# AUTOMATED CLOUD RESOURCE ORCHESTRATION

Changbin Liu

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2012

---

Boon Thau Loo
Assistant Professor, Computer and Information Science
Supervisor of Dissertation

---

Jianbo Shi
Associate Professor, Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Jonathan M. Smith (Chair), Professor, Computer and Information Science

Zachary G. Ives, Associate Professor, Computer and Information Science

Oleg Sokolsky, Research Associate Professor, Computer and Information Science

Yun Mao, Senior Member of Technical Staff, AT&T Labs Research

Prithwish Basu, Senior Scientist, Raytheon BBN Technologies

*Dedicated to my family*

# ACKNOWLEDGEMENT

Five years is a long journey. It is amazing to see how things have changed and how many people have accompanied, helped and influenced me during the journey. The completion of this dissertation is impossible without them.

First, I am deeply grateful to my extraordinary advisor Professor Boon Thau Loo. Boon has been a role model for me by his talent, passion, diligence and persistence throughout my years at Penn. I feel extremely fortunate to have the opportunity to work with him. Boon's extensive knowledge in networking, database, security, formal methods, and cloud computing was instrumental in helping me broaden my research view. Moreover, Boon spent substantial time providing me with excellent and comprehensive guidance on conducting research in a professional manner, ranging from research topic selection, system design and implementation, collaborating with people, writing skills, talk presentation, job and internship searching, to final dissertation. Whenever I had difficulty in research, Boon was always there. Boon was also considerate on my non-research life, and he often generously shared with me valuable life experiences and wisdom. I can never thank him enough for the positive impact that he has on my life.

My research in graduate school centered around cloud computing and wireless networking. In the context of these two areas, I have had the privilege of collaborating extensively with two outstanding researchers–Yun Mao and Prithwish Basu.

They are not only my research collaborators, but also my mentors, and they are serving on my thesis committee. Both of them made significant contributions which made my research possible. We had countless stimulating discussions that led to new ideas. I also want to express my gratitude to Yun for offering me chances to do summer interns at AT&T Labs Research in 2010 and 2011. The work at AT&T evolves into part of this dissertation. Yun gave me tremendous guide on computer science skills, including system deployment, programming, debugging, testing and performance analysis. I immensely enjoyed the time when we were coding together.

I am thankful to my other mentors at AT&T Labs Research. I am very lucky to work with these knowledgeable and experienced researchers: Mary Fernandez, Kobus van der Merwe, Xu Chen, Edwin Lim and Emmanuil Mavrogiorgis.

I want to express my gratitude to Professor Jonathan Smith, Professor Zack Ives and Professor Oleg Sokolsky for serving on my thesis committee. They provided valuable comments and suggestions for improving this dissertation. I also thank Jonathan for chairing my thesis committee and WPE-II committee.

I would like to thank my colleagues and friends at Penn. Wenchao Zhou and I came to Penn in the same day and also finished our oral defenses in the same day. We went through a lot of joy and pain together. I have worked closely with Rick Correa, Xiaozhou Li, Shiv Muthukumar, Ren Lu, Taher Saeed, Tanveer Gill, Harjot Gill, Anduo Wang, Joseph Kopena and Mihai Oprea on a variety of research projects. They are also my great friends. Many thanks to other friends, including but not limited to: Zhijun Liu, Xi Weng, Jianzhou Zhao, Wei Di, Mengmeng Liu, Zhuowei Bao and Bin Yan.

My research was funded in part by the following funding agencies and sources: National Science Foundation under contract CNS-0721845, CNS-0831376, CNS-0845552, CNS-1040672, CCF-0820208, DARPA through Air Force Research

Laboratory (AFRL) Contract FA8750-07-C-0169, and AT&T Labs Research summer student internship program. I appreciate their support.

Last but not the least, I would like to thank my parents Mingzhen Jiang and Songping Liu for their unlimited support, encouragement, and love. My girlfriend Penghui Sun takes the best care of me for these years. I was always her top priority especially when paper deadlines were close. Without her support I could not possibly come anywhere close to finishing the dissertation.

ABSTRACT

AUTOMATED CLOUD RESOURCE ORCHESTRATION

Changbin Liu

Boon Thau Loo

Realizing Infrastructure-as-a-Service (IaaS) cloud requires a control platform for orchestrating the provisioning, configuration, management and decommissioning of a distributed set of diverse cloud resources (i.e., compute, storage, network) serving different clients. Cloud resource orchestration is challenging due to the rapid growth of data centers, the high failure rate of commodity hardware, the enforcement of service and engineering rules, the increasing sophistication of cloud services, and the requirement to fulfill provider operational objectives and customer service level agreements (SLAs).

Towards addressing these challenges, this dissertation makes following contributions: (1) An automated resource orchestration platform that allows cloud operators to declaratively specify optimization goals and constraints given provider operational objectives and customer SLAs. Based on these specifications, orchestration commands are automatically generated to optimize resource configurations and allocations within the cloud; (2) A highly available transactional resource orchestration platform for building IaaS cloud infrastructures. Transactional orchestration procedures automatically guarantee atomicity, consistency, isolation and durability (ACID) properties for cloud operations. Transactional semantics provide a clean abstraction which enables cloud operators to focus on developing high level cloud services without worrying about the complexities of accessing and managing underlying volatile distributed resources.

We present the design and implementation of our transactional automated cloud

orchestration platform. Using realistic scenarios and workloads derived from production cloud services, we demonstrate that our platform is able to automatically orchestrate compute, storage, and network resources within and across geographically distributed data centers to meet operational objectives and SLAs.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

The Infrastructure-as-a-Service (IaaS) cloud computing model exemplified by Amazon EC2 [2] provides users on-demand, near-instant access to a large pool of virtual cloud resources (*i.e.,* compute, storage and network) such as virtual machines (VMs), virtual block devices, and virtual private networks. The **orchestrations** [66] of the virtual resources over physical hardware, such as provisioning, configuration, management and decommissioning, are exposed to the users as a service via programmable APIs. These APIs hide the complexity of the underlying orchestration details.

From the cloud provider's perspective, however, building a robust system to orchestrate cloud resources is challenging. First, today's large data centers typically run on the scale of over 10,000 machines based on commodity hardware [50]. As such, software glitches and hardware failures including power outages and network partitions are the norm rather than the exception. This unreliability not only impacts the virtual resources assigned to users, but also the controllers that orchestrate

the virtual resources. Second, to orchestrate a massively concurrent, multi-tenant IaaS environment, any engineering and service rule must be met while avoiding race conditions.

Moreover, cloud resource orchestration is highly complex due to the increasing sophistication of cloud services. First, as many recent proposals [93, 100, 33, 99, 34, 46] have articulated, cloud management is inherently complicated due to the heterogeneity, infrastructure, and concurrent user services that share a common set of physical resources. Second, configurations of various resource types interact with each other. For example, the locations of VMs have an impact on storage placement, which in turn affects bandwidth utilization within and external to a data center. Third, cloud resources have to be deployed in a fashion that not only realizes provider operational objectives, but also guarantees that the customer service-level agreements (SLAs) can be constantly met as runtime conditions change.

All in all, the orchestration process is complex and potentially tedious and error-prone if performed manually, and motivates the need for scalable and reliable management tools that enable us to **automate** part or all of the decision process without worrying about the complexities of accessing and managing underlying volatile distributed cloud resources.

## 1.2 Existing Solutions

On the public side, there are open-source IaaS cloud control platforms, such as OpenStack [23], OpenNebula [22] and Eucalyptus [12]. First, none of these solutions perform automated cloud resource orchestration. Second, we found a lack of disciplined approaches in them to enforce service and engineering rules – most of them are implemented as condition checks (if-then-else), which are scattered in the code

base. Even when violations are detected, the reconciliation mechanisms are usually ad-hoc and problematic. Third, when unanticipated errors occur during orchestration process, they either choose to fail-stop, entering an intermediate state, or to ignore the errors, leading to undefined behavior. Either way, without proper error handling, the system could end up in unknown and usually undesirable states, causing security issues or resources under-utilization. Last but not least, these solutions have limitations in terms of scale, concurrency control, and high availability. For instance, Eucalyptus can only handle 750–800 VMs (a hard limit) [37], race conditions exist in OpenStack concurrent VM spawnings even in a small scale cluster deployment [26], and high availability is entirely missing in all these platforms.

On the proprietary side, a variety of commercial IaaS providers exist, such as Amazon EC2 [2], Microsoft Windows Azure [29] and VMWare Distributed Resource Scheduler (DRS) [9] . Unfortunately, how they orchestrate cloud resources is proprietary. However, we can still infer that these platforms are not ideal. For instance, as anecdotally indicated by the outage report [28], in April 2011 EC2 encountered problems in enforcing safety and concurrency: a human error in router configuration that violates an implicit service rule and a race condition in storage provisioning contributed significantly to the prolonged downtime. Due to similar reasons, another EC2 outage occurred in August 2011 [27], and Microsoft encountered a cloud outage in September 2011 [4]. More importantly, these solutions are insufficient in orchestrating increasingly sophisticated cloud services due to the lack of appropriate automation tools and programming abstractions. For instance, though VMWare DRS is able to perform automated VM migration, it is narrowly scoped for load balancing and does not allow cloud operators to flexibly customize their policy goals and constraints based on various demands and changing conditions (*e.g.,* migration cost,

load consolidation, data consistency requirement, latency and bandwidth guarantee for customers).

## 1.3 Approach Overview

This dissertation in particular aims to explore the following challenging research questions in cloud resource orchestration:

1. **How can we easily automate cloud resource orchestration?** Given provider operational objectives and customer SLAs as policy goals and constraints, we desire that the platform enables cloud operators to formally model cloud orchestrations, and that it takes the specifications and automatically synthesizes orchestration decisions. As cloud services become increasingly sophisticated, their orchestration logic is complex. In specifying the orchestration logic, imperative languages like C++ [13] or Java [5] often result in multi-hundred or even multi-thousand lines of code, which is error-prone and stove-piped hence inflexible at customizing policies upon user demands. To ease the process of specifying automated orchestration, we aim to design a compact language that allows cloud providers to specify orchestration policies succinctly. This language should as well be general enough to capture a variety of cloud resource orchestration scenarios within and across geographically distributed data centers.

2. **How can we guarantee orchestration robustness in the presence of cloud volatility?** Robustness ensures that failures in an orchestration procedure do not lead to undefined behavior or inconsistent states. This goal is

critical and yet especially challenging because of high volatility in the cloud environment. An orchestration procedure usually involves multiple state changes of distributed resources, any of which can fail due to volatility. For example, spawning a VM typically has the following steps: clone a VM disk image on a storage server; create a VM configuration on a compute server; set up virtual local-area networks (VLAN), software bridges, and firewalls for inter-VM communication; finally start the VM. During the process, an error at any step would prevent the client from obtaining a working VM. Worse, the leftover configurations in the compute, storage and network components become orphans if not properly cleaned up, which may lead to undefined behavior for future orchestrations.

3. **How can we ensure a cloud service is safe?** I.e., the service's orchestration procedures do not violate any constraints. These constraints reflect service and engineering rules in operation. If violated, an illegal orchestration operation could disrupt cloud services or even lead to severe outage [28], *e.g.,* spawning a VM on an overloaded compute server, or migrating a VM to an incompatible hypervisor or CPU with different instruction sets. Enforcing these constraints is challenging as it often requires acquiring the states of distributed resources and reasoning about them holistically.

4. **How can we allow high concurrency of simultaneous orchestrations?** Our objective is to provide a cloud orchestration system at the scale of at least 100,000 cloud resources (*e.g.,* VMs and block devices) [17]. It should be able to perform simultaneous execution of massive orchestration procedures safely, especially when they access the same resources. For example, simultaneous spawning of two VMs on the same compute server may exceed the physical

memory limit of the server. Concurrency control guarantees that simultaneous execution of orchestration procedures avoids race conditions and permits the system to scale.

5. **How can we make a cloud orchestration platform highly-available?** In the era of web-scale applications, unavailability of the cloud system directly translates to loss of revenue and service degradation for customers. Based on our estimation of Amazon EC2's rate of VM creation [3], a mere 10-minute service disruption can result in not fulfilling over 1,400 VM spawn operations in a single region. Such disruptions are unacceptable for mission-critical applications.

To address the above challenges, in this dissertation we present a new point in the design spaces of cloud resource orchestration architectures that aims to achieve scalable and reliable orchestration automation. Specifically, we propose a unified architecture which includes two parts:

- An automated resource orchestration platform [63, 67] that enables cloud operators to specify optimization policy goals and constraints given provider operational objectives and customer SLAs. These policies are compactly specified in a declarative language, which results in orders of magnitude code size reduction compared to imperative languages. Based on these specifications, orchestration commands are automatically generated via integrating a declarative networking engine [69] with a general constraint solver [13] to optimize resource configurations and allocations within and across distributed data centers [39] in the cloud.

- A highly available transactional resource orchestration platform [66, 64] for building IaaS cloud infrastructures. This platform provides an abstraction

to execute the generated commands in a *transactional* style to guarantee orchestration correctness. Transactional orchestration procedures automatically guarantee atomicity, consistency, isolation and durability (ACID) properties for cloud operations. Transactional semantics provide a clean abstraction which enables cloud operators to focus on developing high level automated cloud services without worrying about the complexities of accessing and managing underlying volatile distributed resources.

The hypothesis of this dissertation is that our approach of automated cloud resource orchestration can address the aforementioned challenges. To validate this hypothesis, we have designed and implemented a prototype transactional automated cloud orchestration platform, which is targeted to be used by both private and public cloud operators and service providers, such as Amazon [2], Cloudscaling [8], Rackspace [6], RightScale [7], and Windows Azure [29]. Currently these providers use open-source platforms such as OpenStack [23] or proprietary solutions, which are inadequate in addressing all the problems of orchestration automation, robustness, safety, concurrency and high availability. In Table 1.1 we list the comparisons between our platform and current solutions in terms of addressing the research questions.

| System | OpenStack | Eucalyptus | OpenNebula | Amazon EC2 | Our platform |
|---|---|---|---|---|---|
| Automation | × | × | × | ? | ✓ |
| Robustness | Fail-stop or ignore unexpected errors. Require human intervention | Fail-stop or ignore unexpected errors. Require human intervention | Fail-stop or ignore unexpected errors. Require human intervention | ? | ✓ |
| Safety | Scattered if-then-else | Scattered if-then-else | Scattered if-then-else | Outage in April and August 2011 [28, 27] | ✓ |
| Concurrency | Race conditions (Errors in concurrent VM spawning [26]) | Small scale deployment (750-800 VMs [37]) | ? | Outage in April and August 2011 [28, 27] | ✓ |
| High availability | × | × | × | ? | ✓ |

Table 1.1: Comparisons between current solutions and our platform.

## 1.4 Organization

The remainder of this dissertation presents the architecture of our transactional automated cloud resource orchestration platform, its declarative programming language, two use cases and evaluation of its effectiveness. Chapter 2 presents an architectural overview of the system. Chapter 3 describes how cloud resource orchestration is automated via being formulated as declarative constraint optimizations. Chapter 4 describes how generated resource orchestration commands are executed in a transactional style to guarantee safety, robustness and to provide concurrency control. Chapter 5 presents our evaluation results. Chapter 6 presents related work. Finally, Chapter 7 summarizes the dissertation and discusses future directions.

# Chapter 2

# System Overview

Figure 2.1 presents an overview of the system we are aiming to design and implement in this dissertation. This system is designed for a cloud environment comprising of geographically distributed data centers via dedicated backbone networks or the Internet. At the top, cloud operators specify providers operational objectives and customer SLAs as input. In the middle, the cloud resource orchestration system is comprised of two components, COPE (Cloud Orchestration Policy Engine) [63, 67], and TROPIC (Transactional Resource Orchestration Platform In the Cloud) [66, 64].

COPE is an *automated orchestration* platform that enables cloud providers to automate the process of resource orchestration. In COPE, there are a distributed network of instances communicating with each other. Provider operational objectives and customer SLAs from cloud operators are specified in terms of policy *goals*, which are subjected to a number of *constraints* specific to the cloud deployment scenario. COPE formulates these specifications as well as underlying cloud system states reported by TROPIC as a constraint optimization problem (COP), and automatically synthesizes orchestration commands which are input into TROPIC to be executed as actual orchestration operations.

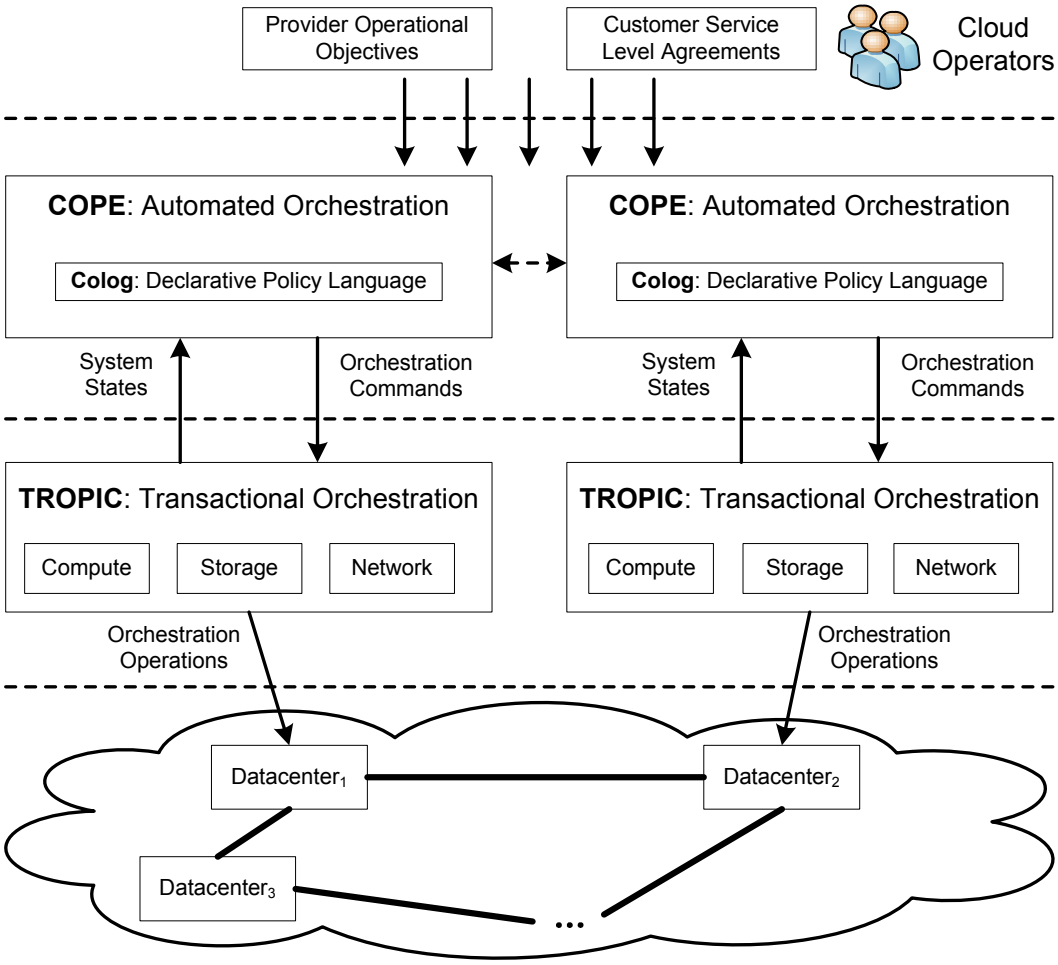


Figure 2.1: System architecture.

To ease the process of specifying orchestration COP models, COPE uses a declarative policy language *Colog* to specify policy goals and constraints. *Colog* specifications are compact and close to mathematical formulation of COPs. *Colog* originates from declarative networking [69], and it is a distributed variant of Datalog with extended constraint solving and distributed optimization capabilities. *Colog* is a general policy language that can be widely applied to capture a variety of cloud resource orchestration scenarios within and across data centers.

Residing underneath COPE, TROPIC manages the underlying cloud resources

(*i.e.,* compute, storage and network) and performs actual orchestration operations to manipulate the resources. TROPIC is a *transactional orchestration* platform with a unified data model that enables cloud providers to develop complex cloud services with safety, concurrency, robustness and high availability. Each data center runs one TROPIC instance. TROPIC translates high-level orchestration commands automatically generated by COPE to low-level operations and executes them in a *transactional* style with ACID properties (atomicity, consistency, isolation and durability). Transactional semantics provide a clean abstraction to cloud providers to ensure that, orchestrations that encounter unexpected errors have no effect, concurrent orchestrations do not violate safety rules or cause race conditions, and committed orchestrations persist on physical devices. As a result, service developers only need to focus on developing high level cloud services without worrying about the complexities of accessing and managing volatile distributed resources.

TROPIC and COPE are two loosely-coupled components in our platform. To orchestrate the cloud, COPE first completes constraint solving, and then the optimization output is all gathered before being sent to TROPIC for transactional execution. In this way, even if TROPIC transactions abort, it does not affect the execution of COPE constraint optimization. Cloud operators only need to rerun *Colog* programs for reoptimizations.

If COPE is deployed in the centralized mode, its failure will result in no automation commands being generated. Cloud operators have to restart COPE in this case. If COPE is deployed in the distributed mode and one or multiple COPE instances fails, the constraint optimization process may produce partial solutions, *i.e.,* only running COPE instances get optimization results. TROPIC adopts a highly available design via using replicated decentralized components, however, it can still

experience failures if there are data center-wide outages. Under this scenario, no orchestration command can be executed in the affected data centers.

# Chapter 3

# Declarative Automated Orchestration

In this chapter, we present COPE (Cloud Orchestration Policy Engine) [63]. COPE aims to answer the research question 1 in Chapter 1. COPE is a declarative optimization platform that enables cloud providers to formally model cloud resources and formulate orchestration decisions as a *constraint optimization problem* (COP) given policy goals and constraints. Central to COPE is the integration of a *declarative networking* [69] engine with an off-the-shelf constraint solver [13]. COPE automates the process of cloud orchestration via the use of a declarative policy language named *Colog*. We have developed the *Colog* language that combines distributed Datalog used in declarative networking with language constructs for specifying goals and constraints used in COPs. Moreover, we apply COPE to two representative cloud orchestration use cases that allow us to showcase key features of COPE.

## 3.1 COPE Architecture

Figure 3.1 presents a system overview of COPE. COPE takes the provider operational objectives and customer SLAs from cloud operators as input. COPE can be deployed in either a *centralized* or *distributed* mode.

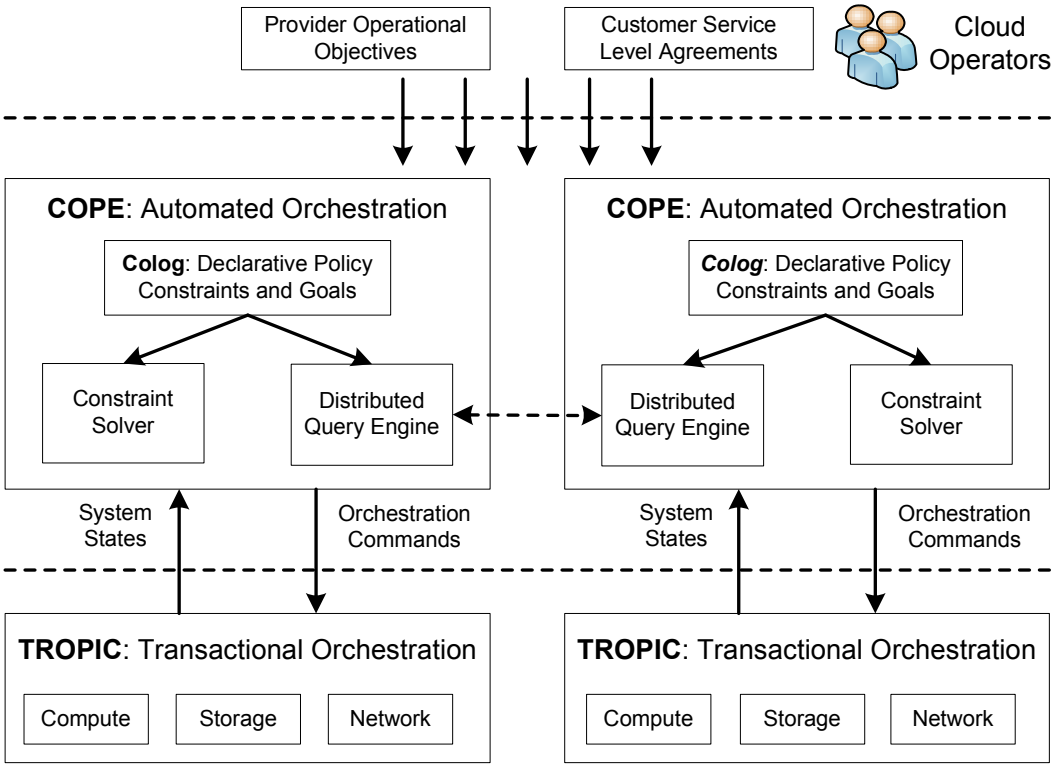


Figure 3.1: COPE Architecture.

In the centralized deployment scenario, the entire cloud is configured by one centralized COPE instance. Provider operational objectives and customer SLAs are specified in terms of *goals*, which are subjected to a number of *constraints* specific to the cloud deployment scenario. These policy constraints and goals are specified in a declarative policy language called *Colog*. COPE additionally takes as input system states (*e.g.,* node CPU load, memory usage, network traffic statistics) gathered from the cloud. The specifications are used by a *constraint solver* to automatically

synthesizes *orchestration commands*, which are then input into TROPIC to perform physical orchestration operations to coordinate resources in each data center. As we will discuss in Chapter 4, TROPIC executes orchestration commands in a *transactional* style, where the ACID properties are preserved for a series of commands grouped within a transaction.

In the distributed deployment scenario, there are multiple COPE instances, typically one for each data center. Each COPE node has a set of neighbor nodes that it can directly communicate with via dedicated backbone networks or the Internet. A *distributed query engine* [25] is used to coordinate the exchange of system states and optimization output amongst COPE instances, in order to achieve a global objective (this typically results in an approximate solution).

A distributed deployment brings two advantages. First, cloud environments like federated cloud [39] may be administered by different cloud providers. This necessitates each provider running its own COPE for its internal configuration, but coordinating with other COPE nodes for inter-data center configurations. Second, even if the cloud is entirely under one administrative domain, for scalability reasons, each cloud operator may choose to configure a smaller set of resources using local optimization commands.

## 3.2 Use Case Examples

We use the following two scenarios (*ACloud* and *Follow-the-Sun*) as our driving examples throughput the thesis. Both examples are representative of cloud resource orchestration scenarios within and across data centers, respectively. We will primarily frame our discussions of the use cases in terms of COP expressed mathematically,

and defer the declarative language specifications and runtime support for realizing these COP computations to later sections.

### 3.2.1 ACloud (Adaptive Cloud)

We consider the cloud service *ACloud*, a simplified version of what current cloud service providers [2] might offer. In ACloud, a customer may spawn new VMs from an existing disk image, and later start, shutdown, or delete the VMs. In today's deployment, providers typically perform load balancing in an ad-hoc fashion. For instance, VM migrations can be triggered at an overloaded host machine, whose VMs are migrated to a randomly chosen machine currently with light load. While such ad-hoc approaches may work for a specific scenario, they are unlikely to result in configurations that can be easily customized upon changing policy constraints and goals, whose optimality cannot be easily quantified.

As an alternative, COPE takes as input real-time system states (*e.g.,* CPU and memory load, migration feasibility), and a set of policies specified by the cloud provider. An example optimization goal is to reduce the cluster-wide CPU load variance across all host machines, so as to avoid hot-spots. Constraints can be tied to each machine's resource availability (*e.g.,* each machine can only run up to a fixed number of VMs, run certain classes of VMs, and not exceed its physical memory limit), or security concerns (VMs can only be migrated across certain types of hosts).

Another possible policy is to minimize the total number of VM migrations, as long as a load variance threshold is met across all hosts. Alternatively, to consolidate workloads one can minimize the number of machines that are hosting VMs, as long as each application receives sufficient resources to meet customer demands. Given these

optimization goals and constraints, COPE can be executed periodically, triggered whenever imbalance is observed, or whenever VM CPU and memory usage changes.

Given the ACloud scenario, we provide its COP-based mathematical model. A COP takes as input a set of *constraints*, and attempts to find an assignment of values chosen from an domain to a set of *variables* to satisfy the constraints under an *optimization goal*. The goal is typically expressed as a minimization over a cost function of the assignments.

In ACloud, suppose cloud providers decide to migrate VMs within a data center in order to balance the load. In this model, there are in total $n$ VMs, denoted as $V_i$, where $i = 1, 2, \ldots, n$. Each VM currently consumes CPU $C_i$ and memory $M_i$. This monitored information is reported by the underneath TROPIC platform, which regularly updates CPU and memory utilization data. There are $m$ compute servers which host the VMs, denoted as $H_j$, where $j = 1, 2, \ldots, m$. Each host $H_j$ has a physical memory capacity of $HostMemCap_j$. Given the resource capacity and demand, the optimization variables of this COP is a VM-to-host assignment $A_{ij}$, which means that VM $V_i$ is assigned to host $H_j$. The value of $A_{ij}$ is 1 (*i.e.,* VM $V_i$ is assigned to host $H_j$) or 0 (otherwise). If we assume that the goal of load balancing is to minimize the standard deviation of host CPUs across the data center, then the COP can be formulated as followings:

$$\min \quad hostStdevCpu \tag{3.1}$$

$$hostAvgCpu = \frac{1}{m}\sum_{j=1}^{m}(\sum_{i=1}^{n}(A_{ij} * C_i)) \tag{3.2}$$

$$hostStdevCpu = \sqrt{\frac{1}{m}\sum_{j=1}^{m}(\sum_{i=1}^{n}(A_{ij} * C_i) - hostAvgCpu)^2} \tag{3.3}$$

subject to:

$$\forall i: \quad \sum_{j=1}^{m} A_{ij} == 1 \tag{3.4}$$

$$\forall j: \quad \sum_{i=1}^{n}(A_{ij} * M_i) <= HostMemCap_j \tag{3.5}$$

**Optimization goal:** The COP aims to minimize the CPU standard deviation *hostStdevCpu* across all hosts. Formula (3.2) aggregates the CPU of all VMs running on each host and computes the average host CPU. Formula (3.3) takes the output of (3.2) and computes the system-wide standard deviation of hosts CPUs. The output from (3.3) is later used by the constraint solver for exploring the search space that meets the optimization goal.

**Constraints.** The COP is subjected to two representative constraints. Constraint (3.4) ensures that each VM is assigned to one and only one host. Constraint (3.5) expresses that no host can accommodate VMs whose aggregate memory exceeds its physical capacity.

### 3.2.2 Follow-the-Sun

Our second motivating example is based on the Follow-the-Sun scenario [93], which aims to migrate VMs across geographical distributed data centers based on customer dynamics. Here, the geographic location of the primary workload (*i.e.,* majority of customers using the cloud service) derives demand shifts during the course of a day, and it is beneficial for these workload drivers to be in close proximity to the resources

they operate on. The migration decision process has to occur in real-time on a live deployment with minimal disruption to existing services.

In this scenario, the workload migration service aims to optimize for two parties: for providers, it enables service consolidation to reduce operating costs, and for customers, it improves application performance while ensuring that customer SLAs of web services (*e.g.,* defined in terms of the average end-to-end experienced latency of user requests) are met. In addition, it may be performed to reduce inter-data center communication overhead [103, 34]. Since data centers in this scenario may belong to cloud providers in different administrative domains (similar to federated cloud [39]), Follow-the-Sun may be best suited for a distributed deployment, where each COPE instance is responsible for controlling resources within their own data center.

We present a COP-based mathematical model of the Follow-the-Sun scenario. In this model, there are $n$ autonomous geographically distributed data centers $C_1, ..., C_n$ at location $1, 2, ..., n$. Each data center is managed by one COPE instance. Each site $C_i$ has a resource capacity (set to the maximum number of VMs) denoted as $R_i$. Each customer specifies the number of VMs to be instantiated, as well as a preferred geographic location. We denote the aggregated resource demand at location $j$ as $D_j$, which is the sum of total number of VMs demanded by all customers at that location. Given the resource capacity and demand, data center $C_i$ currently allocates $A_{ji}$ resources (VMs) to meet customer demand $D_j$ at location $j$.

In the formulation, $M_{ijk}$ denotes the number of VMs migrated from $C_i$ to $C_j$ to meet $D_k$. Migration is feasible only if there is a link $L_{ij}$ between $C_i$ and $C_j$. When $M_{ijk} > 0$, the cloud orchestration layer will issue commands to migrate VMs accordingly. This can be periodically executed, or executed on demand whenever system parameters (*e.g.,* demand $D$ or resource availability $R$) change drastically.

A naïve algorithm is to always migrate VMs to customers' preferred locations. However, it could be either impossible, when the aggregated resource demand exceeds resource capacity, or suboptimal, when the operating cost of a designated data center is much more expensive than neighboring ones, or when VM migrations incur enormous migration cost.

In contrast, COPE's COP approach attempts to optimize based on a number of factors captured in the cost function. In the model, we consider three main kinds of cost: (1) operating cost of data center $C_j$ is defined as $OC_j$, which includes typical recurring costs of operating a VM at $C_j$; (2) communication cost of meeting resource demand $D_i$ from data center $C_j$ is given as $CC_{ij}$; (3) migration cost $MC_{ij}$ is the overhead of moving a VM from $C_i$ to $C_j$. Given the above variables, the COP formulation is:

$$\min \quad (aggOC + aggCC + aggMC) \tag{3.6}$$

$$aggOC = \sum_{j=1}^{n}(\sum_{i=1}^{n}(A_{ij} + \sum_{k=1}^{n} M_{kji}) * OC_j) \tag{3.7}$$

$$aggCC = \sum_{j=1}^{n}\sum_{i=1}^{n}((A_{ij} + \sum_{k=1}^{n} M_{kji}) * CC_{ij}) \tag{3.8}$$

$$aggMC = \sum_{i=1}^{n}\sum_{j=1}^{n}((\sum_{k=1}^{n} max(M_{ijk}, 0)) * MC_{ij}) \tag{3.9}$$

subject to:

$$\forall j : R_j \geq \sum_{i=1}^{n}(A_{ij} + \sum_{k=1}^{n} M_{kji}) \tag{3.10}$$

$$\forall i, j, k : M_{ijk} + M_{jik} = 0 \tag{3.11}$$

**Optimization goal.** The COP aims to minimize the aggregate cost of cloud providers. In the above formulation, it is defined as the sum of the aggregate operating cost $aggOC$ in (3.7) across all data centers, the aggregate communication cost $aggCC$ in (3.8) to meet customer demands served at various data centers, and the aggregate VM migration cost $aggMC$ in (3.9), all of which are computed by summing up $OC_j$, $CC_{ij}$, and $MC_{ij}$ for the entire system.

**Constraints.** The COP is subjected to two representative constraints. In Constraint (3.10), each data center cannot allocate more resources than it possesses. Constraint (3.11) ensures the zero-sum relation between migrated VMs between $C_i$ and $C_j$ for demand $k$.

## 3.3 *Colog* Language

The above COP mathematical formulations can be specified and solved in any general constraint solvers, such as Gecode [13] and Choco [5]. However, we note that if specified in imperative languages like C++ or Java, the COP program typically consists of multi-hundred lines of code, if not multi-thousand, and the program is error-prone and inflexible to customize as user demands may vary over time. COPE uses a declarative policy language *Colog* to concisely specify the COP formulation in the form of policy goals and constraints. *Colog* is based on Datalog [69], a recursive query language used in the database community for querying graphs. Our choice of *Colog* is driven by its conciseness in specifying dependencies among system states, including distributed system states that exhibit recursive properties. Its root in

logic provides a convenient mechanism for expressing modularized solver goals and constraints. *Colog* enables clean separation of policy goals and constraints for better usability. Moreover, there exists distributed Datalog engines [25] that will later facilitate distributed COP computations.

Using as examples ACloud and Follow-the-Sun from Section 3.2, we present the *Colog* language and describe its execution model. The detailed language grammar and syntax can be found in Appendix A. In the rest of this section, we first introduce centralized *Colog* (without constructs for distribution), followed by distributed *Colog*.

### 3.3.1 Datalog Conventions

Throughout this dissertation, we use Datalog conventions in [79] to present *Colog*. A Datalog program consists of a set of declarative *rules*. Each rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`q1` and `q2` and `...` and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1`, `q2`,...,`qn` is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* with *attributes*, or boolean expressions that involve function symbols (including arithmetic) applied to attributes. The predicates in traditional Datalog rules are relations, and we will refer to them interchangeably as predicates, relations, or tables.

Datalog rules can refer to one another in a mutually recursive fashion. The order in which the rules are presented in a program is semantically immaterial; likewise, the order predicates appear in a rule is not semantically meaningful. Commas are interpreted as logical conjunctions (*AND*). Conventionally, the names of predicates, function symbols, and constants begin with a lowercase letter, while attribute names begin with an uppercase letter. Function calls are additionally prepended by `f_`.

Aggregate constructs (*e.g.,* SUM, MIN, MAX) are represented as functions with attributes within angle brackets (<>).

### 3.3.2 Centralized *Colog*

*Colog* extends traditional Datalog with constructs for expressing goals and constraints and also distributed computations. We defer the discussion of distribution to Section 3.3.3, and primarily focus on centralized *Colog* here.

*Colog* specifications are compiled into execution plans executed by a Datalog evaluation engine that includes modules for constraint solving. In *Colog* program, two reserved keywords `goal` and `var` specify the goal and variables used by the constraint solver. The type of goal is either *minimize*, *maximize* or *satisfy*. As its name suggests, the first two minimizes or maximizes a given objective, and the third one means to find a solution that satisfies all given constraints.

*Colog* has two types of table attributes – *regular* and *solver*. A regular attribute is a conventional Datalog table attribute, while a solver attribute is either a constraint solver variable or is derived from existing ones. The difference between the two is that the actual value of a regular attribute is determined by facts within a database, *e.g.,* it could be an integer, a string, or an IP address. On the other hand, the value of a solver attribute is only determined by the constraint solver after executing its optimization modules.

We refer to tables that contain solver attributes as *solver tables*. Tables that contain only regular attributes are referred to as *regular tables*, which are essentially traditional Datalog based and derived tables.

Given the above table types, *Colog* includes traditional Datalog rules that only

contain regular tables, and *solver rules* that contain one or more solver tables. These solver rules can further be categorized as derivation or constraint rules:

- A **solver derivation rule** derives intermediate solver variables based on existing ones. Like Datalog rules, these rules have the form `p <- q1, q2, ..., qn.`, which results in the derivation of `p` whenever the rule body (`q1` and `q2` and `...` and `qn`) is true. Unlike regular Datalog rules, the rule head `p` is a solver table.

- A **solver constraint rule** has the form `p -> q1, q2, ..., qn.`, denoting the logical meaning that whenever the rule head `p` is true, the rule body (`q1` and `q2` and `...` and `qn`) must also be true to satisfy the constraint. In COPE, all constraint rules involve one or more solver tables in either the rule body or head. Unlike a solver derivation rule, which derives new variables, a constraint *restricts* a solver attribute's allowed values, hence representing an invariant that must be maintained at all times. Constraints are used by the solver to limit the search space when computing the optimization goal.

A compiler can statically analyze a *Colog* program to determine whether it is a Datalog rule, or a solver derivation/constraint rule. For ease of exposition, we add a rule label prefix `r`, `d`, and `c` to regular Datalog, solver derivation, and solver constraint rules respectively.

As an example, the following program expresses a COP that aims to achieve load-balancing within a data center for the ACloud resource orchestration scenario in Section 3.2. This example is centralized, and we will revisit the distributed extensions in the next section.

```
goal minimize C in hostStdevCpu(C).
var  assign(Vid,Hid,V) forall toAssign(Vid,Hid).

r1  toAssign(Vid,Hid) <- vm(Vid,Cpu,Mem), host(Hid,Cpu2,Mem2).

d1  hostCpu(Hid,SUM<C>) <- assign(Vid,Hid,V), vm(Vid,Cpu,Mem),
                           C==V*Cpu.
d2  hostStdevCpu(STDEV<C>) <- host(Hid,Cpu,Mem), hostCpu(Hid,Cpu2),
                              C==Cpu+Cpu2.

d3  assignCount(Vid,SUM<V>) <- assign(Vid,Hid,V).
c1  assignCount(Vid,V) -> V==1.

d4  hostMem(Hid,SUM<M>) <- assign(Vid,Hid,V), vm(Vid,Cpu,Mem),
                           M==V*Mem.
c2  hostMem(Hid,Mem) -> hostMemCap(Hid,M), Mem<=M.
```

**Program description.** The above program takes as input `vm(Vid,Cpu,Mem)` and `host(Hid,Cpu,Mem)` tables, which are regular tables. Each `vm` entry stores information of a VM uniquely identified by `Vid`. Additional monitored information (*i.e.*, its CPU utilization `Cpu` and memory usage `Mem`) are also supplied in each entry. This monitored information can be provided by the cloud infrastructure, which regularly updates CPU and memory attributes in the `vm` table. The `host` table stores the hosts' CPU utilization `Cpu` and memory usage `Mem`. Table 3.1 summarizes the

| COP | *Colog* |
|---|---|
| symbol $A_{ij}$ | `assign(I,J,V)` |
| symbol $C_i, M_i$ | `vm(I,C,M)` |
| symbol $H_j$ | `host(J)` |
| formula (3.1) | rule `goal` |
| formula (3.2), (3.3) | rule `d1, d2` |
| formula (3.4) | rule `d3, c1` |
| formula (3.5) | rule `d4, c2` |

Table 3.1: Mappings from ACloud COP to *Colog*.

mapping from COP symbols to *Colog* tables, and COP formulas to *Colog* rules/constraints identified by the rule labels. For instance, in the table the `V` attribute in `assign(I,J,V)` stores the value of $A_{ij}$. Given these input tables, the above program expresses the following:

- **Optimization goal:** Minimize the CPU standard deviation attribute `C` in `hostStdevCpu`.
- **Variables:** As output, the solver generates `assign(Vid,Hid,V)` entries. `V` are solver variables, where each entry indicates VM `Vid` is assigned to host `Hid` if `V` is 1 (otherwise 0). `assign(Vid,Hid,V)` is bounded via the keyword `forall` to `toAssign` table, generated by joining `vm` with `host` in rule `r1`.
- **Solver derivations:** Rule `d1` aggregates the CPU of all VMs running on each host. Rule `d2` takes the output from `d1` and then computes the system-wide standard deviation of the aggregate CPU load across all hosts. The output from `d2` is later used by the constraint solver for exploring the search space that meets the optimization goal. In most (if not all) *Colog* programs, the final optimization goal is derived from (or dependent on) solver variables.
- **Solver constraints:** Constraint `c1` expresses that each VM is assigned to one

and only one host, via first aggregating the number of VM assignments in rule `d3`. Similarly, constraint `c2` ensures that no host can accommodate VMs whose aggregate memory exceeds its physical limit, as defined in `hostMemCap`.

To invoke actual constraint solving, *Colog* uses a reserved event `invokeSolver` to trigger the optimization computation. This event can be generated either periodically, or triggered based on an event (local table updates or network messages). To restrict the maximum solving time for each COP execution, one can set the parameter `solver_time`.

Using *Colog*, it is easy to customize policies simply by modifying the goals, constraints, and adding additional derivation rules. For instance, we can add a rule (continuous query) that triggers the COP program whenever load imbalance is observed (*i.e.,* `C` in `hostStdevCpu` exceeds a threshold). Alternatively, we can optimize for the fewest number of unique hosts used for migration while meeting customer SLAs when consolidating workloads. If the overhead of VM migration is considered too high, we can limit the number of VM migrations, as demonstrated by the rules below.

```
d5  migrate(Vid,Hid1,Hid2,C) <- origin(Vid,Hid1),
                                assign(Vid,Hid2,V), Hid1!=Hid2,
                                C==V.
d6  migrateCount(SUM<C>) <- migrate(Vid,Hid1,Hid2,C).
d7  migrateVM(Vid,Hid1,Hid2) <- migrate(Vid,Hid1,Hid2,C), C==1.
c3  migrateCount(C) -> C<=MAX_MIGRATES.
```

In rule `d5`, the `origin` table records the current VM-to-host mappings, *i.e.,* VM `Vid` is running on host `Hid1`. Derivation rules `d5-7` counts how many VMs are to

be migrated after optimization. In `d5`, `C==V` means that if VM `Vid` is assigned to host `Hid2`, then VM `Vid` should be migrated from host `Hid1` to `Hid2`. Otherwise, `C` is not equal to `V`. Constraint rule `c3` guarantees that the total number of migrations does not exceed a pre-defined threshold `MAX_MIGRATES`.

### 3.3.3 Distributed Colog

*Colog* can be used for distributed optimizations, and we introduce additional language constructs to express distributed computations. *Colog* uses the *location specifier* `@` construct used in declarative networking [69], to denote the source location of each corresponding tuple. This allows us to write rules where the input data spans across multiple nodes, a convenient language construct for formulating distributed optimizations. We illustrate *Colog* using a simple example of two rules that computes all pairs of reachable nodes in a network:

```
r1 reachable(@S,N) :- link(@S,N).
r2 reachable(@S,D) :- link(@S,N), reachable(@N,D).
```

The rules `r1` and `r2` specify a distributed transitive closure computation, where rule `r1` computes all pairs of nodes reachable within a single hop from all input links (denoted by the `link`, and rule `r2` expresses that "if there is a link from `S` to `N`, and `N` can reach `D`, then `S` can reach `D`." The output of interest is the set of all `reachable(@S,D)` tuples, representing reachable pairs of nodes from `S` to `D`. By modifying this simple example, we can construct more complex routing protocols, such as the distance vector and path vector routing protocols [69].

To provide a concrete distributed example, we consider a distributed implementation of the Follow-the-Sun cloud resource orchestration model introduced in Section 3.2. At a high level, we utilize an iterative distributed graph-based computation strategy, in which all nodes execute a *local* COP, and then iteratively exchange COP results with neighboring nodes until a stopping condition is reached. In this execution model, data centers are represented as nodes in a graph, and a link exists between two nodes if resources can be migrated across them. The following *Colog* program implements the local COP at each node `X`:

```
goal minimize C in aggCost(@X,C).
var migVm(@X,Y,D,R) forall toMigVm(@X,Y,D).

r1  toMigVm(@X,Y,D) <- setLink(@X,Y), dc(@X,D).

// next-step VM allocations after migration
d1  nextVm(@X,D,R) <- curVm(@X,D,R1), migVm(@X,Y,D,R2), R==R1-R2.
d2  nborNextVm(@X,Y,D,R) <- link(@Y,X), curVm(@Y,D,R1),
                            migVm(@X,Y,D,R2), R==R1+R2.

// communication, operating and migration cost
d3  aggCommCost(@X,SUM<Cost>) <- nextVm(@X,D,R), commCost(@X,D,C),
                                 Cost==R*C.
d4  aggOpCost(@X,SUM<Cost>) <- nextVm(@X,D,R), opCost(@X,C),
                               Cost==R*C.
d5  nborAggCommCost(@X,SUM<Cost>) <- link(@Y,X), commCost(@Y,D,C),
                                     nborNextVm(@X,Y,D,R),
```

```
                                        Cost==R*C.
d6  nborAggOpCost(@X,SUM<Cost>) <- link(@Y,X), opCost(@Y,C),
                                   nborNextVm(@X,Y,D,R), Cost==R*C.
d7  aggMigCost(@X,SUMABS<Cost>) <- migVm(@X,Y,D,R),
                                   migCost(@X,Y,C), Cost==R*C.

// total cost
d8  aggCost(@X,C) <- aggCommCost(@X,C1), aggOpCost(@X,C2),
                     aggMigCost(@X,C3), nborAggCommCost(@X,C4),
                     nborAggOpCost(@X,C5), C==C1+C2+C3+C4+C5.

// not exceeding resource capacity
d9  aggNextVm(@X,SUM<R>) <- nextVm(@X,D,R).
c1  aggNextVm(@X,R1) -> resource(@X,R2), R1<=R2.
d10 aggNborNextVm(@X,Y,SUM<R>) <- nborNextVm(@X,Y,D,R).
c2  aggNborNextVm(@X,Y,R1) -> link(@Y,X), resource(@Y,R2), R1<=R2.

// propagate to ensure symmetry and update allocations
r2  migVm(@Y,X,D,R2) <- setLink(@X,Y), migVm(@X,Y,D,R1), R2:=-R1.
r3  curVm(@X,D,R) <- curVm(@X,D,R1), migVm(@X,Y,D,R2), R:=R1-R2.
```

**Program description.** Table 3.2 summarizes the mapping from COP symbols to *Colog* tables, and COP equations to *Colog* rules identified by the rule labels. For instance, each entry in table $R_i$ is stored as a `resource(I,R)` tuple. Likewise,

| COP | *Colog* |
|---|---|
| symbol $R_i$ | `resource(I,R)` |
| symbol $C_i$ | `dc(I,C)` |
| symbol $L_{ij}$ | `link(I,J)` |
| symbol $A_{ij}$ | `curVm(I,J,R)` |
| symbol $M_{ijk}$ | `migVm(I,J,K,R)` |
| equation (3.6) | rule `goal, d8` |
| equation (3.7) | rule `d4,d6` |
| equation (3.8) | rule `d3,d5` |
| equation (3.9) | rule `d7` |
| equation (3.10) | rule `d9-10,c1-2` |
| equation (3.11) | rule `r2` |

Table 3.2: Mappings from Follow-the-Sun COP to *Colog*.

the `R` attribute in `migVm(I,J,K,R)` stores the value of $M_{ijk}$. The distributed COP program works as follows.

- **Optimization goal:** Instead of minimizing the global total cost of all data centers, the optimization goal of this local COP is the total cost `C` in `aggCost` within a local region, *i.e.,* node `X` and one of its neighbors `Y`.

- **COP execution trigger:** Periodically, each node `X` randomly selects one of its neighbors `Y` (denoted as a `link(@X,Y)` entry) to initiate a *VM migration* process[1] `setLink(@X,Y)` contains the pair of nodes participating in the VM migration process. This in essence results in the derivation of `toMigVm` in rule `r1`, which directly triggers the execution of the local COP (implemented by the rest of the rules). The output of the *local* COP determines the quantity of resources `migVm(@X,Y,D,R)` that are to be migrated between `X` and `Y` for all entries in `toMigVm`.

[1]To ensure that only one of two adjacent nodes initiates the VM migration process, for any given `link(X,Y)`, the protocol selects the node with the larger identifier (or address) to carry out the subsequent process. This distributed link negotiation can be specified in 13 *Colog* rules, which we omit for brevity.

- **Solver derivations:** During COP execution, rule `d1` and `d2` compute the next-step VM allocations after migration for node `X` and `Y`, respectively. Rule `d3-6` derive the aggregate communication and operating cost for the two nodes. We note that rule `d2` and `d5-6` are *distributed* solver derivation rules (*i.e.,* not all rule tables are at the same location), and node `X` collects its neighbor `Y`'s information (*e.g.,* `curVm`, `commCost` and `opCost`) via *implicit* distributed communications. Rule `d7` derives the migration cost via aggregate keyword `SUMABS`, which sums the absolute values of given variables. Rule `d8` derives the optimization objective `aggCost` by summing all communication, operating and migration cost for both node `X` and `Y`.

- **Solver constraints:** Constraints `c1` and `c2` express that after migration node `X` and `Y` must not have too many VMs which exceed their resource capacity given by table `resource`. Rule `c2` is a distributed constraint rule, where `X` retrieves neighbor `Y`'s `resource` table over the network to impose the constraint.

- **Stopping condition:** At the end of each COP execution, the migration result `migVm` is propagated to immediate neighbor `Y` to ensure symmetry via rule `r2`. Then in rule `r3` both node `X` and `Y` update their `curVm` to reflect the changes incurred by VM migration. Above process is then iteratively repeated until all links have been assigned values, *i.e.,* migration decisions between any two neighboring data centers have been made. In essence, one can view the distributed program as a series of per-node COPs carried out using each node's constraint solver. The complexity of this program depends upon the maximum node degree, since each node at most needs to perform $m$ rounds of link negotiations, where $m$ is the node degree.

Our use of *Colog* declarative language provides ease in policy customizations. For example, we can impose restrictions on the maximum quantity of resources to be

migrated due to factors like high CPU load or router traffic in data centers, or impose constraints that the total cost after optimization should be smaller by a threshold than before optimization. These two policies can be defined as rules below.

```
d11 aggMigVm(@X,Y,SUMABS<R>) <- migVm(@X,Y,D,R).
c3  aggMigVm(@X,Y,R) -> R<=MAX_MIGRATES.

c4  aggCost(@X,C) -> originCost(@X,C2), C<=COST_THRES*C2.
```

Rule `d11` derives total VM migrations between `X` and `Y`. Constraint `c3` ensures that total migrations do not exceed a pre-defined threshold `MAX_MIGRATES`. Rule `c4` guarantees that `aggCost` after migration is below the product of the original cost `originCost` and a threshold `COST_THRES`. `originCost` can be derived by additional 5 *Colog* rules which are omitted here.

In distributed COP execution, each node only exposes limited information to their neighbors. These information includes `curVm`, `commCost`, `opCost` and `resource`, as demonstrated in rules `d2`, `d5-6` and `c2`. This leads to better autonomy for each COPE instance, since there does not exist a centralized entity which collects the information of all nodes. Via distributing its computation, *Colog* has a second advantage: by decomposing a big optimization problem (*e.g.,* VM migrations between all data centers) into multiple sub-problems (*e.g.,* VM migrations on a single link) and solving each sub-problem in a distributed fashion, it is able to achieve better scalability as the problem size grows via providing approximate solutions.

## 3.4 Execution Plan Generation

This section describes the process of generating execution plans from *Colog* programs. COPE's compiler and runtime system are implemented by integrating a distributed query processor (used in declarative networking) with an off-the-shelf constraint solver.

In our implementation, we use the RapidNet [25] declarative networking engine together with the Gecode [13] high performance constraint solver. However, the techniques describe in this section is generic and can be applied to other distributed query engines and solvers as well.

### 3.4.1 General Rule Evaluation Strategy

COPE uses a declarative networking engine for executing distributed Datalog rules, and as we shall see later in the section, for implementing solver derivation and enforcing solver constraint rules. A declarative networking engine executes distributed Datalog programs using an asynchronous evaluation strategy known as *pipelined semi-naïve* (PSN) [68] evaluation strategy. The high-level intuition here is that instead of evaluating Datalog programs in fixed rounds of iterations, one can pipeline and evaluate rules incrementally as tuples arrive at each node, until a global fixpoint is reached. To implement this evaluation strategy, COPE adopts declarative networking's execution model. Each node runs a set of local delta rules, which are implemented as a dataflow consisting of database operators for implementing the Datalog rules, and additional network operators for handling incoming and outgoing messages. All rules are executed in a continuous, long-running fashion, where rule head tuples are continuously updated (inserted or deleted) via a technique known as

*incremental view maintenance* [73] as the body predicates are updated. This avoids having to recompute a rule from scratch whenever the inputs to the rule change.

A key component of COPE is the integration of a distributed query processor and a constraint solver running at each node. At a high level, *Colog* solver rules are compiled into executable code in RapidNet and Gecode. Our compilation process maps *Colog*'s `goal`, `var`, solver derivations and constraints into equivalent COP primitives in Gecode. Whenever a solver derivation rule is executed (triggered by an update in the rule body predicates), RapidNet invokes Gecode's high-performance constraint solving modules, which adopts the standard branch-and-bound searching approach to solve the optimization while exploring the space of variables under constraints.

Gecode's solving modules are invoked by first loading in appropriate input *regular tables* from RapidNet. After executing its optimization modules, the optimization output (*i.e.,* optimization goal `goal` and variables `var`) are materialized as RapidNet tables, which may trigger reevaluation of other rules via incremental view maintenance.

### 3.4.2 Solver Rules Identification

In order to process solver rules, COPE combines the use of the basic PSN evaluation strategy with calls to the constraint solver at each node. Since these rules are treated differently from regular Datalog rules, the compiler needs to identify solver rules via a static analysis phase at compile time.

The analysis works by first identifying initial solver variables defined in `var`.

Solver attributes are then identified by analyzing each *Colog* rule, to identify attributes that are dependent on the initial solver variables (either directly or transitively). Once an attribute is identified as a solver attribute, the predicates that refer to them are identified as solver tables. Rules that involve these solver tables are hence identified as solver rules. Solver derivation and constraint rules are differentiated trivially via rule syntax (<- vs ->).

**Example.** To demonstrate this process, we consider the ACloud example in Section 3.3.2. `assign`, `hostCpu`, `hostStdevCpu`, `assignCount`, `hostMem` are identified as solver tables as follows:

- Attribute `V` in `var` is a solver attribute of table `assign`, since `V` does not appear after `forall`.
- In rule `d1`, given the boolean expression `C==V*Cpu`, `C` is identified as a solver attribute of table `hostCpu`. Hence, transitively, `C` is a solver attribute of `hostStdevCpu` in rule `d2`.
- In rule `d3`, `V` is a known solver attribute of `assign` and it appears in rule head, so `V` is a solver attribute of table `assignCount`.
- Finally, in rule `d4`, since `M` depends on `V` due to the assignment `M==V*Mem`, one can infer that `M` is a solver attribute of `hostMem`.

Once the solver tables are identified, rules `d1`-`d4` are trivially identified as solver derivation rules. Rules `c1` and `c2` are legal solver constraint rules since their rule heads `assignCount` and `hostMem` are solver tables.

In the rest of this section, we present the steps required for processing solver derivation and constraint rules. For ease of exposition, we first do not consider distributed evaluation, which we revisit in Section 3.4.5.

### 3.4.3 Solver Derivation Rules

To ensure maximum code reuse, solver derivation rules leverage the same query processing operators already in place for evaluating Datalog rules. As a result, we focus only on the differences in evaluating these rules compared to regular Datalog rules. The main difference lies in the treatment of solver attributes in selection and aggregation expressions. Since solver attribute values are undefined until the solver's optimization modules are executed, they cannot be directly evaluated simply based on existing RapidNet tables. Instead, constraints are generated from selection and aggregation expressions in these rules, and then instantiated within Gecode as general constraints for reducing the search space. COPE currently does not allow joins to occur on solver attributes, since according to our experience, there is no such use cases in practice. Furthermore, joins on solver attributes are prohibitively expensive to implement and complicate our design unnecessarily, since they require enumerating all possible values of solver variables.

**Example.** We revisit rule `d1` in the ACloud example in Section 3.3.2. The selection expression `C==V*Cpu` involves an existing solver attribute `V`. Hence, a new solver variable `C` is created within Gecode, and a binding between `C` and `V` is expressed as a Gecode constraint, which expresses the invariant that `C` has to be equal to `V*Cpu`.

Likewise, in rule `d4`, the aggregate `SUM` is computed over a solver attribute `M`. This requires the generation of a Gecode constraint that binds a new sum variable to the total of all `M` values.

### 3.4.4 Solver Constraint Rules

Unlike solver derivation rules, solver constraint rules simply impose constraints on existing solver variables, but do not derive new ones. However, the compilation

process share similarities in the treatment of selection and aggregation expressions that involve solver attributes. The main difference lies in the fact that each solver constraint rule itself results in the generation of a Gecode constraint.

**Example.** We use as example rule `c2` in Section 3.3.2 to illustrate. Since the selection expression `Mem<=M` involves solver attribute `M`, we impose a Gecode solver constraint expressing that host memory `M` should be less than or equal to the memory capacity `Mem`. This has the effect of pruning the search space when the rule is evaluated.

### 3.4.5 Distributed Solving

Finally, we describe plan generation involving *Colog* rules with location specifiers to capture distributed computations. We focus on solver derivation and constraint rules that involve distribution, and describe these modifications with respect to Sections 3.4.3 and 3.4.4.

At a high level, COPE uses RapidNet for executing distributed rules whose predicates span across multiple nodes. The basic mechanism is not unlike PSN evaluation for distributed Datalog programs [68]. Each distributed solver derivation or constraint rule (with multiple distinct location specifiers) is rewritten using a *localization* rewrite [69] step. This transformation results in rule bodies that can be executed locally, and rule heads that can be derived and sent across nodes. The beauty of this rewrite is that even if the original program expresses distributed derivations and constraints, this rewrite process will realize multiple centralized local COP operations at different nodes, and have the output of COP operations via derivations sent across nodes. This allows us to implement a distributed solver that can perform incremental and distributed constraint optimization.

**Example.** We illustrate distributed solving using the Follow-the-Sun orchestration program in Section 3.3.3. Rule `d2` is a solver derivation rule that spans across two nodes `X` and `Y`. During COP execution, `d2` retrieves rule body tables `link` and `curVm` from node `Y` to perform solver derivation. In COPE, `d2` is internally rewritten as following two rules via the localization rewrite:

```
d21 tmp(@X,Y,D,R1) <- link(@Y,X), curVm(@Y,D,R1).
d22 nborNextVm(@X,Y,D,R) <- tmp(@X,Y,D,R1), migVm(@X,Y,D,R2),
                            R==R1+R2.
```

Rule `d21` is a regular distributed Datalog rule, whose rule body is the tables with location `Y` in `d2`. Its rule head is an intermediate regular table `tmp`, which combines all the attributes from its rule body. In essence, rule `d21` results in table `tmp` generation at node `Y` and sent over the network to `X`. This rewrite is handled transparently by RapidNet's distributed query engine. Rule `d22` is a centralized solver derivation rule, which can be executed using the mechanism described in Section 3.4.3.

## 3.5 Summary

This chapter presents COPE, a platform enabling automated cloud resource orchestration, and its declarative policy language *Colog*. We have demonstrated the viability of COPE in both centralized and distributed optimization scenarios via two cloud services, where cloud resource orchestration are formulated as constraint optimization problems. We have also discussed *Colog*'s language specification, and the compilation of both centralized and distributed *Colog* programs.

# Chapter 4

# Transactional Orchestration

In this chapter we present the design and implementation of TROPIC (Transactional Resource Orchestration Platform In the Cloud) [66, 64], and how it achieves *transactional orchestration*. Residing underneath COPE, TROPIC translates high-level orchestration commands generated by COPE to low-level operations to execute. TROPIC aims to answer research questions 2–5 in Chapter 1: *robustness*, *safety*, *high concurrency*, and *high availability*, which are essentially the design goals of TROPIC.

## 4.1 TROPIC Architecture

To achieve these design goals, TROPIC platform performs *transactional* cloud resource orchestrations. Transactions provide ACID semantics which fit our design goals well: (i) *Robustness* is provided by the *atomicity* and *durability* properties, which guarantee that committed orchestrations persist on physical devices, while orchestrations that encounter unexpected errors have no effect; (ii) *Safety* is enforced by *integrity constraints* in order to achieve transactional *consistency*; (iii) *Concurrency* is supported by a concurrency control algorithm that permits multiple transactions

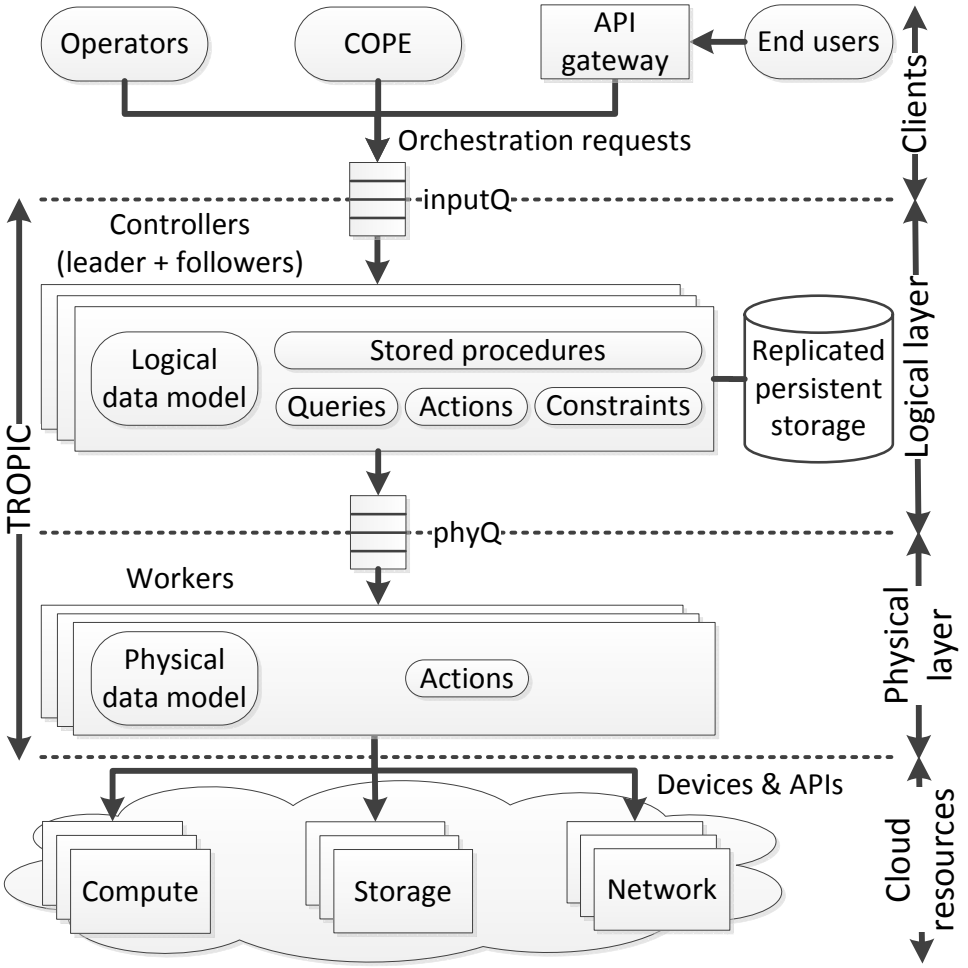


Figure 4.1: TROPIC architecture.

to execute in parallel while preserving the transactional behavior of *isolation*; (iv) *High availability* is enabled by TROPIC's adoption of a decentralized architecture of replicated components.

Figure 4.1 depicts TROPIC's architecture. The orchestration requests of clients are initiated either directly by COPE, or indirectly by cloud operators or end users via the API service gateway. Between the clients and cloud resources, TROPIC provides a two-layer orchestration stack with the *controllers* at the *logical layer* and the *workers* at the *physical layer*.

In the logical layer, the controllers provide a unified data model for representing the control states of cloud resources and a domain-specific language for implementing services. The controllers accept orchestration requests and invoke corresponding orchestration operations—*stored procedures* written in TROPIC's programming language. These stored procedures are executed as transactions with ACID semantics. In the physical layer, the workers straddles the border between the controllers and the physical devices, and provide a physical data model of devices' state. The logical data model contains a *replica* of the physical data model with weak, eventually consistent semantics.

Execution of orchestration operations in the logical layer modifies the logical data model. In the process, actions on physical devices are *simulated* in the logical layer. TROPIC guarantees safety by transitioning the logical model transactionally from one consistent state to another, only after checking that all relevant global safety constraints are satisfied. Resource conflicts are also checked to avoid race conditions. After the checks in the logical layer, corresponding physical actions are executed in the physical layer, invoking device-specific APIs to actually manipulate the devices. Transactional orchestration in both layers is described in detail in Section 4.3.

The separation of logical and physical layers is unique in TROPIC and has several benefits. First, updating physical devices' state can take a long time to complete. Simulating changes to physical devices in the logical layer is more efficient than executing the changes directly at the physical layer, especially if there are constraint violations or execution errors. Second, the separation facilitates rapid testing and debugging to explore system behavior and performance prior to deployment (Section 4.6). Third, if the logical and physical models diverge (*e.g.,* due to physical resource volatility), useful work can still be completed on consistent parts of the

data model, and in the meantime, *repair* and *reload* strategies (Section 4.4) are used to reconcile any inconsistencies.

The TROPIC architecture is carefully designed to avoid single point of failure. The components of TROPIC are connected via distributed queue services (`inputQ` and `phyQ`) that are highly available, which reduce the dependency between the components. In addition, TROPIC runs multiple controllers simultaneously on separate servers. We use a quorum-based leader election algorithm [56, 81] to ensure that there is one lead controller at any time. Other controllers are followers serving as hot standbys. In the event that the leader becomes unavailable and a new leader is being elected, workers can execute pending physical operations, and clients can also send in requests to `inputQ` without service disruptions. Critical states such as transaction state and the logical data model are shared by all TROPIC controllers through a replicated persistent storage. The details of TROPIC high availability design is given in Section 4.5.

## 4.2 Data Model and Language

As illustrated in Figure 4.2, TROPIC adopts a hierarchical data model, in which resources are organized into a tree-like structure. We use a semi-structured data model because it handles heterogeneity of cloud resources well. Each tree node is an object representing an instance of an entity. An entity may have multiple *attributes* of primitive types, and multiple one-to-many and one-to-one *relations* to other entities, represented by children nodes. An entity has a primary key that uniquely identifies an object among its sibling objects in the tree.

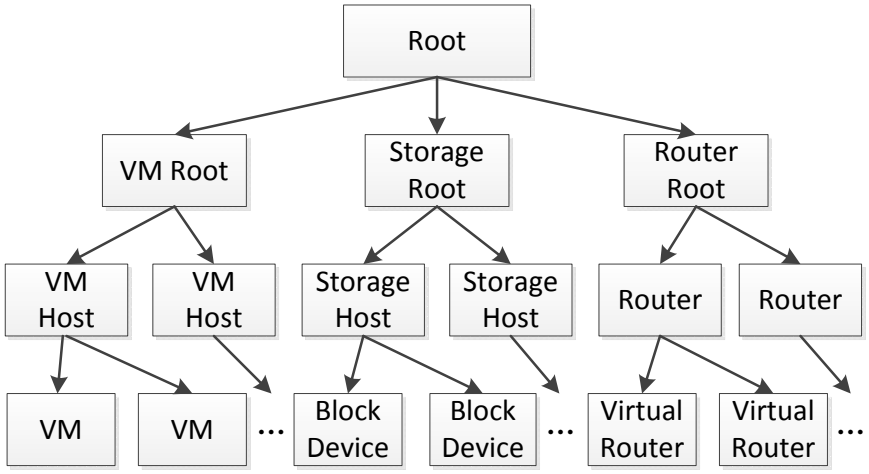


Figure 4.2: An example of TROPIC data model.

Figure 4.3 shows a TROPIC code example that contains the definitions of objects in the data model of Figure 4.2. Although incomplete, the code is very close to the real code in TROPIC. In the example, each entity instance is defined as a class object, and children instances are referenced as one attribute in the object. For example, `Root` denotes the root of the tree, and it has `vmRoot`, `storageRoot` as children. `vmRoot` itself has multiple `VMHost` children, which are comprised of a compute server and several guest VMs (`VM`). Note that not all TROPIC data models are isomorphic to Figure 4.2. In practice, a cloud operator could flexibly define a different hierarchy, for example, by grouping `VMHost` under the entity of `Rack`, which is then grouped under `Pod`, `Availability Zone`, etc.

Each entity has associated expressions and procedures for inspecting and modifying the entity: queries (`@query`), actions (`@action`), constraints (`@constraint`), and stored procedures (`@proc`).

**Definition 1.** *A* **query** *inspects system state in the logical layer and provides* read-only *access to resources.*

```
class VM(LogicalModel):
    name = Attribute(str)
    state = Attribute(VMState)
    mem = Attribute(int)
    @primaryKey # the primary key is attribute 'name'
    def id(self): return self.name
...
class VMHost(LogicalModel):
    name = Attribute(str)
    mem = Attribute(int)
    vms = Many(VM)
    @query
    def allVMs(self):
        for vm in self.vms:
            yield (self, vm.name, vm.state)
    @constraint
    def memLimit(self):
        if sum(vm.mem for vm in self.vms) >= self.mem:
            yield ("not enough physical memory", self)
    @action
    def startVM(self, ctxt, name):
        self.vms[name].state = VMState.On
        ctxt.appendlog(action="startVM", args=[name],
            undo_action="stopVM", undo_args=[name])
...
class VMRoot(LogicalModel):
    hosts = Many(VMHost)
...
class Root(LogicalModel):
    vmRoot = One(VMRoot)
    storageRoot = One(StorageRoot)
...
@proc
def spawnVM(root, vmName, hostName, imageTemplate):
    storageHostName = root.vmRoot.hosts[hostName].vms[vmName].storageHostName
    vmImage = vmName + "_img"
    root.storageRoot.hosts[storageHostName].cloneImage(imageTemplate, vmImage)
    root.storageRoot.hosts[storageHostName].exportImage(vmImage)
    root.vmRoot.hosts[hostName].importImage(vmImage)
    root.vmRoot.hosts[hostName].createVM(vmName, vmImage)
    root.vmRoot.hosts[hostName].startVM(vmName)
```

Figure 4.3: TROPIC code example.

**Example 4.2.1.** *Lines 13–15 define the query* `allVMs`*, which returns the names and states of all VMs defined on a compute server.*

**Definition 2.** *An* **action** *models an* atomic *state transition of a resource.*

Actions generalize the myriad APIs, ranging from file-based configurations, command-line interfaces (CLIs), to RPC-style APIs, provided by vendors to control physical resources. Each action is defined twice: in the physical layer, the action implements the state transition by calling the device's API, and in the logical layer, the action simulates the state transition on the logical data model. Preferably, an action is associated with a corresponding *undo* action. Undo actions are used to roll back a transaction (Section 4.3.1).

**Example 4.2.2.** *Lines 20–22 define the action* `startVM`*, which boots a VM. Lines 23–24 define its corresponding undo action* `stopVM`*. The undo action is recorded to the log within the execution context at runtime (Section 4.3.1).*

**Definition 3.** *A* **constraint** *in TROPIC specifies service and engineering rules.*

TROPIC constraint is similar to database integrity constraints [78]. A constraint is satisfied if and only if it evaluates to an empty list. Otherwise, the list contains messages to help pinpoint the cause of the violation. Constraints support the *safety* property, and TROPIC automatically enforces them at runtime.

**Example 4.2.3.** *The constraint on lines 16–19 specifies that the total memory of guest VMs on a physical host cannot exceed the host's memory limit.*

**Definition 4.** *A* **stored procedure** *is composed of a series of queries, actions and other stored procedures to orchestrate cloud resources.*

**Example 4.2.4.** *For instance, lines 33–41 define a stored procedure that spawns a VM, which consists of five sequential actions*[1]*: cloning the VM storage (*`cloneImage`*) from an image template, exporting the image (*`exportImage`*), importing the image (*`importImage`*), provisioning the VM on the target host (*`createVM`*), and then starting the newly created VM (*`startVM`*).*

**Definition 5.** *Stored procedures specify orchestration logic, and they are executed as* **transactions** *that enforce ACID properties.*

Table 4.1 lists supported actions and their parameters in TROPIC, divided by resource categories (*i.e.,* compute, storage, and network). A valid TROPIC transaction consists of zero or more actions only listed in Table 4.1. In complement to Table 4.1 and Figure 4.3, Appendix B gives more TROPIC code examples. TROPIC is a general IaaS cloud orchestration platform, and cloud operators and service providers can flexibly add more actions, queries and constraints upon needs. We note that TROPIC can only manipulate cloud states which are exposed to it. TROPIC does not handle the states such as VM's internal application states, including network connections and software updates.

To illustrate how orchestration commands generated by *Colog* in COPE are translated to TROPIC transactions, we use the ACloud scenario in Section 3.3.2. After constraint solving in COPE, rule `d7` stores materialized table `migrateVM(Vid, Hid1, Hid2)` that denotes vm `Vid` should be migrated from source host `Hid1` to destination host `Hid2`. `migrateVM(Vid, Hid1, Hid2)` tuples are easily mapped to `migrateVM` procedure (which is a single-action procedure listed in Table 4.1) in TROPIC. Specifically, the procedure's parameter `vmName` is tuple's attribute `Vid`, `srcHost` is `Hid1`, and `dstHost` is `Hid2`.

[1]Network operations, such as VLAN and firewall configurations, are omitted for brevity.

| Resource | Action | Parameter |
|---|---|---|
| Compute (VM) | pause | - |
| | unpause | - |
| | startVM | vmName |
| | stopVM | vmName |
| | createVM | vmName, vmImage |
| | removeVM | vmName, vmImage |
| | saveVM | vmName, path |
| | restoreVM | vmName, path |
| | migrateVM | vmName, srcHost, dstHost |
| | importImage | vmImage |
| | unimportImage | vmImage |
| Storage | exportImage | imageName |
| | unexportImage | imageName |
| | cloneImage | templateImageName, imageName |
| | deleteImage | imageName |
| Network (router) | addBgpRoute | netAddr, nextHop |
| | delBgpRoute | netAddr, nextHop |

Table 4.1: Actions supported by TROPIC.

In general, the output of *Colog* is one or multiple materialized tables. The tuples of these tables are one-to-one mapped into TROPIC transactions in the way that each tuple's *predicate* becomes a *procedure*, and *attributes* become the *parameters* of the procedure. COPE is restricted to invoking TROPIC procedures as APIs. If TROPIC detects an undefined procedure name as input or mismatched parameters, it immediately raises an error to indicate invalid input. In TROPIC, transactional orchestration is expressed in Python syntax, which is Turing-complete in terms of language expressiveness. As long as TROPIC procedures defined in Python syntax are safe, *Colog* output is safe. Therefore, *Colog* is deemed to be safer than raw Python code.

## 4.3 Design

In this section, we describe TROPIC's transaction execution model, and explain how TROPIC can meet our design goals of safety, concurrency, and robustness, through the enforcement of ACID properties in orchestration operations. Specifically, TROPIC makes the following guarantee: if the logical and physical layers are consistent at the beginning of each transaction, ACID properties can always be enforced in the logical layer. Furthermore, in the absence of cross-layer inconsistency caused by resource volatility, these properties are also enforced in the physical layer. We defer the discussion of inconsistency between the logical and physical layers to Section 4.4, and focus on transaction processing here.

We first describe a typical life cycle of a transactional orchestration operation, followed by the execution details in the logical and physical layers. Figure 4.4 depicts the typical steps in executing a transaction $t$, from the initial request submitted by a client until $t$ is committed or aborted.

**Step 1: initialization.** A client issues a transactional orchestration as a call to a stored procedure. The transaction is *initialized* and enqueued to `inputQ`.

**Step 2: acceptance.** The controller (leader) accepts $t$ by dequeuing it from `inputQ` and enqueues it to `todoQ`.

**Step 3: logical execution.** The controller is responsible for scheduling accepted transactions, making sure there is no constraint violation or possible race condition, and generating the execution logs for future undo and physical layer execution. All these steps happen in the logical layer and are explained in Section 4.3.1.

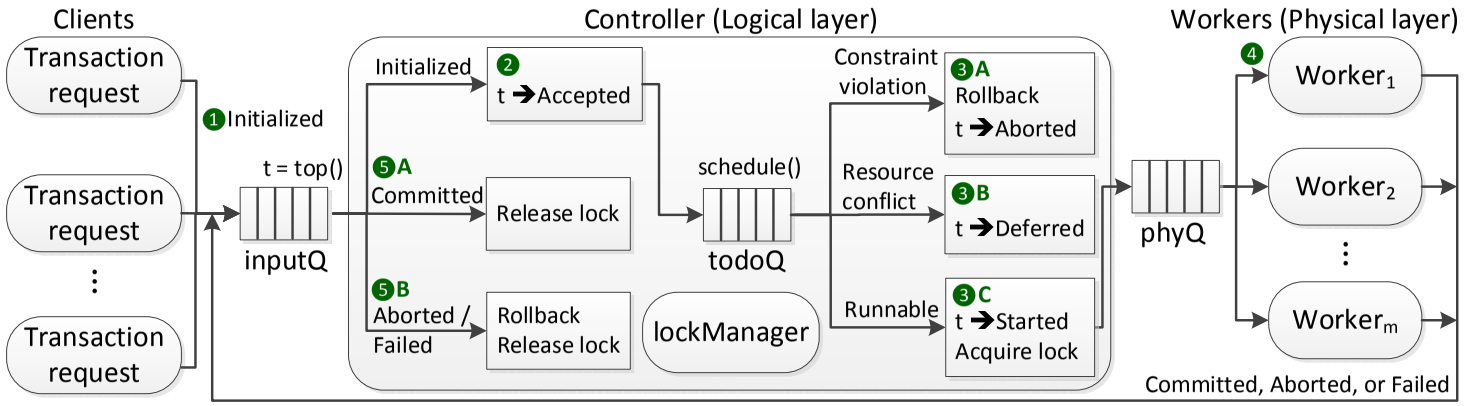


Figure 4.4: The execution flow of transactional orchestration in TROPIC.



| log | resource object path | action | args | undo action | undo args |
|---|---|---|---|---|---|
| 1 | /storageRoot/storageHost | cloneImage | [imageTemplate, vmImage] | deleteImage | [vmImage] |
| 2 | /storageRoot/storageHost | exportImage | [vmImage] | unexportImage | [vmImage] |
| 3 | /vmRoot/vmHost | importImage | [vmImage] | unimportImage | [vmImage] |
| 4 | /vmRoot/vmHost | createVM | [vmName, vmImage] | removeVM | [vmName, vmImage] |
| 5 | /vmRoot/vmHost | startVM | [vmName] | stopVM | [vmName] |

Table 4.2: An example of execution log for `spawnVM`.

**Step 4: physical execution.** Any transaction that has gone through the controller is dequeued from `phyQ` and executed in the physical layer by the physical workers (Section 4.3.2). The execution result (*e.g.,* committed or aborted) is enqueued to `inputQ` to notify the controller.

**Step 5: cleanup.** The controller examines the execution result received from the workers. If it is successful, the transaction state is marked as *committed* and the locks held by the transaction are released (**5A**). Otherwise, if the transaction fails in Step 4, it is marked as *aborted*. The controller then rolls back the logical layer and releases corresponding locks (**5B**).

### 4.3.1 Logical Layer Execution

The logical layer execution logic is depicted as Steps **3A**–**3C** in Figure 4.4. When a transaction $t$ is scheduled to execute (`schedule()` in the figure), it is first dequeued from `todoQ`. The controller decides $t$ is runnable, if and only if: (i) It does not violate any safety constraints, and (ii) It does not access or modify resources that are being used by outstanding transactions (race conditions). If there is a safety violation, $t$ is marked as *aborted* and the controller rolls back the logical layer state (**3A**). If there is a resource conflict, $t$ is put back into the front of `todoQ` for subsequent retry (**3B**). Otherwise, $t$ is runnable. The controller acquires the locks on related resources, and the transaction state is changed to *started* before $t$ is enqueued into `phyQ` (**3C**).

#### Scheduling

In executing the `schedule()` operation, TROPIC adopts a FIFO queue `todoQ` for fairness and simplicity. It dequeues and schedules a new transaction whenever one of the following conditions is met: (i) A transaction is inserted into an empty `todoQ`;

(ii) A transaction is aborted from its logical execution due to a constraint violation; (iii) A transaction finishes its physical execution (either committed or aborted); (iv) A transaction has been identified as runnable and is sent to `phyQ`. More sophisticated scheduling policies are possible (*e.g.,* an aggressive strategy of scheduling transactions queuing behind the one with conflicts). We leave a detailed study of alternative scheduling policies as future work.

### Simulation

Once scheduled, instead of directly executing on the physical resources, a simulation step in the logical layer is used to analyze the transaction for possible constraint violations and infer the resources it reads and writes (*i.e.,* queries and actions in Section 4.2 respectively) for concurrency control. This provides early detection of unsafe operations without touching actual physical resources. Table 4.2 shows an example transaction for spawning a VM (`spawnVM` in Figure 4.3). The transaction consists of 5 actions, which are recorded in an execution log for use in subsequent phases. In simulation, every action within the transaction is applied sequentially, and whenever an action results in a constraint violation, the transaction is aborted. Modifications to the logical layer are rolled back via the *undo* actions in the execution log.

### Concurrency Control

TROPIC adopts a pessimistic concurrency-control algorithm based on multi-granularity locking [78]. A lock manager keeps track of the locks acquired by each transaction and detects possible conflicts. New transactions are allowed to run only if their required locks do not conflict with existing locks used by outstanding transactions.

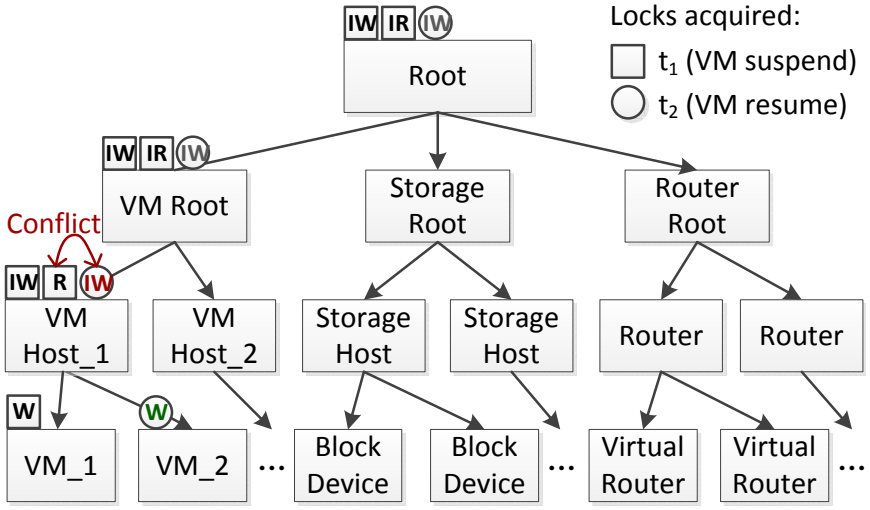


Figure 4.5: An example of lock-based concurrency control. The locks acquired by transactions $t_1$ (VM suspend) and $t_2$ (VM resume) are in squares and circles respectively.

During its execution, a transaction $t$ acquires write (read) locks on resource objects used by individual actions (queries). For instance, in table 4.2, write locks are acquired for each object identified by its resource path. Once these objects and their corresponding lock types are identified, the lock manager acquires read (R) or write (W) locks on the actual object, and *intention locks* (IR/IW) [2] on the ancestors of this object.

Besides acquiring locks on the resources used by transactions, additional locks are also acquired based on the constraints that impact transactions. When a write operation is performed on an object, we find its highest ancestor that has constraints defined and acquire an R lock on the node. As a result, all its descendants are read-only to other concurrent transactions, hence preventing others from making state changes that potentially break safety.

[2]Intention locks are commonly used for managing concurrency in hierarchical data structures. They summarize the locking status of descendant nodes, and allow conflicts to be detected higher up the tree. IW locks conflict with R/W locks, while IR locks conflict with W locks.

For instance, Figure 4.5 shows the locks acquired by two transactions $t_1$ and $t_2$. $t_1$ suspends a VM (`VM_1`) and $t_2$ resumes another VM (`VM_2`) on the same host. A constraint is defined to limit the physical memory utilization of each host (`memLimit` in Figure 4.3). If $t_1$ and $t_2$ were allowed to run concurrently, the abort of $t_1$ and commit of $t_2$ may violate this constraint by exceeding the memory limit of `VMHost_1`.

The locking algorithm prevents this from happening. Assuming $t_1$ is scheduled first, it acquires a W lock on `VM_1`, followed by the IW locks on its ancestors `VMHost_1`, `VMRoot` and `Root`. Due to the `memLimit` constraint, an R lock is also acquired on `VMHost_1`. When $t_2$ is scheduled, it first acquires a W lock on `VM_2`. However, when it tries to acquire subsequent IW locks on its ancestors, there is a R/IW conflict detected on `VMHost_1`. Therefore, $t_2$ is deferred until $t_1$ finishes.

TROPIC controller runs only one thread in the logical layer for transaction scheduling, simulation, and concurrency control that adopts a conservative way of locking. For each transaction $t$, TROPIC lock manager acquires all its locks before it is dispatched to the physical layer. After $t$ is committed or aborted, the locks of $t$ are removed at once in the lock manager. This is similar to strict two phase locking [78] in traditional database, *i.e.,* lock acquiring is the first phase and lock releasing is the second phase. Therefore, we observe no deadlock in TROPIC. Moreover, we note that cloud resource orchestration is different from traditional online transaction processing (OLTP) workload, *i.e.,* millions of transactions with critical performance requirement. As we will demonstrate in the evaluation (Section 5.1), TROPIC has good performance in processing intensive realistic cloud workload and the logical-layer locking overhead is negligible compared to physical resource operations.

### 4.3.2 Physical Layer Execution

Once a transaction $t$ is successfully executed in the logical layer, it is ready for actual execution in the physical layer. $t$ is stored in `phyQ` and dequeued by one of the physical workers in Step 4. Executing $t$ in the physical layer involves replaying the execution log generated in the logical layer simulation. If all the physical actions succeed, $t$ is returned as committed. If any action fails, the worker selects the actions that have been successfully executed, identifies the corresponding undo actions, and executes them in reverse chronological order.

To guarantee atomicity of transactions, each action in a transaction must have a corresponding undo action. In our experience, most actions, such as resource allocation and configuration are reversible. Once all undo actions complete, the transaction is returned as aborted. Using the execution log in Table 4.2 as example, suppose the first four actions succeed, but the fifth one fails. TROPIC reversely executes the undo actions in the log, *i.e.,* record #4, #3, #2 and #1, to roll back the transaction. As a result, the VM configuration and cloned VM image are removed.

If an error occurs during undo in physical execution [3], the transaction is returned as *failed*. The logical layer is still rolled back. However, failures during undo may result in cross-layer inconsistencies between the physical and logical layers.

## 4.4 Handling Resource Volatility

In cloud environments, unexpected software and hardware errors (*e.g.,* power glitches, unresponsive servers, misconfigurations, out-of-band access) may occur. We explore

[3] We choose to stop executing undo actions in the physical layer once an undo action reports an error, because they might have temporal dependencies.

mechanisms in TROPIC for dealing with this volatility of resources during transaction execution. TROPIC does not attempt to transparently tolerate failures of the volatile cloud resources. Instead, it makes the best effort to maintain consistency between the logical and the physical layer, by using two reconciliation mechanisms that achieve eventual consistency. In the event of resource failures, TROPIC provides feedback to the cloud operator, in the form of transaction aborts and timeouts, and recovery is handled at higher layers, in accordance with the end-to-end argument [86].

### 4.4.1 Cross-layer Consistency Maintenance

In order for a transaction to execute correctly, the logical layer needs to reflect the latest state of the physical layer. However, achieving cross-layer consistency at all times is improbable given the volatility of cloud resources. To illustrate, consider three scenarios in which inconsistencies occur: (i) During the physical layer execution, an error triggers the rollback procedure, and the execution of an undo action fails. The transaction is terminated as failed, with the logical layer fully rolled back and the physical layer partially rolled back; (ii) An intentional out-of-band change is made to a physical device. For example, an operator may add or decommission a physical resource, or she may log in to a device directly and change its state via the CLI without using TROPIC; (iii) An unintentional crash or system malfunction changes the resource's physical state beyond TROPIC's knowledge. At the scale of large data centers, these events are the norm rather than the exception, and TROPIC must be able to gracefully handle the resulting inconsistencies.

TROPIC adopts an *eventual consistency* model for reconciliation, which allows the two layers to go out of sync in between reconciliation operations. Inconsistency

can be automatically identified when a physical action fails in a transaction, or can be detected by periodically comparing the data between the two layers. Once an inconsistency is detected on a node in the data model tree, the node and its descendants are marked *inconsistent* to deny further transactions until the inconsistency is reconciled. Any transactions involving inconsistent data are also aborted with rollback.

The two mechanisms for reconciliation are as follows:

**Physical to logical synchronization (reload).** States of specified devices are first retrieved from the physical layer and then used to replace the current ones in the logical layer. Similar to normal transaction execution, the controller ensures `reload` is concurrently executed with outstanding transactions while not violating any constraints. If any constraints are violated, `reload` is aborted.

**Logical to physical synchronization (repair).** Physical states of devices are also first retrieved. TROPIC then compares the two set of states in the logical and physical layers, and performs corresponding pre-defined actions to repair physical devices. For instance, suppose a compute server is unexpectedly rebooted, resulting in all its running VMs being powered off. By comparing the VM states in two layers — one "running" and the other "stopped", `repair` will execute multiple `startVM` actions to start the powered-off VMs. After `repair` the logical layer is intact and hence no constraint violation should be found in this process.

In the event that `reload` and `repair` operations do not succeed due to hardware failures, the failed resources are marked as *unusable*, and future transactions are prevented from using them.

Given that `repair` and `reload` operations are expensive, we do not run them at the beginning of each transaction. Instead, `reload` is called when devices are

added to or decommissioned from TROPIC, and `repair` can be issued in an event-driven style when there is cross-layer inconsistency. Often in practice, cross-layer inconsistency is observed when there are aborted transactions. For example, suppose in the logical layer a VM is running, but in the physical layer the VM is somehow powered off. In this case, suppose a transaction $t$ is issued to stop the VM. $t$ will succeed in the logical layer but fail in the physical layer, finally ended as aborted. When the transaction issuer sees the error message, she can conclude that the state of this VM differs in the logical and physical layers. Then she can issue a `repair` on the affected VM to bring it back to the running state. Alternatively, `repair` can be issued periodically (on either the whole resource tree or specified subtrees), and the frequency is customizable based on cloud operators' preference or failure rates of underlying resources (the higher the rate, the shorter the repairing cycle).

TROPIC can repair cases including VM crash, host machine reboot, and transient network disconnection (which resets the resource roles of replicated storage). Cloud operators and service providers can add more based on demand.

### 4.4.2 Terminating Stalled Transactions

Another source of error induced by resource volatility is the indefinite stalling of a transaction, caused by transient network disconnection, power outage, hardware failure, etc. For example, when a VM is being migrated from source to destination host, the underlying network gets disconnected during half-way. This prevents a orchestration operation from completing (either to a committed, aborted, or failed state) within a bounded period of time.

To handle unresponsive transactions, TROPIC provides clients two mechanisms to end them, by sending either *TERM* or *KILL* signals[4].

[4]Analogous to `SIGTERM` and `SIGKILL` signals to a POSIX-compliant process.

When the controller receives a *TERM* signal for one of its outstanding transactions, it forwards the signal to the corresponding physical worker, which aborts this transaction and rolls back its actions. In this process, graceful cleanups at both the logical and physical layer (*e.g.,* undo actions, lock releasing) are performed so that cross-layer consistency is maintained.

If a transaction is stuck in an uninterruptable physical action and does react to *TERM*, sending a *KILL* signal makes the controller always immediately aborts the transaction, but only in the logical layer. The signal is forwarded to the physical worker as well, but the controller does not wait for it to respond. Once the worker receives the signal, it simply stops executing (without undo). Any resulting cross-layer inconsistencies are then reconciled using the strategies in Section 4.4.1. *KILL* resembles the case where a transaction is *failed* in physical layer execution during undo.

### 4.4.3 Transactional Semantics

In the presence of *volatile* cloud resources, TROPIC maintains a *weakly* and *eventually consistent* logical / physical layer and provides following transactional semantics.

First, TROPIC makes the guarantee that if the logical and physical layers are consistent at the beginning of each transaction, ACID properties can always be enforced in the logical layer. Furthermore, in the absence of cross-layer inconsistency caused by resource volatility, these properties are also enforced in the physical layer.

Second, TROPIC guarantees that the successful completion (*committed*) or abort (*aborted*) of a transaction should preserve the logical-physical layer consistency. During the execution of a transaction, if resource volatility can not be elegantly handled

via rollback, TROPIC can send a *TERM* or *KILL* signal to stop a hanging transaction and later invoke `repair` to reconcile any cross-layer inconsistencies.

After a transaction is finished (either committed or aborted), if *new* resource volatility is introduced, it may result in cross-layer inconsistency. In this case, cloud operators can invoke `repair` to reconcile the inconsistency. If a `repair` transaction unfortunately fails, cloud operators can mark the resource as *unusable*, and need to manually resolve related constraint violations.

## 4.5 High Availability

As highlighted in Section 4.1, all the TROPIC components in Figure 2.1, including the distributed queues (`inputQ` and `phyQ`), the persistent storage service, the controllers and the workers, are architected with redundancy to avoid single point of failure. We adopt ZooKeeper [52] to implement the queues and the storage service with high availability (Section 4.6). We run multiple identical worker instances. The failure of a worker with no active transactions does not affect TROPIC at all. If a worker with active transactions fails, its transactions can be terminated with *KILL* (in a similar fashion whenever resources fail, as described in Section 4.4). In the rest of the section, we describe how we design the controllers to provide high availability while maintaining the transactional semantics.

### 4.5.1 Controller State Management

TROPIC runs multiple controller instances. One of them is the leader, and the rest are followers. Only the leader serves transaction executions in the logical layer. When it fails, the followers among themselves elect a new leader, which then resumes execution after restoring the most recent state of the previous leader. Our design

has the following assumptions: (i) Controllers may crash at any time (*i.e.,* fail-stop) or be subject to network partitions, which are common in a large-scale data center. However, they do not suffer arbitrary Byzantine faults [57], because they are operated in a single trusted administrative domain; (ii) The storage service offers *atomic* key-value pair updates. It maintains sufficient state to allow the new leader to resume execution.

TROPIC controllers only maintain state in local memory as a cached copy for performance reasons and can be safely discarded without impacting the correctness of transaction execution. Whenever the lead controller fails, the new leader elected among the followers has to be able to restore the state of the controller at failure time, by retrieving data from persistent storage. Since the I/O performance in the replicated persistent storage is orders of magnitude worse than in memory, we aim to limit the amount of persistent data to only those required for subsequent failure recovery. Specifically, we maintain the following key data structures in persistent storage:

1. **`txns`** is a hash table indexed by transaction ID (assigned to each transaction as a monotonically increasing number when it is accepted by the controller). Each transaction $t$ has an entry which maintains its parameters and current state throughout its life cycle, *e.g.,* from *accepted* to *started*, followed by *committed* or *aborted* after execution. In Figure 4.5, `txns.set(t,s)` sets the state of transaction $t$ to $s$.

2. **`logs`** is a journal that records all actions of *committed* transactions. Whenever a transaction is committed, the actions on its execution log are atomically appended into `logs` indexed by a monotonically increasing log ID. In Figure 4.6, `logs.append(t)` stores the actions of transaction $t$ in `logs`.

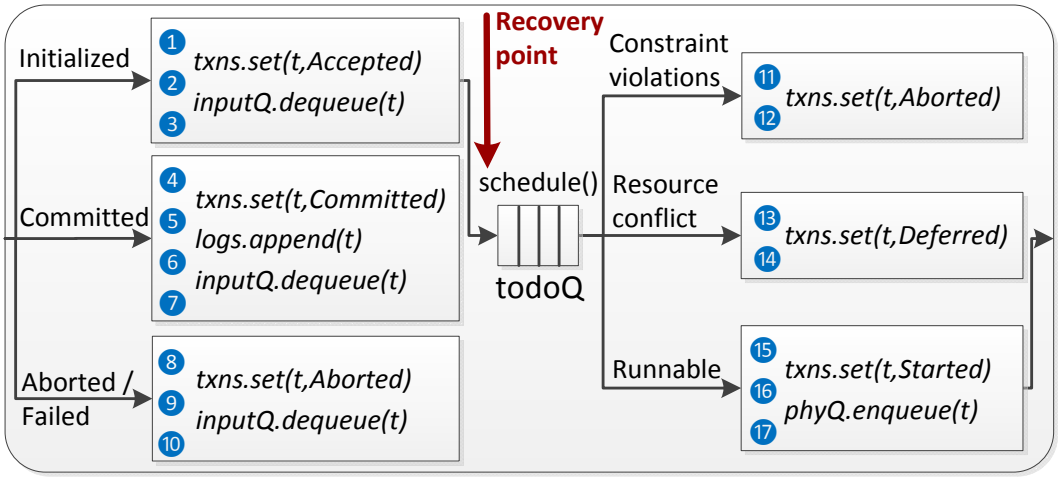


Figure 4.6: The design of TROPIC controller for high availability, from the perspective of a transaction $t$. Circled numbers denote failure points (FP), and italic texts denote data writes to persistent storage and distributed queues. As indicated from Figure 4.4, the controller takes input from `inputQ` and feeds `phyQ`.

Note that in addition to `txns` and `logs`, after the leader crashes, the new leader still has access to the `inputQ` and `phyQ` queues, both of which are persistently stored as highly available distributed queues (Section 4.6).

At recovery time, the new controller can rebuild the internal data structures (such as its `todoQ` and locks) from accepted and started transactions in `txns`, and reconstruct the logical layer data by replaying the actions from `logs`. As an optimization, snapshots can be periodically made to shorten the time of log replay.

### 4.5.2 Controller Failure Recovery

To understand how controller failure recovery works, we consider the scenario where the lead controller $l$ has failed, and a follower $f$ has been elected as replacement for $l$. Because all controller internal data is recoverable from persistent storage, only data *write* to persistent storage and distributed queues affects the the logic of failure recovery. These data writes include updating transaction state in `txns`, appending

the action log of a committed transaction to `logs`, dequeuing and enqueuing of `inputQ` and `phyQ`, as shown in Figure 4.6. Because the storage service offers atomic data write, the problem of recovering failure at any time is reduced to recovering from failures at all possible points before and after each data write, denoted as the 17 *failure points* (FP) numbered in Figure 4.6.

There are two recovery strategies. First, if an FP can be identified at $f$ by examining the states in `txns` and `logs`, then it can be directly recovered by resuming execution from the FP. For instance, in FP `5`, the front of `inputQ` is a transaction $t$ in *committed* state (as retrieved from `txns`). To resume execution, the actions of $t$ are appended to `logs` and $t$ is dequeued from `inputQ`. FP `2`, `6` and `9` are in the same category with `5`. Note that we have carefully designed the write order to make these FPs identifiable. *E.g.,* if we were to dequeue `inputQ` before FP `5`, then it would not have been identified and recoverable.

Second, if an FP cannot be identified, $f$ always resumes from the default *recovery point* (shown in the straight down arrow in Figure 4.6) right before `schedule()`, which corresponds to the scheduling step in Section 4.3.1. This strategy leverages the *idempotent* nature of the steps that occur between the recovery point and the FP. We briefly describe why these steps are idempotent as follows:

1. The immediate next step of FP `3`, `7` and `10` is `schedule()`, which is their natural recovery point.

2. For FP `1`, `4` and `8`, since the corresponding transaction $t$ has not been dequeued from the front of `inputQ`, by starting from `schedule()`, the controller will eventually process the front of `inputQ` again.

3. FP `11`, `13` and `15` repeat the logical simulation of the transaction $t$ at the front of `todoQ`. While this results in extra work already performed by $l$, it does not

affect the correctness of $f$. For instance, in FP `11`, resuming at the recovery point would simply repeat the constraint violation of $t$.

4. In FP `17`, the transaction is slated for execution in the physical layer, and thus has no impact on controller's internal state. Resuming at `schedule()` simply selects the next transaction in `todoQ` to be simulated in the logical layer. This similarly applies to FP `12` and `14`. FP `16` is treated as a special case of FP `17`, and it results in a transaction not enqueued in `phyQ`. This is deemed as a stalled transaction and hence is removed using the *KILL* signal (Section 4.4.2). Note that after *KILL* no cross-layer inconsistency will be found since no actual physical action was ever performed.

## 4.6 Implementation

We have implemented a prototype of TROPIC. We briefly describe some of our implementation choices and outline our experiences in developing two cloud services based on top of TROPIC.

### 4.6.1 Development and Deployment

**Language choice.** We chose Python as our implementation language and the prototype of TROPIC is implemented in 11K lines of code. Python has rich libraries and a large community. Its syntax is highly readable and provides multiple high-level language features (*e.g.,* meta programming, function decoration, list comprehension) that make it easy for us to embed TROPIC's domain-specific language (Section 4.2) inside. As demonstrated in the example code in Figure 4.3, Python enables succinct

and expressive syntax to define resource models and orchestration logic, while hiding the complexity of transaction processing behind the scene.

**Client APIs.** Once resources and orchestration logic are modeled in the language, a client can issue orchestration commands using the TROPIC client library. The library provides four basic APIs: `submit`, `inquire`, `term` and `kill`. Clients use `submit` to submit orchestration requests into `inputQ` to invoke stored procedures. It can be executed in either synchronous mode (block until the transaction finishes) or asynchronous mode (return immediately with a transaction ID for inquiring results later via `inquire`). If a transaction is stalled for too long, the client can try to terminate it with transaction rollback via `term`, or directly kill it via `kill` (Section 4.4.2). We have also developed an interactive command-line shell and a visualization tool.

**Testing and debugging.** TROPIC offers a *logical-only mode* to simplify testing and debugging. In this mode, we bypass the physical resource API calls in the workers, and instead focus on various scenarios in the logical layer execution. In this mode, we can easily plug in arbitrary configurable resource types and quantities to study their possible impact on TROPIC. Our experiments in Section 5.1 heavily use the logical-only mode to explore TROPIC performance under large scale of diverse cloud resources.

### 4.6.2 Coordination and Persistent Storage

We use ZooKeeper [52] as the distributed coordinator to implement leader election and distributed queues (`inputQ` and `phyQ`). ZooKeeper provides highly available coordination services to large-scale distributed systems. These services are usually easier to realize with ZooKeeper APIs than conventional ways [56].

It is worth noting that we have made a few optimizations to the queue implementation for `inputQ`. By default, queue producers insert new items with sequentially increasing IDs as children of a ZooKeeper node (zknode). The queue consumer registers a watch on the zknode to get notifications when new items are enqueued, and uses the `getChildren` API to get the new items. We found that `getChildren` has the I/O overhead linear to the queue length, which is costly when the queue is long. Hence the first optimization in TROPIC is that, when the queue is long, we automatically switch to a *polling mode* where the new items of the queue are probed with the `get` API based on guessing the item IDs. This incurs constant overhead and we observe up to twice the throughput performance under heavy bursty load. Second, to improve transaction latency, `inputQ` is configured as a priority queue, where committed and aborted transactions are prioritized over new client requests.

We also unconventionally use ZooKeeper as a highly available persistent storage engine for storing `txns` and `logs` (Section 4.5). In theory, any replicated SQL databases or key-value stores should work. However, since we already use ZooKeeper for coordination, not introducing another software component reduces operation burdens. ZooKeeper's limitation as a storage engine is that the data size is bounded by the main memory size, which is not a problem for our workload (Section 5.1).

### 4.6.3 Case Study: ACloud

Using TROPIC we have developed the *ACloud* cloud service described in Section 3.2.1. ACloud is deployed in a single data center and has features similar to Amazon EC2. It allows clients to spawn new VMs from disk images, and start, stop, and destroy these VMs. In addition, the operator can migrate VMs between hosts to balance or consolidate workloads.

The data center provides storage servers that export block devices via the network, compute servers that allocate VMs, and a programmable switch layer with VLAN features. Specifically, we use GNBD [15] and DRBD [82] over the Linux logical volume manager (LVM) as storage resources, Xen [36] as compute resources, and Juniper routers as network resources.

These three classes of resources provide very different APIs for orchestration. GNBD and DRBD rely on text-based configuration files and CLIs to update resource roles and other state in the kernel. Xen provides its own APIs, but is also compatible with a generic set of virtualization APIs from `libvirt` [19], a configuration toolkit that works with a variety of virtualization technologies. The process of building data models for GNBD, DRBD and `libvirt` on Xen is entirely manual, requiring user effort to define entities and relationships, and wrapping their API calls to actions in TROPIC. In contrast, since Juniper routers use the XML-based NETCONF protocol [20] for configuration, we are able to automatically import the XML scheme into TROPIC's tree model. The only remaining work is to develop router actions (*e.g.,* configuration commit) and constraints (*e.g.,* network protocol dependencies).

We note that ACloud is developed with minimal effort. Typical VM-related operations (*e.g.,* migrate, clone, start/stop) and constraints require only a handful of lines of code. The data models themselves are also constructed in a straightforward manner, requiring us to wrap device-specific API calls into actions, and importing their configurations into the tree model.

In developing ACloud, we have learned a few lessons: (i) Based on our concurrency control algorithm design (Section 4.3.1), imposing constraints high up in the tree (*e.g.,* in the extreme case, on the root node) is undesirable because it reduces concurrency. Our experience shows that there is no compelling use case in ACloud

for such constraints. The constraints either fit at the low level naturally, or can be moved down to lower levels via data model rearrangements to improve concurrency; (ii) Not all orchestration procedures need to be executed in an atomic manner. For instance, we once tried to perform maintenance on a server by migrating all its VMs away at once. When some of the migrations fail, we observe that it is unnecessary to move the migrated VMs back. Instead, we decomposed the stored procedure into smaller transactions that migrate only one VM each and submitted them separately to TROPIC. We plan to add the feature of allowing a procedure to contain a sequence of TROPIC transactions in future work.

### 4.6.4 Case Study: Follow-the-Sun

Figure 4.7 visualizes another cloud service named Follow-the-Sun (Section 3.2.2) we have realized in TROPIC. Follow-the-Sun involves a multi-data center, cross-domain cloud orchestration scenario. In Follow-the-Sun [93], VMs are live migrated over wide area network (WAN) across geographically dispersed data centers to be closer to where work is being performed. During VM migration, the IP address of the VM does not change, so existing application-level sessions are not disrupted. This involves the orchestration of various cloud resources, *i.e.,* compute, storage and network. Specifically, a layer-2 VPN is first established between migration source and destination data centers, and the storage (we use DRBD) associated with the VM is replicated. Then the VM itself is migrated over and the network router updates by advertising a more specific BGP route to the migrate VM from the destination data center.

We emulate the Follow-the-Sun cloud environment that TROPIC controls and orchestrates on ShadowNet [41], our operational wide-area testbed. In particular we

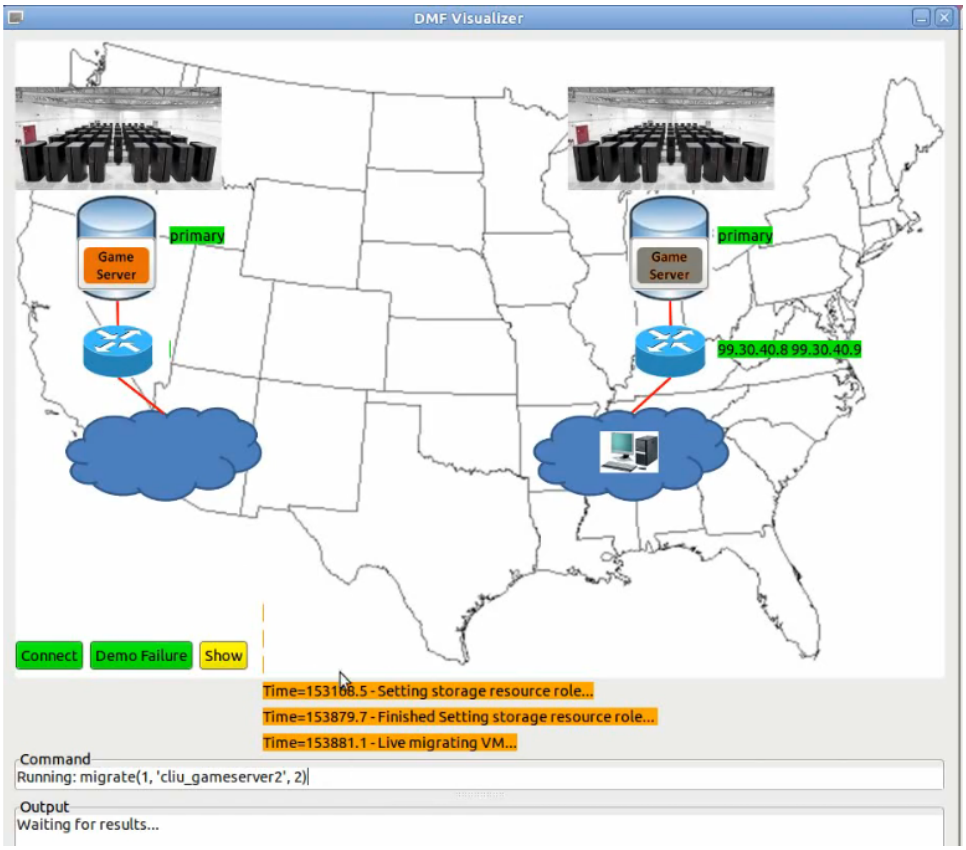


Figure 4.7: Follow-the-Sun cloud service.

create a slice in ShadowNet for TROPIC that consists of a server and a router in each of two ShadowNet locations, *i.e.,* Illinois (IL) and California (CA). The router in each location provides access to the public Internet and is used to create an inter-data center VPN for this slice. The physical servers are configured so that DRBD storage replication can be performed between them and the to-be-migrated VM runs on this storage. We have also implemented a GUI visualizer for Follow-the-Sun and we have a demo video available at [10].

## 4.7 Summary

This chapter presents TROPIC, a highly available transactional framework for service providers to safely and efficiently orchestrate cloud resources. Our experience in building cloud services on top of TROPIC demonstrates its usability in handling errors, enforcing constraints, and eliminating race conditions.

# Chapter 5

# Evaluation

We have developed a prototype of our proposed automated cloud resource orchestration platform which consists of COPE and TROPIC. As shown in Figure 2.1, within the platform TROPIC reports cloud system states to COPE as optimization input, and orchestration commands generated by COPE are fed into TROPIC to perform actual orchestration operations.

In this chapter we present extensive evaluation of this platform in terms of two layers – first the *transactional orchestration layer* and then the *automated orchestration layer*. Specifically, this includes evaluating TROPIC in terms of its design goals, followed by COPE evaluation with two representative resource orchestration scenarios – ACloud and Follow-the-Sun.

## 5.1 Transactional Orchestration Layer

In this section, we present the evaluation of our TROPIC prototype implementation. We emulate cloud orchestration workloads using traces from two production systems. The first trace (*EC2*) is inferred from Amazon EC2 and is representative of the rate

at which VMs are created within a large scale cloud environment. We use this trace to evaluate the *performance* of TROPIC, in particular its ability to achieve the design goal of *high concurrency*, as defined in terms of metrics such as transaction overhead, latency and throughput.

The EC2 trace is limited to VM spawn operations, which does not capture all the complexities involved in cloud orchestration. We therefore make use of a second workload (*hosting*) derived from the traces obtained from a large US hosting provider. We use this second workload to evaluate the *safety*, *robustness* and *high availability* aspects of TROPIC.

Throughout the experiments, we run three TROPIC controllers, instantiated on three physical machines. Each machine has 32GB memory with 8-core $3.0GHz$ Intel Xeon E5450 CPU processors and runs CentOS Linux 5.5, interconnected via Gigabit Ethernet. TROPIC runs one physical worker with multiple threads[1] which co-locates with one of the physical machines. As the distributed coordinator and replicated persistent storage, three ZooKeeper instances reside on the same set of physical machines.

In the first three subsections (Section 5.1.1–5.1.3) of our evaluation, we focus primarily on controller overhead in the logical layer. Actual physical layer benchmarks are presented in Section 5.1.4.

### 5.1.1 Performance

**Workload.** The EC2 workload used to evaluate the performance of TROPIC was collected in July 2011. We measured the number of newly launched VM instances over a week period in the US-east region using the methodology described

[1]TROPIC can of course run multiple workers, but doing so does not alter the conclusions drawn from our evaluation results.

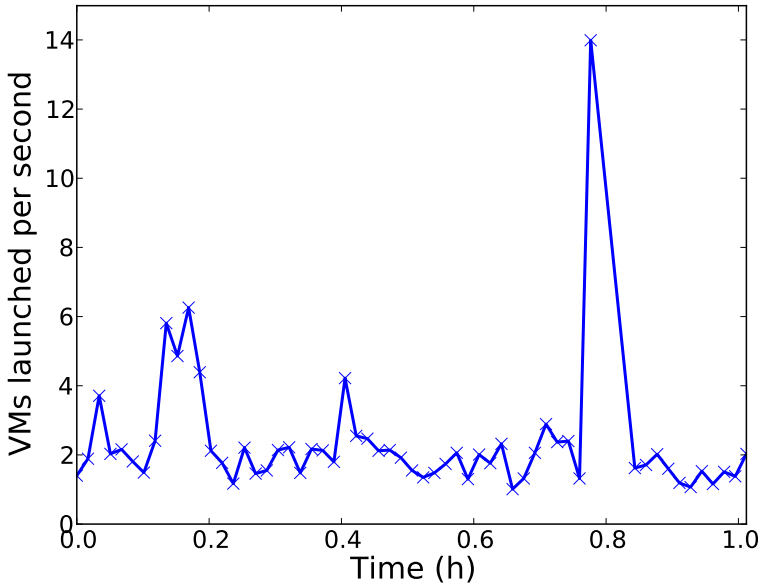


Figure 5.1: VMs launched per second (EC2 workload).

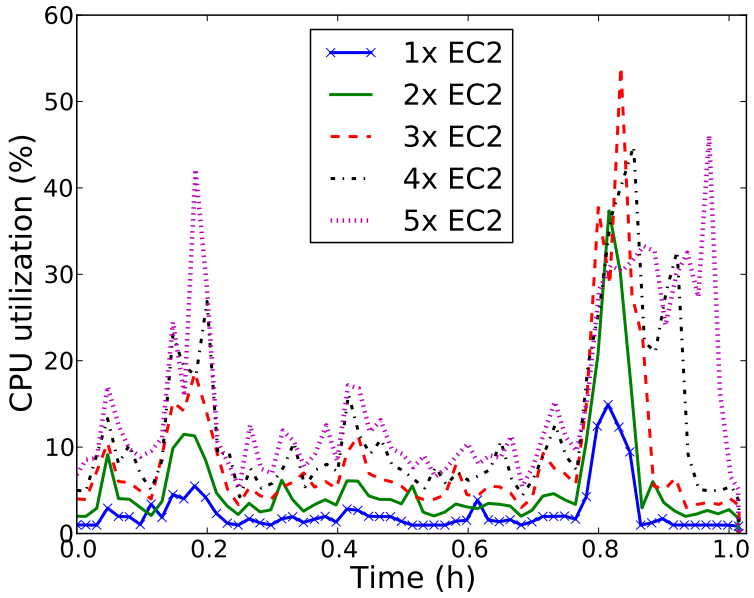


Figure 5.2: Controller CPU utilization (EC2 workload).

by RightScale [3]. Specifically, we created a VM instance every 60 seconds and recorded the VM ID. The ID (after decoding) is unique and the distance between any two consecutive IDs reflects the quantity of VMs spawned in between. Figure 5.1 shows the measured workload in a 1-hour period. The workload in total contains 8417 VM spawnings, with an average of 2.34 per second and a peak of 14.0 at 0.8 hours. We choose this time window because it has a typical average VM launch rate (2 VMs/s) and also the highest peak rate during the week we observed.

**Controller CPU overhead.** Next we use the 1-hour EC2 trace to inject the synthetic workload in TROPIC, by submitting VM spawn transactions every second. To simulate a large-scale cloud environment, we run TROPIC in the *logical-only* mode (Section 4.6) with 12,500 compute servers. Each server has 8 VMs, totaling 100,000 VMs (our target scale). 3,125 storage servers are used to hold the VM images, *i.e.,* 4 compute servers share a storage server. To explore the behavior of TROPIC under higher load, we further multiply the EC2 workload from 2 times (2×) to 5 times (5×), and measure the CPU utilization of the controller (leader) as shown in Figure 5.2.

We observe that the CPU utilization is synchronized with the workloads. As the workloads scale up, CPU utilization rises linearly. However, even during the peak load of 5× EC2 workload, the CPU only reaches as high as 54.0%. After 0.8 hours the CPU peaks of 4× and 5× EC2 workloads retain longer than the workload peak. It is because during the period TROPIC reached the limit of transaction throughput, and hence experienced delays in processing each transaction. Additionally, we measure the memory footprint of TROPIC controller. It is relatively stable, at around 5.4% (of 32GB) for all workloads. We note that the dominant factor contributing to the memory footprint is the quantity of all managed cloud resources, instead of the active workload.

**Transaction latency.** Figure 5.3 shows a detailed breakdown of per-transaction latency results, in the form of a cumulative distribution function (CDF). We define the transaction latency as the time duration from the submission of a transaction until it is successfully committed or aborted. In Figure 5.3 the median latency is less than 1s for all the workloads. For 1× workload, the latency is almost negligible. As expected, 4× and 5× workloads have higher transaction latency, mostly as a result of the workload spike from 0.8 to 1.0 hours.

**Performance breakdown.** To further investigate the factors affecting performance bottlenecks of TROPIC under high load, Figure 5.4 shows a breakdown of time overhead of TROPIC from 0.8 to 1.0 hours. The overhead includes *CPU* (time spent in the logical layer execution), *I/O* (ZooKeeper I/O API calls), and *Idle* (blocking at reading from `inputQ` when there is no input). Not surprisingly, more time are consumed by *CPU* and *I/O* as the workloads scale up. More importantly, we note the dominant overhead comes from *I/O*, on average 2.1 times of *CPU*. This is because ZooKeeper I/O requires the quorum among ZooKeeper instances as well as logging into replicated persistent storage on disks, which is order-of-magnitude longer than in-memory data manipulation in the logical layer.

**Scalability analysis.** The EC2 workload demonstrates that TROPIC can well handle production load of concurrent VM spawn transactions. Our next set of experiments evaluate TROPIC's scalability as cloud resources and loads are increased. Instead of using the EC2 workload, we use a synthetically generated workload that consists of a larger variety of transaction types, including VM *Spawn*, *Start*, *Stop* and *Migrate*. This allows us to study transaction throughput for a wider range of transaction types.

First, we measure transaction throughput as the quantity of resources scales up, *i.e.,* compute servers from 12,500 to 225K, each with 8 VMs. This equals to as

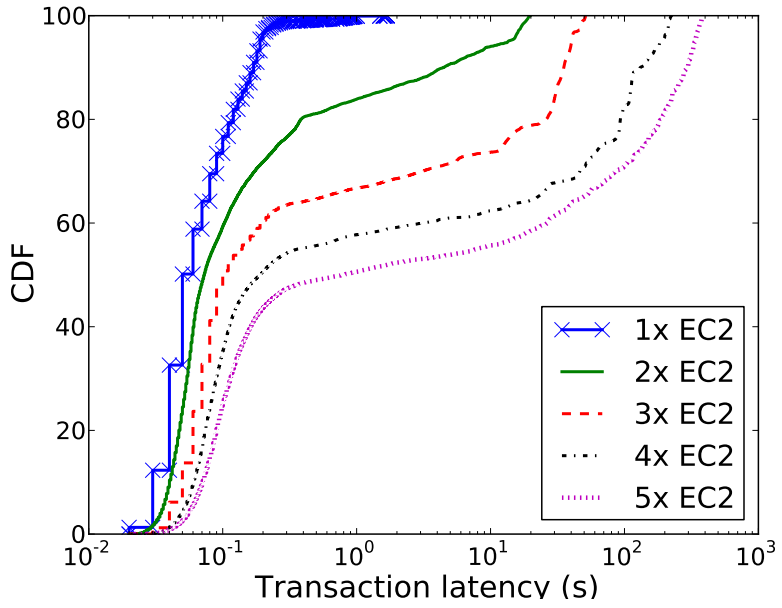


Figure 5.3: CDF of transaction latency (EC2 workload).

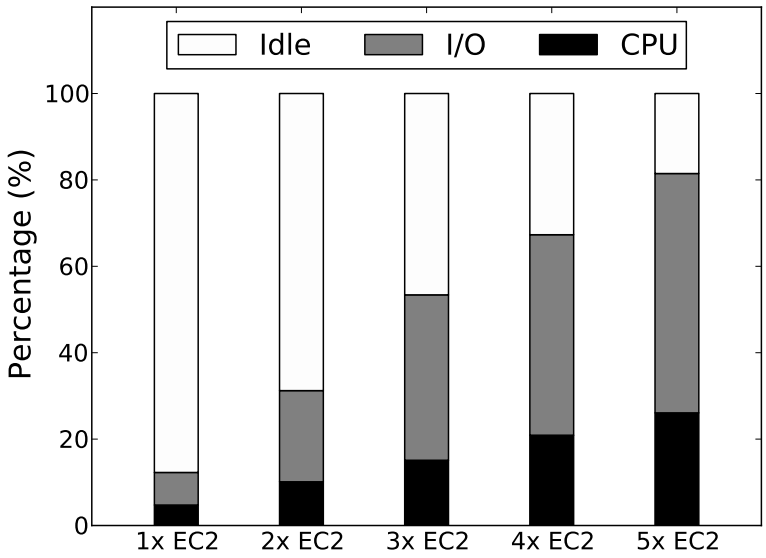


Figure 5.4: Controller overhead at 0.8–1.0 hours (EC2 workload).

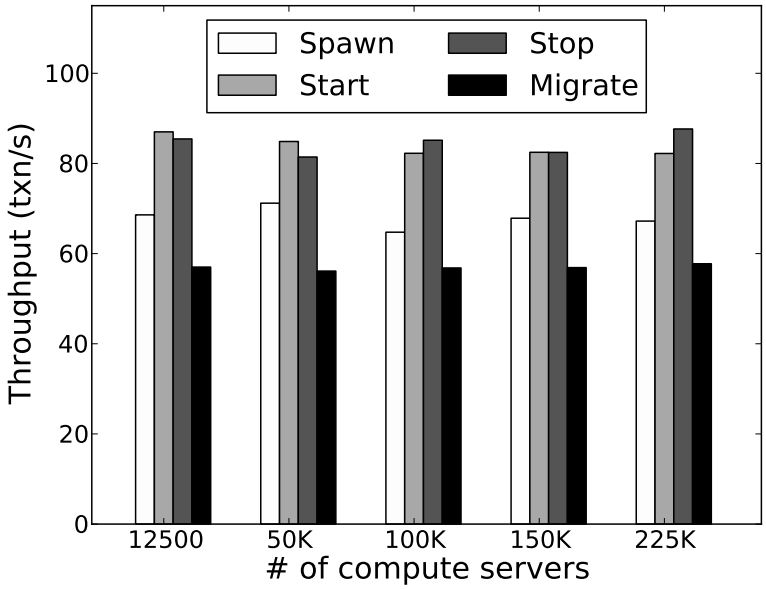


Figure 5.5: Transaction throughput as # of compute servers scales up.

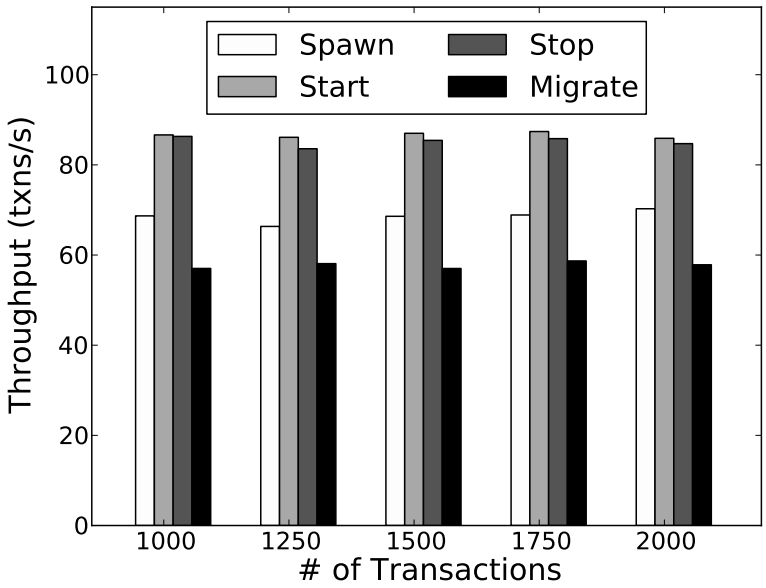


Figure 5.6: Transaction throughput as # of transactions scales up.

many as 1.8 million VMs (more than the scale of a typical data center). We submit 1,500 transactions at the start of the experiment, measure the time to complete all transactions, and derive transaction throughput accordingly.

Figure 5.5 shows that the transaction throughput is steady across all resource scales, averaging 67.9 transactions per second for *Spawn*, 83.8 for *Start*, 84.4 for *Stop* and 56.9 for *Migrate*. VM *Spawn* and *Migrate* take more time than *Start* and *Stop* because they involve more complex logical layer processing (*e.g.,Spawn* consists of five actions as exemplified in Section 4.3).

Our results demonstrate that TROPIC transaction throughput stays constant even as the number of resources increases. This is due in part to our efficient implementation and optimizations (Section 4.6). Moreover, most of the factors affecting throughput (*e.g.,* locking overhead, Zookeeper queue management) incur constant costs. The main bottleneck of TROPIC lies instead with physical memory used to store the data model. For instance, when there are 225K compute servers, the memory consumption is 92.1% on our physical machine. Given our specific hardware, the maximum resource scale TROPIC can handle is 2 million VMs.

We repeat the previous experiment but vary the number of transactions (load input) from 1,000 to 2,000, with a step of 250. We use the setting of 12,500 compute servers with 8 VMs each, totaling 100,000 VMs. We observe that the transaction throughput (given in Figure 5.6) is similar to Figure 5.5 and stable across different loads.

To study TROPIC's logical / physical layer resource inconsistency behavior under automatic `repair` reconciliation mechanism (Section 4.4), we conduct another set of simulation experiment where random resource volatility is introduced. Specifically, in our experiment, we focus on VMs and their random crashes (failures). We run $N = 1024$ VMs in one data center, and use a discrete event simulator to simulate

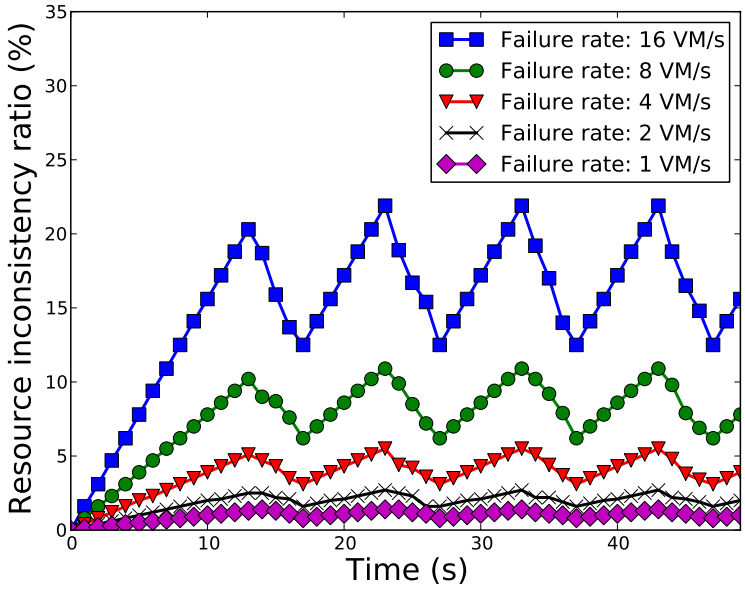


Figure 5.7: Cloud resource failure and TROPIC's `repair`.

random VM crashes. The failure rate $r_f$ ranges from 1 to 16 VMs per second, as we vary the degree of resource volatility. TROPIC `repair` is invoked every $t_r = 10$ seconds. To repair a crashed VM, `repair` first compares the VM state in the logical layer (*i.e.,running*) and the physical layer (*i.e.,stopped*), and then restarts the VM if the two states differ. Each VM is set to take $t_s = 5$ seconds to restart on average, with a standard deviation $\sigma_s = 2$ seconds. Given the above mathematical model, the maximum resource inconsistency ratio (*i.e.,* non-running VMs out of all VMs) can be roughly derived as $r_f * (t_r + t_s)/N$, and the minimum is $r_f * t_s/N$. The inconsistency ratio does not drop to zero because while VMs are being repaired, new VM crashes are constantly introduced.

Figure 5.7 depicts our simulation results. The X-axis corresponds to a total of 50-second simulation, and the Y-axis is the resource inconsistency ratio. We observe that over time the inconsistency ratios across all five groups gradually rise.

Every 10 seconds, the ratios drop since `repair` transactions reconcile the cross-layer inconsistencies by bringing crashed VM back to running. As VMs become more volatile with higher failure rates, the resource inconsistency ratios increase accordingly. Suppose a transaction $t$ consists of a series of actions to orchestrate $n$ VMs (*e.g.*, pause, suspend, or stop), and each action randomly manipulates an independent VM, then the probability that $t$ will commit can be computed as $(1-p)^n$, since each action may fail with a probability of $p$, which is the resource inconsistency ratio. The probability that $t$ aborts is $1-(1-p)^n$. If the resource failure rates are so high that $p$ reaches 100%, then all transactions will abort. In other words, TROPIC cannot perform any effective orchestration.

### 5.1.2 Safety

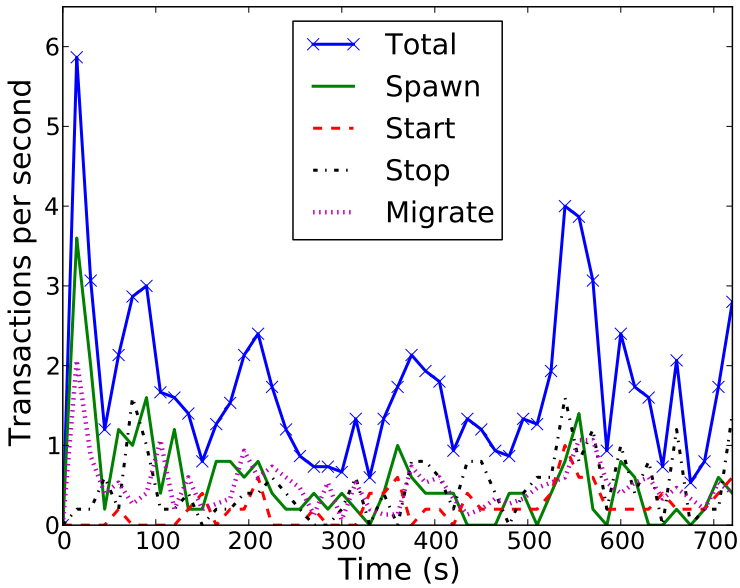


Figure 5.8: Workload derived from a data center hosting trace.

To evaluate the design goals of safety, robustness and high availability of TROPIC,

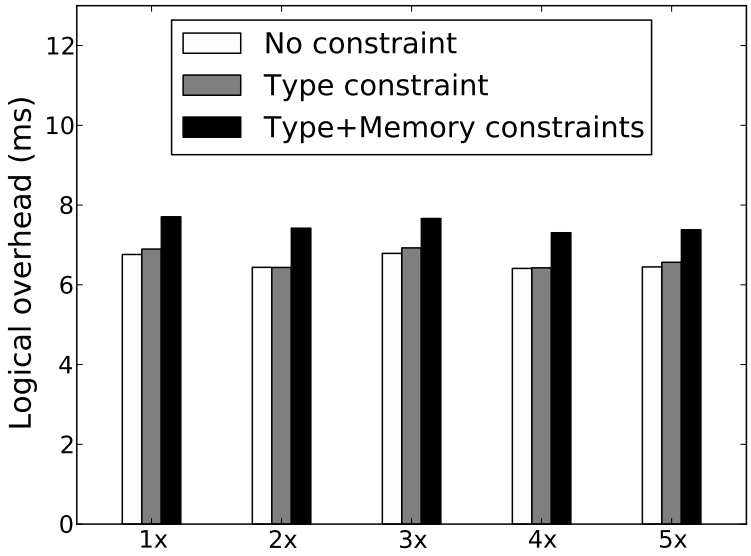


Figure 5.9: Safety overhead (hosting workload).

we use the *hosting* workload derived from a data center trace obtained from a large US hosting provider. Unlike the EC2 workload, it involves a more complex set of orchestration procedures.

The trace contains 248 customers hosted on a total of 1,740 statically allocated physical processors (PPs). Each customer application is deployed on a subset of the PPs. The entire trace is one-month in duration, and it primarily consists of sampling of CPU utilization at each PP gathered at 300 seconds interval. From the trace, we generate the hosting workload consisting of VM *Spawn*, *Start*, *Stop* and *Migrate* operations to mimic a realistic ACloud deployment (Section 4.6). We use 480 servers, of which 384 are compute hosts and 96 are storage hosts. Each compute server runs 8 VMs. Figure 5.8 shows the hosting workload broken down by orchestration types over a period of 720 seconds. It corresponds to an actual 4-hour

period in the original trace (20-to-1 time scale reduction). In total, there are 1257 transactions, with an average of 1.75 transaction per second and a peak at 5.9.

We first use the hosting workload to evaluate the overhead of enforcing constraints in TROPIC. We consider two representative constraints featured in ACloud:

**VM type constraint.** VM migration cannot be performed across hosts running different hypervisors, *e.g.,* from Xen to VMWare. While evaluating an open-source cloud platform, we accidentally performed such an operation. The platform in question accepted the operation without an error. However, the VM was stuck in the "migrating" state indefinitely and became unrecoverable. Using TROPIC, this mistake can be easily avoided by specifying a constraint that ensures each hypervisor can only run VMs with its compatible type (Appendix B). This automatically prevents the illegal VM migration scenario.

**VM memory constraint.** Many cloud orchestrations such as VM spawn and start involve a common operation which is to power on a VM. As mentioned before in Section 4.1, when starting VMs on a host, it is necessary to guarantee that there is enough physical memory, otherwise in our experience the host machine might freeze and lead to service disruption. TROPIC can avoid this with ease by adding a constraint that aggregated VMs memory cannot exceed the host's capacity (lines 16–19 in Figure 4.3.) This constraint inherently guarantees that VMs on each host will not be excessive.

Figure 5.9 shows the logical layer overhead incurred in checking the above constraints. We focus primarily on per-transaction CPU overhead, since the bulk of constraint checking overhead happens at the logical layer. We compare across three cases: (i) *No constraint* does not impose any constraint; (ii) *Type constraint* imposes the VM type constraint; and (iii) *Type+Memory constraint* imposes both constraints.

On average, *No constraint* takes 6.57ms, while *Type constraint* takes 1.2% extra overhead. Due to the additional VM memory constraint in *Type+Memory constraint*, we observe another 12.8% increase in CPU time.

The overhead incurred by the VM memory constraint is significantly higher than the first one. This is because the VM memory constraint needs to retrieve and aggregate data from all underlying affected resources (*i.e.,* all VMs on each host), more expensive to evaluate than the simple "type checking" in the first constraint. To examine TROPIC's performance under different degrees of loads, we scale up the hosting workload by up to 5×, and observe that the results are stable across all workloads.

### 5.1.3 Robustness

To evaluate TROPIC's performance in guaranteeing robustness via transaction atomicity, we highlight two cases from our experiences in deploying ACloud.

**VM spawning error.** As described earlier in Section 4.2, and as implemented in ACloud, VM spawning involves multiple steps. Errors can happen in any step and thus prevent the user from getting a working VM. For example, in the last step of starting a VM in Xen, the Xen daemon may occasionally fail with an "out of memory" error[2], even though the server still has enough memory. This error usually happens when the server is under high load. In this scenario, the VM creation transaction succeeds in the logical layer without any constraint violations, but fails when performing the actual physical operation. Fortunately, TROPIC guarantees transaction atomicity via rollback. This automatically avoids the undesirable scenario that a failed VM creation operation results in the cloned disk image and configuration

[2]A similar problem is reported at "http://lists.xensource.com/archives/html/xen-users/2010-05/msg00646.html".

file becoming "orphans", occupying resources and possibly disrupting future VM creations due to name conflict.

**VM migration error.** In ACloud, VM migration involves three steps: creating VM configuration file on the destination host, migrating the VM from source to destination host, and deleting original configuration file on the source host. If an error happens at the last step (*e.g.,* file deletion failure), TROPIC performs transaction rollback by first migrating the VM back to the source host, and then deleting the configuration file on the destination host.

In our experiment, we measure the logical layer overhead of TROPIC in performing transaction rollback in the presence of the previous two errors. To emulate the errors, we execute TROPIC with the hosting workload, and randomly raise exceptions in the last step of VM spawn and migrate. We run three groups of experiments: *Default*, *Fail 5%* and *Fail 10%* with failure probability of 0%, 5% and 10%, respectively. We repeat the experiment for the workload scaling from 1× to 5×.

Figure 5.10 shows that on a per-transaction basis, *Fail 5%* increases the CPU overhead by 14.5% compared to *Default*, while *Fail 10%* additionally adds 11.1% more computation time. The overhead is mostly due to applying undos in transaction rollback. In all our experiments, the logical layer operations complete in less than 9ms. This demonstrates that TROPIC is efficient at handling transaction errors and rollback.

### 5.1.4 High Availability

In our final set of experiments, we evaluate the availability of TROPIC in the presence of controller failures. Unlike our previous experiments where TROPIC runs in

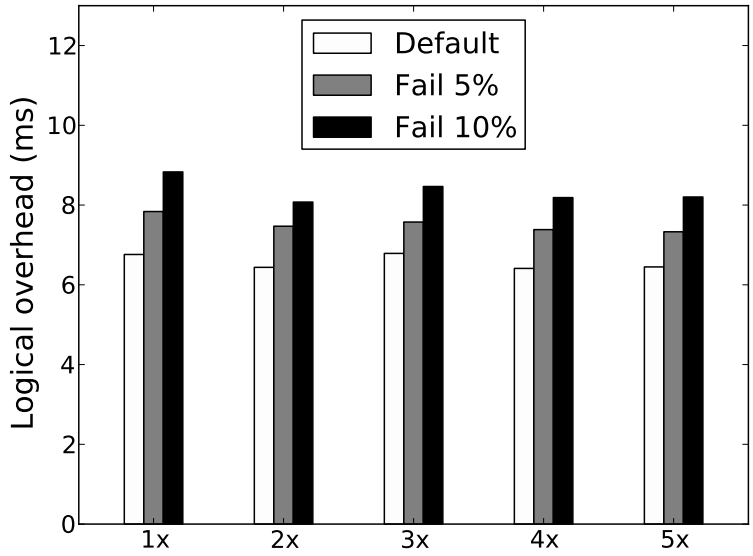


Figure 5.10: Robustness overhead (hosting workload).

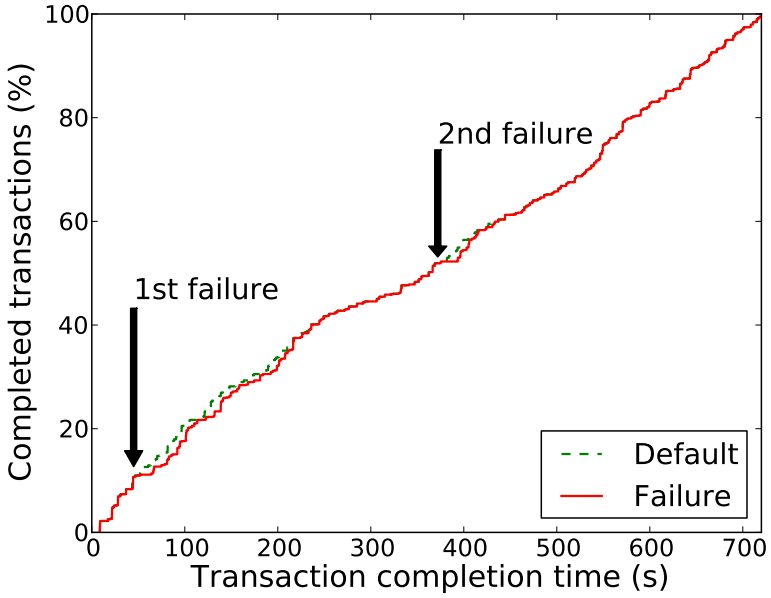


Figure 5.11: High availability. Controller failure at 60s and 390s.

the *logical-only* mode, we execute the hosting workload with parts of the cloud resources as real machines in our ACloud deployment on the ShadowNet [41] testbed. Specifically, among the 480 servers, we use 15 actual machines, accommodating a total of 96 CentOS Linux 5.5 VMs. The physical machines are geographically dispersed across three locations (California, Illinois, and Texas). Each physical machine has 32GB memory, with 8-core 3.0$GHz$ Intel Xeon E5450 CPU processors, and runs CentOS Linux 5.5 as host OS. The machines within each location are connected via Gigabit Ethernet, while a layer-2 VPN provides inter-site connectivity.

**Physical layer overhead.** Table 5.1 lists the average physical execution times for each type of transaction, measured by running TROPIC in ACloud deployment with a smaller benchmark consisting of a subset of the hosting workload. The results show that TROPIC logical layer overhead is negligible compared to the actual physical layer overhead. Comparing across all four, `VM start` is the quickest with 2.10s, `VM stop` takes a much longer time of 9.19s since powering off a VM involves plenty routine procedures in Linux shutdown. For `VM migrate`, the dominant factor is the VM memory size (300MB in ACloud) and the network bandwidth among compute servers. We use the measured times to simulate the physical actions (via sleeping) of the remaining synthetic resources in the workload.

| Transaction | Quantity | Logical (ms) | Physical (s) |
|---|---|---|---|
| `VM spawn` | 16 | 8.00 | 6.40 |
| `VM start` | 6 | 2.33 | 2.10 |
| `VM stop` | 11 | 2.18 | 9.19 |
| `VM migrate` | 8 | 9.75 | 8.56 |

Table 5.1: Breakdown of transactions. The columns include the quantity and the average execution time taken by each transaction.

**Failover time.** Figure 5.11 shows the percentage of completed transactions as a function of time, under two scenarios. In *Default*, we execute the workload with no

controller failure. *Failure* has the same settings as *Default*, except that we manually failed the lead controller at time 60s (1st failure) and the new leader at 390 seconds (2nd failure). After the second failure, only one controller remained and it took the role of being the leader.

Figure 5.11 shows that the failure recovery time is minimal, when comparing *Failure* against *Default*. In each failure case, TROPIC required a small amount of additional time, 12.5s and 11.7s respectively, to process the same percentage of transactions. This demonstrates that TROPIC can recover quickly enough to resume processing on-going transactions. Moreover, the majority of the time gap (about 10s) is incurred by ZooKeeper's leader failure detection, not the failure recovery procedure in TROPIC controller (*e.g.,* rebuild internal data and play transaction logs). One can tune the configuration parameters of ZooKeeper to shorten this period thus making failure detection faster.

**Failover correctness.** Finally, we have explored the above failure scenarios at all 17 failure points presented in Section 4.5. We observe that at each failure point, a TROPIC controller is able to recover gracefully as the new leader, while achieving short failure recovery time without losing any transactions.

## 5.2 Automated Orchestration Layer

This section provides a performance and effectiveness evaluation of COPE. Our prototype system is developed using the RapidNet declarative networking engine [25] and the Gecode [13] constraint solver. COPE takes as input policy goals and constraints written in *Colog*, and then generates RapidNet and Gecode in C++, using the compilation process described in Section 3.4.

Our experiments are carried out using a combination of realistic network simulations and actual distributed deployments, using production traces. In our *simulation-based* experiments, we use RapidNet's built-in support for the ns-3 simulator [21], an emerging discrete event-driven simulator which emulates all layers of the network stack. This allows us to run COPE instances in a simulated network environment and evaluate COPE distributed capabilities. In addition, we can also run our experiments under an *implementation mode*, which enables users to run the same COPE instances, but uses actual sockets (instead of ns-3) to allow COPE instances deployed on real physical nodes to communicate with each other.

Our evaluation aims to demonstrate the following. First, COPE is a general platform that is capable of enabling a wide range of cloud resource orchestrations. Second, most of the policies specified in COPE result in orders of magnitude reduction in code size compared to imperative implementations. Third, COPE incurs low communication overhead and small memory footprint, requires low compilation time, and converges quickly at runtime for distributed executions.

Our evaluation section is organized around various use cases that we have presented in Section 3.2. These include: (1) ACloud load balancing orchestration (Section 3.3.2 and 4.6.3); and (2) Follow-the-Sun orchestration (Section 3.3.3 and 4.6.4). Our cloud orchestration use cases derive their input data from actual data center traces obtained from a large hosting company. In our evaluations, we run COPE over the ns-3 simulator.

### 5.2.1 Compactness of Colog Programs

We first provide evidence to demonstrate the compactness of our *Colog* implementations, by comparing the number of rules in *Colog* and the generated C++ code.

| Protocol | *Colog* | Imperative (C++) |
|---|---|---|
| ACloud (centralized) | 10 | 935 |
| Follow-the-Sun (centralized) | 16 | 1487 |
| Follow-the-Sun (distributed) | 32 | 3112 |

Table 5.2: *Colog* and compiled C++ comparison.

Table 5.2 illustrates the compactness of *Colog*, by comparing the number of *Colog* rules (2nd column) for the three representative programs we have implemented against the actual number of lines of code (LOC) in the generated RapidNet and Gecode C++ code (3rd column) using `sloccount`. Each *Colog* program includes all rules required to implement Gecode solving and RapidNet distributed communications. The generated imperative code is approximately $100X$ the size of the equivalent *Colog* program. The generated code is a good estimation on the LOC required by a programmer to implement these protocols in a traditional imperative language. In fact, *Colog*'s reduction in code size should be viewed as a lower bound. This is because the generated C++ code implements only the rule processing logic, and does not include various COPE's built-in libraries, *e.g.,* Gecode's constraint solving modules and the network layers provided by RapidNet. These built-in libraries need to be written only once, and are reused across all protocols written in *Colog*.

While a detailed user study will allow us to comprehensively validate the usability of *Colog*, we note that the orders of magnitude reduction in code size makes *Colog* programs significantly easier to fast model complex problems, understand, debug and extend than multi-thousand-line imperative alternatives.

### 5.2.2 Use Case: ACloud

In our first set of experiments, we perform a trace-driven evaluation of the ACloud scenario as described in Section 3.2, which has been realized in TROPIC (Section 4.6). Here, we assume a single cloud controller deployed with COPE, running the centralized ACloud program written in *Colog* (Section 3.3.2). Benchmarking the centralized program first allows us to isolate the overhead of the solver, without adding communication overhead incurred by distributed solving.

**Experimental workload.** As input to the experiment, we reuse the data center trace obtained from a large hosting company in the US, as described in Section 5.1. Based on the trace, we generate a workload in a hypothetical cloud environment similar to ACloud where there are 15 physical machines geographically dispersed across 3 data centers (5 hosts each). Each physical machine has 32GB memory. We preallocate 80 migratable VMs on each of 12 hosts, and the other 3 hosts serve as storage servers for each of the three data centers. This allows us to simulate a deployment scenario involving about 1000 VMs. We next use the trace to derive the workload as a series of VM operations:

- **VM spawn:** CPU demand (% PP used) is aggregated over all PPs belonging to a customer at every time interval. We compute the average CPU load, assuming that load is equally distributed among the allocated VMs. Whenever a customer's average CPU load per VM exceeds a predefined high threshold (80% in our experiment) and there are no free VMs available, one additional VM is spawned on a random host by cloning from an image template.

- **VM stop and start:** Whenever a customer's average CPU load drops below a predefined low threshold (20% in our experiment), one of its VMs is powered off to save resources (*e.g.,* energy and memory). We assume that powered-off VMs

are not reclaimed by the cloud. Customers may bring their VMs back by powering them on when the CPU demands become high later.

Using the above workload, the ACloud program takes as input `vm(Vid,Cpu,Mem)` and `host(Hid,Cpu,Mem)` tables, which are continuously being updated by the workload generator as the trace is replayed.

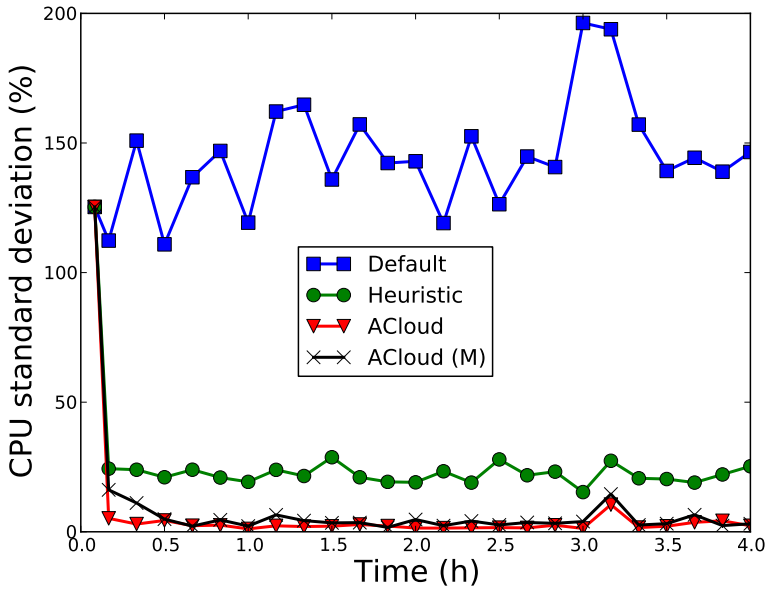


Figure 5.12: Average CPU standard deviation of three data centers (ACloud).

**Policy validation.** We compare two ACloud policies against two strawman policies (default and heuristic):

- **ACloud.** This essentially corresponds to the *Colog* program presented in Section 3.3.2. We configure the ACloud program to periodically execute every 10 minutes to perform a COP computation for orchestrating load balancing via VM migration within each data center. To avoid migrating VMs with very low CPUs, the `vm` table only includes VMs whose CPU utilization is larger than 20%.

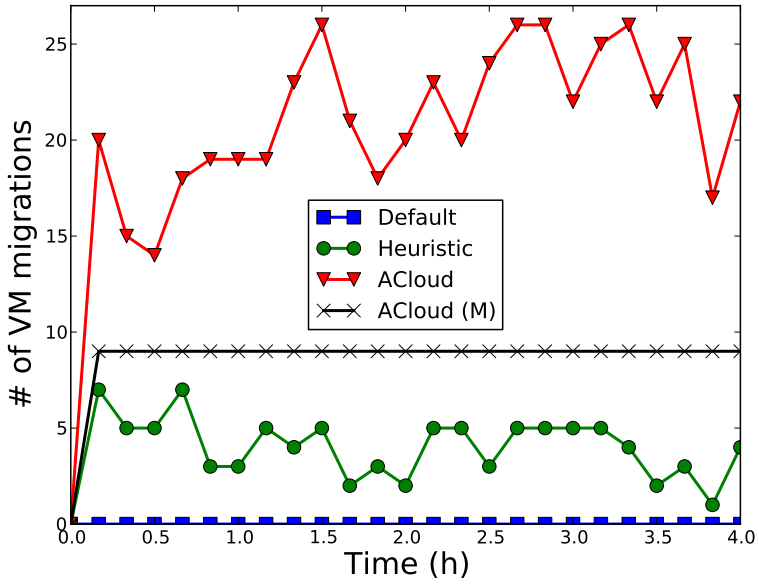


Figure 5.13: Number of VM migrations (ACloud).

- **ACloud (M).** To demonstrate the flexibility of *Colog*, we provide a slight variant of the above policy, that limits the number of VM migrations within each data center to be no larger than 3 for each interval. This requires only minor modifications to the *Colog* program, by adding rules `d5-6` and `c3` as shown in Section 3.3.2.

- **Default.** A naïve strategy, which simply does no migration after VMs are initially placed on random hosts.

- **Heuristic.** A threshold-based policy that migrates VMs from the most loaded host (*i.e.,* with the highest aggregate CPU of the VMs running on it) to the least one, until the most-to-least load ratio is below a threshold $K$ (1.05 in our experiment). *Heuristic* emulates an ad-hoc strategy that a cloud operator may adopt in the absence of COPE.

Figure 5.12 shows the average CPU standard deviation of three data centers achieved by the *ACloud* program over a 4 hours period. We observe that *ACloud* is able to more effectively perform load balancing, achieving a 98.1% and 87.8% reduction of the degree of CPU load imbalance as compared to *Default* and *Heuristic*, respectively. *ACloud (M)* also performs favorably compared to *Default* and *Heuristic*, resulting in a marginal increase in standard deviation.

Figure 5.13 shows that on average, *ACloud* migrates 20.3 VM migrations every interval. On the contrary, *ACloud (M)* (with migration constraint) substantially reduces the number of VM migrations to 9 VMs per interval (3 per data center).

**Compilation and runtime overhead.** The *Colog* program is compiled and executed on an Intel Quad core 2.33GHz PC with 4GB RAM running Ubuntu 10.04. Compilation takes on average 0.5 seconds (averaged across 10 runs). For larger-scale data centers with more migratable VMs, the solver will require exponentially more time to terminate. This makes it hard to reach the optimal solution in reasonable time. As a result, we limit each solver's COP execution time to 10 seconds. Nevertheless, we note from our results that the solver output still yields close-to-optimal solutions. The memory footprint is 9MB (on average), and 20MB (maximum) for the solver, and 12MB (relatively stable) for the base RapidNet program.

### 5.2.3 Use Case: Follow-the-Sun

Our second evaluation is based on the Follow-the-Sun scenario. We use the distributed *Colog* program (Section 3.3.3) for implementing the Follow-the-Sun policies. The focus of our evaluation is to validate the effectiveness of the Follow-the-Sun program at reducing total cost for cloud providers, and to examine the scalability, convergence time and overhead of distributed solving using COPE. Our evaluation

is carried out by running COPE in simulation mode, with communication directed across all COPE instances through the ns-3 simulator. We configure the underlying network to use ns-3's built-in 10Mbps Ethernet, and all communication is done via UDP messaging.

**Experimental workload.** Our experiment setup consists of multiple data centers geographically distributed at different locations. We conducted 5 experimental runs, where we vary the number of data centers from 2 to 10. For each network size, we execute the distributed *Colog* program once to determine the VM migrations that minimize the cloud providers' total cost. Recall from Section 3.3.3 that this program executes in a distributed fashion, where each node runs a local COP, exchanges optimization outputs and reoptimizes, until a fixpoint is reached.

The data centers are connected via random links with an average network degree of 3. In the absence of actual traces, our experimental workload (in particular, operating and communication and migration costs) here are synthetically generated. However, the results still provide insight on the communication/computation overhead and effectiveness of the Follow-the-Sun program.

Each data center has a resource capacity of 60 units of migratable VMs (the unit here is by no means actual, *e.g.,* one unit can denote 100 physical VMs). Data centers have a random placement of current VMs for demands at different locations, ranging from 0 to 10. Given that data centers may span across geographic regions, communication and migration costs between data centers may differ. As a result, between any two neighboring data centers, we generate the communication cost randomly from 50 to 100, and the migration cost from 10 to 20. The operating cost is fixed at 10 for all data centers.

**Policy validation.** Figure 5.14 shows the total costs (migration, operating, and communication) over time, while the Follow-the-Sun program executes to a fixpoint

in a distributed fashion. The total cost corresponds to the `aggCost` (optimization goal) in the program in Section 3.3.3. To make it comparable across experimental runs with different network sizes, we normalize the total cost so that its initial value is 100% when the COP execution starts. We observe that in all experiments, Follow-the-Sun achieves a cost reduction after each round of distributed COP execution. Overall the cost reduction ranges from 40.4% to 11.2%, as the number of data centers increases from 2 to 10. As the network size gets larger, the cost reduction is less apparent. This is because distributed solving approximates the optimal solution. As the search space of COP execution grows exponentially with the problem size, it becomes harder for the solver to reach the optimal solution.

To demonstrate the flexibility of *Colog* in enabling different Follow-the-Sun policies, we modify the original Follow-the-Sun program slightly to limit the number of migrations between any two data centers to be less than or equal to 20, achieved with rules `d11` and `c3` as introduced in Section 3.3.3. This modified policy achieves comparable cost reduction ratios and convergence times as before, while reducing the number of VM migrations by 24% on average.

**Compilation and runtime overhead.** The compilation time of the program is 0.6 seconds on average for 10 runs. Figure 5.14 indicates that as the network size scales up, the program takes a longer time to converge to a fixpoint. This is due to more rounds of link negotiations. The periodic timers between each individual link negotiation is 5 seconds in our experiment. Since the solver computation only requires input information within a node's neighborhood, each per-link COP computation during negotiation is highly efficient and completes within 0.5 seconds on average. The memory footprint is tiny, with 172KB (average) and 410KB (maximum) for the solver, and 12MB for the RapidNet base program.

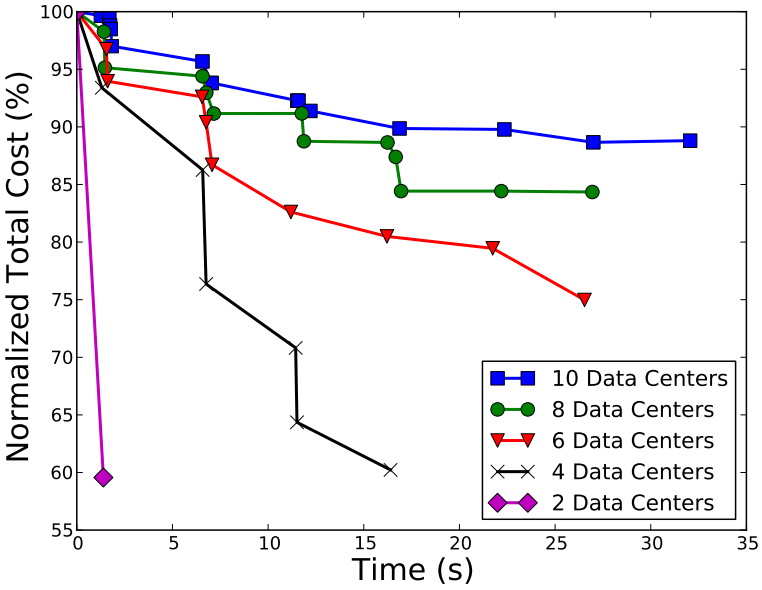


Figure 5.14: Total cost as distributed solving converges (Follow-the-Sun).

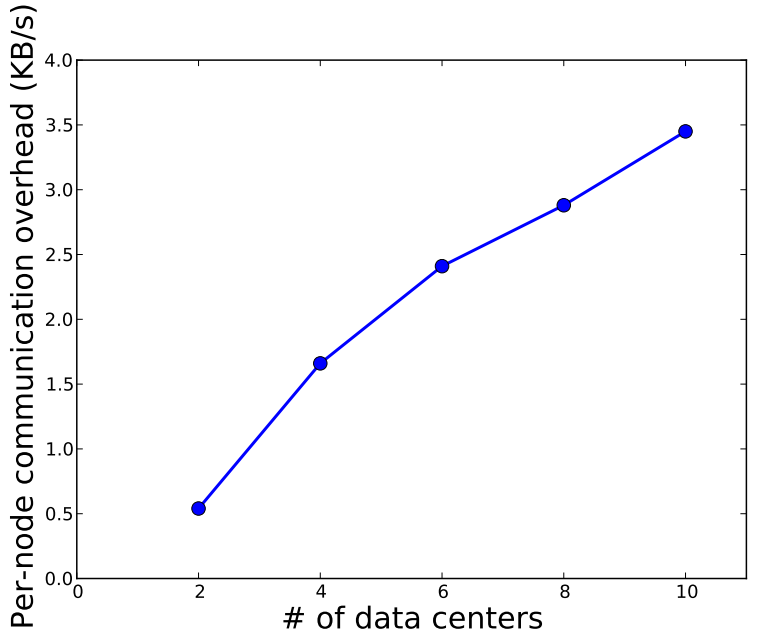


Figure 5.15: Per-node communication overhead (Follow-the-Sun).

In terms of bandwidth utilization, we measure the communication overhead during distributed COP execution. The per-node communication overhead is shown in Figure 5.15. We note that COPE is highly bandwidth-efficient, with a linear growth as the number of data centers scales up. For *10 data centers*, the per-node communication overhead is about $3.5KBps$.

## 5.3 Summary

The evaluation of our TROPIC prototype shows its capability to support production workload at scale with high degrees of concurrency, provide high availability with low overhead, and ensure the transactional semantics of cloud operations. COPE evaluation results demonstrate its feasibility, both in terms of the wide range of policies supported, and the efficiency of the platform itself.

# Chapter 6

# Related Work

In this chapter, we summarize related work in the domains of declarative networking, constraint optimization, and cloud resource orchestration.

## 6.1 Declarative Networking

*Colog* is a superset of a domain-specific programming language *NDlog* (*i.e.,* Network Datalog) used in declarative networking [69]. In declarative networks users specify network protocols as *NDlog* programs which are compiled into dataflow runtime graphs. This is in principle ensembles the Click extensible router dataflows [55]. *NDlog* enables a variety of routing protocols and overlay networks to be specified in a natural and concise manner. For example, traditional routing protocols such as the path vector and distance-vector protocols can be expressed in a few lines of code [71], and the Chord distributed hash table in 47 lines of code [70]. When compiled and executed, these declarative protocols perform efficiently relative to imperative implementations. As evidence of its widespread applicability, declarative techniques have

been used in several domains, including, fault tolerance protocols [88], cloud computing [35], sensor networks [42], overlay network compositions [72], wireless adaptive routing [60, 61], wireless channel selection [59], and as a basis for course projects in a networked systems class [44].

In addition to ease of implementation, another advantage of the declarative approach is its amenability to formal and structured forms of correctness checks. These include the use of theorem proving [94], algebraic techniques for constructing safe routing protocols [95], and runtime verification [105]. These formal analysis techniques are strengthened by recent work on formally proving correct operational semantics of *NDlog* [76]. Finally, the dataflow framework used in declarative networking naturally captures information flow as distributed queries, hence providing a natural way to use the concept of network provenance [104] to analyze and explain the existence of any network state.

## 6.2 Constraint Optimization

Prior to COPE, there have been a variety of systems that use declarative logic-based policy languages to express constraint optimization problems in resource management of distribute computing systems. [97] proposes continuous optimization based on declaratively specified policies for autonomic computing. [85] describes a model for automated policy-based construction as a goal satisfaction problem in utility computing environments. The XSB engine [32] integrates a tabled Prolog engine with a constraint solver. Rhizoma [102] proposes using rule-based language and constraint solving programming to optimize resource allocation. [75] uses a logic-based interface to a SAT solver to automatically generate configuration solution for a single

data center. [43] describes compiling and executing centralized declarative modeling languages to Gecode programs.

Unlike the prior work, COPE is designed for a different application domain – one that aims to achieve cost reduction and maximize infrastructure utilization by automatically orchestrating the cloud based on policies configured by cloud providers and customers. Another unique feature of COPE is its support for distributed optimizations, achieved by using a general declarative policy language–*Colog* which is user-friendly for constraint solving modeling and results in orders of magnitude code size reduction compared to imperative alternatives, and the integration of a distributed query engine with a constraint solver. COPE platform supports both simulation and deployment modes. This enables one to first simulate distributed COP execution within a controllable network environment and then physically deploy the system on real devices. Our grand vision is that COPE is targeted to be a general-purpose declarative distributed platform for constraint optimizations that can support the original use cases and more. Actually in our prior work, we have made initial attempts at developing specialized optimization platforms tailored towards wireless network configuration [59]. Our work [67] generalizes ideas from these early experiences to develop a general framework, a declarative programming language, and corresponding compilation techniques.

## 6.3 Cloud Resource Orchestration

There are a variety of commercial IaaS providers such as Amazon EC2 [2] and Microsoft Windows Azure [29]. Unfortunately, how they orchestrate cloud resources is proprietary. Open-source cloud control platforms, such as OpenStack [23], Eucalyptus [12], and OpenNebula [22] have predefined cloud service models embedded in

their implementations. However, none of them provide transactional resource management at the granularity of cloud operations. In contrast, TROPIC is not simply a cloud service, but a general-purpose programming platform to build safe, robust, and highly available cloud services.

Transaction processing has been studied in database area for decades [78]. As a programming paradigm, it has also received more attentions recently from the systems community. For instance, Locus [96] and TxOS [77] introduce transaction APIs to OS system calls: a system transaction executes a series of system calls in isolation and atomically publishes the effects to the rest of the OS. Transaction support has also been proposed for file systems [90], as well as a user-level library [87] for lightweight data management. In Microsoft Windows Vista/7, the Kernel Transaction Manager (KTM) [30] enables the development of transactional registry and transactional file system. When failure happens, transactions are rolled back to restore previous states.

Although not in the cloud computing scope, there are several related frameworks proposed for the management of systems and networks. From the open-source community, Puppet [24] is a data center automation and configuration management framework. Puppet has a transactional layer, but not in the sense of enforcing ACID properties. Instead it allows user to visually examine the operations before they are submitted as a dry run. Once executed, they are not guaranteed to be atomically committed. COOLAID [40] is a data-centric network configuration management framework. It adopts a relational data model and uses Datalog to encode network protocol dependencies. COOLAID provides transactional configuration updates to routers, but supports neither concurrent transactions nor VM management. Autopilot [53] is a data center software management infrastructure for automating software provisioning, monitoring and deployment. It has repair actions similar to TROPIC,

but it does not provide a transactional programming interface. Its periodic repair procedures maintain weak consistency between the provisioning data repository and the deployed software code.

TROPIC borrows ideas from these prior work, such as undo log based rollback, multi-granularity locking. However, the transactional orchestration in TROPIC is unique, in dealing with the logical and physical layer separation and volatile nature of cloud resources, with a "safety-first" mindset, and in providing high concurrency and availability.

# Chapter 7

# Conclusion

## 7.1 Conclusion

This dissertation presents an automated cloud resource orchestration architecture, a novel approach that provides: (1) COPE, a platform that enables cloud providers to automate the process of cloud orchestration formulated as declarative constraint optimizations. COPE uses a declarative policy language *Colog* to specify orchestration polices concisely and enables distributed optimization; (2) TROPIC, a transactional orchestration platform with a unified data model that enables cloud providers to develop complex cloud services with robustness, safety, high concurrency, and high availability. We further discuss two concrete use cases ACloud and Follow-the-Sun as representative cloud orchestration scenarios, and demonstrate how the *Colog* language enables a wide range of policies to be customized in support of these two scenarios.

We have implemented a complete TROPIC prototype deployed on the ShadowNet testbed [41]. We extensively evaluated our prototype using production-scale traces obtained from EC2 and a large US hosting provider, demonstrating that TROPIC

is able to manage cloud resources at a large scale, while ensuring transactional semantics and high availability. Moreover, we have implemented a prototype of COPE based on RapidNet declarative networking engine and Gecode constraint solver, and have integrated COPE with TROPIC. We have demonstrated the viability of COPE in specifying both centralized and distributed optimization scenarios in *Colog*. We validate our prototype using realistic scenarios and workloads derived from production cloud services that orchestrate compute, storage, and network resources within and across geographically distributed data centers for load balancing and consolidation. Our evaluation results demonstrate the feasibility of COPE, both in terms of the wide range of policies supported, and the efficiency of the platform itself. We expect our platform to be able to rapidly build reliable and sophisticated IaaS cloud services.

Moving forward, we are in the process of releasing TROPIC and COPE (as well as the realized ACloud and Follow-the-Sun cloud services) to the open-source community for use by other researchers and practitioners to quickly build reliable cloud services. We have also started integrating their capabilities into the popular OpenStack platform. This integration of transactional task management was highlighted in AT&T's official announcement [1, 10] in collaboration with OpenStack to launch to a developer friendly cloud.

## 7.2 Open Questions and Future Directions

In this section, we discuss some open questions and future directions in order to make TROPIC and COPE cloud resource orchestration platform into reality. Moreover, look beyond this dissertation, our long-term goal is to enable automated building and management of large scale distributed systems with safety and robustness. This

dissertation work is the first step towards this vision, and we would like to leverage TROPIC and COPE framework to further explore other areas.

### 7.2.1 Open Questions

COPE's general constraint optimization approach has a few limitations. First, there exist problems which can not be easily formulated as constraint optimization problems. For instance, it is hard to accurately quantify network throughput given a strategy of allocating bandwidth shared by VMs [49]. For these problems, we suggest that COPE may not be the best solution. Another limitation of COPE is that when the scale of a constraint optimization problem is very large, COPE could not produce a satisfiable solution using either centralized or distributed (*i.e.,* divide-and-conquer) solving. To deal with these cases, users are advised to design customized heuristic algorithms [58] instead.

While TROPIC is designed within a data center, we intend to investigate distributed orchestrations coordinated by multiple TROPIC instances, *e.g.,* each instance responsible for one type of cloud resources [23]. The open research issues include distributed transactions, data replication and partitioning, and their interactions with manageability.

In the cloud there could be scenarios where customer SLAs and providers operational objectives may not be consistent. For instance, customers may put CPU as load balancing objective while providers desire to balance network bandwidth utilization. We plan to extend our platform so that it can detect such inconsistencies in optimization goals and constraints and report errors to users. Alternatively, the platform can be extended to give users options to remove some conflicting policies so that it can find a solution.

### 7.2.2 Cloud Ecosystem

Building upon TROPIC and COPE, we hope to incorporate into the integrated framework recent models on resource provisioning and deployment analysis in the cloud. We plan to build and evaluate novel and more sophisticated cloud services in the framework to further explore new opportunities, which include cloud storage provisioning and distribution [83, 92, 11, 31], virtualized desktop [46], database consolidation [45], execution runtime [16], network management [34, 33], MapReduce job processing [98], and federated experiment testbed deployment [14]. The eventual goal is to build a cloud ecosystem which provides a variety of services ranging from infrastructure, platform, and software. One interesting use case within this ecosystem is in mobile computing. It is expected that as mobile devices become more pervasive, heavy computing tasks will be mostly shifted into the backend in the cloud. Driven by this emerging trend, we plan to explore the use of this dissertation work as well as our prior research in optimizing wireless network performance [59, 60, 61] (some examples are given in Appendix C) to deploy mobile computing services more effectively. Research questions here include reducing data access latency and increasing data bandwidth between mobile devices and the cloud, providing reliable and highly available services, considering geographical factors which may affect performance.

### 7.2.3 Declarative Distributed Constraint Optimizations

In distributed systems management, operators often have to configure system parameters that optimize performance objectives given constraints in the deployment environment. We believe COPE is a flexible and general distributed constraint optimization platform [67], which not only applies to the cloud, but also to a variety of vastly different domains. We expect that such a platform has tremendous

practical value in facilitating extensible distributed systems optimizations. In fact our prior experience demonstrates COPE's approach in managing and optimizing adaptive wireless channel selection and routing in mobile ad-hoc and mesh networks [62, 59, 60, 65, 61, 74] (Appendix C). As future work, we plan to apply this approach to explore additional use cases in a wide range of emerging domains that involve distributed constraint optimization, including decentralized data analysis and model fitting, engineering design, control and signal processing, finance, resource allocations in other distributed settings, network design and optimizations, etc.

### 7.2.4 Automated Synthesis of Declarative Optimizations

As a longer-term agenda, we hope to develop novel techniques to aid distributed system operators to synthesize *Colog* programs automatically. Currently, COPE requires operators to manually specify optimization models including constraints and goals. In large-scale distributed systems, figuring out these goals and constraints may be a non-trivial process. For instance, it was challenging for engineers to pinpoint the root causes of the recent Amazon [27, 28] and Blackberry outages [18]. To automate this process, we plan to explore the use of machine learning and formal analysis techniques (in particular satisfiability modulo theories solving) to develop new strategies to automatically *synthesize* goals and constraints, by learning from example configurations, operating experiences and even failures in deployed systems. This is an active area of research currently in the programming languages community, as shown by recent work in program sketching [89], concurrent programming [84], and auto-completion of spreadsheets [51]. It is an inter-disciplinary research topic that we hope to explore, in collaboration with researchers in machine learning, programming languages, and formal methods.

We also intend to explore the use of high-level declarative languages for formally reasoning about the convergence time and quantifying the optimality of distributed systems configurations specified in *Colog*. Depending on the specific problem and how *Colog* is modeled, distributed optimization may produce results with different level of optimality. Applying formal techniques to verifying *Colog* program correctness and detecting constraint contradiction will as well be interesting future work beyond this dissertation.

# Appendix A

# *Colog* Grammar and Syntax

The grammar that specifies the *Colog* language syntax is described below, with following conventions:

- Lower-lettered phrases in `typewriter font` are grammar classes.
- Lower-lettered phrases in *italic font* are literal constants.
- Precedence is lower for alternative forms that appear later.
- Upper-lettered phrases are literal constants generated by the lexer. In particular, `NAME` represents the name of a predicate beginning with a lower-case letter; `AGG` represents the name of an aggregation operator (`MIN`, `MAX`, `COUNT`, `SUM`, `AVG`, `STDEV` are currently supported); `VALUE` represents numerical constants; `STRING` represents string constants in double quotes; `VAR` represents variable names starting with a upper-case letter. It may also be the symbol "_" which read as do-not-care; `FUNC` represents function names beginning with "`f_`".

```
program     ::=  clauselist

clauselist  ::=  clause
               | clause clauselist

clause      ::=  rule
               | materialize
               | solver_time
               | solver_goal
               | solver_var

solver_time ::=  solver_time VALUE

solver_goal ::=  goal goal_type NAME in functor

goal_type   ::=  minimize
               | maximize
               | satisfy

solver_var  ::=  var functor forall functor

materialize ::=  materialize(functorname,tablearg,tablearg,primkeys)
               | materialize(functorname,tablearg,tablearg,primkeys,
                 range(VALUE, VALUE))
```

```
tablearg    ::=  VALUE

primkeys    ::=  keys(keylist)

keylist     ::=  VALUE
                | VALUE, keylist

rule        ::=  NAME functor <- termlist.
                | NAME functor -> termlist.
                | NAME delete functor <- termlist.

termlist    ::=  term
                | term, termlist

term        ::=  functor
                | assign
                | select

functor     ::=  functorname functorbody
                | delete functorname functorbody
                | insert functorname functorbody
                | refresh functorname functorbody
                | overwrite functorname functorbody

functorname ::=  NAME
```

```
functorbody ::=  (functorargs)

functorargs ::=  functorarg
               | functorarg, functorargs
               | @ atom
               | @ atom, functorargs

functorarg  ::=  atom
               | aggregate

aggregate   ::=  AGG<VAR>
               | AGG<@VAR>
               | AGG<*>

function    ::=  FUNC(funcargs)
               | FUNC()

funcargs    ::=  funcarg
               | funcarg, funcargs

funcarg     ::=  math_expr
               | atom

select      ::=  expr
```

```
assign      ::=  VAR := expr

expr        ::=  atom
                | function
                | ( expr )
                | not expr
                | expr binaryop expr
                | [ ]
                | [ exprlist ]

exprlist    ::=  expr
                | expr,exprlist

atom        ::=  VALUE
                | VAR
                | STRING
                | NULL

binaryop    ::=  + | - | * | / | and | or | ::
                | == | != | > | < | >= | <=
```

# Appendix B

# TROPIC Code Examples

Below we give more TROPIC code examples on compute (VM), storage, and network (router) cloud resources to complement Figure 4.3 and Table 4.1.

```
class VM(LogicalModel):
    type = Attribute(str)

    @constraint
    def domain0State(self):
        if (self.name == "Domain-0" and
            self.state != VMState.On):
            yield ("Domain-0 must be running", self)

    @action
    def pause(self, ctxt):
        self.state = VMState.Paused
        ctxt.appendlog(action="pause", args=(),
            undo_action="unpause", undo_args=())

    @action
    def unpause(self, ctxt):
        self.state = VMState.On
        ctxt.appendlog(action="unpause", args=(),
            undo_action="pause", undo_args=())
...
```

Figure B.1: TROPIC code example–Compute (VM).

```
class VMHost(LogicalModel):
    type = Attribute(str)

    @constraint
    def typeCompatible(self):
        for vm in self.vms:
            if self.type != vm.type:
                yield ("VM and host types are incompatible", self)

    @action
    def stopVM(self, ctxt, name):
        self.vms[name].state = VMState.Off
        ctxt.appendlog(action="stopVM", args=[name],
            undo_action="startVM", undo_args=[name])

    @action
    def createVM(self, ctxt, name, image):
        vm = VM(name, image)
        self.vms.insert(vm)
        ctxt.appendlog(action="createVM", args=[name, image],
            undo_action="removeVM", undo_args=[name, image])

    @action
    def removeVM(self, ctxt, name, image):
        del self.vms[name]
        ctxt.appendlog(action="removeVM", args=[name, image],
            undo_action="createVM", undo_args=[name, image])

    @action
    def saveVM(self, ctxt, name, path):
        self.vms[name].state = VMState.Off
        ctxt.appendlog(action="saveVM", args=[name, path],
            undo_action="restoreVM", undo_args=[name, path])

    @action
    def restoreVM(self, ctxt, name, path):
        self.vms[name].state = VMState.On
        ctxt.appendlog(action="restoreVM", args=[name, path],
            undo_action="saveVM", undo_args=[name, path])
...
```

Figure B.2: TROPIC code example–Compute (VMHost).

```
class BlockDevice(LogicalModel):
    name = Attribute(str)
    file = Attribute(str)

    @primaryKey
    def id(self):
        return self.name
...

class StorageHost(LogicalModel):
    uri = Attribute(str)
    hostname = Attribute(str)
    devices = Many(BlockDevice)

    @action
    def exportImage(self, ctxt, name):
        dev = BlockDevice(name)
        self.devices.insert(dev)
        ctxt.appendlog(action="exportImage", args=[name],
            undo_action="unexportImage", undo_args=[name])

    @action
    def unexportImage(self, ctxt, name):
        del self.devices[name]
        ctxt.appendlog(action="unexportImage", args=[name],
            undo_action="exportImage", undo_args=[name])

    @action
    def cloneImage(self, ctxt, template, target):
        ctxt.appendlog(action="cloneImage", args=[template, target],
            undo_action="deleteImage", undo_args=[target])

    @action
    def deleteImage(self, ctxt, name):
        ctxt.appendlog(action="deleteImage", args=[name],
            undo_action=None, undo_args=())
...

class StorageRoot(LogicalModel):
    hosts = Many(StorageHost)
...
```

Figure B.3: TROPIC code example–Storage.

```
class Router(LogicalModel):
    name = Attribute(str)
    bgpRoutes = Attribute(dict)

    @primaryKey
    def id(self):
        return self.name

    @action
    def addBgpRoute(self, ctxt, netAddr, nextHop):
        self.bgpRoutes[netAddr] = nextHop
        ctxt.appendlog(action="addBgpRoute", args=[netAddr, nextHop],
            undo_action="delBgpRoute", undo_args=[netAddr, nextHop])

    @action
    def delBgpRoute(self, ctxt, netAddr, nextHop):
        del self.bgpRoutes[netAddr]
        ctxt.appendlog(action="delBgpRoute", args=[netAddr, nextHop],
            undo_action="addBgpRoute", undo_args=[netAddr, nextHop])
...

class RouterRoot(LogicalModel):
    routers = Many(Router)
...
```

Figure B.4: TROPIC code example–Network (router).

# Appendix C

# Wireless Network Configuration

In addition to cloud resource orchestration, another use case of COPE's declarative constraint optimization approach is based on optimizing wireless networks by adjusting the selected channels used by wireless nodes to communicate with one another [59]. In wireless networks, communication between two adjacent nodes (within close radio range) would result in possible interference. As a result, a popular optimization strategy performed is to carefully configure channel selection and routing policies in wireless mesh networks [47, 48]. These proposals aim to mitigate the impact of harmful interference and thus improve overall network performance. For reasonable operation of large wireless mesh networks with nodes strewn over a wide area with heterogeneous policy constraints and traffic characteristics, a *one-size-fits-all* channel selection and routing protocol may be difficult, if not impossible, to find.

To address the above needs, our platform PUMA (Policy-based Unified Multi-radio Architecture) serves as a basis for developing intelligent network protocols that simultaneously control parameters for dynamic (or agile) spectrum sensing and

access, dynamic channel selection and medium access, and data routing with a goal of optimizing overall network performance.

In PUMA, channel selection policies are formulated as COPs that are specified using *Colog*. The customizability of *Colog* allows providers a great degree of flexibility in the specification and enforcement of various local and global channel selection policies. These policy specifications are then compiled into efficient constraint solver code for execution. *Colog* can be used to express both centralized and distributed channel selection protocols.

In the next sections, We first formulate wireless channel selection as a constraint optimization problem (COP), followed by presenting its equivalent *Colog* programs (both centralized and distributed).

## C.1 COP Formulation

In wireless channel selection, the optimization variables are the channels to be assigned to each communication link, while the values are chosen from candidate channels available to each node. The goal in this case is to minimize the likelihood of interference among conflicting links, which maps into the well-known *graph-coloring* problem [54].

We consider the following example that avoids interference based on the *one-hop interference* model [101]. In this model, any two adjacent links are considered to interfere with each other if they both use channels whose frequency bands are closer than a certain threshold. The formulation is as follows:

**Input domain and variables:** Consider a network $G = (V, E)$, where there are nodes $V = \{1, 2, \ldots, N\}$ and edges $E \subseteq V \times V$. Each node $x$ has a set of channels $P_x$

currently occupied by primary users [1] within its vicinity. The number of interfaces of each node is $i_x$.

**Optimization goal:** For any two adjacent nodes $x, y \in V$, $l_{xy}$ denotes the link between $x$ and $y$. Channel assignment selects a channel $c_{xy}$ for each link $l_{xy}$ to meet the following optimization goal:

$$\min \sum_{l_{xy}, l_{xz} \in E, y \neq z} cost(c_{xy}, c_{xz}) \tag{C.1}$$

where $cost(c_{xy}, c_{xz})$ assigns a unit penalty if adjacent channel assignments $c_{xy}$ and $c_{xz}$ are separated by less than a specified frequency threshold $F_{mindiff}$:

$$cost(c_{xy}, c_{xz}) = \begin{cases} 1 & \text{if } |c_{xy} - c_{xz}| < F_{mindiff} \\ 0 & \text{otherwise} \end{cases} \tag{C.2}$$

**Constraints:** The optimization goal has to be achieved under the following three constraints:

$$\forall l_{xy} \in E, c_{xy} \notin P_x \tag{C.3}$$

$$\forall l_{xy} \in E, c_{xy} = c_{yx} \tag{C.4}$$

$$\forall x \in V, |\bigcup_{l_{xy} \in E} c_{xy}| \leq i_x \tag{C.5}$$

(C.3) expresses the constraint that a node should not use channels currently occupied by primary users within its vicinity. (C.4) requires two adjacent nodes to communicate with each other using the same channel. (C.5) guarantees the number of assigned channels is no more than radio interfaces.

[1] Primary users own exclusive rights to certain spectrum in white space networks.

## C.2 Centralized Channel Selection

In centralized channel selection [80, 38], a channel manager is deployed on a single node in the network. Typically, this node is a designated server node, or is chosen among peers via a separate leader election protocol. The centralized manager collects the network status information from each node in the network – this includes their neighborhood information, available channels, and any additional local policies. The following *Colog* program takes as input the `link` table, which stores the gathered network topology information, and specifies the one-hop interference model COP formulation described in Section C.1. We omit the location specifier `@` for brevity.

```
goal minimize C in totalCost(C)
var assign(X,Y,C) forall link(X,Y)

// cost derivation rules
d1  cost(X,Y,Z,C) <- assign(X,Y,C1), assign(X,Z,C2),
                     Y!=Z, (C==1)==(|C1-C2|<F_mindiff).
d2  totalCost(SUM<C>) <- cost(X,Y,Z,C).

// primary user constraint
c1  assign(X,Y,C) -> primaryUser(X,C2), C!=C2.

// channel symmetry constraint
c2  assign(X,Y,C) -> assign(Y,X,C).

// interface constraint
```

```
d3  uniqueChannel(X,UNIQUE<C>) <- assign(X,Y,C).
c3  uniqueChannel(X,Count) -> numInterface(X,K), Count<=K.
```

**Optimization goal and variables:** The goal in this case is to minimize the cost attribute `C` in `totalCost`, while assigning channel variables `assign` for all communication links. Each entry of the `assign(X,Y,C)` table indicates channel `C` is used for communication between `X` and `Y`.

**Solver derivations:** Rule `d1` sets cost `C` to 1 for each `cost(X,Y,Z,C)` tuple if the chosen channels that `X` uses to communicate with adjacent nodes `Y` and `Z` are interfering. Rule `d2` sums the number of interfering channels among adjacent links in the entire network, and stores the result in `totalCost`.

**Solver constraints:** Constraint `c1` restricts the domain of `assign(X,Y,C)` to only valid channel assignments for existing links `link(X,Y)` and ensures that only available channels are considered. Constraint `c2` applies to the input `availChannel` table, and states that a channel `C` at node `X` is only available, if there does not exist a primary user within the vicinity of `X`. Constraint `c3` enforces channel symmetry on the output `assign` table.

In some wireless deployments, e.g. IEEE 802.11, the *two-hop interference model* [101] is often considered a more accurate measurement of interference. This model considers interference that results from any two links using similar channels within two hops of each other. The two-hop interference model requires minor modifications to rule `d1` as follows:

```
d3  cost(X,Y,Z,W,C) <- assign(X,Y,C1), link(Z,X), assign(Z,W,C2),
                       X!=W, Y!=W, Y!=Z,
```

```
(C==1)==(|C1-C2|<F_mindiff).
```

The above rule considers four adjacent nodes `W`, `Z`, `X`, and `Y`, and assigns a cost of 1 to node `X`'s channel assignment with `Y` (`assign(X,Y,C1)`), if there exists a neighbor `Z` of `X` that is currently using channel `C2` that interferes with `C1` to communicate with another node `W`. The above policy requires only adding one additional `link(Z,X)` predicate to the original rule `d1`, demonstrating the customizability of *Colog*. Together with rule `d1`, one can assign costs to both one-hop and two-hop interference models.

## C.3 Distributed Channel Selection

We next demonstrate PUMA's ability to implement distributed channel selection. Our example here is based on a variant of distributed greedy protocol proposed in [91]. This example highlights PUMA's ability to support distributed COP computations, where nodes compute channel assignments based on local neighborhood information, and then exchange channel assignments with neighbors to perform further COP computations. Distributed channel selection provides approximations to the optimal centralized solution, and hence scales better for large networks. Moreover, it has the added advantages of not introducing single points of failure and is amenable to incremental computations as the network topology changes.

The protocol works as follows. Periodically, each node randomly selects one of its links to start a *link negotiation* process with its neighbor. This is similar to the distributed *Colog* program for Follow-the-Sun in Section 3.3.3. To avoid conflicting channel assignments, for any given `link(X,Y)`, the link negotiation protocol selects the node with the larger identifier (or address) to carry out the subsequent channel

negotiation process. Once a link is selected for channel assignment, the result of link negotiation is stored in table `setLink(X,Y)`. The negotiation process then solves a *local* COP and assigns a channel such that interference is minimized. The following *Colog* program implements the local COP operation at every node `X` for performing channel assignment. The output of the program sets the channel `assign(X,Y,C)` for one of its links `link(X,Y)` (chosen for the current channel negotiation process) based on the two-hop interference model:

```
goal minimize C in totalCost(@X,C)
var assign(@X,Y,C) forall setLink(@X,Y)

// cost derivation for two-hop interference model
d1  cost(@X,Y,Z,W,C) <- assign(@X,Y,C1), link(@Z,X),
                        assign(@Z,W,C2), X!=W, Y!=W, Y!=Z,
                        (C==1)==(|C1-C2|<F_mindiff).
d2  totalCost(@X,SUM<C>) <- cost(@X,Y,Z,W,C).

// primary user constraint
c1  assign(@X,Y,C) -> primaryUser(@X,C2), C!=C2.
c2  assign(@X,Y,C) -> primaryUser(@Y,C2), C!=C2.

// propagate channels to ensure symmetry
r1  assign(@Y,X,C) <- assign(@X,Y,C).
```

The distributed program is similar to the centralized equivalent presented in Section C.2, with the following differences:

While the centralized channel selection searches for all combinations of channel assignments for all links, the distributed equivalent restricts channel selection to a single link one at a time, where the selected link is represented by `setLink(@X,Y)` based on the negotiation process. For this particular link, the COP execution takes as input its local neighbor set (`link`) and all currently assigned channels (`assign`) for itself and nodes in the local neighborhood. This means that the COP execution is an approximation based on local information gathered from a node's neighborhood.

Specifically, distributed solver rule `d1` enables node `X` to collect the current set of channel assignments for its immediate neighbors and derive the cost based on the two-hop interference model. In executing the channel selection for the current link, constraint `c1-2` express that the channel assignment for `link(@X,Y)` does not equal to any channels used by `primaryUser`. Once a channel is set at node `X` after COP execution, the channel-to-link assignment is then propagated to neighbor `Y`, hence resulting in symmetric channel assignments (rule `r1`).

# Bibliography


[1] A better cloud for developers. http://www.attinnovationspace.com/innovation/story/a7779660.

[2] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[3] Amazon Usage Estimates. http://blog.rightscale.com/2009/10/05/amazon-usage-estimates/.

[4] Blow-by-blow: Another Microsoft cloud outage. http://www.infoworld.com/t/saas/blow-blow-another-microsoft-cloud-outage-172298.

[5] CHOCO Solver. http://www.emn.fr/z-info/choco-solver/.

[6] Cloud Computing, Managed Hosting, Dedicated Server Hosting by Rackspace. http://www.rackspace.com/.

[7] Cloud Management for Public and Private Clouds by RightScale. http://www.rightscale.com/.

[8] Cloudscaling Open Cloud Solutions. http://www.cloudscaling.com/.

[9] Distributed Resource Scheduler (DRS) - VMware. http://www.vmware.com/products/drs/overview.html.

[10] DMF: Data-centric Managemnet Framework. http://trac.research.att.com/trac/dmf/.

[11] Dropbox. https://www.dropbox.com/.

[12] Eucalyptus Cloud Computing Infrastructure. http://eucalyptus.com/.

[13] Gecode Constraint Development Environment. http://www.gecode.org/.

[14] GENI: Global Environment for Network Innovations. http://www.geni.net/.

[15] GNBD Project. http://sourceware.org/cluster/gnbd/.

[16] Google App Engine. https://developers.google.com/appengine/.

[17] How Big is Amazon's EC2? http://cloudscaling.com/blog/cloud-computing/amazons-ec2-generating-220m-annually.

[18] International BlackBerry outage continues. http://news.cnet.com/8301-30686_3-20118882-266/international-blackberry-outage-continues/.

[19] Libvirt: The virtualization API. http://libvirt.org.

[20] Network configuration (netconf). http://www.ietf.org/html.charters/netconf-charter.html.

[21] Network Simulator 3. http://www.nsnam.org/.

[22] OpenNebula: The Toolkit for Cloud Computing. http://opennebula.org/.

[23] OpenStack: Open Source Cloud Computing Software. http://openstack.org/.

[24] Puppet: A Data Center Automation Solution. http://www.puppetlabs.com.

[25] RapidNet. http://netdb.cis.upenn.edu/rapidnet/.

[26] Running 200 VM instances on OpenStack Compute. http://blog.griddynamics.com/2011/05/running-200-vm-instances-on-openstack.html.

[27] Summary of the Amazon EC2, Amazon EBS, and Amazon RDS Service Event in the EU West Region. http://aws.amazon.com/message/2329B7/.

[28] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/message/65648/.

[29] Windows Azure platform. http://www.microsoft.com/windowsazure/.

[30] Windows Kernel Transaction Manager. http://msdn.microsoft.com/en-us/library/bb986748(VS.85).aspx.

[31] Wuala - Secure Cloud Storage. http://www.wuala.com/.

[32] XSB Project. http://xsb.sourceforge.net/.

[33] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[34] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 281–296, 2010.

[35] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems*, 2010.

[36] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[37] Rick Bradshaw and Piotr T. Zbiegiel. Experiences with eucalyptus: deploying an open source cloud. In *Proceedings of the 24th international conference on Large installation system administration*, 2010.

[38] Vladimir Brik, Eric Rozner, and Suman Banerjee. DSAP: A Protocol for Coordinated Spectrum Access. In *New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on*, pages 611–614, 2005.

[39] Roy Campbell, Indranil Gupta, Michael Heath, Steven Y. Ko, Michael Kozuch, Marcel Kunze, Thomas Kwan, Kevin Lai, Hing Yan Lee, Martha Lyons, Dejan Milojicic, David O'Hallaron, and Yeng Chai Soh. Open Cirrus Cloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 1–5, 2009.

[40] Xu Chen, Yun Mao, Z. Morley Mao, and Jacobus Van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *The 7th*

*International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2010.

[41] Xu Chen, Z. Morley Mao, and Jacobus Van der Merwe. ShadowNet: A Platform for Rapid and Safe Network Evolution. In *Proceedings of the 2009 USENIX Annual technical conference*, 2009.

[42] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, 2007.

[43] Raffaele Cipriano, Agostino Dovier, and Jacopo Mauro. Compiling and Executing Declarative Modeling Languages to Gecode. In *Proceedings of the 24th International Conference on Logic Programming*, pages 744–748, 2008.

[44] CIS 553: Networked Systems. `http://netdb.cis.upenn.edu/cis553projects/`.

[45] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 313–324, New York, NY, USA, 2011. ACM.

[46] Tathagata Das, Pradeep Padala, Venkata N. Padmanabhan, Ramachandran Ramjee, and Kang G. Shin. Litegreen: saving energy in networked desktops using virtualization. In *Proceedings of the 2010 USENIX annual technical conference*, 2010.

[47] Aditya Dhananjay, Hui Zhang, Jinyang Li, and Lakshminarayanan Subramanian. Practical, Distributed Channel Assignment and Routing in Dual-radio Mesh Networks. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 99–110, 2009.

[48] Richard Draves, Jitendra Padhye, and Brian Zill. Routing in Multi-radio, Multi-hop Wireless Mesh Networks. In *Proceedings of the 10th annual international conference on Mobile computing and networking*, pages 114–128, 2004.

[49] Sahan Gamage, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. Opportunistic flooding to improve tcp transmit performance in virtualized clouds. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 24:1–24:14, 2011.

[50] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39:68–73, 2008.

[51] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. In *Communications of the ACM*, 2012.

[52] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX annual technical conference*, 2010.

[53] Michael Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, 2007.

[54] Kamal Jain, Jitendra Padhye, Venkata N. Padmanabhan, and Lili Qiu. Impact of Interference on Multi-hop Wireless Network Performance. In *Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 66–80, 2003.

[55] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[56] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[57] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[58] A. Lim, Y. Zhu, Q. Lou, and B. Rodrigues. Heuristic methods for graph coloring problems. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 933–939, 2005.

[59] Changbin Liu, Ricardo Correa, Harjot Gill, Tanveer Gill, Xiaozhou Li, Shivkumar Muthukumar, Taher Saeed, Boon Thau Loo, and Prithwish Basu. PUMA: Policy-based Unified Multi-radio Architecture for Agile Mesh Networking (full paper and demonstration). In *The Fourth International Conference on COMmunication Systems and NETworkS (COMSNETS)*, pages 1–10, 2012.

[60] Changbin Liu, Ricardo Correa, Xiaozhou Li, Prithwish Basu, Boon Thau Loo, and Yun Mao. Declarative Policy-based Adaptive MANET Routing. In *17th IEEE International Conference on Network Protocols (ICNP)*, pages 354–363, 2009.

[61] Changbin Liu, Ricardo Correa, Xiaozhou Li, Prithwish Basu, Boon Thau Loo, and Yun Mao. Declarative policy-based adaptive mobile ad hoc networking. *Networking, IEEE/ACM Transactions on*, 20(3):770 –783, June 2012.

[62] Changbin Liu, Xiaozhou Li, Shivkumar C. Muthukumar, Harjot Gill, Taher Saeed, Boon Thau Loo, and Prithwish Basu. A Policy-based Constraint-solving Platform Towards Extensible Wireless Channel Selection and Routing. In *PRESTO: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, 2010.

[63] Changbin Liu, Boon Thau Loo, and Yun Mao. Declarative Automated Cloud Resource Orchestration. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 26:1–26:8, 2011.

[64] Changbin Liu, Yun Mao, Xu Chen, Mary F. Fernandez, Boon Thau Loo, and Kobus Van der Merwe. TROPIC: Transactional Resource Orchestration Platform In the Cloud. In *2012 USENIX Annual Technical Conference (ATC'12)*, 2012.

[65] Changbin Liu, Yun Mao, Mihai Oprea, Prithwish Basu, and Boon Thau Loo. A declarative perspective on adaptive manet routing. In *PRESTO: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, 2008.

[66] Changbin Liu, Yun Mao, Jacobus Van der Merwe, and Mary Fernandez. Cloud Resource Orchestration: A Data-Centric Approach. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, pages 1–8, 2011.

[67] Changbin Liu, Lu Ren, Boon Thau Loo, Yun Mao, and Prithwish Basu. Cologne: A Declarative Distributed Constraint Optimization Platform. In *Proc. VLDB Endow.*, volume 5, pages 752–763, April 2012.

[68] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108, 2006.

[69] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking. In *Communications of the ACM*, pages 87–95, 2009.

[70] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[71] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of ACM SIGCOMM Conference on Data Communication*, 2005.

[72] Yun Mao, Boon Thau Loo, Zachary Ives, and Jonathan M. Smith. MOSAIC: Unified Platform for Dynamic Overlay Selection and Composition. In *Proceedings of the 2008 ACM CoNEXT Conference*, 2008.

[73] Mengmeng Liu, Nicholas Taylor, Wenchao Zhou, Zachary Ives, and Boon Thau Loo. Recursive Computation of Regions and Connectivity in Networks. In

*Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1108–1119, 2009.

[74] Shivkumar C. Muthukumar, Xiaozhou Li, Changbin Liu, Joseph B. Kopena, Mihai Oprea, and Boon Thau Loo. Declarative toolkit for rapid network protocol simulation and experimentation. In *ACM SIGCOMM Conference on Data Communication (demo)*, 2009.

[75] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative Infrastructure Configuration Synthesis and Debugging. *J. Netw. Syst. Manage.*, 16(3):235–258, 2008.

[76] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andre Scedrov. Maintaining Distributed Logic Programs Incrementally. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2011.

[77] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[78] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, third edition, 2002.

[79] Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[80] Ashish Raniwala, Kartik Gopalan, and Tzi-cker Chiueh. Centralized Channel Assignment and Routing Algorithms for Multi-channel Wireless Mesh Networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 8(2):50–65, 2004.

[81] Benjamin Reed and Flavio P. Junqueira. A simple totally ordered broadcast protocol. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 2:1–2:6, 2008.

[82] Philipp Reisner. DRBD - Distributed Replication Block Device. In *9th International Linux System Technology Conference*, 2002.

[83] Oriana Riva, Qin Yin, Dejan Juric, Ercan Ucan, and Timothy Roscoe. Policy expressivity in the Anzere personal cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 14:1–14:14, 2011.

[84] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and Avi Mendelson. Programming model for a heterogeneous x86 platform. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 431–440, New York, NY, USA, 2009. ACM.

[85] A. Sahai, S. Singhal, V. Machiraju, and R. Joshi. Automated Policy-based Resource Construction in Utility Computing Environments. In *IEEE/IFIP Network Operations and Management Symposium*, pages 381–393, 2004.

[86] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, November 1984.

[87] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[88] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT Protocols under Fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[89] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 167–178, New York, NY, USA, 2007. ACM.

[90] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In *Proccedings of the 7th conference on File and storage technologies*, 2009.

[91] A.P. Subramanian, H. Gupta, and S.R. Das. Minimum Interference Channel Assignment in Multi-Radio Wireless Mesh Networks. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2007. SECON '07. 4th Annual IEEE Communications Society Conference on*, pages 481–490, 2007.

[92] Ercan Ucan and Timothy Roscoe. Dexferizer: a service for data transfer optimization. In *Proceedings of the Nineteenth International Workshop on Quality of Service*, IWQoS '11, pages 33:1–33:9, Piscataway, NJ, USA, 2011. IEEE Press.

[93] Jacobus Van der Merwe, K.K. Ramakrishnan, Michael Fairchild, Ashley Flavel, Joe Houle, H. Andres Lagar-Cavilla, and John Mulligan. Towards a Ubiquitous Cloud Computing Infrastructure. In *"Proc. of the IEEE Workshop on Local and Metropolitan Area Networks"*, pages 1–6, 2010.

[94] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Towards declarative network verification. In *11th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2009.

[95] Anduo Wang, Limin Jia, Changbin Liu, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally Verifiable Networking. In *8th Workshop on Hot Topics in Networks (ACM SIGCOMM HotNets-VIII)*, 2009.

[96] Matthew J. Weinstein, Thomas W. Page, Jr., Brian K. Livezey, and Gerald J. Popek. Transactions and synchronization in a distributed operating system. In *Proceedings of the tenth ACM symposium on Operating systems principles*, 1985.

[97] S.R. White, J.E. Hanson, I. Whalley, D.M. Chess, and J.O. Kephart. An Architectural Approach to Autonomic Computing. In *International Conference on Autonomic Computing*, pages 2–9, 2004.

[98] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[99] T. Wood, A. Gerber, K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. The case for enterprise-ready virtual private clouds. In *Proc. of HotCloud Workshop*, 2009.

[100] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, 2007.

[101] Yung Yi and Mung Chiang. Wireless Scheduling Algorithms with O(1) Overhead for M-Hop Interference Model. In *Communications, 2008. ICC '08. IEEE International Conference on*, pages 3105–3109, 2008.

[102] Qin Yin, Adrian Schuepbach, Justin Cappos, Andrew Baumann, and Timothy Roscoe. Rhizoma: A Runtime for Self-deploying, Self-managing Overlays. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 10:1–10:20, 2009.

[103] Zheng Zhang, Ming Zhang, Albert Greenberg, Y. Charlie Hu, Ratul Mahajan, and Blaine Christian. Optimizing Cost and Performance in Online Service Provider Networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 33–48, 2010.

[104] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient Querying and Maintenance of Network Provenance at Internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[105] Wenchao Zhou, Oleg Sokolsky, Boon Thau Loo, and Insup Lee. DMaC: Distributed Monitoring and Checking. In *9th International Workshop on Runtime Verification (RV)*, 2009.