Cloud Resource Orchestration: A Data-Centric Approach

Changbin Liu
University of Pennsylvania
3330 Walnut St, Philadelphia PA, USA
changbl@seas.upenn.edu

Yun Mao
AT&T Labs - Research
180 Park Ave, Florham Park, NJ, USA
maoy@research.att.com

Jacobus E. Van der Merwe
AT&T Labs - Research
180 Park Ave, Florham Park, NJ, USA
kobus@research.att.com

Mary F. Fernández
AT&T Labs - Research
180 Park Ave, Florham Park, NJ, USA
mff@research.att.com

ABSTRACT

Cloud computing provides users near instant access to seemingly unlimited resources, and provides service providers the opportunity to deploy complex information technology infrastructure, as a service, to their customers. Providers benefit from economies of scale and multiplexing gains afforded by sharing of resources through virtualization of the underlying physical infrastructure. However, the scale and highly dynamic nature of cloud platforms impose significant new challenges to cloud service providers. In particular, realizing sophisticated cloud services requires a cloud control framework that can orchestrate cloud resource provisioning, configuration, utilization and decommissioning across a distributed set of physical resources. In this paper we advocate a data-centric approach to cloud orchestration. Following this approach, cloud resources are modeled as structured data that can be queried by a declarative language, and updated with well-defined transactional semantics. We examine the feasibility, benefits and challenges of the approach, and present our design and prototype implementation of the Data-centric Management Framework (DMF) as a solution, with data models, query languages and semantics that are specifically designed for cloud resource orchestration.

1. INTRODUCTION

The efficiency and ubiquity of virtualization technologies on modern computer architecture have enabled widespread use of cloud computing. In particular, these Infrastructure-as-a-Service (IaaS) platforms provide cloud computing resources at the granularity of virtual machines (VMs) in both public cloud offerings, *e.g.*, Amazon EC2 [1], and enterprise-based private cloud instances. The latter is enabled by a variety of vendor offerings, or, indeed, an increasing array of open source cloud efforts.

While the basic IaaS model of providing VM instances on-demand, each with an operating system and set of applications of choice, remains the mainstay of cloud computing, more sophisticated cloud service abstractions are increasingly being offered and discussed in the community. Examples include combining cloud computing with virtual private network (VPN) technology to realize *virtual private cloud instances* [24], rapid cloning of VM instances to enable *cloud bursting* to deal with overload conditions [22, 18], *follow-the-sun* cloud services whereby VMs are migrated to be closer to where work is being performed [22] and using the ease of rapidly instantiating new VMs to provide cloud-based *disaster-recovery* services [23].

These more advanced cloud services share all the operational complexities associated with more basic cloud services, including resource allocation and placement, fault management, resource and service guarantees, image management, storage management etc. However, more sophisticated cloud services require the dynamic *orchestration* of resources to realize the service abstractions.

Cloud orchestration involves the creation, management and manipulation of cloud resources, *i.e.*, compute, storage and network, in order to realize user requests in a cloud environment, or to realize operational objectives of the cloud service provider. User requests are driven by the service abstraction and service logic that the cloud environment exposes to them. Examples of operational objectives that require orchestration functions include decreasing costs by consolidating physical resources and improving the ability to fulfill service level agreements (SLAs) by dynamically reallocating compute cycles and network bandwidth.

Cloud orchestration functions must be performed while dealing with service and operational concerns such as servicing large numbers of simultaneous user requests, enforcing policies that reflect service and engineering rules, and performing fault and error handling in a highly dynamic environment. Recognizing that similar problems are elegantly solved through well-known database methodologies and techniques, *i.e.* concurrency control, integrity constraint enforcement, and atomic transactions, we claim that an orchestration framework can and should incorporate database features: a formal, unified data model, a declarative query and constraint language, transactional semantics that provide atomicity, consistency, isolation and durability (ACID) properties, among others.

However, existing techniques for orchestration are rudimentary and meet few of the requirements listed above.

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

First, control interfaces to resources range from simply editing textual or XML configuration files, to low-level command-line interfaces (CLIs), to procedure-oriented application programming interfaces (APIs). None provides a high-level data model for accessing or modifying resources in a consistent, system-wide manner. Currently, orchestration tasks are typically written as procedures in imperative programming or scripting languages and interact directly with resources via the myriad control interfaces. Consequently, specifying and enforcing of policies on a particular class of resources (e.g., physical compute servers) requires touching all the procedures that modify that resource class, replicating the policy throughout the code base. Even worse, exception handling, whether unexpected errors from resources or global constraint violations, is entirely ad hoc. For example, in a multi-step orchestration task, if a resource unexpectedly throws an error in some step, a typical implementation will fail-stop, leaving the system in an inconsistent state, or worse, silently ignore the error and execute subsequent steps, resulting in undefined behavior. Lastly, implementing deadlock-free concurrency control over distributed resources is challenging, especially in scripting languages, but is necessary to avoid race conditions between concurrent tasks that utilize shared resources.

The Data-centric Management Framework (DMF) is our answer to cloud orchestration. DMF is a cloud orchestration programming and execution framework, in which cloud operations can be easily specified and executed while ensuring that service and engineering constraints are satisfied in a system-wide manner. DMF models resources and their state as structured data, and further separates this data into logical and physical layers to avoid system misconfiguration and illegal operations. Orchestration procedures in DMF are transactional and provide well-defined semantics for accessing and updating resource data and for handling exceptions. In particular, DMF can atomically commit a group of operations, maintain consistency between the logical and physical layers, prevent misconfiguration and illegal resource manipulations by evaluating constraints before physical deployment, and provide race-free concurrent transactions. Realizing the benefits of a data-centric approach to cloud orchestration is, however, not a trivial task. In particular, database semantics need to be maintained while performing orchestration functions in a highly distributed environment, applying operations to resources that are not inherently atomic, and on a platform that is inherently much more volatile than conventional database systems. We have developed a prototype of DMF and performed initial experiments in our emulated wide-area cloud computing testbed, using Xen virtual machines [9], DRBD storage [21], and Juniper routers [2].

To illustrate the value of DMF's data-centric approach to orchestration we begin with an apparently simple cloud feature—VM migration across the wide area network—and show the complexity of its actual deployment. We then describe the unique challenges that a data-centric cloud orchestration approach poses outside the existing database research literature, and how they are addressed in the design of DMF. We conclude with a description of our prototype system, and a discussion of how DMF may potentially influence the future design of resource controllers.

2. BACKGROUND AND CHALLENGES

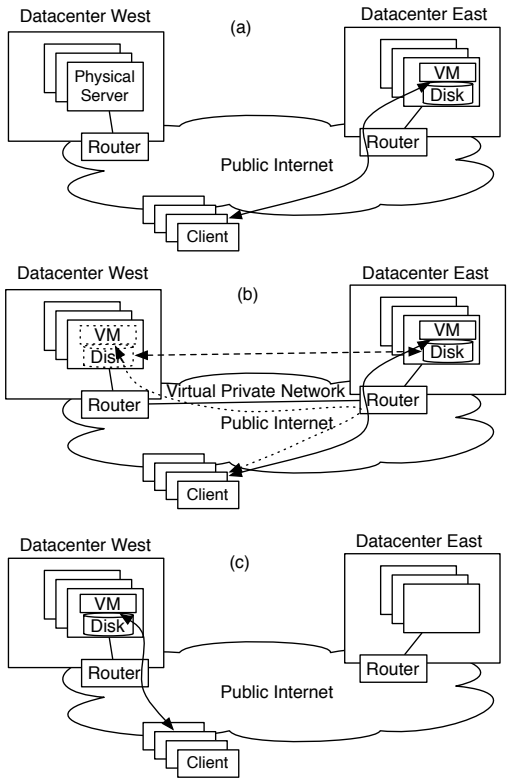

Figure 1: A multi data center, cross-domain cloud orchestration example

Figure 1 depicts our motivating example—a multi data center, cross-domain cloud orchestration scenario. The example illustrates the live migration of a VM from one cloud data center to another. This mechanism, for example, might be a part of the realization of a follow-the-sun service [22].

Figure 1 (a) shows the initial setup with a VM and its associated storage in Datacenter East and with clients accessing a service on the VM via the public Internet. The goal of our example is to "live migrate" the VM from Datacenter East to West to make it closer to the clients. Figures 1 (b) and (c) depict how this can be achieved. First a layer-2 VPN is established between the two data centers, and the storage associated with the VM is replicated to Datacenter West. [1] Then the VM itself is migrated from East to West.

During live VM migration, the IP address of the VM does not change, so existing application-level sessions are not disrupted [13]. As shown by the lower dotted line in Figure 1 (b), during migration, traffic between the clients and the VM follows a circuitous route via Datacenter East and the inter-datacenter VPN. Figure 1 (c) shows the final state after routing is updated so that traffic between the clients and the VM takes the direct path to Datacenter West.

Because routing in IP networks is asymmetric, there are two steps to updating network routing. First, traffic from

[1] In reality two separate VPNs might be used: A management VPN on which the storage and VM data is transferred between sites and a user VPN on which user traffic is carried.

the VM should use the router of its local data center for outgoing traffic, which can be readily achieved by changing the default route on the VM. Second, traffic from the clients to the VM needs to similarly follow the more direct path to Datacenter West. This can be achieved by advertising a more specific route to the migrated VM from Datacenter West.

The key point illustrated by this example is that realizing this dynamic service feature requires the orchestration of resources across different domains (compute, storage, and network), in a sequenced manner, and across geographically distributed data centers. Further, each of the operations described above is in fact relatively complex and thus susceptible to failure. Finally, a cloud orchestration platform would have to concurrently deal with potentially many similar orchestration requests.

Interestingly, similar problems in databases, such as making transactions atomic and controlling concurrent data manipulation have been solved through well-known methodologies and techniques. Solutions in the database literature, ranging from general semantics to specific algorithms, if not directly applicable, should at least inform solutions to related problems in cloud orchestration. To apply a data-centric approach to resource orchestration, however, several major differences from conventional data management must be addressed.

First, in resource orchestration, the main goal of updating data is to achieve a *side effect* of the state transition on a resource. For example, after setting a configuration of a network interface on a router to "enabled", one expects to be able to use the interface to send and receive traffic. Changing a replica of the configuration file outside of the router does not achieve the same result. The primary copy of the data on the device is the only copy that matters. In contrast, the purpose of writing data into a database is to be able to read it later. As a result, the data can be replicated to an arbitrary number of copies, on any media with any data format to serve the purpose. Therefore, data replication and caching must be used carefully in resource orchestration.

Second, an orchestration system manages data in the physical devices. Compared with data in the disk-based storage of databases, state in physical cloud resources is much more volatile. Given the scale, dynamics, and complexity of the cloud environment, crash or malfunction of a resource, such that its state changes, is sometimes inevitable. In addition, resource state may be tampered with, intentionally or even maliciously, via an out-of-band channel, circumventing the orchestration system.

Third, most state transitions in physical resources are expensive. For example, it takes from several seconds to minutes to commit a configuration on certain Juniper router models or perform a VM live migration, and not all state transitions are reversible. The orchestration system must take these factors into account when executing the orchestration procedures.

In the following sections, we will describe the design and implementation of DMF to address these challenges.

3. DATA-CENTRIC APPROACH

In this section we describe the design of Data-centric Management Framework (DMF) for cloud resource orchestration. Figure 2 depicts the architecture of DMF. In a nutshell,

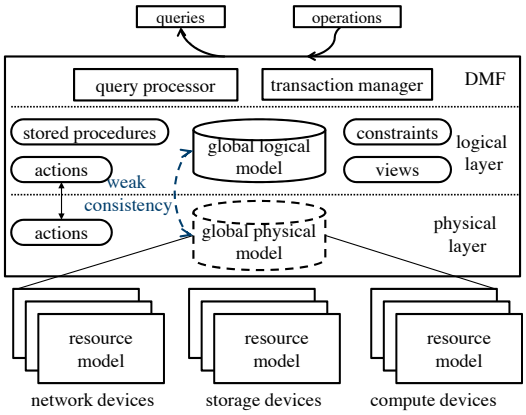

Figure 2: DMF architecture

DMF maintains a conceptually centralized data repository of all the resources being managed, which include compute, storage and network devices, as shown at the bottom of the figure. For every resource object, there are two copies that represent its state: the primary copy at the *physical layer* and the secondary copy at the *logical layer*. The primary copy is stored in the physical device such that read and write operations to the copy are translated into corresponding vendor-specific API calls. The secondary copy at the logical layer is an in-memory replica of the primary copy. DMF provides a weak, eventual data consistency between the two layers. Later in Section 3.3 we will explain why the separation of the two layers is necessary.

A user can specify *views* and integrity *constraints* in a declarative query language on top of the global data model. Views are used to reason about the current state in a system-wide fashion at a high level of abstraction. Constraints specify the policies that reflect service and engineering rules. Views and constraints can be materialized and are maintained by the *query processor*. *Actions* are the atomic operations that the resources provide, and are defined at both the physical and logical layers. A user can invoke transactions specified in *stored procedures* composed with queries, actions and other procedures to orchestrate cloud resources. They are executed by the *transaction manager* that enforces ACID properties.

We will now consider the DMF data model, language abstraction, transaction processing and consistency maintenance in turn.

3.1 Data Model

In DMF all resources are modeled as structured data. In this paper we only model non-volatile resource state, that is, state that changes as an effect of invoking orchestration operations on the devices, such as device configurations. Volatile state, like a server's CPU load or the latency between two routers, is read only and changes continuously. A data-centric approach to modeling volatile resource state has been studied in stream query processing systems [8, 14], and could be incorporated naturally into DMF, but is out of scope for this paper.

We adopt a hierarchical data model in which data is organized into a tree-like structure, as illustrated in Figure 4.

```
1  class LStorageRes(dmf.LogicalModel):
2      name = dmf.Attribute(str)
3      resRole = dmf.Attribute(StorageResRole)
4      @property # the primary key is attribute 'name'
5      def id(self): return self.name
6      @dmf.action
7      def setResRole(self, ctxt, resRole):
8          assert resRole in [StorageResRole.Primary,
9                             StorageResRole.Secondary]
10         origResRole = self.resRole
11         self.resRole = resRole
12         ctxt.appendlog(action="setResRole",
13                        args=[resRole],
14                        undo_action="setResRole",
15                        undo_args=[origResRole])
16 ...
17 class LStorageHost(dmf.LogicalModel):
18     hostname = dmf.Attribute(str)
19     resources = dmf.Many(LStorageRes)
20 ...
21 class LStorageRoot(dmf.LogicalModel):
22     hosts = dmf.Many(LStorageHost)
23     @dmf.view
24     def allResources(self):
25         return [(h.hostname, r.name, r.resRole)
26                 for h in self.hosts for r in h.resources]
27 ...
28 class LRoot(dmf.LogicalModel):
29     vmRoot = dmf.One(LVmRoot)
30     storageRoot = dmf.One(LStorageRoot)
31     @dmf.constraint
32     def vmAlwaysOnPrimary(self):
33         return [("VM not running on Primary Storage", vm)
34                 for host in self.vmRoot.hosts
35                 for vm in host.domains
36                 if self.storageRoot.hosts[host] \
37                     .resources[vm.storageRes].resRole
38                     != StorageResRole.Primary ]
39 ...
40 @dmf.proc
41 def migrate(root, vmName, srcHost, destHost):
42     resName = root.vmRoot.hosts[srcHost]\
43       .domains[vmName].storageRes
44     srcRes = root.storageRoot.hosts[srcHost]\
45       .resources[resName]
46     destRes = root.storageRoot.hosts[destHost]\
47       .resources[resName]
48     destRes.setResRole(StorageResRole.Primary)
49     root.vmRoot.migrate(vmName, srcHost, destHost)
50     srcRes.setResRole(StorageResRole.Secondary)
```

Figure 3: Sample code listing

Each tree node is an object representing an instance of an entity. An entity may have multiple *attributes* of primitive type, and multiple one-to-many and one-to-one *relations* to other entities, which occur as children nodes in the tree. An entity must have a primary key defined as a function of its attributes that uniquely identifies an object among its sibling objects in the tree.

The relational data model and SQL as the de facto standard in databases, are not entirely suitable for resource orchestration. For example, because resources are provided by multiple vendors, it is desirable to encapsulate state and provide modular interfaces in an object-oriented way. Heterogeneous data sources and limited APIs also favor the semi-structured or hierarchical data models, as exemplified in most integration middleware systems.

To illustrate the concepts of DMF, we use the example code in Figure 3, which is in a Python-style language. Although not complete, the code is very similar to the language implemented by DMF. The definitions of objects in the logical data model, in the classes `LRoot`, `LStorageRoot`, `LStorageHost`, `LStorageRes`, form the root node and storage-related branches. We will use this code sample throughout the remaining sections.

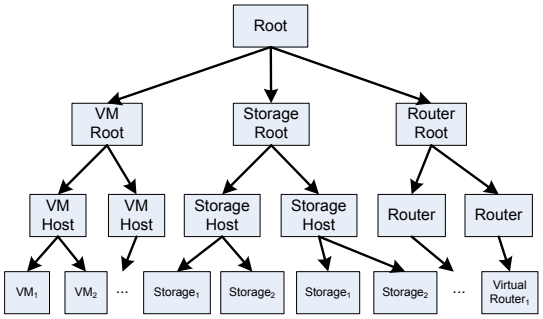

Figure 4: An example instance of the data model

3.2 Language

The programming language of DMF is a domain-specific language for query processing and data manipulation of structured data. It supports the following major constructs, *views, constraints, actions*, and *stored procedures*, which are elaborated below.

Orchestration tasks often require defining and querying views that, for example, inspect global network state, or that specify integrity constraints across multiple resources. Declarative query languages can express complex queries and constraints concisely and are highly amenable to optimization. Due to the tree-like data model, our query language syntax is similar to the FLWOR expressions in XQuery [7] with a subset of the XPath query capability. Because graph reachability queries are common in networked services, we also provide a transitive closure query operator and a more general fixpoint query operator that implements the semi-naïve evaluation algorithm [20]. A constraint in DMF is defined as a special type of view. It is satisfied if and only if it evaluates to an empty list. Otherwise, the list should contain information such as violation messages to help pinpoint the reasons behind the violations.

For example, on line 23 in Figure 3, the view `allResources` extracts attributes from nodes, returning a table of storage host names, and storage resource names and their roles. A more complicated constraint is defined on line 31, dictating that each VM must be running on a storage resource with primary role. [2]

For data manipulation, DMF provides the new concept of *action*, which models an *atomic* operation that is provided by a resource. Actions generalize the myriad APIs, ranging from file-based configurations, CLIs, to RPC-like APIs, provided by vendors to control the physical resources. Due to the varied semantics of these interfaces, DMF does not allow arbitrary insertions, deletions or updates to data, unless they are supported by the resources.

Due to the separation of the two layers, each action must be defined twice: one at the physical layer, which is transformed to the underlying API calls provided by the vendors, and the other at the logical layer, which describes the state

[2] A storage resource is usually a replicated data device that stores a VM image, and the storage resource associated with a running VM must have the resource role "primary".

transition in the data model. Preferably, an action is also decorated with a corresponding *undo* action. Undo actions are used to roll back transactions as described in Section 3.3. For example, on line 6 in Figure 3, the action `setResRole` is defined at the logical layer to change the resource role of a storage device. On lines 12–15, its corresponding undo action is written to the log: it is the same `setResRole` action, except with a different argument to set the resource role back to the original one.

3.3 Orchestration as Transactions

Transaction is the basic unit of orchestration in DMF. Transactions are atomic, consistent, isolated and durable and are realized in DMF as stored procedures. We describe how transactions are executed and how the separation of the physical and logical layers impacts ACID properties. We use the VM live migration procedure on lines 40–50 in Figure 3 as our example transaction. This corresponds to our previous cloud orchestration example in Section 2, with the difference that routing updating is omitted here for simplification.

A transaction is classified by its execution target layer as *logical-only*, *physical-only*, or *synchronized*, the latter meaning it is executed at both layers. Most orchestration tasks are synchronized transactions, because their purpose is both to satisfy constraints defined in the logical layer and to effect state change in the physical layer.

The execution of a synchronized transaction occurs in two phases. In the first phase, all operations in the transaction are executed at the logical layer, which include query evaluation and other actions. During the execution, an execution log is recorded.

In our example transaction, the queries on lines 42–47, access the relevant resource objects, and on lines 48–50, the 3-step orchestration (the destination storage resource role is set to primary, the VM is migrated to the destination, and the source storage resource role is set to secondary) is executed. The procedure is automatically enclosed in a transaction context (code omitted). Table 1 contains the execution log after the first phase.

At the end of the first phase, all integrity constraints are checked on the logical model. If any constraint is unsatisfied, the transaction is aborted, and the logical layer is rolled back to its state where the transaction began. This execution semantics guarantees that before a transaction begins and after it commits, the logical model is *internally consistent*, meaning that all integrity constraints are satisfied. The approach has the additional benefit that system misconfiguration and illegal operations are denied even before the physical resources are touched, thus avoiding the overhead of any unnecessary yet prohibitively expensive state transitions of physical resources.

If the first phase of a transaction succeeds, DMF executes the second phase at the physical layer. During this phase, since all state changes have already been handled in the logical model in previous phase, DMF simply re-plays all the actions in the execution log, executing the physical variant of each action. If all the physical actions succeed, the transaction returns as committed. This execution semantics guarantees that the transaction is *durable*, which means that the state of the physical resources have changed when the orchestration transaction is completed as committed.

If any action fails during the second phase, the transaction is aborted in both layers. At the logical layer, DMF rolls back to the original state, as it would if the first phase had aborted. At the physical layer, DMF selects all actions that have successfully executed, identifies the corresponding undo actions, and executes the undo actions in reverse chronological order. To achieve atomicity of transactions, each action in a transaction must have a corresponding undo action. If an action does not have an undo action, it can be executed stand-alone, but not within a transaction.

In our example, DMF executes the physical actions on the objects identified by their paths in the log. Suppose the first two actions succeed, but the third one fails. DMF executes the undo actions recorded in log, record #2 followed by record #1, to roll back the transaction. As a result, the VM is migrated back to the original location, and the destination storage resource role is reverted to secondary.

Once all undo actions complete, the transaction is terminated as aborted. If an error occurs during the undo phase, the transaction is terminated as failed. In this case, the logical layer is rolled back to an internally consistent state, however, there may be inconsistency *between* the physical and logical layers.

We note that in this execution model, all-or-nothing atomicity is always guaranteed at the logical layer. At the physical layer, it is enforceable if (1) each physical action is atomic, (2) each physical action is reversible with an undo action, (3) all undo actions succeed during rollback, (4) the resources are not volatile during transaction execution.

The first two assumptions can be largely satisfied at design time. According to our experience, most actions, such as resource allocation and configuration are reversible. [3] For (3), because an action and its undo action are symmetric, the undo action usually has a high probability of success given the fact that its action has been successfully executed in the recent past during the transaction.

3.4 Cross-layer Consistency Maintenance

DMF provides a weak, eventual consistency model for cross-layer consistency maintenance. A synchronized transaction maintains cross-layer consistency if the data is cross-layer consistent before the transaction starts, and the transaction can be atomically committed or aborted, under the assumptions described above.

However, in addition to failed undo actions, out-of-band changes to physical devices may cause cross-layer inconsistencies. For example, an operator may add or decommission a physical resource without making the change visible to DMF. Or an operator may log in to a device directly and change its state via the CLI. Furthermore, because resources are complex physical devices, a crash or system malfunction may change the resource's physical state without DMF's knowledge. Whatever the causes, these out-of-band changes occur in practice, and DMF must be able to gracefully handle inconsistencies.

Specifically, in DMF, inconsistency can be automatically identified when a physical undo 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 tree, no more synchronized transactions are allowed at that node or its children until the inconsistency

[3] Although not all physical actions can be undone. E.g., after a server reboots, there is no (easy) way to return the server to its pre-reboot state.

log record #	resource object path	action	args	undo action	undo args
1	/storageRoot/dest/vmRes	setResRole	[primary]	setResRole	[secondary]
2	/vmRoot	migrate	[vmName,src,dest]	migrate	[vmName,dest,src]
3	/storageRoot/src/vmRes	setResRole	[secondary]	setResRole	[primary]

Table 1: An example of execution log for VM migration

is reconciled.

To reconcile inconsistencies, logical-only and physical-only transactions are applied in a disciplined and reliable way. A DMF user can invoke a logical-only transaction to "reload" the logical model from the physical state, or invoke a physical-only transaction to "repair" the resources by aligning them with the logical model. Before a logical-only transaction commits, DMF checks all integrity constraints. If any constraints are violated, DMF aborts the transaction. To execute a physical-only transaction, at the beginning DMF first "reloads" the physical state into the logical layer, then executes the rest of the transaction as if it were a synchronized transaction.

DMF itself does not require a specific consistency maintenance schedule, leaving that schedule to the user. One can periodically invoke repair procedures, or in the case of a new device addition, for example, manually invoke a reload procedure to add that device to the system.

3.5 Summary

To summarize, DMF provides a data-centric programming and execution framework for transactional cloud resource orchestration. DMF provides ACID properties that closely resemble those of SQL databases at the logical layer with a strong consistency guarantee, and "best-effort" transactional semantics at the physical layer to cope with the limitations of the physical devices. A weak, eventual consistency model is used to reconcile the cross-layer differences.

4. PROTOTYPE

4.1 Implementation

We have implemented a prototype of DMF in Python. The primary goal of the prototype is to validate our hypothesis and explore the benefits of the data-centric approach.

The prototype system runs on a centralized server. The resource models, views, constraints, actions and stored procedures are all specified in corresponding Python programs following the code style in Figure 3. Most of the queries are expressed in the form of declarative list comprehensions and string-based path queries, all embedded in Python. The results of a view or a constraint can be optionally materialized to speed up subsequent queries.

We extensively use Python's meta programming and decorator techniques to hide implementation details, like how the model instances are stored, queried, and updated, how the logical and physical layers are separated, and how to execute, commit and abort transactions.

An obvious alternative to designing a new language would be to use an existing query language, like XQuery, which natively supports a tree-structured data model and a compact syntax for querying XML. XQuery's syntax for updates is cumbersome, however, and its XML-friendly syntax is not easily extended to support the DMF constructs described above. In addition, we wanted our network engineers, the first DMF developers, to be able to learn DMF easily. Python and its rich libraries are a familiar and expedient choice. In the end, DMF provides many of the benefits of XQuery, without burdening users with having to learn a new language syntax and library.

We fully implemented the transaction execution model described in Section 3.3. All transactions are serialized internally to provide isolation. A more sophisticated concurrency control model is our future work.

Specifically, we have implemented models for DRBD storage [21], Xen virtual machine hypervisor [9], and Juniper routers [2] in DMF. The three classes of resources provide very different APIs for orchestration. DRBD relies on the text based configuration files and a command-line interface to update resource roles and other state in the kernel. Xen provides its own APIs, but is also compatible with a more generic set of virtualization APIs provided by the `libvirt` library [3] that works with a variety of virtualization technologies, including Xen, VMWare, KVM, etc. Juniper routers use the NETCONF protocol [4] for router configuration. The configuration format is in XML with a predefined XML schema. Therefore, the process of building data models for DRBD and `libvirt` on Xen are entirely manual, such as designing entities and relationships, and wrapping their API calls to actions in DMF. In contrast, because Juniper already provides the XML schema, we are able to automatically generate models from the schema into the DMF modeling language (3197 models imported in total). We still have to manually write actions for operations like configuration commit, and constraints such as network protocol dependencies.

DMF exports an XML-RPC interfaces so that cloud orchestration applications can invoke stored procedures to realize cloud functions. We also have built an interactive command-line shell for rapid testing, debugging and demonstration purposes.

4.2 Preliminary Evaluation

To evaluate the performance of DMF, we experiment with a representative transaction—live virtual machine (VM) migration across a wide area network (WAN) as described in Section 2 and depicted in Figure 1, and measure the transaction execution time for both logical and physical layers. As mentioned above, this transaction consists of several actions, which include destination and source storage resource role setting, live VM migrating, and route updating.

We emulate the cloud environment that DMF controls and orchestrates on ShadowNet, our operational wide-area testbed [12]. In particular we create a slice in ShadowNet for DMF that consists of a server and a router in each of two geographically distributed ShadowNet locations, *i.e.* in 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 per-

data model	average	stdev
logical layer	0.069	0.026
physical layer	38.927	1.934

Table 2: Average and standard deviation of the transaction execution time (in seconds) for VM live migration.

formed between them and the to-be-migrated VM runs on this storage. The VM is allocated with 512MB memory in size, which is the dominant factor affecting the actual migration time.

To make the result more accurate, we in total migrate the VM 10 times and compute the average transaction execution time, as well as the standard deviation. Table 2 lists the experimental results. We note that, as expected, most of the transaction time is spent in physical layer, where actual management operations are performed on the physical devices, while the time for logical layer is negligible, demonstrating the efficiency of DMF implementation.

5. RELATED WORK

Database technologies are routinely used as part of system management and operations. One notable class of existing work, *e.g.* NetDB [10] in network configuration management, uses a relational database to store device configuration snapshots, where one can write queries to audit and perform static analysis of existing configurations in an offline fashion. However, NetDB is a data warehouse, not designed for network resource orchestration. Transaction processing [15] as another database technology also has received more attentions recently as a programming paradigm in system areas. Since Microsoft Windows Vista, the Kernel Transaction Manager (KTM) [6] enables the development of applications that use transactions, for instance, to implement transactional file system and transactional registry. When failure happens, transactions are rolled back to restore system state. Transactional OS (TxOS) [19] explores adding transactions to the 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.

There are several related frameworks proposed for management and orchestration for large-scale systems. Autopilot [17] is a data center software management infrastructure from Microsoft for automating software provisioning, monitoring and deployment. Similar to DMF, it has repair actions to deal with faulty software and hardware. Its periodic repair procedures maintain weak consistency between the provisioning data repository and the deployed software code. From the open-source community, Puppet [5] is a data center automation and configuration management framework using a custom and user-friendly declarative language for server configuration specification. In contrast to DMF, other than having different scopes, Autopilot does not provide a transactional programming interface. Puppet has a transactional layer, but not in the sense of enforcing ACID properties. Instead it allows user to visually examine the detailed operations before a transaction is submitted as a dry run. Once executed, a transaction is not guaranteed to be atomically committed.

Finally, in our earlier work, COOLAID [11] proposes a similar vision of data-centric network configuration management. COOLAID manages router configurations and adopts the relational data model and Datalog-style query language. In contrast, DMF has the significantly expanded scope of cloud resource orchestration where a diverse set of devices are managed. Facing several new challenges, DMF not only utilizes a different data model and query language to improve usability, but also re-architect the system to provide well-defined transaction executions on top of the separated logical and physical layers, in refined transactional semantics.

6. CONCLUSIONS

We presented DMF as the first attempt in adopting a data-centric approach that combines the uses of structured data models, a declarative query language and transactional ACID semantics to support cloud resource orchestration. We argue that these methodologies, mature and well-understood in databases, address many challenges imposed by emerging cloud computing services. Specifically, DMF offers transactional orchestration to atomically commit a group of orchestration tasks and provides well-defined semantics for unexpected error handling. Separating the resource data model into logical and physical layers, DMF maintains cross-layer consistency between the two, and further prevents the system from misbehaving by enforcing integrity constraints as policies. Ultimately only operational experience will tell how successful our approach is. However, we expect that, compared to conventional systems built in imperative languages, DMF's uniform declarative language for query and constraint specification will result in better code scalability and maintainability as the number and heterogeneity of resources increases. We have implemented a preliminary prototype of DMF and evaluated it within an emulated wide-area cloud environment that involves compute, storage, and network devices. The prototype demonstrates the feasibility of DMF design and its practical aspects.

Our future work includes: (1) exploring concurrency control algorithms to improve parallelism under simultaneous transactions from multiple clients; (2) decentralizing DMF architecture to realize high system scalability, reliability and availability; (3) deploying and evaluating DMF in geographically distributed large-scale data centers; (4) adopting query optimization techniques and incremental view maintenance [16] to improve performance; (5) and building more sophisticated cloud services in the framework to further validate our hypothesis and explore new opportunities.

7. APPENDIX: DEMONSTRATION

At the conference we will demonstrate live virtual machine (VM) migration across a wide area network (WAN) as described in Section 2, depicted in Figure 1 and evaluated in Section 4.2.

To demonstrate DMF's capability of orchestrating live VM migration, we run a game server in the VM which starts off in the Illinois (IL) data center. All game clients connects to this server via the IL router. For the demonstration purpose we assume that as more gaming clients connect to the server it becomes apparent that the data center in California (CA) would be more optimally positioned to serve them. As such we migrate the game server from IL to CA. This migration simulates a type of follow-the-sun cloud service in which the server migrates to the time zone where the major-

ity of clients are located, to reduce latency and improve their gaming experience. We run two processes to emulate game clients connecting respectively from IL and CA. A third process runs as DMF administrator and executes the migration transaction.

For the demonstration, the DMF control interface is a simple visualization tool, which displays the cloud resources and their state. The orchestration process proceeds as a transactional execution of (i) setting up the inter-data center VPN, (ii) establishing storage replication between the two data centers, (iii) performing VM migration, (iv) updating routing so that traffic to/from game clients follow the most direct path to the gaming server. As the migration transaction progresses, the visualization is updated to illustrate state change.

To demonstrate DMF transactional semantics, we emulate the failure of route re-configuration. Because route changing is the last step of the migration transaction, all previous steps are rolled back. The rollback operations are reported by the administration process.

If time permits, we will demonstrate another feature of DMF: consistency maintenance between logical and physical layers (described in Section 3.4) after the host machines have crashed and reset their state.

If there is a problem with the Internet connection at the venue, we will play a pre-recorded video for the demo.

Acknowledgments

The authors would like to thank Xu Chen, Carsten Lund, Boon Thau Loo, Emmanuil Mavrogiorgis, Edwin Lim, and the anonymous reviewers for their helpful comments on the earlier versions of the paper. This work is supported in part by the AT&T Labs Research summer student internship program and NSF grant CCF-0820208.

8. REFERENCES

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

[2] Juniper Networks. `www.juniper.net`.

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

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

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

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

[7] XQuery: the W3C XML Query Language. `http://www.w3.org/TR/xquery`.

[8] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan 2005.

[9] 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[10] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The cutting EDGE of IP router configuration. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, November 2003.

[11] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *Proceedings of the 6th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2010.

[12] X. Chen, Z. M. Mao, and J. Van der Merwe. ShadowNet: A Platform for Rapid and Safe Network Evolution. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[14] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, 2003.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[16] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1993.

[17] M. Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[18] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *EuroSys*, 2009.

[19] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

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

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

[22] J. Van der Merwe, K. Ramakrishnan, M. Fairchild, A. Flavel, J. Houle, H. A. Lagar-Cavilla, and J. Mulligan. Towards a ubiquitous cloud computing infrastructure. In *Proceedings of the IEEE Workshop on Local and Metropolitan Area Networks (LANMAN)*, May 2010.

[23] T. Wood, E. Cecchet, K. Ramakrishnan, P. Shenoy, J. van der Merwe, and A. Venkataramani. Disaster recovery as a cloud service: Economic benefits & deployment challenges. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2010.

[24] T. Wood, A. Gerber, K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. The case for enterprise-ready virtual private clouds. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2009.