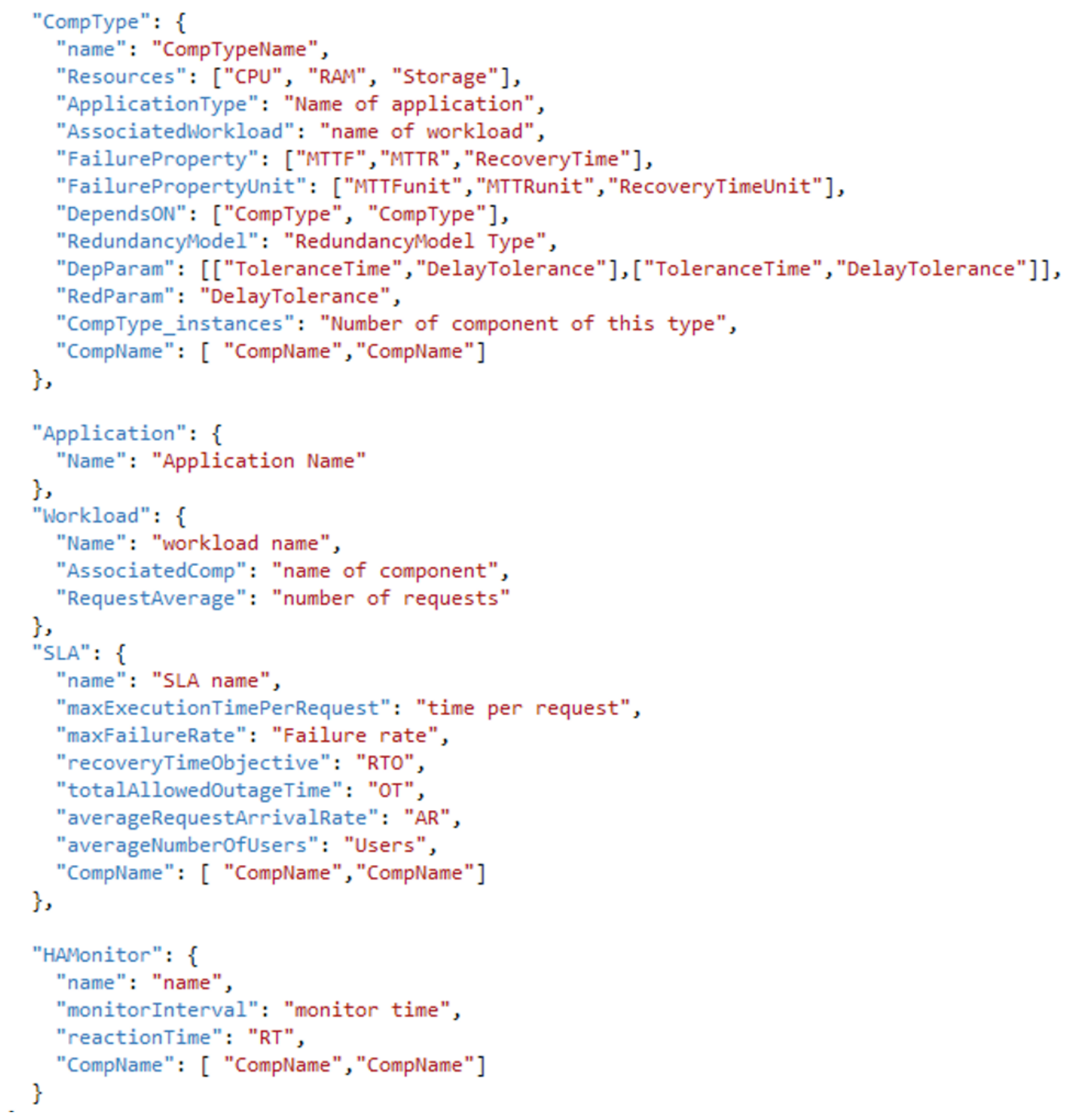
Server: It represents the server details of a given cloud infrastructure. It includes the server *name* (string), *Resources* (array of numbers) that represents its computational resources, *Availability_zone* (string) that hosts it, *HostingShelf* (shelf object) that represents the shelf hosting the corresponding server, *FailureProperty* (array of numbers) that represents its MTTF and MTTR, and *FailurePropertyUnit* (array of strings) that reflects units of MTTF and MTTR. If multiple servers hold similar characteristics and reside on the same shelf, a *Server_count* can be defined to automate their generation and avoid repetition.

3) Virtualization Layer Information:

VM: It represents the VM details of a given cloud environment. It includes the VM *name* (string), *Resources* (array of numbers) that represents its computational resources, *HostingServer* (server object) that represents the server hosting the corresponding VM, and *HostedComponent* (array of multiple component objects) that represents the components hosted on this VM. *HostingServer* and *HostedComponent* properties are populated only if the objective is “evaluation”. Also, the VM includes *FailureProperty* (array of numbers) that represents its MTTF, MTTR, and recovery time, and *FailurePropertyUnit* (array of strings) that reflects units of MTTF, MTTR, and recovery time.

Container: It represents the container details of a given cloud environment. It has same characteristics as the VM except that the *Host* (server and/or VM object) represents the server hosting the corresponding container. As for the *VM* object, it is populated if the container is hosted on a VM, otherwise, it is null.

4) Cloud Application Information:



```
"CompType": {
  "name": "CompTypeName",
  "Resources": ["CPU", "RAM", "Storage"],
  "ApplicationType": "Name of application",
  "AssociatedWorkload": "name of workload",
  "FailureProperty": ["MTTF","MTTR","RecoveryTime"],
  "FailurePropertyUnit": ["MTTFunit","MTTRunit","RecoveryTimeUnit"],
  "DependsON": ["CompType", "CompType"],
  "RedundancyModel": "RedundancyModel Type",
  "DepParam": [["ToleranceTime","DelayTolerance"],["ToleranceTime","DelayTolerance"]],
  "RedParam": "DelayTolerance",
  "CompType_instances": "Number of component of this type",
  "CompName": [ "CompName","CompName"]
},

"Application": {
  "Name": "Application Name"
},
"Workload": {
  "Name": "workload name",
  "AssociatedComp": "name of component",
  "RequestAverage": "number of requests"
},
"SLA": {
  "name": "SLA name",
  "maxExecutionTimePerRequest": "time per request",
  "maxFailureRate": "Failure rate",
  "recoveryTimeObjective": "RTO",
  "totalAllowedOutageTime": "OT",
  "averageRequestArrivalRate": "AR",
  "averageNumberOfUsers": "Users",
  "CompName": [ "CompName","CompName"]
},

"HAMonitor": {
  "name": "name",
  "monitorInterval": "monitor time",
  "reactionTime": "RT",
  "CompName": [ "CompName","CompName"]
}
```

Figure 5.6: JSON-based cloud application template.

CompType: It represents the component type details of a given cloud application. It includes the component type *name* (string), *Resources* (array of numbers) that represents its computational resources, *ApplicationType* (string) and *AssociatedWorkload* (string) that determine the names of the component type application and workload. Additionally, the component type has *FailureProperty* (array of numbers) that represents its MTTF, MTTR, and recovery time, and *FailurePropertyUnit* (array of strings) that reflects units of MTTF,

MTTR, and recovery time. The interaction between component types is also reflected in the template. It has *RedundancyModel* (string) that determines the type of redundancy model (i.e. active/active) and *RedParam* (number) that shows the allowed delay tolerance between the redundant components. It also has *DependsON* (array of multiple component type objects if applicable) that determines the sponsor(s) of the corresponding component type, and *DepParam* (array of numbers) that shows the tolerance time of the corresponding component type and the allowed delay tolerance between the dependent components. The number of *DepParam* sub-arrays is the same as the size of *DependsON*. Finally, the *CompType* determines the number and the names of the components of the same type by populating *CompType_instances* (number) and *CompName* (arrays of strings).
Application: It represents the applications deployed in the cloud and has one property, *name* (string).
Workload: It represents the workload details associated with each component. It includes the workload *Name* (string), the name of associated components, *AssociatedComp* (string), and a number of average requests, *RequestAverage* (number).
SLA: It represents the SLA details associated with applications components. It includes the SLA *name* (string), allowed time per request, *maxExecutionTimePerRequest* (number), acceptable failure rate, *maxFailureRate* (number), allowed recovery time objective, *recoveryTimeObjective* (number), allowed outage time, *totalAllowedOutageTime* (number), average arrival number of requests, *averageRequestArrivalRate* (number), average number of users for each component, *averageNumberOfUsers* (number), and list of monitored components, *CompName* (array of strings).
HAMonitor: It represents the details of HA monitor for applications components. It includes monitor *name* (string), the frequency of monitoring, *monitorInterval* (number), reaction time to handle faulty node(s), *reactionTime* (number), and a list of monitored components, *CompName* (array of strings).

5.4.3 GITS graphical interface

In GITS, it is possible to generate the cloud use cases not only using textual format but using a graphical one as well. In the latter case, users can create scenarios through a graphical interface that implements the syntax of the cloud model at the infrastructure and application

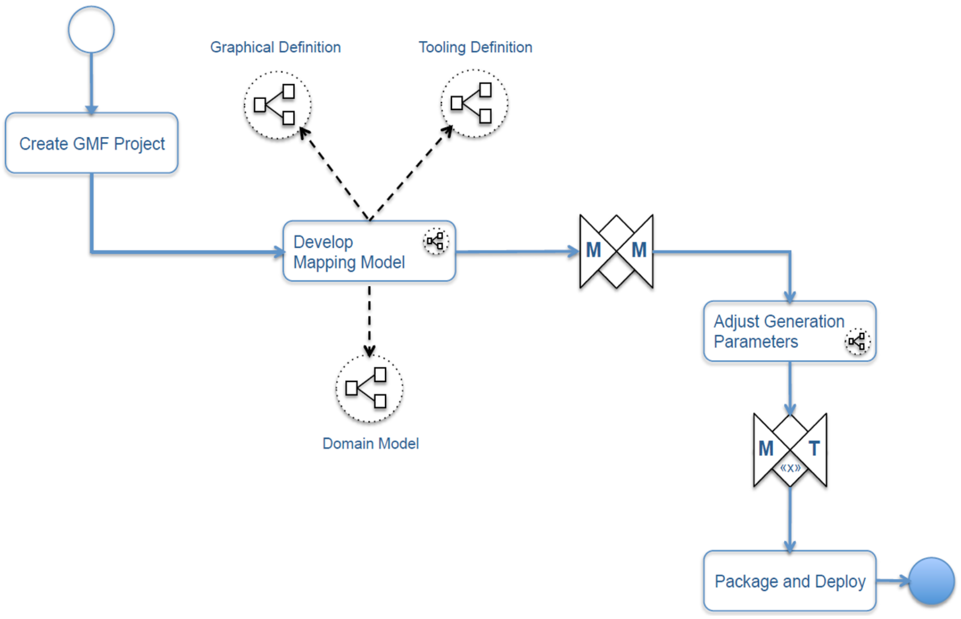


Figure 5.7: Eclipse GMF overview.

levels. The graphical interface is the only interaction with the user, and the transformation of the scenarios to the proper data format of the used cloud simulator happens behind the scene. While the JSON template is a simple data representation and exchange format that constructs the cloud scenarios, the proposed interface allows the users to graphically build their cloud use cases with Graphic Modeling Framework (GMF) interaction. With GMF, a graphical representation of a Domain Specific language (DSL) can be created and mapped to a graphical and textual concrete syntax [48].

Based on the Graphical Editing Framework (GEF) and Eclipse Modeling Framework (EMF), a GMF project provides a model-driven process for developing graphical editors in Eclipse. It has a Model-View-Controller (MVC) architecture that isolates the graphical interface from the domain model, which provides the diagram and domain model, permitting better quality, productivity, and design independency. Fig. 5.7 shows the GMF overview. The required graphical editor has created using Model-to-Model Transformation (M2M) and Model-to-Text Transformation (M2T).

To generate GITS graphical editor, a domain, tooling, graphical, mapping, and generator models are defined to build a functional graphical interface based on the GMF Runtime [49]. The domain model is based on an Ecore model of the above UML class diagram.

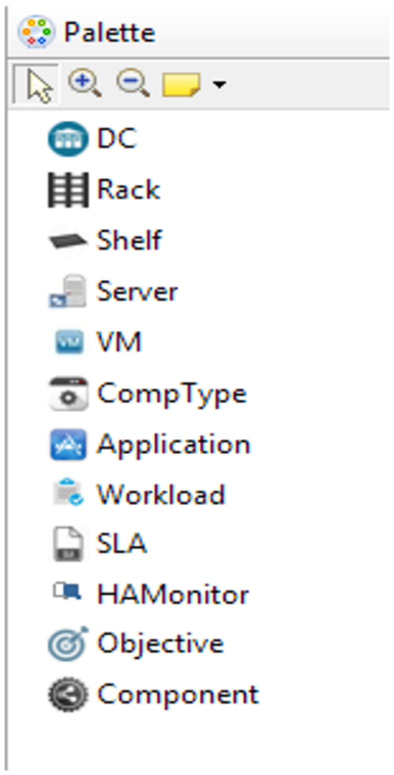
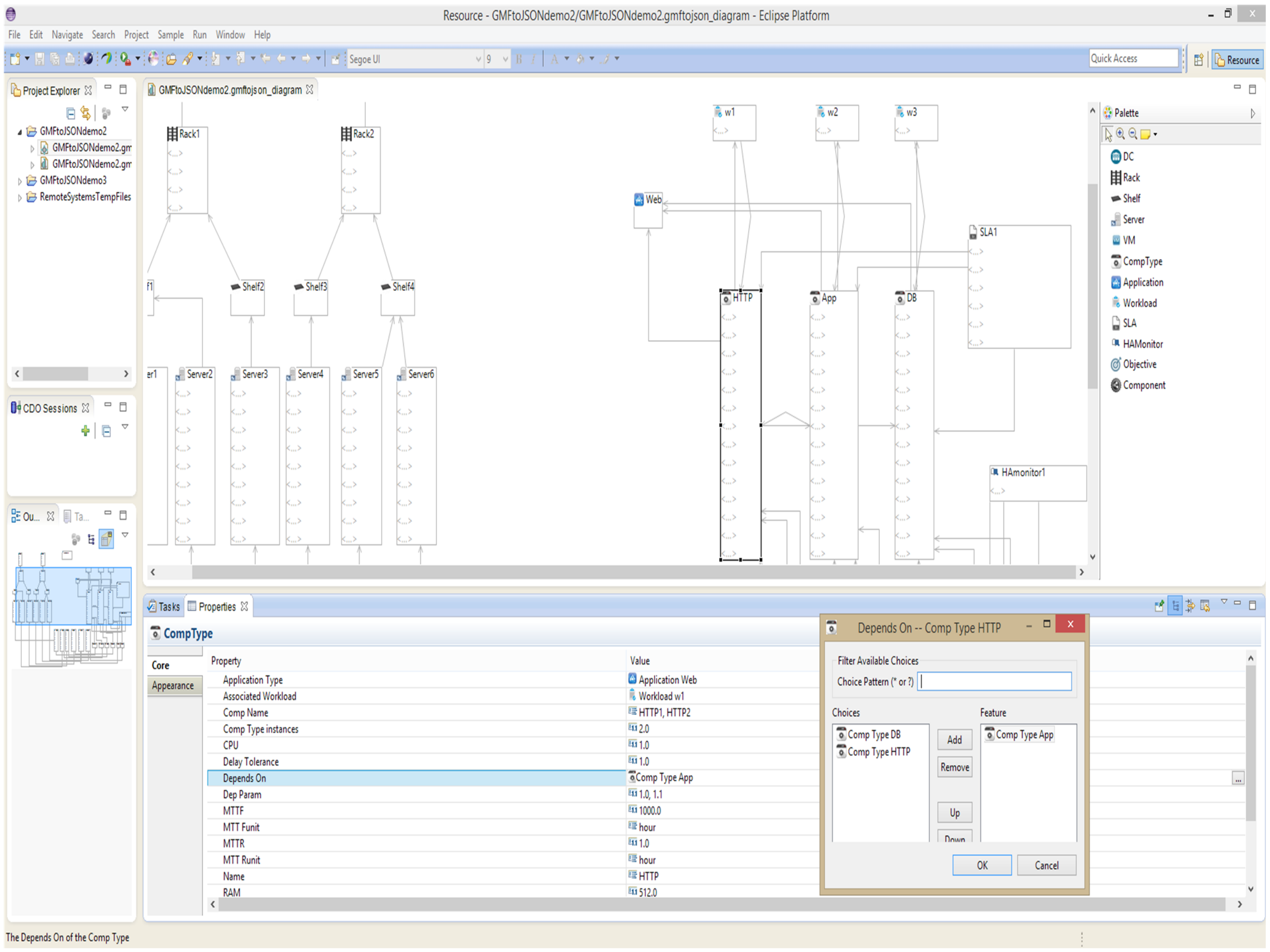


Figure 5.8: GITS tool palette.

Once the Ecore generator model is created, a Graphical Definition Model (GDM) is created that defines the cloud nodes (DC, rack, server, component types, and other nodes) and the connections between these nodes (relationships defined in Ecore model). When the cloud nodes and links are determined, the tool palette of the graphical editor can be created using the Tooling Definition Model (TDM). The TDM describes the cloud elements, their names, and their descriptive icons in the editor palette. Fig. 5.8 shows GITS tool palette. The domain model, GDM, and TDM are combined to generate the Mapping Model. The latter is the base of GMF diagram because it generates the mapping between the nodes, links, and corresponding icons. A successful mapping enables the generation of the desired GMF generator model that produces an extensible graphical diagram based on the GMF runtime [50]. The latter is an industry application framework that bridges the GEF and EMF to create graphical editors. It provides reusable elements such as tool palette, connection handles, and elements properties menu. GITS graphical editor is shown in Fig. 5.9. It is a user-friendly interface where cloud elements can be dragged and dropped from the tool



CompType

Property	Value
Application Type	Application Web
Associated Workload	Workload w1
Comp Name	HTTP1, HTTP2
Comp Type instances	2.0
CPU	1.0
Delay Tolerance	1.0
Depends On	Comp Type App
Dep Param	1.0, 1.1
MTTF	1000.0
MTT Funit	hour
MTTR	1.0
MTT Runit	hour
Name	HTTP
RAM	512.0

Figure 5.9: GITS graphical editor.

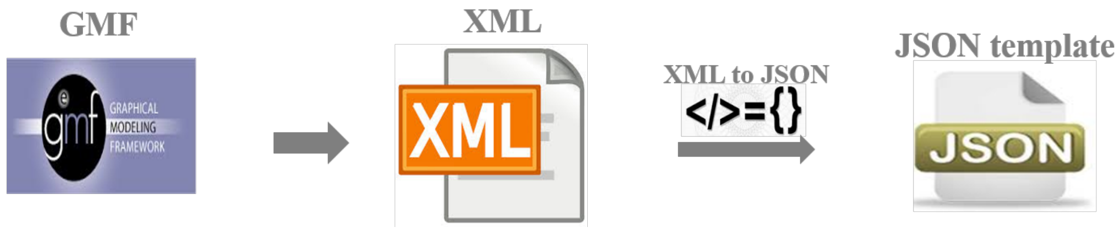


Figure 5.10: GMF2JSON approach.

palette, and their corresponding properties and links are populated in the property panel. Once the cloud elements properties and links are defined, the connections between them are generated automatically.

GMF simplifies the development complexity of a Graphical User Interface (GUI) and reduces maintenance and testing life cycle. It is simple to generate Java codes from the corresponding editor, and the model is stored as an XML file, a standard data exchange format.

5.4.4 GITS transformation algorithm

GITS aims at generating a user-friendly, reusable, and interoperable cloud topology. For this purpose, the users populate the GMF editor with a cloud scenario, and the transformation algorithm ensures the mapping of the graphical scenario into a readable data format by the employed cloud simulator. In this chapter, CloudSim is the cloud simulator that is extended with GITS. To achieve the data transformation, different sub-transformation algorithms are designed:

1) GMF2JSON:

The model generated by the graphical editor can be stored as an XML file. Studies have shown that the JSON files can be efficiently parsed in comparison to XML, and it can replace the XML as the data exchange format used in web applications [51]. Fig. 5.10 shows the GMF2JSON transformation.

Once the XML file is generated, it is inputted to an XML-JSON parser to generate the desired JSON template discussed above. A Document Object Model (DOM) parser is used to create the document builder and examine the nodes, links, and attributes. Jackson 1.x is then used to convert the generated Java objects into JSON data format. In order to create the above JSON template, a mapper algorithm is used to implement the preferred JSON data

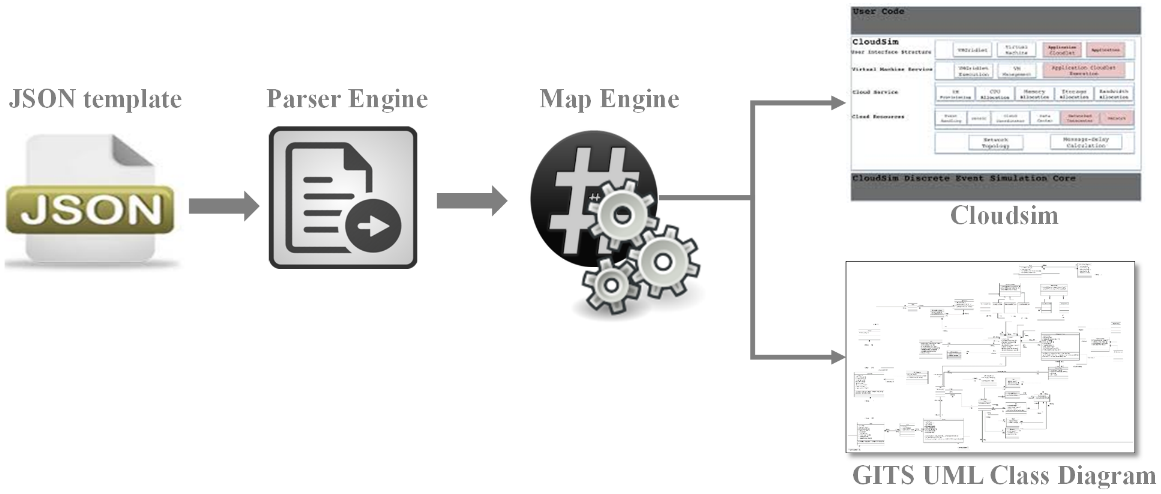


Figure 5.11: JSON2UML2CloudSIM approach.

structure. To this end, the graphical model is transformed to a user-readable and reusable data format.

2) JSON2UML:

In this section, the JSON template is mapped to the above UML class diagram. Using Papyrus, open source UML tool is used to build the cloud UML model. The JSON template is used to populate an instance of the UML model. The objects in JSON file are mapped to Java objects and then mapped to the cloud objects defined in the UML diagram. Fig. 5.11 shows the GMF2JSON transformation.

3) GITS2CloudSimInput:

A Java Archive (JAR file) is used to populate the CloudSim input. The JSON template and the JAR file are inputted to the CloudSim building environment. The CloudSim input can then be populated using the given template. The CloudSim DCs and hosts are populated from the GITS DCs and servers information. As for application level, CloudSim does not model the cloud applications, but it captures the VM/container generation. CloudSim is extended to model the cloud applications and their components as well. The latter is populated from the applications and their component types data that is defined in GITS. The VM/container in CloudSim is populated from the GITS VMs and containers information. As for the other data, such as the SLA and HA monitor, their associated info can be accessed by any scheduling or allocation policy to evaluate certain deployment in terms of

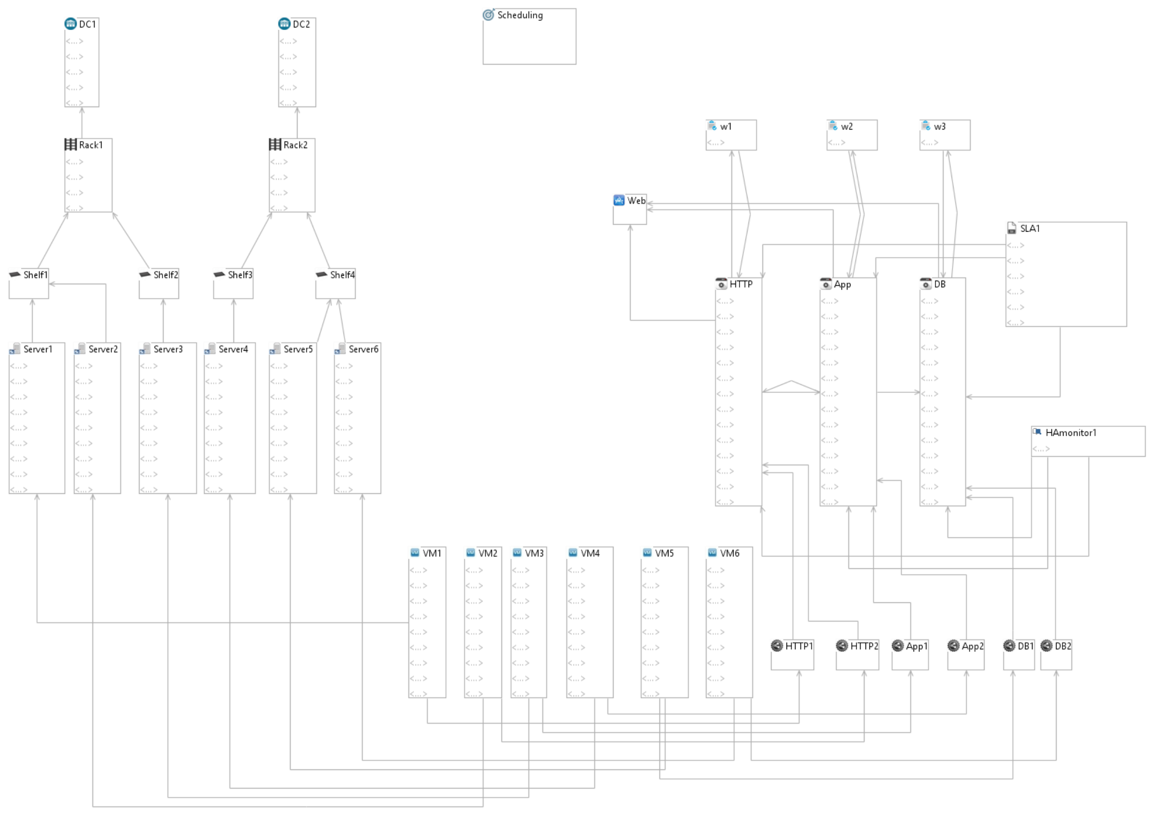


Figure 5.12: Cloud scenario created using GITS.

performance or availability intentions.

Since CloudSim does not support any HA algorithms (failover, redundancy, etc.), some of the template parameters are not used. However, the template and the JAR file can be imported to any cloud simulator that supports performance and/or HA objectives, and the simulator input model is populated accordingly. It is important to note that the JAR file and the JSON template of GITS are available upon request.

5.5 GITS Testbed and Evaluation

In this section, GITS is evaluated on a three-tier web application as a use case for cloud services. The web application consists of active HTTPS server, App logic, and a DB. Each of these component types is backed up with redundant component(s). The functional and protection chains between different component types are also captured as links between different nodes. As for the cloud infrastructure, it consists of two DCs, two racks, four shelves, and six servers. For evaluation purposes, each DC has a rack with two shelves and three servers. The GMF is designed using Eclipse Modeling Tool, Kepler version. In the following, we create cloud scenarios using GITS, test it in CloudSim, and discuss the steps to map GITS to other cloud templates. Fig. 5.12 shows a sample of the cloud scenario generated using GITS GMF.

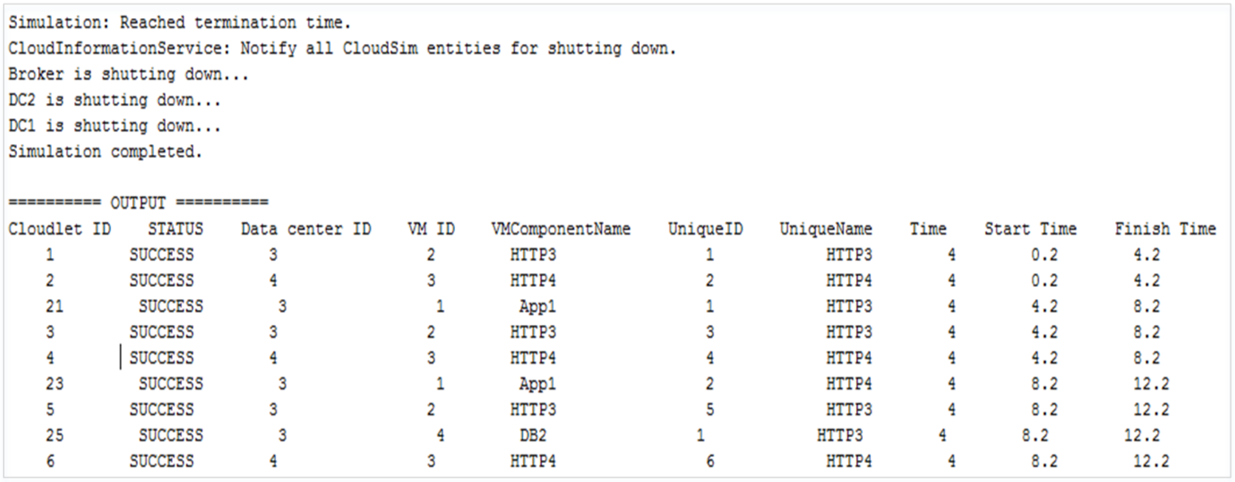
The objective of GITS is to define the configuration information for cloud applications, infrastructure, and interconnection relations and use them to simulate cloud behavior (such as applications scheduling/deployment, VM creation and deletion).

1) Cloud Scenario Creation:

The GMF project is run as a Java application to create a cloud example, but the user can populate the JSON template directly as well. The cloud infrastructure consists of two DCs, each characterized by computational resources and availability metrics (MTTF in hours per year and MTTR in seconds). The availability zone is “Z1”, which means that all the servers of this DC are located in availability zone “Z1”. The *DC_count* is one indicating that only one DC with specified attributes is created. Similarly, the rack is populated with resources and HA features, and each rack should determine its DC. Each of these racks has two shelves, and each shelf has three servers. The server has CPU (cores), RAM (MB), and storage (GB).

In this example, VM is used to represent the virtual mapping between the cloud infrastructure and the cloud applications. Since the objective of this scenario is “scheduling”, the hosting server and hosted component of the VM are populated as “Null”. These properties are populated after triggering an allocation policy of the cloud applications.

The cloud application consists of HTTPS component at the front end, App logic, and DB at the backend. App server sponsors the HTTPS and consequently, the HTTPS component type has *DependsON* property as “App” and the *DepParam* are populated with the corresponding tolerance time and delay tolerance. The same applies to the App component type. Since DB does not have sponsors, its *DependsON* and *DepParam* are “Null”. Each of these types has a redundancy model to back them up upon failure. The redundancy relation is described using *RedundancyModel*, *RedParam*, *CompType_instances*, and *names*. Each component type has its own workload characteristics. It is monitored by an HA middleware and follows an SLA agreement.

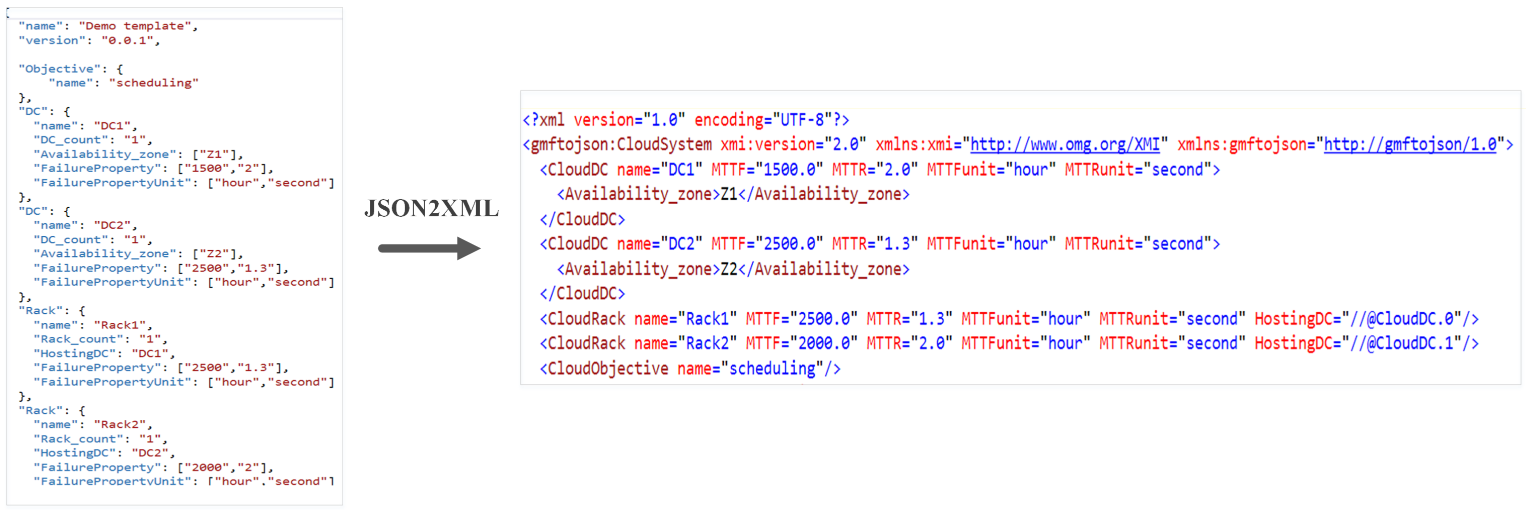


```
Simulation: Reached termination time.
CloudInformationService: Notify all CloudSim entities for shutting down.
Broker is shutting down...
DC2 is shutting down...
DC1 is shutting down...
Simulation completed.

========== OUTPUT ==========
Cloudlet ID    STATUS    Data center ID    VM ID    VMComponentName    UniqueID    UniqueName    Time    Start Time    Finish Time
    1        SUCCESS        3                2        HTTP3              1            HTTP3        4        0.2          4.2
    2        SUCCESS        4                3        HTTP4              2            HTTP4        4        0.2          4.2
    21        SUCCESS        3                1        App1               1            HTTP3        4        4.2          8.2
    3        SUCCESS        3                2        HTTP3              3            HTTP3        4        4.2          8.2
    4       | SUCCESS       4                3        HTTP4              4            HTTP4        4        4.2          8.2
    23        SUCCESS        3                1        App1               2            HTTP4        4        8.2          12.2
    5        SUCCESS        3                2        HTTP3              5            HTTP3        4        8.2          12.2
    25        SUCCESS        3                4        DB2               1            HTTP3        4        8.2          12.2
    6        SUCCESS        4                3        HTTP4              6            HTTP4        4        8.2          12.2
```

Figure 5.13: Evaluation of GITS cloud scenario in CloudSim.

If the user describes the cloud scenarios using GMF, the transformation algorithm, which consists of GMF2JSON, JSON2UML, and GITS2CloudSimInput, is triggered to automate CloudSim population. If the user describes the cloud scenarios using the JSON template, the transformation algorithm, which consists of JSON2UML and GITS2CloudSimInput, automates the CloudSim population. The described cloud example is tested in CloudSim. GITS does not only simplify scenarios creation and models repeatability, it also captures HA properties including redundancy models and HA metrics. We have extended CloudSim to include theses HA features and HA-aware allocation policy [52]-[54]. In the extended CloudSim, application components, their dependents, and redundants are modeled. Therefore, GITS template is evaluated using the extended CloudSim since it supports the application and HA modeling. Fig. 5.13 shows the successful completion of simulation using the above GITS scenario. As seen in Fig. 5.13, each component is hosted on a VM. The extended CloudSim supports the functional chaining; therefore, multiple cloudlets can be created on a VM. The finish time of the processed cloudlet is the start time of the waiting cloudlet in the queue. As the simulation time increases, more cloudlets (requests) are created to model the requests being processed by the application components.

The CloudSim input model is graphically and textually designed as a readable and reusable scenario. CloudSim users do not require experience in the simulator environment and can focus on the simulator features and scheduling extensibility to design solutions that over-



```
{
  "name": "Demo template",
  "version": "0.0.1",

  "Objective": {
      "name": "scheduling"
  },
  "DC": {
    "name": "DC1",
    "DC_count": "1",
    "Availability_zone": ["Z1"],
    "FailureProperty": ["1500","2"],
    "FailurePropertyUnit": ["hour","second"]
  },
  "DC": {
    "name": "DC2",
    "DC_count": "1",
    "Availability_zone": ["Z2"],
    "FailureProperty": ["2500","1.3"],
    "FailurePropertyUnit": ["hour","second"]
  },
  "Rack": {
    "name": "Rack1",
    "Rack_count": "1",
    "HostingDC": "DC1",
    "FailureProperty": ["2500","1.3"],
    "FailurePropertyUnit": ["hour","second"]
  },
  "Rack": {
    "name": "Rack2",
    "Rack_count": "1",
    "HostingDC": "DC2",
    "FailureProperty": ["2000","2"],
    "FailurePropertyUnit": ["hour","second"]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<gmftojson:CloudSystem xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:gmftojson="http://gmftojson/1.0">
  <CloudDC name="DC1" MTTF="1500.0" MTTR="2.0" MTTFunit="hour" MTTRunit="second">
    <Availability_zone>Z1</Availability_zone>
  </CloudDC>
  <CloudDC name="DC2" MTTF="2500.0" MTTR="1.3" MTTFunit="hour" MTTRunit="second">
    <Availability_zone>Z2</Availability_zone>
  </CloudDC>
  <CloudRack name="Rack1" MTTF="2500.0" MTTR="1.3" MTTFunit="hour" MTTRunit="second" HostingDC="//@CloudDC.0"/>
  <CloudRack name="Rack2" MTTF="2000.0" MTTR="2.0" MTTFunit="hour" MTTRunit="second" HostingDC="//@CloudDC.1"/>
  <CloudObjective name="scheduling"/>
```

Figure 5.14: Example of GITS encoding to XML schema.

come other cloud challenges.

2) GITS Encoding:

GITS uses JSON as data exchange schema to define a cloud model. However, this schema can be easily mapped to another encoding format, such as XML and YAML files. Fig. 5.14 shows JSON2XML translation. This ensures the ability to use this template not only for CloudSim input, but it can also be adapted to other cloud providers, simulator, and cloud management systems, such as OpenStack Heat. Heat is an orchestration service for OpenStack that uses template mechanism and control cloud resources groups.

In order to translate GITS to other cloud templates (OpenStack HOT); some key points should be considered:

It is not necessary to use GITS as the base Heat data exchange scheme because the proposed template of Heat can be translated to/from GITS.

- GITS template can be reshaped to meet the standards of HOT. For example, when assigning servers resources, a mapping can be generated between resources number and resources description in HOT (tiny, small, medium, and large instances).
- Multiple JSON-YAML parsers can be adopted in the GITS-HOT translator.
- It is also necessary to determine the relation between stack and OpenStack resources because each module (Nova, Cinder, and Compute) requires different properties defined in the stack parameters section.

The encoding method of cloud model can be easily modified to meet certain cloud system. As long as the template captures the properties needed to manage cloud infrastructure and applications, the translation method can be straightforwardly implemented.

5.6 Conclusion

The design of cloud template that provides simplicity, understandability, repeatability, and interoperability is a paramount step in cloud design to fully exploit its benefits. To this end, it is necessary to define a component-based architecture that describes the cloud infrastructure and applications parameters and enables applications congurability between different cloud platforms. This architecture leverages the challenge of cloud scenarios development, testing, and maintenance. Therefore, in this chapter, we proposed GITS to enable the above features and reduce the complexity of understanding different cloud technologies. In this chapter, graphical and textual interfaces were presented. The graphical interface is defined in GMF to ensure the cloud scenarios visualization by any user without prior knowledge of any data schemas. The textual interface is represented by JSON schema, but it can be encoded in any other alternative data exchange format such as HOT template and XML. JSON format is used for its simplicity, readability, and ability to enable repeatable cloud models. GITS is mapped to CloudSim using a transformation algorithm, but it can be easily translated to fit any cloud simulator or cloud management input.

References

[1] M. Jammal, H. Hawilo, A. Kanso, and A. Shami, “GITS: Generic Input Template for CloudSim and Cloud Simulators,” *Submitted to Elsevier Future Generation Computer Systems*, 2017.

[2] M. Jammal, A. Kanso, and A. Shami, “High Availability-Aware Optimization Digest for Applications Deployment in Cloud,” *IEEE International Conference on Communications (ICC)*, pp. 6822-6828, June 2015.

[3] M. Jammal, A. Kanso, and A. Shami, “CHASE: Component High Availability Scheduler in Cloud Computing Environment,” *IEEE International Conference on Cloud Computing (CLOUD)*, pp. 477-484, 2015.

[4] H. Hawilo, A. Kanso, and A. Shami, “Towards an Elasticity Framework for Legacy Highly Available Applications in the Cloud,” *IEEE World Congress on Services (SERVICES)*, pp. 253-260, July 2015.

[5] Google, “Choosing an App Engine Environment,” `https://cloud.google.com/appengine/docs/the-appengine-environments`, November 2016. [January 15, 2017]

[6] Microsoft Azure, “Microsoft Azure: Cloud Computing Platform & Services,” `https://azure.microsoft.com/en-us/?b=17.05`, 2017. [February 2, 2017]

[7] Datapipe, “Public, Private, Hybrid: Understanding Your Cloud Options,” *https://www.datapipe.com/cloud/*, 2016. [January 20, 2017]

[8] Amazon, “Amazon EC2,” `https://aws.amazon.com/ec2/`, 2017. [February 5, 2017]

[9] R.N. Calheiros, R. Ranjan, A, Beloglazov, C.A.F. De Rose, and R. Buyya, “CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms,” *Software: Practice and Experience Journal*, January 2011, pp. 23-50.

[10] R. Buyya, R. Ranjan, and R. N. Calheiros, "Modeling and simulation of scalable Cloud computing environments and the CloudSim toolkit: Challenges and opportunities," *International Conference on High-Performance Computing & Simulation*, pp. 1-11, 2009.

[11] S. Strauch, V. Andrikopoulos, T. Bachmann, and F. Leymann, "Migrating application data to the cloud using cloud data patterns," *CLOSER*, 2013.

[12] OASIS, "Topology and orchestration specification for cloud applications version 1.0," `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html`, November 2013. [December 2016]

[13] A.F. Antonescu, P. Robinson, and T. Braun, "Dynamic topology orchestration for distributed cloud-based applications," *Second Symposium on Network Cloud Computing and Applications (NCCA)*, pp. 116-123, 2012.

[14] G. Juve and E. Deelman, "Automating application deployment in infrastructure clouds," *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 658-665, 2011.

[15] C. Liu, J. E. V. D. Merwe, and et al., "Cloud resource orchestration: A data-centric approach," *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[16] IBM, "IBM SmartCloud Orchestrator Architected for Extensibility," `http://www.iaas.uni-stuttgart.de/lehre/vorlesung/2013_ws/vorlesungen/smcc/materialien/SCOrchestrator%20Extensibility%20Architecture%201105.pdf`, November 2013. [December 2016]

[17] IBM, "Orchestration Simplifies and Streamlines Virtual and Cloud Data Center Management," `https://goo.gl/8dEqle`, January 2015. [December 2016]

[18] Amazon, "AWS CloudFormation Templates," `https://aws.amazon.com/cloudformation/aws-cloudformation-templates/`, 2017. [Februay 2017]

[19] OpenStack, "Heat Orchestration Template (HOT) Guide," `http://docs.openstack.org/developer/heat/template_guide/hot_guide.html`, 2013. [February 2017]

[20] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable cloud services using Tosca," *IEEE Internet Computing*, 2012.

[21] M. Carlson et al., "Cloud Application Management for Platforms," `https://www.oasis-open.org/committees/download.php/47278/CAMP-v1.0.pdf`, August 2012.

[22] OpenTOSCA, "Open Source TOSCA Ecosystem," `http://www.iaas.uni-stuttgart.de/OpenTOSCA/`, June 2016. [February 2017]

[23] OpenTOSCA, "Winery tool," `http://winery.opentosca.org/winery/servicetemplates/`, November 2013. [January 2017]

[24] O. Kopp, T. Binz, U. Breitenbcher, and F. Leymann, "Winery-A Modeling Tool for TOSCA-based Cloud Applications," *11th International Conference on Service-Oriented Computing*, pp. 700-704, December 2013.

[25] Eclipse, "Eclipse Winery," `https://projects.eclipse.org/projects/soa.winery`, 2017. [February 2017]

[26] W. Tian, Y. Zhao, M. Xu, Y. Zhong and X. Sun, "A Toolkit for Modeling and Simulation of Real-Time Virtual Machine Allocation in a Cloud Data Center," *IEEE Transactions on Automation Science and Engineering*, January 2015, pp. 153-161.

[27] M. C. S. Filho and J. J. P. C. Rodrigues, "Human Readable Scenario Specification for Automated Creation of Simulations on CloudSim," *Springer International Publishing*, 2014.

[28] B. Wickremasinghe, R. N. Calheiros, and R. Buyya, "CloudAnalyst: A CloudSim-Based Visual Modeller for Analysing Cloud Computing Environments and Applications," *24th IEEE International Conference on Advanced Information Networking and Applications*, pp. 446-452, 2010.

[29] P. Jakovits, S. N. Srirama, and I. Kromonov, "Stratus: A Distributed Computing Framework for Scientific Simulations on the Cloud," *IEEE 14th International Conference on High-Performance Computing and Communication & IEEE 9th International Conference on Embedded Software and Systems*, pp. 1053-1059, 2012.

[30] S. Srirama, O. Batrashev, and E. Vainikko, "SciCloud: Scientific Computing on the Cloud," *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 579-580, 2010.

[31] S. Guo, F. Bai and X. Hu, "Simulation software as a service and Service-Oriented simulation experiment," *IEEE International Conference on Information Reuse & Integration*, pp. 113-116, 2011.

[32] M. Balmer et al., "MATSim-T: Architecture and Simulation Times," *Multi-Agent Systems for Traffic and Transportation Engineering*, 2009, pp. 57-78.

[33] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzewicz, "SUMO-Simulation of Urban Mobility An Overview," *Third International Conference on Advances in System Simulation*, 2011.

[34] P. Altevogt, W. Denzel, and T. Kiss, "Cloud Modeling and Simulations," *IBM Research Report*, April 2013.

[35] Michael Armbrust et al., "Above the Clouds: A Berkeley View of Cloud Computing," *University of California, Berkeley Technical Report*, February 2009.

[36] J. Hillston, "SimJava," `http://www.inf.ed.ac.uk/teaching/courses/ms/notes/note12.pdf`, November 2002.

[37] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten and B. N. Jorgensen, "Dynamic and selective combination of extensions in component-based applications," *Proceedings of the 23rd International Conference on Software Engineering*, pp. 233-242, 2001.

[38] Oracle, "JavaBeans Spec," `http://www.oracle.com/technetwork/articles/javaee/spec-136004.html`, 2017. [February 2017]

[39] CORBA, "OMG Specifications," `http://www.omg.org/spec/#MW,January2017`. [February 2017]

[40] Microsoft, "Distributed Component Object Model, `https://technet.microsoft.com/en-us/library/cc958799.aspx`, 2017. [February 2017]

[41] H. Petritsch, "Service-Oriented Architecture (SOA) vs. Component Based Architecture," *Vienna University of Technology white paper*, `http://www.petritsch.co.at/download/SOA_vs_component_based.pdf`, 2006.

[42] X. Wang, “Concept and Implementation of a Graphical Editor for Composite Application Templates,” *Thesis*, `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.465.1373&rep=rep1&type=pdf`, 2010.

[43] Google, “G Suite,” `https://gsuite.google.com/`, 2017. [February 2017]

[44] Salesforce, “Bring your CRM to the future,” `https://www.salesforce.com/crm/`, 2016. [February 2017]

[45] P. Salehi, A. Hamoud-Lhadj, P. Colombo, F. Khendek and M. Toeroe, “A UML-Based Domain Specific Modeling Language for the Availability Management Framework,” *IEEE 12th International Symposium on High Assurance Systems Engineering*, pp. 35-44, 2010.

[46] M. Eriksson and V. Hallberg, “Comparison between JSON and YAML for data serialization,” *Bachelor Thesis*, `http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand11/Group2Mads/victor.hallberg.malin.eriksson.report.pdf`, 2011.

[47] Dojo, “dojox.json.ref,” `http://dojotoolkit.org/reference-guide/1.10/dojox/json/ref.html`, 2017. [February 2017]

[48] R. C. Gronback, “Eclipse Modeling Project-A DomainSpecific Language (DSL) Toolkit, Addison-Wesley Longman, https://sisis.rz.htw-berlin.de/inh2009/12371395.pdf, 2009.

[49] Eclipse, “Graphical Modeling Project,” `http://www.eclipse.org/modeling/gmp/`, 2017. [February 2017]

[50] F. Plante, “Introducing the GMF Runtime,” `http://www.eclipse.org/articles/Article-Introducing-GMF/article.html`, January 2006

[51] D. Peng, L. Cao, and W. Xu, “Using JSON for Data Exchanging in Web Service Applications,” *Journal of Computational Information Systems*, 2011, pp. 5883-5890.

[52] M. Jammal, A. Kanso, P. Heidari, and A. Shami, “Availability Analysis of Cloud Deployed Applications,” *IEEE International Conference on Cloud Engineering (IC2E)*, April 2016.

[53] M. Jammal, A. Kanso, P. Heidari, and A. Shami, “A Formal Model for the Availability Analysis of Cloud Deployed Multi-Tiered Applications,” *3rd IEEE International Symposium on Software Defined Systems*, April 2016.

[54] M. Jammal, H. Hawilo, A. Kanso, and A. Shami, “Mitigating the Risk of Cloud Services Downtime Using Live Migration and High Availability-Aware Placement,” *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, December 2016.

Chapter 6
ACE: Availability-aware CloudSim Extension

6.1 Introduction

Although the cloud computing is not new, it is considered a game-changing concept in the information and communications technology (ICT) fields. The cloud outsources the information technology (IT) infrastructure to cloud provider not only to minimize the management challenges but also to allow new providers to enter the market with different capabilities, infrastructure requirements, and costs. As for the cloud user, they encounter multiple challenges, such as providing automated operational tasks (software upgrades control and management), satisfying service level agreements (SLAs), and other quality of service (QoS) concerns. Studies show that cloud services have evolved to everything or anything as a Service (XaaS), which will be responsible for the growth in the market of the cloud services [2]. The XaaS includes software as a service (SaaS), infrastructure as a service (IaaS), and platform as a service (PaaS) where X denotes “everything/anything” as a service. Although large and medium enterprises have the capitals and manpower to invest in the infrastructure, it is expected that these companies are going to drive the growth of XaaS [3]. Additionally, XaaS will be needed by more enterprises to guarantee the balance between their legacy systems with new versions.

In order to ensure the cloud adoption in many enterprises, different challenges should be addressed. These issues range from compliance and legal concerns, security, interoperability, and other services management issues [4]. However, for the cloud paradigm, availability is one of the key factors to ensure an optimal cloud performance and satisfy QoS and quality

---

The content of this chapter have been submitted for publication [1].

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC-STPGP 447230) and Ericsson Research.

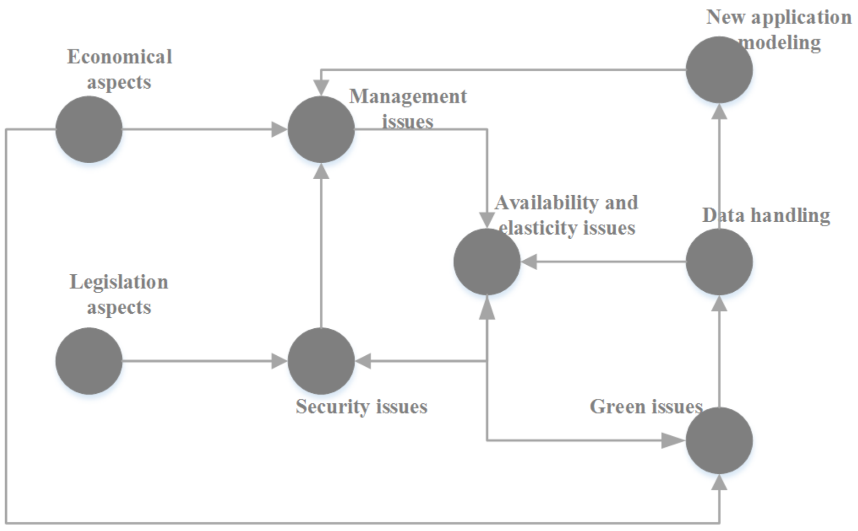


Figure 6.1: Different cloud challenges.

of experience (QoE). Fig. 6.1 shows different cloud challenges and their dependencies. For instance, the data management and applications' availability affect the availability and elasticity issues in the cloud. In fact, cloud services and data are safely stored and maintained in well-managed cloud platforms because the latter has backups and other reliability policies and consequently, is more trustworthy compared to on-premise infrastructure. However, outages can happen even on these platforms. For example, GitLab has faced data loss due to accident deletion on Feb. 1, 2017, which has caused the permanent loss of "six hours' worth" of data [5]. Similarly, Dropbox, Microsoft Azure, Google, and Amazon Web Services have suffered cloud outages in the last few years [6] [7] [8] and [9]. Additionally, according to [10], 86% of IT decision makers determine that high availability (HA) is an important criterion in choosing a cloud service provider. Therefore, availability and reliability are main concerns to be addressed in large distributed systems and applications, mainly cloud platforms [11]. With the emergence of the internet of things (IoT), network function virtualization (NFV), software defined networking (SDN), and Big Data, these HA desires are continually growing in many applications including communication, finance, health, and social networking [12]. Fig. 6.2 shows different emerging technologies in the cloud domain. Different approaches can be adopted to maintain HA in the cloud. This includes backups, redundancy models, failover policies, and HA-aware deployments. It is necessary to note that availability is the measure of the percentage of time a system is

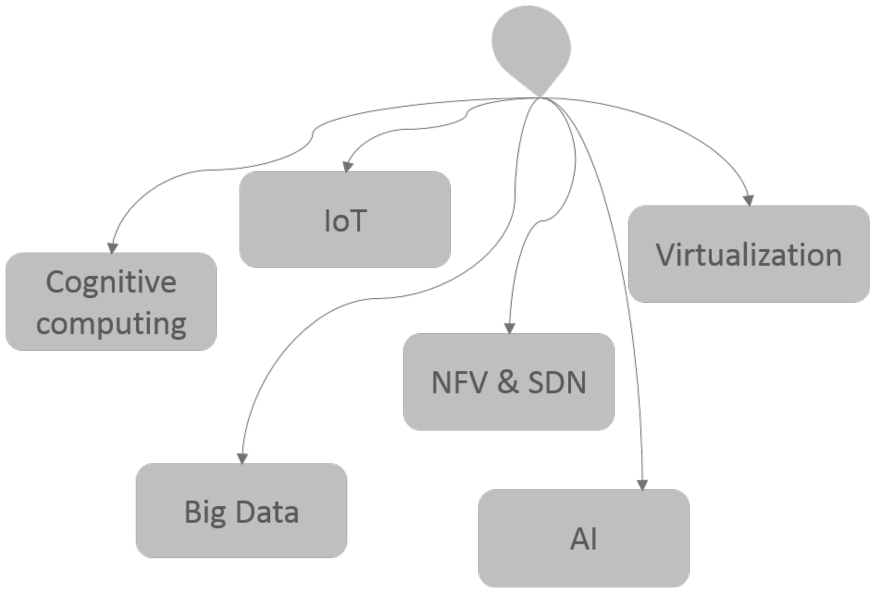


Figure 6.2: Different emerging technologies.

available for normal usage in a given time interval [13].

On the endeavor to ensure a highly available cloud services is to design and implement a cloud model and simulation that emulate real cloud outages and recover them accordingly. Relying on the reliability guaranties of the cloud provider may not be enough to assure the applications of the cloud users will maintain their HA status. Using real cloud settings (i.e. Amazon Elastic Compute Cloud (EC2)) to model applications and service and evaluate their behavior under certain performance policies is restricted by cloud platform configurations and infrastructure. As an alternative, modeling and simulations can be used to model the cloud, build new algorithms and policies, test them before the actual deployment in a real cloud, and enhance the performance of large-scale distributed systems. This can save the tenants significant time and effort and some degree of reassurance about the level of HA they can expect. On the other hand, cloud provider can benefit from the simulations to evaluate new features/extensions to their cloud and check if their offered HA guaranties are realistic.

Due to their scalability and efficiency characteristics, discrete event simulators can be used to in the modeling and evaluation of the distributed systems [14]. CloudSim is a simulation framework used for the scheduling and resource allocation algorithms on cloud infrastructure. CloudSim is built over a discrete event simulator, where discrete events are the simulator triggers. However, the simulator is not designed to model HA constructs and therefore, overlooks the availability and failures of the cloud applications. The adoption of an HA-aware model is indispensable to ensure an accurate modeling and evaluation of the behavior of real cloud environment under a faulty nature.

In this chapter, we extend CloudSim simulator to include high availability constraints, HA-aware policies, and HA metrics. The proposed extension, ACE (Availability-aware CloudSim Extension), allows the injection of failures and failure-dependency between cloud applications, where failures can happen in any cloud entity. The extension supports load balancing and allows the separation between the cloud as a provider consisting of data centers (DCs) and servers, and cloud user where applications components are modeled to form functional chains and protection groups. ACE allows realistic detection of failed entities and provides recovery and repair solutions for the users. With these extensions, the simulated scenarios can be used not only to schedule cloud applications with HA objectives but to evaluate fault tolerant cloud scheduling approaches as well. The extension can be used to adopt reactive, proactive, and adaptive fault tolerant approaches. Any failure type can be injected into the cloud infrastructure and applications as long as it is associated with its failure and repair attributes, such as mean time to failure (MTTF) and mean time to repair (MTTR). Also, ACE uses JavaScript Object Notation (JSON) data format to provide generic and repeatable input templates for cloud simulators, GITS. Through ACE, the CloudSim is extended with the following:

- Input template (JSON-based) for the application and the failure/recover/repair information
- Automated requests generation
- Computational path between cloud applications
- Load balancing module
- Failure injection module
- Recovery/failover module
- Requests processing module
- Repair module

The rest of this chapter is structured as follows. In Section 6.2, the related work is presented for cloud simulators and scheduling approaches for distributed systems. Section 6.3

presents the problem background and motivation where it defines CloudSim, different outages, fault tolerant approaches, scheduling in the cloud, and complexity of cloud models. In Section 6.4, ACE design and implementation are described. Section 6.5 defines the evaluation results of ACE. Finally, Section 6.6 presents the conclusion.

6.2 Related Work

Several research studies address the cloud scheduling approaches in terms of availability, performance, and other QoS objectives. Other literature efforts have investigated the cloud behavior and tackled cloud simulators designs and implementations.

6.2.1 Cloud simulators

Wickremasinghe et al. propose CloudAnalyst as an extension to CloudSim [15]. The main feature of CloudAnalyst is the graphical user interface (GUI) extension. In other words, the proposed simulator can separate the scenarios creation from the simulation development. It can be applied to large-scale applications and allows simulations repetition while chaining of experiments parameters. CloudAnalyst extends CloudSim with a module for visualizing the simulation results where the simulation settings can be saved as an Extensible Markup Language (XML) file and the results can be exported as a Portable Document Format (PDF) file. Although CloudAnalyst focuses on modeling simulations rather than development, it supports neither HA-aware metrics nor a generic input template with HA features.

Garg et al. extend CloudSim with NetworkCloudSim to include network model for DCs [16]. NetworkCloudSim models the DC network (DCN) in terms of latencies and sharing of bandwidth (BW) thus allowing modeling of different topologies of DCNs. It allows the design of efficient resource allocation, management, and scheduling algorithms, but it is limited to small DCN because of high simulation time and memory restrictions. Besides, NetworkCloudSim discards the HA considerations in terms of simulation scenarios, network design, and placement approaches.

Kliazovich et al. propose GreenCloud as an energy-aware simulator for cloud DCs [17]. It extends the Ns 2 network simulator. GreenCloud models the energy consumption of cloud infrastructure (DCs, servers, and network links) and packet-level communication configurations. GreenCloud differentiates between computing energy, infrastructure energy consumption, and communication energy, to ensure detailed modeling of energy in the cloud DCs. Gupta et al. propose Green Data Center Simulator (GDCSim) as another energy-aware simulator to model DC behavior and resource management in terms of power objectives [18]. GDCSim consists of a BlueSim module to generate simulation scenarios using XML files and performs heat circulation studies to thermally evaluate DCs using Computational Fluid Dynamics (CFD) simulations. Although energy and HA are two main concerns in the cloud, GreenCloud and GDCSim exclude any HA modeling in the cloud DCs.

Zhou et al. extend CloudSim with FTCloudSim to include reliability mechanisms [19]. It evaluates the system performance under faulty events and generates the necessary details to determine the pros and cons of the approach under evaluation. FTCloudSim supports reactive fault tolerant mechanisms, such as checkpointing module and repairing mechanisms. Although FTCloudSim supports some reliability features, it discards redundancy between applications components as well as the dependency relations, which is highly affected by any failure event. It does not support the automated generation of requests within the functional chain of a certain application. Also, the recovery policies do not ensure a failover to a redundant component and do not trigger the repair policy of the faulty component.

Calheiros et al. design EMUSIM on the top of CloudSim and Automated Emulation Framework (AEF) [20]. It uses the application behavior to extract information and generate the simulation scenarios accordingly. Tighe et al. propose Data Center Simulator (DCSim) to evaluate different DC management and scheduling algorithms [21]. It is a Java-based event-driven simulator to model DC providing IaaS to cloud users. Although DCSim models multi-tier applications and supports the dependency and replication simulations between virtual machines (VMs), it discards other HA features (failure injection, repair, recovery, and load balancing).

Lim et al. propose MDCSim as a discrete event simulator to model the infrastructure characteristics of different components of a DC (links, switches, and servers) [22]. Ostermann et al. propose GroudSim as Grid and Cloud simulator based on discrete events [23]. According to different distribution functions, GroudSim can simulate the execution of jobs on computing resources and calculate the associated cost and workload. Sriram proposes Simulation Program for Elastic Cloud Infrastructures (SPECI) to explore scalability of cloud

DCs [24]. It evaluates DCs behaviors in terms of a given design scheme to explore new aspects of future scalability. Although SPECI proposes scalability suggestions while considering failure rate of DCs, it discards any other HA impact on the evaluation process.
While Fittkau et al. extend CloudSim with CDOSim to simulate SLAs violations, response time, cost, and other performance granularities of a cloud deployment option (CDO) and choose the effective deployment accordingly [25], TeachCloud introduces Rain workload generator framework [26]. TeachCloud provides a GUI for generating cloud infrastructure scenarios and visualizing simulation results. Both simulators overlook the HA features in terms of attributes, modeling, and cloud applications scheduling.

6.2.2 Scheduling approaches in distributed systems

Ta-Shma et al. propose a continuous data protection and live migration-based checkpointing algorithms to enhance the VMs availability [27]. This approach reverts VM state to recover any operator error. While Ta-Shma et al. use data protection and live migration checkpointing schemes [27], Wang et al. provide a checkpointing technique that stores periodical checkpointing using a Copy-on-Write-Basic (CoW-B) mechanism [28]. Malik et al. address the cloud failures by proposing an adaptive fault tolerance approach [29]. Based on the VM reliability level, the approach validates if it is removed or not from the cloud infrastructure. The approach consists of a VM node that triggers the application algorithm and an adjudicator node to check and assess the reliability of VM. While Cully et al. propose Remus to achieve HA using asynchronous VM replications [30], Nakano et al. provide ReVivel Input/output (I/O) undo and redo approach to deal with I/O in an HA recovery servers [31]. Although these approaches attempt to improve cloud availability, they overlook many of HA metrics and constraints including MTTF, the dependency between different cloud applications/VMs, load balancing, protection groups, location and anti-location metrics.
Qureshi et al. implement different load balancing techniques [32]. A load balancing approach tends to enhance request response time while preventing the overloading state. Yiqiu et al. and Sadhasivam et al. propose task scheduling scheme based on load balancing in the cloud [33] [34]. The scheduler ensures load balancing of the tasks from an application to a VM according to required resources. Then it enables load balancing from the VM

to a server that satisfies the resources demands. Wang et al. provide a three-level cloud network consisting of a service node, service, and request managers [35]. This approach supports Opportunistic Load Balancing (OLB) in the scheduling algorithm. While Gahlawat et al. evaluate the performance of cloud scheduling approaches using first come first serve (FCFS) and Shortest Job First schemes [36], Pawar et al. propose a dynamic cloud resource scheduling approach [37]. James et al. propose a weight load balancing algorithm in CloudSim [38]. According to the processing power, VMs are weighted accordingly to process users' requests. While James et al. use weight load balancing [38], Tawfeek et al. propose an ant colony optimization model to map users requests to the best-fit VMs [39]. Although these approaches support load balancing, one of the HA mechanisms, they discard other availability constraints, such as dependency relations, redundancy models, affinity and anti-affinity restrictions.

Jin et al. propose fault detection and recovery approaches in the Grids [40]. Through monitoring and checkpointing, fault detection and recovery mechanisms are achieved. Cox et al. propose loosely synchronized redundant virtual machines (LSRVM) approach to address the hardware fault tolerance using virtualization [41]. While Chun et al. build a prototype that serves users' requests using time-based CPU sharing [42], Garg et al. propose cost and time-based resource allocation [43]. Both studies overlook the HA constraints during the allocation process.

The literature has many workflow management approaches that are extended to address cloud resources utilization [44], [45], and [46]. However, these approaches are limited in terms of availability of cloud applications in contrary to the work proposed in this chapter. Unlike other literature studies, we distinguish ourselves in this chapter with a unique well-defined availability-aware extension of CloudSim simulator. The extension does not only capture an HA-aware input templates for the simulator, but it supports different HA metrics and features. This includes failure injection module, applications components recovery and repair, HA-aware allocation mechanism, automated request generation to maintain application functional chains, and load balancing. Besides, the ACE captures different redundancy models and multiple distributions functions for the failure/repair rates.

6.3 Background and Motivation:

Unlike on-premise systems, the cloud is an ecosystem that can be accessed anytime and anywhere. The cloud is considered a provisioning and management paradigm that does not depend on a specific technology. It is characterized by different economic, technical, and non-functional properties, such as multi-tenancy, data management, elasticity, reliability, agility, quality of service, the return on investment (ROI), and pay per use [47]. The cloud provides wider functionality and lines of business (LoB) options while reducing maintenance and licensing costs and replacing the capital expenditures (CAPEX)-based models of organization infrastructure by operational expenditures (OPEX)-based models. In contrary to monolithic on-premise IT infrastructure, the cloud has an elastic nature in a way that it can expand according to the enterprises' needs.

However, multiple challenges arise from the above cloud properties when attempting to achieve them. According to [48], 82% of enterprises have a hybridized cloud model that integrates legacy on-premise systems with private and public cloud solutions. With the hybrid model and the increase dependency on the cloud, different concerns are facing the cloud adoption including security issues, the absence of expertise, growing cloud costs, and outages impacts. HA remains one of the primary dilemmas to be addressed in any cloud solution. Realizing an HA-aware cloud system entails an intricate planning. However, to design a cloud solution that alleviates the HA issues, a modeling and simulation environment is needed to model several cloud properties, such as availability, security, and energy. A simulation environment can be applied to evaluate multiple scenarios under different performance and HA constraints/limitations. Therefore, it is necessary to realize the cloud simulator to model HA solution, the different failure natures and fault tolerant types, the characteristic of a well-defined scheduling, and the nature of cloud applications.

6.3.1 CloudSim simulator

CloudSim is an extensible cloud-based simulator built in the CLOUDS Laboratory at the University of Melbourne, Australia. It models and simulates cloud systems including infrastructure (DCs and servers), VMs, computational resources, and different scheduling and allocation policies [49], [50], and [51]. Many of the existing simulators of the distributed systems are extensions of CloudSim, such as WorkflowSim, CloudSimEx, Simple-

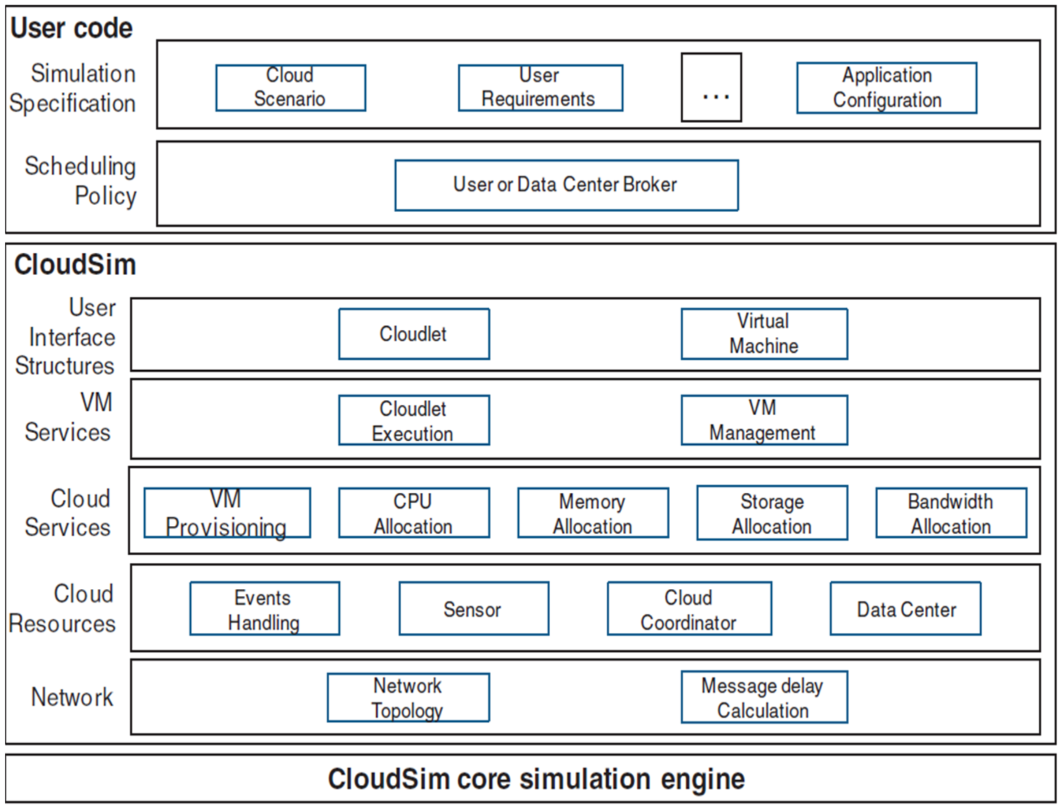


Figure 6.3: CloudSim architecture.

Workflow, CloudReports, and CloudAnalyst [52]. This is because CloudSim implements common provisioning schemes that can be easily extended. With CloudSim, researchers can discard the complexity of event-driven modeling and can focus on evaluating certain cloud objectives (energy and HA) [15]. CloudSim is an open-source simulator and is built on the top of a discrete event simulator, SimJava [15] [53]. CloudSim supports the following features:

- Simulation and modeling of cloud environments, such as DCs, hosts (servers), VMs, and containers.
- Modeling cloud information systems, cloud broker, and different time and space-shared allocation and provisioning policies.
- Simulation and modeling of network connectivity between different cloud entities.

The CloudSim components interact with each other using a message passing technique. Fig. 6.3 shows the CloudSim architecture. The lower layer represents the core simula-

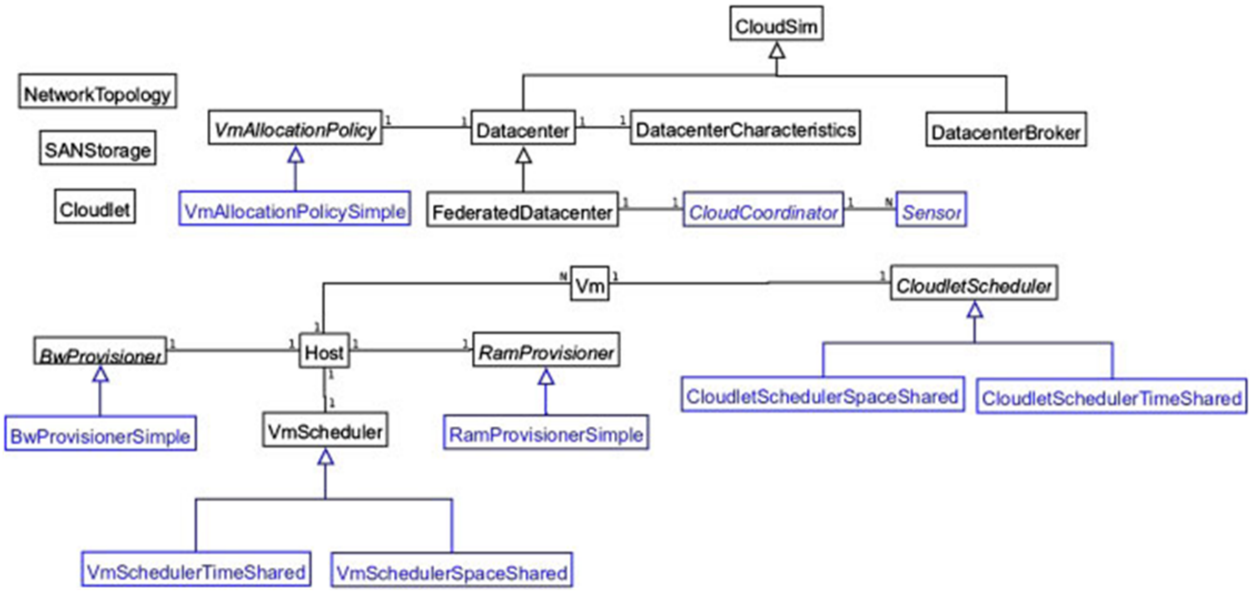


Figure 6.4: CloudSim class diagram.

tion engine that provides event-based functionalities including events processing, queueing, cloud entities creation/pausing/deletion, cloud components interactions, and the simulation clock. The CloudSim layer contains the cloud model entities and corresponding allocation/scheduling approaches. The User Code layer defines the number of users, broker specifications, and configurations for hosts, applications tasks, and VMs. Fig. 6.4 shows the CloudSim class diagram. These classes are extensible and represent the main building blocks for generating cloud models.

Although CloudSim is a toolkit for modeling and simulating cloud use cases, it does not support availability-aware properties, constraints, and/or allocation policy. Also, it does not support a "ready-to-use" setting to generate cloud scenarios, but it needs a Java-based code to create any cloud set-up using its components (DC, host, broker, VM, and allocation policies). Therefore, this chapter aims at extending CloudSim with HA features and generic input template for creating cloud scenarios while ensuring repeatability, portability, understandability, and simplicity.

6.3.2 Outages and fault tolerant approaches

Service outage does not only affect the QoE, but it is realized also as revenue losses. For example, according to the International Working Group on Cloud Computing Resiliency

(IWGCR), Cisco, GitHub, and Facebook outages result in loss of 200,000 USD per hour [54]. To alleviate these challenges, it is necessary to build an HA system that integrates different approaches including redundancy, failover, auto-scaling, monitoring, and load balancing. In order to determine the right HA solution for a certain cloud environment, the failure types, impacts, and associated fault tolerance types should be clearly realized to extract lessons that improve an HA solution. For instance, on January 10, 2014, a script bug in Dropbox causes a reinstallation of some active machines, which affects the replica components and brings the service down [55]. A verification layer is added to Dropbox HA solution to alleviate such kind of failures. Also, on January 24, 2014, some Google services (Gmail and Google Docs) face an outage for one hour due to a software bug in the configuration-based system [55]. Afterward, Google updates the HA solution to include additional validation tests and enhanced failure detection and analysis module. In any case, failure can happen due to planned or unplanned outage [56] [57], but the organization should assess the failure and remodel their management, analysis, and recovery strategies [58].

1. Faults types

System failures can be a transient or permanent fault (hardware level), a bug/design error (software level), an operator error, and/or external errors/faults. In a cloud system, faults are realized as resources failures whether the resource is application or infrastructure. The common two main types of failures behaviors in the cloud are:

Fail-stop/Crash failures: The component of a system changes to a failure state that can be detected by other system components [59]. In other words, the faulty component is halted as in the case of power outages.

Byzantine failures: Upon a failure, the component shows malicious and random behavior, which sometimes collides with other components and causes the system to perform in an arbitrary mode (unpredictable outputs) [59]. Byzantine failure is considered the worst-case scenario due to its disruptive property. Therefore, any system should be designed to overcome such failures.

With these two behaviors, different failures can occur in a system.

Residual defects at the application and infrastructure level can generate errors that escalate to critical failures. If recovery solution fails to happen, a cascading failure is triggered.

For example, if a database server fails and is not quickly recovered, the dependent components, such as App logic in a web application, will fail as well. A myriad of failures cases can be major, such as failure of critical application component or minor outage due to planned upgrade of system software. In any case, noticeable failures are conceived as periods of service degradation and affect the HA metrics calculation. Therefore, any HA solution should not overlook the availability metrics when performing any deployment, redundancy, or failover solution.

2. Fault tolerance measures and policies

Any cloud solution should be designed to tackle or prevent any failure. A system is considered a fault-tolerant one if it continues to function normally in a sense that some components of the system are faulty during the specific time interval. Fault-tolerance or availability of a system is expressed in terms of MTTF and MTTR where MTTF determines the time in which the system functions normally before failure, and MTTR is the time needed to resume the functionality of a failed system. The availability A is calculated as follows:

$$A = \frac{MTTF}{MTTF + MTTR} \qquad (1)$$

Fault tolerance policies can be represented in terms of three different types [60]:

Reactive fault tolerance: When the failure occurs, this policy is used to leverage its impact on the execution of the system component. Replay-and-retry, replication, task-resubmission, and checkpointing are examples of reactive fault tolerance techniques.

Proactive fault tolerance: It aims at preventing failures/errors recovery by predicting them and replacing a faulty component with a normal one. Software rejuvenation, self-healing, load balancing, and preemptive migration are examples of proactive fault tolerance techniques.

Adaptive fault tolerance (AFT): It adapts to the components changes/states and improves the fault tolerance policy accordingly. In a cloud environment, this technique monitors the cloud state and reshapes its configurations to maintain its stability upon fault detection. Byzantine fault tolerance cloud, intermediate data fault tolerant (IFT), MapReduce fault tolerance with low latency, and adaptive anomaly detection system for cloud computing

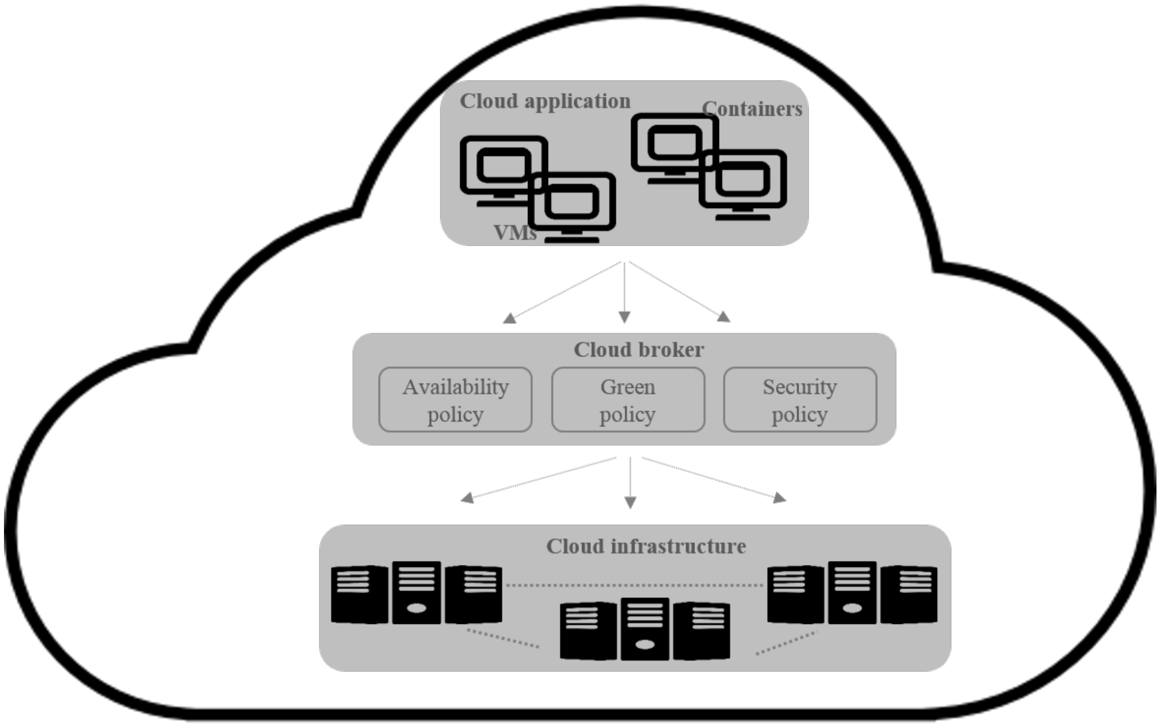


Figure 6.5: Different roles in the cloud model.

infrastructures (AAD) are examples of AFT techniques [61] [62] [63] and [64].

6.3.3 Scheduling in the cloud

To ensure the fully-exploitation of cloud capabilities, it is necessary to design an HA-aware solution while maintaining an efficient utilization of computational resources. Each cloud DC hosts thousands of servers with hundreds of VMs. While VMs process multiple tasks, the cloud receives new batches of users' requests. In order to have a seamless processing, these requests should be hosted by the VM/server that can satisfy computational needs while maximizing their availability. Therefore, task scheduling and assignment are paramount approaches to prevent any SLA violation in terms of HA and performance of the cloud. Optimization models can be an option to perform task-host assignments, but they are generally characterized by Non-Polynomial (NP) complexities including long processing time to search and find optimal solutions [65] [66]. Instead, scheduling heuristics can be used in the cloud to perform the assignment while finding near-the-optimal results. Many scheduling algorithms are used in the cloud environment including Round-Robin, Min-Min, First come First serve (FCFS), Min-Min, and meta-heuristic algorithms (Tabu search, simulated annealing, and genetic algorithm (GA)) [67].

The scheduling aims at maximizing the cloud utility through well-defined metrics that generate statements regarding certain cloud allocation policies. With scheduling, different cloud metrics and objectives can be evaluated in terms of each other (HA-energy-security or HA-performance-fairness) to generate a tradeoff that satisfies the desirable SLA and QoS. In order to perform scheduling in a cloud environment, different phases should be executed:

- Determination phase: Defining type of “to-be-processed” requests/task, such as rigid tasks (predefined resources by users), evolving tasks (changeable resources through simulation), and moldable tasks (constrained resources by the scheduler) [68].
- Discover phase: Resource/HA/Energy-based pooling and filtering of available infrastructure
- Decision phase: Choosing target host (DC, server, and VM)
- Process phase: Submitting the request/task to the host to be processed.

In this chapter, the scheduling and allocation policies in the CloudSim are extended to include HA attributes and constraints (affinity and anti-affinity restrictions, geo-redundancy and dependency models).

6.3.4 Cloud model

Similar to Service Oriented Architecture (SOA), different roles can be defined in any cloud environment [47]. Fig. 6.5 shows the cloud model. These roles can be distributed as follows:

Cloud provider offers PaaS and IaaS to the users. It consists of multiple DCs hosting thousands of servers. Each infrastructure component is characterized by its resources and HA metrics.

Cloud broker is an intermediate negotiator between the cloud service provider and consumer.

Cloud aggregators combine different cloud providers’ platforms to offer a larger and hybrid infrastructure to cloud customers. Aggregators aim at achieving economy of scale by matching the emerging the industry needs and the customer demands by offering customized cloud services.

Cloud users consist of multiple applications components that use the cloud capabilities to execute certain computations or to process requests. These components are characterized by different dependency and redundancy relations. A 3-tier web application is an example of cloud applications [69]. At the front-end, a Hypertext Transfer Protocol Secure (HTTPS) server processes requests and forwards them to an App server. At the back-end, a database (DB) server stores the users' data and sponsors the App server that generates the required information. The dependency interaction between these component types constitutes the functional/computational path that should be followed by a request to be successfully executed.

Even though, the cloud remains to be a complex system where cloud providers should maintain the service delivery while isolating the underlying infrastructure complexity from the cloud applications users. Therefore, to maintain certain availability baseline, the cloud providers and users should maximize the applications HA using efficient HA-aware deployment models with an indispensable service delivery. With the proper availability and outsourcing solution, the Total Cost of Ownership (TCO) can be reduced while increasing the ROI of the cloud model.

Although modeling and simulation environments are broadly used in different ICT branches, some challenges arise when applying them to the cloud [70].

- All the cloud infrastructure entities should be taken into consideration during any simulation experiment.
- Multiple intricacies between the cloud provider and cloud applications should be considered to ensure the best HA-aware mapping between applications and corresponding hosts.
- The dynamic property of cloud market requires a prompt prototyping where simulations of new cloud approaches should be executed in a well-timed setting.
- The "always available" property of the cloud should be associated with a vertical and horizontal scaling of cloud applications components. This can add hiccups to existing deployment solutions. This requires an elasticity-aware technique to handle sudden changes in resources or applications architecture.
- With the modern reshaping of DCs architecture to handle IoT, data warehousing,

data analytics, data lakes, virtualization, cloud, and real-time computing, scalable solutions are required [71].

To this end, this chapter provides an abstract and generic simulation approach where different cloud nodes (DC, server, application components, VMs), load balancer, and HA features are well-defined and modeled.

6.4 ACE Design

Discrete event simulation (DES) provides a flexible way to evaluate multiple approaches designed for cloud systems, without the need to implement and assess them in a real-world environment. CloudSim is one of the well-known cloud simulators that is built on the top of a DES paradigm. It allows modeling and simulation of large-scale cloud scenarios. However, contemporary requirements of the cloud, such as HA, should be addressed as well in cloud-based simulators. HA is considered a hidden agenda behind the migration to the cloud, and consequently, it is an open challenge for many IT enterprises.

For this purpose, we propose ACE, an availability-aware CloudSim extension that simulates and evaluates cloud systems having an erroneous nature. The objective of ACE is to design and evaluate HA-aware mechanisms for the cloud. ACE contributions are summarized as follows:

- Define an architecture for HA-aware cloud (generic template for cloud model that captures HA features).
- Provide automated generation of requests while discovering the functional chains (computational path) and protection group (redundancy group) for cloud applications.
- Integrate HA-aware cloud allocation algorithm that places cloud applications while maximizing their HA and satisfying other SLA performance requirements.
- Design a load balancing algorithm at each tier of a cloud application.
- Provide a failure injection module and recovery/repair mechanisms to ensure self-healing upon failures of DCs, servers, VMs (representing cloud applications).
- Implement a modular and reusable HA-aware extension for CloudSim.
- Evaluate availability of different HA-aware deployments of cloud applications.

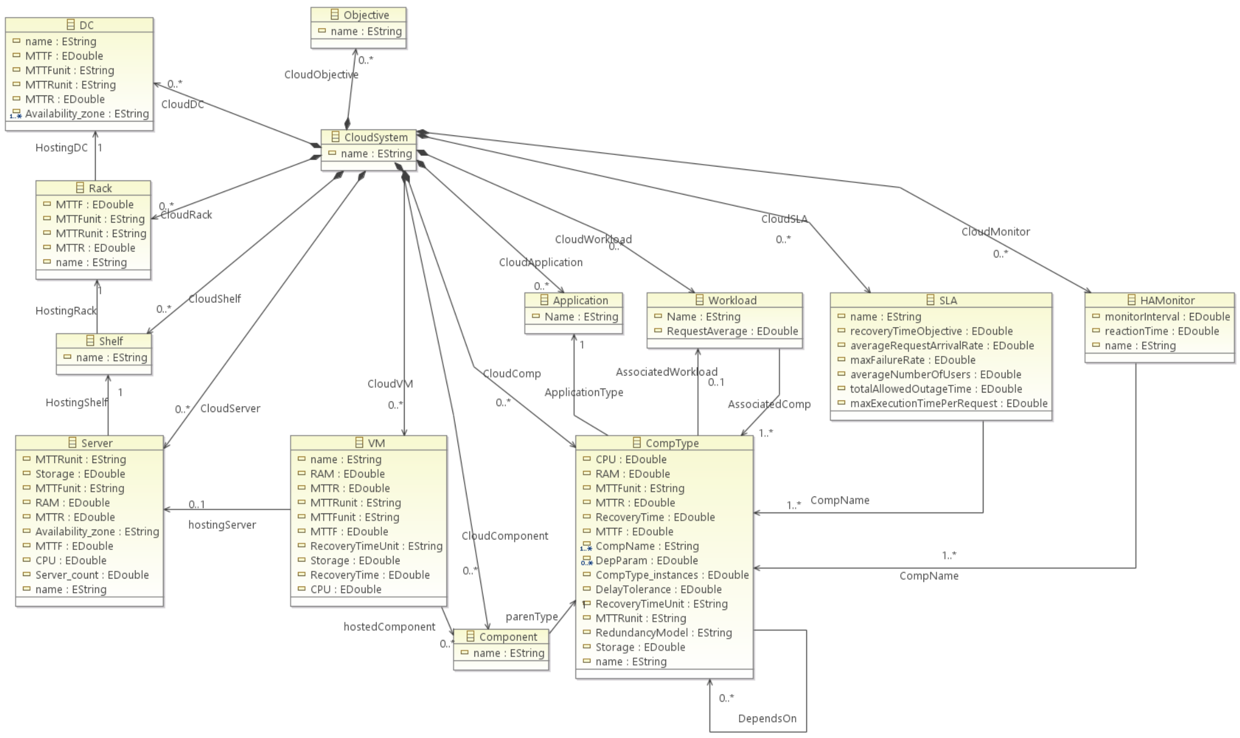


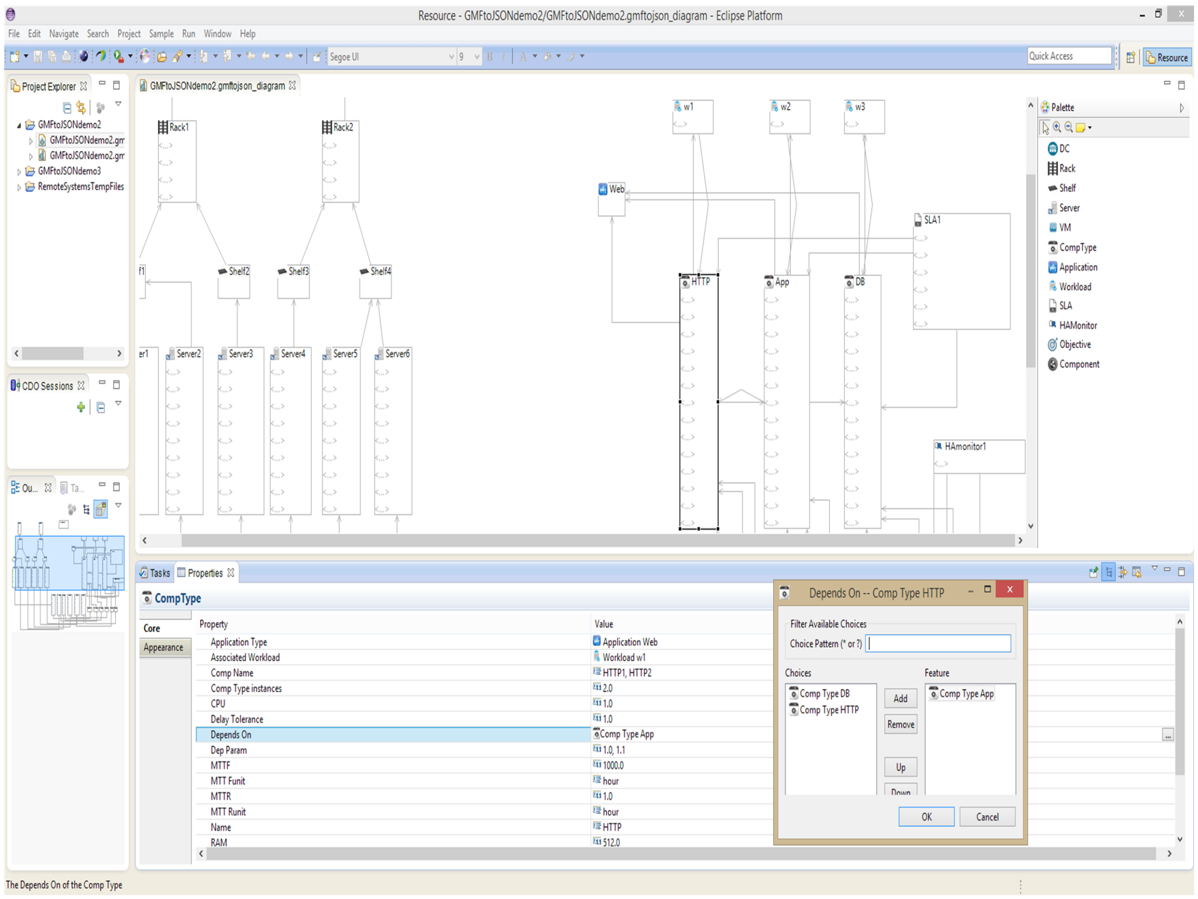
Figure 6.6: ACE model (Using Eclispe Ecore representation).

The source codes of ACE (extension for CloudSim) will be available in the GitHub repository. It is available now upon request. In the following, we describe the detailed design of ACE where different entities, features, and mechanisms are modeled in an abstract way to ensure modularity, reusability, extensibility, and scalability.

6.4.1 ACE modules

Input template module: CloudSim is extended with a user-friendly method, GITS (generic input template for cloud simulators), for generating a scenario, without exposing the cloud user to the details of development and coding examples in the simulator [72]. Manual creation of use cases in CloudSim can be a tedious and erroneous job. Therefore, GITS aims at providing an imperative way to generate any cloud scenario that ensures configurations reusability, repeatability, applications portability, and automated orchestration between different cloud providers while minimizing error, cost, and time-to-value.

GITS models the cloud provider, cloud user, and virtualization mapping between them through VM/containers. It captures different HA attributes associated with each entity of

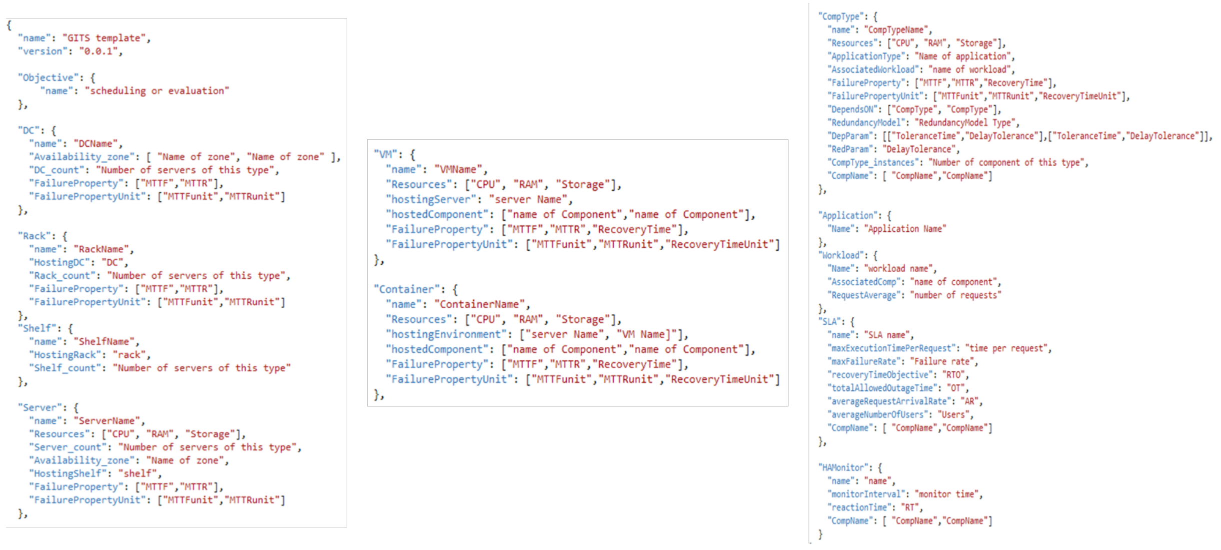


Property	Value
Application Type	Application Web
Associated Workload	Workload w1
Comp Name	HTTP1, HTTP2
Comp Type instances	2.0
CPU	1.0
Delay Tolerance	1.0
Depends On	Comp Type App
Dep Param	1.0, 1.1
MTTF	1000.0
MTT Funit	hour
MTTR	1.0
MTT Runit	hour
Name	HTTP
RAM	512.0

Figure 6.7: ACE graphical editor.

the cloud including HA statistical measures (MTTF, MTTR, and recovery time), redundancy model, failure types, and recovery mechanisms. GITS models the cloud as a cloud provider consisting of multiple DCs hosting multiple racks and servers and a cloud application consisting of multiple components of different types. Each type is associated with a failure type, redundancy model, SLA requirements, workload characteristics, and redundancy model. Different redundancy and dependency relations between component types are captured as well. Components can be modeled in an active-active redundancy model, active-standby (cold and hot) model, and active-spare model. Fig. 6.6 shows Ecore diagram for GITS cloud model.

To maintain modularity and easy-to-use features, GITS consists of a multi-layer input model. At the frontend layer, an Eclipse graphic modeling framework (GMF) project is



```
{
  "name": "GITS template",
  "version": "0.0.1",

  "Objective": {
      "name": "scheduling or evaluation"
  },

  "DC": {
    "name": "DCName",
    "Availability_zone": [ "Name of zone", "Name of zone" ],
    "DC_count": "Number of servers of this type",
    "FailureProperty": ["MTTF","MTTR"],
    "FailurePropertyUnit": ["MTTFunit","MTTRunit"]
  },

  "Rack": {
    "name": "RackName",
    "HostingDC": "DC",
    "Rack_count": "Number of servers of this type",
    "FailureProperty": ["MTTF","MTTR"],
    "FailurePropertyUnit": ["MTTFunit","MTTRunit"]
  },
  "Shelf": {
    "name": "ShelfName",
    "HostingRack": "rack",
    "Shelf_count": "Number of servers of this type"
  },

  "Server": {
    "name": "ServerName",
    "Resources": ["CPU", "RAM", "Storage"],
    "Server_count": "Number of servers of this type",
    "Availability_zone": "Name of zone",
    "HostingShelf": "shelf",
    "FailureProperty": ["MTTF","MTTR"],
    "FailurePropertyUnit": ["MTTFunit","MTTRunit"]
  },
```

```
"VM": {
  "name": "VMName",
  "Resources": ["CPU", "RAM", "Storage"],
  "hostingServer": "server Name",
  "hostedComponent": ["name of Component","name of Component"],
  "FailureProperty": ["MTTF","MTTR","RecoveryTime"],
  "FailurePropertyUnit": ["MTTFunit","MTTRunit","RecoveryTimeUnit"]
},

"Container": {
  "name": "ContainerName",
  "Resources": ["CPU", "RAM", "Storage"],
  "hostingEnvironment": ["server Name", "VM Name]"],
  "hostedComponent": ["name of Component","name of Component"],
  "FailureProperty": ["MTTF","MTTR","RecoveryTime"],
  "FailurePropertyUnit": ["MTTFunit","MTTRunit","RecoveryTimeUnit"]
},
```

```
"CompType": {
  "name": "CompTypeName",
  "Resources": ["CPU", "RAM", "Storage"],
  "ApplicationType": "Name of application",
  "AssociatedWorkload": "name of workload",
  "FailureProperty": ["MTTF","MTTR","RecoveryTime"],
  "FailurePropertyUnit": ["MTTFunit","MTTRunit","RecoveryTimeUnit"],
  "DependsON": ["CompType", "CompType"],
  "RedundancyModel": "RedundancyModel Type",
  "DepParam": [["ToleranceTime","DelayTolerance"],["ToleranceTime","DelayTolerance"]],
  "RedParam": "DelayTolerance",
  "CompType_instances": "Number of component of this type",
  "CompName": [ "CompName","CompName"]
},

"Application": {
  "Name": "Application Name"
},
"Workload": {
  "Name": "workload name",
  "AssociatedComp": "name of component",
  "RequestAverage": "number of requests"
},
"SLA": {
  "name": "SLA name",
  "maxExecutionTimePerRequest": "time per request",
  "maxFailureRate": "Failure rate",
  "recoveryTimeObjective": "RTO",
  "totalAllowedOutageTime": "OT",
  "averageRequestArrivalRate": "AR",
  "averageNumberOfUsers": "Users",
  "CompName": [ "CompName","CompName"]
},

"HAMonitor": {
  "name": "name",
  "monitorInterval": "monitor time",
  "reactionTime": "RT",
  "CompName": [ "CompName","CompName"]
}
```

Figure 6.8: ACE JSON template.

built to provide a user-friendly approach. An Extensible Markup Language (XML) file is generated from the GMF, which will be inputted to the mid-layer and parsed into a JavaScript Object Notation (JSON) template. JSON data format is used because it is a human readable and reusable approach, which is mapped to a Unified Modeling Language (UML) class diagram at the backend layer. GITS is generic in a sense that it can be easily modified to fit any cloud simulator. Fig. 6.7 and Fig. 6.8 show the GMF and JSON template.

It is necessary to note that the output of any simulation is saved as an excel sheet where requests information is included.

Computational path and request generation module: Each application consists of multiple components. Each component belongs to a certain type that can depend on and/or sponsor other types. For example, a web application consists of 3 types: HTTPS-based component type, App-based type, and DB-based type where HTTPS depends on the App that is sponsored by DB. This interdependency communication between applications components forms the computational path or functional chain. In other words, it is the route followed by a user request to be successfully executed. Note that a request refers to a cloudlet. CloudSim is extended to include this components chaining. Each computational path consists of the different levels (three levels in case of the web application). The first level

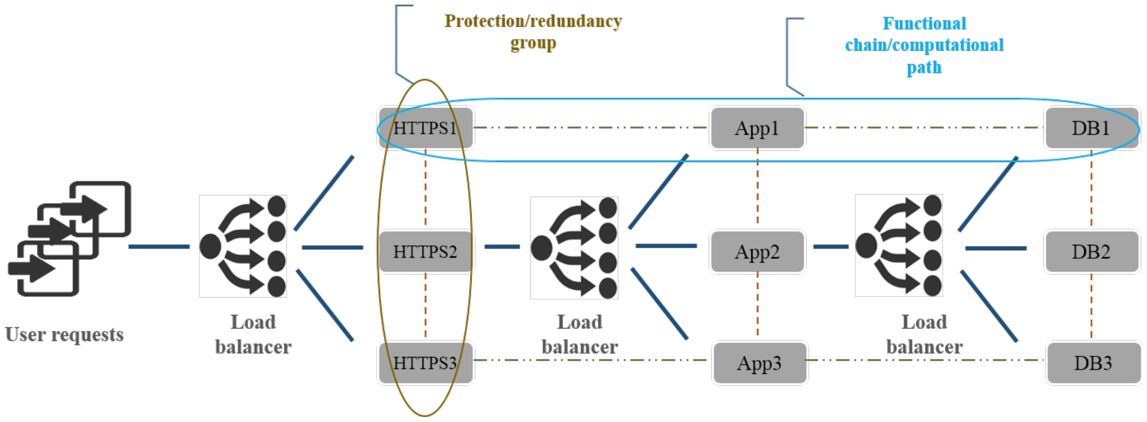


Figure 6.9: Example of three-tier web application.

represents the components types that do not have any dependents (HTTPS type in case of the web application). The requests arrive at the load balancer to be forwarded to the first level/tier of the chain. The first tier represents the primary component and its redundant ones. It is necessary to note that this redundancy relation forms a protection group (primary and redundant components). The requests are distributed on the active components of the first level. Once a request is processed, a sub-request is generated and forwarded again to the load balancer to be distributed on the active components of the second tier. The same process goes on until the request reaches the last tier. Fig. 6.9 shows the web application with computational path and protection group. A request is successfully processed if all the subrequests created at all the tiers of the path are successfully executed. CloudSim is also extended to include automatic generation of requests. Upon completion, a new request is automatically generated and forwarded to be distributed by the load balancer to the different tiers of the chain. It is necessary to note that user can either define a number of requests at the beginning of simulation or trigger the automated generation of requests while defining the simulation time.

HA-aware placement module: CloudSim provides space and time-based allocation policies, but it overlooks HA objective and constraints. ACE provides an HA-aware allocation policy for applications components. A simulator user can use either the default policy or the proposed HA-aware approach. The HA-aware approach is divided into sub-algorithms. Prior to the applications components placement, a criticality analysis is performed to differentiate between applications components priorities [66]. Once defined, the applications

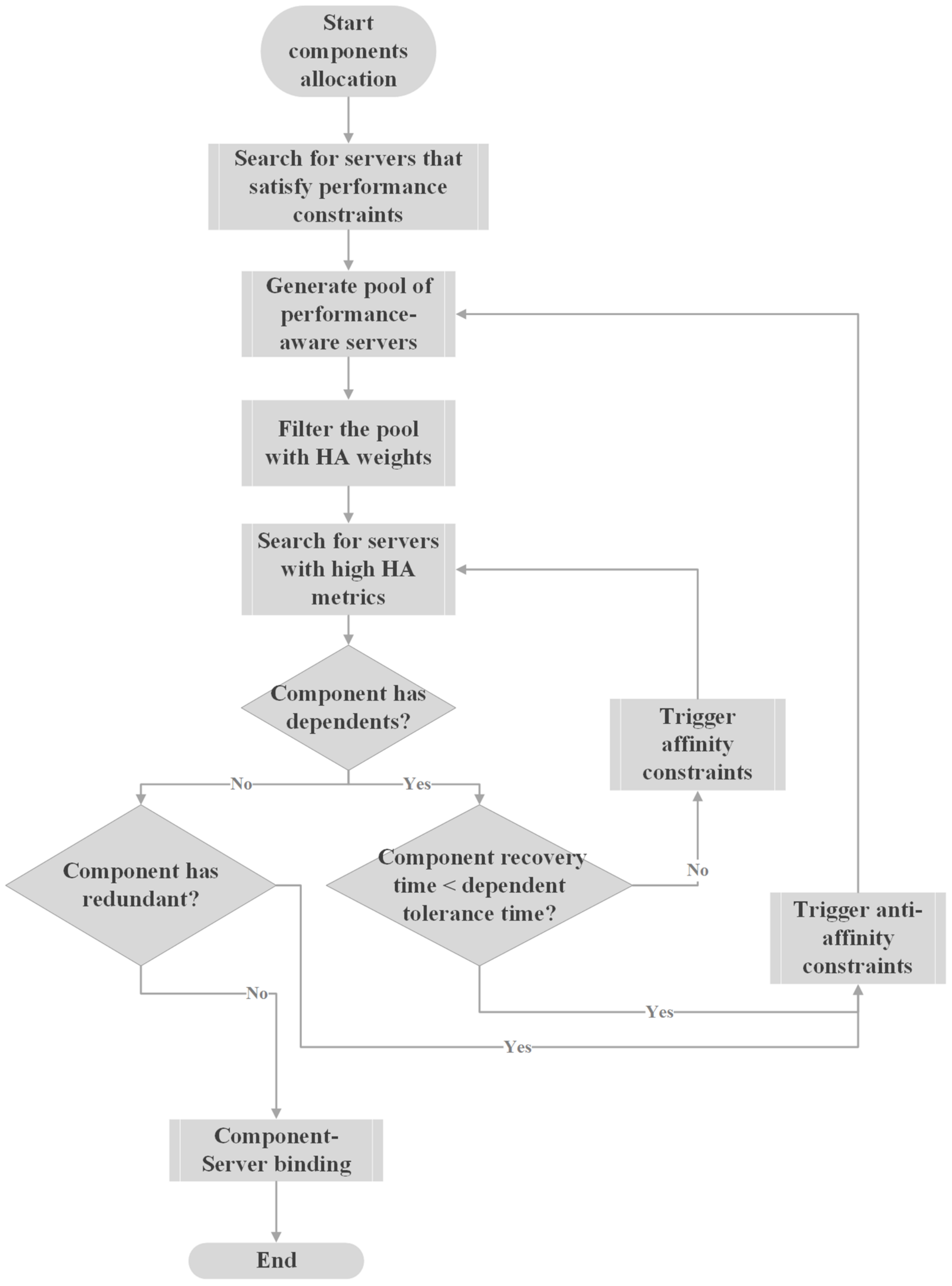

Start components allocation
Search for servers that satisfy performance constraints
Generate pool of performance-aware servers
Filter the pool with HA weights
Search for servers with high HA metrics
Component has dependents?
No
Yes
Component has redundant?
Component recovery time < dependent tolerance time?
No
Trigger affinity constraints
Trigger anti-affinity constraints
No
Yes
Yes
Component-Server binding
End

Figure 6.10: Flowchart of the HA-aware placement algorithm in ACE.

components are then inputted to the placement algorithm. The first step towards allocation is to find set of servers that can satisfy the performance demands of the applications components (computation resources and latency). For this purpose, a performance-aware

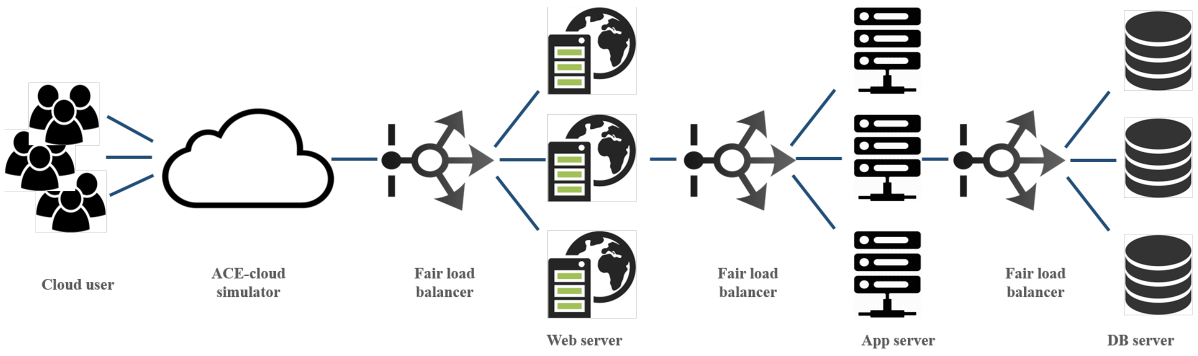


Figure 6.11: ACE load balancing module.

sub-algorithm is triggered to generate a pool of apt servers. The servers pool is imported to the availability algorithm to find the best server while maximizing the HA of the applications components. To that end, the availability sub-algorithm is executed to select a server from the pool with the highest availability measure (highest MTTF and lowest MTTR). However, the chosen server should satisfy the delay, affinity, and anti-affinity constraints. In other words, a component should be placed on a server that enables its communication with its redundant components. As for the affinity constraints, the availability algorithm restricts the placement of a component and its redundant ones on the same server (geo-redundancy policy). It also places the dependent components on their sponsor server if they cannot tolerate the sponsor failure. Otherwise, the algorithm provides different locations for the sponsor component and its dependents. Fig. 6.10 shows the flow chart of the placement algorithm.

The HA-aware allocation algorithm generates the mapping between applications components and their hosts (servers and DCs). It is necessary to note that the VMs represent the applications components where each VM has the same characteristics (resources and HA measures) as its component. Prior to simulation, the algorithm is executed, and the VM-host (component-host) is defined. To that end, the CloudSim broker is extended to include the VM-host binding at the beginning of the simulation. This extended broker class performs the binding of the VM to the required server in order to ensure that we can access the VM list of any server and the server of any VM, especially upon failure. Note that, in the extension, a VM refers to an application component.

Load balancing module: A load balancing algorithm is added to CloudSim. At each tier of the computational path, a load balancer is responsible for the distribution of the requests between available VMs. A fair load balancing algorithm is implemented to ensure a fair workload distribution among different entities. First, the fair load balancer searches for active components (VMs) to process a request. Then assigns the workload to the VM having the least waiting queue size (least number of requests in its queue). The load balancer does not only distribute the requests on the relevant VMs, but it is also responsible for the redistribution of requests upon failure of their corresponding VMs and/or hosts (servers and/or DCs). Fig. 6.11 shows the load balancing model of ACE.

Failure injection module: Each cloud entity (DCs, servers, and application components (VMs)) is associated with availability measures. With MTTF and MTTR, the availability ratio is calculated using (1). The failure time is then determined by multiplying this ratio with the simulation duration. Once determined, a failure is injected into the simulation. The faulty entity is considered "destroyed", and its corresponding requests are redistributed to the redundants.

Recovery and repair module: Once the failure is injected, the extended broker detects and isolates it to protect the rest of the cloud system. Simultaneously, it triggers the recovery and repair policies. If the failure happens at the level of the VM, the VM "recovery time" attribute determines when to trigger the recovery policy, and the MTTR determines when the repair policy is launched. The broker triggers "DestroyVM" method to generate a failure event and acknowledges the DC with this event. To that end, the "CloudSim class" in the core engine is extended to include dynamic future queue where its size can be updated anytime due to any unplanned event (failure, recovery, and repair) during the simulation. Since we can access the host id from a VM, the broker iterates over the "VM HashMap" of the host of faulty VM and changes the latter status to inactive, which is simultaneously updated in the VM list of the load balancer. The broker then determines the cloudlet queue of the faulty VM, which is extracted from the scheduling policy. The cloudlet in the execution queue is considered "failed". As for the cloudlets (requests) in the waiting queue, they are released and associated with a "failed" flag. Concurrently, the broker calls the load balancer to determine the available active redundant VM with the least cloudlet queue size. It is necessary to note that the broker is extended to act as the brain of the simulation and to ensure modularity and reusability of the code. For instance, any load balancer policy

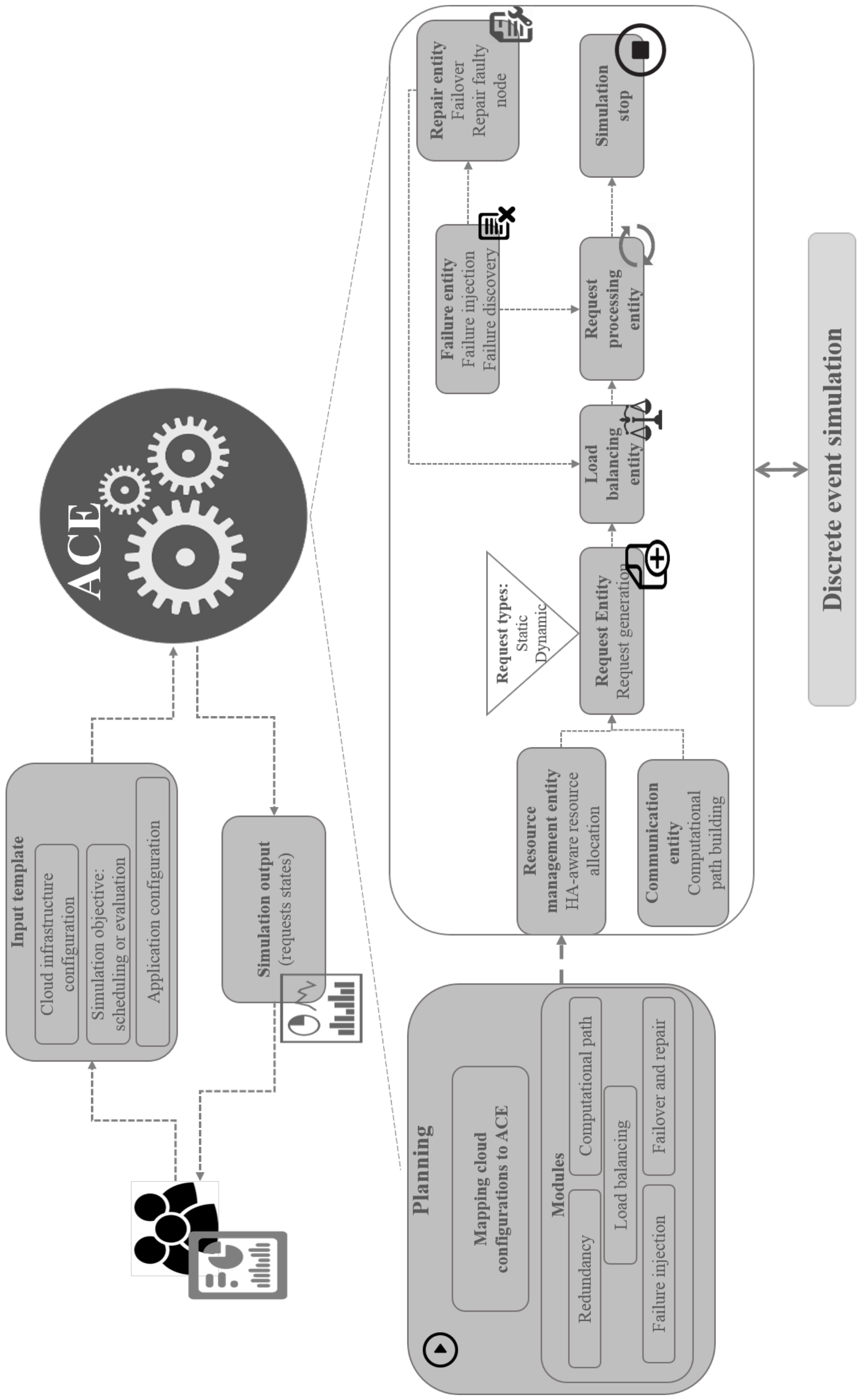


Figure 6.12: ACE architecture and different modules.

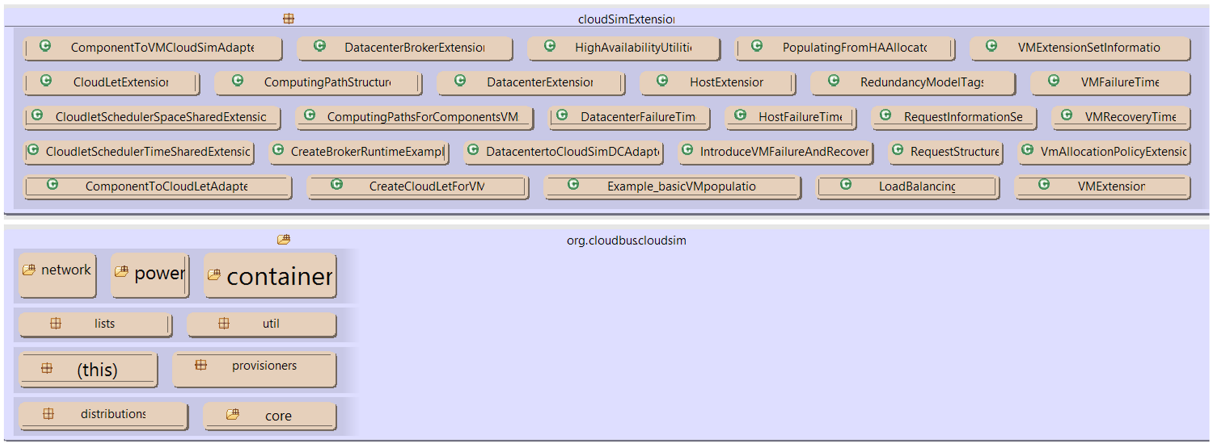


Figure 6.13: Different building blocks of ACE.

can replace the proposed one without affecting the simulation. Once the broker gets the apt redundant VM, it generates new cloudlets holding the same ids as the old ones and triggers their failover to the corresponding VM. After VM repair time, the faulty VM is active again and ready to process new requests. If the failure happens at the level of the host, the broker iterates over its “VM HashMap” and repeats the previous VMs recovery and repair policies. After MTTR of the faulty server, its status changes to normal, and it is ready to host new VMs. Similarly, if the failure happens at the DC level, its servers fail automatically, and the same applies to their hosted VMs. The VMs and its cloudlets are recovered/repaired as discussed above. The faulty DC and its servers are considered healthy again after their MTTR. It is necessary to note that DC and server failures are associated only with repair plans; we do not consider a hardware recovery policy (hardware redundancy), only redundancy (recovery) is assigned at the level of applications components.

6.4.2 ACE building blocks

This section explains the different classes used to extend CloudSim with ACE. Fig. 6.12 and Fig. 6.13 show ACE architecture and its main classes.

CloudletExtension: This class extends the Cloudlet class in CloudSim to reflect availability measures and computational path metrics (id, the status of completion, and dependents/sponsors).

CloudletScheduler classes: These classes include CloudletSpaceSchedulerSpaceSharedExtension and CloudletTimeSchedulerSpaceSharedExtension, which extend the classes of the scheduler in CloudSim. They are extended to release the resources of the faulty VM in order to ensure that it will have its full resources when it becomes healthy. Also, these classes determine the waiting and execution lists of VMs and include the failure injection and recovery mechanisms (failure/resume of cloudlets).

ComponentToCloudletAdapter: This class maps the components of the ACE input template (JSON template) to the extended cloudlet class (HA measures (MTTF, MTTR, tolerance time) and dependents/sponsors). At this point, it is assumed that each component can process one request at a time, but this class can be easily extended to include multiple requests processing concurrently.

ComponentToVMCloudSimAdapter: This class ensures that each application component is represented as a VM in the simulation. Consequently, the components of the ACE input template (JSON template) are mapped to the extended VM class (HA measures (MTTF, MTTR, tolerance time) and dependents/sponsors). Each VM is also associated with a unique id, to be used for example, as a reference when searching the VM list of its hosts.

ComputingPathsForComponentsVMs: This class is used to determine the dependent(s) and/or sponsor(s) of each component type and generate the computational path or the functional chain of the corresponding applications.

ComputingPathStructure: This class determines the structure of the computational path where the application component type of the first tier of the path, application components of the path, and a number of active components in each tier are defined.

CreateCloudletForVM: This class generates the cloudlets of each VM (a component of a specific type) where each cloudlet should have the characteristics of its corresponding VM (resources, delay, and HA metrics).

DatacenterBrokerExtension: This class is considered the brain of ACE simulation. It is extended to include failure injection of VM/server/DC, dynamic generation of cloudlets/VMs and computational path, dynamic destruction of VM/server/DC upon failure, recovery, and repair policies. The broker is also extended to support static requests, dynamic requests, and fluctuated workload generation. The main functionalities of ACE are implemented and triggered in the extended broker to ensure code modularity and reusability where different policies (load balancing and placement) can be added.

DatacenterExtension: This class is extended to capture HA measures (MTTF and MTTR) of the DC and include the acknowledgment for a VM failure and the binding between the VM and host according to the proposed HA-aware algorithm.

FailureTime classes: These classes include DatacenterFailureTime, HostFailureTime, and VMFailureTime. They determine the data structure of the failure time of the DC, server, and VM.

DatacenterToCloudSimDCAdapter: This class maps the DC of the ACE input template (JSON template) to the extended DC class in CloudSim.

HighAvailabilityUtilities: This class is used to calculate the time to inject the failure of DC, server, and VM based on the simulation duration.

HostExtension: This class is extended to include HA measures (MTTF and MTTR) of the server and its map of the hosted VMs.

IntroduceVMFailureAndRecovery: This class tracks the simulation time to determine the time to inject failures and trigger recovery. This class considers failure priority in a sense that if DC, server, and VM fail at the same time, it will trigger DC failure then host followed by VM. Also, if the MTTF is given same as MTTR of a VM, this class can handle this error. It will initially trigger a VM failure followed by repair. This feature can be used to redistribute requests of a certain VM to its redundants upon its overload.

LoadBalancing: This class is used to fairly distribute the requests to the active VMs at each tier of the computational path. It also redistributes the requests to the redundant VMs upon DC/server/VM failure.

PopulatingFromHAAllocator: This class is used to trigger the HA-aware allocation and get the placements of the applications components on the best servers while maximizing the components HA. These placements are used to perform the binding between the VMs and servers in ACE.

RedundancyModelTags: This class defines tags for redundancy types (active, standby, or spare) to be associated with each VM.

RequestInformationSet: This class generates HashMap of requests to facilitate the search.

RequestStructure: This class determines the structure of the request where it defines the request unique id, status, final request state, and the sub-cloudlets. The latter is generated at the different tiers of the chain, and it represents the main request.

VMAllocationPolicyExtension: This class is extended to include a method that releases the resources of the faulty VM from its host. This is triggered when a VM fails where the broker gets the host id of the VM and releases its resources to simulate real-time scenarios.
VMExtension: This class is extended to capture HA measure of a VM, its component type, broker, host id, and its cloudlet mapping.
VMExtensionSetInformation: This class generates a mapping between different ACE entities where HashMap is used instead to facilitate the search.
VMRecoveryTime: This class determines the data structure of the recovery time of a VM.
CloudSim: It is one of the classes of CloudSim core engine. This class is extended to include a dynamic update of the future queue. For instance, when the failure of an entity is injected during the simulation, a failure event is generated. This event should be added to the future queue of the DES, and consequently, the queue size should be updated accordingly.

6.5 ACE Evaluation

This section provides an evaluation of ACE to show the impact of availability metrics on the cloud performance. ACE is assessed on a three-tier web application. Amazon Web application can be an example [73]. The application consists of different types of interaction: the dependency relation between different types of the applications and the redundancy relation between applications components of the same type. Each application component id is mapped to a VM in ACE, and each VM is bound to a server based on the proposed HA-aware placement of ACE. The MTTF, MTTR, and recovery time are the metrics used to measure HA of deployed components (VMs), inject failures, and recover faulty nodes.
It is necessary to note that the downtime of an application component *C* is calculated in terms of outage hours per year, and its availability A_C is calculated as follows [65]:

$$A_C = \left(\frac{8760 - downtime_C}{8760} \right) \times 100 \qquad (2)$$

In this section, availability of each deployed VM (components) is measured in terms of outage hours per year, and the availability of a cloud scenario where its VMs are already deployed is measured in terms of a number of served of successfully processed requests. ACE can be used to test and evaluate different cloud-based objectives:

- Evaluate multiple availability and performance-aware allocation techniques.
- Assess the resiliency of cloud model under study in terms of different failures and recovery policies.
- Provide availability analysis of any cloud placement solution. The analysis does not only detect failures, their effects, and recovery/repair schemes, but it calculates the availability of a cloud model under various stochastic and deterministic events (failure, recovery, and overload). It is necessary to note that CloudSim with the new extension (ACE) can be used as dependability analysis tools, such as Petri Net models, Reliability Block Diagram (RBD), Functional Failure Analysis (FFA), and Markov analysis [58].
- Assess the capability of each application component to process user requests under different configurations.
- Evaluate the impact of redundancy models on the number of served requests, their response, and waiting time. Similarly, ACE can assess the impact of failure injection on the requests.
- Extract different HA-aware lessons to improve the cloud solution resiliency to failure in the future (anticipated elasticity to meet future needs or performance requirements).
- Model and evaluate different requests distribution where ACE can model fixed number requests, workload fluctuation (different distribution of workload to model real-case scenarios, such as peak and normal periods), and automated generation of requests while defining their arrival rate *AR*.

ACE is implemented in Eclipse on a Linux VM with 26GB of RAM and 6 vCPUs running Ubuntu12.04. For all scenarios, simulations are run multiple times to define a confidence level of 95% based on the t-Table [74].

6.5.1 ACE configuration

In order to generate cloud scenarios in ACE, the user should define the JSON template for generating the input model. JSON template can be either defined by a user or generated

HA measures	**Distribution**	**Distribution metrics (hours)**
MTTF	Exponential	μ=2500
Tolerance time	Exponential	μ=10
MTTR	Truncated Normal	μ=[0.05-3]; σ=[0.016-1]

Table 6.1: Different HA metrics distribution

Scenario metrics	**Server**	**VM**
CPU (cores)	16-32 cores	1-2 core(s)
RAM	25-35 GB	256-1024 MB

Table 6.2: Computing metrics

from the GMF project. Once the input is defined, the user should determine if the requests are fixed or dynamically generated during the simulation. For this purpose, the user can define the number of requests (arrival rate *AR*) arriving simultaneously at the active nodes. The user should also define the simulation duration *SD* to run certain scenario. Different failure and recovery times can be defined. Before starting the simulation, ACE builds the computational path of a given application, executes the HA-aware placement to generate VMs deployments, and introduces failures and repair times.

The results are evaluated on a network of 3 DCs, 6 racks, and 70 servers. The MTTF of the cloud nodes is generated using an exponential distribution, and MTTR time is generated using truncated normal distribution [75] [76]. Different HA metrics are available online [54] [77] and [78]. Table 6.1 and Table 6.2 show the different configuration metrics of the cloud scenario. As for the computing configurations, VMs can be configured in different instances (small, medium, large, or x-large) [79].

To capture the interdependency and redundancy relations between applications components (VMs), ACE is evaluated on a real-time 3-tier Web application. The redundancy model of this application is active/active where the number of active components is changed during the simulation to capture their impact.

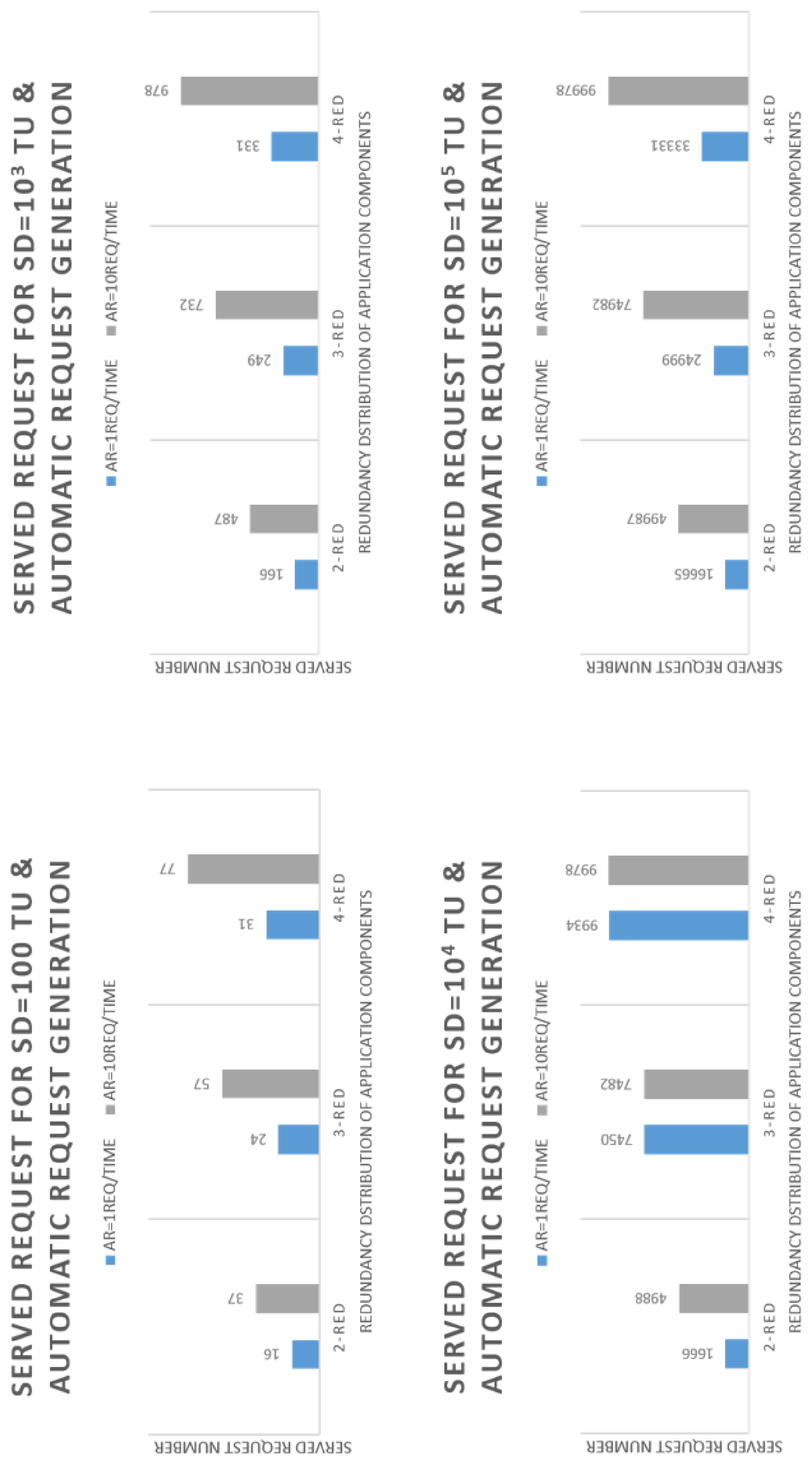


Figure 6.14: Impact of redundancy models on the number of the served requests for automatic request generation.

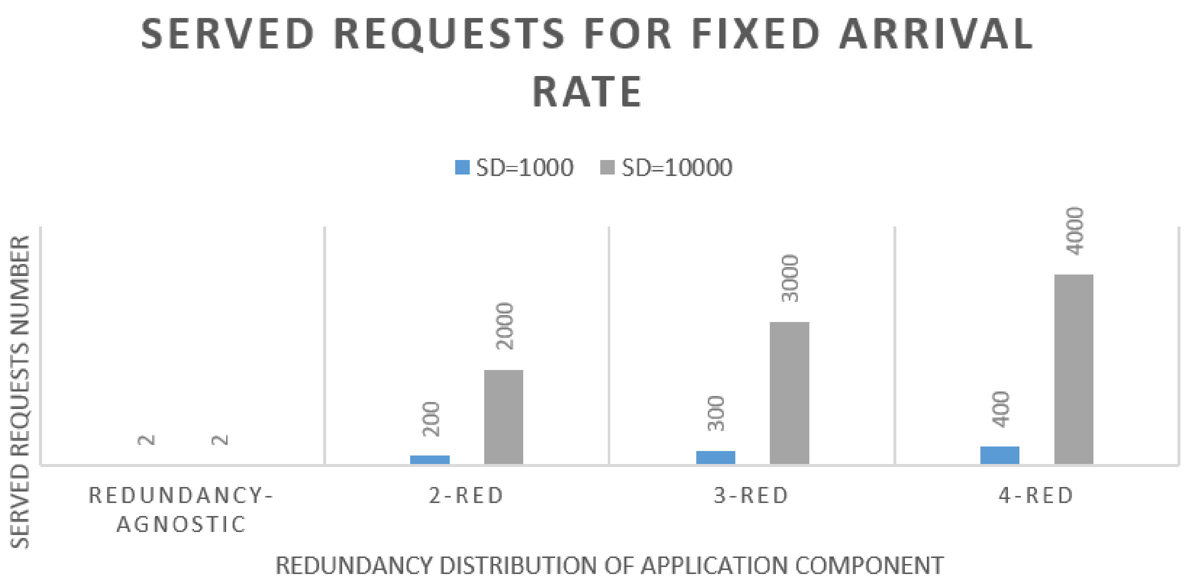


Figure 6.15: Number of served requests for static request generation and different redundancy models.

6.5.2 Results

In this section, ACE is evaluated to measure the following:

1. Redundancy impact on request states

The number of applications components per type is changed to measure the impact of redundancy model on the number of served requests, their response, and waiting times. Different redundancy models are defined where 2-RED represents 2 active components per type, 3-RED represents 3 active components per type, and 4-RED represents 4 active components per type. The request states are evaluated under different simulation durations *SD* and arrival rates *AR* where *SD*= x TU (x is simulation time measured in time unit (TU)) and *AR* = X req/time (X request arrives at each active node). Fig. 6.14 shows the impact of the redundancy model on the number of served requests for different *SD* and *AR*. It is noticeable that the number of served requests increases as the number of components increases. For example, the system can serve 37 requests for 2-RED while 4-RED allows the serving of 77 requests under same *SD*=100 TU and *AR*=10 req/time. Changing the simulation time allows serving more requests, which increase from 77 to 99,978 for 4-RED.

To measure the impact of redundancy model and recovery policies upon failure injection,

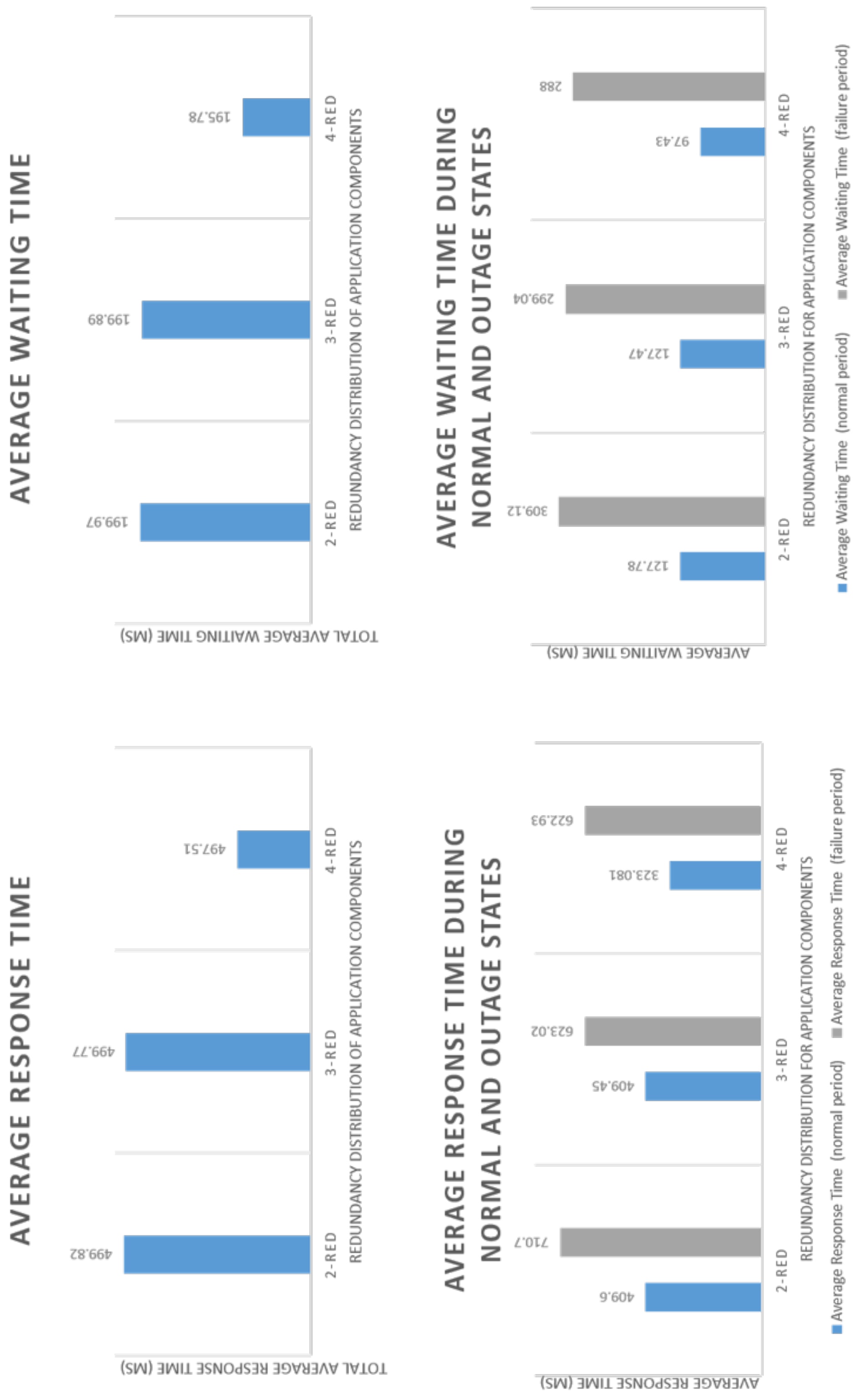


Figure 6.16: Impact of the redundancy models on the request's response and waiting times.

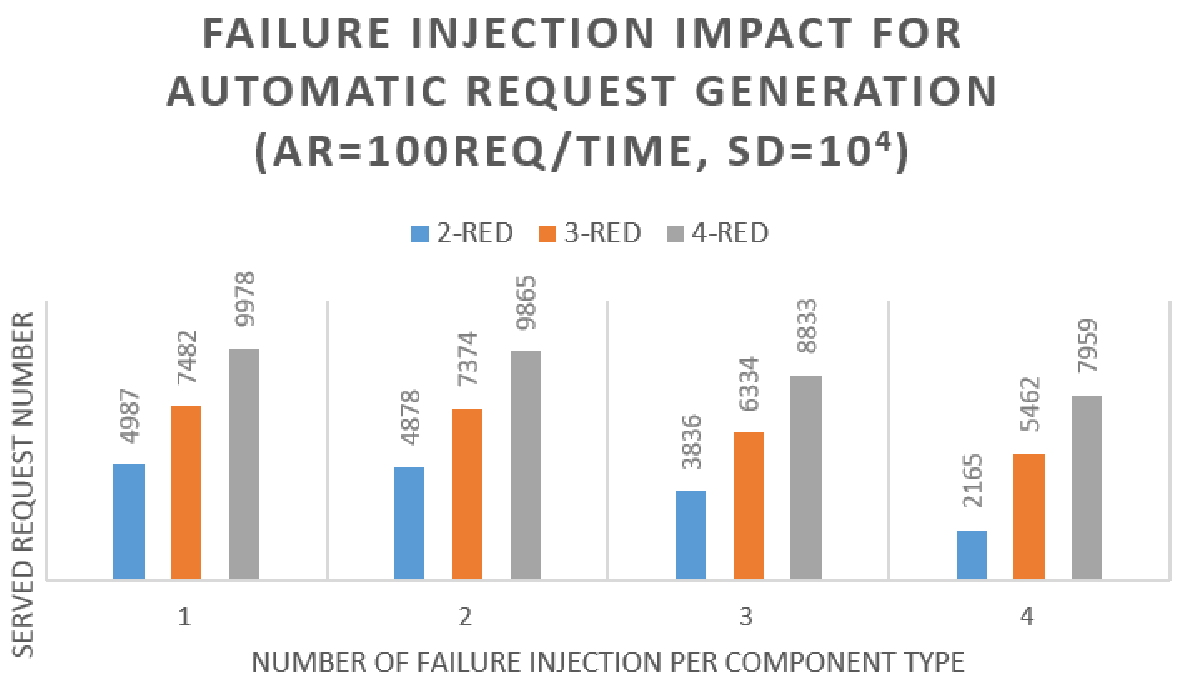


Figure 6.17: Impact of the number of failure injections on the request number for different redundancy models.

we define a fixed number of requests where *AR* = 100req/time for *SD* = 103 TU and *AR* = 100req/time for *SD* = 104 TU. The number of successfully completed requests is measured for redundancy-agnostic and redundancy-aware models as shown in Fig. 6.15. The failure is injected at the beginning of the simulation. If the components are configured with redundancy model and a recovery policy, all the requests are successfully completed where 3-RED for instance, can serve 3000 requests for *AR* = 100req/time for *SD* = 104 TU because it has 3 active components per type. As for the redundancy-agnostic model, it does not support any recovery solution and consequently, can only serve 2 requests regardless of *AR* and *SD*.

To measure the impact of the redundancy model on the request response and waiting times upon failure, we define *AR* = 10req/time for *SD* = 102 TU while requests are dynamically generated as long as a healthy active component(s) are available. In Fig. 6.16, the total average response and waiting times of the requests are measured. Although a failure is injected to the system, the response and waiting times do not exceed the allowed response and waiting times (500 and 200 milliseconds (ms) respectively) [80]. It is noticeable that the response and waiting times decrease with the increase in the number of components per type. In this case and upon a failure, the requests failover, and the load balancer executes the fair distribution algorithm to distribute the requests of a faulty node to its redundant with the least queue size. When the number of components per type increases, a wider request redistribution space is available, and consequently, its response and waiting times decrease. However, the average response and waiting times during the outage period are higher compared to the times during the normal period where the outage period represents the failure injection and failover states. Fig. 6.16 shows these measures. The average response and waiting times for the requests of the faulty node(s) represent those times during the outage states. Although these measures decrease as more components are added to a type, the response and waiting times of the requests during the outage period might violate the acceptable times (response of 500 ms and waiting of 200 (ms)).

2. Failure injection impact on request states

The number of failure injection per component type is changed to measure the impact of failures on the number of served requests, their response, and waiting times. The number of served requests is evaluated under SD = 104 TU and AR = 100req/time. Fig. 6.17 shows the impact of failure injection on the number of served requests for the above SD and AR. As more failures are injected per type, the number of served requests drops. For example, the requests served drop from 9978 to 7959 for 4-RED. Although a recovery solution is executed upon failure, the number of requests decreases in case of a faulty system because the requests response and waiting times increase.

To measure the impact of the failure injection per component type on the request response and waiting times, we define AR = 10req/time for S D= 102 TU for 2-RED while requests are dynamically generated as long as a healthy active component(s) are available. In Fig. 6.18, the total average response and waiting times of the requests are measured. It is noticeable that the total average response and waiting times increase with the increase in the number of failure injections per type. For example, the response and waiting times for one failure injection/type are 499.77 and 199.64 ms respectively, which increase to 499.82 and 199.97 ms respectively for 3 failures/type. Although the numbers are close to the acceptable ones, they do not violate them (response of 500 ms and waiting of 200 (ms)) [80].

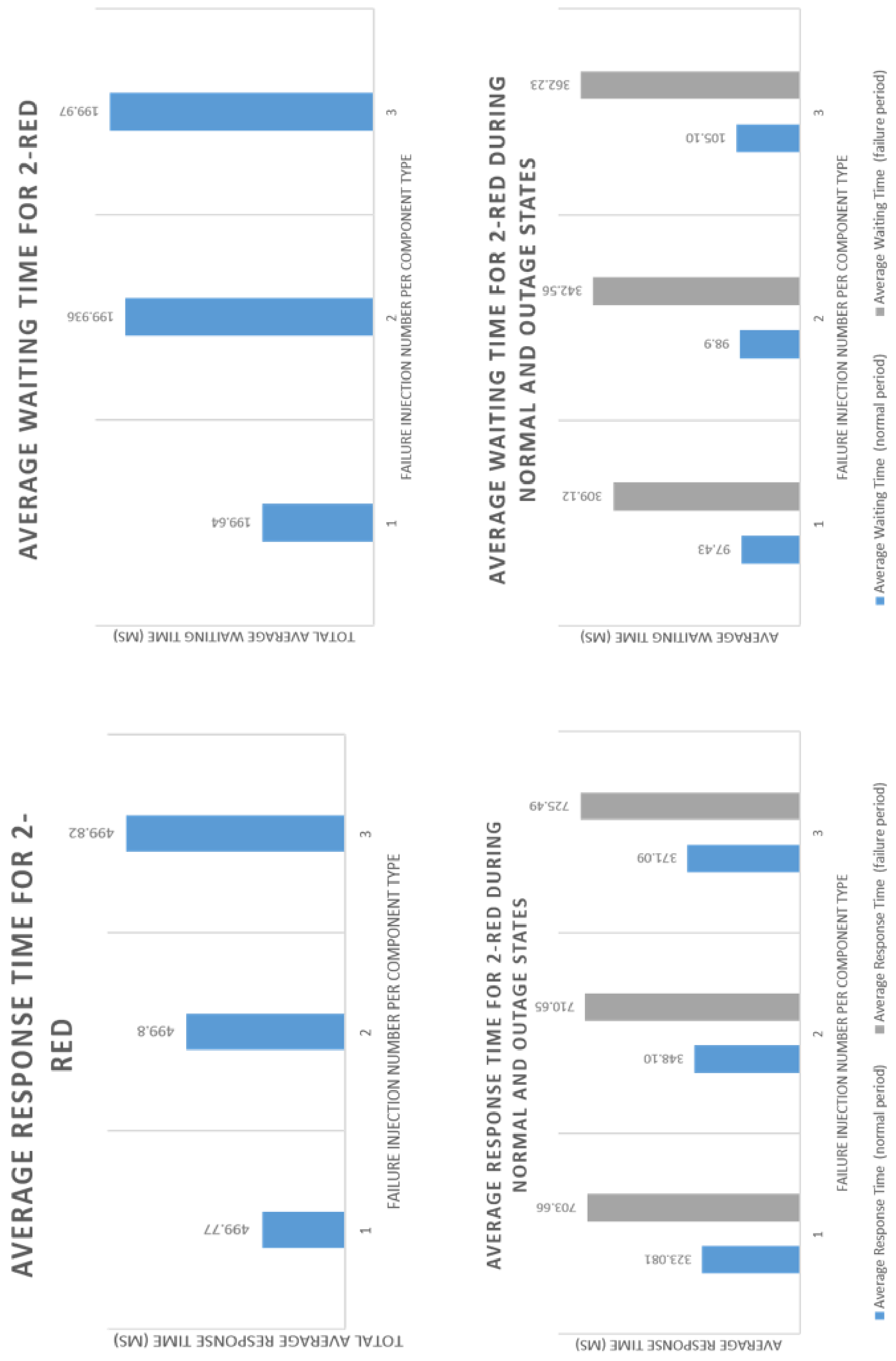


Figure 6.18: Impact of the number of failure injections on the request's response and waiting times.

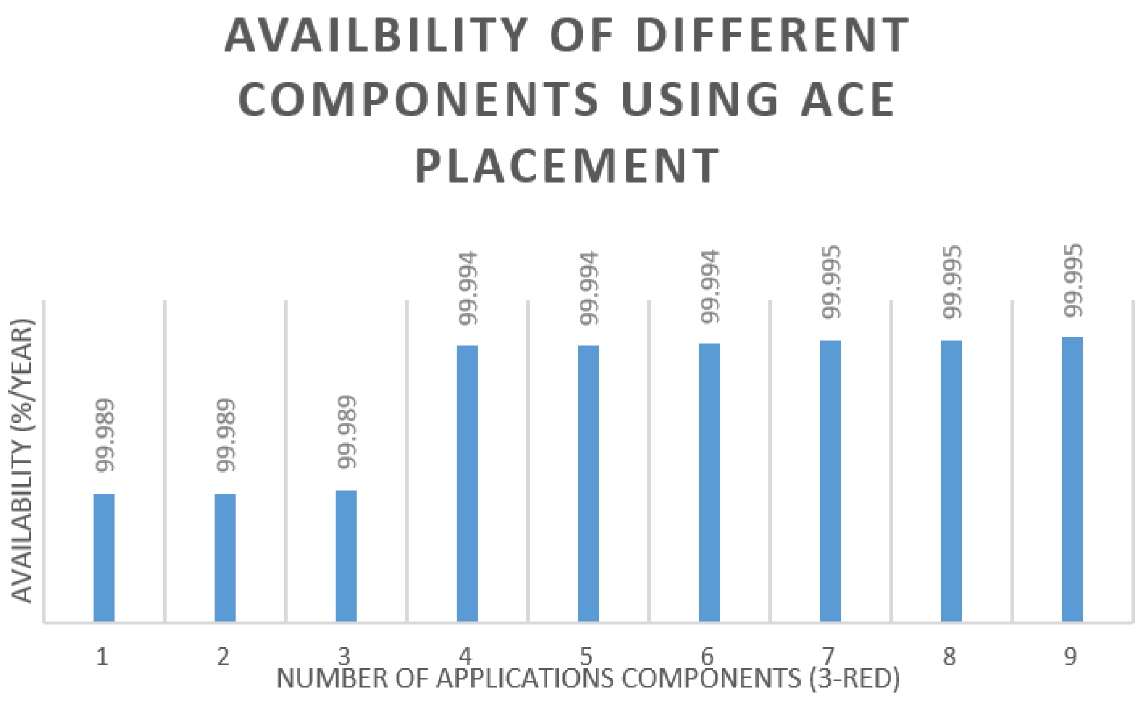


Figure 6.19: Availability of each deployed component.

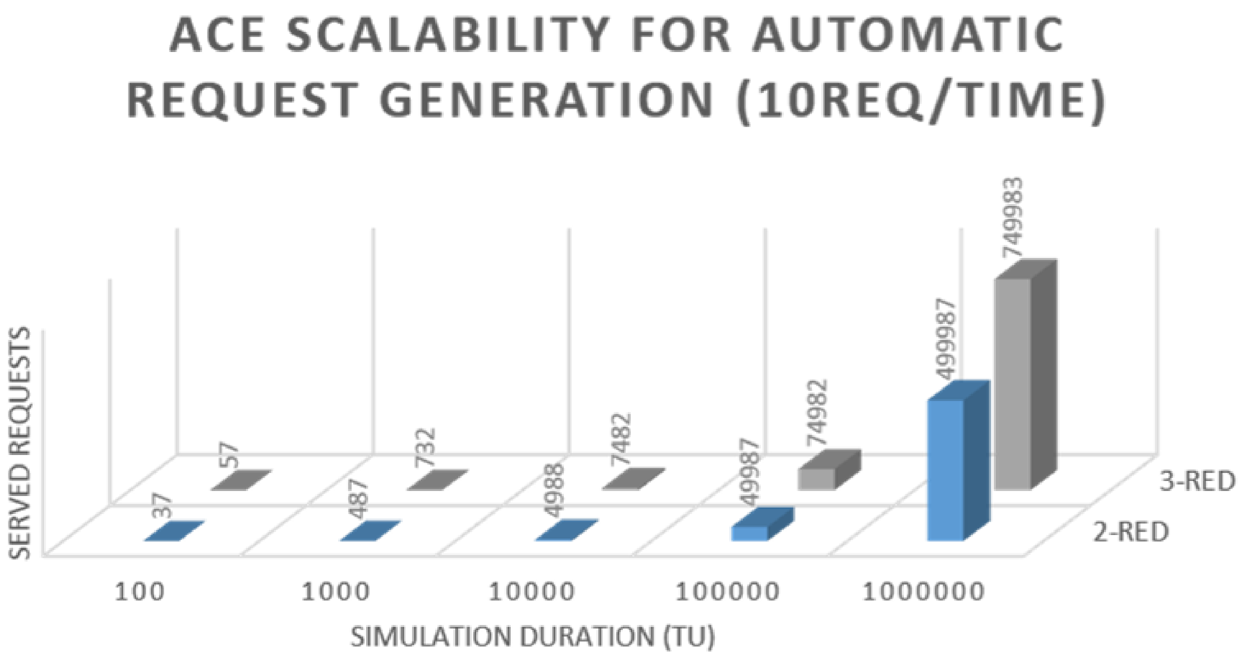


Figure 6.20: ACE scalability: Number of request processed using ACE for different redundancy models.

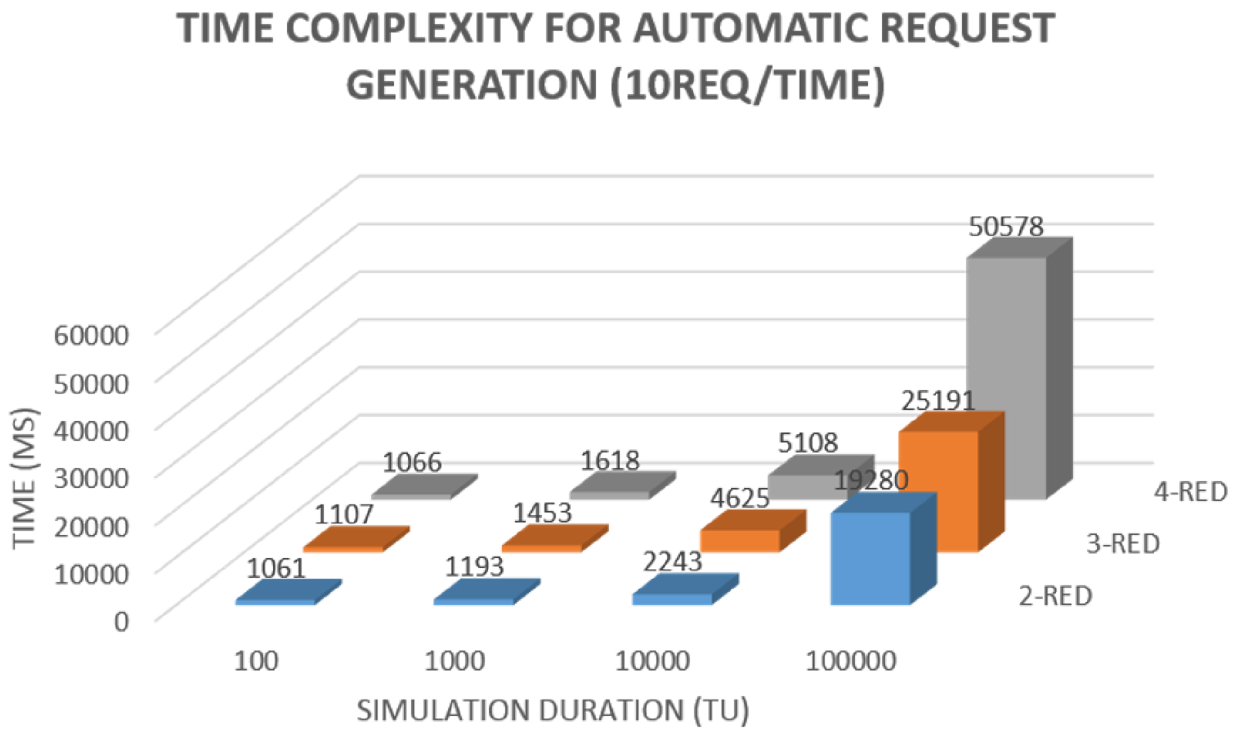
In this case and upon failures, the load balancer redistributes the requests using a fair distribution. When multiple failures are injected, the number of waiting requests for each node increases and consequently, causes an increase in the response and waiting times. Scaling up the system can be a solution as shown above. Although total average response and waiting times do not violate the SLA, the average response and waiting times during outage period are higher compared to the times during the normal period. Fig. 6.18 shows these measures. For the 2-RED scenario under study, the average response and waiting times of the requests of faulty nodes increase from 703.66 and 309.12 to 725.49 and 362.23 ms respectively and thus violate the SLA. As for the average response and waiting times during normal periods, they are within the acceptable range.

3. Availability of deployed components

The HA-aware placement algorithm is executed to place the components on the servers while maximizing their availability, which is measured using (1). Fig. 6.19 shows the availability of the components of the three-tier Web application where each tier consists of 3 components/type. The availability of different components ranges between three to four nines. The proposed algorithm prioritizes the component types to ensure that mission-critical applications are given the priority to be allocated. This results in the change of the availability nines where high-priority components are placed on the servers that guarantee the highest HA. Normal-priority components are allocated on the servers that ensure high HA, but it is not necessary the same as the critical ones. It is necessary also to take into consideration the difference between MTTF, MTTR, recovery time, and other HA metrics of different cloud nodes.

4. ACE scalability

Different simulations are performed to measure the number of requests ACE can process during different *SDs* for *AR* = 10req/time. These simulations are executed on 2-RED and 3-RED cases. Fig. 6.20 shows the scalability of ACE. For 2-RED case, ACE can process 37 requests for *SD* =100 TU to reach 499,987 requests for *SD* =106 TU. It is expected that this number increases for 3-RED as shown above. For 3-RED, ACE can process 57 requests for *SD* =100 TU to reach 749,983 requests for *SD* =106 TU. With these experiments, it is noticeable that ACE can model and simulate multiple real-time cloud scenarios where thousands of requests are processed.



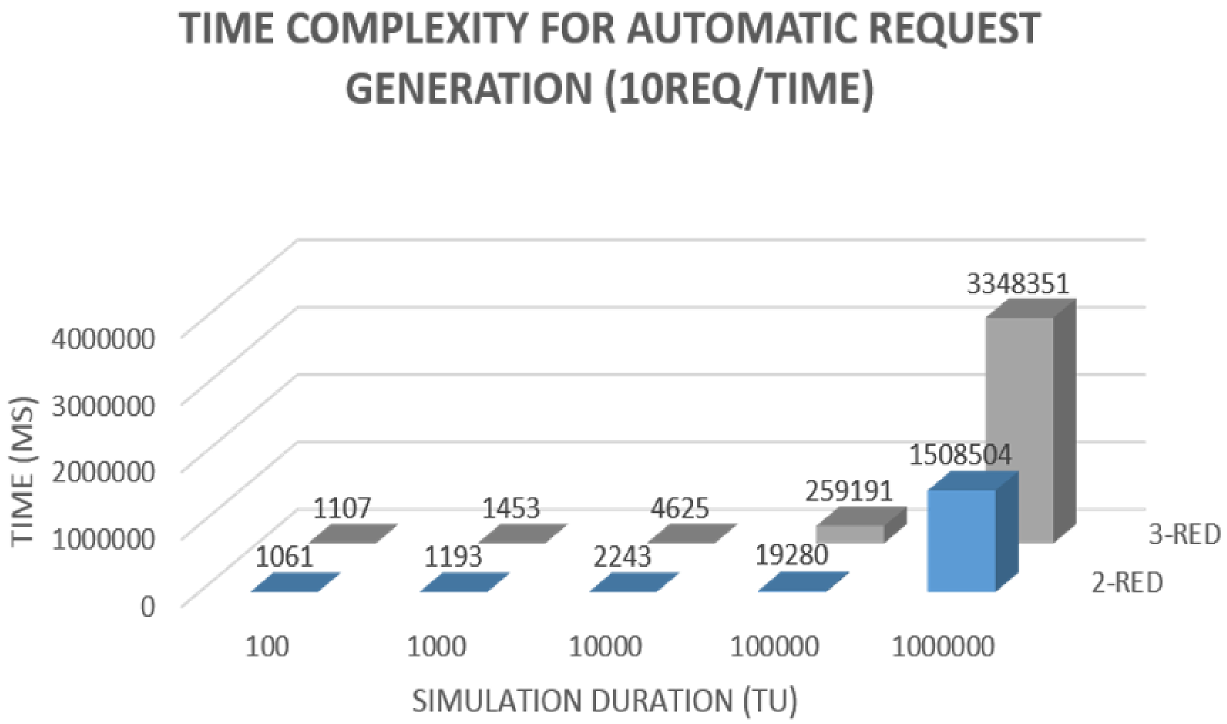


Figure 6.21: ACE time complexity for dynamic requests generation.

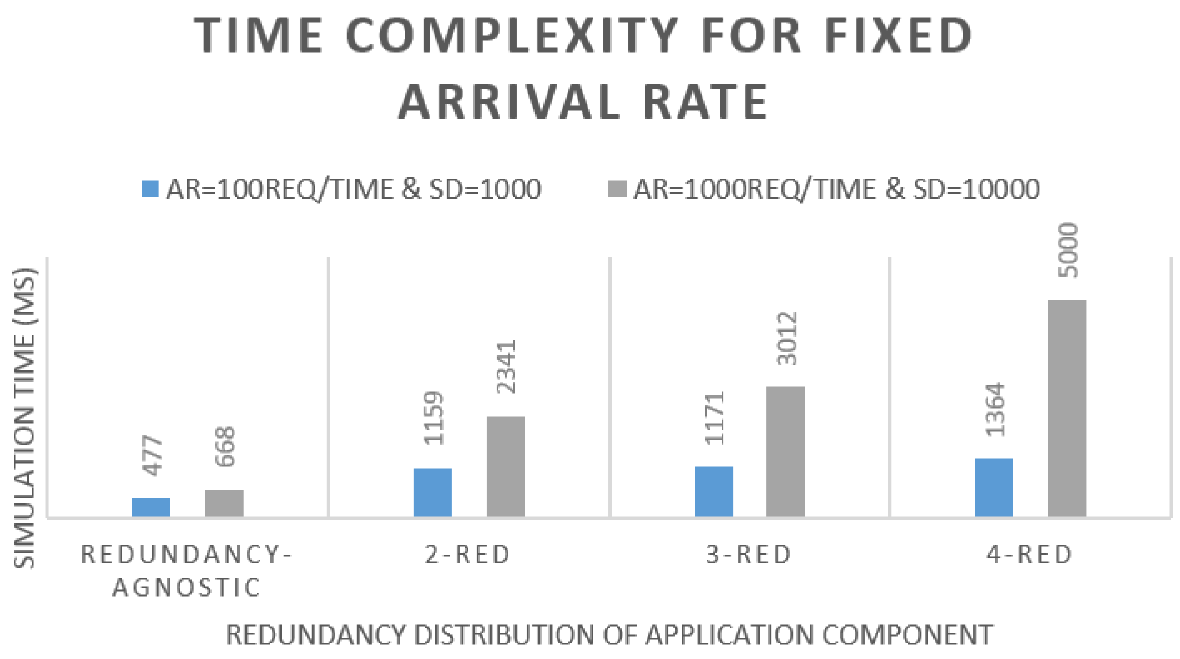


Figure 6.22: ACE time complexity for static requests generation.

5. Simulation time complexity

Although ACE can simulate a high number of requests, the time for simulating such cases on a limited resources environment can be a hiccup. Fig. 6.21 shows the time needed to finish simulations for different *SDs* for dynamic *AR* =10req/time. The simulation time increases with the increase in the *SD* and the number of components/type. For *SD* = [100-106 TU], the 2-RED time increases from 1,061 ms to 1,508,504 ms while the 3-RED simulation time increases from 1,107 ms to 3,348,351 ms. As for the 4-RED, the simulation time reaches 50,578 ms for *SD* = 105 TU. Fig. 6.22 shows the time needed to finish simulations for different *SDs* for fixed requests number. For fixed requests (*AR* = 1000) for *SD* = 104 TU, the simulation time increases from 668 ms for the redundancy-agnostic model to 5000 ms for the 4-RED case.

6.6 Conclusion

Providing a resilient cloud is imperative to underpin enterprises availability and performance requirements. Multiple stochastic failures can hinder the functionality of the cloud and impede the service delivery. It is of great importance to design an approach that does not only provide HA-aware placements of applications but also assess the cloud elasticity and provide the necessary HA-based lessons to improve the services availability. Simulation tools are one of the best ways to model the cloud and simulate it in terms of multiple QoS objectives. With this in mind, we extended CloudSim simulator with ACE to include HA properties in a sense that failures can be injected and recovered from. To that end, we proposed a JSON template, GITS, to generate cloud scenarios while keeping the development complexities behind the scene. GITS does not only model the cloud, but it captures different HA properties. ACE implements these properties in CloudSim. Once the simulation starts, ACE generates HA-aware placements of different cloud applications and builds the application functional chain to capture dependency and redundancy. It also injects failures, provides failover using redundancy models, and repairs the faulty nodes. Also, it provides a fair-based load balancing algorithm to distribute the dynamic and static requests.

References

[1] M. Jammal, H. Hawilo, A. Kanso, and A. Shami, "ACE: Availability-aware CloudSim Extension," *Submitted to IEEE Transactions on Cloud Computing*, 2017.

[2] TechRepublic, "The state of IaaS: Growing as cloud adoption continues," http://www.techrepublic.com/article/the-state-of-iaas-growing-as-cloud-adoption-continues/?ftag=TRE9ae7a1a&bhid=25335435715452818046449410599561, January 31, 2017. [February 15, 2017]

[3] Chargify, "The Future Is XaaS: What you need to know about Everything-as-a-Service," https://www.chargify.com/blog/xaas-everything-as-a-service/, February 7, 2017. [February 19, 2017]

[4] L. Wang, R. Ranjan, J. Chen, and B. Benatallah, "Cloud Computing, Methodology, Systems, and Applications," *CRC Press*, http://www.infosys.tuwien.ac.at/Staff/sd/papers/Buchbeitrag%20V.%20Emeakaroha.pdf, October 2011.

[5] WHIR Hosting Cloud, "GitLab's Not Alone: AWS, Google, and Other Clouds Can Lose Data, Too," http://www.thewhir.com/web-hosting-news/gitlabs-not-alone-aws-google-and-other-clouds-can-lose-data-too?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+thewhir+%28theWHIR.com+-+Daily+Web+Hosting+News%2C+Features%2C+Blogs+and+more%29, February 8, 2017. [February 15, 2017]

[6] ZDNet, "Dropbox sync glitch results in lost data for some subscribers," http://www.zdnet.com/article/dropbox-sync-glitch-results-in-lost-data-for-some-subscribers/, October 13, 2017. [January 20, 2017]

[7] InformationWeek, "Social Science Site Using Azure Loses Data," http://www.informationweek.com/cloud/cloud-storage/social-science-site-using-azure-loses-data/d/d-id/1252716, May 14, 2014. [December 9, 2016]

[8] ComputerWorld, “OOPS: Google ”loses” your cloud data (sky falling; film at 11),” `http://www.computerworld.com/article/2973600/cloud-computing/google-cloud-loses-data-belgium-itbwcw.html`, Auguest 20, 2015. [December 9, 2016]

[9] BusinessInsider, “Amazon’s Cloud Crash Disaster Permanently Destroyed Many Customers’ Data,” `http://www.businessinsider.com/amazon-lost-data-2011-4`, April 28, 2011. [January 11, 2017]

[10] Lynda Stadtmueller, “Which Cloud Storage Service Delivers the Performance You Need? Comparing IBM Cloud Object Storage and Amazon S3,” `https://essextec.com/wp-content/uploads/2016/08/Frost-and-Sullivan-Report-IBM-Cloud-Object-Storage-vs-Amazon-S3.pdf`, July 2016.

[11] H. Hawilo, A. Kanso, and A. Shami, “Towards an Elasticity Framework for Legacy Highly Available Applications in the Cloud,” *IEEE World Congress on Services (SERVICES)*, pp. 253-260, July 2015.

[12] 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, no. 6, pp. 18-26, December 2014.

[13] M. Toeroe and F. Tam, “Service Availability: Principles and Practice,” *John Wiley & Sons*, `http://ca.wiley.com/WileyCDA/WileyTitle/productCd-1119954088.html`, May 2012.

[14] A. Boteanu and C. Dobre, “A Simulation Model For Fault Tolerance Evaluation,” *U.P.B. Sci. Bull*, vol. 73, 2011.

[15] B. Wickremasinghe, R. N. Calheiros and R. Buyya, “CloudAnalyst: A CloudSim-Based Visual Modeller for Analysing Cloud Computing Environments and Applications,” *24th IEEE International Conference on Advanced Information Networking and Applications*, pp. 446-452, 2010,

[16] S. K. Garg and R. K. Buyya, “NetworkCloudSim: Modelling Parallel Applications in Cloud Simulations,” *Fourth IEEE International Conference on Utility and Cloud Computing*, 2011.

[17] D. Kliazovich, P. Bouvry, Y. Audzevich and S. U. Khan, “GreenCloud: A Packet-Level Simulator of Energy-Aware Cloud Computing Data Centers,” *IEEE Global Telecommunications Conference GLOBECOM*, pp. 1-5, 2010.

[18] S. K. S. Gupta, Rose Robin Gilbert, A. Banerjee, Z. Abbasi, T. Mukherjee and G. Varsamopoulos, “GDCSim: A tool for analyzing Green Data Center design and resource management techniques,” *International Green Computing Conference and Workshops*, pp. 1-8, 2011.

[19] A. Zhou, S. Wang, Q. Sun, H. Zou, and F. Yang, “FTCloudSim: A Simulation Tool for Cloud Service Reliability Enhancement Mechanisms,” *Middleware Posters and Demos Track*, December 2013.

[20] R. N. Calheiros, M. A.S. Netto, C. A.F. De Rose, and R. Buyya, “EMUSIM: An Integrated Emulation and Simulation Environment for Modeling, Evaluation, and Validation of Performance of Cloud Computing Applications,” `http://www.cloudbus.org/cloudsim/emusim/`, 2012.

[21] M. Tighe, G. Keller, M. Bauer and H. Lutfiyya, “DCSim: A data centre simulation tool for evaluating dynamic virtualized resource management,” *8th international conference on network and service management (cnsm) and workshop on systems virtualiztion management*, pp. 385-392, 2012.

[22] S. H. Lim, B. Sharma, G. Nam, E. K. Kim and C. R. Das, “MDCSim: A multi-tier data center simulation, platform,” *IEEE International Conference on Cluster Computing and Workshops*, pp. 1-9, 2009.

[23] S. Ostermann, K. Plankensteiner, R. Prodan, and T. Fahringer, “GroudSim: An Event-based Simulation Framework for Computational Grids and Clouds,” *Euro-Par Parallel Processing Workshops*, pp. 305-313, August 31, 2010.

[24] I. Sriram, “SPECI, a simulation tool exploring cloud-scale data centres,” *Proceedings of the 1st International Conference on Cloud Computing*, December 2009.

[25] F. Fittkau, S. Frey and W. Hasselbring, “CDOSim: Simulating cloud deployment options for software migration support,” *IEEE 6th International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pp. 37-46, 2012.

[26] Y. Jararweh, Z. Alshara, M. Jarrah, M. Kharbutli and M.N. Alsaleh, “TeachCloud: A cloud computing educational toolkit,” *International Journal of Cloud Computing*, 2013.

[27] P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor, “Virtual Machine Time Travel Using Continuous Data Protection and Checkpointing,” *Newsletter ACM SIGOPS Operating Systems Review*, pp. 127-134, 2008.

[28] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar, “Checkpointing virtual machines against transient errors,” *16th International On-Line Testing Symposium (IOLTS)*, pp. 97-102, July 2010.

[29] S. Malik and F. Huet, “Adaptive Fault Tolerance in Real Time Cloud Computing,” *IEEE World Congress on Services*, pp. 280-287, 2011.

[30] B. Cully , G. Lefebvre , D. Meyer , M. Feeley , N. Hutchinson, and A. Warfield, “Remus: high availability via asynchronous virtual machine replication,” *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pp. 161-174, 2008.

[31] J. Nakano, P. Montesinos, K. Gharachorloo and J. Torrellas, “ReViveI/O: efficient handling of I/O in highly-available rollback-recovery servers,” *Twelfth International Symposium on High-Performance Computer Architecture*, pp. 200-211, 2006.

[32] K. Qureshi, A. Rehman and P. Manuel, “Enhanced GridSim architecture with load balancing,” *The Journal of Supercomputing*, 2010, pp. 265-275.

[33] F. Yiqiu, W. Fei and G. Junwei, “A Task Scheduling Algorithm Based on Load Balancing in Cloud Computing,” *WISM*, vol. 6318, pp. 271-277, 2010.

[34] S. Sadhasivam, N. Nagaveni, R. Jayarani and R. V. Ram, “Design and Implementation of an efficient Two- level Scheduler for Cloud Computing Environment,” *International Conference on Advances in Recent Technologies in Communication and Computing*, vol. 148, pp. 884-886, 2009.

[35] S. C. Wang, K. Q. Yan, W. P. Liao and S. S. Wang, “Towards a Load Balancing in a three-level cloud computing network,” *3rd IEEE International Conference on Computer Science and Information Technology*, vol. 1, pp. 108-113, July 2010.

[36] M. Gahlawat and P. Sharma, "Analysis and Performance Assessment of CPU Scheduling Algorithm in CloudSim," *International Journal of Applied Information System*, July 2013.

[37] C. S. Pawar and R. B. Wagh, "Priority Based Dynamic Resource Allocation in Cloud Computing," *International Symposium on Cloud and Services Computing*, pp. 1-6, 2012.

[38] J. James and B. Verma, "Efficient VM Load Balancin Algorithim For A Cloud Computing Environment," *In Proceeding of International Journal on Computer Science and Engineering (IJCSE)*, September 2012.

[39] M. A. Tawfeek, A. El-Sisi, A. E. Keshk and F. A. Torkey, "Cloud task scheduling based on ant colony optimization," *8th International Conference on Computer Engineering & Systems*, pp. 64-69, 2013.

[40] H. Jin, X. Shi, W. Qinag, and D. Zou, "DRIC: Dependable Grid Computing Framework," *Transactions on Information and Systems*, 2006.

[41] A. Cox, K. Mohanram, and S. Rixner, "Dependable Unaffordable," *1st workshop on Architectural and system support for improving software dependability*, 2006.

[42] B. N. Chun and D. E. Culler, "User-Centric Performance Analysis of Market-Based Cluster Batch Schedulers," *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 30-30, 2002.

[43] S.K. Garg, R. Buyya, and H.J. Siegel, "Time and cost trade-off management for scheduling parallel applications on utility grids," *Future Generation Computer Systems*, pp. 1344-1355, 2009.

[44] A. Nagavaram et al., "A Cloud-based Dynamic Workflow for Mass Spectrometry Data Analysis," *IEEE Seventh International Conference on eScience*, pp. 47-54, 2011.

[45] S. Ostermann, R. Prodan and T. Fahringer, "Extending Grids with cloud resource management for scientific computing," *10th IEEE/ACM International Conference on Grid Computing*, pp. 42-49, 2009.

[46] S. Pandey, D. Karunamoorthy, and R. Buyya, "Workflow engine for clouds," *Cloud Computing: Principles and Paradigms*, `http://www.cloudbus.org/papers/CloudWorkflow-Chapter2011.pdf`, pp. 321-344, 2011.

[47] K. Jeffery and B. Neidecker-Lutz, "The Future Of Cloud Computing Opportunities For European Cloud Computing Beyond 2010," *Expert Group Report*, `http://cordis.europa.eu/fp7/ict/ssai/docs/executivesummary-forweb_en.pdf`, 2012.

[48] D. Newman, S. Kramer, and O. Blanchard, "Overview Of The Seven Core Technologies Driving Digital Transformation," `https://hosteddocs.ittoolbox.com/the-seven-core-technologies-driving-digital-transformation.pdf`, 2016.

[49] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, pp. 23-50, 2011.

[50] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya, "CloudSim: a novel framework for modeling and simulation of cloud computing infrastructure and services," *Technical Report of GRIDS Laboratory, The University of Melbourne, Australia*, 2009.

[51] R. Buyya, R. Ranjan, and R. N. Calheiros, "Modeling and simulation of scalable cloud computing environments and the CloudSim toolkit: challenges and opportunities," *The International Conference on High Performance Computing and Simulation*, pp. 1-11, 2009.

[52] The Cloud Computing and Distributed Systems (CLOUDS) Laboratory, "CloudSim: A Framework For Modeling And Simulation Of Cloud Computing Infrastructures And Services," `http://www.cloudbus.org/cloudsim/`, 2009. [December 4, 2016]

[53] W. Zhao, Y. Peng, F. Xie and Z. Dai, "Modeling and simulation of cloud computing: A review," *IEEE Asia Pacific Cloud Computing Congress*, pp. 20-24, 2012.

[54] C. Cerin et al., "Downtime statistics of current cloud solutions," *http://iwgcr.org/wp-content/uploads/2013/06/IWGCR-Paris.Ranking-003.2-en.pdf*, June 2013.

[55] P.T. Endo et al. "High availability in clouds: systematic review and research challenges," *Journal of Cloud Computing: Advances, Systems and Applications,* 2016.

[56] M. Jammal, A. Kanso, P. Heidari, and A. Shami, "Availability Analysis of Cloud Deployed Applications," *IEEE International Conference on Cloud Engineering (IC2E)*, April 2016.

[57] M. Jammal, A. Kanso, P. Heidari, and A. Shami, "A Formal Model for the Availability Analysis of Cloud Deployed Multi-Tiered Applications," *3rd IEEE International Symposium on Software Defined Systems*, April 2016.

[58] M. Jammal, H. Hawilo, A. Kanso, and A. Shami, “Mitigating the Risk of Cloud Services Downtime Using Live Migration and High Availability-Aware Placement,” *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, December 2016.

[59] K. Bilal et al. “Fault Tolerance in the Cloud,” Encyclopedia of Cloud Computing, `http://sameekhan.org/pub/B_K_2015_BC_MB.pdf`, May 2016.

[60] G.R. Kalanirnika and V.M. Sivagami, “Fault Tolerance in Cloud Using Reactive and Proactive Techniques,” *International Journal of Computer Science and Engineering Communications*, pp. 1159-1164, 2015.

[61] Y. Zhang, Z. Zheng and M. R. Lyu, “BFTCloud: A Byzantine Fault Tolerance Framework for Voluntary-Resource Cloud Computing,” *IEEE 4th International Conference on Cloud Computing*, pp. 444-451, 2011.

[62] S.Y. Ko, I. Hoque, B. Cho, and I. Gupta, “Making cloud intermediate data fault-tolerant,” *In Proceedings of the 1st ACM symposium on Cloud computing*, pp. 181-192, 2010.

[63] Q. Zheng, “Improving MapReduce fault tolerance in the cloud,” *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1-6, 2010.

[64] H. S. Pannu, J. Liu and S. Fu, “AAD: Adaptive Anomaly Detection System for Cloud Computing Infrastructures,” *IEEE 31st Symposium on Reliable Distributed Systems*, pp. 396-397, 2012.

[65] M. Jammal, A. Kanso, and A. Shami, “High Availability-Aware Optimization Digest for Applications Deployment in Cloud”, *IEEE International Conference on Communications (ICC)*, pp. 6822-6828, June 2015.

[66] M. Jammal, A. Kanso, and A. Shami, “CHASE: Component High Availability Scheduler in Cloud Computing Environment,” *IEEE International Conference on Cloud Computing (CLOUD)*, pp. 477-484, 2015.

[67] E. Shimpy and J. Sidhu, “Different Scheduling Algorithms In Different Cloud Environment,” *International Journal of Advanced Research in Computer and Communication Engineering*, September 2014.

[68] J. Weinberg, "Job Scheduling on Parallel Systems," `http://cseweb.ucsd.edu/~j1weinberg/papers/weinberg06researchExam.pdf`, 2005.

[69] A. E. Elsanhouri, M. A.Ahmed, and A. H. Abdullah, "Cloud Applications Versus Web Applications: A Differential Study," *The First International Conference on Communications, Computation, Networks and Technologies*, 2012.

[70] P. Altevogt, W. Denzel, T. Kiss, "Cloud Modeling and Simulations," `http://domino.research.ibm.com/library/cyberdig.nsf/papers/59BB74A8B2BA887B85257C3800462A09`, 2013.

[71] DBTA Trends and Applications "DBTA Best Practices, : Moving to a Modern Data Architecture," `http://www.dbta.com/DBTA-Downloads/WhitePapers/DBTA-Best-Practices-Moving-to-a-Modern-Data-Architecture-5806.aspx`, 2017. [February 1. 2017]

[72] M. Jammal, H. Hawilo, A. Kanso, and A. Shami, "GITS: Generic Input Template for CloudSim and Cloud Simulators," *Submitted to Elsevier Future Generation Computer Systems*, 2017.

[73] Amazon Web Services, "AWS Template Format," `https://s3-us-west-2.amazonaws.com/cloudformation-templates-us-west-2/AutoScalingMultiAZWithNotifications.template`, September 2010. [March 2016]

[74] T table, "t-table," `http://www.sjsu.edu/faculty/gerstman/StatPrimer/t-table.pdf`, 2007. [December 2016]

[75] Reliability HotWire, "Availability and the Different Ways to Calculate It," `http://www.weibull.com/hotwire/issue79/relbasics79.htm`, September 2007. [September 20, 2016]

[76] EventHelix, "System Reliability and Availability," `http://www.eventhelix.com/RealtimeMantra/FaultHandling/system_reliability_availability.htm#.WKz9jVUrKUk`, 2014. [September 20, 2016]

[77] The availability digest, "Comparing Clouds with CloudHarmony," `http://www.availabilitydigest.com/public_articles/1001/cloud_comparisons.pdf`, Januaray 2015. [September 20, 2016]

[78] CloudHarmony, "CloudSquare Service Status," `https://cloudharmony.com/status-1year`, 2017. [February 13, 2017]

[79] Microsoft, "How to: Change the Size of a Windows Azure Virtual Machine," `https://msdn.microsoft.com/en-us/library/dn168976(v=nav.70).aspx`, 2013. [September 20, 2016]

[80] InfoQ, "Real-time Data Processing in AWS Cloud," `https://www.infoq.com/articles/real-time-data-processing-in-aws-cloud`, November 2015. [January 20, 2017]

Chapter 7
Conclusion

Cloud computing and its service models, such as Platform and Software as a Services, have changed the way the computing resources are allocated to Information and Communications Technology enterprises and users. The cloud provides pay-as-go models, services, elasticity, provisioning, energy efficiency, and other features that enable economy of scale to different cloud providers and users. However, the growing dependency of users on social media, telecommunication services, mobile applications, banking amenities, and other cloud services that are expected to be available anywhere and anytime requires a plan that mitigates inevitable failures and ensures the always-on access to these services. Planned and unplanned outages can cause failure of many critical applications. However, the need to deliver increasing levels of availability continues to accelerate as enterprises re-engineer their solutions to gain competitive advantage. Most often, these new solutions rely on immediate access to critical business data. It is not always the case that a service failure would result in just an inconvenience, but some outages can cause loss of revenue, loss of productivity, damaged customer relationships, bad publicity, lawsuits, and, at the worst, loss of life. If a mission-critical application becomes unavailable, then the enterprise is placed in jeopardy. With the proliferation of on-demand cloud applications, the cost of downtime can quickly grow in enterprises that are dependent on their cloud solutions to provide services. Availability means money in today's global and competitive business environment, and many organizations need HA and continuous availability of their services. This emanates high availability concerns regarding the adoption of cloud. Cloud providers offer different availability zones with geo-redundancy to protect their infrastructure and consequently, their tenants against failures and natural disasters. Cloud tenants are encouraged to deploy their applications across multiple zones and use elastic load balancing to distribute the workload. Nevertheless, different zones may have different reliability levels depending on the hardware equipment, the geo-location, and the energy source powering the facility. Hence, the ability to design an HA solution is extremely important for both the cloud tenants and providers that are bound by a service level agreement. The objective of this thesis is to pave the way for cloud applications to adopt HA-aware solutions and mitigate the above challenges. This is achieved by investigating different HA-aware approaches, designing novel HA-oriented cloud scheduling techniques and availability assessment models, and implementing them in real cloud settings.

7.1 Thesis Summary

The summary of each chapter is described as follows.

7.1.1 Chapter 2 summary

This chapter provides an abstract model that separates the cloud provider and cloud users and ensures their integration through virtual mapping (virtual machine (VM) or container). It then demonstrates the impact that the placement strategy of cloud-based applications has on their high availability. It starts by defining a mixed integer linear programming (MILP) model as an optimal solution for scheduling applications' components in small-scale network. Then it follows a more pragmatic approach, where CHASE, component's HA-aware scheduler, is proposed. Using CHASE, the availability of applications components is attained while considering functionality requirements, applications' interdependencies, and redundancies. It also considers different failure scopes and conducts criticality analysis to give mission-critical components higher scheduling priorities compared to normal 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. This scheduling technique implements different patterns and approaches that deploy redundant models and failover solutions. These practices are achieved through geographically distributed redundant applications' deployments without violating the interdependency requirements of application components. This eliminates the single point of failure caused at the level of the VM, cluster, or cloud. Also, CHASE overcomes the challenges of maintaining HA-aware application's deployment and compromises between different functionality and failover constraints. 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 the OpenStack environment.

7.1.2 Chapter 3 summary

This chapter defines the different forms of stochastic failures that can happen at the level of the cloud infrastructure and cloud applications. It then proposes a formal stochastic model to quantify the expected availability offered by an application deployment. For this purpose, a Stochastic Colored Petri Net model (SCPN) is designed to capture the stochastic characteristics of the cloud and decode them into elements of an availability model. The model captures the elements of the cloud model (DCs, servers, VMs, and containers), their redundancy schemes, and the impact of cascading failures among them. It then models the functional chain between the multi-tiered application components to process requests. Then the proposed SCPN model quantifies the expected availability of a given deployment and provides HA-aware key points to any cloud scheduling solution. Additionally, the chapter integrates the SCPN model with a performance-aware scoring system to mitigate the other cloud challenges, such as energy, cost, and other performance concerns. The proposed cloud scoring system selects the optimal deployment when multiple eligible HA-aware solutions are assessed using the SCPN model. The scoring tool is extensible and consequently, can handle different selections criteria, such as security, portability, and other norms.

7.1.3 Chapter 4 summary

This chapter proposes a live migration approach to maintain service delivery upon a sudden failure, a VM/infrastructure overload, or maintenance. First, it defines different design considerations to achieve HA-aware applications placement. The latter considers VMs/applications deployments in geographically distributed data centers and supports applications interdependency and other HA and performance requirements. Then the deployments are assessed using a formal Petri Net model to improve them in terms of HA. The proposed placement is then used in the migration approach to find new hosts for the

VMs. The chapter also develops an optimization MILP model that minimizes the migration downtime based on the VM memory pages and its HA-aware placement.

7.1.4 Chapter 5 summary

This chapter addresses the issues regarding the orchestration of cloud applications among multiple cloud providers. It then proposes GITS, a generic input template for CloudSim and other cloud simulators. GITS models cloud provider and user while ensuring their separation. This partitioning enables the migration of application between different providers to satisfy the quality of service requirements. The proposed template focuses on modeling the cloud not only in terms of computational resources but also in terms high availability (HA) properties associated with the cloud infrastructure and applications. This includes redundancy models, failure types/rates, recovery policies, and other HA-related metrics. GITS provides a graphical modeling framework (GMF) interface for visualizing the cloud model. The latter is then translated into a JavaScript Object Notation (JSON) schema to ensure cloud scenarios simplicity, readability, and reusability. The JSON format is then mapped to the CloudSim input model.

7.1.5 Chapter 6 summary

This chapter provides ACE, availability-aware CloudSim extension. ACE provides a graphical modeling project and a JavaScript Object Notation (JSON) template to ensure simplicity, repeatability, and reusability of cloud scenarios. ACE also extends CloudSim to include static and dynamic generation of requests, failure injection, redundancy and interdependency interactions (computational path), failover solutions, repair policies, load balancing, and HA-aware deployments. ACE is evaluated on a three-tier web application to measure the impact of availability features on applications deployments, request number, response, and waiting times.

7.2 Thesis Future Work

Although the proposed approaches are proven to maintain HA, there are still some open cloud challenges that can be integrated with them. As a future work, this thesis can be extended as follows.

7.2.1 Elasticity and storage mechanisms

In order to fully realize HA in the cloud, efficient resources provisioning, monitoring, and elasticity mechanisms should be investigated. Different elasticity techniques can be defined in the cloud architecture. This includes defining different horizontal and vertical auto scaling policies, which can be based on gathering different runtime information, analyzing them, defining thresholds (i.e., overload, power consumption, cyber-attacks), and reacting to any objectives violations. In order to support an elastic-aware framework, it is necessary to associate it with efficient resource provisioning and monitoring mechanisms. Machine learning models can be used to ensure an automated self-healing system that detects any sudden change and handles it seamlessly. Additionally, redundancy models study can be associated with the elastic techniques. This study can provide an intelligent selection and update on the type of redundancy model to be associated with the new components whether a scale-up or scale-down policy is triggered.
To implement and evaluate elasticity mechanisms with real cloud scenarios, Chapter 6 can be extended to include elasticity features where scaling up and down will be implemented to support dynamic scaling of the interconnected application's components, overcome failures, meet performance requirements, and assess the elasticity mechanism. Additionally, Chapter 5 can be extended to communicate with an HA management framework to automate the creation HA properties and update them on the fly. Besides, it can be also extended to include data visualization module that represents simulation results according to predefined policies.

Storage is another aspect that affects the cloud HA. It is necessary to ensure that the cloud-based applications are associated with a fault-tolerant-aware storage system. For instance, the storage is sometimes directly associated with the VM. In this case and upon failure of the VM, the temporary storage will go down and cannot be recovered. For this purpose, an intelligent architecture should be defined to differentiate between applications states and determine how these states can be persisted and how the data are extracted and dispatched to a “non-volatile” storage upon failure of a VM instance.

7.2.2 Multi-objective cloud management system

There are still some concerns regarding the security, interoperability, and energy of the cloud. These issues add other elements of challenges to the cloud HA. It is necessary to consider such issues during the design of a HA plan where trade-offs can be made to attain the level of availability and other concerns needed for a specific application. For example, in order to integrate the HA objective with energy efficient models, different metrics that reflect the DC efficiency, DC productivity, and DC performance per energy should be explored. The energy efficiency model can also be integrated with a prediction model that can determine the energy consumption in each server based on a statistical workload distribution and resource utilization. For this purpose, the proposed approaches of Chapter 2 and the Chapter 4 can be extended to support a multi-objectives model (maximize HA and resource utilization while minimizing carbon footprint). Also, different conditions can be added to Chapter 3 where the SCPN model will not only assess applications deployment upon failures but will evaluate other events (i.e. migration between cloud providers and exceeding power thresholds) impact on the cloud system performance as well.
Chapter 5 and Chapter 6 can be extended to support new performance policies. In Chapter 5, GITS can update the template and its parsers with the new metrics, such as power and security measures. As for Chapter 6, ACE can include new allocation policies and events that capture certain performance incidences (i.e. overload).

7.2.3 Container Management Framework

Containers have reshaped the information technology (IT) world through offering a new lightweight virtualization concept. They offer a new grade in deploying the workload and migrating their placements due to overlaid, maintenance, or other norms. They have smaller in size compared to VMs, which facilitates their migration between different cloud models. However, container management is one of the main challenges facing the container adoption. It is necessary to provide a framework that facilitates building, managing, availability, and scalability of the containers.

Besides, multiple challenges are still hindering the functionality of the cloud and should be addressed. They can be summarized as follows:

- The specifications of the cloud abstractions changes with their DCs ontology and the hierarchy of their resources. So when trying to move to a different cloud, there will be an incompatibility between the used deployment model and the particulars of the new IaaS model.
- Stateless cloud applications may not support enterprise requirements.
- Legacy architecture and absence of a standard application program interfaces (APIs) to support automation policies are another cloud concerns. The absence of cloud-aware orchestration solutions impedes the deployment of the cloud application in other vendors due to the inconsistency between APIs, resources, and abstraction levels of DC.

Curriculum Vitae

Name	Manar Jammal
Post-secondary	2013-2017 Ph.D.
Education and	Software Engineering
	Western University
	London, Ontario, Canada
	2011-2012 M.E.Sc.
	Electrical and Computer Engineering
	Ecole Doctorale des Sciences et de la Technologie (EDST)
	Lebanese University & University of Technology of Compiegne
	Beirut, Lebanon & Compiegne, France
	2006-2011 B.Eng.
	Electrical and Electronic Engineering
	Lebanese University
	Beirut, Lebanon
Related Work	2013-2016
Experience	Teaching Assistance
	Western University
	London, Ontario, Canada
	2013-2017
	Research Assistance
	Western University
	London, Ontario, Canada
	2013-2015
	Research Assistance
	Ericsson Research
	Montreal, Canada
	2011-2012
	Electronic Engineer

Methode Electronics
Lebanon

Honours and Awards

Western University Graduate Research Scholarship 2013-2017
Dean's Honor List 2009-2011

Publications

[P1] A. Kanso, P. Heidari, and M. Jammal,
High availability multi-component cloud application placement
using stochastic availability models,
P48033US1, November 2015.

[P2] A. Kanso, M. Jammal, and A. Shami,
Component High Availability Scheduler,
P44248 US1, October 2014.

[J1] M. Jammal, H.Hawilo, A. Kanso, and A. Shami,
"GITS: Generic Input Template for CloudSim and Cloud Simulators,"
Submitted to Elsevier Future Generation Computer Systems,
2017.

[J2] M. Jammal, H.Hawilo, A. Kanso, and A. Shami,
"ACE: Availability-aware CloudSim Extension,"
Submitted to IEEE Transactions on Cloud Computing,
2017.

[J3] H.Hawilo, M. Jammal, and A. Shami,
"Exploring Microservices as the Architecture of Choice
for Network Function Virtualization Platforms,"
Submitted to IEEE Communications Magazine,
2017.

[J4] M. Jammal, A. Kanso, P. Heidari, and A. Shami,
"Scrutinize High Availability-aware Deployments Using
Stochastic Petri Net Model and Cloud Scoring Selection Tool,"
Submitted to IEEE Transactions on Services Computing,
November 2016.

[J5] M. Jammal, T. Singh, A. Shami, R. Assal, and Y. Li,
"Software Defined Networking: State of the Art
and Research Challenges,"
Computer Networks Journal,

	vol. 72, 29 October 2014, pp. 74-98.
[J6]	M. Abu Sharkh, M. Jammal, A. Ouda, and A. Shami, "Resource Allocation In A Network-Based Cloud Computing Environment: Design Challenges," *IEEE Communication Magazine*, vol. 51, no. 11, Nov. 2013, pp. 46-52.
[C1]	M. Jammal, H. Hawilo, A. Kanso, and A. Shami, "Mitigating the Risk of Cloud Services Downtime Using Live Migration and High Availability-Aware Placement," *IEEE International Conference on Cloud Computing Technology and Science*, December 2016.
[C2]	M. Jammal, A. Kanso, P. Heidari, and A. Shami, "A Formal Model for the Availability Analysis of Cloud Deployed Multi-Tiered Applications," *IEEE International Symposium on Software Defined Systems*, April 2016.
[C3]	M. Jammal, A. Kanso, P. Heidari, and A. Shami, "Availability Analysis of Cloud Deployed Applications," *IEEE International Conference on Cloud Engineering*, April 2016.
[C4]	M. Jammal, A. Kanso, and A. Shami, "High Availability-Aware Optimization Digest for Applications Deployment in Cloud," *IEEE International Conference on Communications (ICC)*, June 2015.
[C5]	M. Jammal, A. Kanso, and A. Shami, "CHASE: Component High-Availability Scheduler in Cloud Computing Environment," *IEEE International Conference on Cloud Computing*, June 2015.
[C6]	H. Sbeity, R. Younes, and M. Jammal, "Improvement of Markov Chain Processes for Mathematical Optimization of Cancer Treatment," *IEEE Conference on Biomedical Engineering and Sciences*, 2014.