Network Working Group
Internet Draft
Intended status: Informational
Expires: July 2012

A. Dalela
Cisco Systems
M. Hammer
January 4, 2012

Service Orchestration Protocol (SOP) Requirements
draft-dalela-orchestration-00.txt

Status of this Memo

Copyright Notice

Abstract

Cloud services need to interoperate across cloud providers, service vendors and private/public domains. To enable this interoperability, there is need for a standard protocol for exchanging service information. This draft describes requirements for such a protocol. Current cloud implementations expose application level APIs, which are not syntactically and semantically compatible with each other. One approach to interoperate cloud services is to standardize the protocol while leaving the API definition implementation specific. Standard protocols have been used widely in the Internet and can be extended to cloud. Use of such protocols is compatible with existing cloud APIs, which can exchange information in a standard protocol. New APIs may also be developed using a standard protocol. By this, it would be possible to interoperate diverse APIs across service providers, service vendors and service users.

Table of Contents

1. Introduction

Cloud computing has become important for an on-demand delivery of a variety of services, broadly called XaaS, such as Infrastructure, Platform and Software as a Service [NIST]. Users of such services may be individuals, enterprises, content providers, or other cloud providers. These users need to be able to request and manage services seamlessly across private, public, hybrid, or community clouds. Lack of interoperability across these domains will lead to new kinds of cloud silos, which will in turn hinder economies of scale.

Current cloud deployments use web-services (SOAP or REST) to deliver services over the Internet. Each provider exposes different APIs that generally do not interoperate, because each API has different syntax and semantics. To interoperate, we must either converge on one API format, or translate between them. Both alternatives are hard. API translations are difficult because APIs have different semantics. Converging to one API means current services may be broken. We want to maintain diverse APIs, while enabling interoperability.

Historically, in Internet, different APIs have interoperated through use of standard protocols. Basically, we separate the network view of information from the application view. Network carries information via protocols while applications consume information via APIs.

Web-services equate the network view of information with the application view. Basically, each API has its own packet format which is derived from the API, and changes to API syntax or semantics will change the packet format. This is at the root of interoperability issues. As applications proliferate, each API will project its view of information into the network. As a result, there will be as many communications "protocols" as there are applications. This is contrary to the (unstated) assumption in Internet that there are far fewer protocols than there are applications, so that many applications can communicate using the same protocols.

To remedy this problem we should separate the network and application views of information and design them independently. Applications may design APIs in many ways and two applications should communicate using a standard protocol whether or not they use the same API. This document describes requirements for such a standard protocol.

2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying RFC-2119 significance.

3. Terms and Acronyms

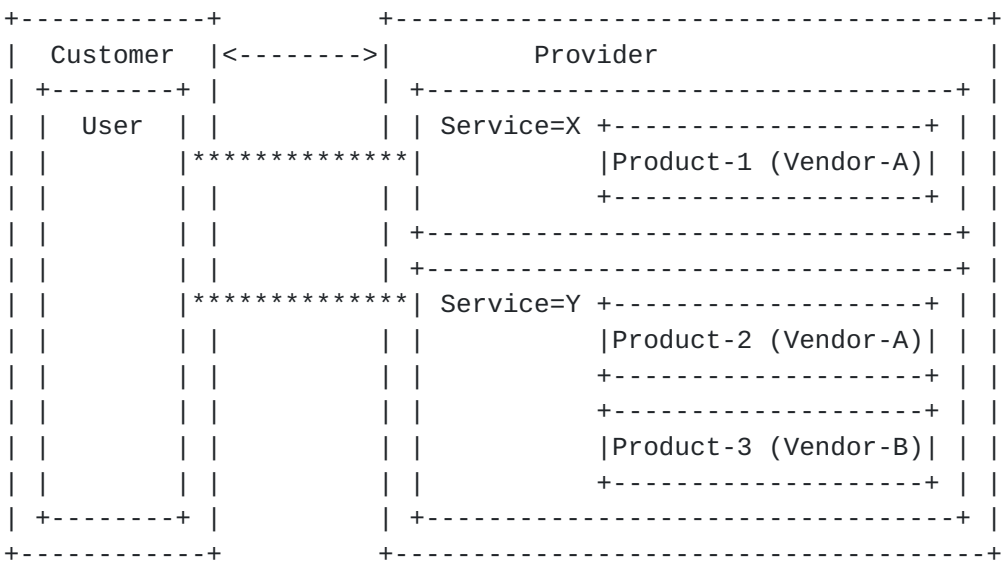

Fig-1: Cloud Ecosystem and Relations

Provider: A Provider is a supplier of cloud services who offers these services to cloud Customers and Users, per some business agreement.

Service: Any virtual instance of a hardware or software product that can be owned by a Customer or User for their personal use.

Vendor: A Vendor is a hardware/software product vendor who provides the technology implementation of a service. In some cases, Providers and Vendors may be the same business entity.

Product: A unit of software or hardware entity that is sold by the Vendor to the Provider to be made available as a service.

Customer: A business entity that enters into an agreement with a Provider to source cloud services for their users. A customer would be an enterprise that buys cloud services. A customer would define

policies for service and may authenticate its users. Customers may also be called Subscribers of a cloud provider's services.

User: A user is the end consumer of cloud services. Users belong to the customer and place service requests on the provider. These requests are controlled by Customers who could be enterprises, individuals or other cloud providers who source services from one cloud and provide them to another.

Virtual Provider: A Provider who does not host or manage services, but redirects requests to other providers who do that. A Virtual Provider has customers but does not operate services.

Orchestration: This is the act of creating, modifying, moving or deleting services. It may involve one or more actions performed in sequence or in parallel. These actions could be invoked on hardware and software services, or even on other cloud providers.

Service Domain Name (SDN): This is a dotted-decimal notation to represent service names hierarchically. For example, a virtual machine can be represented as iaas.compute.virtual. Each SDN will be associated with a set of service specific attributes.

4. Interoperability Scenarios

The following interoperability scenarios should be covered by the protocol. We list them here because depending on the context interoperability may mean different things.

Scenario S-1. Users Interoperate Across Cloud Providers. Users must be able to use cloud services from different cloud providers in the same way. This allows a user to move across providers, or source the same service in a different geography from a different provider. In Figure-2, a user accesses the compute service across various providers in the same way.

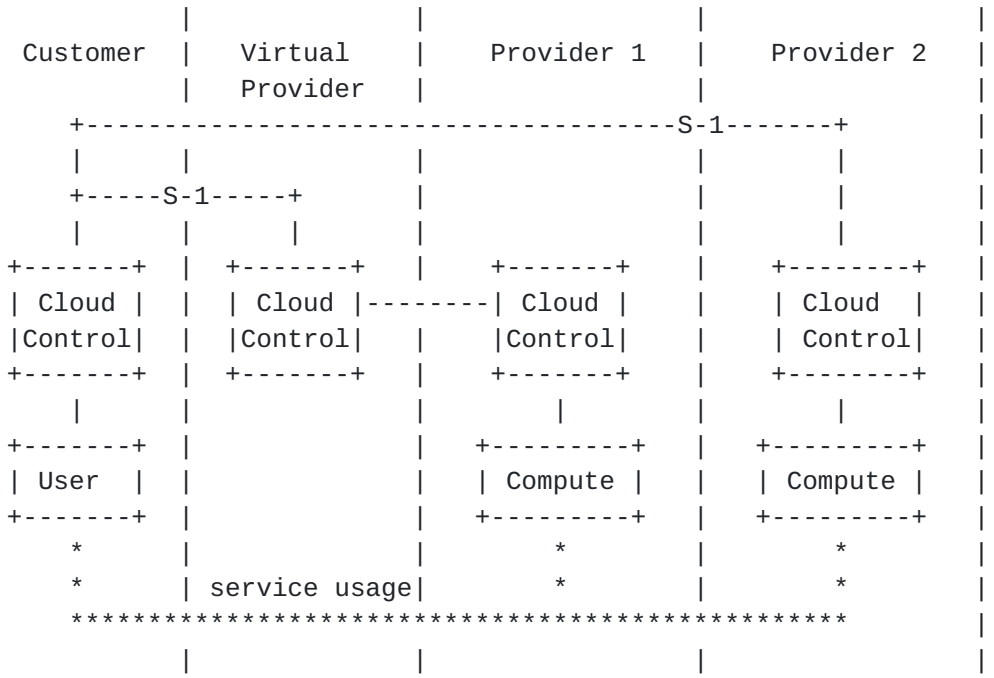

Fig-2: Scenario S-1 - User Interoperates Across Providers

Scenario S-2. Users Interoperate Between Private and Public Clouds. Users should be able to interoperate private clouds with those in the provider domain. This could involve moving workloads between private and public clouds. It also means creating virtual services in the same way in the public cloud as that in the private cloud. In Figure-3, a user creates compute in the private and public clouds in same way. Private storage is accessed from public and private compute.

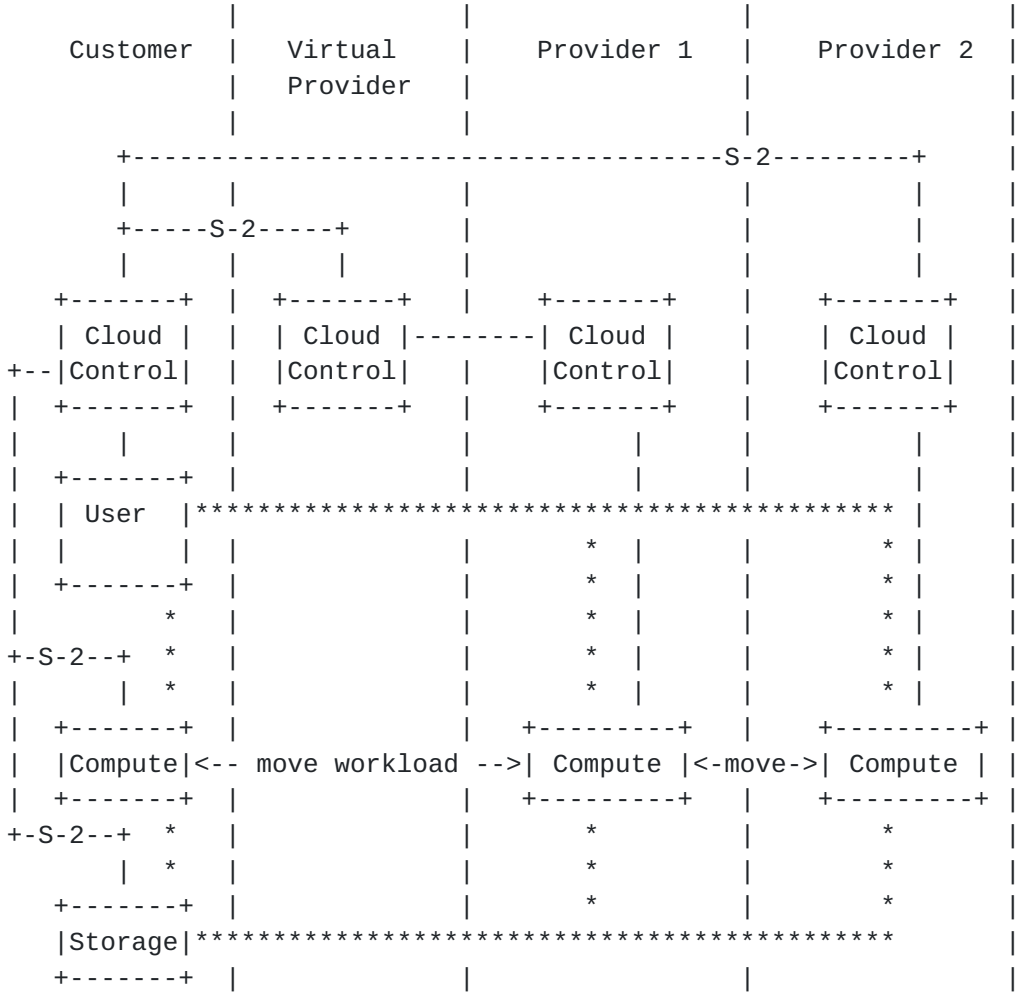

Fig-3: Scenario S-2 - User Interoperates Across Private and Public

Scenario S-3. Providers Interoperate With Other Providers. Providers should be able to interoperate their services with other providers. This could mean sourcing each other services when the demand suddenly grows, or using one vendor's services as backup or for disaster recovery under an outage. Providers might agree to host services in each other clouds in a follow the sun models, where workload moves between providers located in different geographies. There could be "provider of providers" - a virtual provider that sources services across different providers by using interoperability. In Figure-4, a provider sources the storage service to complement their compute service, and offers compute and storage as a bundle to the user.

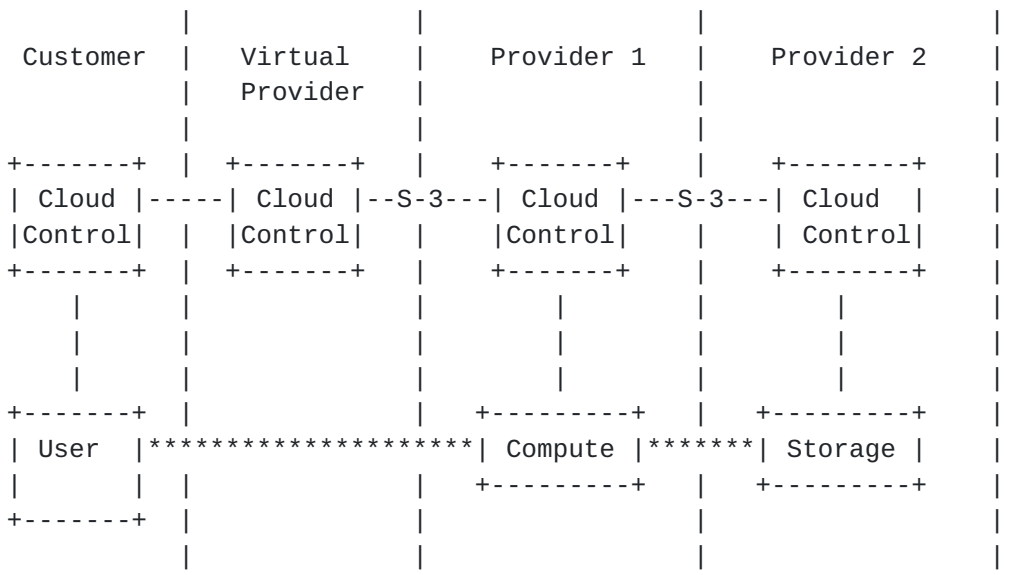

Fig-4: Scenario S-3 - Providers Interoperate Amongst Providers

Scenario S-4. Providers Interoperate Services Across Service Tiers. A cloud provider may deliver many kinds of services, layered on top of one another. For instance, SaaS may use PaaS, which in turn may use IaaS, network and security services, etc. Since cloud providers build services incrementally, it should be possible to interoperate services across these tiers, without having to build a new IaaS system for every new PaaS, or a new PaaS for every SaaS. In Figure-5, a provider sources IaaS services for their PaaS from another provider in the same way as they source them internally.

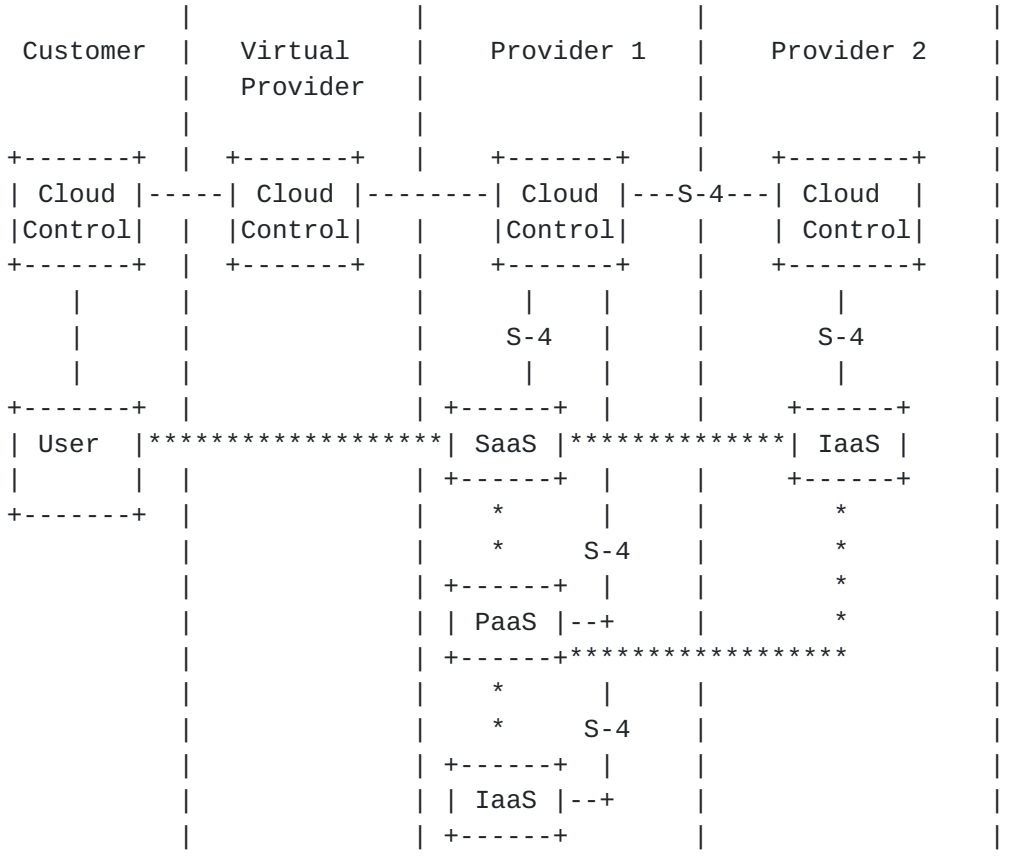

Fig-5: Scenario S-4 - Providers Interoperate Across Tiers

Scenario S-5. Providers Interoperate Across Service Vendors. A cloud provider may source a service from more than one vendor. Examples of these include compute virtualization, storage, network, security, etc. A customer's existing orchestration solution should be able to orchestrate multi-vendor products and services. In Figure-6, providers deliver a service using offerings from multiple vendors in the same way. These inter-vendor services may also be connected.

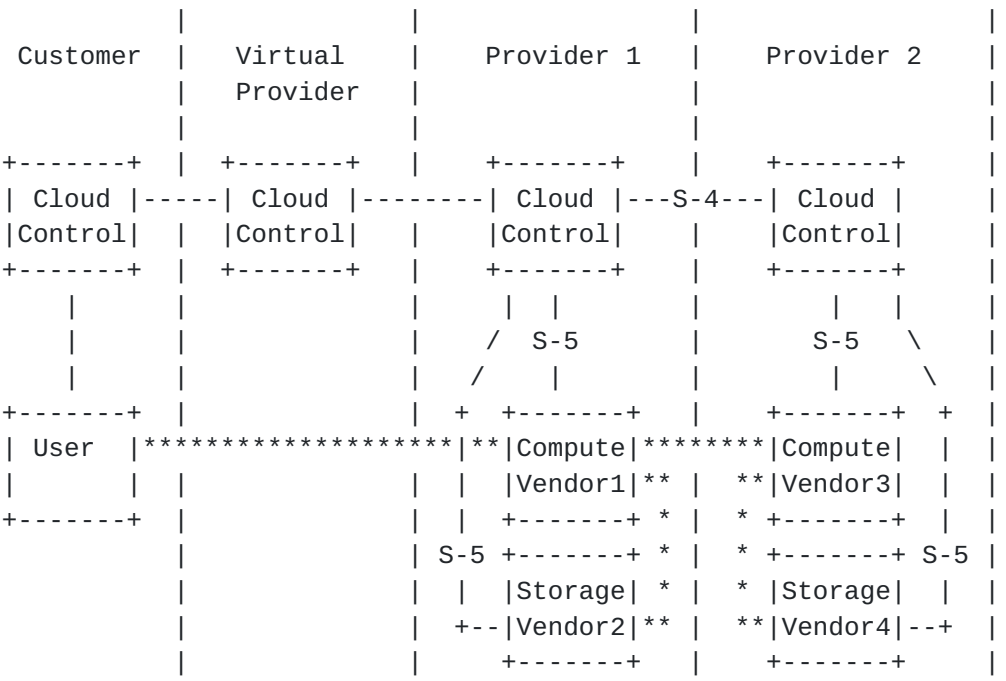

Fig-6: Scenario S-5 - Providers Interoperate Across Service Vendors

The above scenarios are illustrative and non-exhaustive. There could be many permutations of the above scenarios. Standardization will benefit users, vendors and providers - the total cloud ecosystem.

5. Cloud Open Source and Open Standards

Some efforts towards cloud openness today are focused on Open Source implementations of cloud services. This leads to the question of the relation between Open Source and Open Standards, as different ways to achieve interoperability. Obviously, cloud will not be totally open or closed source. The key problem in cloud is not the ability to inspect and modify code, which open source enables, but to integrate services, both open and closed. To integrate Open and Closed Source services, Open Standards are required, which may be implemented as Open Source. Open standards don't detract us from open source.

On the other hand, lack of open standards can make open source less attractive because there can be many open source implementations that are incompatible. Within an implementation, various versions may be incompatible. This means that Open Source alone cannot solve problems of interoperability unless everyone converges to a common code base and contributes their private changes back into the common base. This is impossible to mandate and unlikely to happen.

Open Standard implementations on the other hand will be interoperable even when implementations are enhanced in different ways. So Open Standards enhance rather than detract from benefits of Open Source. The key problem for cloud is service integration across vendors, providers and customers. These services will be Open Source, Closed Source, or multiple variations of Open Source. Integrating the variety of services is best done through Open Standards.

6. Is Cloud Control an Internet Problem?

Given that problems of cloud interoperability need to be addressed through standards, it may not be obvious that they need to be addressed by IETF. Why is cloud control an IETF problem?

First, to create, modify or move a distributed system, orchestrators need to know network topology. For instance, if firewall rules have to be installed for a VM, they must be installed on a device that lies in the "path" to the VM. To know which firewall lies on the path to a given VM, topology needs to be known. Similarly, if bandwidth needs to be provisioned between two sites, it is necessary to know which routers are at the edge of the two sites so that bandwidth can be provisioned between those routers. Likewise, if a VM is moved from one location to another, all associated network port configuration (such as VLAN or policy) needs to be dragged along with the VM. That requires the orchestrator to know which port the VM was attached on, and where it is going to move next. In some cases, the VLAN and policy may need to be provisioned not just in the access but also on the trunk ports to permit the packet flow. That requires the orchestrator to know which access is mapped to which trunks. To ensure that performance of a VM does not degrade after a move, it may be necessary to determine whether sufficient bandwidth is available at the destination location before the move is made. That requires knowledge of the paths that will be used and if those paths are congested. An orchestrator may need to assess the "distance" between the compute and network storage and between the user's location and the service's location for optimal performance.

There are also cases when knowledge of topology is needed for network optimization. For example, the network paths may not be optimal after

a VM move, and the paths may need to be re-provisioned. Such things are common with multicast and broadcast traffic that uses trees. During outages, network topologies are dynamically reconfigured. Recovery procedures must be aware of this network reconfiguration. The above examples illustrate a close relation between network information and orchestration of services. These two are currently treated as separate domains, and they need to be linked.

Second, cloud service discovery is about knowing the capability of devices in the Internet. Today, IP routing allows us to discover the location of IP addresses, but not their capabilities. For instance, the same IP address can belong to a PC, a router, a storage array, an IP-TV, a mobile phone, etc. Network protocols don't tell us the "semantics" of the IP, namely what that IP can "do". This of course is not a new problem, but cloud makes this problem very important. Cloud is about the ability to know which capabilities are available where in the network. This would be achieved if some protocol advertizes capabilities of IP addresses. Ideally, the systems that advertize addresses and those that advertise capability should be linked because the capability is of the address. To reach that capability, we need to translate it into an address.

When a service is yet to be created, it needs to be referred by its capability because the DN or IP for that service is yet to be created. This capability can be advertized by some service orchestrator that can create the service based on a request. In the Internet, a service naming mechanism is needed to advertize and request services by their "type" instead of DN or IP (DN and IP are useful for advertizing and requesting services that exist). These names can have a similar structure like DNS or IP addresses (dotted-decimal) but need to belong to a separate address space. We can call these "type" names Service Domain Names or SDNs.

Third, a cloud user may not care about the IP or DN of a service. What users care about is the "type" of service they are looking for. This service may be fulfilled anywhere in the network. The user will issue a request referencing the SDN, and would expect the request to be automatically routed to its correct destination. This is possible if SDNs have been advertised in the network. A user can forward a request to service aware router, and the router will map the request to destination. Mappings between service types and addresses can be done at the edges of the Internet allowing users to be unaware of IP addresses while the Internet to be unaware of services. A variety of policy controls can be built at the network edges to determine how a service "prefix" is mapped into an IP "prefix".

The problem of routing based on "types" is similar to routing based on IP addresses. In both cases, addresses need to be discovered, aggregated by some meaningful prefix, and advertized to routers upstream. These similarities imply that service routing can be implemented in ways similar to Internet routing in the past.

Fourth, thus far the link between capability and address has been done for services that are already created, generally within an administrative domain. For instance, it is possible to use DNS to discover the address of a printer or email server. Cloud deals with creation of services on-demand. This discovery over the Internet needs a somewhat different ability, such as policy control, routing, billing of services, authentication, security from denial of service, SLA announcements, etc. There is a greater amount of complexity in advertizing service information, publishing service interest, policies to control per-user services, etc. However, these issues are similar to things that have been done in IETF earlier.

In summary, orchestration needs to know network topology. The network can learn and advertize service capabilities like IP addresses. A mapping between addresses and capability is needed to perform service request routing. Such mappings have been created in the past, but just not to the extent required for cloud. The problem is both relevant for IETF and optimally solved within IETF.

7. Overview of Standard Work

To run the service exchange network over the current Internet, three important enhancements to the current schemes are proposed.

First, we need a service naming convention that addresses services by their "types" rather than by their DN or IP addresses. This naming system should also be hierarchical, in order to aggregate service types into "classes" of services. For instance, virtual machines may be referred by the name iaas.compute.virtual and firewalls by the name iaas.network.services.firewall. Each class of service may be associated with one or more attributes, or may be further divided into sub-classes, or sub-sub-classes, with suitable names. We can refer to these names as Service Domain Names or SDNs.

Second, we need a protocol that advertizes SDNs and routes service requests based upon these SDNs. This protocol will facilitate service aggregation based on names, service discovery, advertisement, selective publishing and indication of service interest, besides mechanisms to route the request based on where it can be fulfilled. We can refer to this as a Service Routing System (SRS).

The SRS needs to map service "prefixes" into IP "prefixes" and will interact with a policy based control system, where users, customers or providers can define rules for routing requests to a destination. The SRS discovers services and their locations and provides the mapping between Service Names and IP or DN. Using this mapping it is possible to identify the service by its name as well as type. The SN, DN and IP names are orthogonal name spaces. That is, any SN may map to any DN, which may in turn map to any IP.

Third, there is need for a common format to specify service attributes. This common format can be XML and it is necessary to define cross-service-domain orchestration rules. For example, in a L3 network, the IP of a host must belong to the subnet configured on the switch. The IP access-list on the switch must permit the IP address on the host. The ports open on a host must also be open on the firewall. The file systems accessible to a VLAN must align with the VLAN configured on the host access interface. The user-ids provisioned on the server must be available to authentication on the network storage. The speed of the virtual host interface must be equal to the bandwidth allowed to the host on the virtual or physical network interface. The virtual MAC allocated to a VM must not clash with any other virtual or physical MACs allocated anywhere else on the VLAN. The authentication system must use a combination of the tenant-id on the network in addition to the user-id on the host.

These relations represent semantic "rules" of orchestration. Today, we can't express these rules because information schemas across domains are incompatible. In effect it requires us to map some parameter in some CLI to some OID in another MIB. Or, some attribute in some XML schema to some TLV in another Protocol. Or, the value of a resource in a GUI to a range specified via another API. If all services are described in a common format (such as XML) then orchestration rules can be easily specified. This will allow rapid customization of services by defining orchestration rules in a high-level language rather than programming in a low-level language.

8. Deficiencies of Current Models

Cloud deployments today use HTTP web-services (SOAP and REST) to distribute service information and manage services. Web-services were designed for distributed application objects, where one object executes requests on other objects. This leads to the question if treating cloud orchestration as a distributed application object is the right approach to thinking about cloud services. In this section we will describe limitations of the web-service model. The web-service model is constrained by the capabilities of HTTP in service discovery, publishing and transaction management.

8.1. Service Discovery

HTTP was designed to connect clients to servers, but not designed for clients and servers to discover each other. HTTP assumes that client-server discovery happens through other mechanisms. The Universal Description Discovery and Integration (UDDI) web-service standard for instance defines registries where providers could publish their services but this mechanism is manual and not widely used.

In the cloud network, operators require services to be automatically discovered and advertized to consumers. Dynamic service discovery is also needed because as services are allocated or de-allocated, capacity dynamically changes. Manually detecting these changes would be nearly impossible for any large deployment.

HTTP does not have procedures by which a network of clients and servers can DISCOVER others and ADVERTISE their presence. HTTP allows a client to connect to a server after it has been discovered.

8.2. Service Publishing

With millions of possible services, users may rarely be interested in all such services. They may instead define selected types of service "interest" and expect to be "notified" when new services of interest are available. HTTP does not support SUBSCRIBE and PUBLISH mechanisms by which a client can SUBSCRIBE to select interests and would be notified of new services through a PUBLISH.

To know of the existence of new services, a Client must query a registry periodically. This makes service publishing a synchronous phenomenon and can be very hard to scale if millions of users query available services at regular intervals. To scale service publishing, it is necessary to make publishing an asynchronous phenomenon. HTTP is not designed to deal with asynchronous publishing.

8.3. Persistent Identities

HTTP loses the identity of a client after a transaction (such as GET or POST) has been completed. This means that every new transaction has to be authenticated and may require a new key-exchange. When millions of service instances have to advertize their presence or publish capabilities periodically, it is imperative that the underlying control protocol can maintain identity information persistently across these multiple transactions.

For instance in Session Initiation Protocol [SIP] users REGISTER with a SIP Proxy, at which time they are authenticated. Subsequent session

initiations don't require authentication. The identity established at the time of registration can be used across all transactions. This mechanism can be very useful as a single sign-on capability because after registering once, every other service does not require the user to be authenticated. The user can interact with all services by using the identity established during the registration. HTTP does not enable this because authentication is done by the server.

8.4. Blocking Calls

In a web-service call, a client blocks waiting for a response from the server. There is no mechanism for the client to timeout on a request, or cancel the request midway. If the server fails to respond to the request, the client must separately terminate the connection. This is not ideal because the server may in fact be taking a longer period of time to fulfill the request. When requests are used to orchestrate complex services, a server needs to send provisional responses indicating that a "session is in progress".

When a service involves multiple independent but related components (such as network, storage and compute), failure in one component may render the entire service unusable. In such cases, it is necessary to cancel the request midway. HTTP blocks for the server to respond and cannot cancel on-going transactions. The only mechanism to terminate the transaction mid-way is to close the HTTP connection, which can then result in leaked resources or incomplete actions.

8.5. Transaction Support

Complex orchestration scenarios need to treat multiple operations as a single atomic "transaction". For instance, an orchestration request may allocate compute, storage, network and security resources in a single request. Unless all of these operations have succeeded, the resulting service is not useful and must be cancelled as a whole. If all operations have succeeded, then they must be committed as a whole. Complex orchestrations thus need transaction support.

There are two ways to build this transaction support. First, each service can have its own transactions and cancelations. Second, transactions can be available natively in the orchestration protocol. Obviously, the first approach is very complex, and the preferred route is to have transaction support in the protocol.

HTTP does not have the ability to create transactions. HTTP request-response is atomic and considered complete individually. One HTTP request-response is independent of prior or subsequent request-response even to the same server, let alone another server.

Orchestration requires the ability to correlate request-responses across multiple servers and commit or cancel them as a whole.

If an orchestrator that uses HTTP web-services fails after making a request, the client will believe that the transaction has failed, while the service nodes continue to allocate resources towards completion. The client cannot be billed for the service, although the services would be created. To address reliability issues, each service must build application level transactions, and these will rapidly grow as services are modified. A native mechanism at the protocol level is required to address this.

8.6. Interactive Behaviors

Incompatibilities between a cloud request and cloud policies or partial failures in service orchestration may require an orchestrator to prompt a user with questions and/or confirmation before proceeding. For example, if a VM has been allocated but the requested amount of network storage is not available, the orchestrator may need to prompt the user to allocate a reduced amount of storage. Such interactive behaviors need to pause a transaction waiting for a confirmation from a user. HTTP does not allow a server to make another client connection to ask this question during an on-going transaction. Also, if the question is passed as a provisional response to the user, a user's response would be treated as a new request. HTTP has no schemes to tie a request to another request in the past, as all requests are independent.

9. Extensibility Considerations

One of the key issues in standardizing service orchestration is how this standard can be extended for service variety. To make the orchestration standard extensible to many services, we need to separate things that are service independent from those that are service dependant. Through this separation, it would be possible to extend a service protocol to transmit information about a variety of diverse services. This separation is described below.

9.1. Service-Independent Components

- Orchestration Verbs. Regardless of the kind of service that is being offered, there is need for service Discovery, Creation, Modification, Deletion, Migration, etc. There is also need for Confirming and Canceling requests midway through a transaction or indicating Successes and Failures upstream. Cloud involves many such useful "verbs" which are service independent. Whether we are creating a VM, VPN or Disk, the "CREATE" verb can be used to

indicate the operation of service creation. This common "CREATE" can be used for a variety of create tasks, and its meaning can depend on the receiver. Defining the verb once eliminates the need to redefine the same operation for each new service. A collection of such verbs can be standardized for any service to use.

- Transaction Nouns. To construct orchestration message transactions, there is need to address messages to destinations and identify their source, match requests with responses, bundle multiple such messages into a single complex exchange, sequence requests in the correct order with sequence numbers, have message fields to identify type of content and content lengths, common procedures for challenge and authentication of requestors, and many other such transaction level functions. Like orchestration verbs, these are service independent and can be standardized, without limiting service diversity and flexibility.

- Workflow and Task Language. Different users will request different combinations of services. One user might request a VM with only an IP address, but another user may also require storage allocation, bandwidth reservation, a secure firewall and a VPN to be setup automatically when a VM is allocated. To accommodate variety of service requests, a generic mechanism to define Workflows is required. A Workflow identifies a set of tasks to be performed for service orchestration. Users or providers may define Workflows at various levels of abstractions. Hence, it is important to distinguish Workflows from actual Tasks. A Workflow might equal to one Task, or a Workflow might comprise of several Tasks bundled as a single request. A service independent language to describe Tasks and Workflows is needed. A User should be able to refer to Workflows and Tasks using unique identifiers.

- Service Domain Names. To name services, a classification scheme is required. Classification allows us to combine attributes across similar types of services. We can take an object oriented approach for defining service domains. For example, "network" can be a root domain, "switching", "routing" and "network-services" can be child domains of the root "network" domain, "security" and "packet inspection" can be child domains of the "network-services" domain, etc. Child domains may inherit properties of the parent domain. A child domain may override the parent domain's attributes by redefining them in the child domain. Once a domain naming is well understood, service Proxies only need to advertize domains, with references to well-understood domain schemas. Users who request services will know what they are requesting based on domain name of the service. They will also know each domain's attributes. This abstracts a service implementation from the service user.

9.2. Service-Dependent Components

- Service Domain Parameters. Each service domain can have its own service specific parameters. They can reuse existing parameters by inheriting an existing domain. Domain parameters are inputs into a request, and effectively can be used like parameters being passed into APIs. Each domain may be associated with its own schema so that an orchestrator that does not understand a domain can still validate the request before forwarding it. The parameters of a domain can be defined in a sufficiently generalized way to apply to a wide variety of services in that domain.

- Vendor Specific Domains. Some service might not be standardized through well-defined domain definitions. These definitions cannot be understood by all clients or users. These may however be understood between select network end-points that choose to use such definitions. Using Vendor Specific Domains, experimental or customized domains may be defined.

10. Protocol Requirements

A protocol that supports service variety must separate service-independent and service-dependant parts of information. The service-dependant and service-independent information may be carried in the same message. This section describes needed capabilities for various service-independent and service-dependant functions.

P-1. N-way transactions - an orchestration controller will need to perform multi-domain (e.g. storage, compute, network, etc.) service operations. The protocol should be able to stitch these varieties of service domains into a single context. All transactions in the client-server model are 2-way, so this needs a new protocol.

P-2. It should be possible to sequence and parallelize messages within a single context. Sequences or parallelization would depend on the specific needs of a particular kind of service. For instance, compute and network services may be provisioned in parallel, while workload movement across geographical regions must take place sequentially. Accordingly, the responses to such requests may also be received in sequential or parallel fashion.

P-3. When using requests in a parallel or sequential fashion, it should be possible to "commit" these operations as a whole. If errors are encountered in any one of the transactions, it should be possible to "cancel" the entire service context as a whole.

P-4. For reliability, the protocol should support timers and timeouts on requests. These timers may be used to expect a response to a request within the specified timeframe. When the timer expires, recovery actions should be possible. This is also useful in case of network failures, and on-going transactions can be automatically reversed. Through use of timers, and automated reversal, failures would not result in leaked resources, incorrect accounting, etc.

P-5. The protocol should support explicit mechanisms to advertize services and discover other service agents in a network. That is, configuration of service agents should be minimized and the protocol should facilitate automated discovery and advertisement.

P-6. The protocol should support selective propagation of service information through use of publish-subscribe mechanisms. It should be possible for a client to request specific kinds of service information that it supports and expects to know about.

P-7. It should be possible to define workflows and tasks at various levels of abstraction. Some users will prefer abstract requests that are translated to concrete requests at some point before fulfillment. Others may prefer that they define every service parameter. The protocol must be able to support both these cases.

P-8. The protocol must support the CRUD (Create, Read, Update and Delete) operations to transact services, after discovery of agents and selective service exchange. These operations are part of HTTP and should be present in the new protocol as well.

P-9. It should be possible to refer to services using standard names. Use of standard names establishes convention on how services will be referred to, which in turn facilitates interoperable service publishing, advertizing, discovery and requests.

P-10. It should be possible to associate each service name with service-specific properties. These properties may be mandatory or optional. It should be possible to re-use these properties by inheriting a service name into another service name.

11. Separating Control and Policy Planes

Each service may be customized according to a variety of needs such as customer profile, user roles, location awareness, service design, SLAs, etc. The set of rules that are used to customize a service represent the "policy plane" as they specify how a service must be designed. This policy must obviously interact with the protocol messages ("control plane") to control service orchestration.

There are two broad approaches in which policy and control can interact. First, we might collapse the difference between control and policy, and just have a single plane that is designed for specific services. Second, we might separate control and policy planes, and allow independent evolution of policy and control planes. These options and their relative merits are discussed below.

In many orchestration schemes, the policy and control planes are collapsed into one. The orchestrator is designed and pre-programmed to automate a few types of services. This scheme works well if the desired service variety is small. Basically, for a small number of service types, a few service templates can be hardcoded and published to users. Users may choose from amongst available service templates to create services on-demand. A service template defines a set of business rules using which services would be created, deleted, modified or moved. If pre-defined rules meet the requirements of users, this is a huge simplification over manual service creation, and a good starting point for service automation.

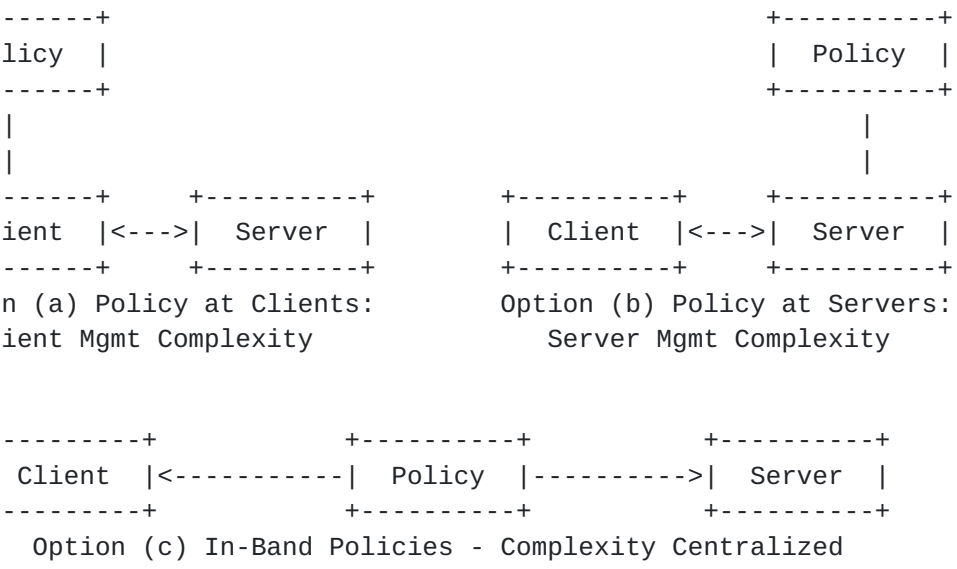

Figure-7 Policy Deployment Models

However, as the service variety grows, this approach cannot scale because the number of orchestrators will increase linearly with the number of service types, and the complexity in each orchestrator will increase exponentially with customization of business rules. Now, it is necessary to separate definition of business rules ("policy") from execution of rules ("control"). Interoperable control requires a protocol and interoperable policy requires an abstract high-level language to define orchestration rules. If the language of rules and

protocol have been separated and standardized, then the hurdles to deploying new services have been significantly reduced.

There are still multiple policy deployment options where policy is deployed at different points in the network, and these options can make important differences to the ease of service management. Different policy deployment options are shown in Figure-7.

First, policy may be attached to the user, such that users tune their personalized policies about services. Second, policy may be attached to each service, and the hardware-software vendor must give a configurable system for policy controlling each service, which the provider will have to customize to suit the needs of their deployment. Third, policy may be attached to the orchestrator, which may be defined either by provider or customer or jointly. The key difference between these options is who controls the service.

Client-based policies are totally in control of clients. Server-based policies are in provider control, but require the provider to individually manage policies on each service instance. When services are created dynamically, these service instances may have to download policies dynamically and refresh them when policies change. Dynamic changes to policies may disrupt existing services unless each server has the intelligence to process policy rules per request. If common policies have to be implemented across a set of clients, then these clients must be updated with the new policy rules. There must also be intelligence in client or server to deal with policy inconsistencies across client and servers. All this entails a significant amount of complexity in implementing and managing services.

Orchestrator based policies in contrast are easy to manage because they can be controlled at few network points. When policies change, the client and server don't have to be updated because policies are enforced run-time. Orchestrator policies can also be controlled either by provider or customer or jointly. It is architecturally important to place this control in the right point in the network to facilitate the best control scenarios. Obviously, orchestrator based policy control is more flexible and easier than others.

When policies are attached to orchestrators, clients and servers remain unaware of policy. Policy is now enforced at a small number of customer and provider edges. While the total number of policy rules remains unchanged, the complexity in managing these rules is reduced by centralizing the intelligence to define and apply policies. Challenges related to policy consistency are also addressed.

To apply these policies, client requests must be intercepted, policy transformed and policy routed before they reach the server. The clients and servers don't need to be aware of this behavior. The rules for controlling service requests can be defined through configuration in a policy server. Now, an orchestrator can download policy rules for a service, and execute those rules in real-time.

The separation of the control and policy planes allows the same control plane to be re-used for a variety of policies. Policies can be defined through configuration instead of being programmed in the orchestrator. And a common control plane can be used to orchestrate variety of services. Through this separation, a service orchestrator becomes a "Programmable Orchestrator", because it does not hardcode service logic. Rather, orchestrators can be "programmed" through policies defined by users in a user-friendly language. This approach eases service creation and customization of existing services while reducing overall management complexity.

12. Service Management Policies

This section describes different types of policies that might be used in cloud services. A few of these policies are currently being employed in the industry today, while many of them are desired features of cloud services in future. The totality of these policy types create a level of complexity that cannot be deployed by embedding policy in client or server. These policies should exist in a separate policy plane that interacts with the control plane.

12.1. Routing Policies

A service may be sourced from multiple destinations and to route a request to the correct destination, various types of routing policies may be applied. For example, a service request may be routed to the geographically nearest provider. Or, it might be routed to a location that offers the cheapest service rate or, to a different location based on time of day. There might be routing rules based on SLAs. Each user's request may be routed differently based on their roles. There could be rules specific to a type of service, or routing may be determined by the locations that have the necessary capacity. Routing may be determined by legal or governmental regulations.

These rules may be dynamically changed, and different rules may apply to different types of services, users, locations, roles etc. The provider and customer may independently or jointly define these policies, and enforce them at customer edge, provider edge, or both.

12.2. Security Policies

Security in the context of services encompasses a broad spectrum of issues spanning authentication, authorization and accounting (AAA). For instance, a customer may authenticate its users based on internal user-databases, while a provider owns the authorization and accounting of the service request. Or, a customer may own user-specific authorization and authentication while the provider owns the accounting. As users join or leave a customer, the provider may not own user-specific authentication and policies.

The AAA functions are best performed at the provider or customer edges. First, each service should not be required to do AAA; it is inefficient and complex. Second, service nodes must be protected from DoS attacks by preventing unauthorized requests from entering the network. Third, services may only be accounted as a bundle (e.g. network, compute and storage form a single usable service bundle) and not individually. Fourth, request logging for business analytics is best done at the network edges and not in individual services.

A provider may also wish to hide network topology of services, and may abstract locations from user-visibility. For instance, a provider may publish one interface to access all services although these services are orchestrated by service-specific orchestrators. And these orchestrators may be situated in different locations.

12.3. Service Policies

Complex services require coordination of multiple resources. A VM for instance may need network attached storage, network based security and network quality of service. The VM service may be regarded incomplete without the combination of all services. But, much of this is a matter of policy. Some VMs may require network attached storage, while others don't. Some VMs may need firewalls, while others may just need encryption of data. Some services may need a specific amount of network bandwidth to be available.

Policies associated with services can be abstracted from clients and servers. Accordingly, when a client requests for a VM, the request may be modified to include storage, security and quality of service requests before it reaches the server. Likewise, if a user is not authorized to request high-end services, their requests might be automatically downgraded to the appropriate grade of service. This is a function of policies that a provider and customer define.

This means that an AllocateVM request may do different things for different classes of users. Users may be upgraded or downgraded in

the level of services, while using the same AllocateVM request. This means that the syntax and semantics of a request is not fixed in advance. Rather, it is determined based on context, and different factors may be used to modify these requests in transit.

It is important to restrict the syntax and semantics of a request from an end-user perspective. It is also important to offload this restriction from the service itself. Thus, a server should be able to support a superset of request parameters, to allow any user to access the service in different ways. But each client may only request a well-defined subset of those parameters, based on prior customer or provider defined policies or SLAs. The validation and tweaking of request parameters in a user-specific manner should be controlled by policy in transit. In effect, the requests that a client makes and the requests that a server receives can be very different based upon the policies that modify the request in the middle.

13. Architecture Requirements

The general principle embedded in the following requirements is sub-system re-use by identifying common requirements and avoiding duplication for every new service (XaaS) needing to be deployed.

A-1. To ease the creation of varied services, there SHOULD be a separation between policy and protocol. Policy MUST deal with abstract rules about which components make up a service, and how those individual components must be created, deleted, modified or moved. Protocol MUST deal with the execution of these rules.

A-2. The interaction between policy and protocol SHOULD take place at the service orchestrators. Embedding this interaction in the client and server increases complexity and makes it harder to deploy new services or customize existing ones.

A-3. The Policy control MUST contain rules for service Authorizing and Accounting. That is, it must have rules about which users are allowed to access which services, or how services are customized for users and the user-specific charging rules to be applied.

A-4. Orchestration MUST be able to use the same Identity Management infrastructure for all services. Authentication should be performed by a coherent system across current and new applications. That is, each new service should not require new sets of mechanisms. Rather existing support systems should be extensible. Note, this may also span both provisioning and use of any particular service.

A-5. Orchestration MUST be able to utilize the same Accounting system across multiple services. New accounting systems should not be required for each service. Rather, the orchestrator MUST be able to use the same accounting system to create charging records.

A-6. Orchestration MUST be able to integrate with existing Fault management systems. Orchestrators MAY offload and/or automate intelligence to recover from failures.

A-7. Orchestration MUST be able to integrate with existing Performance management systems. Orchestrators MAY offload and/or automate intelligence to recover from performance issues.

A-8. Orchestration MUST be able to use common Operational Support Systems (OSS) such as DNS, DHCP and BOOTP systems.

A-9. Orchestration MUST be able to integrate with existing customer support and billing systems and/or provisioning new customers (BSS). This is to enable a single customer interface for all services.

14. IANA Considerations

Not applicable.

15. Conclusions

Interoperable ways of creating, delivering and consuming services is essential for cloud. To create this interoperability, there is need for an open standard protocol for exchanging service information. This document captures the requirements for such a protocol.

We envision that such a protocol can be an essential ingredient of Cloud Controllers / Proxies to exchange services across multiple private, public, hosted, community and other clouds.

16. References

16.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

16.2. Informative References

[NIST] DRAFT Cloud Computing Synopsis and Recommendations http://csrc.nist.gov/publications/drafts/800-146/Draft-NIST-SP800-146.pdf

[SIP] Session Initiation Protocol
http://www.ietf.org/rfc/rfc3261.txt

17. Acknowledgments

This document was prepared using 2-Word-v2.0.template.dot.

Authors' Addresses

Ashish Dalela
Cisco Systems
Cessna Business Park
Bangalore
India 560037

Email: adalela@cisco.com

Mike Hammer
Reston
Virginia
USA 20190

Email: mphmmr@gmail.com

Monique Morrow
Cisco Systems [Switzerland] GmbH
Richistrasse 7
CH-8304
Walllisellen
Switzerland

Email: mmorrow@cisco.com

Peter Tomsu
Cisco Systems Austria GmbH
30 Floor, Millennium Tower
Handelskai 94-96
A-1200 Vienna
Austria

Email: ptomsu@cisco.com