Title

Routing-as-a-Service (RaaS): A Framework for Tenant-Directed Route Control in Data Center

Journal

IEEE/ACM Transactions on Networking, 22(5)

ISSN

1063-6692

Authors

Chen, Chao-Chih
Yuan, Lihua
Greenberg, Albert
et al.

Publication Date

2014

DOI

10.1109/tnet.2013.2277880

Routing-as-a-Service (RaaS): A Framework For Tenant-Directed Route Control in Data Center

Chao-Chih Chen Lihua Yuan Albert Greenberg Chen-Nee Chuah Prasant Mohapatra

***Abstract*—[1] In a multi-tenant data center environment, the current paradigm for route control customization involves a labor-intensive ticketing process where tenants submit route control requests to the landlord. This results in tight coupling between tenants and landlord, extensive human resource deployment, and long ticket resolution time.**

We propose Routing-as-a-Service (RaaS), a framework for tenant-directed route control in data centers. We show that RaaS-based implementation provides a route control platform where multiple tenants can perform route control independently with little administrative involvement, and landlord can set the overall network policies. RaaS-based solutions can run on commercial off-the-shelf (COTS) hardware and leverage existing technologies, so it can be implemented in existing networks without major infrastructural overhaul. We present the design of RaaS, introduce its components, and evaluate a prototype based on RaaS.

***Index Terms*—Computer Networks, Computer Network Management, BGP, Border Gateway Protocol, Routing-as-a-Service, RaaS**

I. Introduction

Data center is a key infrastructure for on-line service providers (OSP) to provide always-on and responsive services to end-users. Typically consisting 1,000's to 100,000's of servers, data centers are designed to handle tremendous computations, large storage, and quick service delivery. However, the computational resources in a data center are not used monolithically. Often, the resources are multiplexed between different **tenants** – clients of the data center resources – so they can simultaneously perform computations, store data, and provide services to end-users.

In this paper we focus on *routing as a service to tenants*. Recent cloud computing infrastructures such as Amazon's EC2 [2] show promising direction in tenant-empowerment; for example, EC2 grants user control of IP-to-Virtual Machine (VM) binding without administrative involvement. Extending this notion, routing-as-a-service to tenant promotes the idea that tenants can programmatically re-route traffic for their services. For example, instead of a single server handling user traffic, a tenant might want to load-balance incoming traffic across 10 machines. The traditional paradigm for achieving such per-tenant routing customization involves a labor-intensive ticketing process, which we outline below.

Figure 1 shows a typical ticketing process for routing customization. A tenant first submits a request for routing customization (a "ticket") to a ticket distribution system, upon which a **landlord** (e.g., a network administrator) is assigned the ticket. After rounds of clarification between the tenant and landlord, the landlord sets up routing policies. Further clarifications might be required if the installed routing policy is unsatisfactory to the tenant. Finally, when both the tenant and landlord are content with the routing policy, the routing customization request is considered fulfilled and the ticket is mark resolved.

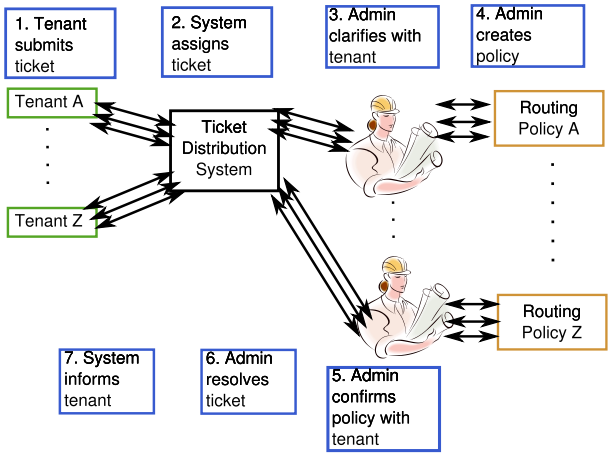


Fig. 1: Ticketing process.

The following problems are common with this paradigm.
Labor intensive process: Many of the steps in Figure 1 involve manual intervention, which burdens both the tenants and landlord, but more so the landlord because it takes away time the landlord can spend improving and maintaining the network. While tolerable when the request volume is small, such a system is unsustainable as the volume and variety of customization increases.
Lack of automated control: The traditional paradigm takes away tenants' ability to automatically control routing to their services. Therefore, tenants often have to submit routing policies that satisfy a certain class of scenarios (e.g., the average/worst case scenario). In addition, reacting fast to changes in this paradigm means more tickets inundated to the ticket distribution system, further overwhelming the landlord.
Long ticket resolution time: As a byproduct of having a labor-intensive process, the landlord might not resolve the tickets quickly. The resolution process could take days if tenants and the landlord communicate via e-mail, or weeks if in-person meetings are required. Such a delay might not be acceptable if tenants desire a quick response to changes in the network.

This paper proposes the Routing-as-a-Service (RaaS) framework. RaaS promotes automated route control to tenants while

This work was supported in part by the National Science Foundation through the grant CNS-0716741.

[1]Part of this work has appeared in [1].

retaining the landlord's authority in setting the overall network policy. The RaaS architecture consists of **MultiSpeakers**, **Controllers**, and **Tenant Applications**, with the former two under landlord's control and the latter maintained by the tenants.

Our contributions are the following:

- We propose a framework that provides a programmatic environment for tenants to use routing as a service, while reducing landlord's management effort, resulting in reduced personnel cost (Section III).
- We build a prototype of RaaS (Section IV) based on commercial-off-the-shelf (COTS) components and existing protocols, demonstrating that RaaS is immediately applicable to data center networks.
- We conduct detailed performance micro-benchmarks of RaaS, in terms of its processing delay, memory consumption, network overhead and success rate in serving requests, showing that it does not cause overwhelming burden on the network (Section V).
- We present two scenarios, *load balancing* and *workload migration*, where the RaaS framework can enable routing customization and automation. In both scenarios we find that tenants can easily specify their own routing policy via the tenant application, without knowing the innards of the network.

The paper proceeds with a high-level discussion of RaaS' approach in Section II, a system overview in Section III and the implementation in Section IV. An implementation based on RaaS, along with a theoretical model for the service availability, are evaluated in Section V. Related work are discussed in Section VII and we conclude the paper in Section VIII.

II. Overview of RaaS Approach

This section defines the resource provisioning problem outlined in Section I and the challenges in overcoming the provisioning problem. Based on the challenges presented, a high-level approach to the solution is presented, with the details left to Section III.

A. Resource Provisioning Problem

The resource provisioning problem that RaaS solves is the following: given that tenants have a set of **services** and **resources** that handle the service workload, allow multiple tenants to simultaneously change the resources that will handle their services, while avoiding the problems of *labor intensive ticketing process*, *lack of automation*, and *long ticket resolution time*. **Services** in the context of RaaS are externally visible offerings, such as search engine or multimedia streaming; **resources** are servers made available to tenants to handle the workload generated by the services. Since the resources are the physical servers, several tenants could use them at the same time. We will refer to both the services and resources by the IP addresses assigned to them, with the service IP address being the **tenant IP address** (TIA) and the resource IP address the **resource IP address** (RIA).

B. RaaS Approach

The RaaS approach to the resource provisioning problem is to enable *automation*. By automating the ticket resolution process, administrators are relieved from handling the routing requests, and tenants can re-route traffic to their resources more quickly. RaaS achieves automation by exposing a set of application programming interfaces (APIs), whereby tenants can use their own applications to submit routing requests, and quickly receive replies back on the success of their requests. Under this environment, tenants can automatically control routing of their services, and the ticket resolution process is automated and shortened. Achieving automation requires overcoming three design obstacles: practicality, policy control, and multi-tenancy. We outline the three problems and RaaS' approach to solve these problems below.

C. RaaS Practicality

One design requirement for RaaS is that it can be deployed without major infrastructural overhaul. This constrains us to consider methodologies and tools that are widely available, and prohibits us from modifying the router in any way. To achieve this, RaaS leverages well-established routing protocol as the platform for route control, similar to the approaches taken in [3]–[5]. Tenants' routing customization requests are then realized by manipulating the routing protocol. RaaS introduces a component called **MultiSpeaker** to interact with and manipulate routers running the routing protocol (Section III-B).

Using a well-established routing protocol also has two benefits: first, routing protocol already enables the main functionality that RaaS requires (i.e., distributing routes to services), so there is no need to re-invent the wheel. Second, a well-established routing protocol is likely to be implemented on all commercial routers, so RaaS is not dependent on a specific vendor's router. These two traits imply that RaaS can be practically implemented in any existing network.

D. Hierarchy of Control

Giving tenants control over routing raises a new source of tension between the landlord and the tenants. While giving tenants control over routing relieves landlords from handling tenant requests, giving tenants too much control could be disastrous, since tenants generally do not understand the overall network policy. This means tenants could route their services to resources not intended for them, or even worse, change the routing of other tenants' services.

The proper balance would be to enforce *least privilege*, a concept well-known in computer security. Least privilege in RaaS' context means tenants should have only the minimum ability they need to control routing to their services; the overall network policy should remain in the control of the landlords. RaaS achieves this by delegating the software component that handles tenant requests, the **Controller**, to the landlord. The Controller can then examines each incoming tenant request and reject those that violate the policies set by the landlord (Section III-C).

The task division between the Controller and MultiSpeaker also creates abstraction layers where administrator can hide information about their network, making RaaS not only practical but more acceptable to administrators. The physical topology is shielded from users and Controller via MultiSpeaker, where routing requests can be issued by simply knowing the MultiSpeaker address and API. IP topology can also be hidden away from users by assigning tenants with virtual server IP addresses, and only map these virtual server IP addresses to physical server IP addresses when the routing request is accepted. In the rest of the paper we assume server IP addresses will be used, with an understanding that hiding the IP topology is also possible.

E. Multi-Tenancy Tolerance

Another design requirement for RaaS is the ability for an instance of RaaS to handle requests from multiple tenants. In addition, tenants should be able to make requests simultaneously and independently of other tenants, and RaaS should quickly respond to each tenant's request. To make RaaS capable of handling multiple tenant requests in a timely manner, Controller and MultiSpeaker are made to be scale-out, so redundant components can be brought up when the tenant request volume exceeds available capacity. A scale-out architecture is possible because redundant Controllers and MultiSpeakers do not have heavy dependency amongst themselves, so they can service different tenant requests independently of each other.

F. Illustrative Examples

To tie the ideas presented together, consider the *traffic migration* example outlined in Figure 2. Tom and Alice are two tenants that are making requests, where both want to move traffic destined for their service from their initial resource to their new resource. Note that the new resource for Tom and Alice happens to be the same RIA. To make this transition, both Tom and Alice make a request, via their application, to the Controller. The Controller inspects the requests to ensure that Tom and Alice are making a request to their service only, and they are allowed to use the RIA outlined in the request. Once the Controller approves the requests, it replies to Tom and Alice that the request was approved, and notifies the MultiSpeaker. The MultiSpeaker, upon receiving the Tom and Alice's requests, interacts with routers via the routing protocol to change routing for their TIA from the initial RIA to the new RIA. If the lone Controller in the figure is overwhelmed with requests, another Controller can be brought online and some tenant requests can be diverted to the new Controller. Besides the network policy set by the landlord, the new Controller does not need any other information, so the new Controller and the original Controller can serve tenant requests independently of each other. Similar expansion is also possible for the lone MultiSpeaker because new MultiSpeakers do not need to know about requests processed by the existing MultiSpeaker.

Besides the traffic migration usage scenario, another practical scenario possible under RaaS is *load balancing*, shown in Figure 3. Here Tom and Alice want to add additional resource to serve their load, so they go through the same process as before, except now they request for more resource. Once approved by the Controller, MultiSpeaker will enable the added resources (RIA2) by instructing the routers to insert RIA2 without removing their initial resource.

III. RaaS Design

This section presents the design of RaaS, and details the components that enable tenant-directed route control. The RaaS framework consists of three components: **MultiSpeakers**, **Controllers**, and **Tenant Applications**. MultiSpeakers act as an interface to the network elements, shielding Controllers and Tenant Applications from the innards of the network. Controllers provide an API for tenants to submit routing requests, check validity of tenants' requests, and implement landlord's overall routing policies. Tenant applications implement tenants' routing policy logic, and issue routing requests to the Controllers based on their routing policy.

A. Design Considerations

In designing the RaaS framework, we task ourselves to come up with a framework that not only achieves the practicality, hierarchy of control, and multi-tenancy tolerance outlined in Section II, but also design the critical components to be lightweight and stateless when possible, so they can be deployed in various configurations. In the end, RaaS is designed to be a modular framework that is capable of giving multiple tenants routing customizations without burdening the existing network infrastructure.

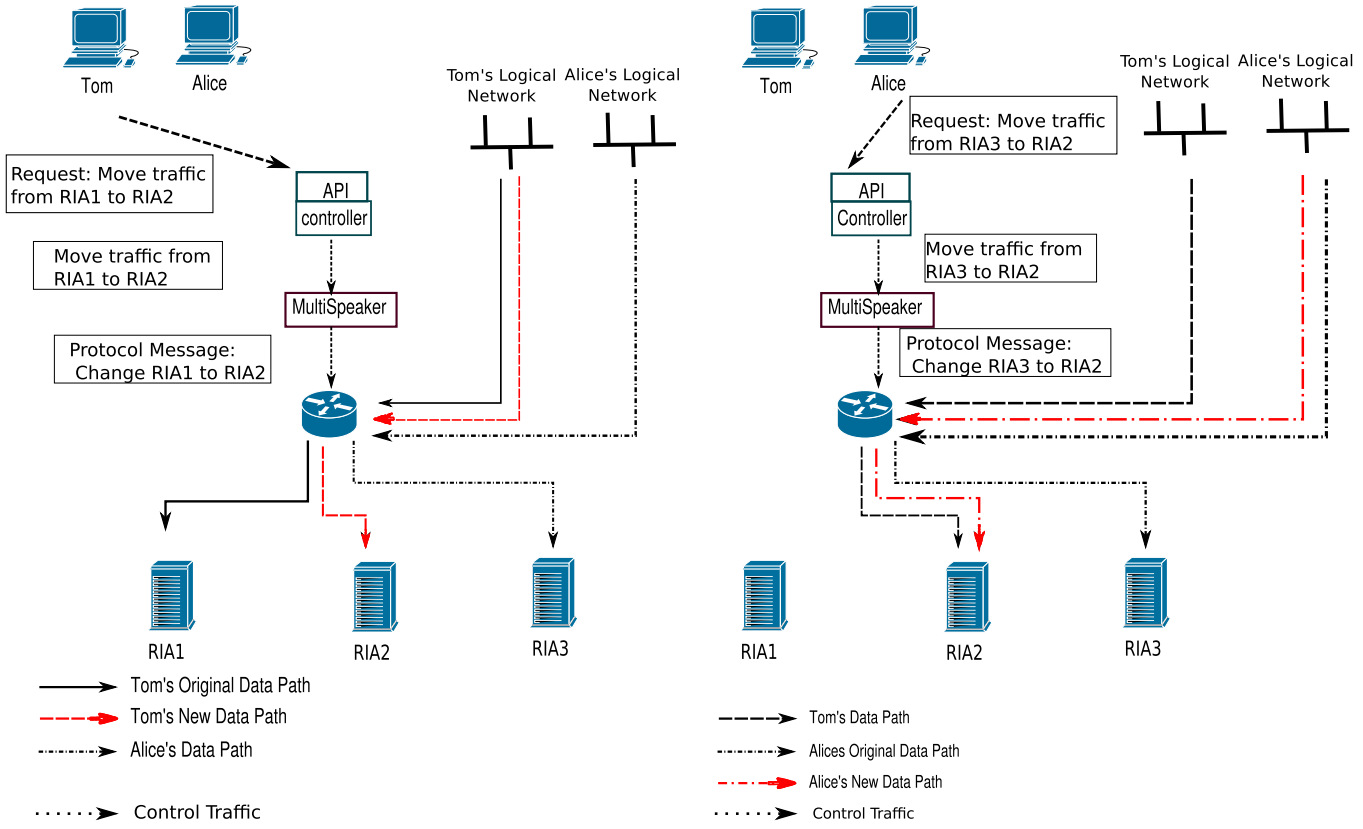
B. MultiSpeaker

MultiSpeakers actively maintain routing sessions to the router, so it could relay the requests approved by the Controllers. To ensure no fundamental changes are made to routers, MultiSpeakers communicate with routers over well-known protocols. In RaaS, MultiSpeakers use Border Gateway Protocol (BGP) [6] to install tenants' routing requests. MultiSpeakers provide API for the Controllers to relay approved tenant routing requests to the router.

Deployment of redundant MultiSpeakers is easy in RaaS, since MultiSpeakers do not cross-communicate – all the coordinations are orchestrated by the Controllers. Also, MultiSpeakers do not store states that would otherwise require a coherence protocol (e.g., BGP messages sent by the MultiSpeakers), or need to be persisted across restarts. This enables MultiSpeakers to be lightweight and stateless agents that act as relays for tenants' routing requests.

It may seem counterintuitive to use BGP, an inter-domain solution, for route control within a single administrative domain. Indeed, Interior Gateway Protocols (IGPs) such as Routing Information Protocol (RIP) [7], Open Shortest Path First (OSPF) [8], and Immediate System to Immediate System (IS-IS) [9], [10] are IGPs that are well established and may seem more suitable within a single administrative domain. However, there are several good reasons for using BGP:

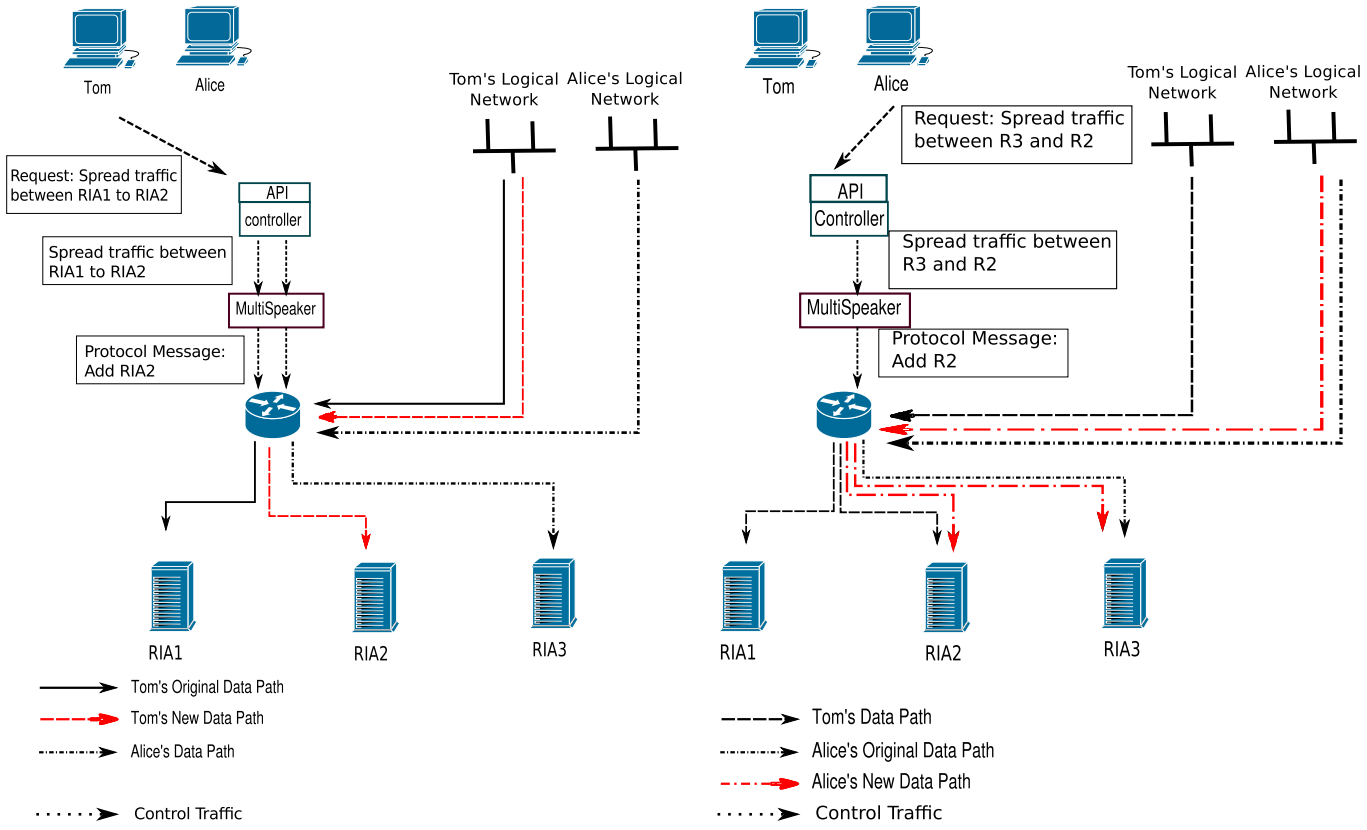
Simple State Machine: Compared to protocols such as OSPF,



(a) Tom making request to move his traffic from RIA1 to RIA2.

(b) Alice making request to move her traffic from RIA3 to RIA2.

Fig. 2: Example of traffic migration using RaaS.



(a) Tom making request to spread his traffic between RIA1 and RIA2.

(b) Alice making request to spread her traffic between RIA3 and RIA2.

Fig. 3: Example of load balancing using RaaS.

the state machine necessary to establish a functional session is simpler in BGP. A simpler state machine not only eases code verification to minimize bugs, it also makes additional augmentations easier, as explored in Section IV-D.

Flexible placement of MultiSpeakers: While a simple state machine such as RIP is desirable, flexible placement of MultiSpeakers is a desirable trait that RIP cannot satisfy. In RIP, each router exchanging RIP messages must be directly connected. This constrains the placement of MultiSpeakers to machines that are one hop away from routers, thus diminishing MultiSpeakers' flexibility and agility. BGP supports a mode ("Multihop eBGP") that enables BGP-capable peers to exchange routing messages even when they are not directly connected. Under multi-hop eBGP, it is now possible for MultiSpeakers to exchange messages with routers that are more than one hop away.

Easy Resource Management: In RaaS, resource management equates to manipulating routing to specific RIA (to be discussed in more detail in Section III-D). If the routing is manipulated by IGPs such as OSPF, it could affect the data plane and cause route instability for their external counterpart (e.g., BGP). For example, consider a RaaS alternative where OSPF is used to interact with routers that also have BGP-learned routes. If a tenant distributes the traffic over several RIAs, OSPF would need to change link metrics to ensure the path metric to all RIAs are equal. Changing the link metric, however, can affect the egress point of BGP-learned routes. On the other hand, a BGP-based RaaS implementation avoids such a ripple effect because changing routing to an RIA only affects the said RIA.

C. Controller

Before tenant requests are received by MultiSpeakers, they must first pass through the Controller. The Controller provides an API for tenants to submit routing requests per their policy. By providing an API to tenants, RaaS lessens the need for landlord to manually change tenants' routing, since such a task can now be automated via the Controller.

To prevent tenants from making erroneous routing requests, landlord and tenants need to agree on the set of resources $\Re$ (i.e., the RIAs) where tenants can host their services. Upon agreeing on the $\Re$, the landlord can implement policies that reject routing requests for resources that are not in $\Re$. The admission policy can be much more complicated, involving dynamic conditions of the network, and it will be up to the landlord to set up the admission policy. Since tenant-specific policies are now delegated to the tenants, landlord now only need to understand and implement the constraints (i.e., $\Re$s) imposed on each tenants.

In addition to providing API and policy enforcement, the Controller also coordinates MultiSpeakers. When the Controller accepts tenants' routing requests, it first records the requests and to which MultiSpeaker they are destined before forwarding them. This helps the Controller to verify if future routing requests are duplicates, a likely indication of tenant application error, and inform the tenant application of such a duplication. Storing the requests also allows MultiSpeakers to be bootstrapped upon restart; this enables the Controller to be the state memory for MultiSpeakers. To eliminate the need for Controllers to synchronize their states, each tenant can be assigned to communicate exclusively with one of the Controllers and only that Controller will hold the routing request history for that tenant.

D. Tenant Application

Tenant application is the component is implemented by tenants and executes their routing policies. Through the APIs provided by the Controller, tenants can choose how to control traffic to their services. In order for tenants to control routing to their services, RaaS requires each tenant to be assigned unique TIAs that are bound to the services and subsequently used for routing requests.

To control routing to their services, tenants issue API calls to the Controller to change the binding between TIAs and the RIAs. Instead of network administrators manually configuring routing policies, tenants can develop programs to automatically change routing to their resources (i.e., changing the TIA-to-RIA binding).

One nice property of using TIAs is that independent and safe route control is possible. For each routing request, the Controller checks the origin of the call through a security token. If the TIA is not listed under the requesting tenant's control, the request will be rejected. Since TIAs are unique to each tenant, they are mutually exclusive and tenant applications cannot modify routing to TIAs they do not own; this prevents unintentional or malicious route hijacking by other tenants. Also, tenant applications are separated, so tenants can control routing to their resources independently.

Although the TIAs are unique to each tenant, the RIAs being routed to are shared amongst tenants. For example, if $\Re_{Alice}$ = resources for Alice and $\Re_{Bob}$ = resources for Bob, $\Re_{Alice} \bigcap \Re_{Bob}$ is not necessarily an empty set. This separation of virtual resources (i.e., TIAs) and physical resources (i.e., RIAs) enables resource multiplexing amongst different tenants, while providing safe route control amongst tenants.

E. TIA-RIA Mapping and BGP

The discussion thus far presents tenant routing in the context of changing the TIA-RIA mapping, but how is the mapping installed and changed using BGP? In BGP, routing changes are announced via the **BGP Update** message type, in which an IP prefix originator (i.e., the entity who owns the IP prefix) announces or withdraws a route to the prefix. In a route announcement, the BGP Update message contains the destination IP prefix and next hop address, where the next hop address indicates the next network device that packets should traverse to reach the IP prefix. In a route withdrawal, the BGP update message simply contains the IP prefix and the routing entry corresponding to the prefix is removed from routers.

In the context of TIA-RIA mapping, the TIA address is represented by the IP prefix, and the RIA is represented by the next hop address. Thus, to install a TIA-RIA mapping, a BGP Update message to the router should be an announcement, with the TIA address being the IP prefix and the RIA being the

next hop address. To change the TIA-RIA mapping, one BGP Update message to the router should be a route withdrawal to delete the existing mapping, followed by a second BGP Update message announcing the new TIA-RIA mapping. Alternatively, sending just a BGP UPDATE message with the new next hop address will achieve the same effect, since the router will treat it as an implicit withdraw.

F. More on Using BGP

While BGP is a common protocol, it is possible that switches in some hierarchy of the data center might not have the routing stack. One such possible location is the top-of-rack (ToR). In cases where the switches at a certain hierarchy is purely switched, RaaS could not be deployed. But we note that large-scale data center network often have a L2/L3 boundary to aggregate traffic (as shown in [11]), in these cases we can RaaS can be deployed at the L3 routers and perform control routing control from there on.

One concern for using BGP is that in a general network, different factors such as diverse MRAI [12] can induce prolonged or even unstable convergence behavior. However, we note that data center environment is often managed by a single administrative entity, and this unique property can be exploited to encourage BGP convergence stability. Under a single administrative entity, parameters such as MRAI timer and BGP policies can be unified and adjusted to modify BGP's convergence behavior. While other factors such as pathological physical topology [13] may cause prolonged convergence but is difficult to re-architect, they can be altered via other means (e.g., deactivating links) to remove pathological cases. They key takeaway is that, while having visibility into and modifying policies and parameters globally is difficult in the interdomain setting, these tasks are feasible within the data center and can be exploited to ensure the network remains stable under RaaS.

IV. RaaS Implementation

This section presents the implementation of the MultiSpeaker and Controller. Tenant application will be briefly mentioned, since the actual implementation is tenant-dependent.

In addition, we present one enhancement to MultiSpeaker here. The MultiSpeaker and Controller components and their overall interactions are shown in Figure 4.

A. Tenant Application

When tenants want to customize the routing to their RIAs ($\Re$), their applications can issue calls to the Controller's API, which is shown in Table I. For portions of the policy that involve changing the TIA-to-RIA mapping, the applications can issue calls to the Controller's API. As mentioned in Section III-E, changing the TIA-to-RIA mapping equates to changing the next hop of the IP prefix. So, if a tenant Alice was given $\Re = \{server_1, server_2, server_4\}$, to initialize her service to $server_1$, she sets FirstServiceRoute = {destination: TIA_{Alice}, next_hop: IP_{server_1}}, and calls AddRoute(FirstServiceRoute, $Token_{Alice}$). To switch the service-to-resource mapping to

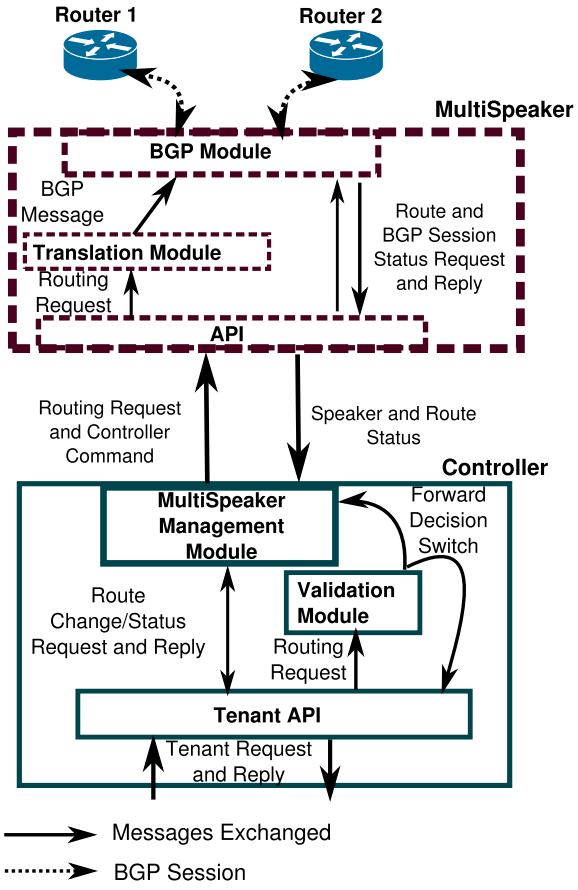


Fig. 4: MultiSpeaker and Controller components and interactions between them.

$server_4$, Alice would create a new route ReplaceServiceRoute = {destination: TIA_{Alice}, next_hop: IP_{server_4}}, and call WithdrawRoute(FirstServiceRoute, $Token_{Alice}$) followed by AddRoute(ReplaceServiceRoute, $Token_{Alice}$). Additional capabilities such as service fault recovery can also be implemented using these primitives.

B. Controller

Controller implements three modules: Tenant API, Validation Module, and MultiSpeaker Management Module.

The tenant API enables on-demand remote procedure calls and reliable messaging exchange via TCP. Setting up the API this way ensures each request can be reliably sent to the Controller without having to implement a reliable service at the application layer. The API exposed by the Controller is shown in Table I. Although the methods provided are few, they are sufficient in producing complicated resource remapping logics.

The validation module takes in tenants' routing requests as input, decides whether the routing request is valid and then outputs a binary result. The output is fed to both the MultiSpeaker management module – for the module to determine whether to forward the request onto the MultiSpeaker – and the tenant API so it can indicate to tenants the outcome of the request. Validity of the tenant request depends on whether the TIA in the request belongs to the tenant and whether the RIA is assigned to the tenant.

The MultiSpeaker management module manages the communication between the Controller and the MultiSpeaker. In addition to passing routing requests and route inquiries, it also ensures MultiSpeaker states reflect the state memory stored at the Controller. To achieve this, both the MultiSpeaker and Controller maintain an acknowledgement table. Each table entry is a (TIA, destination IP, action type) tuple that denotes a request that Controller has forwarded to the MultiSpeaker but does not know whether it has been submitted to routers. The MultiSpeaker management module also detects MultiSpeaker restart so the Controller can bootstrap MultiSpeakers when they restart; MultiSpeaker management module can detect MultiSpeaker restart by periodically polling the MultiSpeaker.

C. MultiSpeaker

MultiSpeaker consists of three components: protected API, BGP module, and translation module.

The protected API specifies methods for MultiSpeaker to exchange messages with Controller's MultiSpeaker management module. The methods are similar to those exposed by the Controller in Table I, so we omit it here.

The translation module takes tenant requests as input, and outputs well-formed BGP UPDATE messages. By translating the messages here, tenants and Controllers are shielded from having to know the innards of the network.

For WithdrawRoute() calls, the translation module generates a BGP UPDATE messages with the WITHDRAWN ROUTES fields filled. For AddRoute() call, the module generates a BGP UPDATE message that includes the NEXT_HOP and the IP Prefix (NLRI) fields. In addition to the destination IP prefix and next hop IP address, Update messages for AddRoute() calls also include the AS paths. AS path is a mandatory attribute that encodes the autonomous system (AS) numbers by which the BGP UPDATE message has traversed since the prefix origin. Even though tenants are the origins in supplying the destination IP prefix, having tenants supply the AS number would imply tenants having knowledge of the innards of the network. To avoid such a burden on tenants, MultiSpeakers act as the origin of tenants' prefixes. Thus, the translation module uses the AS number – possibly a private AS number that are removed at the data center boundary – of the MultiSpeaker as the first AS in the AS path.

The features and attributes implemented by the BGP module is minimized to the set of features necessary to establish BGP sessions, add/withdraw routes, and react to BGP notifications in order to reduce MultiSpeaker complexity.

Using a BGP module, MultiSpeaker provides information isolation between the tenants and routers, much like BGP_MUX [14]. For tenants, they are isolated from the interactions between MultiSpeakers and routers, but are still able to perform route control. On the other hand, routers are not exposed to the RaaS internals and interact with MultiSpeaker as if it is another BGP-capable speaker. This separation enables RaaS' implementation to vary with minimal impact to routers and the tenants.

Method Name	Purpose
bool AddRoute(Route r, Token t)	Adds specified route to router
bool RemoveRoute(Route r, Token t)	Removes specified route to router
Status GetRouteStatus(Route r, Token t)	Check status of route

TABLE I: Controller interface to tenants. Route = resource routing info, token = tenant identity.

D. Equal-Cost Multi-Path Enhancement (ECMP)

Discussions on the BGP module thus far assumes each BGP module can only establish one BGP session with each router (as depicted in Figure 4). Such a configuration would be fine if tenants only announce a single TIA-RIA mapping at a time. However, in cases where tenants announce one-to-many TIA-RIA mappings (e.g., for load balancing), multiple MultiSpeakers would be required. This method would require the number of MultiSpeakers, N, to be $k \times \max_{\forall t \in tenants} mappingSize_t$, where k is the number of routers a MultiSpeaker connects to, and mappingSize is the cardinality of one-to-many TIA-RIA mapping. Intuitively, the equation above says the number of MultiSpeakers needed is the number of routers establishing a BGP session to a MultiSpeaker, multiplied by the maximum count of one-to-many TIA-RIA mapping needed by any tenant. If redundancy is required, an unmanageable number of MultiSpeakers would need to be deployed. A simple extension to the BGP module could be implemented to avoid such an explosion, in which **each BGP module instantiates multiple BGP sessions** (hence the name **Multi**Speaker), with each session capable of announcing one TIA-RIA mapping per tenant. Implementing this extension simply requires the BGP module to keep separate state machines and data structures for each session. Since there is no need for the instantiated sessions to share state, MultiSpeaker complexity does not change. We note that implicit withdraw (Section III-E) will not work here if the message is sent over a different peering session, as router will treat it as another equal-cost multi-path (ECMP) route.

V. Component Evaluation

In this section we evaluate the performance of our RaaS-based implementation via several micro-benchmarks. We present the methodology in Section V-A and the evaluation results in Section V-B.

A. Methodology

The main metrics of interest are i) the time for the Controller and MultiSpeaker to process each request, ii) memory consumptions of various data structures, iii) network overhead incurred by the requests, and iv) availability of the Controller to serve tenant requests. To demonstrate the utility of RaaS, we developed a prototype based on RaaS using C# and Windows Communication Foundation (WCF) [15] for the remote procedure calls. Our choice of programming language was based on the ease of development and the use of WCF was its seamless integration with C#. The experiments were carried out on COTS hardware that include a dual-core 2.80GHz machine with 4GB of RAM and two single-core 1.7GHz machine with less than 1GB of RAM. The timing experiments were carried

out on the dual-core machine, and the network overhead experiments were carried out across the three machines.

To collect detailed memory usage of the various data structures, a custom program loads each data structure, one at a time, and drives realistic loads on the data structures. For example, to collect the memory usage of the acknowledgement table, the program loads an acknowledgement table and inserts various amount of entries to it. The processing time is collected by implementing a tenant application that sends routing requests and collect the end-to-end response time back to the tenant. Both the memory usage and processing time experiments described above were carried out on a single machine, since they are not affected by the network. A second set of experiments was carried out between two machines to measure the network overhead.

Since MultiSpeaker's configuration affects the processing time and the memory consumption of both the Controller (e.g., time: route assignment, memory: MultiSpeaker state table) and MultiSpeaker (e.g., memory: BGP sessions, time is heavily impacted only if all request-serving threads are being contended), we vary MultiSpeaker's configuration parameters and collect the time and memory metrics. Specifically, for each experiment, we vary the number of routers (denoted as RS) and ECMP sessions to each router (denoted as E). In addition, for the Controller we also vary the number of MultiSpeakers (denoted as S) being managed by the Controller. Because we do not have many routers for the MultiSpeaker to establish BGP sessions, we implemented a simple router emulator that simply waits for and maintains BGP sessions once they are established.

We also demonstrate RaaS' feasibility by showing that with a small number of redundant components and pessimistic settings for equipment uptime and replacement time, tenants has a good chance of successfully submit routing request on the first try. To do so, we use alternating renewal process (ARP) [16] to formulate a theoretical model for the availability of all equipments on the path from tenant to the Controller, and evaluate the success rate for a given tenant request to reach the Controller. We only model the success rate from the tenant to the Controller because tenants only interact with the Controller. Additional details can be found in the appendix.

B. Evaluation

1) Speaker-only evaluation: Table II shows MultiSpeaker's processing time for route announcement and withdrawal operations. The processing time measures, per BGP session, the time between receiving the route operation request from the Controller and sending the well-formed BGP request out. Since the time taken to send BGP messages to routers is partially influenced by network delay, which we cannot control, we eliminate the network delay by co-locating the router emulator and MultiSpeaker. The result shows that MultiSpeaker can handle Controller's request quickly, often under 1 ms. Factoring in the network delay, the true response time might be over 1 ms, as the MultiSpeaker can establish BGP session with routers via Multi-hop BGP for better MultiSpeaker placement flexibility. Barring network anomaly, given the current network bandwidth in data centers and the small size of BGP messages, the network delay should be small.

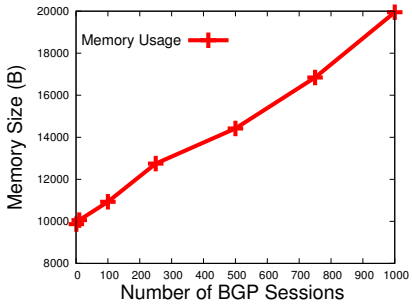
Figure 5a shows MultiSpeaker's memory usage with respect to number of BGP sessions established. Here we do not distinguish whether the session is connected to the same or different routers, because the amount of states being kept for each BGP session is the same regardless. This figure shows that the MultiSpeaker can maintain 1,000 sessions with moderate amount of memory, making a single MultiSpeaker process scalable up to thousands of sessions.

Figure 5b shows MultiSpeaker's memory consumption when storing outstanding entries with various configurations of RS and E. In this experiment, we assume that the number of outstanding entries per sessions are the same across all BGP sessions. Memory usages are similar for configurations where RS=1 or RS=10, so we only plot one configuration here (RS=10, E=16). We observe significant differences when RS=1000; RS=1000 and E=16 configuration has only two data points as memory usage exceed 2GB when outstanding entry is 1000. The reason that memory usage increases rapidly when RS=1000 is due to the number of total entries added, as each additional outstanding entry per session results in 1,000$\times$E total outstanding entries. For example, in the configuration when RS is 1,000 and E is 4, having 10 outstanding entries per session results in 40,000 total outstanding entries and having 1,000 outstanding entries per session results in 4,000,000 total outstanding entries. In reality we do not expect the amount of outstanding entries to be as high as 4,000,000, unless they are not periodically cleared by Controller.

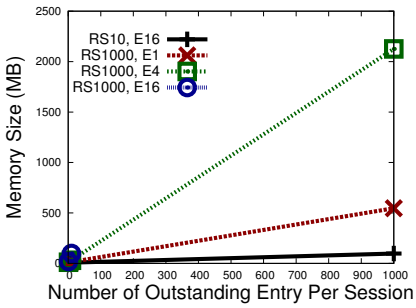
2) Controller-side evaluation: The Controller processing times for route operations are shown in Table II. We show the result for a MultiSPeaker configuration that amounts to little over 1,000 total BGP sessions, corresponding to the maximum memory usage shown in Figure 5a. Assuming the maximum ECMP possible (i.e 16), the MultiSpeaker is connected to 63 routers.

Table II shows the average and standard deviation of Controller's processing time for the AddRoute and RemoveRoute operations. It shows that both operations can respond to the tenant request within milliseconds of receiving the request, and thus can handle close to 1,000 requests per second on average. This processing speed is fast considering that for each tenant request, the Controller has to inspect up to 1,000 sessions to find route assignments for all the routers. Route addition is slightly slower than route removal because it performs one additional check for the case when the route was withdrawn over a session but is still outstanding (i.e., the route removal has not been sent to the router). In this case the AddRoute operation use the same session in order to avoid a temporary and unintended ECMP.

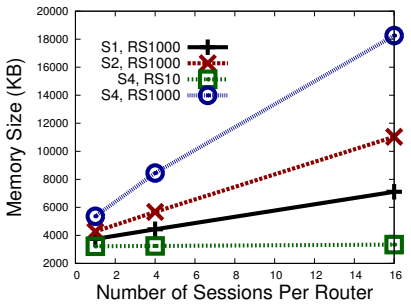
Figure 5c shows the memory usage to store the MultiSpeaker state. We vary the number of MultiSpeakers managed by the Controller (S = 1, 2, 4). And for each MultiSpeaker we vary the number of routers it connects to (RS = 1, 10, 1000), and the number of ECMP sessions per router (E= 1, 4, 16). Memory utilizations are similar for all configurations where RS$\neq$1000, so we only plot one configuration here (S=4,RS=10). The plot shows that the memory consumption



(a) MultiSpeaker memory usage with various number of BGP sessions.



(b) MultiSpeaker outstanding ACK table memory usage. RS = # of routers per MultiSpeaker, E = # of ECMP sessions.



(c) Controller memory needed to keep MultiSpeaker state. S = # of MultiSpeaker per Controller, RS = # of routers per MultiSpeaker.

Fig. 5: Memory Usage Results.

	Mean	Std dev
Controller Processing Times (ms)		
Adding Route	1.24	3.38
Removing Route	1.14	2.93
MultiSpeaker Processing Times (ms)		
Announcing Route	0.091	0.67
Withdrawing Route	0.082	0.38

TABLE II: Route Operation Processing Time.

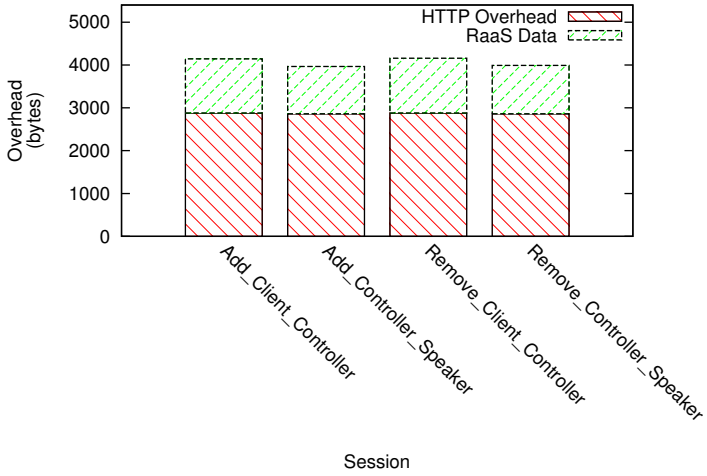


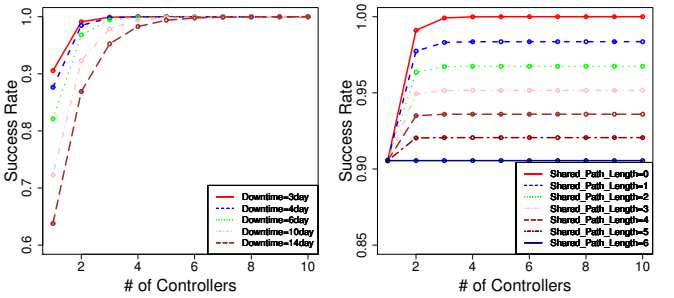
Fig. 6: Network overhead over various operations. x_y_z refers to operation x, with caller x and callee z.

increases noticeably only when the number of routers per MultiSpeaker (i.e., RS) is 1,000. This is intuitive because when the number of routers is 1,000, each additional ECMP session per route adds 1,000 more total entries to the MultiSpeaker state table. We note that in the worst case (4 MutiSpeakers, 1000 routers per MultiSpeaker, 16 ECMP sessions per router, 64,000 total sessions), the per-session state consumes about 300 bytes of memory.

3) Network Overhead: In this experiment we are interested in observing the network overhead for the communication between tenant/Controller and Controller/MultiSpeaker. We capture the network traffic at the Controller to record traffic between the tenant/Controller and Controller/MultiSpeaker, and later filter the traces to separate the two types of traffic. While running the experiment using the ECMP configuration, we realized it was difficult to separate traffic from different ECMP sessions. Therefore, we enable only one session and sent a single request to capture serialized conversation between the tenant/Controller and Controller/MultiSpeaker. The result can be easily scaled to multiple ECMP sessions, as each distinct session will have roughly the same amount of network overhead.

Result from Figure 6 implies that, given a typical 1Gbps edge bandwidth, our prototype will saturate the link at around 12,500 requests/second (Assuming around 4KB per request. 4KB incoming request and 4KB outgoing reply). Since our prototype serves around 1,000 requests/second, we will only be using up to 10% of the link capacity. We also see room for improvement, as a majority of the overhead comes from HTTP, which includes exchanges needed by the WCF framework. Additional bandwidth can also be conserved by avoiding the use of the serialization engines in WCF [17], which converts data into XML format.

4) Service Availability: Based on the derivation made in the appendix, we use R [18] to evaluate the amount of redundant Controllers needed to be deployed. We perform the evaluation by setting various values for the number of Controllers deployed and their expected downtime, and record the success rate. We use the Weibull distribution to model the uptime distribution, due to its ability to model different hazard rate characteristics with age. Weibull distribution has the shape (k) and scale (λ) parameter, with the former affecting the hazard rate over time and the latter the expected uptime. To understand the effect of the hazard rate parameter, we plot the success rate against varying k, setting path length = 6, expected uptime = 6 months, expected downtime = 3 days, required equipment uptime during request submission (i.e., ΔT) = 4 minutes. We choose these parameters based on the reference data center topology shown in the Cisco reference [11], a pessimistic estimation of a typical equipment's uptime and time required to replace it, and maximum possible TCP retransmission timeout (RTO) as defined by RFC 1122 [19]. We do so because that is the maximum time the tenant will wait before considering the Controller dead. We found that the request success rate is insensitive to k, with the difference between the maximum and the minimum success rate less than 0.1% across all k. This is due to the fact that the stable-state success rate is dominated by the ratio of the expected up/downtime, and the temporal variation of



(a) No shared path.

(b) Shared path with expected downtime = 3 days.

Fig. 7: Availability experiment results.

the uptime distribution becomes insignificant. Following this observation, we set k=1 for subsequent evaluations. Setting k=1 results in exponential distribution, a common distribution used in reliability engineering. Next, we evaluate the effect of having several Controllers for the request to be served. We assume that each Controller is deployed under different routers (so malfunction of one router does not remove all of the Controllers) and has a completely disjoint path from other Controllers. Then, given the number of disjoint paths, we calculate the success rate using the same parameters as the previous experiment. The result is shown in Figure 7a. It shows that, given the same pessimistic setting, the availability of the overall service is above 90% even when the Controller is hosted in only one location, and the overall service availability quickly converges after having more than 2 distinct paths. To gain more insight about the availability, we increase the expected downtime and obtain the new success rates. We find similar conclusion holds: the success rate converges to a stable value quickly and overall service remains highly available. We also see that deploying multiple Controllers can be high beneficial when services are expected to be down for a prolonged period of time. In the case of expected downtime equal to 2 weeks, adding another Controller increases the overall service availability by 30%.

The experiment above makes an unrealistic assumption that paths to additional Controllers are disjoint, but in reality many components are shared amongst different paths. We modified our formulation to take into account the shared path length and performed the calculations again. Figure 7b shows the result for the case when expected downtime = 3 days. We find that Controller availability relative to the ideal case can deteriorate as much as 10% when shared path is taken into account, and a maximum availability is visibly less when paths are shared. The upside is that the overall service availability is above 90% in all cases. This is an indication that network administrators should be careful in deploying the Controllers, and should strive to have as much path diversity as possible.

In summary, the experiments show that RaaS obviates the need for landlord to deal with individual requests, resulting in less personnels needed to process tenants' requests. In addition, RaaS components can be implemented on COTS hardware, making it easily deployable into data centers; and the deployment will not cause overwhelming burden to the network due to the small number of redundancy required. This makes RaaS a flexible framework that can be used to reduce personnel cost and expose network programmability to multiple tenants.

VI. Usage Scenarios

In this section we investigate real-world applications of the RaaS framework. We experiment using the RaaS prototype, and present the results for two likely scenarios: *load balancing* and *workload migration*. We detail the experimental set-ups and evaluation results in the respective subsections below.

A. Load Balancing

Load balancing refers to a technique where the traffic is distributed evenly across multiple outgoing links. With load balancing enabled, the network increases its resilience against random link failures. Resilience against random link failures is an important feature, because as failures occur closer to the core the effect might become more severe. We demonstrate that RaaS can empower tenants to programmatically perform load balancing on their traffic by requesting additional links to their VIP; the set-up is shown in Figure 8a.

Two servers, 10.0.0.2 and 10.0.0.3, are used to receive the traffic destined for an application with its service IP set to 10.0.10.1. The two servers host the service IP on its loopback interface, so no IP address conflict occurs during the address resolution process. A third server hosted on a different subnet acts as traffic generator, sending traffic to the service IP during the run of the experiment. The traffic generator generates two simultaneous 10Mbps UDP traffic streams, with the source IP addresses chosen so that the two streams will be routed to different servers if multiple paths were available. To ensure no devices in the network can reach each other initially, all routing information are removed with the exception of the router as the default gateway. At the router, equal-cost multipath for BGP is enabled. Initially, all the traffic destined for the service IP are directed to 10.0.0.2, and after some time tenant Tom's application issues a load balancing request to add 10.0.0.3 to serve the load.

Figure 9a shows the result. Initially, all of the traffic are sent to 10.0.0.2 while none are going to 10.0.0.3, as demonstrated by the 20Mbps traffic observed at 10.0.0.2. At time 30, a load balancing request is issued, and after several seconds of traffic disruption the traffic is redistributed as intended. We note that traffic never drop to 0 Mbps, and load balancing is completed in two seconds.

We note that although in this experiment we pre-select the IP addresses to allow the two traffic volume to be directed to different servers, in general this should hold true as well. When ECMP is enabled on a router, the router uses a hashing scheme to choose the traffic destination, with the goal of distributing the incoming flow equally amongst the available next-hops. While some imbalance might occur due to bad luck (i.e., most of the incoming traffic are hashed to the same destination), this chance occurrence should happen less as the number of flow in the network increases. Since in this experiment we want

to demonstrate that load balancing is possible, we exclude the possibility of such imbalance by choosing specific IP addresses so the hashing scheme chooses different destination.

B. Workload Migration

Workload migration is the idea of moving traffic from one place to another. We show that such a simple idea, when coupled with the RaaS framework, can be used as a foundation upon which custom policies can be built. The policies possible for these expressive actions is only constrained by the information available to tenants, which is enormous due to the amount of information modern operating systems expose. For instance, the Windows event tracing framework [20] exposes some 600+ data sources upon which users can query. In this experiment, we developed a simple application to demonstrate how a tenant could implement a custom policy using the RaaS framework.

The policy tenant Tom wishes to achieve here is service availability, with the constraint that at least one server must be used to serve incoming traffic. Furthermore, there is a set of primary and secondary servers which imposes the order that Tom would like to use the servers. The application then should direct traffic to the primary server whenever possible, and only use the secondary servers when the former is unavailable. The set-up of the experiment is similar to that of the load balancing experiment, with the change that 10.0.0.2 is designated as the primary server and 10.0.0.3 the secondary server. The traffic generator in this experiment is emitting only one 10Mbps stream to the service IP (10.0.10.1).

The result is shown in Figure 9b. As the policy indicated, upon start-up the traffic is directed towards 10.0.0.2. Some time later (time 30) 10.0.0.2 is rendered unavailable by disconnecting its Ethernet cable. After a similar disruption phase observed in the load balancing experiment, the traffic is restored onto 10.0.0.3. The result shows that during both transitions there were some packet loss. This is expected as the connection terminated abruptly and it takes time to detect the connection termination. It is worth noting that traffic migration technique is still possible for session-oriented traffic. The concern here is that once the session-oriented traffic has been migrated, the new server would not be able to serve these requests, as it does not have the necessary network stack states established. This problem can be mitigated by process and virtual machine migration [21], in which the system a priori migrate necessary states over to the new systems. As many multi-tenant platform (e.g., cloud providers) already employ virtual machines, migrating system states is simplified. As migration is outside the scope of this paper, we refer the reader to [21] for more details on virtual machine migration.

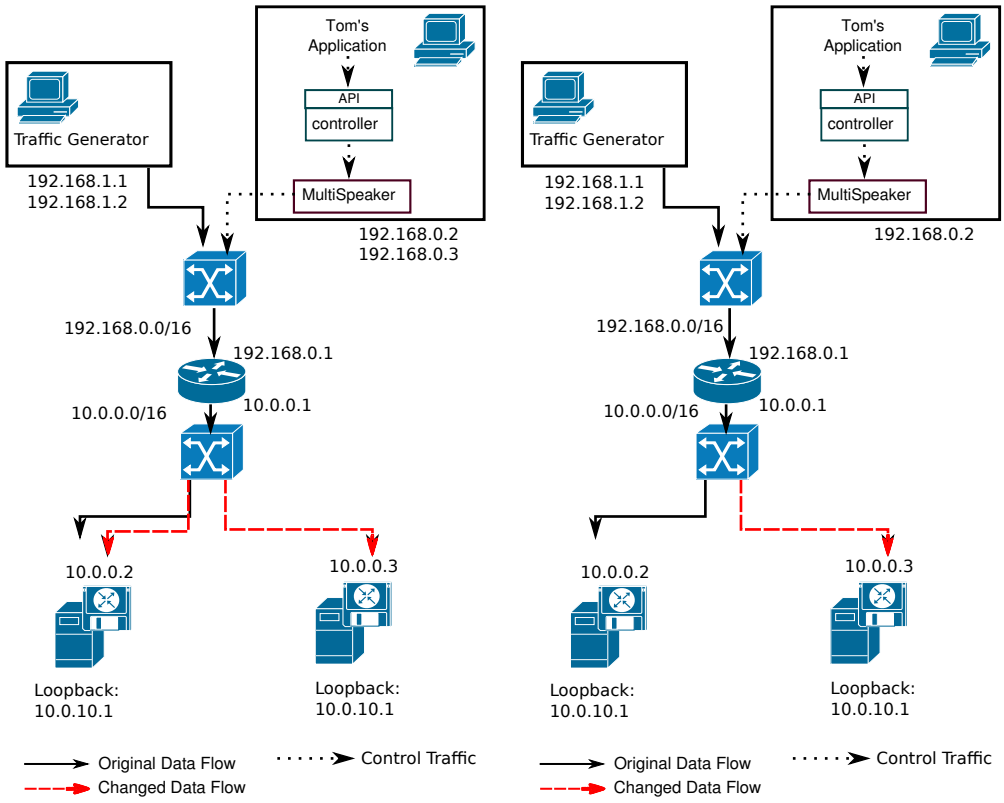
One can imagine that more complicated policies than the one shown here can be defined. For example, one could add a restore policy to describe the condition under which traffic will be restored to the primary server. Or, the policy can be updated to include additional considerations such as bandwidth and processing latency. With the RaaS framework, tenants are no longer constrained to receiving alerts and contacting landlords for routing modifications, they can now control how their services react to adverse conditions.

In summary, we have shown that the RaaS framework can be used in realistic scenarios such as load balancing and traffic migration. Even though the use of custom policy was only demonstrated for traffic migration, it is also possible to create custom policies for load balancing (i.e., defining conditions upon which to expand or shrink redundant paths).

VII. Related Work

Dynamic and programmable routing platforms are not unique to RaaS, as there are prior proposed works in both academia and industry. Here we discuss the relevant works and the difference between RaaS and them.

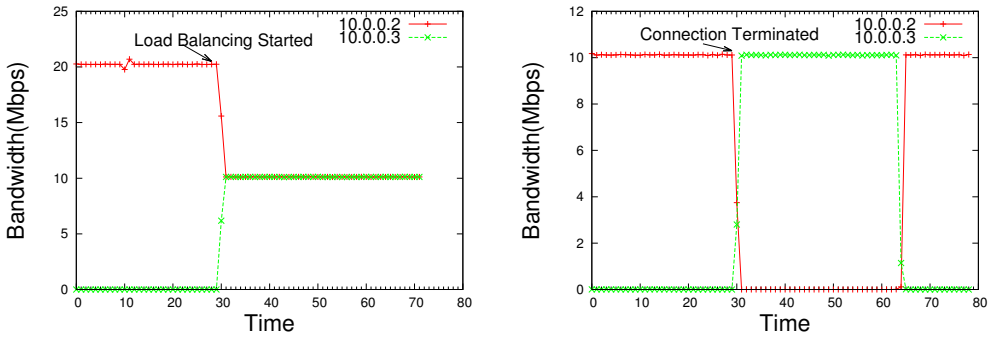
Previous academic works such as OpenFlow [22], NIRA [23], Tesseract [24], RAS [25], PaaS [26], Morpheus [3], Transit Portal [5], and RCP [4] proposed customizable routing. These works had a similar goal in providing end-users or tenants with the ability to choose how their packets would be routed. However, some of these works ([23], [24]) require technologies that do not yet exist in the transport hardware or are in nascent stage, whereas RaaS leverage well-known and mature technologies. This allows RaaS to be implemented without infrastructural overhaul. OpenFlow [22] is a proposed interface specification that can also implement RaaS-like framework, and project such as FlowVisor [27] leverages OpenFlow-capable switches to provide traffic slicing and custom actions in a traffic slice. While OpenFlow allows for FIB programming to enable functionality similar to what RaaS provides, RaaS builds upon a more mature and widely available technology, making deployment to legacy data centers possible. This makes RaaS complementary to OpenFlow by giving administrator an evolution path to immediately enable route programming in legacy data centers, and over time introduce OpenFlow-capable switches into the network to enable additional programming. Another approach to enable network programmability is to implement OpenFlow capabilities at network edge (e.g., Open vSwitch [28]), and use existing routing protocol in the network core. While this is a valid alternative, the maintenance cost scales with the number of participating servers while RaaS scales with the number of switches, which is generally smaller than the number of servers. Other works leverage existing routing technologies, such as BGP, to control routing either within a single AS [3] [4] or to various upstream ISPs [5]. RaaS also leverages the same set of technologies to make route-control possible, but it also provides programmatic interface to tenants directly, while providing performance isolation and independent route control. These were not discussed at great length or at all in previous works. There are also proposals that attempt to extract routing purely as a service [25], which is similar to what RaaS is achieving. However, RaaS provides this control directly to tenants, instead of going through a third party, providing routing as a first-class service. PaaS [26] provides a similar abstraction to tenants, however, it is unclear the technologies required and what fundamental changes are required. In RaaS we provide a concrete framework and working prototype to demonstrate the utility of a tenant-directed route control framework.



(a) Load balancing scenario setup. The entire RaaS framework is hosted inside a single machine. Unless indicated, addresses are installed on physical interfaces.

(b) Workload migration scenario setup. The entire RaaS framework is hosted inside a single machine. Unless indicated, addresses are installed on physical interfaces.

Fig. 8: Evaluation set-ups.



(a) Load balancing result.

(b) Workload migration result.

Fig. 9: Experimental results.

On the industry side, services such as Amazon's EC2 [2] and Route 53 [29], Internap's Performance IP [30], RouteScience's(RouteScience has been acquired by Avaya) [31] PathControl offer route control services to end-users. EC2 is an infrastructure-as-a-service (IaaS) system that gives their tenants control over virtual machine (VM) placement, load balancing (in a service called elastic load balancing, also known as ELB), and IP-to-VM mapping. RaaS differs from EC2 in that RaaS offers the underlying routing plane as a service instead of individual capabilities. Rather than providing IP-to-VM mapping, for example, RaaS can support mapping of IP to any entities in the network that is IP-addressable. Internap's Performance IP service offers automatic route control based on network conditions, and would automatically change routing so customers' packets traverse through the optimal ISP links. RouteScience's PathControl solution is similar to Internap's Performance IP, and it is sold as a hardware solution [32]. However, in both solutions there is no programmable API for tenant to implement their own route-control logic. While Route 53 does not offer routing ability on the IP address level, it is able to direct user traffic by changing the domain names to IP address mapping. RaaS could complement services such as Route 53 by providing another level of control after the domain names have been translated to IP address.

VIII. Conclusion

The traditional paradigm for routing customization involves a laborious and lengthy process, in which landlord and tenants are tightly coupled. In this paper we introduced the Routing-as-a-Service (RaaS) framework, where the coupling between landlord and tenants are lessened. In the RaaS framework, the landlord only needs to understand the resource set $\Re$ of the tenants, and tenants can perform route customization independently of other tenants. This results in less dedicated personnel to process tenants' requests and more independent route control for the tenants. We showed that our prototype based on the RaaS framework can process requests quickly, often less than a second after receiving the request. In addition, we also showed that it is possible to offer more aspects of the data center as a service without major infrastructural overhaul. With data centers becoming more popular and widespread, we believe RaaS is an important addition to the set of services that can be offered to tenants.

Appendix A
Service Availability

A. Basics

We model the data center network as a fat-tree, with non-leaf nodes as routers/switches and leaf nodes as servers, with the service reside on the servers. The availability of the RaaS depends on the availability of the path from one leaf of the tree to another leaf; we assume the worst case scenario, where all requests to Controller travel the longest path. For simplicity, we also assume that no redundancies are in place. While this assumption does not hold in practice, we assume this as a worst-case scenario and derive our result for such a case. For a path p, each equipment's availability is modeled by an alternating renewal process (ARP) [33]. Mathematically, let $A_a(t), A_b(t), ..., A_n(t)$ be ARPs for components $a, b, .., n$, along a path where

$$A_k(t) = \begin{cases} 1, & \text{if component k is in the up state at time t} \\ 0, & \text{if component k is in the down state at time t} \end{cases} \tag{1}$$

$A_k(t)$ is described by bivariate independent and identically distributed (iid) random variables $\{(U_n^k, D_n^k), n \geq 1\}$, where (U_n^k, D_n^k) are random variables describing the n^{th} up-time and down-time intervals for component k, respectively.

S be a random variable where,

$$S(t, \Delta T) = \begin{cases} 1, & A_a(t_a) = 1 \cap ... \cap A_n(t_n) = 1, \\ & t \leq t_a \leq t + \Delta T, .., t \leq t_n \leq t + \Delta T \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Intuitively, the success of the request is dependent on the components along the path to be in the up state, and the minimum up-time across all components be at least the time needed to service the request (ΔT). We are ultimately interested in the stable-state probability that a request is served:

$$\lim_{t \to \infty} Pr(S(t, \Delta T) = 1) \tag{3}$$

B. Result

Theorem 1. *Given a path p to the services, where p is the set of nodes $\{N_1, N_2, ..., N_n\}$. If the state of each node and S(t) are defined as in Section A, then:*

$$\lim_{t \to \infty} (Pr(S(t, \Delta T) = 1) = \left(\frac{E[U] - \int_0^{\Delta T} (1 - F_U(t)) dt}{E[U] + E[D]} \right)^n \tag{4}$$

Proof: The proof consists two parts: Finding $Pr(S(t, \Delta T) = 1)$ and finding the limiting probability of $Pr(S(t, \Delta T) = 1)$ as $t \to \infty$. To find $Pr(S(t, \Delta T) = 1)$, note that

$$\begin{aligned} Pr(S(t, \Delta T) = 1) &= Pr(A_a(t_a) \\ &= 1 \cap A_b(t_b) = 1 \cap ... \cap A_n(t_n) = 1, \\ &\quad t \leq t_a \leq t + \Delta T, .., t \leq t_n \leq t + \Delta T) \end{aligned} \tag{5}$$

based on the definition of $S(t, \Delta T)$. Using the assumption that equipments act independent of each other, we have

$$Pr(S(t, \Delta T) = 1) = \prod_{i=1}^{n} Pr(A_a(t_a) = 1, t \leq t_a \leq t + \Delta T) \tag{6}$$

Taking the limit as $t \to \infty$, we have

$$\begin{aligned} &\lim_{t \to \infty} Pr(S(t, \Delta T) = 1) \\ &\quad = \lim_{t \to \infty} \prod_{i=1}^{n} Pr(A_a(t_a) = 1, t \leq t_a \leq t + \Delta T) \\ &\quad = \prod_{i=1}^{n} \lim_{t \to \infty} Pr(A_a(t_a) = 1, t \leq t_a \leq t + \Delta T) \end{aligned} \tag{7}$$

$\lim_{t \to \infty} Pr(A_a(t_a) = 1, t \leq t_a \leq t + \Delta T)$ is known as the limiting interval reliability and the derivation is given in [33]. Starting with the equivalent result stated in [34] and re-arranging the derivation gives the desired result. ■

References

[1] C.-C. Chen, L. Yuan, A. Greenberg, C.-N. Chuah, and P. Mohapatra, "Routing-as-a-service (raas): A framework for tenant-directed route control in data center," in *INFOCOM, 2011 Proceedings IEEE*, april 2011, pp. 1386 –1394.

[2] Amazon, "Elastic Compute Cloud." [Online]. Available: http://aws.amazon.com/ec2/

[3] Y. Wang, I. Avramopoulos, and J. Rexford, "Morpheus: making routing programmable," in *INM '07: Proceedings of the 2007 SIGCOMM workshop on Internet network management*. New York, NY, USA: ACM, 2007, pp. 285–286.

[4] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and Implementation of a Routing Control Platform," in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[5] V. Valancius, N. Feamster, J. Rexford, and A. Nakao, "Wide-Area Route Control for Distributed Services," in *USENIX Annual Technical Conference*, 2010.

[6] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4) – RFC 4271," 2006.

[7] G. Malkin, "RIP Version 2 – RFC 2453," 1998.

[8] J. Moy, "OSPF Version 2 – RFC 2328," 1998.

[9] R. Callon, "Use of OSI IS-IS for Routing in TCP/IP and Dual Environments – RFC 1195," 1990.

[10] D. Oran, "OSI IS-IS Intra-domain Routing Protocol – RFC 1142," 1990.
[11] Cisco, "Data Center Design IP Network Infrastructure," 2009.
[12] A. Fabrikant, U. Syed, and J. Rexford, "There's something about mrai: Timing diversity can exponentially worsen bgp convergence," in *INFOCOM, 2011 Proceedings IEEE*, april 2011, pp. 2975 –2983.
[13] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, "Delayed internet routing convergence," in *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '00. New York, NY, USA: ACM, 2000, pp. 175–187. [Online]. Available: http://doi.acm.org/10.1145/347059.347428
[14] V. Valancius and N. Feamster, "Multiplexing bgp sessions with bgp-mux," in *CoNEXT '07: Proceedings of the 2007 ACM CoNEXT conference*. New York, NY, USA: ACM, 2007, pp. 1–2.
[15] Microsoft, "Windows Communication Foundation," 2006.
[16] V. G. Kulkarni, *Modeling and Analysis of Stochastic Systems (Chapman & HallCRC Texts in Statistical Science)*, 1996.
[17] Microsoft, "Windows Communication Foundation, Data Contracts," 2006.
[18] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2010, ISBN 3-900051-07-0.
[19] R. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122 (Standard), Internet Engineering Task Force, Oct. 1989, updated by RFCs 1349, 4379, 5884.
[20] I. Park and R. Buch, "Improve Debugging And Performance Tuning With ETW." [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc163437.aspx
[21] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.
[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.
[23] X. Yang, "NIRA: a new internet routing architecture," in *FDNA '03: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*. ACM, 2003.
[24] H. Yan, D. Maltz, T. Ng, H. Gogineni, H. Zhang, and Z. Cai, "Tesseract: A 4D Network Control Plane," *NSDI '07: 4th USENIX Symposium on Networked Systems Design & Implementation*, 2007.
[25] K. K. Lakshminarayanan, I. Stoica, S. Shenker, and J. Rexford, "Routing as a service," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-19, Feb 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-19.html
[26] E. Keller and J. Rexford, "The "platform as a service" model for networking," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, ser. INM/WREN'10. Berkeley, CA, USA: USENIX Association, 2010.
[27] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," 2009.
[28] O. vSwitch. [Online]. Available: http://www.openvswitch.org
[29] Amazon, "Amazon Route 53." [Online]. Available: http://aws.amazon.com/route53/
[30] Internap, "Network-Based Service: Performance IP."
[31] Avaya, "http://www.avaya.com."
[32] G. Goddard and R. Vaughn, "Review: RouteScience's PathControl," 2002.
[33] R. E. Barlow and F. Proschan, *Mathematical Theory of Reliability*. Society for Industrial Mathematics, 1987.
[34] L. A. Baxter, "Availability measures for a two-state system," *Journal of Applied Probability*, vol. 18, no. 1, pp. 227–235, 1981.

Chao-Chih Chen is currently a fifth-year Ph.D student in the department of Computer Science at the University of California, Davis. Chen's research interests include distributed systems, data center network, network management, and distributed algorithms/protocols.

Lihua Yuan received his PhD in Electrical and Computer Engineering from the University of California, Davis in 2008. His research interests are in the area of computer network and distributed systems, with a focus on network management, measurement, and security. Currently, Dr Yuan is with Microsoft Corp where he is working on Next-Generation Data Center network.

Albert Greenberg is at Microsoft, where he works on data center networks, cloud service infrastructure, enterprise network management, and monitoring. He joined Microsoft in 2007, after many years at Bell Labs and AT&T Labs Research, where he was an Executive Director and AT&T Fellow, and where he helped build the systems and tools for engineering and managing AT&Ts networks. Albert is an ACM Fellow.

Chen-Nee Chuah is a Professor in Electrical and Computer Engineering at the University of California, Davis. She received her B.S. from Rutgers University, and her M. S. and Ph.D. in Electrical Engineering and Computer Sciences from the University of California, Berkeley. Her research interests include Internet measurements, network management, anomaly detection, online social networks, and vehicular ad hoc networks. She received the NSF CAREER Award in 2003, and the Outstanding Junior Faculty Award from the UC Davis College of Engineering in 2004. In 2008, she was named a Chancellors Fellow of UC Davis. She has served on the executive/technical program committee of several ACM and IEEE conferences and is currently an Associate Editor for IEEE/ACM Transactions on Networking.

Prasant Mohapatra is currently a Professor in the Department of Computer Science at the University of California, Davis. Dr. Mohapatra received his Ph.D. in Computer Engineering from the Pennsylvania State University in 1993. He was/is on the editorial board of several IEEE- and ACM-sponsored journals, including program Vice-Chair of INFOCOM 2004, and the Program Co-Chair of the First IEEE International Conference on Sensor and Ad Hoc Communications and Networks (SECON 2004). Dr. Mohapatra's research interests include wireless networks, sensor networks, Internet protocols and QoS. His research has been funded though grants from the National Science Foundation, Intel Corporation, Siemens, Panasonic Technologies, Hewlett Packard, and EMC Corporation.