Computer Networks 68 (2014) 166–186

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Application delivery in multi-cloud environments using software defined networking

Subharthi Paul [a,*,1], Raj Jain [a,1], Mohammed Samaka [b,2], Jianli Pan [a,1]

[a] Washington University in St. Louis, School of Engineering & Applied Science, Department of Computer Science & Engineering, Bryan Hall, CB 1045, 1 Brookings Drive, Saint Louis, MO 63130, USA
[b] University of Qatar, College of Engineering, Computer Science and Engineering Department, P.O. Box 2713, Doha, Qatar

ARTICLE INFO

Article history:
Received 5 June 2013
Received in revised form 4 November 2013
Accepted 4 December 2013
Available online 22 February 2014

Keywords:
Application delivery
Layer 7 traffic steering
Cloud computing
Software defined networks
Service centric networking
Middleboxes

ABSTRACT

Today, most large Application Service Providers (ASPs) such as Google, Microsoft, Yahoo, Amazon and Facebook operate multiple geographically distributed datacenters, serving a global user population that are often mobile. However, the *service-centric* deployment and delivery semantics of these modern Internet-scale applications do not fit naturally into the Internet's host-centric design. In this service-centric model, users connect to a *service*, and not a particular *host*. A *service* virtualizes the application endpoint, and could be replicated, partitioned, distributed and composed over many different hosts in many different locations. To address this gap between design and use, ASPs deploy a *service-centric* network infrastructure within their enterprise datacenter environments while maintaining a (virtual) host-centric service access interface with the rest-of-the-Internet. This is done using data-plane mechanisms including *data-plane proxying* (virtualizing the service endpoint) and *Layer 7 (L7) traffic steering* (dynamically mapping service requests to different application servers and orchestrating service composition and chaining). However, deploying and managing a wide-area distributed infrastructure providing these service-centric mechanisms to support multi-data center environments is prohibitively expensive and difficult even for the largest of ASPs. Therefore, although recent advances in cloud computing make distributed computing resources easily available to smaller ASPs on a very flexible and dynamic pay-as-you-go resource-leasing model, it is difficult for these ASPs to leverage the opportunities provided by such multi-cloud environments without general architectural support for a *service-centric Internet.* In this paper, we present a new service-centric networking architecture for the current Internet called OpenADN. OpenADN will allow ASPs to be able to fully leverage multi-cloud environments for deploying and delivering their applications over a shared, service-centric, wide-area network infrastructure provided by third-party providers including Internet Service Providers (ISPs), Cloud Service Providers (CSPs) and Content Delivery Networks (CDNs). The OpenADN design leverages the recently proposed framework of Software Defined Networking (SDN) to implement and manage the deployment of OpenADN-aware devices. This paper focuses mostly on the data-plane design of OpenADN.

* Corresponding author. Tel.: +1 (773) 679 7723.
E-mail address: spaul@wustl.edu (S. Paul).
[1] Tel: +(314) 935 6160, Fax: +(314) 935 7302.
[2] Tel: +974 4403 4240, Fax: +974 4403 4241.

http://dx.doi.org/10.1016/j.comnet.2013.12.005
1389-1286/

1. Introduction

In this paper, we address the problem of application delivery in a multi-cloud environment. All enterprises, including banks, retailers, social networking sites like Facebook, face this problem. The enterprises are what we call Application Service Providers (ASPs) that want to deploy, manage, and delivery their applications over third party computing infrastructure leased from multiple Cloud Service Providers (CSPs) located all over the world. The Internet Service Providers (ISPs) provide the network connecting the users, ASPs, and CSPs. We describe the problem in two steps. First, we explain how the problem is handled in a private data center and then how it becomes more complicated with the emergence of cloud computing.

1.1. Service partitioning in a datacenter

The Internet's host-to-host application delivery model does not naturally map to the *service-centric* deployment and delivery semantics of modern Internet-scale applications. In this service-centric model, users connect to a *service,* and not a particular *host*. A *service* virtualizes the application endpoint, and could be replicated, partitioned, distributed and composed over many different hosts. However, in-spite of this, the Internet has successfully sustained the proliferation of services, without any fundamental changes to its architecture. This has been achieved by introducing *service-centric* deployment and delivery techniques within the enterprise infrastructure boundaries while maintaining a standard *host-centric* application access interface with the rest of the Internet. Initially, load balancers were introduced to support ***service replication*** over multiple hosts. A load balancer is a data plane entity that intercepts service requests in the data plane and dynamically *maps* it to the least loaded server that can serve the request. Although interposed in the application's data plane, the load balancer actually makes its mapping decision only on the end-to-end *connection setup* control messages (e.g., TCP SYN, flow setup over UDP, etc.) that are exchanged between the two end-hosts (in the host-centric design) to setup their transport/application layer connection state. Therefore, since the underlying communication protocols in the data plane are still host-centric, the load balancer needs to masquerade itself in the middle, using network address translation (NAT). A number of other DNS (domain name system) related techniques can be used to allow ***service replication***. However, what is more difficult is ***service partitioning***. Service partitioning is required to optimize the performance of a service by partitioning a monolithic service into smaller service components. Services may be partitioned based on:

Application content: For example, hosting video, images, user profile data and accounting data for the same service on separate server groups.
Application context: For example, different service API (Application Programming Interface) calls requiring different processing time and resources sent to different server groups.
User context: e.g., service access from different user devices (smart phones, laptops, and desktops), user access network capability (wired vs. wireless) being served differently.

Service partitioning improves performance, resilience, and also enables more efficient use of computing resources by allowing each service partition to scale independently through replication. For example, certain partitions may need to be replicated more than others owing to their higher resource requirements, criticality, popularity, etc. Data-plane proxying and L7 (Layer 7) traffic steering techniques are used to support service partitioning within enterprise environments. Data-plane proxying, unlike a network-layer load balancer, terminates L4 (TCP) connections and acts as a virtual application endpoint, thus allowing the service to maintain a standard host-centric interface for service access. The data-plane proxy then applies L7 traffic steering policies on each application message and independently routes them to a different application servers hosting the different service partition.

Also, in modern enterprise environments, different L3–L7 middleboxes provide several application delivery functions including message/packet filtering and monitoring (firewalls, intrusion detection, access control), message/packet transformation (transcoders, compression, SSL offload, protocol gateways) and application delivery optimization (WAN optimizers, content caches). Some of these services are message-level (L4–L7) and hence need to be implemented as data-plane proxies, whereas others are packet-level and may be deployed as network services. Different messages for the **same** service may need to be steered differently through a different set of middleboxes depending on the message context. Therefore, approaches that pre-establish an end-middle-end path through explicit control plane signaling such as in [1] are not sufficient. Instead a more dynamic approach of L7 traffic steering is required that enables *context-aware* service chaining where each message may be directed through a different sequence of middlebox services.

Therefore, data-plane proxying and L7 traffic steering, which we generically refer to as **Application Policy Routing** (APR), are the key techniques underlying intelligent enterprise fabrics that allow enterprises to support service-centric application deployments over a host-centric Internet design. Although not directly related to L7 traffic steering, but the application-level context needed to make APR decisions also informs the QoS requirements of delivering the message over the network. If the underlying network provides differentiated data transport services, such services may be invoked more efficiently (at per-message granularity rather than to all messages of the flow) and accurately.

1.2. Service partitioning in multi-cloud environments

Modern Internet-scale services are increasingly under pressure to move out of their centralized single datacenter environments to distributed multi-datacenter environ-

ments to be able to serve their global user populations more *efficiently* and *reliably*. However, for services to migrate to multi-data center environments a globally distributed *middle-tier* data plane proxy infrastructure providing APR services needs to be deployed. Enterprises that already operate multiple datacenters have this problem. For example, Google operates multiple datacenters across different geographical locations. Also, it operates a private WAN infrastructure [2] to connect its datacenters and also to connect to the end users as shown in Fig. 1. The Google WAN peers with user ISPs. At these POPs, Google probably operates L7 proxies to intercept user requests and route them to the datacenter best suited to serve the request. Till now, multi-datacenter environments were only accessible to a few large Application Service Providers (ASPs) who could also (probably) afford to setup their own distributed data-plane proxy infrastructure.

Cloud computing brings new opportunities to deploying and delivering modern Internet-scale online services and distributed applications in general. Application Service Providers (ASPs) can easily lease computing resources from third-party cloud providers under a flexible pay-as-you-go leasing arrangement. This allows ASPs to dynamically add or remove computing resources to their applications, allowing them to handle variable load patterns more efficiently and cheaply. However, providing service partitioning over third-party cloud infrastructures connected via public Internet is very challenging.

With cloud computing, opportunities to distribute applications across multiple geographical locations exist for smaller ASPs. In-fact multi-cloud environments provide a more dynamic application deployment environment compared to a multi-datacenter environment; partly because of the server virtualization technology in clouds and partly because of the pay-per use dynamic service deployment model offered by cloud providers. However, these smaller ASPs cannot fully leverage the opportunities offered by multi-cloud environments without a distributed data plane proxy infrastructure similar to Google, which they cannot afford owing to the high cost of owning and maintaining such an infrastructure. Also, it defeats the purpose why they adopted cloud computing in the first place.

Our proposed solution – OpenADN (Open Application Delivery Networking) [3,4] – allows ASPs to share a distributed data plane proxy infrastructure providing shared (hence cheap) APR services available to these ASPs. OpenADN is designed to allow third-party infrastructure providers such as ISPs and CDNs (Content Delivery Networks) to offer such a service to smaller ASPs. Also, ASPs can lease distributed computing resources from multiple-cloud providers, allowing them to cater to distributed user populations and make their applications more resilient against infrastructure failures and network load conditions.

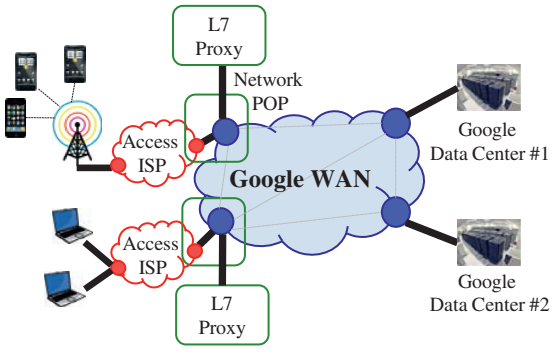


Fig. 1. Google WAN and Proxy infrastructure.

1.3. Contribution and plan

The key contribution of this paper is the design of a new session-layer abstraction for the Internet's networking stack called OpenADN. It extends previous *extensible* session-layer design ideas [5] to provide an integrated approach addressing all three aspects of application **deployment**, **delivery** and **access**.

Application deployment: OpenADN is designed to support next generation service-centric application deployments including service partitioning, service replication, service chaining and service composition; over third-party leased resources from cloud and multi-cloud environments allowing the deployment to dynamically instantiate/move/remove application instances over geographically distributed sites to optimize for cost, user experience, resilience, etc.
Application delivery: OpenADN provides a standard programmable interface though which ASPs may specify separate APR (Application-level Policy Routing) policies for each application-level flow. This allows the ASP to optimize application delivery **for a given application deployment scenario** that include parameters such as short term load variations, intermittent failures, user access location and user context (e.g., wired vs. wireless, smart phone vs. laptop) by steering the application traffic to the right cloud, the right application servers, through the right set of performance and security middleboxes, and accessing the right QoS services from the underlying network link.
Application access: OpenADN provides a new service-centric application access primitive which is aware of service-centric deployment requirements including service partitioning, service replication, service chaining and service composition. Also, OpenADN is designed to handle **dynamicity** in service access as a result of user and server mobility and interim failures requiring redirection of the application traffic.

Some initial ideas on the OpenADN architecture were presented in [3,4]. In [3] we tried to justify the need for an OpenADN-like architecture through a specific use-case example of mobile application delivery over distributed cloud environments while in [4] we tried to make a more general case for the need of a generic and open application delivery networking platform to leverage the opportunities provided by distributed cloud infrastructures. In this paper we present the design of the OpenADN architecture in much more detail and also present an initial prototype implementation as a proof-of-concept of the design.

In the rest of this paper, we will see how the OpenADN design achieves these goals. We present a high-level architecture of OpenADN in Section 2, followed by a bit of background into the service-centric deployment techniques used currently within enterprise environments and their limitations in Section 3. In Section 4, we discuss how the OpenADN architecture addresses these limitations followed by the detailed design of OpenADN in Section 5. In Section 6, we present the implementation approach and an initial proof-of-concept implementation of an OpenADN proxy, followed by related work in Section 7, a short discussion on the deploy-ability of OpenADN in Section 8, future work in Section 9, and discussion and conclusions in Section 10.

2. OpenADN: High level architecture

In this section, we present the high-level design of OpenADN, which provides a new session-layer abstraction for the network stack. There are two types of OpenADN entities that constitute an OpenADN-enabled application deployment and delivery scenario.

Application-data Visible Entities (ADV): ADVs are OpenADN-aware entities that have visibility (and access) to application-level data. They include ASP-trusted entities including end-users and ASP-owned middleboxes, application servers, storage nodes, etc. capable of doing processing, filtering, and transformation on application-level data in addition to the functions in the OpenADN-layer.

Application-data Blind Entities (ADB): ADBs are OpenADN-aware entities that do not have visibility into application-level data. They include third-party OpenADN proxies that are restricted to only perform functions in the OpenADN layer and such middlebox processing functions that do not require visibility into the application-level data including several data compression and other WAN optimization functions.

The distinction between ADBs and ADVs is important since the ASP may need infrastructure support from ISPs without compromising the privacy of application-level data. For example, a bank may not want ISP's to look at the data and determine whether it is a read or write and so to which cloud the message should be steered to. In general, ASPs would like to keep control over ADVs while delegating ADBs to ISPs.

Fig. 2 presents a high-level view of service access over OpenADN. A *service* in OpenADN is a virtual entity representing a set of access policies, including both, statically specified policies (for example mapping a user request to the right service partition) and dynamically determined policies (for example mapping a user request to the least loaded application server or middlebox). To access the service, OpenADN exposes a new *OpenADN session* primitive to applications. An OpenADN session is an application-neutral, ***logical*** communication channel between a *user* and a *service*. The user accesses the service (Service X) over this logical channel through *OpenADN messages.*

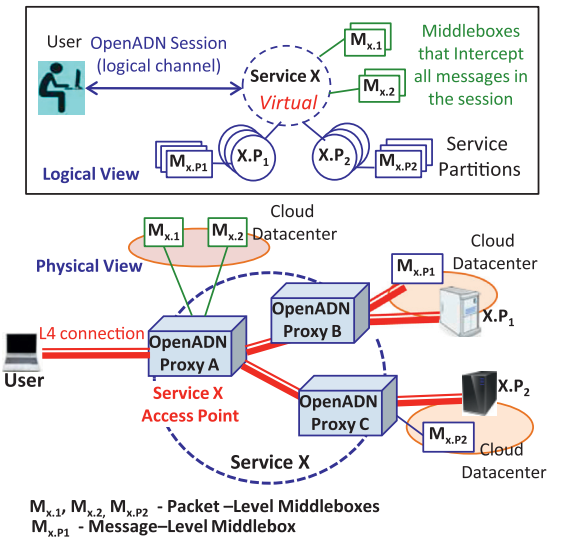


Fig. 2. Service access over OpenADN.

In the physical view (bottom part of Fig. 2), this channel maps to a set of OpenADN L7 traffic steering proxies (Proxy A, B, and C) that enforce the service access policies. These OpenADN proxies may be **ADBs** operated by a third party. All other entities shown in Fig. 2, including the user, middleboxes and application servers, are **ADVs**. Among these, Proxy A is user-facing and serves as the channel ingress/egress location for the user. To access Service X, the user needs to setup a L4 connection with this proxy. The user forwards all OpenADN messages (application message with an application-neutral OpenADN header) over this connection to Proxy A.

Proxy A applies L7 traffic steering policies (also referred to as Application Policy Routing or APR policies) to route the message. In the example shown in Fig. 2, Service X is partitioned into 2 partitions $X.P_1$ and $X.P_2$. Each partition has a middlebox service associated with it. Also, middlebox services, $M_{x.1}$ and $M_{x.2}$, may be security middleboxes such as stateful firewalls or intrusion detection systems (IDS) that need to reconstruct the application session to detect security threats and hence need to intercept all traffic belonging to an OpenADN session for Service X. Therefore, on receiving an OpenADN message from the user, Proxy A will first send it through $M_{x.1}$ followed by $M_{x.2}$, and then decide which service partition the message needs to access and accordingly forward it to either Proxy B or Proxy C. To do so, the user's L4 connection is spliced dynamically (at per-message granularity) to a pre-established L4 connection (multiplexed) with the next-hop OpenADN proxy. Also, each middlebox and application servers may be replicated and instantiated at different locations (on different cloud datacenters).

OpenADN uses several SDN (Software Defined Networking) concepts. SDN is, therefore, explained briefly next.

2.1. Using SDN framework for application delivery

SDN is an approach towards taming the configuration and management complexities of large-scale network

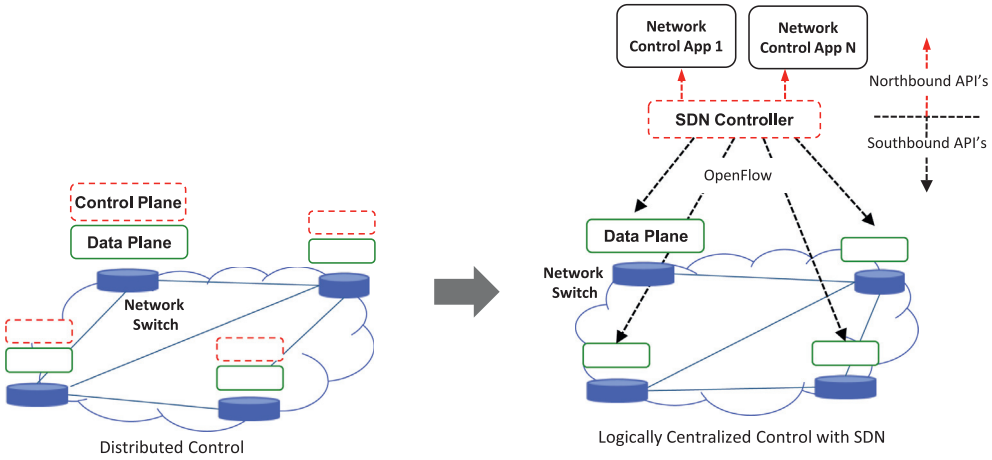


Fig. 3. Logically centralized control with SDN; Northbound and Southbound interfaces.

infrastructures through the design of proper abstractions. SDN proposes a separation between the network control and data planes (Fig. 3). This would allow the control plane to be logically centralized, making it easier to manage and configure the distributed data plane components (switches and routers) involved in the actual packet forwarding. To implement this separation, SDN needs to design a basic abstraction layer interposed between the control plane and the data plane.

Therefore, the SDN design needs to implement at least two sets of APIs – *Northbound APIs* and *Southbound APIs*. The Northbound APIs would allow different control plane applications to be written over the SDN controller, in a manner similar to different user-space applications written over traditional operating systems. The SDN controller abstracts the details of the underlying distributed resources (network switches and routers) that implement the network and provide a clean standard interface (and a virtual view) to the control applications. The Southbound APIs are needed to provide a generic interface between the controller and the actual data plane switches and routers. Network devices implementing the Southbound APIs can become part of the SDN controlled network infrastructure. Presently, the most popular Southbound API set is the one proposed by the OpenFlow standard [6].

As shown in Fig. 4, the OpenADN design extends the SDN framework to add support for application delivery. Currently, most of the momentum in SDN design is focused on developing proper northbound and southbound abstractions for managing packet-switching network devices. In OpenADN, we are developing new extensions to Southbound APIs to manage and configure L7 traffic steering (or APR) policies in OpenADN-aware data-plane devices including middleboxes, application servers and proxies. In contrast to packet-switching network devices, these new APIs have support for enforcing policies at the granularity of packets, messages and sessions. Also, new Northbound APIs need to be specified that will expose a uniform configuration interface through which the OpenADN data plane entities may be configured. A distinct feature (and also a challenge) of the OpenADN Northbound API design is that unlike the network controller, the design of the application delivery controller may span across more than one ownerships where the data plane infrastructure may belong to an ISP that may be shared and individually configured by multiple ASPs – each with different APR policies.

3. Current approaches to application delivery in data centers

Application delivery in general and L7 traffic steering in particular includes the following four functional components:

1. *Data plane proxying:* L7 traffic steering runs over a *data plane proxy* that poses as the *virtual* service endpoint to intercept all application traffic.
2. *Message classification:* The *proxy* needs to classify the application-level messages based on classification rules specified over application-level metadata.
3. *Application Policy Routing (APR):* Based on the result of the classification, APR policies are applied on the message to bind it to a workflow consisting of (*optionally*) a set of middlebox services (L3–L7) and a service endpoint(s).
4. *Message switching:* The message is switched through the components in the workflow.

There are three current approaches for these components as explained below:

1. *Consolidated hardware platforms:* These include monolithic, high-capacity, specialized hardware-based data plane proxies commonly called Application Delivery Controllers (ADCs) [7–9]. As shown in Fig. 5, ADCs provide a vertically integrated solution in which the same hardware device provides both, L7 traffic steering (*APR functions*) and *middlebox functions*. The *proxy* allows the ADC to serve as a virtual service endpoint, thus allowing it to intercept all application traffic from a given user. The *control* specifies the *rules* to extract

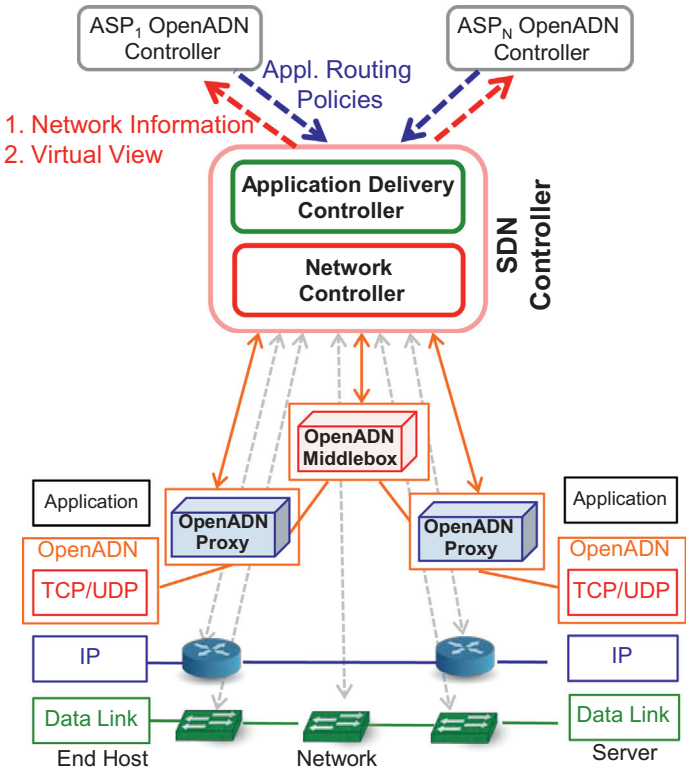


Fig. 4. Extending SDN to L3–L7 devices using OpenADN.

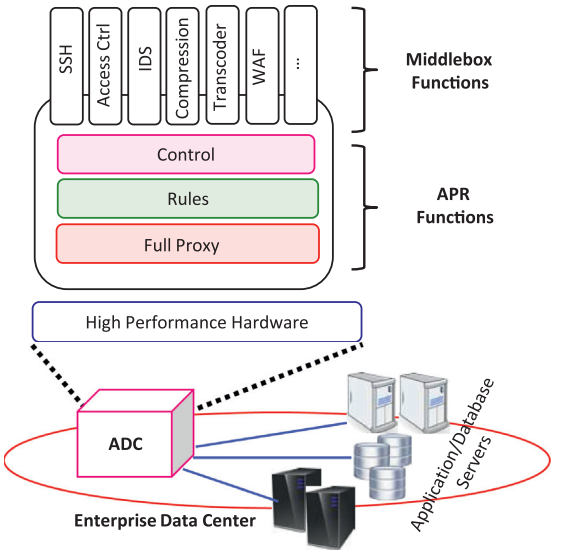


Fig. 5. Consolidated hardware platform.

application-layer context from the application-layer messages and then apply *APR policies* to steer the message through the right set of *middlebox functions* (deployed as modular plugins in the ADC device) and finally forward it to the right application server. Therefore, the ADC consolidates all the four L7 traffic steering functions on a single hardware device.

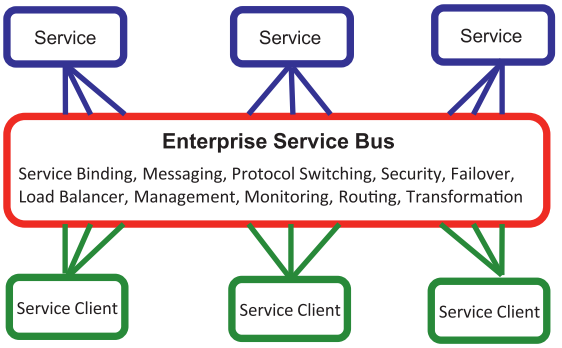


Fig. 6. Enterprise Service Bus (ESB).

2. *Middleware platforms:* Application platforms such as Oracle's Weblogic [10] or IBM's Websphere [11] include service integration middleware solutions, generically called the Enterprise Service Bus (ESB) [12]. An ESB provides an indirection middleware layer (Fig. 6) through which loosely coupled service components can be integrated. L7 traffic steering is the underlying mechanism to implement such integrations and is done using forwarding over SOAP (Simple Object Access Protocol) routing headers using HTTP (most commonly) as the underlying transport. SOAP headers are plain text (thus increasing the message size) and are placed deep inside the application layer. Also, it may be noted that, to

make dynamic APR decisions such as selecting a suitable (available and least loaded) instance of a replicated service or to determine the services that need to be chained to a particular message context, the ESB has to implement a content/context based router with load balancing capabilities. All messages in the SOA deployment will need to access this infrastructure component of the ESB. This component could be a bottleneck for the platform. An ESB provides an indirection middleware layer to which all services connect. The ESB appears like a single component to services, but unlike an ADC it is a distributed architecture consisting of two types of components: (1) content/context-based routers – consolidating *data-plane proxying, message classification* and *APR* functions, and (2) message transformation/filtering components – each configured to forward the message to the next-hop in the workflow, thus doing *distributed message switching*.

3. *Non-integrated approach:* In this approach, L7 traffic steering is deployed as just another middlebox (e.g., Content Based Router [13]) service in the group of L3–L7 services deployed in the enterprise environment, and provide support only for *service partitioning/replication*. Network services are chained through ad-hoc network configuration techniques [14] and message-level middleboxes are chained using static configurations. There is no support for *context-based service chaining* techniques and message-level middleboxes are chained using static configurations. There is no support for *context-based service chaining*.

3.1. Limitations of current approaches

There are several limitations of current approaches.

3.1.1. L7 traffic steering functions

None of the current approaches interface with the underlying network to access differentiated data transport services. Also, the ESB platform is primarily designed to serve as a message middleware layer and hence does not have support for integrating packet-level network services.

3.1.2. General deployment issues

The ADC-like integrated hardware devices are easy to manage and configure, however they are expensive and difficult to *scale-out*. ESB-like platforms are also easy to manage but some components such as the Content-Based Router may become bottlenecks since they need to process all application traffic. In the non-integrated approach, each component may be scaled independently by adding more instances, but such solutions are difficult to manage.

3.1.3. Cloud deployment issues

We consider two cloud deployment scenarios: single-cloud and multi-cloud.

3.1.3.1. Single-cloud-datacenter environments. There are two ways to provide ADC-like proxying solutions in cloud datacenters – *Over-the-Cloud (OTC)* and *Under-the-Cloud (UTC)*. In the OTC approach (Fig. 7a), the ADC is deployed as a virtual appliance over a generic virtual machine and administered completely by the ASP. The problem with this approach is that monolithic ADC proxy designs provide both, L7 traffic steering and middlebox services. Mapping this monolithic design over a virtual appliance will have serious performance issues. Without hardware-optimized designs (e.g., efficiently distributing different functions over different processor cores), each ADC virtual appliance will impose a high and non-deterministic (as a result of application-level processing diversity) overhead for all application traffic. In the UTC approach (Fig. 7b), the cloud provider provides ADC proxying as a service to the ASP. There are two problems with this approach. First, physical ADCs are expensive and their cost is amortized over handling large volumes of application traffic. So, a physical ADC will need to be virtualized and shared by multiple ASPs. Owing to application diversity (e.g., application traffic classification take different time based on the complexity of the rules), it is difficult to virtualize an ADC and share it across multiple ASPs. Also, each ASP will need to have different configurations for the middlebox services and hence these services cannot be shared effectively. The second problem is that the ADC will need to have full access to application-level data. ASPs may not be comfortable sharing their data with the CSP.

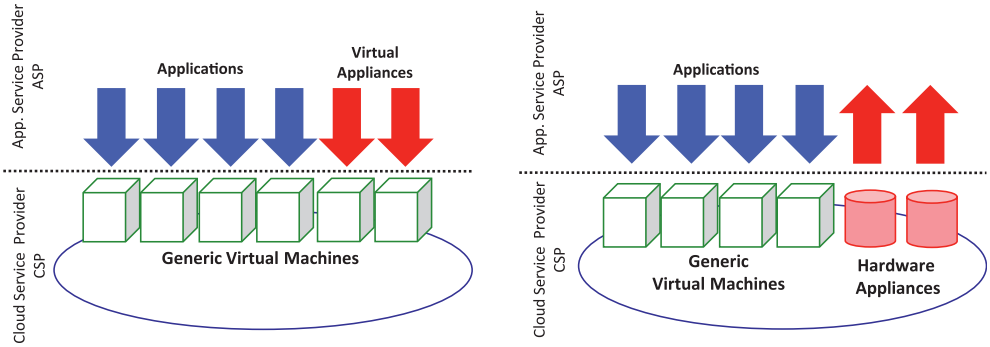


Fig. 7. OtC and UtC approaches.

3.1.3.2. Multi-cloud datacenter environments. For services to migrate to multi-datacenter environments a globally distributed L7 traffic steering service needs to be deployed. Smart, enterprise – owned private WANs such as the Google WAN discussed in Section 1 are prohibitively expensive to own and operate by smaller ASPs. Therefore, although these smaller ASPs have similar opportunities to distribute their application servers and middleboxes across multiple cloud datacenters, they cannot afford a globally distributed L7 traffic steering infrastructure. One approach is to provide L7 traffic steering as a shared (hence cheap) service by a WAN provider (such as an ISP). The problem with providing such a service are the same as those discussed for the UTC approach in a single cloud datacenter, with the additional requirement that the service should be geographically distributed.

3.1.4. Architectural issues

L7 traffic steering is needed by both, the ASP domain; to manage and deliver their service more efficiently, and the user domain; to access value-added services and enforce user policies. This calls for a generic *platform independent and application neutral* L7 traffic steering solution that is capable of enforcing both *sender and receiver policies.*

Our goal in designing OpenADN is to overcome the above 4 issues. These issues are summarized in Table 1.

4. OpenADN architecture

OpenADN provides an application-neutral, standardized, session-layer (message/session routing and forwarding) overlay over IP. It introduces a new protocol stack with two new layers (Fig. 8) – L4.5 (message routing) and L3.5 (packet switching, data-plane proxying, transport connection splicing/multiplexing, etc.). We now describe the functions of these layers in detail.

4.1. Layer 4.5 – OpenADN session abstraction

L4.5 (OSL or OpenADN Session Layer) provides a new session-layer abstraction to applications. This layer exposes two key primitives: *OpenADN session* and *OpenADN message*. The OpenADN session is a logical channel between the *user* and the *service* (unlike host-to-host sessions). Users can access the service over this logical channel through OpenADN messages.

The channel may steer each OpenADN message independently through a workflow consisting of a set of L3–L7 services and endpoint application server(s). Note that OpenADN does not simplify the *implementation* of L3–L7 service by providing any sort of common functional abstraction. It only provides a common platform abstraction for easily managing and controlling the deployment of these services. Some of the design requirements, issues and solution approaches that need to be addressed include the following:

4.1.1. Delegation

Just like the L3 interface (IP routing) in the current Internet, using L4.5 the ASP may now ***delegate*** application

Table 1
Comparison of different L7 traffic steering techniques in enterprise environments and OpenADN.

Layer 7 Traffic Steering	Layer 7 Traffic Steering Functions			General Deployment Issues		Cloud Deployment Issues			Architectural Issues	
	Service Partitioning & Replication	Dynamic & Context-aware Layer 4-7 Service Chaining	Access Data Transport Services	Scale-out	Easy Management	Over-the-Cloud (OTC)	Under-the-Cloud (UTC)	Multi-Cloud	Sender & Receiver Policies	Platform Independent
ADC-like solution	Y	Y	N	N	Y	Performance- N	Performance – Y, Sharing –N, Privacy- N	N	N	Y
ESB-like	Partitioning – Y Replication –May be	Message Level- Y Packet Level -N	N	Some Components	Y	Performance- N	Performance – Y, Sharing – N, Privacy - N	N	N	N
Non-Integrated	Y	N	N	Y	N	NA	Performance – Y, Sharing: May be, Privacy - N	N	N	Y
OpenADN	Y	Y	Y	Y Disaggregation	Y SDN Abstraction	Hybrid Approach (Delegation) Performance –Y, Sharing –Y, Privacy -Y		Y Delegation, Distribution	Y	Y

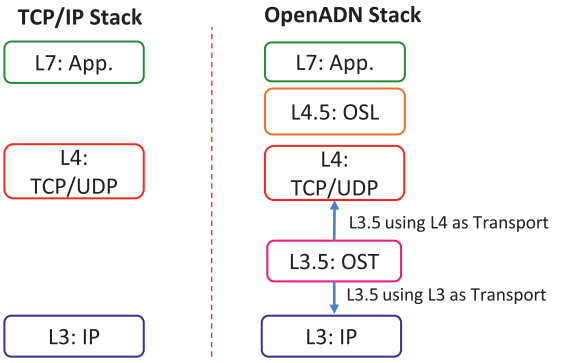


Fig. 8. TCP/IP protocol stack vs. OpenADN stack.

traffic (L7) steering to a third-party provider such as an ISP or a CSP. This will allow OpenADN to address the issues of cloud and multi-cloud deployments. A wide-area proxy infrastructure operated by an ISP will allow the ASP to distribute its servers and middleboxes across multiple cloud platforms and intelligently route application traffic to the right service partition/instance through the required set of middleboxes. Also, delegating to a CSP will allow an ASP to deploy its services in a cloud datacenter using a ***hybrid*** approach (in contrast to pure OTC or UTC approach), where the middleboxes and application servers are operated by the ASP while all application traffic is in-directed through an intelligent traffic steering proxy operated by the CSP in the aggregation tier. However, delegating APR to a third-party creates the following two design challenges:

1. *Privacy:* The message classification step in implementing APR requires access to application-level data. The ASP may not want a third-party (ISP), to have access to its application data owing to privacy concerns.
2. *Diversity:* For a third-party, it is difficult to share an APR proxy among multiple ASPs owing to the diversity of application-layer message classification rules. Each classifier will have different computational overhead and so it will be difficult to share common resources. A non-shared solution would be too expensive to be feasible.

The OpenADN design addresses these issues by moving message classification to the edge (Fig. 9); to ASP owned

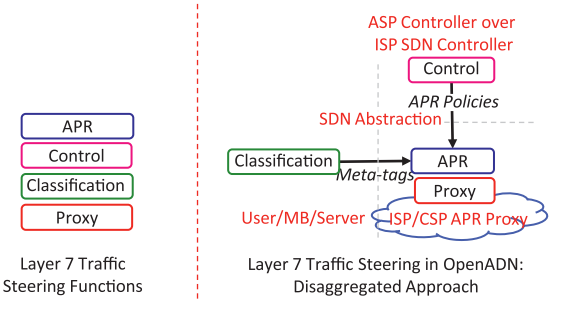


Fig. 9. Disaggregation of L7 traffic steering functional components in OpenADN.

L4.5 endpoints including user hosts, ASP middleboxes and application servers. The result of these classifications is recorded as *meta-tags*, which are fixed sized, flat identifiers, and carried in the OpenADN message header. Thus, each OpenADN message is routed at layer 4.5 by the *meta-tag*, and the third-party APR proxies map these tags to a *workflow* based on APR policies.

4.1.2. Deployment

Although moving message classification to the edge makes it easier to delegate the APR proxy to a third party, it still does not solve the following two deployment issues:

1. *Cross-domain configuration:* The APR proxy may be delegated to a third-party but the APR policies (*meta-tag → workflow mapping*) needs to be configured by the ASP based on its deployment policies and optimization criteria. Therefore, the ASP will need to configure the APR proxies owned and operated by the ISP. However, the ISP may not be comfortable to allow its infrastructure to be configured by a third-party ASP or reveal the internal deployment structure of its infrastructure.
2. *Management complexity:* Even if the ASP were allowed to configure the ISP's APR proxy infrastructure, it is difficult for them to select which APR proxies to configure from a group of proxies, manage a distributed configuration and make sure that their application traffic is directed to the right APR proxy.

OpenADN addresses these two deployment issues with the help of the **Software Defined Networks (SDN)** abstraction. The OpenADN APR function is disaggregated further, as shown in Fig. 9, allowing the control plane and data plane to be separated. The ASP is provided with a standard APR policy configuration interface through which the ASP control plane module can communicate its policies to the ISP's (using Northbound APIs) control plane that then programs its shared OpenADN data plane (using Southbound APIs) accordingly. Here, the role of the SDN abstraction is vital. SDN abstracts the distributed data plane deployment

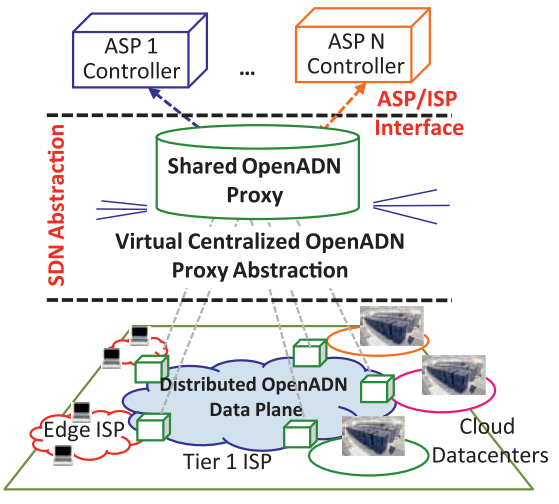


Fig. 10. Virtualized Proxy abstraction using SDN.

and presents a *virtual view* of a centralized, monolithic data plane proxy to the control application (Fig. 10). This serves two purposes. First, the ASP control plane module does not have direct access to the data plane but has access to a *virtual view*. Second, it is easier to write the control module (for a logically centralized proxy) and this helps in managing the complexity.

4.1.3. Deploying scale-out architecture

L4.5 maps a *meta-tag* to a *workflow*. To implement a scale-out architecture, each individual component in the workflow needs to be scaled independently to match the *throughput* requirements of the workflow. For example, suppose for

$$\text{Workflow } W = \{\text{firewall}, \text{IDS}, \text{app.server type A}\}$$

a total throughput 'xT' is required; and the throughput of each individual component (one indivisible unit which may be physical/virtual) are as follows: *firewall = T/2, IDS = T, application server of type A = T/4.* Therefore, W will require 'x' IDS', '4x' application servers and '2x' firewalls to satisfy the total throughput requirements. Therefore, in addition to just mapping the *meta-tag* to a *workflow*, the APR policies must also be able to distribute the load across each individual component in the workflow.

One possible approach is through a completely distributed control in which each component independently makes the next-hop decision based on, either some information about the global state (e.g., load, availability) of the next hop components, or based on just local information (e.g., simple hashing or weighted round-robin like load-balancing of incoming traffic). But, SDN was specifically designed to avoid this complexity of distributed control. Also, the selection of the next hop needs to optimize the total service time, that is, the time taken from the start of a workflow till the finish, which includes the service times of each component and the network delay. For example, in a workflow consisting of three components, A → B → C, when there are multiple instances of each, only those instances form a valid (feasible) workflow path that meet some criteria of the total service delay between 'A' and 'C.'

In OpenADN, we take a centralized control approach in which the centralized OpenADN controller pre-computes feasible workflow paths. Each path is *flow-balanced* (e.g., 1 application server, ½ firewall, and ¼ IDS) and also meets the total service time criterion. Also, the path computation is dynamic and paths may be updated based on dynamic network information (congestion, network failures) and also workflow component failures. Therefore, the workflow has many pre-computed paths with distributed ingress and egress, and each workflow component is pre-configured with the next-hop information. Note that this is very similar to a MPLS-like data plane with multiple pre-computed label switched paths between an ingress and egress as opposed to an IP-like data plane with per-hop destination-based forwarding. Another very strong motivation to make this design choice is that the components in the workflow may be L4–L7 functions, therefore requiring to communicate over a L4 connection. Dynamic next-hop selection would require dynamically setting up the L4 connection with the next-hop. Instead, in OpenADN, the pre-configuration step can setup the L4 connection with the next hop and multiplex traffic over it.

In OpenADN parlance, a workflow is called a *segment* (for reasons, discussed next) and each valid workflow path is called a *segment stream*. Therefore, in the data plane, for each OpenADN message, a L7 traffic steering proxy makes the following mapping: *meta-tag* → *workflow* (segment) → *workflow path* (segment stream); further explained next.

4.1.4. Multi-segment

An OpenADN session does not consist of a single workflow but many smaller workflow **segments** concatenated dynamically through APR policies. This is necessary to express the following APR policies:

Context-based L3–L7 service chaining: To understand this, consider a simple workflow shown in Fig. 11. The APR policy may dictate that **all** application traffic needs to pass through the Firewall, and then depending on the message context, is either forwarded to the Web-server through an Application Protocol Gateway or to a DB server through an Identity Verifier. To implement this context-based service chaining, the workflow is disaggregated into three separate segments as shown in Fig. 11; **All** application traffic is sent over segment 1; at the egress of segment 1, based on the message context *(meta-tag)*, APR policies select whether to send to segment 2 or segment 3.

Chaining stateful vs. stateless services: Another level of complexity is added to the above example due to the presence of *stateful* and *stateless* services. The firewall is a stateful device and therefore all OpenADN messages within an OpenADN session will need to access a particular *stream* of segment 1 for the lifetime of the session (e.g., stream 1.2 in Fig. 11), whereas segments 2 and 3 are stateless and so each OpenADN message may be sent through a separate segment stream for segments 2 and 3.

Sender and receiver policies: OpenADN needs to allow both sender and receiver policies, and hence an OpenADN session implicitly has at least two separate segments that need to be concatenated dynamically.

Each OpenADN message may carry multiple *meta-tags* that are mapped to specific *segment streams* by APR proxies. Initially, in the OpenADN session *creation* step, the user

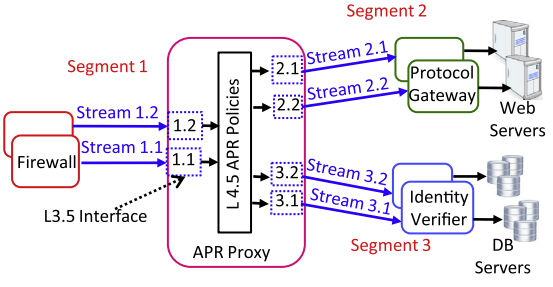


Fig. 11. Multi-segment, multi stream OpenADN session.

sends a Session SYN packet that is bound to two root segment streams based on the User and Service IDs (note: not *meta-tag*); (1) User ID → user-domain root segment stream, and (2) Service ID → ASP-domain root segment stream. The root segments consists of the L3–L7 services that need to intercept **all** application traffic in the OpenADN session (e.g., segment 1 in Fig. 11). For simplicity, let us just consider how an OpenADN message is forwarded in the ASP domain, After the session setup, all data-plane OpenADN messages are sent over the root segment stream, *by default*, and at the egress of the root segment stream, APR policies are applied to map the *meta-tag* to a *segment stream* of the next segment (either segment 2 or segment 3 in Fig. 11). If this new segment stream comprises of *stateful* services, then this binding is saved for the rest of the OpenADN session and stored at the user. Subsequent OpenADN messages with the same *meta-tag* carry this binding instead of the *meta-tag*. Therefore, a typical OpenADN message will carry a stack of labels comprising of a mix of *unbound* meta-tags and *bound* segment stream IDs.

4.2. Layer 3.5 – OpenADN segment transport

While L4.5 *routes* OpenADN *messages* over segment streams based on APR policies, L3.5 (OpenADN Segment Transport or OST) provides an overlay (over IP) packet-switching layer to switch *packets* through the components within a L4.5 segment stream. A L4.5 segment stream is uniquely identified (within an ASP or user domain namespace) by the 2-tuple <Segment ID, Stream ID>. Each component (middleboxes, application servers and user hosts) within this L4.5 segment stream needs to implement a L3.5 interface. The L3.5 interface is uniquely identified by the 3-tuple <Segment ID, Stream ID, Entity ID>. A component serving more than one L4.5 segment streams needs to implement multiple L3.5 interfaces, one for each L4.5 segment stream.

The OST (L3.5) interface design is unique in that it provides both; a L3 packet interface and a L4 transport endpoint interface. Therefore, it combines the concepts of packet-based MAC-over-IP distributed edge overlay tunnels (such as STT [15], VXLAN [16] and NVGRE [17]) and proxy-chaining tunnels such as those used to implement the WS-Routing [18] interface in SOAP-based SOA. This allows OST to support both, message-oriented services (e.g., transcoders) and packet-level network services (e.g., IDS) within the same L4.5 segment stream.

The OST packet header carries three different entity IDs for the same <Segment ID, Stream ID> – Two entity IDs representing L4 source and destination and the third representing the next hop OST interface. This is required because the transport-layer destination entity ID may not be same as the next hop OST interface, as is the case in a segment such as Firewall (Message-level) → IDS (Packet-level) → Transcoder (Message-level).

As shown in Fig. 11, a L4.5 APR routing function connects to multiple OST interfaces to be able to route application traffic across different segment streams. An OST interface may be configured (by an SDN controller) as, (1) simple packet based interface (e.g., IDS middlebox), (2) with two-TCP-like interfaces – one for incoming and the other for outgoing (e.g., message-level middleboxes), (3) multiple incoming TCP-like interfaces with one out going TCP-like interface (e.g., ingress to a segment stream), (4) multiple incoming and outgoing TCP-like interfaces (e.g., in a third party proxy where the next-hop is not configured in the individual components).

To summarize, the different design choices of the OpenADN session abstraction and the OpenADN segment transport address the issues with current L7 traffic steering solutions, summarized previously in Table 1, and provide a generic architectural solution to data-plane proxying and L7 traffic steering to support modern service-centric applications over the Internet.

5. OpenADN design approach

In this section we discuss the key aspects of the OpenADN design that address the architectural goals discussed in Section 4.

5.1. ID/Locator split

OpenADN is designed as an overlay over IP. Hence, each OpenADN entity (end host, proxy, middlebox, application server) is assigned a globally unique and *fixed identifier* which is separate from its *locator*. The ID is represented by the <Authoritative Domain ID, Entity ID> 2-tuple; where the authoritative domain is the SDN domain that controls the entity and assigns it an unique ID within its namespace. It could be an enterprise, organization or ASP. An entity under multiple authoritative domain control will have multiple IDs. All communication in the OpenADN layer (L3.5 and L4.5) are between these IDs. When the OpenADN layer hands over a packet to the IP layer, it needs to access an ID/Locator *mapping* function that will map the ID to an IP address (locator). This ID/Locator split is necessary to uniquely identify and address an application level entity. This is required for the following 3 reasons:

1. Enforcing sender/receiver policies (Section 4.1.4),
2. Specifying session affinity over a middlebox sequence and endpoints (Section 4.1.4), and
3. Correctly steering the application traffic across locator changes as a result of server/user mobility.

All IDs are 128-bit UUID (Universally Unique Identifiers). Locators can either be IPv4 (32 bits) or IPv6 (128) bits.

5.2. OpenADN data plane

The OpenADN data plane design needs to be standardized to allow high performance implementations (and commoditization). Also, standardization will allow the data plane to be outsourced to third party providers. Presently, the data plane implements MPLS-like label switching and stacking mechanisms, which we call APplication Label Switching (APLS).

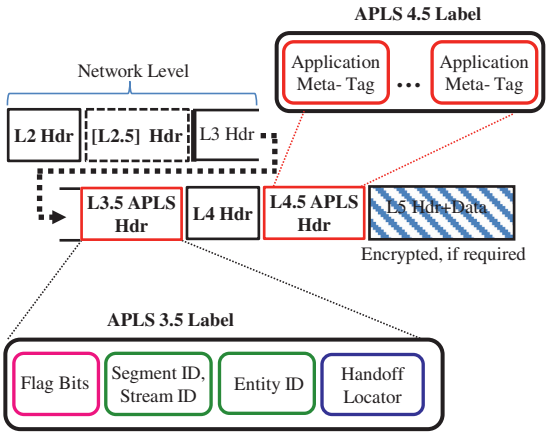


Fig. 12. APLS labels.

5.2.1. Label switching mechanism

Fig. 12 shows APLS labels for L3.5 and L4.5. The APLS labels, unlike MPLS labels, are composite labels consisting of several sub-labels. Each sub-label may be individually switched by different entities to support different requirements. The switching rule is specified by the ASP control plane module installed over SDN. The control module has full access to the ASPs deployment state (such as application partitions, replications, distributions, current load, current user access patterns and liveness and failure information) that are required to compute the switching rule from the ASPs application-level policies (such as optimize for cost, user perceived quality and availability). It sends the rule to the data plane entities through an *extended* OpenFlow protocol (discussed in Section 6). We now discuss each sub-label in the context of the design requirements.

5.2.2. Layer 4.5 APLS label

The L4.5 APLS label carries a stack of *application meta-tags* representing a sequence of L4.5 segments in the data path (Fig. 12). The *meta-tags* are inserted by the application end-points. As discussed in Section 4.1.1; implementation of application-level routing may require access to the application level data. However, owing to the constraint of application data privacy (Section 4.1.1) in *outsourcing* application-level routing to third-party providers, the implementation needs to be divided into two separate steps – the *classification step* and the *binding decision step.* In APLS, the two steps are connected through an *application meta-tag* (or simply referred to as the *meta-tag*) that encodes the result of the application-level classification. In the current prototype implementation of OpenADN, *meta tags* are encoded as flat 128-bit identifiers. Later versions will implement it as a variable sized <Tag, Length, Value> field to allow more flexibility. The *classification step* needs to be implemented by an ASP trusted entity that has full access to the application data and the communication context, such as the end-hosts. A middlebox (or a chain of middleboxes) belonging to the outsourced third-party enforces the *binding decision* based on the *meta-tag.* The *binding decision* is configured as a rule, which is installed by the ASPs control plane module. Apart from application data privacy, the *meta-tag* has performance benefits as well. It allows the high-speed implementation of application-level routing at the outsourced APR proxy by removing the requirement of general purpose processing needed to do application-level classification.

As shown in Fig. 13, at the egress of an application segment, an APR proxy advances the *active tag* pointer (equivalent to popping in the label stack) to the next *meta-tag* and maps it to a <Segment ID, Stream ID> *two-tuple* of the next L4.5 segment. The *two-tuple* is placed into the L3.5 APLS header, discussed next.

5.2.3. Layer 3.5 APLS label

A L3.5 APLS label represents a single L4.5 application segment. It serves as a hop-by-hop transport header through the sequence of *waypoints* implementing the L4.5 segment. It consists of the following three fields.

1. *<Segment ID, Stream ID> 2-tuple:* The *<Segment ID, Stream ID>* two-tuple represents a specific instance of an application segment as discussed in Section 4.1.4. For entities such as the IDS in Fig. 13, which hosts more than one segment stream (for flow-balancing as discussed in Section 4.1.3), each segment stream has a virtual L3.5 interface uniquely identified by the <Segment ID, Stream ID> 2-tuple.
2. *Entity ID:* This field holds the *entity ID* (Section 5.1) of the next hop entity within the L4.5 segment stream. For example in Fig. 13, *segment stream* (defined in Section 4.1.3) SS 2.1 is composed of Firewall 1(FW1), IDS and Application Server. Each of these entities has a unique entity ID within the ASP namespace (Section 5.1). If an intermediary entity, such as the firewall (FW1 in SS 2.1) knows the next hop to be the IDS, it will update the entity ID field with the entity ID of the IDS. Else, it returns the packet with a *flag bit* (in the *Flag Bits* field in Fig. 12) set indicating that the packet has already travelled the entity recorded in the *entity ID field* and needs to be switched to the ID of the next-hop entity in the segment stream. This will be explained in a bit more detail in §5.4 where we discuss an example of packet forwarding through L3.5 label switching within a segment.
3. *Handoff-locator:* The handoff locator is used to deploy OpenADN as an overlay over IP. Again, the use of the *handoff locator* will become more clear in the packet forwarding example presented in Section 5.4.

5.2.4. Label stacking mechanism

Label stacking is used for *enforcing sender/receiver policies* (Section 4.1.4). In the forward path (user → service), the user *pushes* two sets of L4.5 labels (*Note:* there may be more than one label in each set) into an OpenADN message – *sender labels for the user (U) domain stacked over the receiver labels for the service (S) domain.* For simplicity, Fig. 14 shows a scenario in which only one sender and one receiver label are stacked into the OpenADN message. The outer *sender label* invokes the sender policies in the sender's policy domain. Similarly, the inner *receiver label* invokes the receiver policies in the receiver's policy do-

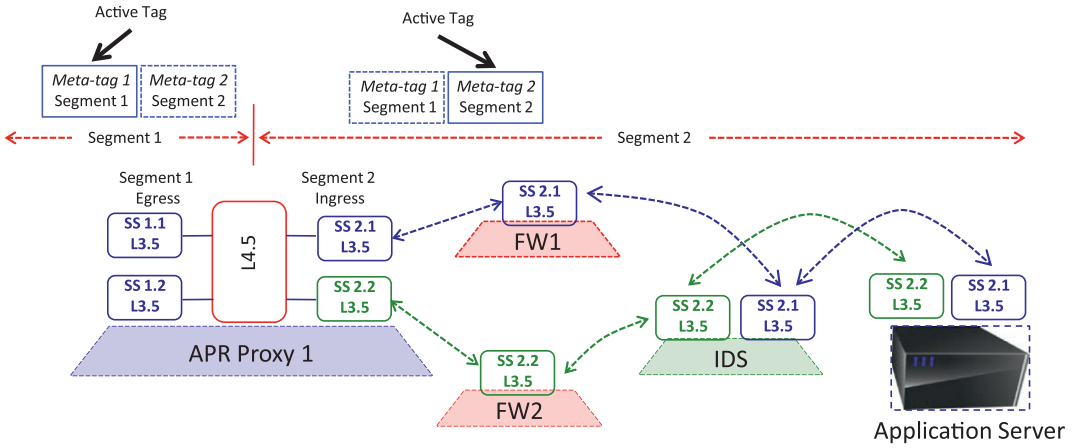


Fig. 13. Concept of segments and segment streams in the OpenADN data plane.

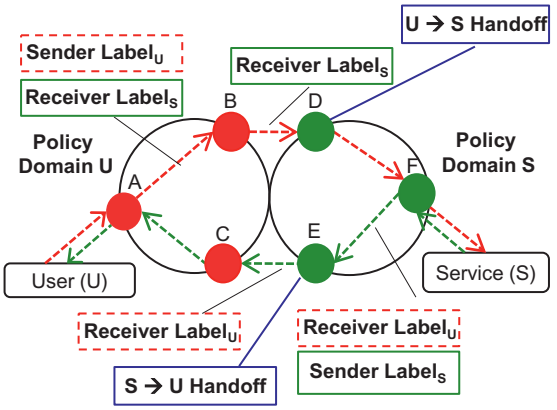


Fig. 14. APLS label switching.

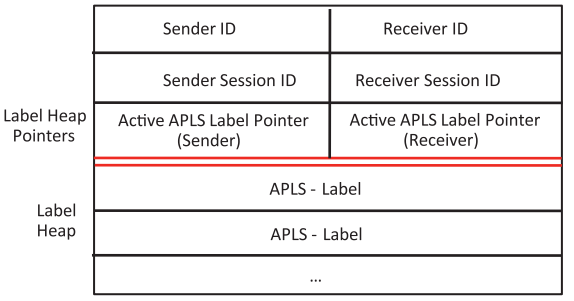


Fig. 15. L4.5 APLS header.

main. Initially the message is routed to the ingress of the user's policy domain (Node A in Fig. 14). The *sender label meta-tag* is mapped to a <Segment ID, Stream ID> 2-tuple and the message is routed through the entities in the segment stream of the user's policy domain. The egress node (Node B) of the user's policy domain segment stream, *pops* out the *sender label* and forwards the packet to the ingress of the service' policy domain (Node D) with only the *receiver label*. Within the service' policy domain, the *receiver label meta-tag* is mapped to a <Segment ID, Stream ID> 2-tuple and the message is routed through the entities in the segment stream. The *handoff locator* provides an explicit in-direction mechanism to determine the ingress of each policy domain. We expect that in most use cases, policy domains will use IP anycasting to advertize their ingress locators. In the return-path (service → user), only the roles of the sender and receiver are reversed and the same sequence of label pushing and popping operations are performed, as shown in Fig. 14.

Fig. 15 shows the L4.5 APLS header. It may be noted that the L4.5 APLS header informs message-level routing policies and hence it is a message header (OpenADN message frame) that is not carried in every IP packet. It carries a list of *application meta-tags*. Each application meta-tag is dynamically mapped to a L3.5 Segment Stream ID or SS-ID (<Segment ID, Session ID> 2-tuple) by an OpenADN proxy. To support *session affinity*, the list is a mix of unbound *meta-tags* and *already bound SS-IDs* (meta tags that have already been bound to SS-IDs for the duration of the OpenADN session). A one-bit flag indicates whether a label is a meta-tag or a bound SS ID. In the current implementation, both application *meta-tags* and *SS IDs* have the same size (128-bits) and hence the list is homogeneous. In the future, if the *meta-tags* are implemented with *<Tag, Length, Value> (TLV)* encoding, the list will need to be heterogeneous (with each label also encoded in the TLV format). The label heap stores the APLS-label list with the active APLS-label pointer (offset) in the header (Fig. 15) pointing to the currently active label in the label heap. The *sender* and *receiver ID* fields are needed to uniquely identify the policy context on shared infrastructures while the *sender* and *receiver session IDs* specify the particular communication context between the sender and receiver entities. When a message arrives at an OpenADN proxy, it reads the next active APLS Label. If the APLS label is a meta-tag, the OpenADN proxy maps it to a SS ID and then copies it to the L3.5 APLS header (packet-level). If it is already a SS ID it directly copies it to the L3.5 APLS header. It then updates the pointer to point to the next APLS label in the heap and forwards the message. Several other fields such as a

protocol field, header size, and flags are not discussed because they do not relate to any interesting OpenADN specific functions.

5.2.5. Label-switching example

Fig. 16 shows a L3.5 label-switching example within a L4.5 *segment stream*. The example illustrates how OpenADN is implemented as an overlay over IP and the how a distributed infrastructure of APR proxies, doing both L3.5 and L4.5 switching helps in steering the traffic through the right set of middleboxes and application servers. In this example we just show how an OpenADN message is forwarded through a single sender segment stream before it is handed-off to the receiver domain for receiver-side policy enforcement. This is a 6-step process as explained next.

In **Step 1**, the user (or sender) copies the *handoff locator* from the sender label to the IP destination field and forwards it to the *ingress* (of the sender policy domain) OpenADN proxy A over the IP network. The <*Segment ID, Stream ID*> is set to <X, 1>, assuming that this $Tag_U \rightarrow$ <X, 1> mapping is already resolved and stored at the user. The '0' represents an empty *entity ID* field. The '*W flag*' bit is set to 1 indicating that the packet has already traversed the entity indicated by the *entity ID* field and needs to be forwarded to the next entity in the <X, 1> segment stream.

At OpenADN proxy node A (**Step 2**), <X, 1> is looked up to determine the next-hop *entity ID;* which is M_1. Node A then looks up the locator for M_1 and copies it to the IP destination address field. In this case, this is not the actual locator of M_1 but the locator of the *egress* node of M_1 (node B). Whether M_1 maps to its actual locator or maps to fixed egress node of M_1 depends on how the sender domain implements the ID-to-locator mapping function. If a DHT based lookup mechanism is used, then the egress node may be the lookup node for M_1. If the mapping is actively distributed to all nodes, then M_1 may be mapped directly to its locator. The mapping mechanism will depend on the network size and the frequency of updates. It may also depend on the entity type. For example, for small number of middlebox nodes (mostly fixed), an active distribution mechanism is not too costly; for user IDs (large numbers, frequent mobility), DHT may be a better choice.

The *handoff locator* field stores the locator of the OpenADN proxy node to which the waypoint node (node M1) needs to send back the message after processing it. At this point, there are two options. The first option is to set it to IP_A. This is done if the sender domain implements a centralized orchestration model, in which a single OpenADN Proxy (such as Node A in this case) orchestrates the whole workflow. In a more plausible *distributed* solution, it will be set to an anycast IP of the policy domain. This allows the waypoint return message to be intercepted by the closest OpenADN proxy node. In this example, the *distributed solution* is assumed. The *W flag* bit is *set* and the packet is forwarded towards node B. Note that in both these approaches, we assume that the workflow orchestration is done by the OpenADN proxy infrastructure, and the waypoint nodes (Node M1 and M2) are unaware of the next-hop to which they should forward the message. This may not be the case in actual deployments. We consider this case to show the most general deployment scenario that the OpenADN design may need to handle.

In **Step 3**, Node B checks the *W flag bit;* looks-up the *ID to locator* mapping for M_1; and *re-writes* the IP destination

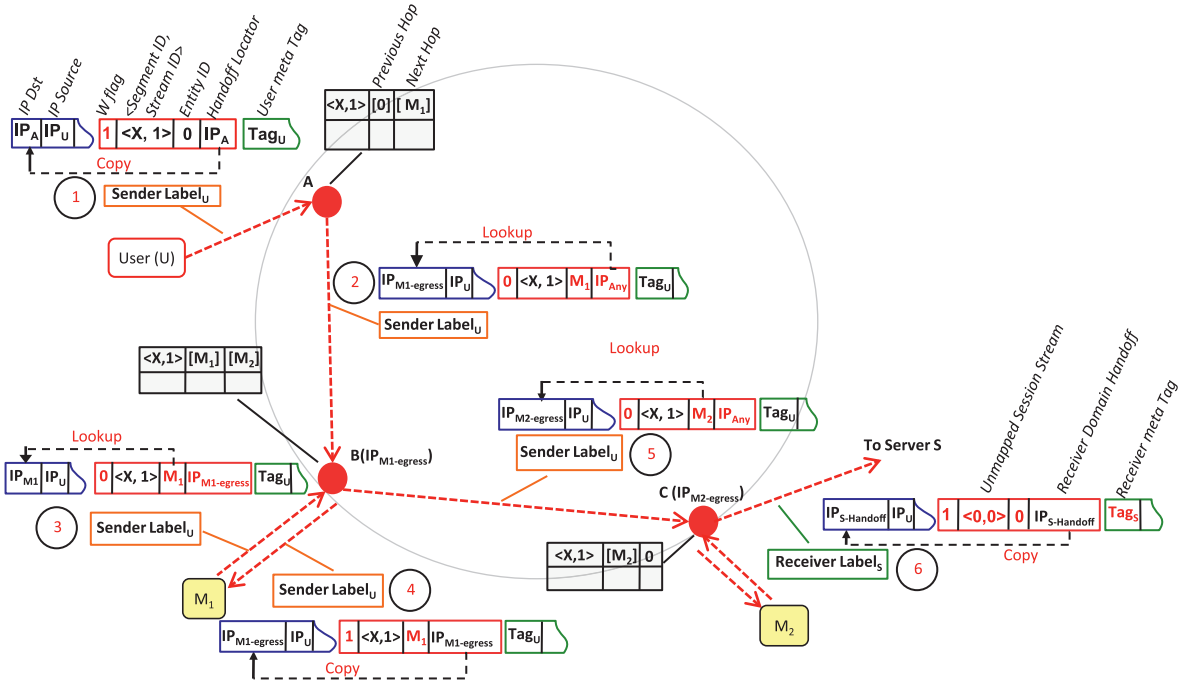


Fig. 16. APLS label switching example.

field. Again, for the *handoff locator,* there are two choices. In this example, node B sets the *handoff locator* to its own IP address. After finishing processing, M_1 (step 4), will *unset* the *W flag bit* and copy the *handoff locator* to the IP destination field. On receiving the packet back from M_1 **(Step 5)**, B checks the *W flag* that indicates that the packet has already been processed by M_1, and so it looks up <X, M_1> to determine the next hop. The next hop is M_2. It *looks up* the locator for *M_2;* re-writes the IP destination field; re-writes the *waypoint entity ID* to M2; *unsets* the *W flag;* and *re-writes* the *handoff locator* to the domain's anycast IP address.

The same sequence of steps (Steps 3 and 4) is repeated for M_2. However, in **Step 6**, node C on receiving the OpenADN message from M_2 realizes that M_2 is the last entity in the <X, 1> segment stream. The mapping of <X, 1>, $M_2 \rightarrow 0$, indicates this. Thus, node C pops out the L3.5 *sender label* and looks up the L3.5 *receiver label* corresponding to the L4.5 receiver label (Tag_R). In this case the L3.5 receiver label corresponding to Tag_R only has the handoff locator mapped ($IP_{S\text{-}Handoff}$) but the next segment stream in the receivers domain is unbound. Therefore Node C *copies* the *handoff locator* in the L3.5 *receiver label* to the IP destination address field; and forwards the packet to the receiver domain's *ingress* node. At this node, the Tag_R will be mapped to the <Segment ID, Stream ID> workflow that the message needs to traverse.

5.3. OpenADN control plane

The OpenADN control plane needs to be *diversified*-different for different application deployment scenarios. The OpenADN control plane is responsible for enforcing application delivery policies by generating a set of application traffic management rules for distributed enforcement in the data plane. These application delivery policies are specific to the application provider (e.g., ASP) and each may use different optimization criteria (cost, performance, availability) with different policies. Thus, no "one" design of the control plane is appropriate. Therefore, OpenADN needs to decouple the control plane from the data plane, allowing many different implementations of the control plane to drive its standardized data plane.

6. OpenADN implementation approach: OpenADN over SDN

SDN presents a framework for network architecture designs based on control plane-data plane separation. The control plane is implemented in software over a logically centralized abstraction. The logically centralized abstraction makes writing control software easier while the software implementation allows the control plane implementation to be flexible. The data plane is implemented over fast, commoditized hardware and is programmable (by the control plane) through a standardized abstraction. Although, other designs may be possible in the future, currently the only public standard available for this standardized abstraction is OpenFlow [6]. OpenFlow presents a *flow-based* abstraction to programming the data plane. Packets are classified into *flows* based on the *flow classification rules* that can be specified over a combination of L2 (including L2.5), L3 and L4 header fields. The *flow classification rules* are installed into the data plane by the control plane through OpenFlow. Associated with each *flow classification rule* is a set of *flow-level actions* that determine *what needs to be done with the packets* identified as belonging to a flow.

Thus, we observe that in an OpenFlow-based SDN, a control application needs to provide two types of rules: (1) *flow classification rules* to identify packets as belonging to a flow of interest (policy equivalence class) to the control application, and (2) *flow-level enforcement rules* to specify the set of actions over each packet belonging to the flow. Given this design paradigm, we observe that the type of control applications that may be designed over SDN depends on whether, (1) the packet carries enough context that would allow the control application to express its interest at the right level of granularity, and (2) support for the desired header fields are present in OpenFlow.

The requirements of the OpenADN architecture exactly match the general motivation of SDN – commoditized data plane and a flexible, diversified control plane. However, the ADN abstraction is not represented in the OpenFlow standard and hence OpenADN control plane applications cannot program the OpenADN data plane through current OpenFlow standard. This requires extending the OpenFlow standard and the OpenFlow base implementation (data plane pipeline and set of actions) with APLS (the data-plane implementation of OpenADN, Section 5.2) header fields and APLS switching actions.

> *Control plane:* The control rules for handling application traffic will be provided by Application Service Providers (ASPs). Note that we make a distinction between network traffic and application traffic. Application traffic is network traffic annotated with application semantics such as context, message boundaries, sessions, etc. for application-level policy enforcements.

Leveraging the power of abstraction, network infrastructure domains implementing SDN can now easily allow externally provided ASP control applications without relinquishing control of their infrastructure. As shown in Fig. 17, SDN has three abstraction layers – virtualization, network operating system and network control applications. The OpenADN control modules are implemented on top of the network control modules. The OpenADN control modules implement *policy virtualization* to partition application traffic into different application-level policy groups and enforce actions to manage the application deployment. They do not interfere with the network management policies implemented by the network control modules in the layer below it. The network control modules implement *network virtualization,* virtualizing the network resources and allocating them to be managed by the corresponding control module. After enforcing application-level policies on the application-traffic, the OpenADN control module hands-off the packet to a network control module to invoke a network transport service implemented within a virtualized network partition.

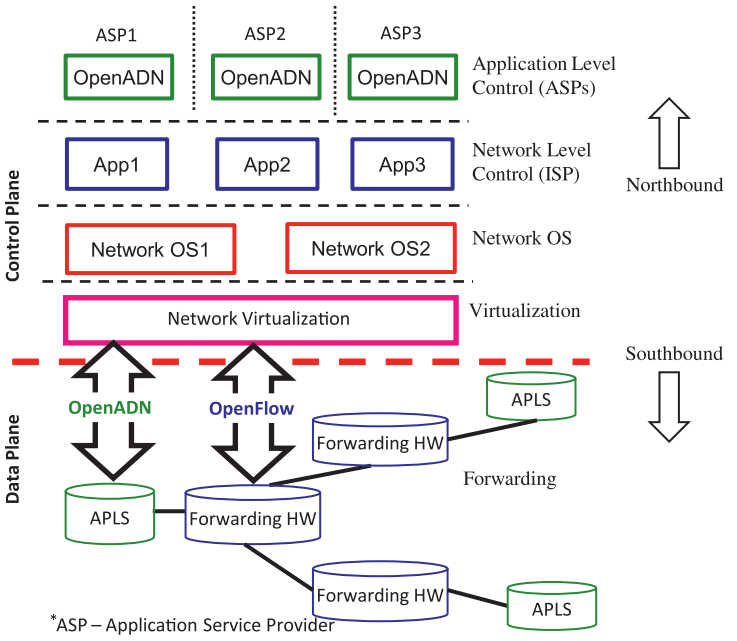


Fig. 17. OpenADN extends the Southbound API with new meta-tags in the header and introduces a northbound interface for policy communication.

Data plane: OpenADN needs to augment OpenFlow to add support for OpenADN processing in the data plane. In the data plane, OpenADN specific processing precedes OpenFlow specific processing. This follows naturally from the layered abstraction in the control plane. Hence, it is required that in the data plane, OpenADN directly interfaces with OpenFlow. We observe that the OpenFlow data plane specification [6] already provides support for explicitly chained virtual tables. Incoming packets are first passed through a generic flow-identification table, which then redirects the packet through a virtual table pipeline for more flow-context specific processing. Using this virtual table support, the OpenADN data plane may interpose application-level policy enforcement (Fig. 18) before handing off the flow for network-level flow processing. We propose a three level naming hierarchy for virtual tables. The first level identifies whether it is performing application-level or network-level processing. The second level identifies the SDN control module that configures the virtual table (e.g., ASP IDs for OpenADN, infrastructure service IDs for OpenFlow). The third level identifies the specific flow-processing context within an SDN control module. In Fig. 18, we only show a packet being explicitly handed-off at level 1 in the hierarchy, from application-level policy enforcement to network level flow processing. However, it is also possible to allow packet handoffs at level 2 and level 3 in the hierarchy, to accommodate layered abstraction in the control plane.

6.1. Initial prototype implementation

We implemented an initial prototype of an OpenADN proxy using the click modular router [19]. Currently, our implementation is limited to UDP-based service access. This allows us to treat OpenADN messages to be one

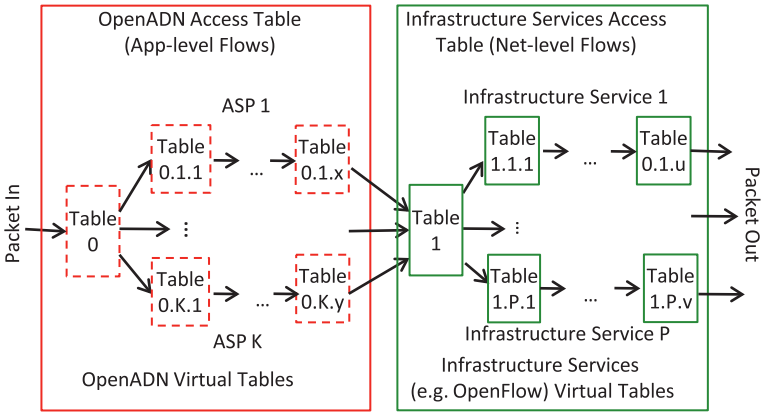


Fig. 18. OpenADN and OpenFlow processing.

datagram long. Therefore, although L3.5 headers are packet-level and L4.5 headers are message-level, the difference is not noticeable in our current implementation. Work is under progress to implement OpenADN over TCP transport.

To validate the functionality, we simulated a simple use-case scenario (Fig. 19) over NS3 [20] showing distributed application deployment over multi-cloud environment. It consists of three application servers (AppServer1$_A$, AppServer1$_B$, and AppServer2), two waypoints (IDS A and IDS B), a user (simulating multiple traffic sources), an OpenADN controller and an OpenADN data plane proxy. Each of these nodes is OpenADN aware.

The scenario implements two OpenADN L4.5 segments – (1) Segment 1 consisting of two *segment streams;* SS1$_A$ = <IDS$_A$, AppServer1$_A$>, and SS1$_B$ = <IDS$_B$, AppServer1$_B$>; and (2) Segment 2 consisting of one *segment stream* SS2 = <AppServer2>. SS1$_A$ and SS1$_B$ also serve as the failover instance for each other (when SS1$_A$ fails SS1$_B$ may takeover and vice versa). The deployment assumes that the ISP OpenADN proxy acts as a central coordinator for steering the application traffic through the different nodes in the segment stream. The nodes themselves are unaware of the next hop; for example IDS$_A$ is not aware that it needs to send the traffic towards AppServer1$_A$ and sends back the messages to the OpenADN proxy after processing it. This allows the controller to be simple in that it needs to configure only the OpenADN proxy while the other OpenADN nodes operate in their default configuration (Implement an OpenADN stub that is statically configured to return the traffic to the interface over which it received it.). In our current implementation, the *controller* is manually configured.

The controller configures the OpenADN proxy with *static* as well as *dynamic* APR policies. The *static* policies include mapping the *meta-tag* in the OpenADN header to the L4.5 segment to which the application traffic is bound. *Dynamic* polices are specified to choose a particular segment stream, if a segment has multiple instances, based on real-time load or fault information.

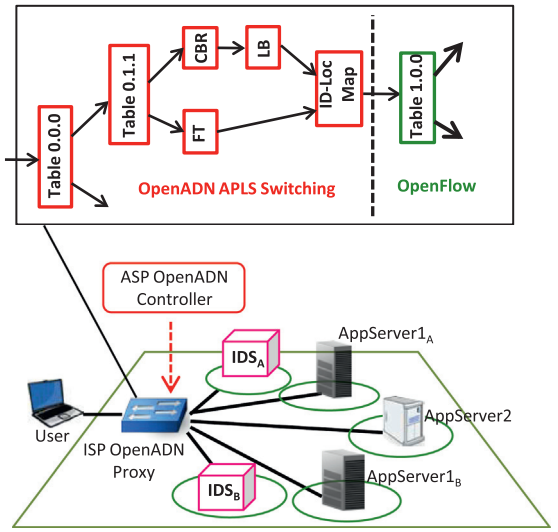


Fig. 19. Use-case scenario: application traffic steering.

In the scenario shown in Fig. 19, the user simulates a distributed user population and sends OpenADN messages over UDP to the ISP OpenADN proxy. It randomly includes either of the two *meta-tags* Tag$_A$ and Tag$_B$ in each of these messages. Each message is first intercepted by Table 0.0.0 in the OpenADN Proxy that classifies the message based on the ASP ID. For ASP ID = 1, the message is forwarded to Table 0.1.1; similarly for ASP ID = 2, the message is forwarded to Table 0.2.1, and so on. Table 0.1.1 determines whether the OpenADN message is already bound to a *segment stream* (part of an OpenADN session with session affinity requirements, i.e. the requirement that all messages needs to be bound to the same segment stream). If the message is unbound, it is sent to the *CBR* (Content-based Router) module that maps the *meta-tag* to the L4.5 segment. In our example, the following two mappings are stored:

$$\text{Tag}_A \rightarrow \text{Segment 1} \rightarrow \textit{Unspecified}$$

$$\text{Tag}_B \rightarrow \text{Segment 2} \rightarrow < \textit{AppServer2} >$$

If the message is carrying Tag$_B$, it is mapped directly to AppServer2 and sent to the *ID-Loc* module that maps the ID of AppServer2 to its IP address and hands-off the message for OpenFlow-based processing. The OpenADN processing may set certain QoS flags (in L3.5 *flags* field) that may be interpreted by the OpenFlow processing pipeline to provide the required differentiated transport services to the message.

If the message is carrying Tag$_A$, then the CBR can only determine that the message belongs to segment 1. It sends the message to the *LB (Load Balancer)* module determine which of the two *segment streams* (SS1$_A$ or SS1$_B$) the message needs to be bound to. Also, in the case that a message arrives at Table 0.1.1 with Tag$_A$ but is already bound to either of the *segment streams,* the message is sent to a *FT (Fault Tolerance)* module that determines whether the *segment stream* that the message is bound to is still *up;* otherwise redirects the message to the failover instance. This scenario is illustrated by the experiment shown in Fig. 20. The *x*-axis represents time (*T*) in seconds and the *y*-axis represents the number of OpenADN messages. We fix the number of bound and unbound OpenADN messages to 1000. The time interval *T* = 0 through *T* = 14, shows load balancing in action for new unbound messages. At *T* = 15, we stop sending unbound messages and only send already bound messages. At *T* = 19, *AppServer1$_A$* fails (thus SS1$_A$ becomes unavailable) and comes back at *T* = 29. During this time, all messages are re-directed to *segment stream SS1$_B$* by the *FT*. After App*Server1$_A$* comes back at *T* = 30, the *FT* resumes sending messages bound to SS1$_A$ towards SS1$_A$, as usual. Note that the proportion of traffic directed to the two *segment streams* dynamically follows the load balancing as specified by the ASP and listed at the bottom of Fig. 20.

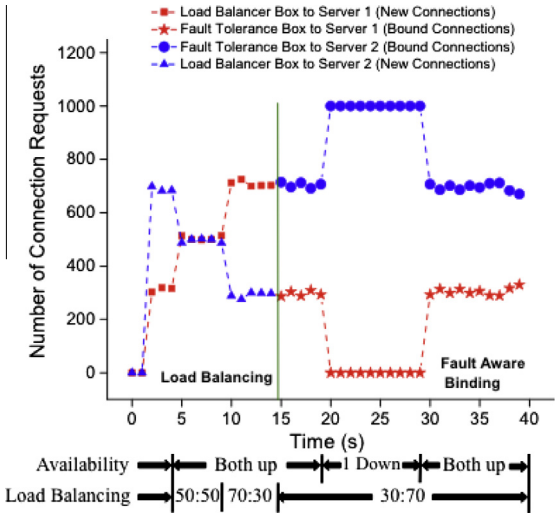


Fig. 20. Validation of the use case presented in Fig. 19.

6.2. Discussion

Adding OpenADN headers to application messages and network packets will add bandwidth and computation overheads to the network data plane. However, we expect the functional benefits of OpenADN in terms of simplified and dynamic management and control of large-scale, massively distributed application deployment and delivery to be compelling enough to justify these overheads. Although our current prototype implementation is not performance tuned to provide an objective evaluation, we may analytically claim that these overheads will be well within the acceptable limits and should not be a major deterrent to the adoption of OpenADN. First, packet level (L3.5) OpenADN header processing at intermediary hops is similar to MPLS label-swapping on fixed-length identifiers. Second, message-level header processing will be done only at intermediary OpenADN proxies and not at each application-level entity. Third, although OpenADN header allows L4.5 labels to be stacked over each other and hence the actual size of the L4.5 header may vary, in real use-cases we generally expect the stacking level to be between two and four. In cases where each message is independent, that is there is no session affinity, each message shall carry two labels (in most cases) – one for each sender and receiver. For cases that need to implement session-affinity, each message will carry four labels (in most cases) – two for each sender and receiver. One of these labels shall be bound to a session (indicating a path through session-level entities such as firewalls and IDS') whereas the other label shall allow the message to be routed based on the message content (message headers and application data) through transformation and optimization functions. Fourth, L3.5 carries the L4.5 *meta-tag* to *session-stream* mapping (plus additional fields to implement hop-by-hop routing, see Section 5.2.1) of the currently active L4.5 meta-tag label. Therefore, it is sufficient to carry one L3.5 label (Section 5.2.1) in each packet, or two if the implementation needs to save the last accessed L4.5 session segment in the packet for each sender and receiver domains.

7. Related work

Several research efforts have focused on the problem of adding explicit support for middleboxes into the Internet architecture [1,21]. However, the ad-hoc network configuration techniques and explicit middlebox support in HTTP somehow managed the show and thus the motivation for adopting an architectural change never became too strong. We believe that in the context of multi-cloud environments the problem is more urgent. Also, the problem is very different since the environment imposes a new requirement that the network infrastructure ownership is separate from the application provider. None of the previous proposals try to abstract out application-level semantics into standard representation for a generic and high-performance implementation while still preserving *some richness* of application diversity; allowing application level policies to be enforced on third-party infrastructures.

Research efforts on in-network processing may be considered to lie in the broader periphery of the OpenADN research scope. The in-network processing proposed in ***active networks*** *research* [22] allowed ASPs to write programs in the packet headers. This violated the administrative boundary between ASP and ISP by giving direct control of ISP's router to ASPs. OpenADN avoids this by providing a very restricted and standardized switching abstraction making some allowance for representing the application context. OpenADN may be considered similar to **Rule-based Forwarding architecture (RBF)** [23]. RBF proposes a flexible forwarding plane design by forwarding packets to a *rule* instead to a *destination address.* The key difference is that RBF provides more *flexibility* at the cost of *performance* whereas OpenADN is designed to tradeoff *flexibility* for *higher performance.*

Serval [24] is another recent approach that addresses the problem of accessing geographically distributed service deployments. Serval provides a point solution for accessing distributed services through a service router infrastructure. The service router infrastructure is similar in spirit to the requirement of *outsourcing application-level routing* in OpenADN. However, OpenADN provides a richer interface than just service IDs to a generic middlebox-switching solution which allows flexible implementation of any service access mechanism by composing specific mechanisms such as CBR, load balancer, cluster fault-manager, etc. Also, OpenADN allows application context to be included into the switching abstraction. Other distinguishing features include support for middlebox-switching both in the data and control planes and support for both sender and receiver policies.

CloudNaaS [25] proposed an OpenFlow-based data plane to allow ASPs deploy off-path middleboxes in Cloud datacenters. OpenADN is different from CloudNaas in that it provides a session-layer abstraction to applications preserving session-affinity properties in an environment where the application servers and virtual appliances may need to scale dynamically. OpenADN allows both the user and ASP policies, unlike CloudNaaS that allows only ASP policies. Also, CloudNaas does not allow application-level flow processing like OpenADN, thus it is unable to provide support for dynamic *service partitioning* and *replication*,

routing on user context and *context-based service composition*.

L7 traffic steering provided by OpenADN is a more dynamic data-plane primitive as compared to earlier proposals such as NUTTS [1], SIP [26], Serval [24], UIA [27] and DONA [28] which use separate control messages (routed-by-name instead of address) to setup an end-to-end or end-middle-end session over which data plane messages are exchanged. In this regard, OpenADN is more similar to the Named Data Network (NDN) [29] proposal in which each *request message* is routed independently towards the data source. However, NDN lacks service-centric semantics including *session affinity* and *service chaining*.

Some other works, such as **CoMB** [30] and **APLOMB** [31] have proposed delegating middlebox services to third-party providers. However, they directly conflict with the design principle of OpenADN that third party providers should not be able to have access to the ASPs application-level data for privacy reasons. Therefore, OpenADN provides a platform, where, instead of third parties, ASPs themselves can manage their middleboxes distributed across different cloud sites more easily.

8. OpenADN deploy-ability

Deploying a new network architecture is hard and hence deploy-ability of an architecture is a metric to judge the chances of an architecture to impact the real-world. Hence, while designing OpenADN we paid special attention to addressing this issue. Our case for the deploy-ability of OpenADN is based on the following observations:

1. Layer 4.5 is an application-level header and it should be relatively easy to introduce this as an extension header to application protocols such as X-prefixed headers in HTTP. Also, application providers stand to benefit most from OpenADN (in terms of managing/controlling their large-scale distributed application deployments) and hence should adopt this new application-layer header that can easily piggyback over existing application message frame formats.
2. Introducing a new layer 3.5 is more challenging than layer 4.5. However, note that layer 3.5 is an overlay over IP and hence does not assume anything at layer 3 or below to change. The main difficulty, however is that since it is between layer 4 and layer 3 we would ideally expect it to be implemented as a kernel module which may not always be feasible. One way to address this issue is through user-space libraries that run on raw-IP sockets. In our current prototype implementation, however, we provide layer 3.5 as a kernel-level click element while layer 4.5 is provided as a user-level click element.
3. In general, network virtualization technologies have started leading the way for new tunneling encapsulation techniques including STT [15], VXLAN [16] and NVGRE [17]). OpenADN may either piggyback on one of these new tunneling encapsulation headers or have its own header if it is successful.

9. Future work

The implementation of OpenADN is still a work-in-progress. The initial prototype implementation is simply to provide a proof-of-concept for our proposed design. We are working towards a more generic implementation that will address the following:

1. The current implementation is not tuned for performance and hence it is not possible to provide insights into the performance overhead of introducing a new software layer. Our future implementation will address this aspect.
2. For our initial proof-of-concept implementation we have used a centralized controller. However, in future work we are working towards more distributed approaches. There are several aspects of distributing the controller that we are presently considering. The first aspect is to add redundancy to address availability issues by adding a hot-standby. The second aspect is to distribute load by partitioning the control functions across several controllers located at a central location. The third aspect is to distribute the controllers across different sites and constructing a hierarchical controller deployment. This requires logically partitioning the control functions into multiple feedback loops of different latencies.
3. The current implementation uses only UDP transport. In the future, we will add support for TCP transport including support for TCP connection pooling and multiplexing.
4. Currently, we use custom OpenADN message frame formats. Future implementations shall provide support for at-least HTTP frame formats with the OpenADN L4.5 header piggybacking on an X-prefixed field in the HTTP header.

10. Discussion and conclusions

In this paper, we presented a new service-centric network design called OpenADN that will allow ASPs to fully leverage multi-cloud application deployment environments to more efficiently deploy and deliver their services to a highly distributed, dynamically changing and mobile user population. OpenADN has the following features:

1. It provides a new OpenADN session abstraction that creates a logical channel between the user and a service. Both, the user and the service may be *virtual* entities representing a set of application-level routing (APR) policies which are basically a set of *access policies* for accessing the service or the user and specified at *per-message* granularity. Using the new OpenADN L4.5 message framing header, each OpenADN message, is forwarded based on a *meta-tag* (instead of a destination address). The meta-tag can be uniquely mapped to the right set of APR policies that need to be applied on it by the channel. APR policies include application-level traffic steering policies such as selecting the right ser-

vice partition/replica, service chaining, service composition and user access to different value-added network services.

2. OpenADN allows the ASP to delegate the APR function to third-party infrastructure providers such as ISPs, CSPs, and CDNs without compromising on the privacy and security of application-level data. This is done by using the concept of the *meta-tag* in the OpenADN message frame header (L4.5 APLS header). The *meta-tag* encodes the result of the message-classification that is done at an ASP trusted entity such as the end-user host or the service end-host. Therefore, this prevents the need to do session reconstruction and/or deep-packet inspection in the third-party network node to be able to apply APR policies. Also, note that apart from meeting the security and privacy requirements, this design choice also makes for a more efficient, simple and standardize-able (hence commoditize-able) OpenADN data-plane design in-spite of the diversity of the application layer.
3. OpenADN introduces a new layer 3.5 that provides a wrapper around existing layer 4 transport protocols including TCP and UDP. The idea is to make a general transport layer for application delivery that supports a mixed application delivery workflow consisting of both, message-level entities (requiring data plane proxying, transport connection splicing and multiplexing) and packet-level middleboxes (that do not serve as a transport end-point).
4. OpenADN is designed for control and data plane separation in which the data plane is standardize-able and hence delegate-able to third-party infrastructure providers; while all the application-related diversity and intelligence is moved to the control-plane, which is under the control of the ASP. We are in the process of incorporating OpenADN into the general SDN framework. Currently, a set of simple APIs, for direct communication between the OpenADN controller and the OpenADN data-plane node, has been defined. To port the OpenADN deployment to an SDN-based deployment, this direct communication between the OpenADN controller and the OpenADN data-plane will need to be mediated by the SDN controller. Thus, a new set of Northbound and Southbound APIs for SDN will need to be designed to support OpenADN.

In summary, OpenADN is an effort to evolve the current Internet design to make it suitable for modern (and future) Internet-scale, massively-distributed application deployment and delivery. OpenADN is designed to be an overlay over IP, thus increasing its deploy-ability significantly. Also, we believe that this is the right time to design a new architecture such as OpenADN while the networking community is still trying to converge on the SDN ideas.

Acknowledgement

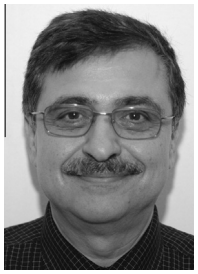
This work has been supported under the Grant ID NPRP 6-901-2-370 for the project entitled "Middleware Architecture for Cloud Based Services Using Software Defined Networking (SDN)," 2013–2016, which is funded by the Qatar National Research Fund (QNRF).

References

[1] S. Guha, P. Francis, An end-middle-end approach to connection establishment, ACM SIGCOMM (2007).
[2] P. Gill, M. Arlitt, Z. Li, A. Mahanti, The flattening Internet topology: natural evolution, unsightly barnacles or contrived collapse, PAM (2008) 1–10.
[3] S. Paul, R. Jain, OpenADN: mobile apps on global clouds using OpenFlow and software defined networking, in: IEEE Globecom Workshop on Management and Security Technologies for Cloud Computing (ManSec-CC), 2012.
[4] S. Paul, R. Jain, J. Pan, J. Iyer, D. Oran, OpenADN: a case for open application delivery networking, ICCCN (July) (2013).
[5] J. Salz, A.C. Snoeren, H. Balakrishnan, TESLA: a transparent, extensible session-layer architecture for end-to-end network services, USITS (2003).
[6] Open Networking Forum, OpenFlow Switch Specification, Version 1.3.1, September 6, 2012. <https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.3.1.pdf>.
[7] F5 Networks. <http://www.f5.com/>.
[8] Citrix Systems Inc. <http://www.citrix.com/>.
[9] Blue Coat Systems Inc. <http://www.bluecoat.com/>.
[10] Oracle Weblogic. <http://www.oracle.com/technetwork/middleware/weblogic/overview/index.html>.
[11] IBM Websphere. <http://www-01.ibm.com/software/websphere/>.
[12] D. Chappell, Enterprise Service Bus, O'Reilly Media, Inc., 2004.
[13] HAProxy. <http://haproxy.1wt.eu/>.
[14] D.A. Joseph, A. Tavakoli, I. Stoica, A policy-aware switching layer for data centers, SIGCOMM (2008).
[15] B. Davie, Ed., J. Gross, A Stateless Transport Tunneling Protocol for Network Virtualization (STT), IETF Draft draft-davie-stt-03.txt, March 12, 2013, 19 pp. <http://tools.ietf.org/html/draft-davie-stt-03>.
[16] M. Mahalingam, D. Dutt, K. Duda, et al., VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks, IETF Draft draft-mahalingam-dutt-dcops-vxlan-03.txt, February 22, 2013, 22 pp. <http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-03>.
[17] M. Sridharan, A. Greenberg, N. Venkataramiah, et al., NVGRE: Network Virtualization using Generic Routing Encapsulation, IETF Draft draft-sridharan-virtualization-nvgre-02.txt, February 25, 2013, 17 pp. <http://tools.ietf.org/html/draft-sridharan-virtualization-nvgre-02>.
[18] H. Nielsen, S. Thatte, Web services routing protocol, Microsoft (October) (2001).
[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. Frans Kaashoek, The click modular router, ACM Trans Comput Syst 18 (3) (2000) 263–297.
[20] NS3. <http://www.nsnam.org/>.
[21] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, S. Shenker, Middleboxes no longer considered harmful, OSDI (2004).
[22] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, G.J. Minden, A survey of active network research, IEEE Commun. (1997).
[23] L. Popa, N. Egi, S. Ratnasamy, I. Stoica, Building extensible networks with rule-based forwarding (RBF), USENIX OSDI (2010).
[24] E. Nordström et al., Serval: an end-host stack for service-centric networking, NSDI (April) (2012).
[25] T. Benson, A. Akella, A. Sheikh, S. Sahu, CloudNaaS: a cloud networking platform for enterprise applications, in: Symposium on Cloud Computing SOCC, 2011.
[26] J. Rosenberg et al., SIP: Session Initiation Protocol, IETF RFC 3261, June 2002.
[27] B. Ford, Unmanaged Internet Protocol: Taming the Edge Network Management Crisis, HotNets-II, November 2003.
[28] T. Koponen, M. Chawla, B.G. Chun, et al., A data-oriented (and beyond) network architecture, SIGCOMM Comput. Commun. Rev. 37 (4) (2007) 181–192.
[29] V. Jacobson, D.K. Smetters, J.D. Thornton, M.F. Plass, N.H. Briggs, R.L. Braynard, Networking named content, CoNEXT (2009).
[30] Vyas Sekar, S. Ratnasamy, M.K. Reiter, N. Egi, G. Shi, The middlebox manifesto: enabling innovation in middlebox deployments, ACM HotNets (2011).
[31] Justine Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, V. Sekar, Making middleboxes someone else's problem, network processing as a cloud service, ACM SIGCOMM (2012).

Subharthi Paul received his BS degree from University of Delhi, Delhi, India, and Masters degree in Software Engineering from Jadavpur University, Kolkata, India. He is presently a doctoral student in Computer Science and Engineering at Washington University in St. Louis, MO USA. His primary research interests are in the area of future Internet architectures including Software Defined Networks (SDN), datacenter network architectures and application delivery networking for cloud and multi-cloud environments. He is a student Member of IEEE.

Raj Jain is a Fellow of IEEE, a Fellow of ACM, a winner of ACM SIGCOMM Test of Time award, CDAC-ACCS Foundation Award 2009, and ranks among the top 50 in Citeseer's list of Most Cited Authors in Computer Science. Dr. Jain is currently a Professor of Computer Science and Engineering at Washington University in St. Louis. Previously, he was one of the Co-founders of Nayna Networks, Inc. – a next generation telecommunications systems company in San Jose, CA. He was a Senior Consulting Engineer at Digital Equipment Corporation in Littleton, Mass and then a professor of Computer and Information Sciences at Ohio State University in Columbus, Ohio. He is the author of ''Art of Computer Systems Performance Analysis,'' which won the 1991 ''Best Advanced How-to Book, Systems'' award from Computer Press Association. His fourth book entitled '' High-Performance TCP/IP: Concepts, Issues, and Solutions,'' was published by Prentice Hall in November 2003. Recently, he has co-edited ''Quality of Service Architectures for Wireless Networks: Performance Metrics and Management,'' published in April 2010.



Mohammed Samaka is an associate professor of Computer Science in the Department of Computer Science and Engineering (CSE), College of Engineering at Qatar University. He obtained his PhD and Master from Loughborough University in England and a Post Graduate Diploma in Computing from Dundee University in Scotland. He obtained his Bachelor of Mathematics from Baghdad University. His current areas of research include Wireless Software Architecture and Technology, Mobile Applications and Services, Networking, e-Learning, and Computing Curricula Development.

Jianli Pan received his B.E. from Nanjing University in Posts and Telecommunications (NUPT), Nanjing, China, and M.S. from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China. He is currently a PhD student in the Department of Computer Science and Engineering in Washington University in Saint Louis, MO, USA. His current research is on the Future Internet architecture and related topics such as routing scalability, mobility, multihoming, and Internet evolution. His recent research interest also includes intelligent buildings, green buildings, and smart energy in networking context.