Navigating the clouds with a MAP

Mark Shtern, Bradley Simmons, Michael Smit, Marin Litoiu
York University, Canada
{mshtern,bsimmons,msmit,mlitoiu}@yorku.ca

***Abstract*—This paper presents a management platform referred to as the Morphable Applications Platform (MAP) that supports the deployment and management of applications on a multi-cloud. Specifically, MAPs support two dimensions of adaptation not addressed previously: application-informed request routing and transposition across clouds. Specifically, a cluster is automatically moved from one datacenter to a second datacenter without service interruption and while preserving its state. A MAP represents the creation of a personal cloud fabric across a set of cloud providers. We have implemented a MAP on the public cloud and used it to deploy and manage an application.**

***Keywords*—cloud, multi-cloud, live migration**

I. Introduction

Cloud computing – particularly the provisioning of infrastructure-as-a-service (IaaS) where computation and storage resources are provided on-demand – is believed to offer potential benefits such as increased ability to scale, lower total cost of ownership (TCO), and the potential to improve availability. These resources are acquired on demand from a third-party (the *public cloud*), or can be managed within an organization in private data centers (the *private cloud*)[1].

Transitioning to a multi-cloud environment is the subject of research interest [1]–[3]. We identified two open research questions around the move to a multi-cloud. First, in order to take advantage of this environment, management decisions should be applied cross-provider; for example, the ability to dynamically move application clusters among a set providers as dictated by business requirements. The current accepted solution to this problem involves a complex workflow, including at least the allocation of new resources and the termination of existing resources, which does not necessarily maintain application state and is typically performed manually.

Second, deploying an application that spans multiple providers has the potential to increase charges if bandwidth usage is poorly planned. It is imperative to make more sophisticated (e.g., application-level) choices with regards to traffic routing, at the level of handling application requests. Automatic intelligent application-informed request traffic management reduces the impact of cross-provider traffic, which is charged at a higher rate than traffic internal to a datacenter.

The objective of our work was to develop a methodology and platform for deploying and managing a dynamically scalable application to a set of cloud providers, that above-and-beyond standard deployment and management features should enable the dynamic movement of a running application from one cloud to another in an on-demand fashion in addition to facilitating intelligent traffic management.

[1]IaaS provider and cloud provider will be used interchangeably for the remainder of this paper.

We achieved our research objective via the design and implementation of a platform referred to as the *Morphable Applications Platform* (MAP). One of the key features of a MAP is *transposition*[2], Figure 1, in which we move portions of whole applications across cloud providers without interrupting application service while preserving application state. A transposition algorithm was defined and implemented in prototype form.

Application-informed request routing enables the application to directly influence routing decisions. For example, requests may be routed based on geography, lowest latency, common backbone providers, load, or even based on the particular functions or features offered at one site but not another. An *Application Routing Service* is included in the platform prototype that provides a generic mechanism, facilitated through the use of plugins, to route traffic based on application-specific context. This general solution can be used to reduce costs associated with transposition by routing incoming requests over the cheapest available network connection.

The remainder of the paper is structured as follows. § II describes the basic function of a MAP, in particular how cross-provider management is achieved. § III introduces the transposition operation on a MAP and provides a brief description of its implementation and a brief overview of an experiment to verify its efficacy. § IV introduces our general, application-informed request routing framework and briefly introduces a case study. § V discusses open problems and further work. The paper concludes in § VI.

II. Morphable Applications on the Multi-cloud

Morphable applications are capable of both flexibility (adding/removing components, re-arranging organization of components) and mobility (spanning multiple providers, moving freely among providers and datacenters). A Morphable Applications Platform (MAP) imbues run-time scalable applications with these properties, deploying the application to enable morphing operations (e.g. add node, transpose, replicate, delete) and then generating and executing workflows for the operations. It manages requests sent to the application via an *Application Routing Service (ARS)* that manages the distribution of incoming requests to the best application ingress point.

Figure 1 shows a high-level view of a morphable application being managed by a MAP. The application is organized into *cluster*s, comprised of *nodes* that work together to provide application functionality. Each node consists of an outer instance (the instance provided by the cloud provider,

[2]We borrow the meaning of the word *transpose* from musical notation in which it means to "…write or play (music) in a different key from the original": http://oxforddictionaries.com/definition/english/transpose.

denoted as filled rounded rectangles) and a nested (inner) instance over which the MAP has full control (running the actual service, denoted with an abstract server image). In this case, a cluster consists of a load balancer and a set of application servers. One or more clusters are deployed to one or more datacenters from one or more providers; in this case, one cluster per one datacenter for two providers (with one cluster shown mid-transposition to a third provider; this and the other operations will be described in §III). The nodes in each cluster are connected by virtual channels (not shown). Each provider has an application routing component (top right of each provider) that routes incoming requests in coordination with the application routing service in the private datacenter (Section IV).

Supporting services are hosted in a private datacenter. A central datastore is used, as well as a central DNS service for load-balancing between the two clusters. The ARS coordinates routing decisions with the provider- and cluster-level routing components. The controller makes decisions and executes workflows, using an image store to manage available images.

To achieve the vision shown in Figure 1, in addition to the contributions described in this paper existing techniques must be deployed to enable cross-provider application management. At a low-level technical perspective, the differences in cloud providers accrue quickly, making transferring existing data, workflows, and configurations (like images, scaling policies, DNS entries...) to other providers difficult. Absent provider support, the ability to overcome boundaries is limited by the surrender of some control over the infrastructure required when using IaaS. The goal is to unify the base platform to which an application is deployed by using technology that adds abstraction or restores control. For example, *virtual channels* (e.g. VPNs) enable network configurations not natively supported by all providers (for example, maintaining a private IP even after moving an instance to another cloud provider, or multicasting). Abstraction layers[3] hide API differences and cloud metadata services (e.g. CloudHarmony.com or CloudyMetrics.com [4]) help understand the available resources. Cloud brokers [3] enable cross-provider allocation and low-level management of IaaS resources. Cross-provider federated monitoring of heterogeneous resources unifying metrics [5].

A valuable technique for abstraction is nested virtualization (e.g. [6]), the use of virtualization by an already virtualized resources. With two virtualization layers, a hypervisor running on bare-metal manages virtual machines that themselves run hypervisors (*containers*) with their own set of virtual machines. Xen-Blanket [7] allows nested virtualization on public cloud providers running Xen as a hypervisor (which includes Amazon EC2 and Rackspace Cloud).

Nested virtualization provides control over the nested VMs, including the ability to pause and migrate, and a measure of provider-independence as the images for the nested VM (which contain applications) are provider-independent [7]–[9]. It does impose some overhead depending on the operation (3% to 10% on real processing tasks in [6], [7]).

As the starting point for implementing the MAP, we used the Wrapt architecture, a realization of the AERIE reference

[3]E.g., Deltacloud. http://deltacloud.apache.org/

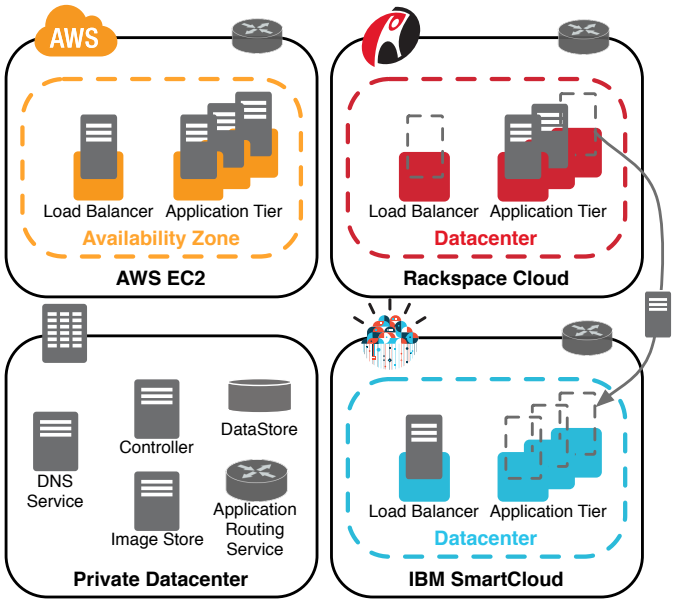

Fig. 1: Transposition of a cluster (dashed lines) from one provider to another in the multi-cloud.

architecture [10]. Wrapt allows application developers to deploy applications to the cloud without migration effort through the use of nested virtualization and virtual channels to provide an isolated, available, protected, cross-provider environment. Though we introduce enough of the platform to understand our implementation, the full benefits of the Wrapt architecture are not described here.

We used the existing implementation of the Wrapt architecture as the basis for our proof-of-concept implementations for the contributions described in this paper. Of particular relevance is the nested virtualization, which uses QEMU version 0.14.1 abstracted with virsh version 0.9.2. A guest QEMU image was created and stored on an encrypted partition. The operating system for the guest image was Ubuntu 11.10 64 bit, and it was configured to run J2EE applications with a default installation of Apache Server version 2.2.20, Tomcat version 6.32 and mysql 5.1.61. The load balancer ran Apache Server version 2.2.20 as a reverse proxy, and distributes requests among application server instances evenly. The central DNS service was lbcd version 3.30 [11]. All nodes and clusters are connected by VPN channels (OpenVPN 2.2.0). An automation solution was used [12] which provided the ability to both deploy and manage the runtime of the application. In this architecture, the supporting services and a database are deployed to a private domain while deploying web clusters to the public cloud.

III. Transpose, Delete and Replicate

One of the benefits of deploying application on clouds is elasticity. There is interest in designing mechanisms to effectively implement an application's elasticity policy [13] at runtime. However, it is also desirable to move from one cloud to another in various use cases:

- A hurricane is forecast to impact the Eastern seaboard of the United States, causing flooding and widespread

power outages. Proactive cloud users move their running application to other data centers in advance without downtime; reactive cloud users want to move their running application without loss of state before the bucket brigade transporting fuel to the generator is exhausted[4].

- A game development company wants to move their game engine from a beta site to the public cloud without disturbing the current players.
- An application running on Amazon EC2 attempts to reduce costs by dynamically jumping from one set of spot instances to another[5].
- A professor returning from sabbatical requires that a long running experiment he began running while away must be completed locally without interruption.
- A large research project[6] envisions future clouds as a tiered federation in which the lowest level of the tier is geographically close to the user and to which an application may move to improve performance characteristics.

At present, this operation is challenging at best and impossible at worst, without automated solutions, and may be an obstacle to cloud adoption [14]. An important feature of a MAP is to provide a solution to this problem, resulting in the introduction of the capability referred to as transposition, Figure 1. We also include the following cluster-level management capabilities for completeness: graceful-delete, delete and replicate. This full set of capabilities allows an application deployed on a MAP to be highly flexible and mobile, creating a personal cloud fabric across the entire set of providers.

Delete Operations A cluster of an application deployed on a MAP can be deleted in one of two ways: *graceful* or *forced*. The delete operation may be performed as long as the availability property of the deployed application is maintained. Specifically, this means that there must exist two or more clusters currently deployed[7].

The delete operation proceeds as follows. In the case of a graceful delete operation, verification must be made that all requests have completed. At this point, a termination signal is sent to each node of the cluster. Alternatively, in the case of delete no request for verification is made and the termination signals sent immediately to all nodes in the cluster. Note that to verify the completion of all requests a monitoring subsystem is consulted.

Replicate Operation The replicate operation takes as parameters the cluster to be replicated (including all relevant metadata; e.g., `clusterID`, `clusterName`, `location`), the destination (either within the current cloud or another cloud entirely), and the image for the particular cloud provider to which the cluster is being deployed (from which an instance can be created).

The operation proceeds as follows. For each member node of the cluster a virtual machine is instantiated at the destination based on the image. The virtual machine is configured identically to the source node including its nested virtual machine if applicable. The set of newly initialized nodes is then organized into a cluster. The new cluster then registers with the ARS indicating its availability for traffic.

Transposition Operation Through the transposition operation, the indicated cluster is moved spatially from one location to another. An important aspect of this operation is that it is continuous in nature (i.e., no explicit re-start command is ever issued). Further, the internal state of the cluster is fully preserved and no service interruption occurs. The main criteria that facilitates the successful execution of this operation is the logical de-coupling of physical resources and logical resources. More precisely, the use of virtual channels allows us to set up private addressing scheme that relates the nodes in a single cluster, and a container that allows for live VM migration allows preservation of state (§II).

A transposition requires the same information as the replicate operation: cluster, destination, image. It produces a new cluster in the destination. We assume that a transposition operation triggers an event to be raised by which deployed management policy [15], [16] sets could be dynamically adjusted. For example, different policies may need to be active on different providers to achieve a similar goal. A high-level view of this process is presented in Algorithm 1.

The idea motivating shutting down the elasticity policy (line 2) on related management domains acts as a guard against adding/removing resources in an interval of uncertainty[8] to eliminate thrashing in terms of resource acquisition and release. Following the notification to the ARS (line 3) all traffic is stopped. As described in §IV, when we notify the ARS about the completion of the transposition operation (line 13) the ARS must clear its cache of request routes.

It is important to note that the application of the replication operation to an application cluster on a MAP followed by a graceful delete operation on the original cluster allows for an operation that is not equivalent to transposition. The runtime state that is maintained during a transposition operation is lost when performed in this combination.

A. Implementation

A J2EE application that we have worked with previously [17] was deployed on a MAP, which is implemented as an extension of the Wrapt prototype as described in Section II. We deployed the MAP to two datacenters: us-east-1 (east) and us-west-1 (west). An active elasticity policy for the application server tier was established with thresholds on the mean CPU utilization across the application tier. When the mean CPU exceeded 65 an increase of two nodes was applied. Alternatively, when the mean CPU fell below 30 a decrease of one node was applied. Traffic was generated and directed at the application's URL. The traffic pattern was based on a modified subset of the FIFA workload information [18]. Following two autoscaling events transposition is initiated on the cluster

[4]http://status.squarespace.com

[5]Spot instances are offered at a reduced price, based in part on customer bids; however, they may be terminated abruptly.

[6]http://savinetwork.ca

[7]To fully remove a deployed application the undeploy operation must be used.

[8]Notice, that this is a policy decision that we recommend but it is not mandatory to the algorithm.

Algorithm 1: Algorithm used by the Autonomic Manager to transpose an application cluster across a set of clouds on a MAP.

Input: in-cluster, image, destination
Output: out-cluster

```
begin
    Shut down elasticity policy for related management domains (optional)
    Notify the ARS about the in-cluster to be transposed to the corresponding destination cloud
    Wait for the cluster to finish all initiated requests
    foreach virtual machine instance n ∈ in-cluster do
        Instantiate a virtual machine instance d based on the image on the destination cloud
        Configure all infrastructural services on virtual machine instance d
        Live-migrate specified processes from node n to d
        Terminate virtual machine instance n
        Add virtual machine instance d to out-cluster
    end
    Test for successful transposition of the out-cluster
    Notify the ARS about the completion of the Transposition process
    Turn elasticity policy back on for related domains (optional see above)
end
```

deployed on the eastern data center with the west datacenter as the target. We then tested application functionality before triggering a transposition. Service was maintained and the elasticity policy was temporarily disabled. After completion, the cluster originally on the east datacenter was located on the west datacenter with full function and preserved state, which we verified by inspecting a special status page. The detailed experimental results are omitted due to space limitations.

B. Related Work

Application transposition moves running application topologies among different clouds or datacenters, as requirements and/or workloads change. Migration is important [19] but existing work is at a low level of abstraction, enabling live-migration of individual guest VMs when one has hypervisor access, or process migration based on checkpointing [20]. We have increased the level of abstraction, exploring the automated migration of large complex application topologies composed of multiple processes with defined roles (eg, application server) and relationships across multiple cloud infrastructures. The transposition implementation demonstrated the feasibility of our approach, transposing without service interruption and while persisting application state. The Oracle 12c Pluggable Database feature is a potential approach to achieve for data what we have achieved for the application runtime; we intend to extend this work to include transposing data and will explore various approaches.

IV. Application-informed Request Routing

As described in the previous sections, application-informed request routing allows the management of requests in a multi-provider environment. In particular, the MAP has an Application Routing Service which gives the application control over which clusters receive which types of requests. Rather than expending effort ensuring there is a complete and accurate picture of the global state available to all nodes, we describe a dynamic, zero-sharing, scalable infrastructure for routing incoming application requests to the "best" application ingress point, for customizable definitions of "best" (perhaps geographical load balancing for static *and dynamic* content, or balancing to a cluster that provides translation to a particular language, or to a cluster with functions or features not offered at another site).

This section describes application-informed request routing in general, and provides a case study describing how this general approach has been used to reduce network traffic expenses in a cloud environment.

A. Application Routing Service

The Application Routing Service (ARS) implements application-informed request routing through a set of components and an algorithm for correctly routing incoming requests. The *client* is the end user of the application(s), sending requests to the application URI. A *coordinator* serves as the first point of contact for new clients, responsible for directing clients to the correct *application ingress point (AIP)*. There may be multiple application clusters in multiple datacenters, each capable of providing application services, each accepting requests through its respective AIP. Similarly there may be many coordinators. Every AIP is associated with at least one *councillor*; collectively, the councillors act as information brokers, intermediaries between the coordinators and application ingress points. Finally, *query routers* are connected in a networking lattice connecting them to other query routers and to a set of coordinators and to a set of councillors. The query router transmits requests between each {coordinator,councillor} pair as needed, and passes requests for other councillors to neighbouring routers. This ensures there are always multiple pathways between any two nodes, while adding logical separation of reliable message passing from the core request routing functions. It allows the components to be organized without global state; each component knows only which components are connected to them, plus any temporarily cached information. An overview of these components is shown in Figure 2 in a specific implementation, where the query routers are logically grouped in the councillor (C) nodes.

Certain deployment-specific decisions must be made when deploying the request-routing infrastructure. First, *application traffic routing rules* enable councillors to determine which associated AIP is a suitable choice to process a given request. What defines suitability depends on what the application is attempting to optimize, which can be simply (e.g. load) or complex (e.g. is the requested feature is available at the local application cluster). Second, a *static application routing policy* must be provided to the coordinator, which specifies a default request route that may not be optimal but is functional.

Given this set of components, an infrastructure is first assembled using a mix of discovery protocols and guided composition via a management framework. The query routers are assembled in a lattice by the framework. When councillors and coordinators are deployed, they are associated with a particular query router. They contact that query router and provide basic

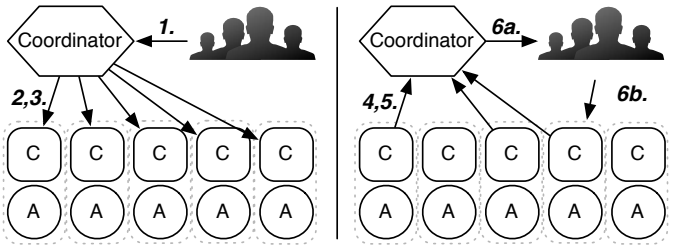

Fig. 2: Application-aware request routing; councillors and query routers (C) and AIPs (A) work with a coordinator to provide routing information to clients.

metadata information regarding what requests should be routed to them, etc. They will re-send this information periodically, or when asked to do so by a query router. Each has an ID; two (or more) coordinators/councillors with the same ID will have requests balanced between (among) them. When an application is deployed, it is associated with a councillor; it contacts this councillor to provide it with metadata.

Request routing (illustrated in Figure 2) proceeds as follows:

1. The client creates and sends a request. If only the application URI is known, the request is sent to that URI. If the best application ingress point for this client is cached at the client (optional), it sends a request directly; after a timeout, it defaults to sending to the general application URI.
2. Requests arrive at the coordinator. If a route is cached at the coordinator, the request is forwarded to the correct application ingress point. If a route is not cached, the coordinator first evaluates whether it is the best coordinator for that client (for example, the client may be better served by a coordinator in the same datacenter). If it is not, a redirect is sent to the client; if it is, processing proceeds. The coordinator formulates a Route Query (RQ) providing the parameters of the client request, and passes it to the query router asking that all councillors be sent the RQ.
3. The query router passes the RQ to any councillor components, and to any attached query routers. This repeats iteratively, with duplicates dropped. The RQ propagates through the networking lattice to all councillors in the infrastructure.
4. Upon receipt of the RQ, each councillor executes the application traffic routing rules to determine if any of the local application ingress points are suitable points for this request. If yes, it sends a response to the query router providing the AIP. If no, it sends nothing. In some implementations, the councillor may also send weights, balancing information, and the length of time for which this information is valid (TTL). If a councillor is overloaded, it may ignore incoming RQs.
5. The query router passes Route Responses to *all* coordinators. Coordinators who did not send an RQ may cache this information in case they are later contacted by a similar client.
6. The originating coordinator waits for responses for a fixed length of time, then chooses from the responses received (either based on the provided weights if implemented, or otherwise a load-balanced approach like round-robin). If no responses are received, the static application routing policy is used to identify the default route. The selected application ingress point is cached. The coordinator may send the original request to the chosen AIP on behalf of the client and return the result (when client caching is not in use), or send a redirect to the client for caching.

If the application environment changes, caches must be cleared (for this reason, client caches should have low TTL). In the event of application transposition (§III), the caches may be preserved partially, depending on the workflow generated for the transposition.

This organization of components offers several useful benefits. Each component is stateless and has no unique knowledge. This provides fault-tolerance. A failed coordinator will simply stop receiving requests, and requests will be processed by other coordinators. A failed query router will be routed around using the networking lattice. A failed councillor prompts the application ingress points to either launch a new councillor or to register with another existing councillor. No shared state or central coordination also ensures no single point of bottlenecking/failure. Multiple coordinators will share the handling of incoming requests. Routers can be organized in clusters, where requests from the associated components go to only one router in the cluster. Councillors can similarly be organized into clusters, with the router balancing incoming requests among the cluster members.

B. Case study

We implemented the key components of application-informed request routing and added them to our MAP implementation, which we then deployed to Amazon EC2 distributed across regions (us-east-1 and us-west-1). A set of application traffic routing rules was defined based on a business-to-business (B2B) data transfer use case, designed to minimize network traffic costs by routing traffic from one business's cloud resources on local networks to the cloud resources of the second business, where possible. We then implemented the steps described previously using a pluggable infrastructure developed in Java, and deployed a J2EE application as the server in the client-server interaction. We generated workload to send to this application based on the FIFA workload information [18] scaled down to not overload our application tier. We used a custom monitor [5] to measure whether traffic was sent on the private (free) or public (expensive) network, and determined that our approach resulted in cost savings for the provider, correctly routing network traffic to the low-cost network when possible. The detailed experimental results are omitted due to space limitations.

C. Related Work

Intelligent network routing is a well-known research topic and there have been many different approaches proposed to address it (eg, DiffServ [21], MPLS [22] and NGSON [23]). Most approaches address routing on physical networks. The intelligent routing problem has now been shifted into a novel context, namely clouds. NGSON [23] is positioned between end users and cloud providers and focuses on traffic delivery outside of the cloud. The intra/inter-cloud routing problem is unaddressed; the lack of knowledge about the underlying

infrastructure and the lack of direct control mechanisms necessitates a re-imagining of the problem and a novel solution. Our approach enables cost savings and performance improvements by making use of application-level request routing. (For example, routing requests over the Amazon private network can reduce bandwidth charges.) Other optimizations can be implemented using our generalized application informed routing methodology.

V. Discussion

Based on our experience designing and implementing a MAP, we have identified several areas that merit further investigation.

Image management: One of the most important open problems for spanning cloud providers is image management. Creating images from which to launch instances is a time-consuming process that must be performed for each cloud provider, and often for each region of each cloud provider. Updates to these images are similarly complex. Preliminary work in [6], [10] demonstrates the use of nested virtualization, which mitigates this problem, though the outer instance running the nested hypervisor suffers from the same issues.

Pattern-based adaptation: A common approach to managing complex systems is self-adaptive runtime management [24], which will change the deployment from what the deployer specified to what the adaptive system calculates is necessary to meet business objectives. The ongoing drift of realized systems from high-level models is a known problem in software engineering [25]. It merits further study in this context. We suspect that future adaptive systems may address this problem by adapting at the model level, translating these model-level changes to the realized system.

Hierarchical clouds: Our application-informed request routing enables the use of request routing to bring users to a "local" cloud: one with lower latency, geographic relevance, and/or reduced bandwidth charges. Another approach to this problem involves the hierarchical composition of clouds, with the lowest level of the hierarchy local to the user. Applications deployed to the hierarchical cloud can be migrated among the tiers depending on the application and user requirements and the resource constraints at each tier. Cloud providers have data centers in different geographic regions, so the ordering of the tiers may vary based on the location of the region; for example, a client in Dallas would see a Rackspace cloud tier as "local", whereas a client in southern Maine would be closest to the Amazon Eastern US cloud tier.

Bursting in nested virtualization: Cloud bursting refers to the temporary allocation of cloud resources in response to non-permanent increases in workload. The use of nested virtualization offers an approach to cloud bursting: because VMs are typically allocated to physical hosts without over-provisioning, physical hosts with idle VMs have spare capacity that could be temporarily deployed to assist an overloaded application.

Cross-provider networking: When deploying a multi-cloud, it is important to understand the network links among the datacenters involved, including capacity, speed, and congestion. Traffic on the public internet will not compare to node-to-node traffic inside a datacenter, but important differences in backbone internet service providers can impact cross-provider performance. For example, our anecdotal testing shows that a Rackspace datacenter in Dallas has higher transfer speeds to the Amazon US East datacenter (1.99 M/s) than to the Amazon North California datacenter (1.60 M/s), despite being within a few hundred kilometres of equidistant from both. Further exploration of these issues are required.

Addition of a merge operation: It was observed in our transposition implementation, that following transposition two independent clusters were running on the west datacenter simultaneously. One cluster had much higher mean CPU utilization than the other. This disproportionate utilization of resources occurred because both clusters have a different number of nodes. This suggests a need for additional operations on MAPs. For example, it may make sense to have a merge operation that will allow for consolidation of a pair of clusters.

Limitations: In the current implementation of our prototype we utilize QEMU as a container to achieve our uniformity requirement. Unfortunately, our version of QEMU has several performance limitations. a) it is single threaded, b) there are incompatibilities (especially live migration) between various versions c) it incurs a high CPU load on boot-up. Our prototype is designed to be flexible (i.e., it utilizes `virsh`) which allows for alternative containers to be used (e.g., XenBlanket).

While our experiments are run across two regions (i.e., multiple datacenters) they are still only run within one provider's domain (i.e., Amazon). However, the management framework we use is capable of cross-provider resource allocation [3], and our use of nested virtualization allows our application layers to be platform independent [7].

Our approach requires that no hard dependency exists between the pair of clusters involved in a transposition operation[9]. For situations where this dependency exists a potential solution is to combine the pair into a single super-cluster and then perform the transposition on a pair of super-clusters.

Applications which include clusters whose member nodes have hard-realtime requirements cannot be transposed.

While presently our work assumes that all clusters are identical in functionality, the algorithms are sufficiently generic and experimenting with heterogeneous clusters is an interesting and likely avenue of future research. We also assumed all nodes in a cluster are deployed to a single cloud provider; interesting research problems arise when this requirement is relaxed.

VI. Conclusion

In this paper we introduced a management platform MAP that supports application adaptation on multi-clouds. This is a significant contribution and step forward as MAPs offer the creation of a personal cloud fabric across a set of cloud providers. Specifically, their novel contributions are that they support adaptive management along two dimensions of adaptation: application-informed request routing and transposition across clouds.

[9] An example of this may be two databases in a high-availability mode.

ACKNOWLEDGMENT

This research was supported by IBM Centres for Advanced Studies (CAS), the Natural Science and Engineering Council of Canada (NSERC) under the Smart Applications on Virtual Infrastructure (SAVI) Research Network, Ontario Centres of Excellence (OCE) and Amazon Web Services (AWS). Special thanks to Cornel Barna for use of his management framework.

REFERENCES

[1] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, "Blueprint for the intercloud - protocols and formats for cloud computing interoperability," in *Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 328–336.

[2] R. Buyya, R. Ranjan, and R. N. Calheiros, "Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services," in *ICA3PP (1)*, 2010, pp. 13–31.

[3] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski, "Introducing STRATOS: A cloud broker service," in *IEEE 5th International Conference on Cloud Computing (CLOUD)*, 2012, pp. 891 –898.

[4] M. Smit, P. Pawluk, B. Simmons, and M. Litoiu, "A web service for cloud metadata," in *IEEE Congress on Services*. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 24–31.

[5] M. Smit, B. Simmons, and M. Litoiu, "Distributed, scalable monitoring of heterogeneous clouds using stream processing," *Future Generation Computer Systems*, To Appear, 2013.

[6] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: design and implementation of nested virtualization," in *The 9th USENIX conference on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.

[7] D. Williams, H. Jamjoom, and H. Weatherspoon, "The Xen-Blanket: Virtualize once, run everywhere," in *7th ACM European Conference on Computer Systems (Eurosys)*, Bern, Switzerland, April 2012.

[8] D. Williams, E. Elnikety, M. Eldehiry, H. Jamjoom, H. Huang, and H. Weatherspoon, "Unshackle the cloud!" in *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)s*, Portland, OR, June 2011.

[9] Z. Pan, Q. He, W. Jiang, Y. Chen, and Y. Dong, "Nestcloud: Towards practical nested virtualization," in *Proc. Int Cloud and Service Computing (CSC) Conf*, 2011, pp. 321–329.

[10] M. Shtern, B. Simmons, M. Smit, and M. Litoiu, "An architecture for overlaying private clouds on public providers," in *8th International Conference on Network and Service Management, CNSM 2012, Las Vegas, USA*, 2012, pp. 371–377.

[11] R. J. Schemers, III, "lbnamed: A load balancing name server in Perl," in *Proceedings of the 9th USENIX conference on System administration*. Berkeley, CA, USA: USENIX Association, 1995, pp. 1–12.

[12] C. Barna, B. Simmons, M. Litoiu, and G. Iszlai, "Multi-model adaptive cloud environments (MACE)," November 2011, http://www.ceraslabs.com/projects/management-services-for-cloud-computing.

[13] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implement an elasticity policy," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, 2011, pp. 716–723.

[14] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[15] M. Sloman, "Policy driven management for distributed systems," *J. Network Syst. Manage.*, vol. 2, no. 4, pp. 333–360, 1994.

[16] J. Strassner, *Policy-based network management: solutions for the next generation*. Morgan Kaufmann, 2003.

[17] C. Barna, M. Shtern, M. Smit, V. Tzerpos, and M. Litoiu, "Model-based adaptive dos attack mitigation," in *Proceedings of 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2012, pp. 119–128.

[18] M. Arlitt and T. Jin, "A workload characterization study of the 1998 World Cup web site," *IEEE Network*, vol. 14, no. 3, pp. 30–37, 2000.

[19] M. Sethi, N. Sachindran, M. Soni, M. Gupta, and P. Gupta, "A framework for migrating production snapshots of composite applications to virtualized environments," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2011, pp. 578–585.

[20] D. Nguyen and N. Thoai, "EBC: Application-level migration on multi-site cloud," in *2012 International Conference on Systems and Informatics (ICSAI)*. IEEE, 2012, pp. 876–880.

[21] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," Internet Requests for Comments, RFC Editor, RFC 2475, December 1998.

[22] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," Internet Requests for Comments, RFC Editor, RFC 3031, January 2001.

[23] C. Shan, C. Heng, and Z. Xianjun, "Inter-cloud operations via NGSON," *IEEE Communications Magazine*, vol. 50, no. 1, pp. 82–89, 2012.

[24] Y. Brun, R. Desmarais, K. Geihs, M. Litoiu, A. Lopes, M. Shaw, and M. Smit, "A design space for self-adaptive systems," in *Software Engineering for Self-adaptive Systems II*, ser. Lecture Notes in Computer Science, vol. 7475. Springer Berlin / Heidelberg, 2013, pp. 33–50.

[25] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models," *SIGSOFT Softw. Eng. Notes*, vol. 20, no. 4, pp. 18–28, 1995.