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‡ and Prasant Mohapatra*
*Department of Computer Science. University of California, Davis, CA 95616
Email {cchchen,pmohapatra}@ucdavis.edu
†Microsoft, Redmond, WA
Email: {lyuan,albert}@microsoft.com
‡Department of Electrical Engineering. University of California, Davis, CA 95616
Email: chuah@ucdavis.edu

***Abstract*—In a multi-tenant data center environment, the current paradigm for route control customization involves a labor-intensive ticketing process, in which tenants submit route control requests to the landlord. This results in a 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 for multiple tenants to perform route control independently with little administrative involvement, and for the landlord to 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.

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 consist of 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 [1] show promising direction in tenant-directed control, allowing 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 end-user requests. For example, instead of a single server serving 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 ticketing process, which we outline below.

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

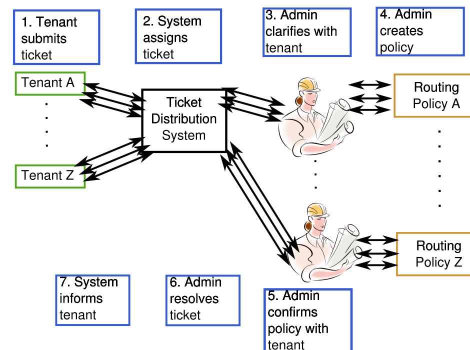


Fig. 1: Ticketing process.

Figure 1 shows a typical ticketing process for tenants requesting routing customization. A tenant first submits a request for routing customization (a "ticket") to a ticket distribution system, upon which a **landlord** (i.e., data center resource owner and/or manager, in this case a network administrator) is assigned to the ticket. After rounds of clarification between the tenant and landlord, the landlord sets up routing policies based on his understanding. Further clarifications might be required if the installed routing policy is unsatisfactory to the tenant. Finally, when both sides are content with the routing policy, the ticket is considered resolved and the routing customization request is considered fulfilled.

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 traffic scenario). In addition, reacting fast to changes in this paradigm means more tickets inundated to the ticket distribution system, resulting in overwhelming workload

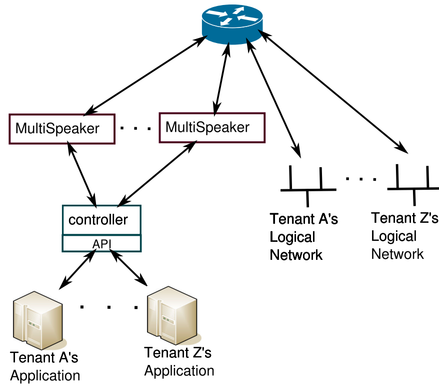


Fig. 2: RaaS overview with a single Controller set-up.

to 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 environment.

This paper proposes the Routing-as-a-Service (RaaS) framework. RaaS promotes automated route control to tenants while retaining the landlord's authority in setting the network policy. The RaaS architecture consists of **MultiSpeakers**, **Controllers**, and **Tenant Applications**, with the former two under the landlord's control and the latter maintained by the tenants.

Our contributions are threefold:

- 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 II).
- We build a prototype of RaaS (Section III) 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 evaluations 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 IV).

The paper proceeds with a system overview in Section II and the implementation in Section III. An implementation based on RaaS, along with a theoretical model for the service availability, are evaluated in Section IV. Related works are discussed in Section V and we conclude the paper in Section VI.

II. RAAS OVERVIEW

This section gives a high-level overview of RaaS, and introduces the components that enable tenant-directed route control. An overview of the system is shown in Figure 2.

A. Design Considerations

In designing the RaaS framework, we task ourselves to come up with a framework that not only allows tenants to customize their routing, but are able to do it safely. This means tenants can control their routing without unintentionally changing the routing policies of other tenants, or even worse, the overall network policies. In addition, components in the RaaS framework should not require major infrastructural overhaul, and should be flexible enough to be assembled in various configurations (e.g., 1+1 redundancy). We tackle these issues by first leveraging the capabilities of existing routing protocols [2]–[4]. Then, we design 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 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) [5] to install tenants' routing policies. MultiSpeakers provide an API for the Controllers to relay approved tenant routing requests to the router.

Deployment of redundant MultiSpeakers is easy in RaaS, since no communication occurs between MultiSpeakers – 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). This enables MultiSpeakers to be lightweight and stateless agents that simply act as relays for tenants to install their routing policy.

It may seem counterintuitive to use BGP, an inter-domain solution, for routing control within a single administrative domain. Indeed, Interior Gateway Protocols (IGPs) such as Routing Information Protocol (RIP) [6], Open Shortest Path First (OSPF) [7], and Immediate System to Immediate System (IS-IS) [8], [9] are IGPs that are well established and enjoy a wide adoption. However, there are several good reasons for using BGP, and they are outlined below.
Simple State Machine: Compared to protocols such as OSPF, 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 augmentation easier, as explored in Section III-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' placement flexibility. BGP supports a mode ("Multihop eBGP") that enables two BGP speakers to

exchange routing messages even if they are not directly connected. This makes it possible for MultiSpeakers to exchange messages to core routers, without having to directly connect to them.

Easy Resource Management: In RaaS, resource management equates to manipulating routing to specific physical resources (to be discussed in more detail in Section II-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 other BGP-learned routes that are also in the routers. If a tenant distributes the traffic over machines in several subnets, IGP would need to change link metrics to ensure the path metric to all servers are equal. Changing the link metric, however, can affect the egress point of BGP-learned routes, causing a ripple-effect to other ASes. Thus, using BGP avoids unintentional changes to the data plane while keeping resource mapping manipulation possible.

C. Controller

Before routing policies are received by MultiSpeakers, they must first pass through the Controller, as shown in Figure 2. The Controller provides an API for tenants to submit routing requests per their routing policy. By providing an API to tenants, RaaS lessens the need for landlord to dedicate large amount of time to change tenants' routing, since such a task can now be assigned to the Controller.

To prevent tenants from making erroneous routing requests, the landlord and tenants need to agree on the set of resources $\Re$ (i.e., servers) where the tenants can host the service. Upon agreeing on $\Re$, the landlord can implement policies that reject routing requests for resources 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. Because of the Controller, the landlord only needs to understand the constraints on tenants' routing policies, and can leave the actual routing policy implementation to the tenants. This reduces the amount of manual labor the landlord has to invest in allowing tenant to customize their routing.

In addition to providing an API and policy enforcement, the Controller also coordinates MultiSpeakers. When the Controller accepts tenant's routing requests, it records the requests and to which MultiSpeaker it is destined before forwarding the routing request. This helps the Controller check if duplicate routing requests have been received, 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, making the MultiSpeakers lightweight and stateless.

D. Tenant Application

Tenant application is the component that allows tenants to implement 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 tenant IP addresses (TIAs). These addresses will then be bound to the services tenants develop, and used for subsequent routing requests.

To control routing to their services, tenants issue API calls to the Controller to change the binding between TIAs and the set of resources available to the tenant. 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-resource binding). Tenants can now develop complex programs to install policies without landlord intervention.

With the use of TIAs and tenant applications, independent and safe route control is possible. For each routing request, the Controller checks the owner of a tenant application issuing 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, tenants' address spaces are mutually isolated, and tenant applications cannot modify routing to addresses they do not own; this prevents unintentional or malicious route hijacking by other tenants. Also, the tenant applications are separated, so each tenant can control routing to their resources independent of other tenants.

However, the actual resources being routed to are shared amongst tenants. For example, if $\Re_{Alice}$ = resources for Alice and $\Re_{Bob}$ = resources for Bob, $|\Re_{Alice} \bigcap \Re_{Bob}|$ could be greater than 0. This separation of virtual resources (i.e., the TIAs) and physical resources (i.e., servers) enables resource multiplexing amongst different tenants, while providing safe route control amongst tenants.

E. TIA-Resource Mapping and BGP

The discussion thus far presents tenant routing in the context of changing the TIA-resource 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 node the packets should use 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-resource mapping, the TIA address is represented by the IP prefix, and the resource is represented by the next hop address. Thus, to install a TIA-resource mapping, a BGP Update message to the router should be an announcement, with the TIA address being the IP prefix and the IP address of the resource being the next hop address. To change the TIA-resource 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-resource mapping. Alternatively, sending just a BGP UPDATE message with the new next hop

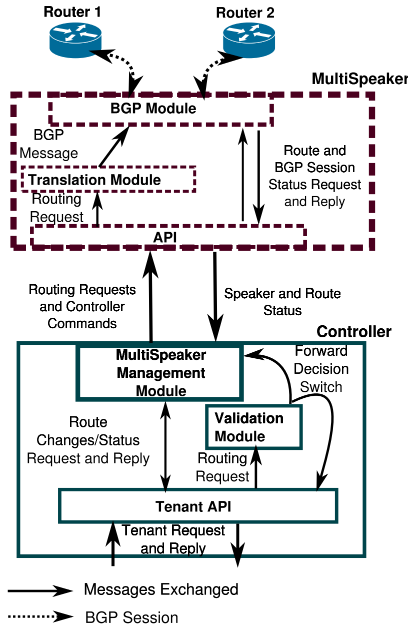


Fig. 3: MultiSpeaker and Controller components and interactions between them.

address will achieve the same effect, as the router will treat it as an implicit withdraw.

III. System Design

This section presents the design of the MultiSpeaker and Controller. Tenant application will be briefly mentioned, since the actual implementation is tenant-dependent. In addition, enhancements to the MultiSpeaker are possible and one such enhancement is presented here. The MultiSpeaker and Controller components and their overall interactions are shown in Figure 3.

A. Tenant Application

When tenants want to customize the routing policy to the resources ($\Re$) they have, 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-resources mapping (i.e., changing which resources service user requests), the applications can issue calls to the Controller's API. As mentioned in Section II-E, changing the TIA-to-resource mapping equates to changing the next hop of the destination. 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 $server_4$, 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.

Controller Interface	
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.

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 major API methods 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 and outputs a binary answer. 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. The validation module takes the route and token and checks whether the TIA and destination address belong to the tenant who owns the token, and marks the request valid/invalid pending the result. Then, the validation module passes the validation outcome to the MultiSpeaker management module and the tenant API.

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 entry of the table contains a (TIA, destination IP, action type) tuple, denoting an entry that the Controller and MultiSpeakers has to acknowledge as an entry that has been sent to the router. 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 transformation module.

The protected API implements 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, and so we omit it here.

The translation module takes tenant requests as input, and outputs well-formed BGP Update messages. For With-

drawRoute() 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 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 for 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 between routers and MultiSpeakers. 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 router notifications to reduce MultiSpeaker complexity. Using a BGP module, MultiSpeaker provides information isolation between the tenants and routers, much like BGP_MUX [10]. 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 set-up within the RaaS architecture, and interact with MultiSpeaker as if it is another BGP-capable speaker. This separation provides flexibility for the implementation of RaaS to vary with minimal impact to routers and the tenants.

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 3). Such a configuration would be fine if tenants only announce a single TIA-resource mapping at a time. However, in cases where tenants announce one-to-many TIA-resource 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-resource mapping. Intuitively, the equation above says the number of MultiSpeakers needed is the number of routers needing a BGP session, multiplied by the maximum count of one-to-many TIA-resource mapping needed by any tenant. If redundancy is required, an unmanageable amount of MultiSpeakers would need to be deployed. A simple extension to the BGP module could be implemented, in which **each BGP module instantiates multiple BGP sessions to the router**, with each session capable of announcing one TIA-resource 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 cross-communicate, MultiSpeaker complexity does not change much. We note that implicit withdraw (Section II-E) will not work, as router will treat it as another equal-cost multi-path (ECMP) route.

IV. Evaluation

In this section we evaluate the performance of our RaaS-based implementation via several micro-benchmarks. We present the methodology in Section IV-A and the evaluation results in Section IV-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 APIs, 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) [11] 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 route requests and collect the time taken for the Controller and MultiSpeaker to respond to the requests. 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 utilization.

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 largely unaffected because each incoming request is served in separate threads), we vary the parameters of the MultiSpeaker's configuration 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 waits for the MultiSpeaker to initiate BGP sessions and maintains the session by periodically sending KEEPALIVE messages.

	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.

We also demonstrate the feasibility by showing the success rate of tenant able to submit to the Controller on the first try is high with a small number of redundancies, given pessimistic settings for equipment uptime and replacement time. To do so, we use alternating renewal process (ARP) [12] to formulate a theoretical model for the availability of all equipments on the path from tenant to Controllers, 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 announcing and withdrawing 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 of sending the BGP message to the router is partly influenced by the 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. Adding to the fact that MultiSpeakers and Controller communicate to each other independent of tenant requests, this processing delay is only noticeable by tenants if they query for the route operation status immediately after submitting the route request.

Figure 4a shows MultiSpeaker's memory usage with respect to number of BGP sessions established. Here we do not distinguished 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 of the router. 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.

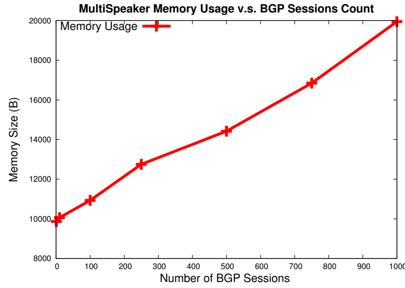
Figure 4b shows MultiSpeaker's memory consumption to store outstanding entries with various configurations of router per MultiSpeaker (RS) and ECMP session per router (E). In this experiment, we assume that the number of outstanding entries per sessions are the same across all BGP sessions. Memory usages 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×E outstanding entries maintained by the MultiSpeaker. 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 total number of outstanding entries to be as high as 4,000,000, unless the outstanding entries 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 configuration that amounts to little over 1,000 total BGP sessions at the MultiSpeaker, corresponding to the maximum memory usage shown in Figure 4a. Assuming the maximum ECMP possible (i.e 16), the MultiSpeaker is connected to 63 routers.

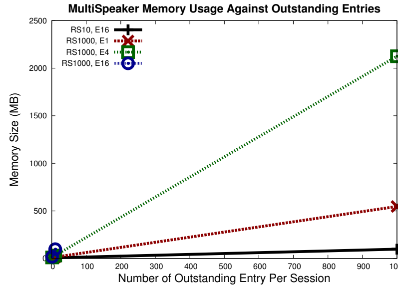
Table II shows the average and standard deviation of Controller's processing time for the AddRoute and RemoveRoute operations. It shows that both operations can respond to the tenant request within milliseconds of receiving the request, and thus can handle close to 1,000 requests per second on average. 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 4c 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≠1000, so we only plot one configuration here (S=4,RS=10). The plot shows that the memory consumption 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 nodes 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.

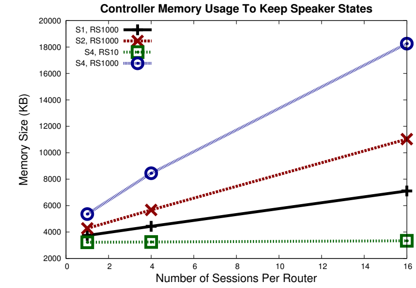
3) Network Overhead: In this experiment we are interested in observing the network overhead for the communication between tenant/Controller and Controller/MultiSpeaker. We



(a) MultiSpeaker memory usage with various BGP sessions.



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



(c) Controller memory. S = # of MultiSpeaker per Controller, RS = # of routers per MultiSpeaker.

Fig. 4: Memory Usage Results.

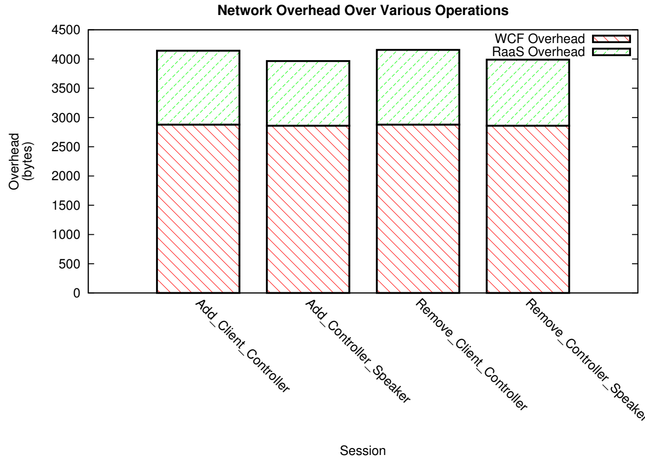
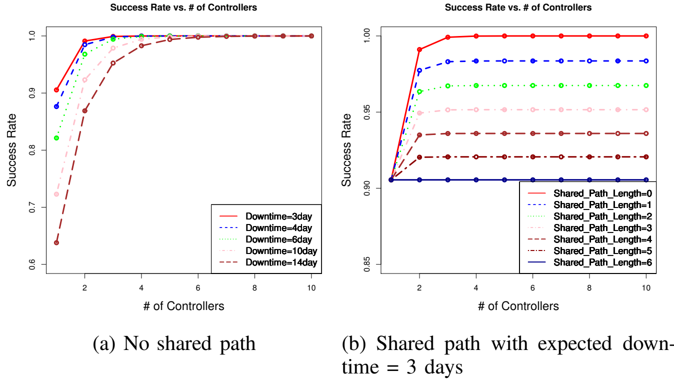


Fig. 5: Network Overhead.

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.

Figure 5 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 the usage of the WCF framework. Additional bandwidth can also be conserved by avoiding the use of the serialization engines in WCF [13], which converts all data into XML format.

4) Service Availability: Based on the derivation made in the appendix, we use R [14] to evaluate the number of distinct network subnets Controllers need to be deployed. We perform the evaluation by setting various values for the number of distinct subnets and expected downtime, and record the success rate. We use the Weibull distribution for the uptime distribution in Equation 4, 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 [15], 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 [16]. 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 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 distinct subnets for the request to be served. We assume that each distinct subnet has a completely disjoint path from other subnets. 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 6a. 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 in the gain, we increase the expected downtime and obtain the new success rates. We find the similar conclusion holds: the success rate converges to a stable value quickly and overall service remains highly available. We also see the gain in having multiple deployment can be significant when services are expected to be down for a prolonged period



(a) No shared path

(b) Shared path with expected downtime = 3 days

Fig. 6: Availability experiment results.

of time. In the case of expected downtime equal to 2 weeks, adding the service to another subnet increases the overall service availability by 30%.

The experiment 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 re-ran the experiments. Figure 6b shows the result for the case when expected downtime = 3 days. We find that the relative availability can be affected by up to 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, results in less personnels needed to process tenants' requests. In addition, RaaS components can be implemented on COTS hardware, making it easily deployable into data center servers; 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 increase network programmability to multiple tenants.

V. Related Works

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 [17], NIRA [18], Tesseract [19], RAS [20], PaaS [21], Morpheus [2], Transit Portal [4], and RCP [3] 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 ([18], [19]) require a technology that does 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 [17] is a proposed interface specification that can also implement RaaS-like framework. We note that however OpenFlow is more akin to a protocol specification, so a closer comparison should be drawn to BGP itself rather than RaaS; we also note that RaaS builds upon a more mature and widely available technology, making immediate deployment easier. Other works leverage existing routing technologies, such as BGP, to control routing either within a single AS [2] [3] or to various upstream ISPs [4]. 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 [20], 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 [21] 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.

On the industry side, services such as Amazon's EC2 [1], Internap's Performance IP [22], RouteScience's(RouteScience has been acquired by Avaya) [23] 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, and IP-to-VM mapping. RaaS differs from EC2 in that RaaS offers the underlying routing plane as service. 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 [24]. Since both of these solutions are intended for Internet-side route customization, there is no programmable API for tenant to implement their own route-control logic.

VI. Conclusion

The traditional paradigm for route 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 within a second of submitting the request. In addition, we also showed that it is possible to offer more aspects of the

data center as a service without major infrastructure 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. 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 services 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 the services 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) [25]. 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 nth 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

Due to space limitation, we state the theorem here without showing the complete proof.

Theorem 1. *Given a path p to the services, where p is a 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:*

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

To prove the theorem, we have to find the limiting interval reliability, in which the result is shown in [25]. The theorem can then be obtained by applying the independence argument to the starting equation and re-arranging the terms.

REFERENCES

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

[2] 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.

[3] 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.

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

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

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

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

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

[9] D. Oran, "OSI IS-IS Intra-domain Routing Protocol – RFC 1142," 1990.

[10] 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.

[11] Microsoft, "Windows Communication Foundation," 2006.

[12] V. G. Kulkarni, *Modeling and Analysis of Stochastic Systems (Chapman & HallCRC Texts in Statistical Science)*, 1996.

[13] Microsoft, "Windows Communication Foundation, Data Contracts," 2006.

[14] 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.

[15] Cisco, "Data Center Design IP Network Infrastructure," 2009.

[16] R. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122 (Standard), Internet Engineering Task Force, Oct. 1989, updated by RFCs 1349, 4379, 5884.

[17] 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.

[18] 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.

[19] 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.

[20] J. D. Hobby, "Routing as a service," University of California, Berkeley, Computing Science Technical Report EECS-2006-19, 2006, available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-19.pdf.

[21] 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.

[22] Internap, "Network-Based Service: Performance IP."

[23] Avaya, "http://www.avaya.com."

[24] G. Goddard and R. Vaughn, "Review: RouteScience's PathControl," 2002.

[25] R. E. Barlow and F. Proschan, *Mathematical Theory of Reliability*. Society for Industrial Mathematics, 1987.