TROPIC: Transactional Resource Orchestration Platform In the Cloud

Changbin Liu* Yun Mao† Xu Chen† Mary F. Fernández†
Boon Thau Loo* Jacobus E. Van der Merwe†
*University of Pennsylvania †AT&T Labs - Research

Abstract

Realizing Infrastructure-as-a-Service (IaaS) cloud requires a control platform to orchestrate cloud resource provisioning, configuration, and decommissioning across a distributed set of diverse physical resources. This orchestration is challenging due to the rapid growth of data centers, high failure rate of commodity hardware and the increasing sophistication of cloud services. This paper presents the design and implementation of TROPIC, a highly available, transactional resource orchestration platform for building IaaS cloud infrastructures. TROPIC's orchestration procedures that manipulate physical resources are transactional, automatically guaranteeing atomicity, consistency, isolation and durability of cloud operations. Through extensive evaluation of our prototype implementation, we demonstrate that TROPIC can meet production-scale cloud orchestration demands, while maintaining our design goals of safety, robustness, concurrency and high availability.

1 Introduction

The Infrastructure-as-a-Service (IaaS) cloud computing model exemplified by Amazon EC2 [1] provides users on-demand, near-instant access to a large pool of virtual cloud resources such as virtual machines (VMs), virtual block devices, and virtual private networks. The *orchestrations* of the virtual resources over physical hardware, such as provisioning, configuration, 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 in terms of both scale and fault tolerance, as shown by recent studies [8, 13] on the open-source cloud platforms. First, today's large data centers typically run on the scale of over 10,000 machines based on commodity hardware [15]. 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, the control logic is inherently complex. In particular, any engineering and service rule must be met while avoiding race conditions. The postmortem from the EC2 outage in April 2011 [10] anecdotally reinforces our arguments: 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.

To address these challenges, we propose TROPIC, 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. Specifically, we make the following contributions:

Transactional abstraction. In TROPIC, orchestration procedures are executed as *transactions* 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 underlying volatile distributed resources.

Transaction processing. While TROPIC adopts standard database transaction processing techniques such as write-ahead-logging for atomicity and a hierarchical intention locking scheme for concurrency control [20], we propose a two-layer transaction processing stack to cope with the unique challenges in the cloud. In the *logical layer*, each transaction is analyzed for possible resource contention and constraint violations prior to actual execution. This provides early detection of unsafe operations without touching physical resources. Once deemed safe, the transaction is then executed in the *physical layer*. In the presence of resource failures, TROPIC provides reconciliation mechanisms to handle cross-layer inconsistencies.

High availability. TROPIC adopts a highly available decentralized architecture where all components are decoupled to avoid single point of failure. TROPIC runs multiple controllers, and provides efficient recovery mechanisms such that whenever the lead controller fails, another controller can take its place without service disruption while maintaining transactional semantics.

Prototype implementation and evaluation. We have implemented a complete TROPIC prototype deployed on the ShadowNet testbed [14]. 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.

2 TROPIC Overview

2.1 Design Goals

Our objective is to provide a cloud orchestration platform at the scale of at least 100,000 cloud resources (*e.g.,* VMs and block devices) in one data center [5] with the following characteristics.

First, the platform should guarantee that a cloud service is ***safe***, that is, 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, *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.

The second goal is that the platform should allow ***high concurrency***, *i.e.,* performing 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 platform to scale.

Third, the platform should guarantee that a service is ***robust*** in the presence of unexpected failures. Robustness ensures that failures in an orchestration procedure do not lead to undefined behavior or inconsistent states. This goal is challenging because of high volatility in the cloud environment, caused by software bugs, unstable hardware, transient network disconnections and power outages [9], etc. An orchestration procedure usually involves multiple state changes of distributed resources, any of which can fail due to volatility. For example, spawning a VM 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.

Our last goal is to guarantee ***high availability*** of cloud services and the platform that manages them. In the era of web-scale applications, unavailability of the cloud platform directly translates to loss of revenue and service degradation for customers. Based on our estimation of Amazon EC2's rate of VM creation (see §6), a mere 10-minute service disruption can result in not fulfilling 1,400 VM spawn operations in a single region. Such disruptions are unacceptable for mission-critical applications.

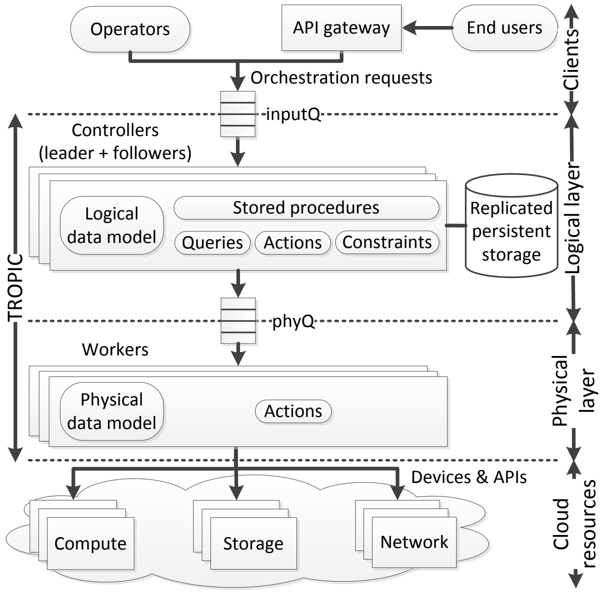

Figure 1: TROPIC architecture

2.2 Architecture

To achieve these design goals, we present the TROPIC platform, which performs *transactional* cloud resource orchestrations. Transactions provide ACID semantics which fit our design goals well: (i) *Safety* is enforced by *integrity constraints* in order to achieve transactional *consistency*; (ii) *Concurrency* is supported by a concurrency control algorithm that permits multiple transactions to execute in parallel while preserving the transactional behavior of *isolation*; (iii) *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; (iv) *High availability* is enabled by TROPIC's adoption of a decentralized architecture of replicated components.

Figure 1 depicts TROPIC's architecture. The orchestration requests of clients are initiated either directly by cloud operators (*e.g.,* for maintenance), or indirectly by the cloud end users via the API service gateway (*e.g.,* to spawn VMs). 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 §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 (§5). 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 (§4) are used to reconcile any inconsistencies.

TROPIC adopts a semi-structured hierarchical data model because it handles heterogeneity of cloud resources well. Each tree node is an object representing an instance of an entity. Each entity has associated expressions and procedures for inspecting and modifying the entity: queries, actions, constraints, and stored procedures. A *query* inspects system state in the logical layer and provides *read*-only access to resources. An *action* models an *atomic* state transition of a resource. 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, which is used to roll back a transaction (§3.1). *Constraints* specify service and engineering rules. Constraints support the *safety* property, and TROPIC automatically enforces them at runtime. Orchestration logic is specified as *stored procedures*, composed of queries, actions and other stored procedures to orchestrate cloud resources. [11, 18] provide more details on TROPIC's data model and programming constructs.

2.3 High Availability

The TROPIC architecture is designed with redundancy to avoid single point of failure. First, the components of TROPIC are connected via distributed queue services (`inputQ` and `phyQ`) that are highly available, which reduce the dependency between the components. Second, the persistent storage service is pluggable to any backend system that offers replicated, atomic key-value storage with strong consistency. We adopt ZooKeeper [16] to implement the queues and the storage service (§5).

TROPIC runs multiple controller instances. One of them is the leader, and the rest are followers, decided by a quorum-based leader election algorithm [21]. 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. 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 at any possible failure points, the new leader elected among the followers are able to restore the state of the controller at failure time, using state from persistent storage. Due to space constraint, we refer interested readers to the technical report [11] for details of the replicated state design and the idempotent recovery protocol.

3 Transactional Orchestration

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 §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 2 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`.

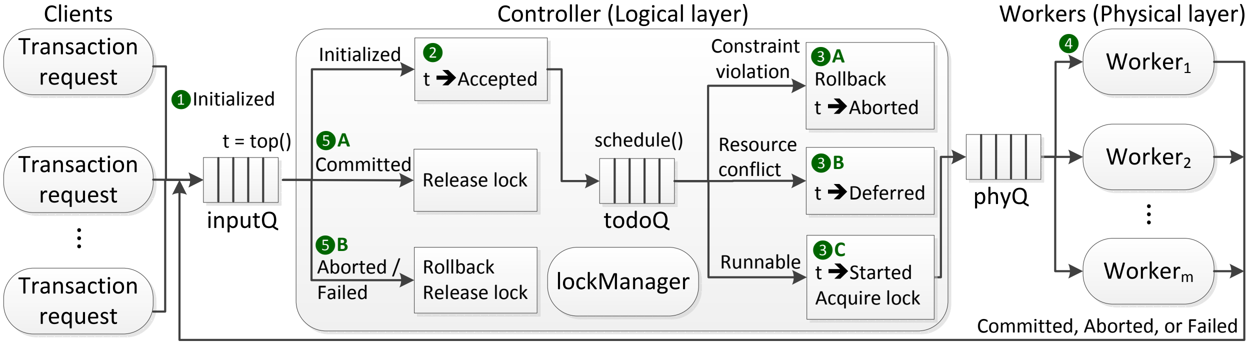

Figure 2: The execution flow of transactional orchestration in TROPIC.

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 §3.1.

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 (§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**).

3.1 Logical Layer Execution

The logical layer execution logic is depicted as Steps **3A**–**3C** in Figure 2. 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**).

3.1.1 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.

3.1.2 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) for concurrency control. This provides early detection of unsafe operations without touching actual physical resources. Table 1 shows an example transaction for spawning a VM. 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.

3.1.3 Concurrency Control

TROPIC adopts a pessimistic concurrency-control algorithm based on multi-granularity locking [20]. 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.

During its execution, a transaction *t* acquires write (read) locks on resource objects used by individual actions (queries). For instance, in table 1, 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)[1] on the ancestors of this object.

Besides acquiring locks on the resources used by transactions, additional locks are also acquired based on

[1] 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.

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

Table 1: An example of execution log for `spawnVM`

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 may break safety.

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 1 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[2], 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 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 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 [23].

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

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

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, `repair` is periodically invoked at a frequency customized by cloud operators, and `reload` is called when devices are added to or decommissioned from TROPIC.

Another source of error induced by resource volatility is the indefinite stalling of a transaction. This prevents the transaction from completing (either to a committed, aborted, or failed state) within a bounded period of time. To handle unresponsive transactions, TROPIC provides two mechanisms, by sending either *TERM* or *KILL* signals[3]. A *TERM* signal aborts the outstanding transaction via rollback with graceful cleanups at both the logical and physical layer (*e.g.,* undo actions, lock releasing) so that cross-layer consistency is maintained. A *KILL* signal makes the controller always immediately abort the transaction, but only in the logical layer. Any resulting cross-layer inconsistencies are then reconciled using `repair`.

5 Implementation

We have implemented a prototype of TROPIC. We briefly describe some of our implementation choices and outline a cloud service developed based on top of TROPIC.

We have chosen Python as our implementation language and the prototype of TROPIC is implemented in 11K lines of code. We use ZooKeeper [16] 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. We also unconventionally use ZooKeeper as a highly available persistent storage engine for storing the transaction states and logs.

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 §6 heavily use the logical-only mode to explore TROPIC performance under large scale of diverse cloud resources.

Using TROPIC we have developed a cloud service named *TCloud*. TCloud is deployed in a single data center and has features similar to Amazon EC2. It allows end users to spawn new VMs from disk images, and start, stop, and destroy these VMs. 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 [4] and DRBD [22] over the Linux logical volume manager (LVM) as storage resources, Xen [12] as compute resources, and Juniper routers as network resources.

6 Evaluation

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 3.0*GHz* Intel Xeon E5450 CPU processors and runs CentOS Linux 5.5, interconnected via Gigabit Ethernet. TROPIC runs one physical worker with multiple threads[4] 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.

6.1 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 by RightScale [2]. Specifically, we created a VM instance every 60 seconds and recorded

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

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

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 3 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 (§5) 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 4.

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%. 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. 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.

Transaction latency. Figure 5 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 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.

We further investigate the factors affecting performance bottlenecks of TROPIC under high load. Our experimental results [11] indicate that the dominant overhead comes from ZooKeeper API calls (*I/O*) instead of TROPIC logical layer simulation (*CPU*). To analyze scalability, we measure transaction throughput as the quantity of resources and transactions (load input) scales up. Our results demonstrate that TROPIC transaction throughput stays constant as the number of resources and transactions increases. This is due in part to our efficient implementation and optimizations. 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. Given our specific hardware, the maximum resource scale TROPIC can handle is 2 million VMs.

6.2 Safety

To evaluate the design goals of safety, robustness and high availability of TROPIC, 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. From the trace, we generate the hosting workload consisting of VM *Spawn*, *Start*, *Stop* and *Migrate* operations to mimic a realistic TCloud deployment (§5).

We first use the hosting workload to evaluate the overhead of enforcing safety constraints in TROPIC. We consider two representative constraints featured in TCloud: (1) VM type constraint: VM migration cannot be performed across hosts running different hypervisors; and (2) VM memory constraint: aggregated VMs memory cannot exceed the host's capacity. We focus primarily on per-transaction CPU overhead, since the bulk of constraint checking overhead happens at the logical layer. Our experimental results [11] show the logical layer overhead incurred in checking the above constraints is reasonably low (less than 10ms).

6.3 Robustness

In order to evaluate TROPIC's performance in guaranteeing robustness via transaction atomicity, we carry out two error scenarios, drawn from our experiences in deploying TCloud: VM spawning error and VM migration error. 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. In all our experiments [11], on a per-transaction basis, the logical layer operations complete in less than 9ms. This demonstrates that TROPIC is efficient at handling transaction errors and rollback.

6.4 High Availability

Finally, we evaluate the ability of TROPIC to recover in the presence of controller failures. We deploy TCloud running the hosting workload on the ShadowNet testbed using machines geographically dispersed across the US. Our results [11] demonstrate that TROPIC can recover quickly (within 12.5 seconds) to resume processing ongoing transactions in the presence of controller failures. No transaction submitted during the recovery time is lost. The recovery time is dominated by ZooKeeper's failure

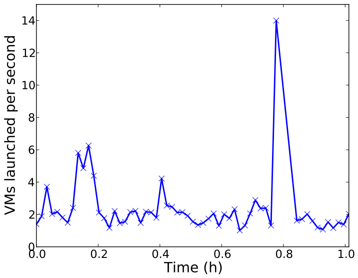

Figure 3: VMs launched per second (EC2 workload).

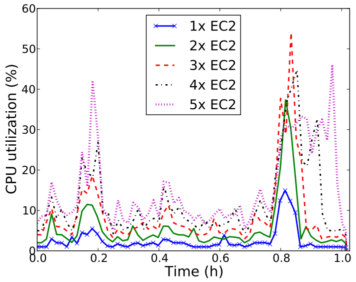

Figure 4: Controller CPU utilization (EC2 workload).

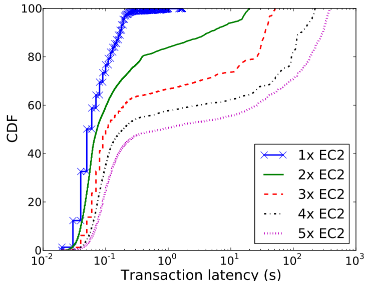

Figure 5: CDF of transaction latency (EC2 workload).

detection time as the heart-beat interval, suggesting that the recovery time can be reduced by adopting a more aggressive failure detection in ZooKeeper.

7 Related Work

Besides proprietary cloud orchestration platforms from commercial IaaS providers such as Amazon EC2 [1], open-source cloud control platforms, such as OpenStack [6] and Eucalyptus [3], 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 [20]. As a programming paradigm, it has also received more attentions recently from the systems community. These include transactional OS system call APIs [19], file systems [25], and user-level library [24] for lightweight data management. Puppet [7] is a data center automation and configuration management framework. Puppet has a transactional layer, but not in the sense of enforcing ACID properties. Autopilot [17] 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. 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.

8 Conclusion

This paper presents TROPIC, a 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. The evaluation of the TROPIC prototype shows its capability to support workload with high degrees of concurrency, provide high availability with low overhead, and ensure the transactional semantics of cloud operations.

9 Acknowledgments

This work is supported in part by NSF grants CCF-0820208, NSF CNS-0845552, and NSF CNS-1040672.

References

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

[2] Amazon Usage Estimates. `http://bit.ly/poIkqk`.

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

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

[5] How Big is Amazon's EC2? `http://bit.ly/rjy4Zp`.

[6] OpenStack. `http://openstack.org/`.

[7] Puppet: A Data Center Automation Solution. `http://puppetlabs.com`.

[8] Running 200 VM instances on OpenStack Compute. `http://bit.ly/n7LyMx`.

[9] Summary of the Amazon EC2, Amazon EBS, and Amazon RDS Service Event in the EU West Region. `http://bit.ly/r7aXXR`.

[10] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. `http://bit.ly/jFdKAR`.

[11] TROPIC: Transactional Resource Orchestration Platform In the Cloud. Extended Technical Report. AT&T TechDoc TD-100446. `http://www.netdb.cis.upenn.edu/papers/tropic_tr.pdf`.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[13] R. Bradshaw and P. T. Zbiegiel. Experiences with eucalyptus: deploying an open source cloud. In *LISA*, 2010.

[14] X. Chen, Z. M. Mao, and J. Van der Merwe. ShadowNet: A Platform for Rapid and Safe Network Evolution. In *Proc. USENIX ATC*, 2009.

[15] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39:68–73, 2008.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

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

[18] C. Liu, Y. Mao, J. Van der Merwe, and M. Fernandez. Cloud Resource Orchestration: A Data-Centric Approach. In *CIDR*, pages 1–8, 2011.

[19] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, 2009.

[20] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, third edition, 2002.

[21] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *LADIS*, pages 2:1–2:6, 2008.

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

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

[24] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.

[25] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling transactional file access via lightweight kernel extensions. In *FAST*, 2009.