Rethinking Cloud Service Marketplaces

Zhe Wu and Harsha V. Madhyastha
University of Michigan

ABSTRACT

Services offered by cloud providers (e.g., for key-value storage and for redirecting clients to nearby data centers) simplify the development of applications deployed in the cloud. However, since any cloud provider's service offerings are far from complete, many cloud providers host a service marketplace, wherein third parties can advertise new services.

In this paper, we argue that existing cloud service marketplaces need to be redesigned so that third-parties can not only offer new independent services but also augment the capabilities of existing cloud services. Enabling efficient implementation of such third-party add-ons requires a careful examination of the API that the cloud provider should expose and the manner in which add-ons can be interposed on interactions between applications and cloud services. We discuss how recent advances in software-defined networking with P4 can be leveraged to facilitate this redesign of cloud service marketplaces.

1. INTRODUCTION

Deploying applications in the cloud is an attractive proposition due to the opportunities available for reducing cost: usage-based pricing, ability to scale up and down on demand, economies of scale, and multiplexing of resource usage. In addition, the use of the cloud is also desirable because cloud providers offer many services that greatly simplify application development; examples include services to store data at scale (such as Azure virtual hard disk) and services to redirect users to nearby data centers (such as Amazon Route 53). In fact, the range of services offered by Amazon AWS is considered one of the key reasons for it being the market leader in cloud infrastructure [9].

However, the service offerings by any cloud provider are far from complete. Many functions that are offered by one cloud provider are not offered by others. For example, Azure's key value storage service enables tenants to acquire per-key leases to mediate concurrent operations on the same key [4], whereas Amazon's equivalent S3 service offers no support for concurrency control. Moreover, there are several other functions lacking in all cloud services, such as support for consistent geo-replication of data. Thus, many applications are each redundantly forced to implement the same functionality in the absence of support from the cloud provider.

Perhaps recognizing this gap between what they offer and what application developers desire, many cloud providers support a marketplace of third-party services [1, 6, 7]. The marketplace lists a variety of functionality that have been implemented by third-parties but recognized by the cloud provider as being compatible with application deployments on its platform. Any application deployed in the cloud can therefore not only use services offered by the cloud provider but also those available in the marketplace, thereby simplifying the development of the application.

A service marketplace offers several benefits. First, like any service offered by cloud providers, a third-party service, once implemented, can be reused across all applications. Second, since cloud providers vet every service listed on their marketplace, applications can use the services offered in a marketplace without having to vet them on their own. Third, the marketplace will democratize the ability to add to the cloud's functionality and thereby lead to a greater rate of innovation than when all the onus lies on the cloud provider. This in turn will help cloud providers attract more applications, thus increasing revenue.

However, existing cloud marketplaces suffer from a key limitation: they only enable third-parties to offer new functionality that is independent from any service offered by the cloud provider. Third-party functionality listed on the marketplace can augment end-host capabilities (e.g., a virtual machine image), expose APIs to new services (e.g., in-memory data storage [3]), or interpose on an application's traffic either between its virtual machines (VMs) in the cloud or between its private and public deployments (e.g., to eliminate redundancy in network traffic [8]). To the best of our knowledge, no existing cloud marketplace enables third-parties to augment the functionality of existing cloud provider-offered services. This is a significant shortcoming because, as we describe later, many of the limitations in today's cloud services stem from the need to either add new operations to a service or to interpose on an application's interactions with a service, not just traffic between VMs.

Redesigning cloud service marketplaces such that third-parties can augment existing cloud services, and not simply offer new independent services, is however far from straightforward. To see why, consider a third-party add-on that adds the ability to acquire and enforce key-specific leases. To ensure that only the lease holder can modify the value for a key, simply relaying all interactions between any application and

HotNets-XV, November 09 - 10, 2016, Atlanta, GA, USA

 ISBN 978-1-4503-4661-0/16/11…$15.00

DOI: http://dx.doi.org/10.1145/3005745.3005774

cloud storage via the add-on would make it onerous for the add-on's developer to support the scale necessary to sustain the aggregate storage throughput of all applications using the add-on. Whereas, having every application's VMs first validate their lease with the add-on's VMs before issuing a write to cloud storage would not only increase write latency but does little to prevent the holder of an invalid lease from updating a key. In addition to these reasons, enabling the use of third-party add-ons is further complicated by the fact that an application may wish to use multiple add-ons in combination (e.g., first verify that lease is valid and then confirm that the value for the key being modified has not been updated since the writer last read the key), despite every add-on's implementation being oblivious to others.

For cloud providers to facilitate a rich set of add-ons, while maximizing efficiency for both applications and add-on providers, our key insight is to enable the implementation of add-ons as middleboxes that are placed off the data path but can be interposed on application-cloud service interaction when necessary. Leveraging recent advances in software-defined networking (SDN) with P4 [11], our design has application VMs and cloud services insert custom add-on headers into the subset of packets that switches need to hand off to add-on VMs. Switches in the cloud network use P4 programs provided by add-on providers to appropriately forward packets either to add-on VMs or to the destination cloud service.

In this paper, we 1) motivate the need for cloud marketplaces to support add-ons with examples of functionality that are currently lacking in popular cloud services, 2) present our P4-based design for efficient and flexible realization of add-ons, and 3) discuss several challenges that cloud providers will need to tackle to apply our design in practice. Overall, our vision for redesigning cloud marketplaces to enable third-parties to not only offer new services but augment the functionality of existing cloud services offers the potential to significantly accelerate the rate of innovation in cloud infrastructure, ease the development of applications deployed in the cloud, and thereby make the use of cloud infrastructure even more attractive than it is today.

2. MOTIVATION

Taking a geo-distributed webservice as an example of an application that would be deployed in the cloud, we begin by highlighting application-desired functionality currently missing in cloud services. We then argue why such functionality is hard for third-parties to implement without support from cloud providers.

2.1 Examples of missing functionality

As shown in Figure 1, the presence of several geographically distributed data centers in popular cloud platforms makes it attractive for webservices to be deployed in the cloud. The webservice provider can deploy user-facing webservers (called front-ends) in VMs at every data center so as to serve any user from the nearest data center. In addition, webservices that enable users to share content (e.g., online social networks, collaborative document editing) can geo-replicate data, storing user uploaded content in the local storage service at every data center. Geo-replication enables front-ends to read or write any data item by accessing a subset of nearby replicas, rather than a centrally located copy.

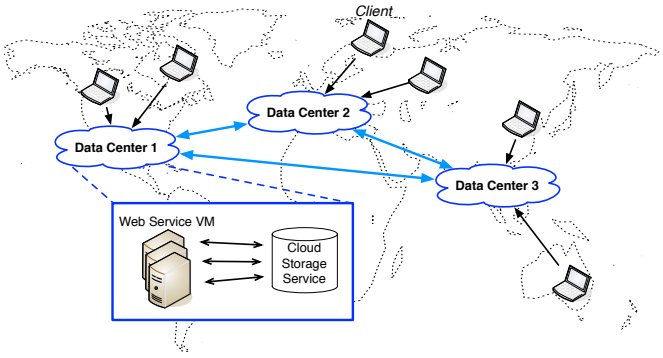

Figure 1: Generic architecture of a geo-distributed webservice deployed in the cloud.

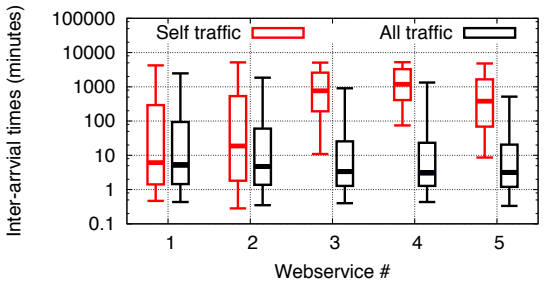

Figure 2: Inter-arrival times for a webservice's clients in its own traffic and in traffic from all webservices (95^{th}, 75^{th}, 50^{th}, 25^{th}, and 5^{th} percentiles across prefixes shown).

Need for different abstraction. To aid webservice providers in redirecting users to nearby data centers, cloud providers offer redirection services such as Amazon Route 53 and Azure DNS. These services let a webservice register a DNS name, and any client that attempts to resolve this name is redirected to the data center to which it has the lowest latency among the ones on which the webservice is deployed.

However, leaving the task of redirecting clients to an independent redirection service is not always desirable. For example, to enable more fine-grained redirection policies that account for a webservice's replication of data (e.g., redirect requests for photos older than a year to one of three data centers that serve as backups for cold data), the task of managing redirection is best left to individual webservices. Whereas, for webservices that seek to spread their deployments across multiple cloud providers given the associated performance and cost benefits [17, 13], no provider has the incentive to support redirection to external data centers. Webservices that seek to retain control over client redirection for such reasons will need to dynamically modify their redirection policy in reaction to changes in the performance and availability of Internet paths between clients and data centers.

Though every webservice could potentially monitor the Internet paths to its clients itself, monitoring would be more efficient and timely if the cloud provider offered this as a service. This is because webservices often have overlaps in their paths of interest. For example, based on the traffic of 186 webservices hosted on a multi-tenant platform, we observe that there exist significant intersections between the sets of IP prefixes in which different webservices have clients; almost half of the 143K prefixes interact with multiple webservices, and over 25% of them access at least 5 webservices each. Since the cloud provider is privy to the Internet traffic of all applications hosted on its platform, a shared Internet path monitoring service will reduce the need for probing as measurements from the passive monitoring of any webservice's traffic can be shared with all other webservices. For example, for 5 (chosen at random) of the 186 webservices mentioned above, Figure 2 shows that the frequency of traffic from client prefixes increases by up to 3 orders of magnitude when considering the aggregate traffic across all webservices.

Need for additional operations. When a webservice's front-end receives a request from a user, it may need to read or update geo-replicated data in order to serve the request. If strong consistency is desired (e.g., Google Docs), the subset of replicas that a front-end writes to must overlap with the subsets that any other front-end reads from or writes to. In this setting, since typical webservice workloads are dominated by reads [10], it is desirable that any front-end reads from a subset of replicas smaller than the subset it writes to. In the extreme, every front-end can read any data item from the closest replica but update all replicas when writing.

Now, between any front-end reading an object and writing back its update, another front-end may potentially have updated the object in the meanwhile. Hence, to preserve linearizability of writes, any front-end will need to issue conditional-PUTs to update an object, so that its update is successful only if the version of the object in cloud storage matches that it previously read.

For this, it would be ideal to have a conditional-PUT operation which lets a front-end specify the hash of the content that it believes it is overwriting and the operation succeeds only if this matches the content currently stored. Instead, on legacy cloud storage services that do support conditional-PUTs (e.g., Azure), the condition can be a function only of either the last modified time or an opaque ETag header.[1] Since both the last modified time and the ETag header can only be discovered from the cloud storage service's response to a GET request, a front-end needs to have previously read a replica of an object in order to update that replica with a conditional-PUT. As a consequence, if every front-end reads from a smaller subset of an object's replicas than the subset it writes to, when a front-end seeks to update an object, it first incurs one round trip of wide area communication to read from replicas that it had not previously read from and a second round trip thereafter to update replicas.

We quantify the impact of these additional reads on write latency for a webservice that deploys front-ends and stores data across all Azure data centers. We consider replication of every object at 5 data centers, with every front-end reading an object from the closest 2 replicas and writing to the closest 4 replicas. Based on the subset of front-ends that can access an object—we refer to this subset of front-ends as the object's *access set*—we pick the 5 data centers at which to store replicas of the object so that the maximum read latency across the front-ends in the access set is minimized. Across every front-end in all possible access sets, we see that the need to read from distant replicas before updating an object doubles median write latency from 150ms to 300ms.

Need for modified protocol. For a webservice that geo-replicates data, inter-data center transfers can account for a significant fraction of cost. One way to reduce these costs is via redundancy elimination [16], i.e., when any chunk of data that has previously been transferred between a pair of data centers needs to be sent again, it suffices to transmit only the content hash and size of the chunk.

While an application can apply such redundancy elimination on the communication between its own VMs, doing the same for traffic between the application and a cloud service is not feasible without the cloud provider's support. For example, when an application VM in one data center writes data to cloud storage in another data center, the storage service must be able to recognize when the data it receives is the content hash and size of a chunk it has previously received or sent out. Similarly, when responding to a read request from a remote data center, the storage service must send only the metadata and not the actual data for a chunk that has been previously transmitted to the same VM.

2.2 Challenge for third-party implementations

Our examples of desirable functionality lacking in cloud services are by no means comprehensive. As another example, for concurrency control on cloud storage, an application may wish to enforce that its VMs acquire an object-specific lease before modifying any object, a capability that some cloud services offer (e.g., Azure) but not others (e.g., AWS). Our goal is to enable third-parties to implement any such functionality desired by applications, rather than application providers being completely dependent on cloud providers.

On-path middleboxes impose high cost. Since all the examples we have discussed require either interposing on or altering the interaction between applications and cloud services, one way to augment cloud services would be to have third-parties run middleboxes that sit on the data path between applications and cloud services, as shown in Figure 3(a). Such a middlebox can monitor network performance metrics, implement new operations on behalf of a cloud service, or modify the protocol for communication with the cloud service.

[1]The HTTP specification recommends that a client make no assumptions about how a server generates ETag headers [2].

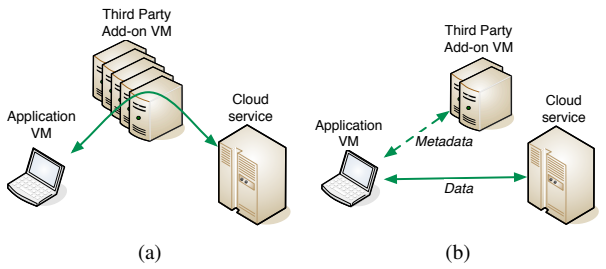

Figure 3: Third-party add-ons (a) as on-path middleboxes imposes high cost on add-on providers, whereas (b) deploying them off-path either degrades performance for applications that use the add-on or renders some functionality infeasible to implement.

The primary downside is the high cost this approach imposes on add-on providers, since they must deploy a sufficiently large number of VMs to support the throughput of interactions between applications that use their add-on and the cloud service that their add-on augments. This is unnecessary in many cases because there is often no need for the add-on to inspect all data exchanged between applications and a cloud service. For example, for shared Internet path monitoring, it suffices for the add-on to learn the timings and sizes of packets exchanged between external users and cloud services, and inspecting packet content is unnecessary. Similarly, an add-on that hands out and enforces leases needs to confirm that a VM holds the corresponding lease when executing a write but does not care about the data being written.

Off-path add-ons degrade performance or functionality. Alternatively, add-ons can be implemented such that they do not sit on the data path between applications and cloud services, but instead communicate off-path with either or both (Figure 3(b)). For example, one can implement as follows an add-on that offers the new form of conditional-PUT previously described. Whenever an application seeks to issue a conditional-PUT, it can first check with the add-on's VM that its condition is true and that there is no other ongoing attempt to update the same object. Thereafter, it can issue the PUT to update the object in cloud storage. The application VM can then inform the add-on's VM whether its update was successful. When using A1 instances for such an implementation on Azure, we find that we would need $\frac{1}{6}^{th}$ the number of VMs as we would when relaying all PUTs through the add-on's VMs.

However, this alternate realization of add-ons introduces other limitations. On the one hand, it can degrade performance. For example, when executing an application VM in one data center seeks to execute a conditional-PUT on data stored in another data center, having the VM first check with the add-on's VM in the remote data center that the condition is true and then execute the PUT doubles write latency. On the other hand, having add-ons off-path makes some functionality infeasible. For example, an add-on cannot enforce that an application VM that does not hold the lease for an object cannot update the object in cloud storage.

3. DESIGN

The discussion in the previous section highlights that our main challenge is to address the following conflict: to maximize efficiency and minimize cost for add-on providers, it is desirable that the VMs which implement an add-on's functionality not be on the data path between applications and cloud services, yet this degrades application performance and renders some functionality infeasible to implement.

To reconcile these conflicting requirements, we exploit support from the network. Our key insight is to instrument switches on the path between application VMs and cloud services so that these switches can flexibly interpose add-on VMs on or inform them of application-cloud service interaction when appropriate. This requires switches to not only route packets to their destination, but to also recognize packets that need to be handled by add-ons and send a version of such packets (e.g., the entire packet, a portion of the header, or a digest of the payload) to the relevant add-on's VMs.

Recent advances in software-defined networking with P4 [11] enables the efficient realization of our vision. Unlike traditional SDN switches, which only have a programmable control plane, P4 enables programs written in a high level language to be installed in the data plane. In our setting, every add-on developer provides a P4 program, which the cloud provider installs in its switches.

To leverage the flexibility enabled by P4 in implementing third-party add-ons, we must answer three questions.

- First, how to enable switches to identify traffic that needs to be handled by an add-on?
- Second, how to ensure that third-party add-ons can efficiently implement their desired functionality by minimizing the fraction of traffic handed off to their VMs?
- Lastly, how to enable the composition of add-ons?

3.1 Edge-informed routing for add-on intervention

When a switch on the path between an application VM and a cloud service receives a packet, the switch must identify which third-party add-on, if any, needs to process the packet. Unlike prior work on using SDN to integrate middleboxes into end-to-end communication [15], the key challenge here is that the information that identifies any traffic's relevance to an add-on is only available at the application layer. For example, for an add-on that implements the previously described content hash based conditional-PUT, only the traffic corresponding to an application VM's invocations of the new form of conditional-PUT is relevant, but not GETs or other types of PUTs. But, having switches inspect packet payloads (e.g., to identify whether a conditional-PUT request is being issued, and if so, of which type) would dramatically decrease the network's throughput. Therefore, it is necessary that a switch be able to figure out what to do with a packet just by inspecting the packet's headers.

Unlike recent work [14], rewriting MAC addresses to enable flexible routing is not an option in our setting. Encod-

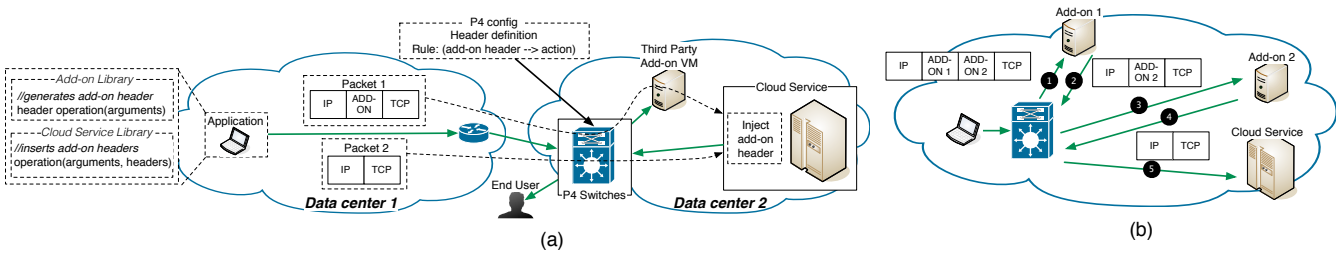

Figure 4: **(a) Illustration of cloud support to enable efficient and flexible implementations of add-ons. (b) Composition of add-ons.**

ing information into the MAC address is feasible only when both end-points—in our case, an application VM and a cloud service—are in the same layer-2 network. Whereas, as we have seen in the previous section, a geo-distributed webservice's front-end may have to read from or write to cloud storage in a different data center.

Therefore, encoding of information as to when switches inside a cloud provider's infrastructure need to hand off traffic to an add-on's VMs must be encoded at the IP layer or above. Using IP options is not an option, as space in the IP header is limited, and packets with IP options are often dropped [12]. Instead, we include an additional header (as shown in "Packet 1" in Figure 4(a)) between the IP and transport headers. This optional intermediate header includes the necessary information for switches that sit in front of a cloud service to recognize if a packet must be handed off to an add-on's VMs, and if so, which one.

But, how is this optional add-on header inserted into packets? We propose that every add-on provide an application library that includes every cloud service operation that the add-on augments. Every function in this add-on library, when invoked, returns an appropriate add-on header. When the application invokes the corresponding function in the library provided by the cloud service, it provides this add-on header for insertion into packets sent out. Whereas, for traffic sourced from the cloud service, it looks up the tenant corresponding to the traffic and uses its knowledge of that tenant's selections in the cloud marketplace to appropriately rewrite packets. In either case, as shown in Figure 4(b), add-ons strip the header specific to them before forwarding packets, and therefore the destination of any packet can be oblivious about the add-on.

3.2 Implementing add-on functionality

Though a switch can use a packet's add-on header to identify the specific add-on for which the packet is relevant, three questions must be addressed in order to leverage this capability for maximizing the efficiency of add-ons: 1) which packets in the traffic between an application and a cloud service should include the add-on header, 2) when a switch does find a particular add-on's header inside a packet, what information about the packet should the switch share with the add-on's VMs, and 3) how should add-ons process the packets they receive in order to implement the functionality they offer? We address these issues jointly in two ways.

Add-on specific metadata in add-on header. First, as discussed previously, having an add-on inspect all traffic in an application's interaction with a cloud service can be both expensive and unnecessary. Therefore, we seek to minimize the number of packets handled by any add-on's VMs.

To meet this goal, we augment add-on headers to not only specify the add-on that must handle a packet but to also include add-on specific metadata. This metadata, which switches do not need to interpret, serves as a way of either end-point—application VM or cloud service—communicating add-on specific information to the add-on's VMs. For example, in the case of the add-on that adds a new type of condition for conditional-PUTs to cloud storage, the add-on header can include the key being written to, the hash of the value that the application VM believes is currently stored for this key, and the hash of the new value that it is attempting to write. As a result, when an application VM issues a conditional-PUT, it needs insert the add-on header only into the first data packet and have the add-on's VMs inspect only this packet, rather than requiring the add-on's VMs to parse all the data being written and compute the hash of the new value being written. Similarly, for an add-on that implements leases, the add-on only needs to inspect the first packet of any PUT operation; that packet's add-on header can specify the key being written to, and the add-on can check whether the source of this packet currently holds a lease for this key.

Packet processing by add-ons. For some add-ons, when a switch identifies a packet that matches, the switch only needs to inform the add-on's VMs of a portion of the packet's contents (e.g., the packet's IP header in the case of the Internet path monitoring add-on) and forward the packet to its destination. For many other add-ons, the switch will need to forward the packet to the add-on's VMs instead of to the packet's destination. In such cases, if the add-on eventually delivers the packet unmodified to the destination, the application's interaction with the cloud service works as it would without the add-on. For example, with the redundancy elimination add-on described earlier, one of the add-on's VMs in the source data center may rewrite a packet's contents to include only the content hash and size of the packet. But, another of the add-on's VMs in the destination data center would replace the packet with its original contents before forwarding to the destination.

However, there do arise cases where an add-on needs to alter the interaction between an application VM and a cloud service. For example, consider the add-ons that implement leases or the content-based conditional-PUT operation. When

a PUT should not be permitted to proceed (the client VM issuing the PUT does not hold the lease or the condition does not match), the client needs to receive a negative acknowledgment and the PUT operation should not execute on cloud storage. If the add-on VM itself sends to the client a negative acknowledgment on behalf of the storage service, the client-side and storage service-side TCP connection states would be out of sync in terms of the amount of data sent from the service to the client. The client and the storage service therefore cannot continue to communicate on this TCP connection, and the connection must be torn down. Thus, the client's subsequent operation to cloud storage would incur the additional latency of re-establishing a TCP connection with the storage service.

The solution to this issue is that, whenever an add-on needs to alter a cloud service's response to an application VM, it should elicit that response from the cloud service rather than generating the response itself. An add-on's VM can modify packets before it forwards them, but it must preserve the size of every packet so that both ends of a TCP connection are consistent with each other with respect to sequence numbers. For example, consider the case of the conditional-PUT add-on. If the add-on receives a valid request, the add-on VM will forward the first packet unmodified to the cloud storage service, and switches will forward the remaining packets of the request to the service without the add-on's intervention. Whereas, for invalid conditional-PUTs, the add-on's VM must rewrite the first packet so that the request will be rejected by cloud storage, thereby causing it to send back a negative acknowledgment to the application.

3.3 Composing add-ons

In some cases, it is desirable to interpose multiple add-ons on a single request from an application to a cloud service. For example, a PUT request can first undergo lease validation by one add-on, then have its associated condition checked by another add-on, before the request finally being submitted to cloud storage. However, add-ons are developed independently and are therefore oblivious to other add-ons.

We observe that the solution is to allow for a stack of add-on headers in any packet. When invoking a particular operation on a cloud service, an application can get the add-on headers from the libraries of all the add-ons it is using to augment this operation. The application can then pass on this set of add-on headers in the appropriate sequence when invoking the operation on the cloud service's library.

As shown in Figure 4(b), when a P4 switch receives a packet, it inspects the first add-on header and takes the appropriate action to process the packet, e.g., sending the packet header to the add-on's VMs. After the packet is processed in one of the add-on's VMs, the VM will remove its own add-on header before putting the packet back on to the wire. The network will now process the packet based on the new first add-on header in the packet. If no more add-on headers exist, the packet is forwarded to its destination.

4. OTHER CONCERNS

Security and privacy. A cloud provider can use several approaches to address potential security and privacy concerns. On the one hand, the onus of inserting add-on IDs into add-on headers can be left to the cloud service's library to prevent malicious add-ons from spoofing. On the other hand, to prevent exposure of an add-on's header to other downstream add-ons, every add-on header can be split into two parts: an add-on ID in plain text, and add-on specific metadata encrypted by the tenant's private key and the add-on's public key. In addition, instead of having add-ons manage their own VMs, cloud providers can run and manage the deployment of add-on code, so that every add-on will only deal with its own header and have no ability to inspect and change other add-ons' headers.

Testing and monitoring. Unreliable offerings in a cloud service's marketplace reflect poorly on the cloud provider. Therefore, cloud providers must verify the quality of third-party add-ons. For example, Azure has a certification program [5] that third-party offerings must pass before they are listed in Azure's marketplace. Testing of a third-party service should combine application-layer test cases that examine the responses when the API exported by a third-party is used and inspection of network traffic to confirm that the add-on correctly manipulates packet headers both to implement its claimed functionality and for it to be compatible with other add-ons. Furthermore, if add-on providers manage their own deployment, cloud providers must monitor add-on deployments to detect and flag availability and performance concerns.

Pricing. Our design enables many add-ons to charge per-operation (e.g., the previously discussed add-ons for supporting conditional-PUTs and leases need to inspect only the first data packet of every write to storage), as opposed to the per-byte pricing that would be necessary if add-ons were implemented as on-path middleboxes. Add-ons that do need to be informed of every packet (e.g., shared Internet path monitoring or redundancy elimination) will have to offer per-byte pricing even with our design, but will incur significantly lower cost thanks to not being on the data path, thereby making them cheaper for customers.

5. CONCLUSIONS

In this paper, we presented a redesign of cloud service marketplaces to enable efficient deployment and use of third-party add-ons. Leveraging P4, our design exploits support from the network such that an add-on's VMs can interpose on an application's interactions with a cloud service, but need to do so only for a portion of the application's traffic. The net effect of our proposed redesign of cloud service marketplaces is a win-win for all parties involved: cloud providers benefit from a greater rate of innovation, applications can avail richer functionality without sacrificing performance, and third-party developers incur low cost.

6. REFERENCES

[1] AWS Marketplace. https://aws.amazon.com/marketplace.

[2] Hypertext transfer protocol (HTTP/1.1): Conditional requests. RFC 7232.

[3] MemCachier on Microsoft Azure Marketplace. https://azure.microsoft.com/en-us/marketplace/partners/memcachier/memcachier/.

[4] Microsoft Azure - Lease blob. https://msdn.microsoft.com/en-us/library/azure/ee691972.aspx.

[5] Microsoft Azure Certified. https://azure.microsoft.com/en-us/marketplace/programs/certified/.

[6] Microsoft Azure Marketplace. https://azure.microsoft.com/en-us/marketplace/.

[7] Rackspace Marketplace. https://marketplace.rackspace.com/.

[8] Riverbed SteelConnect Gateway on AWS Marketplace. https://aws.amazon.com/marketplace/pp/B0163DOLQ6.

[9] Why AWS dominates the cloud services market. http://thenextweb.com/offers/2016/03/11/amazon-web-services-dominates-cloud-services-market/.

[10] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3):87–95, 2014.

[12] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. IP options are not an option. Technical Report UCB/EECS-2005-24, 2005.

[13] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon. Supercloud: Opportunities and challenges. *ACM SIGOPS Operating Systems Review*, 49(1):137–141, 2015.

[14] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *NSDI*, 2016.

[15] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using SDN. In *SIGCOMM*, 2013.

[16] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *SIGCOMM*, 2000.

[17] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.