Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Raj Jain, Chair
Viktor Gruev
Chenyang Lu
Paul Min
Mohammed Samaka

Software Defined Application Delivery Networking

by

Subharthi Paul

A dissertation presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

June 2014
Saint Louis, Missouri

Contents

List of Tables

List of Figures

Acknowledgments

I would like to express my deepest gratitude to my advisor, Prof. Raj Jain for granting me the opportunity to be a part of his research group and for his support and guidance throughout the research. I am extremely thankful to him for providing me the intellectual freedom to work on the ideas I found interesting while at the same time teaching me how to express abstract ideas in more concrete terms that may be more objectively evaluated.

I would like to thank my co-workers, lab-mates and friends with whom Ive worked, learnt and have had fun together during the course of many different projects here. I would especially like to thank Chakchai, Abdel, and Jianli with whom I have collaborated on several research projects, many of which resulted in joint publications in leading journals and conferences. I would also like to thank Gabor who contributed to the AppFabric project and to Lav and Deval who will now lead this project forward.

I would like to thank all the amazing faculty members whose courses I took and learnt so much. Special thanks to the members of my dissertation committee, Prof. Gruev, Prof. Lu, Prof. Min and Prof. Samaka, who have reviewed my research and given invaluable advise to making it better.

Another important part of my graduate life has been my internship at Intel where I was mentored by Dr. Shi-Wan Lin. It was a tremendous learning experience for me and I cannot thank Shi-Wan enough for guiding me through it and for being such an influence in my life.

I am fortunate to have many great friends who are no less than family to me. Special thanks to Ritun, Sandip, and Saurav (wanted to avoid the possibility of any confrontation and so put their names strictly in alphabetical order) for the spontaneous, long, intellectual discussions on just about anything under the sun.

Also, I could not possibly be a researcher without the coffee and ambience of the Kayaks coffee shop, at the intersection of Forest Park and Skinker, where I spent most of my graduate life.

I am thankful to my parents and my sister for everything they mean to me and everything I mean to them. Also, I am fortunate to have an amazing extended family (parents-in-law and brother-in-law) and would like to thank them for always supporting me and believing in me.

I would like to thank James uncle for simply being "amazing" and defining what "amazing" is.

Finally, I would like to thank Anu, my wife and my partner in crime. She is the reason I am, and beyond that, nothing really matters.

Subharthi Paul

Washington University in Saint Louis
June 2014

Dedicated to the rational mind that seeks.. and then seeks some more.

ABSTRACT OF THE DISSERTATION

Software Defined Application Delivery Networking

by

Subharthi Paul

Doctor of Philosophy in Computer Science

Washington University in St. Louis, June 2014

Research Advisor: Professor Raj Jain

In this thesis we present the architecture, design, and prototype implementation details of **AppFabric**. AppFabric is a next generation application delivery platform for easily creating, managing and controlling massively distributed and very dynamic application deployments that may span multiple datacenters.

Over the last few years, the need for more flexibility, finer control, and automatic management of large (and messy) datacenters has stimulated technologies for virtualizing the infrastructure components and placing them under software-based management and control; generically called Software-defined Infrastructure (SDI). However, current applications are not designed to leverage this dynamism and flexibility offered by SDI and they mostly depend on a mix of different techniques including manual configuration, specialized appliances (middleboxes), and (mostly) proprietary middleware solutions together with a team of extremely conscientious and talented system engineers to get their applications deployed and running. AppFabric, 1) automates the whole control and management stack of application

deployment and delivery, 2) allows application architects to define logical workflows consisting of application servers, message-level middleboxes, packet-level middleboxes and network services (both, local and wide-area) composed over application-level routing policies, and 3) provides the abstraction of an application cloud that allows the application to dynamically (and automatically) expand and shrink its distributed footprint across multiple geographically distributed datacenters operated by different cloud providers. The architecture consists of a hierarchical control plane system called Lighthouse and a fully distributed data plane design (with no special hardware components such as service orchestrators, load balancers, message brokers, etc.) called OpenADN. The current implementation (under active development) consists of 10000 lines of python and C code.

AppFabric will allow applications to fully leverage the opportunities provided by modern virtualized Software-Defined Infrastructures. It will serve as the platform for deploying massively distributed, and extremely dynamic next generation application use-cases, including:

- **Internet-of-Things/Cyber-Physical Systems:** Through support for managing distributed gather-aggregate topologies common to most Internet-of-Things(IoT) and Cyber-Physical Systems(CPS) use-cases. By their very nature, IoT and CPS use cases are massively distributed and have different levels of computation and storage requirements at different locations. Also, they have variable latency requirements for their different distributed sites. Some services, such as device controllers, in an Iot/CPS application workflow may need to gather, process and forward data under near-real time constraints and hence need to be as close to the device as possible. Other services may need more computation to process aggregated data to drive long term business

intelligence functions. AppFabric has been designed to provide support for such very dynamic, highly diversified and massively distributed application use-cases.

- **Network Function Virtualization:** Through support for heterogeneous workflows, application-aware networking, and network-aware application deployments, AppFabric will enable new partnerships between Application Service Providers (ASPs) and Network Service Providers (NSPs). An application workflow in AppFabric may comprise of application services, packet and message-level middleboxes, and network transport services chained together over an application-level routing substrate. The Application-level routing substrate allows policy-based service chaining where the application may specify policies for routing their application traffic over different services based on application-level content or context.

- **Virtual worlds/multiplayer games:** Through support for creating, managing and controlling dynamic and distributed application clouds needed by these applications. AppFabric allows the application to easily specify policies to dynamically grow and shrink the application's footprint over different geographical sites, on-demand.

- **Mobile Apps:** Through support for extremely diversified and very dynamic application contexts typical of such applications. Also, AppFabric provides support for automatically managing massively distributed service deployment and controlling application traffic based on application-level policies. This allows mobile applications to provide the best Quality-of-Experience to its users without

This thesis is the first to handle and provide a complete solution for such a complex and relevant architectural problem that is expected to touch each of our lives by enabling exciting new application use-cases that are not possible today. Also, AppFabric is a non-proprietary

platform that is expected to spawn lots of innovatins both in the design of the platfrom itself and the features it provides to applications. AppFabric still needs many iterations, both in terms of design and implementation maturity. This thesis is not the end of journey for AppFabric but rather just the beginning.

Chapter 1

Introduction

Over the last few years, the need to have more flexibility, finer control and automatic management of large (and messy) datacenters has stimulated the development of new technologies, tools and software to virtualize the different infrastructure components including compute, network and storage. Software-defined Infrastructure or SDI is a generic term that is used to refer to such virtualized infrastructures with automatic software-based control and management. Some examples of SDI implementations include OpenStack[94], EC2[35], Eucalyptus[38] , CloudStack[28], OpenDayLight[112], and FloodLight[43]. Clearly, this added flexibility and dynamism of the underlying infrastructure layer brings new opportunities to the way we deploy our applications over them. These include:

- **On-demand, dynamic allocation and management of virtual resources:** SDI allows resources to be allocated dynamically. Therefore, unlike physical infrastructures, applications do not need to pre-allocate all the resources, allowing them to scale-up and scale-down on-demand. Also, virtualization can more effectively mask failures in the underlying physical infrastructure providing the abstraction of an always-available resource pool to the consumer.
- **Service-oriented Interface for accessing virtual resources:** SDI makes virtual resources accessible through a service-oriented interface. This allows different resource providers including Cloud Service Providers (CSPs) and Network Service Providers (NSPs) to make their resources available by publishing service APIs for their virtual resources. Application Service Providers (ASPs) consume these APIs to dynamically launch their applications across resources leased from different resource providers. This allows ASPs to dynamically distribute their applications across different providers to

build application deployment topologies optimizing the cost of deployment, reliability and latency.

- **Delegation of administrative control of virtual resources from resource provider to resource consumer:** Unlike physical resources, SDI allows the resource provider, such as CSPs and NSPs, to safely delegate the administrative control of a virtual resource to the consumer. This allows the resource consumer (ASP) to create and manage their own application clouds consisting of virtual compute and storage resources over a programmable virtual network. It can setup its own application specific policies within its private application cloud. For example, the application can now easily control traffic priority (differentiated services) between its compute nodes by programming the virtual network between them.

- **Convergence of infrastructure layers:** Convergence of infrastructure layers is one of the most exciting possibilities that would be enabled by AppFabric. Presently, the different infrastructure components including compute, storage and network are managed separately, and often by separate ownerships. Each of these components try to optimize their own specific context instead of optimizing jointly for the application that runs on them. For example, the network is largely unaware of application requirements and the applications try to use whatever the network provides to them and adapt in application-specific ways (e.g. application overlays [30, 6, 134, 144] , multi-streams [44, 128], differential compression [116, 22], intelligent and contextual degradation [114, 113], etc.) to handle situations where the network does not provide them with the required support. With SDI, it may be possible to end this constant tussle between the different infrastructure layers by creating virtual resources and placing them under the direct control of the application.

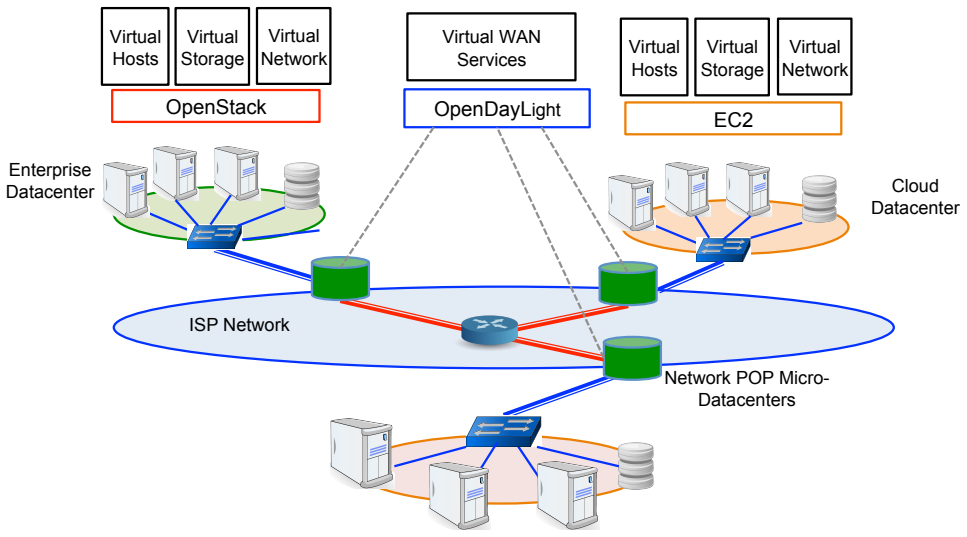

Figure 1.1: Virtual Software-Defined Infrastructure

Therefore,in AppFabric we propose a 2-level control and management architecture. At the bottom level, platform specific SDI controllers would be responsible for directly managing the physical infrastructure and creating a virtual resource pool.These virtual resources are then leased to different ASPs to deploy their applications on top of it. The ASP runs an application controller that is responsible for creating a distributed resource pool for the application by leasing virtual resources from multiple geographically distributed resource providers. The application controller is also responsible for managing the deployment and delivery of the application over this distributed virtual resource pool.

However, current applications are not designed to leverage these new opportunities provided by SDI. In-fact current application delivery practices consists of a loose mix of several different techniques including manual configuration of infrastructure components, plumbing-in specialized appliances (middleboxes), and use of (mostly) proprietary middleware solutions together with a team of extremely conscientious and highly talented system engineers who get the show running. Clearly, this traditional approach towards application delivery fails to exploit fully the advantages of SDI over physical infrastructures.

1.1 Objectives

In this thesis we will describe the architecture, design and proof-of-concept implementation of AppFabric. AppFabric is a next generation application delivery platform for easily creating and managing massively distributed and very dynamic application deployments that may span multiple datacenters. It is designed to allow future application deployments to fully leverage the flexibility and dynamism of SDI. More specifically, the AppFabric architecture seeks to achieve the following objectives:

- **Automated creation and maintenance of an application delivery network or ADN:** An application delivery network or ADN is an integrated infrastructure comprising of both, message-level devices and packet-level devices, that host application-layer services as well as network-layer services. Application-layer services are those that need to access and act upon application-layer data whereas network-layer services are those that need to access only network layer (Layer 3 and below of the OSI stack) information. Application-layer services may be hosted either on message-level devices or packet-level devices. Examples of application-layer services on message-level devices include application servers such as web servers, database servers, etc. and message-level middleboxes such as Web-Application Firewalls (WAF), application transcoders, SSH offloaders, etc. Example of application-layer services on packet-level devices include intrusion detection and prevention systems(IDS/IPS), packet scrubbers, WAN optimizers, etc. On the other hand, network-layer services, as the name indicates, perform only packet-level functions such as packet routing and forwarding and therefore need access to only Layer 3 (and below) packet headers. Therefore, network-layer services are always hosted on packet-level devices. The AppFabric platform should be able to seamlessly integrate these different service/device types and automatically create and maintain an ADN over which the application can be deployed and delivered.
- **Automated deployment of a distributed ADN over multi-datacenter environments:** The platform should not only be able to create an ADN on a single datacenter but should also support use-cases where the ADN may need to be deployed over multiple datacenters. This will allow AppFabric to support massively distributed application use-cases including Internet-of-Things/Cyber-Physical Systems, virtual worlds, online games and mobile apps.

- **Runtime automated control of a distributed ADN:** The platform needs to allow the application administrator to configure policies for distributing the ADN deployment across multiple datacenters and managing and controlling the deployment at runtime. These policies include specifying how to distribute the ADN deployment initially and during the runtime when and where to instantiate/shutdown/move new/existing instances to support change in the application context including user mobility, user access patterns (load, location, etc.), planned maintenance, and infrastructure failures.

- **Automatically create and manage a virtual resource pool for deploying the ADN:** The platform needs to automatically provision and manage a virtual resource pool as required by the ADN. Also, it needs to be able to dynamically provision and un-provision resources as required in order to optimize the cost of delivering the application. There are two high-level technical challenges that it needs to address to maintain this virtual resource pool:

  - **Leased vs. owned resources:** The virtual resource pool could comprise of both leased resources (from cloud providers and network providers) and enterprise-owned resources (private datacenters and enterprise networks). During runtime, the platform needs to automatically tradeoff between the security and cost attributes of deploying the application on leased resources with the performance of the application in terms of delay and distribution factors. Also, the platform takes care of all the authentication, authorization and billing functions with the different providers on behalf of the ASP.
  - **Interoperability across multiple SDI platforms:** The virtual resource pool could comprise of resources from multiple SDI platforms including OpenStack, EC2, CloudStack, OpenDayLight, etc. The platform needs to be able to interoperate across these different platforms so that the ADN can be easily distributed over resource from many different providers.

1.2 Approach

Fig. 1.2 shows the schematic view of the AppFabric platform architecture. As shown in the figure, AppFabric sits between the virtual resource layer and the application, and creates the

necessary abstractions through its northbound (between the platform and the application) and southbound (between the platform and SDI) interfaces.

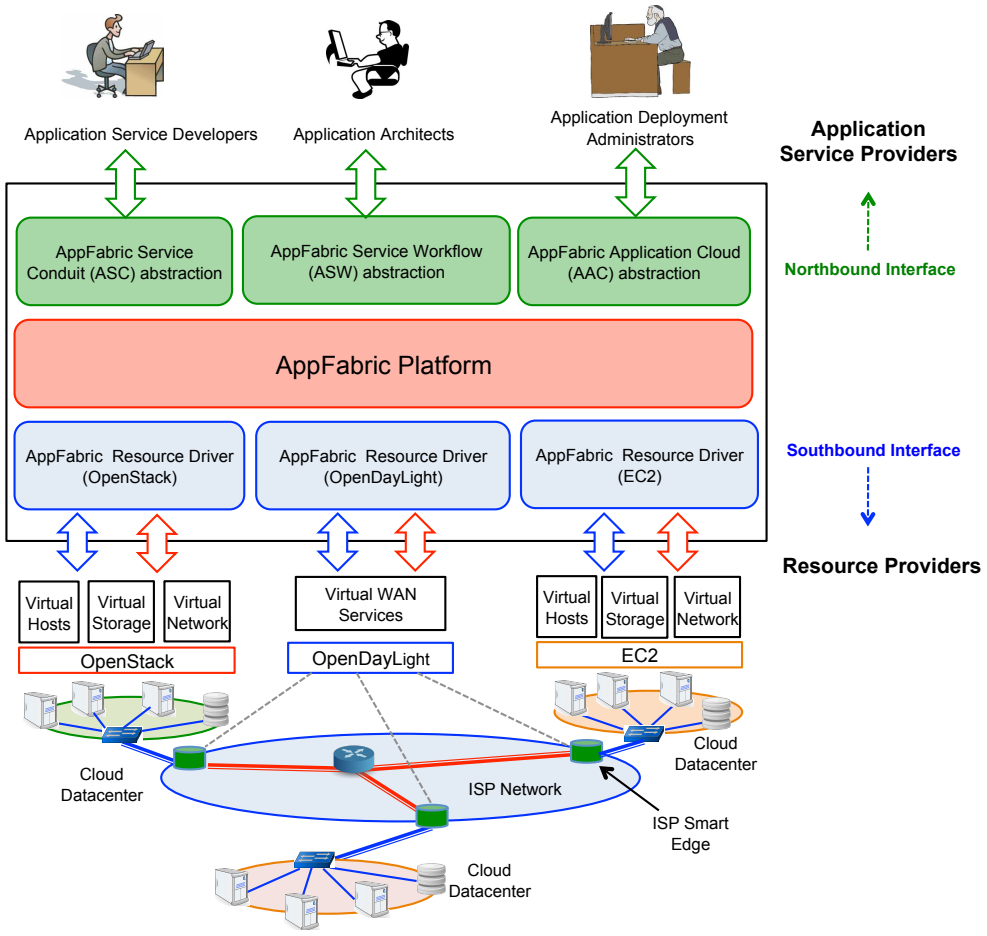

Figure 1.2: A Schematic View of the AppFabric Platform

- **The northbound interface:** The northbound interface is defined between the AppFabric platform and the application. It provides three key abstractions:
  - **The AppFabric Service Conduit abstraction:** The AppFabric Service Conduit (ASC) abstraction provides a communication channel through which a *service* may connect to the AppFabric platform and communicate with other services in the application. The ASC frees the service developer from having to know the service' deployment context (the identity and location of the service(s) to which it will need to communicate at runtime) at design time. Also, the service does

not need to statically bind the communication channel to any particular transport mechanism. The service just reads and writes messages/packets from the conduit. the platforms runtime mechanism will take care of binding the communication channel to the appropriate destination(s) over the appropriate transport(s). It may be noted that services need to be AppFabric-aware to connect the platform thorough the ASC. AppFabric-unaware services can also connect to the platform through a AppFabric Gateway Node (AGN) that is described in chapter 4.

– **The AppFabric Service Workflow abstraction:** The AppFabric Service Workflow (ASW) abstraction allows the application architect to choreograph the application over a set of independent services. These services could be both application-layer services or network-layer services and maybe hosted over message-level devices or packet-level devices. Some of these services may be implementing application logic (e.g. application servers, web servers, etc.) while others may be part of the deployment environment (e.g. Web-application firewalls, Intrusion Detection Systems, etc.). Also, some of these services may not be directly under the control of the ASP but operated by a third-party provider. The key mechanism underlying service choreography in AppFabric is application-level policy routing or APR. The application architect may specify message/packet routing policies based on application-layer content or context of the message/packet and these policies are compiled into forwarding rules to be enforced by the platform on the application traffic. Fig. 1.3 shows a schematic representation of an ASW.

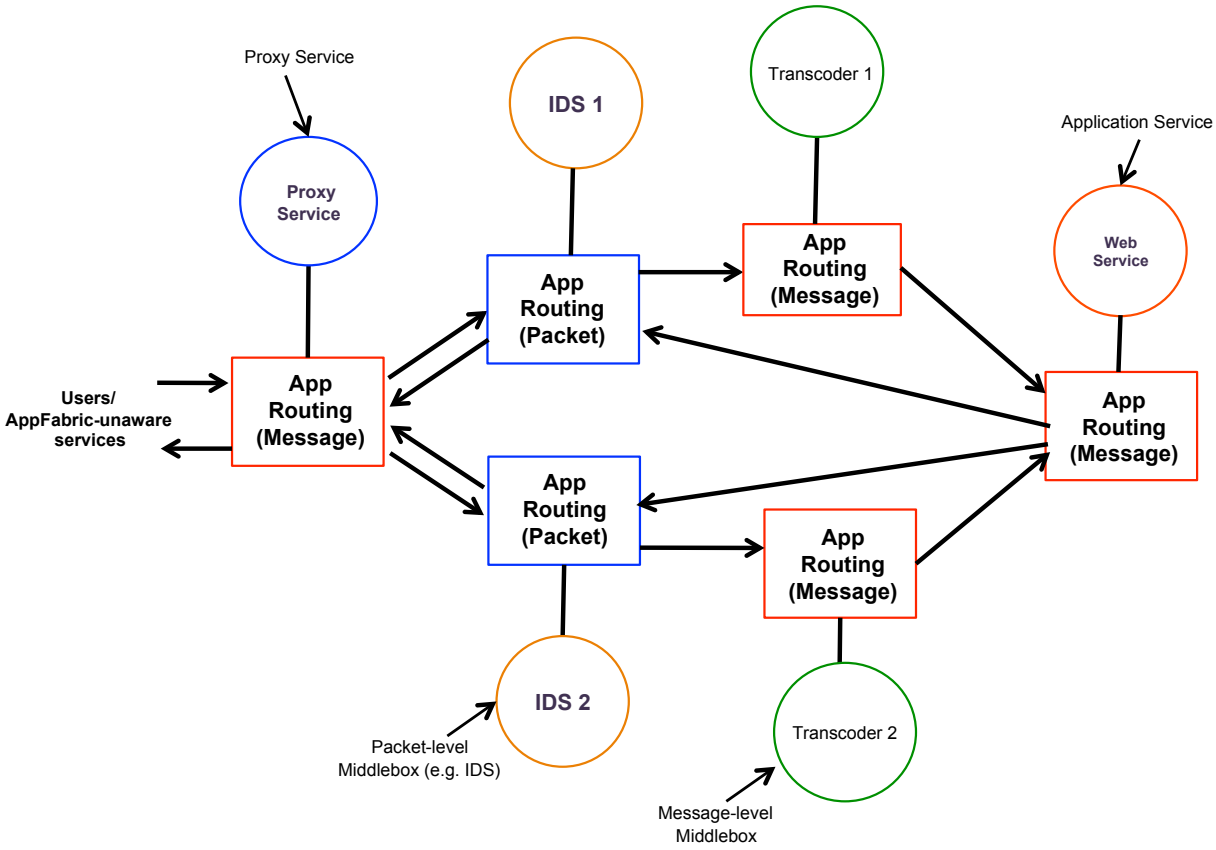

Figure 1.3: Schematic representation of an AppFabric Service Workflow

– **The AppFabric Application Cloud abstraction:** The AppFabric Application Cloud (AAC) abstraction allows the application administrator to specify application deployment and delivery policies for delivering the application over multi-datacenter environments. These policies might specify the rules for optimizing the application delivery over various cost and performance tradeoffs including distributing the application instances across different geographical regions and across multiple private/cloud dataceneters, optimizing for variable usage patterns (load, distribution and context of users) and handling infrastructure failures and planned maintenance situations. Fig. 1.4 shows a schematic representation of an AAC where the application cloud spawns several ASW instances that are distributed over resources leased from multiple Cloud and Internet Service providers.

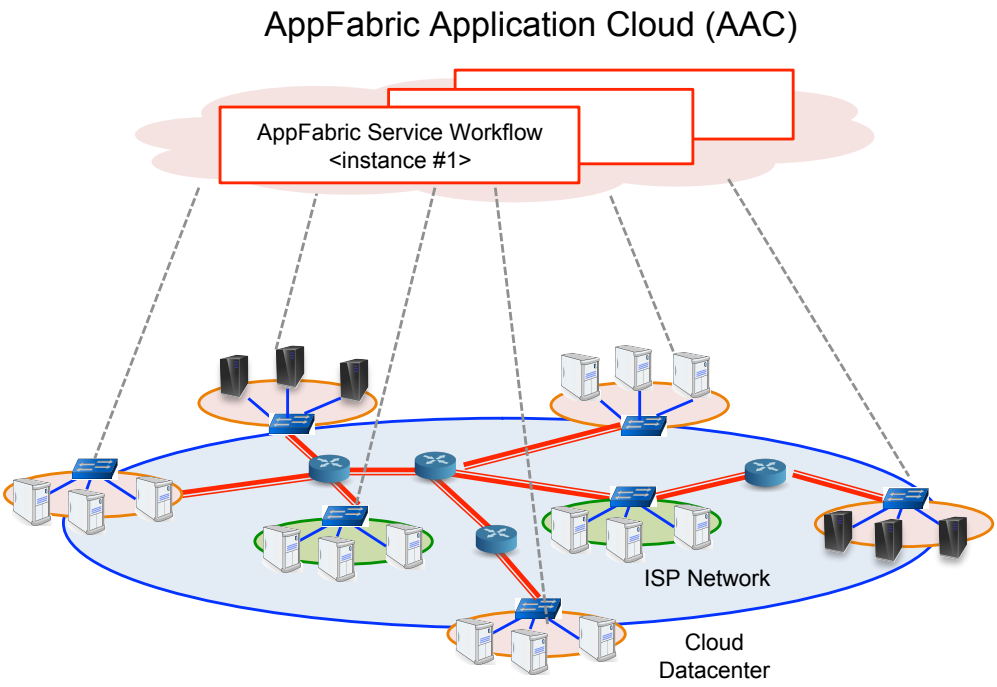

Figure 1.4: Schematic representation of an AppFabric Application Cloud

- **The southbound interface:** The southbound interface is defined between the AppFabric platform and SDI providers. It implements two types of interfaces:
  - **The AppFabric Southbound Management Interface:** Through the southbound management interface, AppFabric talks to different SDI platform management planes (e.g. EC2, OpenSTack, OpenDayLight, etc.) to lease/allocate resources and place them in the common virtual resource pool for instantiating the ADN over them.
  - **The AppFabric Southbound Control Interface:** Once the virtual resources are allocated to the virtual resource pool, they are programmed through the AppFabric Southbound Control interface with the application runtime deployment and delivery rules compiled from the ASP policies specified thorough the AppFabric northbound interface.

1.3 Contribution

This thesis makes several key contributions.

- **Definition and architectural framework for Application Delivery Network (ADN):** The term application delivery networking is often used by middlebox vendors such as F5[39], Citrix[25], etc. to refer to their consolidated middlebox platforms that provide essential security, performance and optimization services to application deployments. However, it is clearly not appropriate to call such platforms ADN since although these devices may be part of the network, neither do they create a network nor are they the only components required for delivering the application. We re-define the term ADN to refer to a network of all the different components including applications servers, storage servers, packet level middleboxes, message-level middleboxes, and network transport services that are required to deliver an application. Also, we provide a standard framework of creating and operating such ADNs over distributed resources that may be leased from third-party providers.

- **Automatically managed fabric-approach:** Although the concept of a network-fabric, referring to automatically managed distributed network infrastructure has been around for quite some time, we are probably the first to propose the idea of an application fabric. An application fabric is a single, flat, automatically managed distributed application delivery and deployment infrastructure as against the existing application delivery environments that are fractured across multiple administrative domains and device/service types that are managed either thorough ad-hoc manual techniques or through centralized (or logically centralized) orchestration services.

- **Separation of control and data planes in application delivery:** AppFabric is probably the only solution as of now to extend the idea of control and data plane separation proposed by the Software-Defined Networking (SDN) architecture beyond network infrastructures to ADNs comprising of both, network-layer service as well as application-layer services.

- **Hierarchical control plane architecture:** Unlike the centralized or logically centralized controller architecture proposed by most SDN architectures, AppFabric proposes a hierarchical control plane architecture to be able to effectively manage and

control massively distributed ADN deployments that may be distributed across an extended geographical region. The centralized controller approach is not appropriate for such use cases since there would be non-negligible communication latency between the data plane entities and the centralized controller.

- **Introducing a multi-datacenter application deployment platform:** While cloud computing has truly revolutionized the computing landscape, AppFabric will allow applications to truly leverage the opportunities provided by cloud computing. Many application use-cases that involve massively distributed geographical presence with dynamically changing application footprint such as Internet-of-Things/Cyber-Physical Systems, virtual worlds, online games and mobile apps are expected to benefit immensely from such a platform.

- **Introducing the concept of cloud networking:** The AppFabric architecture motivates the need for cloud networking; that is applying the concepts of cloud computing to networking, including virtualization and on-demand, dynamic provisioning of services through a standard service-oriented interface.

- **Prototype implementation:** We have implemented a prototype of the AppFabric that includes all the essential features discussed in this thesis. The prototype has been implemented in a mix of Python and C and contains 10,000 lines of code. This proof-of-concept implementation is quite extensive and helps validate the architectural claims that we make in this thesis and also provide the basic framework that can be easily extended to real, production-level implementation in the future.

1.4 Potential Impact

AppFabric is designed to provide a deployment and delivery platform for massively distributed and extremely dynamic application use-cases. In this section we will look at some application areas where such application use-cases are relevant and how AppFabric supports the deployment of these use-cases. It may be noted that as with any platform solution, all the application use-cases discussed in this section is possible without AppFabric. AppFabric just makes it easier by providing the required abstractions.

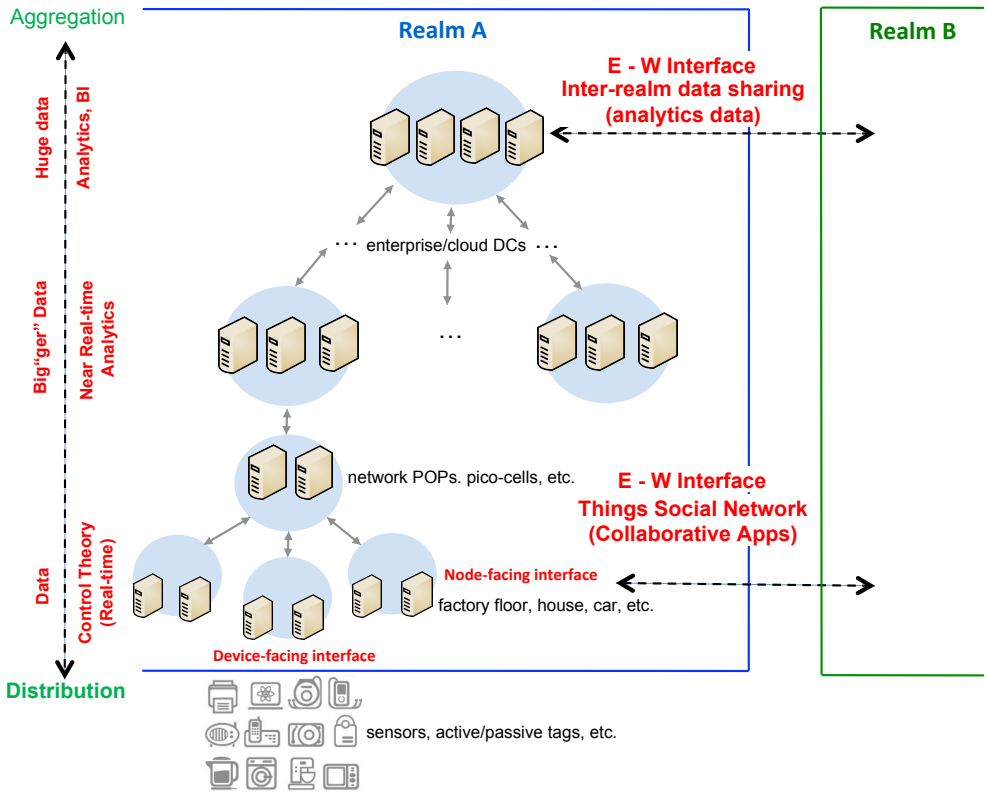

Figure 1.5: Distribute/Aggregate Topology for Iot/CPS use cases

- **Internet-of-Things (IoT) and Cyber-Physical Systems (CPS):** IoT and CPS use-cases inherently have a hierarchical distribute-aggregate topology (as shown in Fig. 1.5) with a large number of data collection points and a relatively fewer number of data aggregation points. Also, the requirements at each level in the hierarchy vary from latency-sensitive near real-time processing at the edge controllers to long-term big data analytics at the core. Managing such a massively distributed application topology with such a diversified set of service requirements is hard. This is especially true as we approach next generation IoT and CPS use-cases that will see a lot of integration across industry verticals. For example, in the future, vehicular networks will be connected to public safety systems, the car manufacturers system for tracking the proper functioning of the car, insurance companies for providing incentives for safe driving and traffic monitoring and mapping systems to control smooth distribution of traffic. The possibilities are limited only by ones imagination. AppFabric will provide a common

platform for deploying and managing these different application use-cases. AppFabric makes it possible to easily manage massively distributed application deployments, typical to these application use-cases. Also, AppFabric is designed to allow application deployments to leverage SDI to setup application-specific clouds. For example, assuming that resource providers make different classes of infrastructure services available through their SDI interfaces, AppFabric could select the right providers, at the right locations to get the right types of resources required to support the application. Finally, AppFabric allows the application cloud to be dynamic. Again this is typical of many IoT and CPS applications where the topology of the application deployment may need to change over time.

- **Mobile Apps:** Over the past few years, the popularity of smart phones and other smart mobile devices has led to an explosion in the number of diverse applications being created for these platforms. These mobile apps, as the name suggests, are mobile. Access to these applications are extremely context-aware (type and capability of the device and nature of the connectivity) and inherently distributed. Also, as users move, the application cloud needs to adapt dynamically to be able to serve its users efficiently and ensure that the quality of user experience. The mobile apps platform has opened up the doors for innovations where any individual who has an idea can easily create an app in a relatively small time and get into business. But, one of the limitations is that it is difficult for an individual to provide a high quality app that requires substantial back-end support due to the complexities and capital and operational expenditures of maintaining a distributed service back-end. Therefore, creating and maintaining such apps may in fact be more challenging than providing traditional online services. AppFabric will help lead this innovation forward by providing a platform that would easily allow the app provider to create and manage a dynamic application cloud that automatically scales to the number of users and more importantly distributes the application deployment on-demand. It would thus greatly reduce both, the CAPEX and OPEX of providing a quality app that in-turn further reduces the risk of investing in an idea.

- **Network Function Virtualization (NFV):** Telecom and network providers are trying to virtualize their infrastructures to make them ready to handle the next generation cloud-based application traffic. The primary goal is to virtualize network functions so

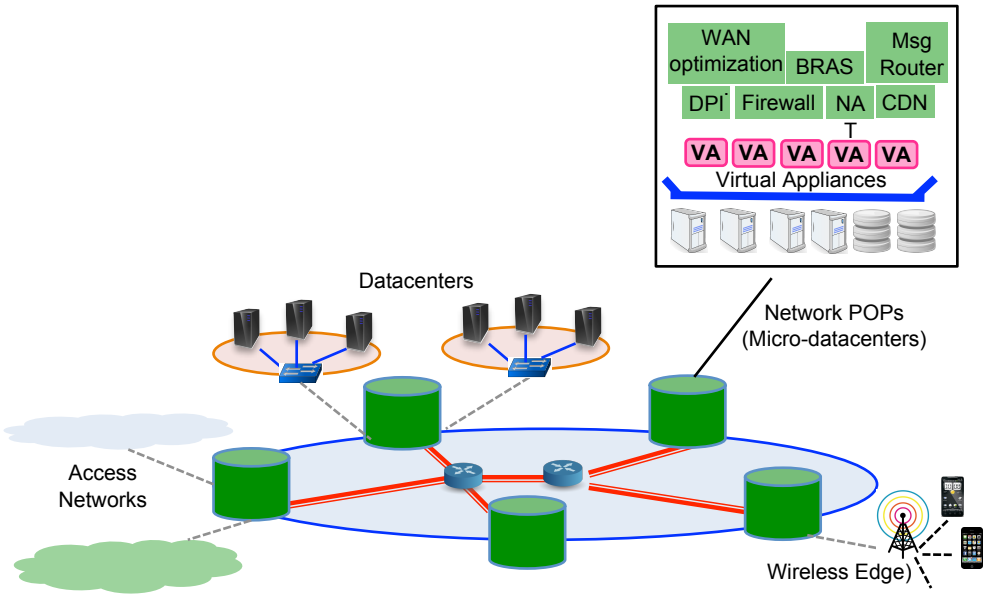

Figure 1.6: Network Function Virtualization (NFV) use case

that they can be dynamically deployed anywhere and anytime. This requires moving network services out of specialized hardware and deploying them over commodity hardware in micro-datacenters distributed across the network operators points-of-presence. Architecting POPs this way would be a huge revenue source for these operators and they can provide a lot of services much more easily, cheaply and dynamically to distributed application deployments. AppFabric provides support for NFV in two ways. First, the deployment of virtualized network services across multiple POPs dynamically is itself another instance of massively distributed and dynamic application deployment context that AppFabric is designed to address. Second, to an AppFabric-based application deployment, NFV services are just another type of SDI provided by a network provider. As we will see in our discussion, the AppFabric data plane will allow third-party network services to be easily inserted into an applications service composition context.

- **Online Gaming, Virtual Worlds:** Online gaming and virtual world (for example, virtual office, next generation virtual meeting rooms, etc.) applications have been around for quite some time but havent really been deliver to their full potential. These applications have scaling and latency requirements that can only be solved by having

a massively distributed compute infrastructure. Another requirement is that the application should be able to change its topology dynamically based on usage patterns; similar to mobile apps. Also, for many such applications, communication between the distributed servers may critical and also sensitive to latencies. These are exactly the issues that AppFabric is designed to address.

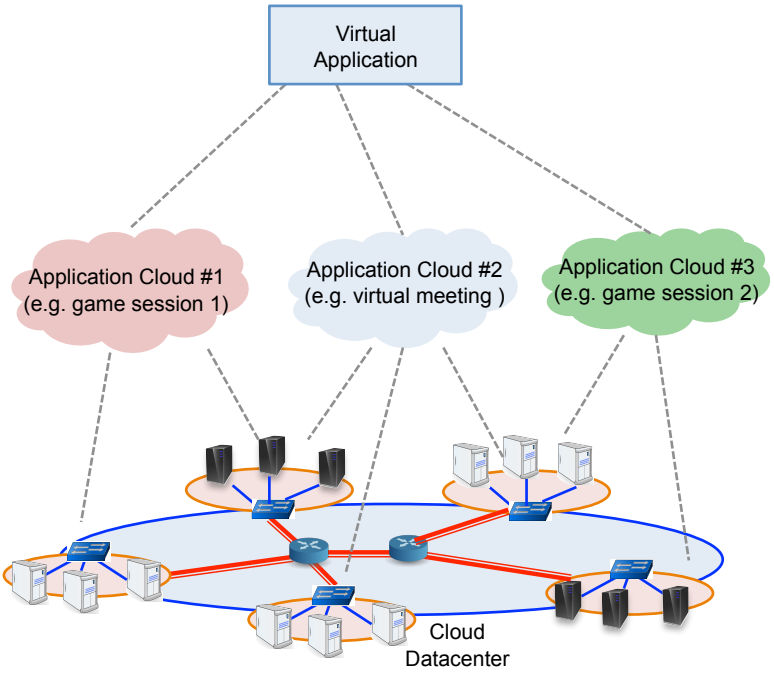

Figure 1.7: Virtual Worlds use case

1.5 Organization

The organization of this thesis is as follows. Chapter 2 provides additional background and related work. In chapter 3, we present the high-level architecture of AppFabric, detailing the requirements of such a platform and motivating the current design choices. Chapter 4 presents the details of the *OpenADN* (Open Application Delivery Network) design, which is the data plane of AppFabric. In Chapter 5 we present the AppFabric control and management plane called *Lighthouse*. Chapter 6 details the proof-of-concept implementation

of AppFabric. Finally, we summarize our findings in chapter 7 and point to a number of interesting avenues for future work.

Chapter 2

Background

In this chapter we will give some background on the various concepts that are directly or indirectly related to the discussion in this thesis. AppFabric is a horizontal platform that integrates several technology verticals.Therefore, the discussion in this chapter will touch upon many different and seemingly unrelated areas. The high-level idea that connects all these different areas is that they all contribute, either directly or indirectly, to advancing the frontier of deploying and delivering large-scale, massively distributed and extremely efficient applications in the future. Keeping this high-level idea in mind will allow the reader to better understand the discussion in this chapter while at the same time appreciate the relevance, challenges and enormity of the problem that we have set ourselves to address.

2.1 What does Software-Defined mean?

Currently, there seems to be a huge hype around the phrase **software-defined** and as typical of all technologies in the hype phase, it is presented as the elixir to all system design problems. All product vendors are touting their solutions to be software-defined-*, creating confusion as to what these terms really mean. In order to be able to cut through the hype and get to the real applicability of any technology it is important to have a more objective and practical perspective. One of the ways to do it is to study the historical perspective of its origin and try to objectively see how it is different from past efforts and what is unique in the current approach that will make it better. Therefore, before delving into any specific discussion on proposed frameworks and use-cases, it would be helpful to try and place software-defined-* technologies in their right perspective.

The term software-defined was coined in the context of **software-defined networks (SDN)**. SDN is a new approach to manage and operate computer networks. Computer networks are typical examples of large-scale distributed systems and hence extremely hard to design, deploy and operate. This is because it is often tricky to ensure system stability, robustness, correctness and optimality when dealing with distributed state management (synchronization, update delays, etc.) and designing distributed algorithms under partial system-level visibility. Also, it is extremely difficult to locate and debug problems arising from node failures, configuration errors and protocol design issues in such systems. SDN tries to address these issues by proposing a new architectural design based on the separation of the control and data planes. Currently, each network node runs an instance of the distributed control plane algorithm that locally computes the data plane forwarding policies for that node. In SDN, the distributed control plane is replaced by a centralized control plane; with all control plane functions placed in a centralized node called the controller. The controller is responsible for computing per-node data plane policies and pushing these policies to the data plane nodes. The **OpenFlow** protocol provides the standardized southbound APIs through which the central controller can speak to the multi-vendor data plane nodes. Both, the advantages and disadvantages of this architectural design choice are obvious. It is much easier to design control plane algorithms with full system visibility. Also, since the controller is not in the data plane, its being centralized does not directly affect the performance, availability and scalability of the system. However, controller failures may still severely impact the system, especially if the controller remains unavailable for long durations, gets compromised or looses sensitive state information. These are important open problems for the SDN research to address.

Therefore, SDN greatly simplifies network management and control. However, it is not the first of its kind. There have been similar proposals before such as centralized route controllers [15] that compute the forwarding tables for each of the distributed switches and routers in the network and centralized Path Computation Elements (PCE) [93, 9, 74] for traffic engineering in MPLS networks. Sure enough, SDN is more generic (in terms of allowing many other types of polices and at a finer level of granularity) than these previous proposals, but this is not the only reason for its unprecedented popularity. Most of SDNs popularity is owing to its (sort of) re-purposed role as the enabler of **network virtualization**.

Virtualization is another term that has become extremely popular lately in the context of cloud computing, or more specifically in the context of the ***-as-a-service** paradigm. However, the concept of virtualization is not new to computing. Loosely defined, virtualization is the technique of creating an emulated hardware or software resource(s) that may have very different properties (in terms of size, complexity, functionality, representation, etc.) than the actual underlying hardware or software resource(s) from which it is created. For example, one of the earliest examples of virtualization is the concept of virtual memory. Virtual memory presents applications with a much larger memory resource than the actual available physical memory by emulating a non-persistent memory device over persistent storage. Another example is the concept of virtual LANs that emulate many logically separate layer 2 LAN segments sharing the same physical layer 2 network. Virtualization is implemented through an abstraction layer, also called the virtualization layer that sits between the actual resource(s) and the emulated resource(s). Examples of this virtualization layer include the hypervisor that creates one or more virtual machines over a single physical machine, the RAID driver that presents an array of redundant storage disks as a single large storage device, the memory management module within an OS implementing the virtual memory abstraction, the VLAN tag header in network packets and the software in switches that can interpret these tags, and the load balancer proxy that sits in front of a replicated software service but exposes a single virtual IP (VIP) location for accessing the service.

Coming back to our discussion on SDN, although it was originally designed to ease the pains of managing and controlling large-scale, distributed physical network infrastructures, it was soon realized that it was very difficult to transform old and legacy networks including enterprise, telecom and ISP networks. These networks have billions of dollars invested in hardware-based networking equipment and therefore they are very resistant to any kind of disruptive new technology. Clearly, SDN with its requirement of programmable data plane switches and routers was not going to be easily adopted in these use-cases. Instead, a more plausible use-case was data center networks. Modern datacenters with thousands of computing and storage devices required a well managed and easy to control network infrastructure. SDN fitted the bill perfectly. This was validated by Googles announcement that it used OpenFlow and SDN -like technologies to operate its intra-datacenter and inter-datacenter networks. These conditions helped in starting the hype around SDN. However, this hype is not all empty. SDN does have the potential to considerably reduce the CAPEX (by making the switching hardware commoditized) and OPEX (by making operations much

more easier and automated) of network infrastructures significantly and therefore has genuine value as a technology. But, the real value of SDN is realized in the context of virtualized datacenter architectures.

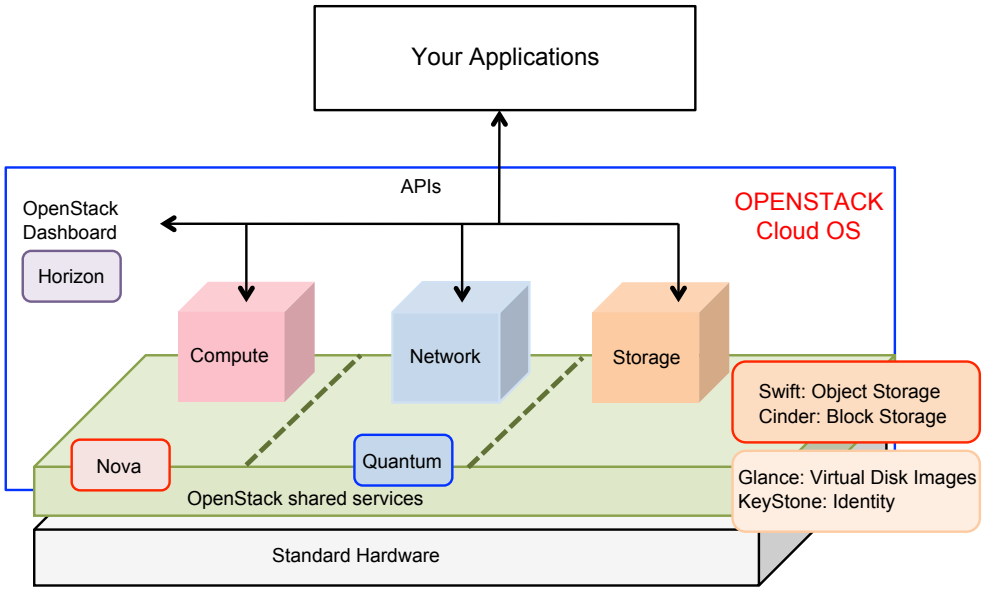

Figure 2.1: Schematic Diagram of the OpenStack Architecture

Today, majority of the data centers, both cloud and on-premise, want to virtualize their infrastructures.Virtualization allows them to improve resource utilization and the ability to dynamically generate application-specific infrastructures for diverse application workloads. In a virtualized data center all resources are virtualized including computing, network, storage, software services and software platforms. However, it is extremely hard to manage this large and distributed virtualized resource pool and be able to create and maintain higher-level resource abstractions over them. But, this is the same problem that SDN was designed to address in the context of distributed, physical networking resources. Therefore, the same idea of a software-based centralized controller could be re-purposed for other types of resources as well. This led to the popularity of the phrase software-defined that was borrowed from SDN to refer to this generic architectural notion of software-based, logically-centralized control and management. Fig. 2.1 shows a schematic representation of the OpenStack architecture. OpenStack is one of the most popular, open source datacenter management software stack among others including Amazons EC2 (proprietary), CloudStack, Eucalyptus, etc. As shown in Fig. 2.1, a virtualized datacenter is enabled by a three-layer virtualization stack.

At the bottom, layer 1, is the basic resource virtualization layer that creates a virtual or emulated resource such as a virtual machine, virtual NIC, virtual disk, and virtual service, over the actual physical resource. Layer 2 is the virtual resource management and control layer that is responsible for efficiently consolidating this virtual resource pool and creating higher-level resource abstractions such as a virtual compute cluster, virtual three-tier web infrastructure, virtual composed service or software platform etc. Layer 2 has standard southbound APIs to talk to layer 1 and standard northbound APIs to talk to layer 3. The topmost layer or layer 3 is responsible for exposing these virtual resources to applications through a set of standard APIs. Application deployments can add/remove/modify (state)/update (state) resources to their deployment contexts through these APIs. Mostly these APIs are designed to allow multi-tenancy (by including the identity of the tenant/user in the API call) such that each tenant/user may operate their own, isolated virtual infrastructure context that share the same underlying physical infrastructure. Therefore, layer 3 also needs to implement API management functions including verifying the credentials of the tenant/user, monitoring usage/rate limiting, and handling billing and SLA. It must be noted that support for multi-tenancy needs to be built into the resource management layer (Layer 2) as well in order to ensure performance and security isolation across tenants sharing the same physical resources, optimizing resource allocation on per-tenant basis (e.g. implementing rack affinity) and enabling differentiated services. This three-layer virtualization stack implements the **software-defined datacenter** where the infrastructure is virtualized and the control and management of the datacenter is fully automated. This is in contrast to traditional datacenters that are mostly manually managed physical infrastructures.

Within this context, the key question now is how to leverage the flexibility and dynamism of software-defined infrastructures to improve application deployment and delivery. There are two ways of looking at this issue. The first is from the perspective of infrastructure design- to determine what generic abstractions can a software-defined infrastructure provide that will help applications benefit from it without having to make significant changes to the application design itself. The second is from the perspective of the application design to determine how application design can change to benefit from this new software defined paradigm in general.

2.2 Software-Defined Networking (SDN)

In the last section, we touched upon the term Software-Defined Networking or SDN but did not discuss it in much detail. In this section, we will discuss SDN in more detail and try to give an overview of the key ideas underlying this architectural concept.

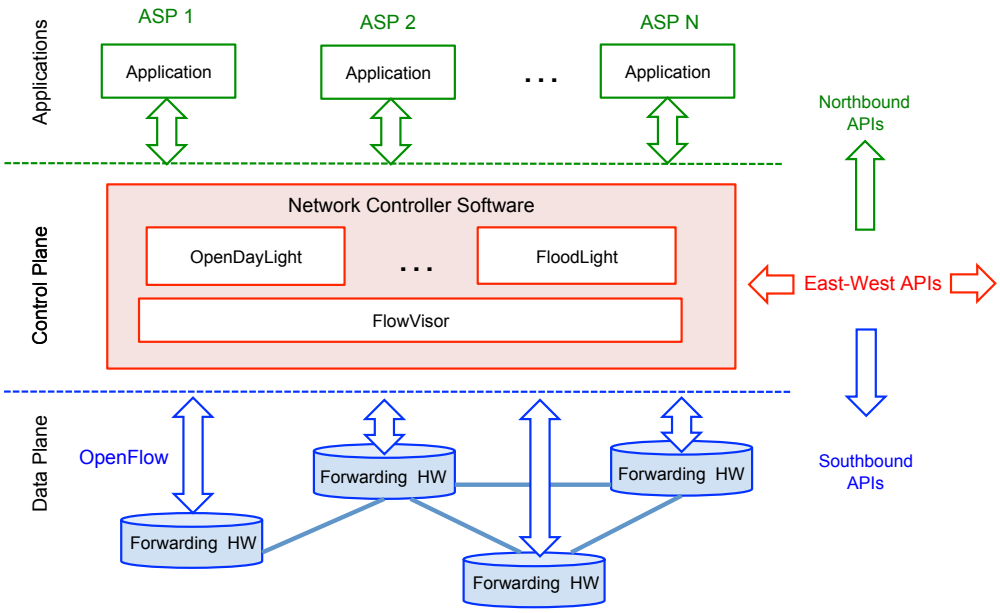

Figure 2.2: Software Defined Networking (SDN) Architecture

The SDN architecture comprises of four key ideas:

- **Separation of control and data planes:** Networking protocols are often arranged in three planes: data, control, and management. The data plane consists of all the messages that are generated by the users. To transport these messages, the network needs to do some housekeeping work, such as finding the shortest path using L3 routing protocols such as Open Shortest Path First (OSPF) [59] or L2 forwarding protocols such as the Spanning Tree Protocol [110]. The messages used for this purpose are called control messages and are essential for network operation. In addition, the network manager may want to keep track of traffic statistics and the state of the various networking equipments. This is done via network management. Management, although

important, is different from control in that it is optional and is often not done for small networks such as home networks. One of the key innovations of SDN is that the control should be separated from the data plane. The data plane consists of forwarding the packets using the forwarding tables prepared by the control plane. The control logic is separated and implemented in a controller that prepares the forwarding table. The switches implement data plane (forwarding) logic that is greatly simplified. This reduces the complexity and cost of the switches significantly.

- **Centralization of the control plane:** The U.S. Department of Defense funded Advanced Research Project Agency Network(ARPAnet) research in the early 1960s to counter the threat that the entire nationwide communication system could be disrupted if the telecommunication centers, which were highly centralized and owned by a single company at that time, were to be attacked. ARPAnet researchers therefore came up with a totally distributed architecture in which the communication continues and packets find the path (if one exists) even if many of the routers become non-operational. Both the data and control planes were totally distributed. For example, each router participates in helping prepare the routing tables. Routers exchange reachability information with their neighbors and neighbors neighbors, and so on. This distributed control paradigm was one of the pillars of the Internet design and unquestionable up until a few years ago. Centralization, which was considered a bad thing until a few years ago, is now considered good, and for good reason. Most organizations and teams are run using centralized control. If an employee falls sick, he/she simply calls the boss, and the boss makes arrangements for the work to continue in his/her absence. Now consider what would happen in an organization that is totally distributed. The sick employee, say John, will have to call all his co-employees and tell them that he/she is sick. They will tell other employees that John is sick. This will take quite a bit of time before everyone will know about Johns sickness, and then everyone will decide what, if anything, to do to alleviate the problem until John recovers. This is quite inefficient, but is how current Internet control protocols work. Centralization of control makes sensing the state and adjusting the control dynamically based on state changes much faster than with distributed protocols. Of course, centralization has scaling issues but so do distributed methods. For both cases, we need to divide the network into subsets or areas that are small enough to have a common control strategy. A clear advantage of centralized control is that the state changes or policy changes propagate much faster

than in a totally distributed system. Also, standby controllers can be used to take over in case of failures of the main controller. Note that the data plane is still fully distributed.

- **Programmable control plane:** Now that the control plane is centralized in a central controller, it is easy for the network manager to implement control changes by simply changing the control program. In effect, with a suitable API, one can implement a variety of policies and change them dynamically as the system states or needs change. This programmable control plane is the most important aspect of the SDN. A programmable control plane in effect allows the network to be divided into several virtual networks that have very different policies and yet reside on a shared hardware infrastructure. Dynamically changing the policy would be very difficult and slow with a totally distributed control plane.

- **Standardized API's:** As shown in Fig. 2.2, SDN consists of a centralized control plane with a southbound API for communication with the hardware infrastructure and a northbound API for communication with the network applications. The control plane can be further subdivided into a hypervisor layer and a control system layer. A number of controllers are already available. Floodlight [43] is one example. OpenDaylight [112] is a multi-company effort to develop an open source controller. A networking hypervisor called FlowVisor [122] that acts as a transparent proxy between forwarding hardware and multiple controllers is also available. The main southbound API is OpenFlow [78, 46] , which is being standardized by the Open Networking Foundation. A number of proprietary southbound APIs also exist, such as OnePK [24] from Cisco. These later ones are especially suitable for legacy equipment from respective vendors. Some argue that a number of previously existing control and management protocols, such as Extensible Messaging and Presence Protocol (XMPP) [118, 119], Interface to the Routing System (I2RS [57]), Software Driven Networking Protocol (SDNP)[63], Active Virtual Network Management Protocol (AVNP) [14], Simple Network Management Protocol (SNMP) [17], Network Configuration (Net-Conf) [36], Forwarding and Control Element Separation (ForCES) [142, 34, 54], Path Computation Element (PCE) [93, 9, 74], and Content Delivery Network Interconnection (CDNI) [51], are also potential southbound APIs. However, given that each of these was developed for another specific application, they have limited applicability as a general-purpose southbound control

API. Northbound APIs have not been standardized yet. Each controller may have a different programming interface. Until this API is standardized, development of network applications for SDN will be limited. There is also a need for an east-west API that will allow different controllers from neighboring domains or in the same domain to communicate with each other.

Networking industry has shown enormous interest in SDN. SDN is expected to make the networks programmable and easily partitionable and virtualizable. These features are required for cloud computing where the network infrastructure is shared by a number of competing entities. Also, given simplified data plane, the forwarding elements are expected to be very cheap standard hardware. Thus, SDN is expected to reduce both capital expenditure and operational expenditure for service providers, cloud service providers, and enterprise data centers that use lots of switches and routers. SDN is like a tsunami that is taking over other parts of the computing industry as well. More and more devices are following the software defined path with most of the logic implemented in software over standard processors. Thus, today we have software defined base stations, software defined optical switches, software defined routers, and so on. Regardless of what happens to current approaches to SDN, it is certain that the networks of tomorrow will be more programmable than today. Programmability will become a common feature of all networking hardware so that a large number of devices can be programmed (a.k.a orchestrated) simultaneously. The exact APIs that will become common will be decided by transition strategies since billions of legacy networking devices will need to be included in any orchestration.

2.3 Application-Delivery Networking (ADN)

The term Application Delivery Networking or ADN is often used by middlebox appliance vendors such as F5 networks [39], Citrix [25], Redback [115], Brocade [13], Layer 7 Technologies [73], Array Networks [8], and several other companies to market their products. In fact, the ADN business is a burgeoning multi-billion dollar industry. Market reports project that the size of the middlebox or network appliances market will grow from 1.5 billion dollars in 2009 to 2.25 billion by 2013 [131]. This projection is only for acceleration and optimization middleboxes and does not include security appliances, which is projected to grow to a 10

billion dollar market itself by 2016 from 6 billion in 2010 [120]. Fig. 2.3 shows a block-level representation of a device often sold under the name of an Application Delivery Controller. As can be seen in this figure, rather than being a stand-alone middlebox serving a single specific function, these new breed of devices consolidate many different services and are hence marketed under the name of *Application Delivery Networking Platforms.* However, we think that the term ADN is mis-used in this context; it describes a *device* rather than a *network*. There is no doubt that these devices are essential components of todays application delivery and deployment environments, but surely the term ADN should have a more generic definition.

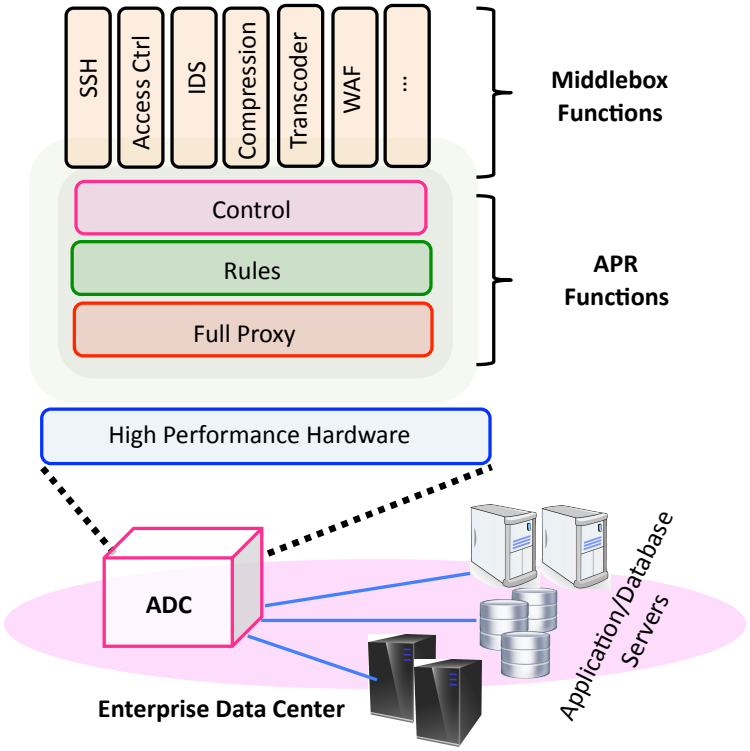

Figure 2.3: Schematic Representation of an Application Delivery Controller

In this thesis, the term ADN will refer to a distributed network architecture comprising of many different components including; 1) Application servers, 2) Packet-level middleboxes, 3) Message-level Middleboxes, and 4) Network transport services, that are required to deploy

and deliver modern applications. While we will see in the rest of this thesis how AppFabric provides a consolidated platform for easily creating, managing and controlling such ADNs, in this section we will look at some of the components that make up these ADN infrastructures.

2.3.1 Virtual compute, network and storage infrastructures

In our discussion in the last section on the meaning of the term **software-defined**, we suggested that *virtualization* was the key mechanism underlying modern flexible and dynamic software-defined infrastructures. In this section we will go into a little more detail into the server, storage, and network virtualization technologies to give a little more idea of what it takes to virtualize the infrastructure.

- **Server virtualization:** The concept of server virtualization has been around for the last 40 years or so. For a detailed survey on server virtualization please refer to [135, 23] . Here we provide a short wikipedia-ish [140] level discussion to provide a quick summary of some of the key aspects.
  There are several reasons why one would want to virtualize the server. Some of them are:
  - **Isolation:** Most modern operating systems provide a *process* abstraction for resource sharing among multiple (pseudo) concurrently running applications. However, the process abstraction enforces only a weak notion of security and performance isolation. For example, a rogue application may easily jeopardize other applications running on the same system. Virtualization, on the other hand enforces a much stronger notion of isolation where each virtual machine may have exclusive access to a set of resources.
  - **Better utilization of physical compute resources:** Sometimes, application services may be distributed across many different servers such that the average utilization per server is extremely low. In such cases, virtualization allows server consolidation by creating many virtual machines over the same physical machine and assigning a virtual machine to each application. Therefore, instead of running on a separate physical machine, the applications run on separate virtual machines consolidated over a few physical hosts. The resources allocated to the virtual

machines is just enough to run the application. This way, most of the underutilized server infrastructure can be freed up to do other useful work, or shut down to save energy costs.

- **Separate execution environments:** Using virtualization, the same physical server can run different operating systems on different virtual machines. This may be useful in scenarios where a software developed for multiple platforms need to be tested and debugged. Also, another scenario may be when the user needs to run a software created for only one type of platform, or for an older version of the operating system. In such situations, a separate VM running the required OS platform may be useful.
- **Dynamics and ease of management:** Managing virtual resources is often easier than managing actual physical resources because of the management interfaces provided by software layer providing the virtualization. As a result, virtual resources may be allocated/de-allocated dynamically and virtual machines may be moved from one physical host to another as required.

These were some of the key benefits of server virtualization. Now let us look at some of the mechanisms through which server virtualization may be implemented. There are three main techniques:

- **Full-virtualization:** As shown in Fig. 2.4, in full-virtualization, the Virtual Machine Monitor (VMM) emulates the physical hardware and provides a mapping between the virtual resources and the physical resources. The user can create one or more virtual machines on top of the VMM. This technique is called full virtualization because the virtual machine is fully abstracted from the underlying physical machine and hence the guest OS (the OS on the VM) does not need to be changed in any way. In fact it does not even need to be aware that it is running in a virtualized environment. Note that the VMM is running in Level 0 (also called Ring 0) which is the most privileged hardware mode in which generally the OS runs. The guest OS runs in Level 1(Ring 1). With multiple guest OS' running, they may all make privileged instruction calls to the hardware layer simultaneously. The VMM intercepts these calls and either executes it on the processor or emulates the response. Full virtualization provides the best isolation in terms of security. In terms of performance, indirection through an additional

software layer (the VMM) has some performance penalties compared to running directly on bare-metal. Example of full virtualization products include VMWare ESx [137] and Microsoft Virtual PC.

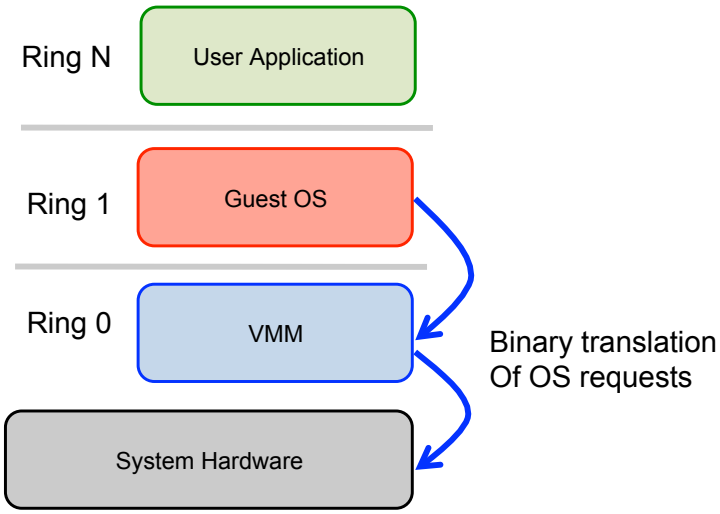

Figure 2.4: Full Virtualization

– **Para-virtualization:** As shown in Fig. 2.5, in para-virtualization, the virtualization layer (hypervisor) does not fully emulate the underlying hardware layer and instead provides a software interface (through hypercalls) to the guest OS. Therefore, the guest OS needs to be changed to be able to communicate with the hypervisor instead of communicating directly with the hardware layer. The rationale for para-virtualization was to improve the efficiency and performance of the communication between the guest OS and the hypervisor as compared to full virtualization in which the guest OS is unaware of the presence of the hypervisor (VMM) and hence cannot optimize to avoid the performance penalties as a result of this added layer of indirection. The VMM has to silently interpose and perform binary conversion on all guest OS calls. The most famous example of para-virtualization is the Xen [10] open source project.

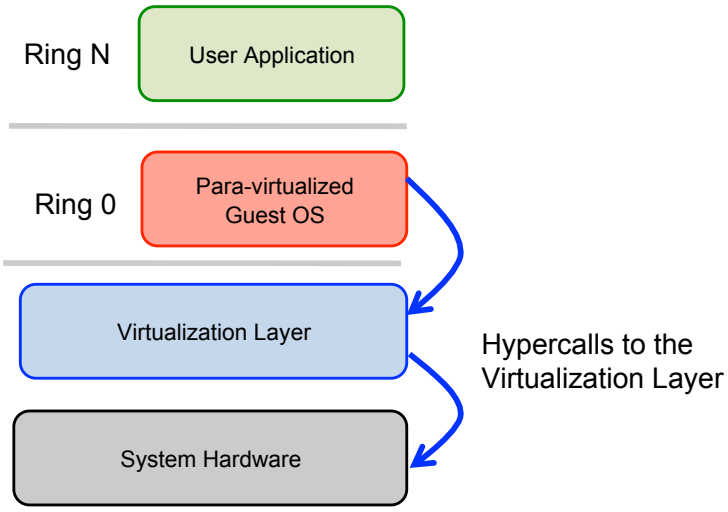

Figure 2.5: Para-Virtualization

– **Hardware-assisted virtualization:** While the two techniques discussed above are completely software-based solutions, hardware vendors have over time added support to assist virtualization. Example of such platform include Intel-VT [60] and AMD-V [5]. It provides special privileged instructions and a new CPU execution mode. As shown in Fig. 2.6The hypervisor can run in this new execution mode which has a higher privilege than the Level 0, also called Ring 0, in which the operating system kernel runs. Without this special hardware support, the hypervisor has to run in Level 0 (or Ring 0) while the guest OS has to run on Level 1 (or Ring 1). This means that privileged calls by the guest OS would be able to trap the hypervisor only through binary translation (as in the case of full-virtualization) or para-virtualization (specially designed hypercalls). Now with special hardware support the guest OS can make privileged calls by automatically trapping the hypervisor.

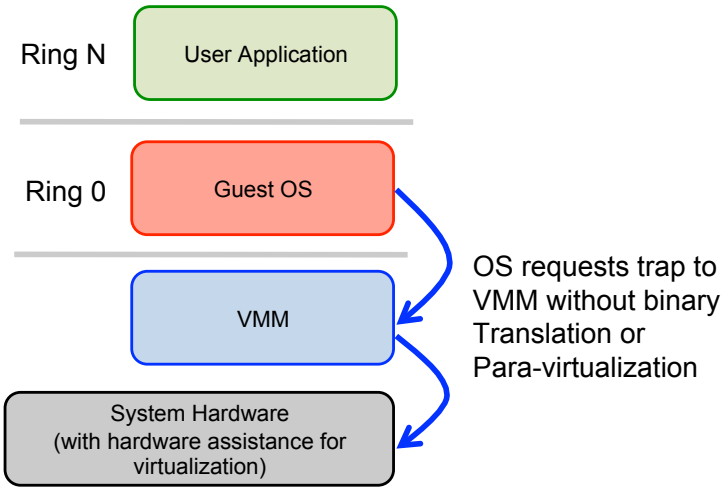

Figure 2.6: Hardware-assisted Virtualization

Apart from these three main techniques for virtualization at the Hardware Abstraction Layer (HAL); that is, at the junction of the hardware and software layers of a server, other forms of virtualization may also be useful. One of the oldest attempts to virtualize was idea of FreeBSD Jails[66]. The idea was to create several secure and isolated operating environments (Jail) within the same OS instance and run applications within these *jailed* environments. Another example of virtualization is User-Mode Linux (UML) [33] that runs the Linux kernel within an user mode application process and thus provides a lightweight virtualization platform. Cooperative Linux [4] is another example of UML-like virtualization. Virtualization is also a commonly used concept nowadays in programming language runtime environments to achieve interoperability of the program across platforms. The most common example is the Java Virtual Machine (JVM) abstraction that runs Java bytecodes. Therefore, although different flavors of virtualization exist, we are interested mostly in the three key virtualization techniques at the HAL because they have a direct bearing to cloud computing; and more specifically to Infrastructure-as-a-Service (IaaS) [7] offerings.

- **Storage virtualization:** In modern datacenters, storage and compute are separated to prevent fate-sharing during failures; that is if a server fails the data should still be available through other servers. Storage virtualization, just like server virtualization abstracts the diversified and distributed storage infrastructure and presents a common

interface for easily storing and retrieving data to the compute tier (application-tier). The key advantages of storage virtualization include:

- **Increased disk utilization:** Similar to server virtualization, storage virtualization too results in higher disk utilization owing to the consolidation techniques applied while storing the data in the actual physical storage from the separate virtual storage disks presented to the application tier.
- **Resilience and robustness:** The storage tier can use different techniques, most importantly redundancy, to make several copies of the data to protect it against failures. To the compute tier, the storage tier can expose different levels of resilience against data loss as a tunable parameter while internally it implements this by replicating the data on multiple physical disks that may further be spatially distributed to protect against physical damage as a result of some natural or unnatural disasters.
- **Mask diversity:** The storage virtualization layer masks different device types and access technologies (e.g. Ethernet, Fibre Channel, etc.) to provide a standard interface to the application tier. Also, it can create virtual storage devices of the desired size by partitioning or aggregating (e.g. RAID [102]) the underlying physical storage. Thus, the compute tier can dynamically request and get any sized storage that they may want.
- **Optimized access and storage from many distributed locations:** Modern large-scale distributed storage systems such as Google File Systems (GFS) [47], BigTable [18], Cassandra [71], HDFS[12], Dynamo[32], etc. allow the application layer to store and retrieve data from many distributed locations very efficiently. The storage system takes care of keeping the data synchronized across the many distributed locations.

The two key aspects of implementing a storage virtualization solution include the following:

- **Where to place the virtualization engine?:** Depending on where the virtualization engine is placed, the solution could be different. Some of the choices are:

* **Host-based virtualization:** In host-based virtualization, the virtualization software resides on the server (possibly as a software-based virtual appliance). This virtual appliance manages a pool of storage drives (possibly different devices from different vendors) and presents it to the application as a single storage pool. The storage drives in the backend may be organized as a RAID system. RAID stands for Redundant Array of Inexpensive Disks. It was designed to tradeoff between the high cost of reliable storage and the high failure rates of inexpensive storage. It is based on the idea of using array of many inexpensive discs and maintain many redundant copies of the data to avoid data loss as a result of individual disk failures. Host-based storage virtualization is suitable for small environments since it is per-host and each host needs to be separately configured.
* **In-band network appliances:** In this solution, a special storage management appliance sits between the hosts and the storage drives and creates a common storage pool for all the hosts to use. It is called in-band since it intercepts all forward and reverse traffic between the hosts and the storage devices. As is obvious, this may not be desirable since the appliance could itself be a performance bottleneck and also a single source of failure.
* **Out-of-band network appliances:** This is similar to an in-band appliance except that it does not directly intercept data plane traffic between the host and the controllers. Rather, it intercepts control plane messages where the host queries the appliance for information on how and what storage system it needs to access. While it solves the single-point-of-failure and performance bottleneck problems of in-band appliances, it requires that each host be configured separately to first query the appliance and then access the storage.

– **NAS and SAN:** NAS [20, 48] stands for Network-attached Storage while SAN [130] stands for Storage-Area Networks. NAS and SAN allowed the decoupling of the storage and the compute infrastructure by allowing the storage to be connected to the compute through the network. This has the advantage of decoupling the fate of the two infrastructures, as already discussed. Also, this allowed the creation of a central storage pool that could be shared by many servers; thus improving the utilization. The key difference between SAN and NAS is that in SAN the storage servers are connected using some special purpose storage network (e.g.

Fibre Channel) while in NAS the storage servers are connected using a general purpose network technology such as Ethernet.

Current storage virtualization solutions use variants of these techniques and/or combine them to address all the design goals. For more details on storage virtualization, please refer to [126].

- **Network virtualization:** Similar to server and storage virtualization, the concept of network virtualization has been around for quite some time. The initial motivation to virtualize was policy isolation. The idea of virtual LANs or VLANs [1, 2]was conceived to logically partition a physical LAN infrastructure into several logically isolated virtual LANs. This allowed the physical infrastructure to be shared among many different policy groups that required that their traffic be isolated from the others. For example, the finance department's traffic in a corporation may need to be kept separate from the marketing department owing to security concerns. Similarly, the idea of Virtual Private Networks (VPN) was conceived to allow a remote user access to a resource within the enterprise' secure network environment. The enterprise may setup policies to prevent any entity outside the enterprise' secure network from accessing the resource. Using a VPN, a remote user can connect to the resource by behaving as if it is part of the same enterprise environment although it may actually be accessing the resource over the Internet. This is done by creating a secure tunnel between the user and the enterprise VPN server that authenticates and authorizes the user. The VPN therefore creates a virtual periphery of the enterprise' secure network to allow remote sites and users that are not directly (physically) connected to the network to be still be able to access resource/information in the local network infrastructures.

  Over the past few years, there has been a renewed interest in network virtualization technologies owing to the rise of cloud computing. Cloud computing allows multiple tenants to share the computing and storage infrastructure in a datacenter. Obviously, they need to share the network as well. However, sharing the network infrastructure without proper policy and performance isolation amongst the different tenants (which could very well be competing commercial organizations) may lead to problems. Also, another reason to virtualize emerges from hybrid public-private application deployment architectures. The enterprises need to be able to seamlessly offload some of their

workloads to public cloud datacenters when their private datacenters are overloaded. This may be done either through live migration of VMs [136, 27, 70] from the enterprise datacenter to the cloud datacenter or by instantiating new service instances in the cloud. In such scenarios, the enterprise secure network boundary may also need to be extended to connect the instances hosted in the cloud. To support such dynamic expansion of the network across many different administrative domains, on-demand, the network infrastructure needs to be virtualized. Apart from these, another very clear motivation for network virtualization is that it allows creating isolated network contexts over the same physical infrastructure that can be tuned for application specific requirements, thus moving away from the traditional *one suit fits all* model of networking architectures.

While we have been using the term *network virtualization* quite freely without qualifying it further, it is actually an umbrella term that may refer to many different virtualization technologies. A network may be virtualized at different levels. Here we will look at some of the techniques of network virtualization at different levels.

– **Virtualization of Network Interface Cards (NICs):** Each computer system needs at least one L2 NIC (Ethernet card) for communication. Therefore, each physical system has at least one physical NIC. However, if we run multiple VMs on the system, each VM needs its own virtual NIC. As shown in Fig. 2.7 , one way to solve this problem is for the hypervisor software that provides processor virtualization also implements as many virtual NICs (vNICs) as there are VMs. These vNICs are interconnected via a virtual switch (vSwitch) which is connected to the physical NIC (pNIC). Multiple pNICs are connected to a physical switch (pSwitch). We use this notation of using p-prefix for physical and v-prefix for virtual objects. the figures, virtual objects are shown by dotted lines, while physical objects are shown by solid lines.

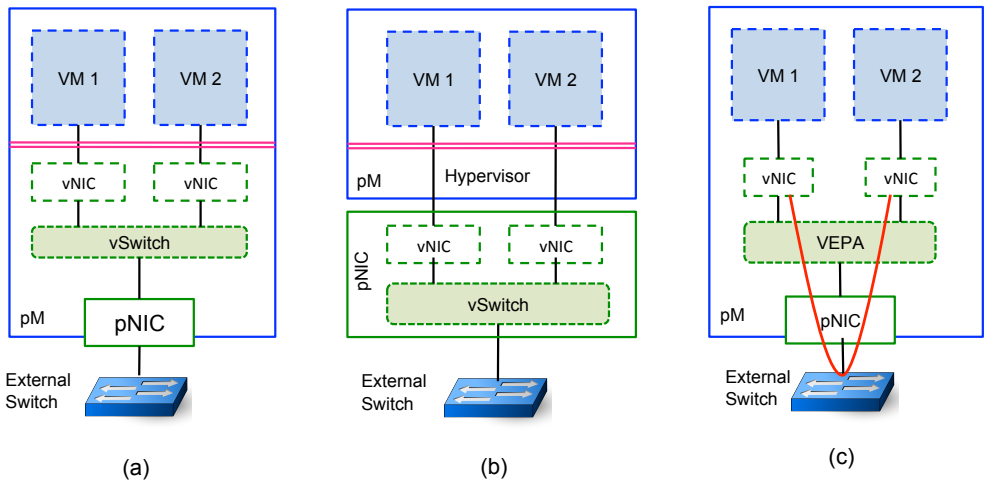

Figure 2.7: Three Approaches to NIC Virtualization

Virtualization of the NIC may seem straightforward. However, there is significant industry competition in this area. Different segments of the networking industry have come up with competing standards. Fig. 2.7 shows three different approaches. The first approach, providing a software vNIC via hypervisor, is the one proposed by VM software vendors. This virtual Ethernet bridge (VEB) [68] approach has the virtue of being transparent and straightforward. Its opponents point out that there is significant software overhead, and vNICs may not be easily manageable by external network management software. Also, vNICs may not provide all the features todays pNICs provide. So pNIC vendors (or pNIC chip vendors) have their own solution, which provides virtual NIC ports using single-root I/O virtualization (SR-IOV) [123] on the peripheral-component interconnect (PCI) bus [108]. The switch vendors (or pSwitch chip vendors) have yet another set of solutions that provide virtual channels for inter-VM communication using a virtual Ethernet port aggregator (VEPA) [87], which passes the frames simply to an external switch that implements inter-VM communication policies and reflects some traffic back to other VMs in the same machine. IEEE 802.1Qbg [3] specifies both VEB and VEPA.

– **Virtualization of switches:** A typical Ethernet switch has 32-128 ports. The number of physical machines that need to be connected on an L2 network is typically much larger than this. Therefore, several layers of switches need to be

used to form an L2 network. IEEE Bridge Port Extension standard 802.1BR [109], shown in Fig. 2.8, allows forming a virtual bridge with a large number of ports using port extenders that are simple relays and may be physical or virtual (like a vSwitch).

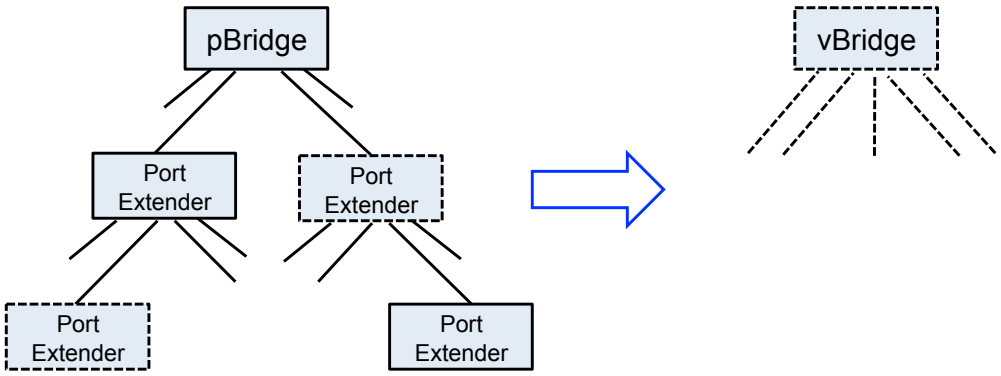

Figure 2.8: IEEE 802.1BR bridge port extension

– **Virtual LAN in Clouds :** One additional problem in the cloud environment is that multiple VMs in a single physical machine may belong to different clients and thus need to be in different virtual LANs (VLANs). As discussed earlier, each of these VLANs may span several data centers interconnected via L3 networks, as shown in Fig. 2.9. Again, there are a number of competing proposals to solve this problem. VMware and several partner companies have proposed virtual extensible LANs (VXLANs) [77]. Network virtualization using generic routing encapsulation (NVGRE) [127] and the Stateless Transport Tunneling (STT) protocol [31] are two other proposals being considered in the Network Virtualization over L3 (NVO3) working group [58] of the Internet Engineering Task Force (IETF).

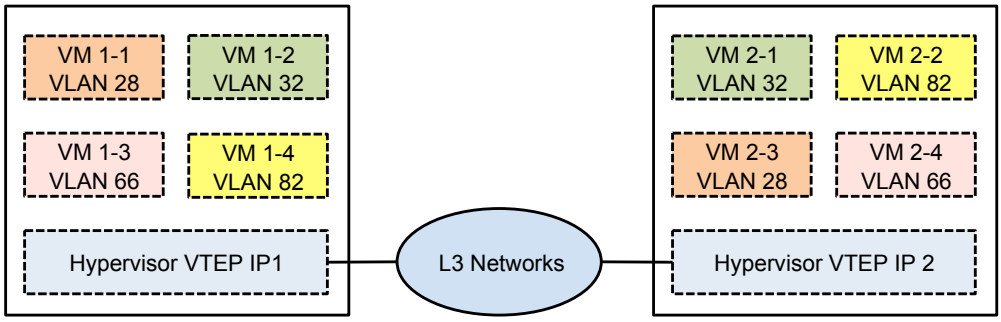

Figure 2.9: Virtual Machines on different VLANs

– **Virtualization for multi-site datacenters:** If a company has multiple data centers located in different parts of a city, it may want to be able to move its VMs anywhere in these data centers quickly and easily. That is, it may want all its VMs to be connected to a single virtual Ethernet spanning all these data centers. Again, a medium access control (MAC) over IP approach like the ones proposed earlier may be used. Transparent Interconnection of Lots of Links (TRILL) [133], which was developed to allow a virtual LAN to span a large campus network, can also be used for this.

– **Network Function Virtualization:** Standard multi-core processors are now so fast that it is possible to design networking devices using software modules that run on standard processors. By combining many different functional modules, any networking device L2 switch, L3 router, application delivery controller, and so on can be composed cost effectively and with acceptable performance. The Network Function Virtualization (NFV) group of the European Telecommunications Standards Institute (ETSI) is working on developing standards to enable this [37].

2.3.2 Middleboxes and Middleware

Apart from the compute, storage and network infrastructures, *middleboxes* and *middlewares* are two of the most essential components of modern application delivery environments. In this subsection, we will provide a detailed background on these two components with respect

to the current state-of-the-art.

- **Middleboxes:** The term "middlebox" was coined to refer to a set of functions that could neither be classified as pure network functions (such routing, switching and forwarding) nor application level functions implementing application logic. These functions included deep packet inspection on network devices to identify and prevent security attacks, maintaining packet caches to optimize the delivery of an application by avoiding the need to fetch a packet from the source every time, offload compute intensive functions such as encryption and decryption to specialized hardware appliances, and many more such cross-layer functions that could not be cleanly classified as belonging to any specific layer of the network stack. However, the fact that they are extremely essential to any modern application delivery environment can be gauged from the constantly growing market for such devices. Also, as shown in Table 2.1, the number of middleboxes deployed in modern enterprise application environments is comparable to the number of routers.

Table 2.1: Middlebox Deployment in a large Enterprise Environment [120]

Appliance Type	**Number**
Firewalls	166
NIDS	127
Conferencing/Media Gateways	110
Load Balancers	67
Proxy Caches	66
VPN Devices	45
WAN Optimizers	44
Voice Gateways	11
Middleboxes Total	636
Routers Total	900

Middleboxes are diverse, both in terms of functions and features. In-fact the term *middlebox* truly captures, perhaps the only common property that all middleboxes share; that they are *middle* or *intermediary* devices that are inserted between standard devices such as a network switch and an application server. A middlebox may refer to

a separate physical hardware device or to a virtual function within a standard switch or a router. For an extensive taxonomy and precise description of different types of middleboxes, please refer to [16]. For our purposes here, we classify middleboxes along two attributes:

- **Protocol layer:** This indicates the protocol layer at which a middlebox operates. Most middleboxes operate at more than one protocol layers. However, we are concerned in classifying middleboxes into two distinct groups based on which protocol layers it operates on - **1) Network-layer midddleboxes** that operates on Layer2, Layer 3 and Layer 4 packet headers, and **2) Application-layer middleboxes** that can operate on application layer protocols and application-layer data in addition to the Layer 2, 3 and 4 headers. For example a Network Address Translator (NAT) acts only on the IP layer while TCP relays and TCP performance enhancing proxies act at the Layer 4. Examples of application-layer middleboxes include Intrusion Detection and Prevention Systems that filter packets at each protocol layers (Layer 2 through Layer 7) to stop potential security attacks. This classification is important in designing AppFabric because we need to distinguish between middleboxes that are operated directly and thus trusted by the Application Service Provider against middlebox functions that can be delegated to third-party providers such as ISPs and cloud providers.
- **Granularity**: Granularity refers to whether the middlebox acts over network packets (and possibly a flow of packets) or if it terminates the Layer 4 connection and acts over messages and application sessions. The reason why this classification is important for our purposes is because the mechanism for inserting packet-level middleboxes into an application deployment is very different from that of inserting message-level middleboxes.

  Packet-level middleboxes are treated as part of the network infrastructure and so inserting them into the application deployment is handled by the network administrators. However, it may be obvious that this is the source of many problems. Firstly, middleboxes are deployed to support the application and need to be inserted based on policies specified by the application. The infrastructure on the other hand does not need to be application-aware (at-least currently it is not) and so their control and management is determined by network policies (such as high utilization of network links, avoiding congested or failed links, etc.).

Therefore, managing middleboxes as part of the network infrastructure creates a tussle which is difficult to address. Secondly, this solution assumes that the infrastructure and the application both have the same ownership. While this was generally true till a few years back, with cloud computing, this assumption is no longer valid. Unlike enterprise datacenter environments, the application can no longer delegate the responsibility of inserting (appropriate) middleboxes in the application path to the network control. Third, inserting packet-level middleboxes becomes extremely hard when more than one middlebox needs to be inserted. This is because the network control mechanisms generally do not mandate the specific hops in a particular route. As a result, often to simplify the deployment of middleboxes, all traffic is made to pass through all the middleboxes in a *statically* determined sequence.

Inserting message-level middleboxes is a whole new story. Message-level middleboxes need to terminate an end-to-end Layer 4 connection. However, since it is not the real application end-point, it needs to splice the message on another transport connection. If there are more than one message-level middleboxes, all of them need to splice the transport-level connection to the next hop. Now, if we wanted to efficiently process messages in this environment, all messages may not need to be processed by all the message-level middleboxes. Instead, based on message-level or session-level application policies, messages should be routed through an appropriate set of middleboxes. To do this, all these message-level middleboxes need to be configured accordingly. In the current state-of-the-art, administrators either need to configure these policies individually into each middlebox or all the middleboxes are lumped together on a specially-designed hardware device often marketed as *Application Delivery Controllers* or ADCs, as was described in the beginning of this section (Fig. 2.3). The problem with manual configurations is that it is tedious, error prone and static while centralized solutions such as ADCs are expensive, may introduce bottlenecks, introduce a central point of failure and is difficult to scale.

Therefore, although being an extremely important component of modern application deployment environments, middleboxes are often treated as a nuisance that breaks many of the beautiful architectural assumptions (owing to its cross-layer functionality) and thus makes it difficult to *safely* extend or change an existing infrastructure

configuration. In-effect, they may turn out to be a nightmare to manage and debug. AppFabric addresses some of these issues by making it much simpler to deploy and manage middleboxes by tying their control and management to a common controller that manages all the components of the application delivery network infrastructure. The details will be presented in the next chapters.

- **Middlebox Deployment:** We mentioned in the last point that it is hard to deploy middleboxes in application delivery environments. Let us look at some of the deployment contexts of middleboxes and how they vary across different application delivery environments.

  - **Enterprise Datacenter Environments:** Middlebox deployment issues in enterprise environments is a long-standing problem that has been addressed before [65, 139, 129]. The root of the problem is the lack of explicit support for middleboxes in IP. As a result, datacenter administrators need to resort to ad-hoc configuration techniques for inserting middleboxes into the applications data path. These techniques include physically interposing middleboxes in front of the application servers (Fig. 2.10) or artificially changing network parameters, such as link weights, to route application traffic through a middlebox. The problem becomes orders of magnitude harder for deploying a sequence of middleboxes. Also, these non-standard techniques are extremely error prone and cannot guarantee correctness, especially under routing dynamics such as IP path changes.

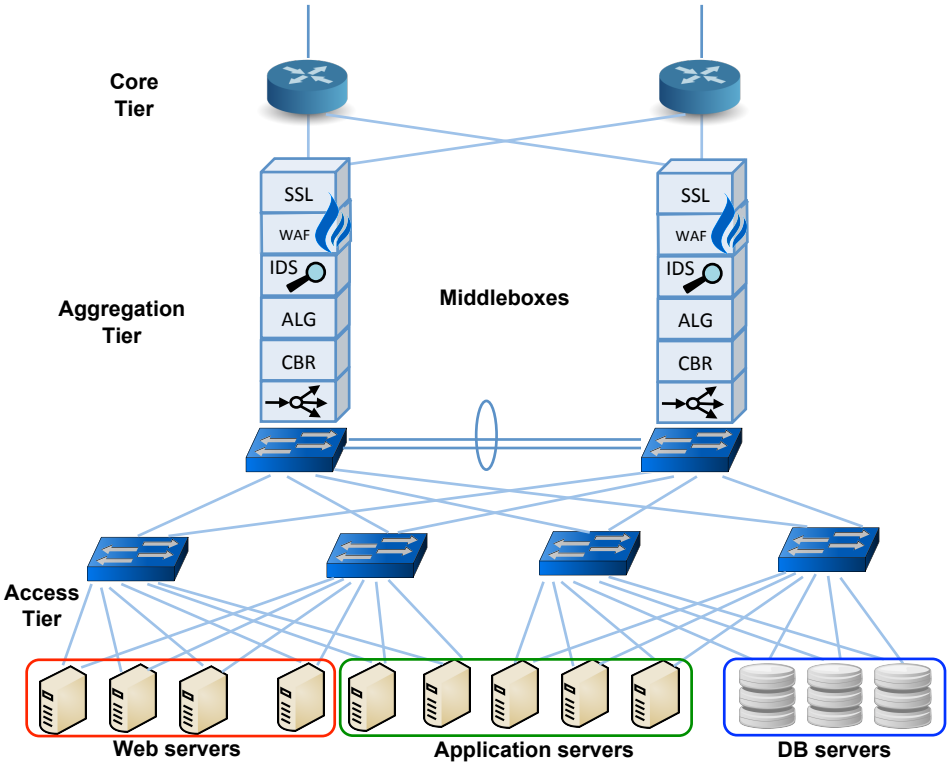

Figure 2.10: Middlebox Deployment in Enterprise Datacenters

To alleviate these middlebox deployment problems a new class of centralized middlebox platforms called Application Delivery Controllers (ADCs) emerged (Fig. 2.11). ADCs are proprietary middlebox solutions (e.g. F5 Networks Big IP [39], Citrix NetScalar [25]) based on special purpose, high-performance hardware. These boxes are extremely popular in enterprise application environments since they solve most of the problems associated with middlebox deployments. They provide a centralized and consolidated middlebox solution and hence are easy to deploy. They are based on specially designed high-performance hardware and hence each ADC may serve a considerably large pool of application servers. From the high-level design of an ADC shown in Fig. 2.11, it may be noted that the core platform consisting of the full proxy, rules and control layers implement a configurable APR services tier while the ADP services (SSH, Access Control, IDS, Transcoder, etc) are installed on top of it. Thus, ASPs can more efficiently

manage the ADP services that the application traffic accesses by creating deployment specific policies in the Rules layer.

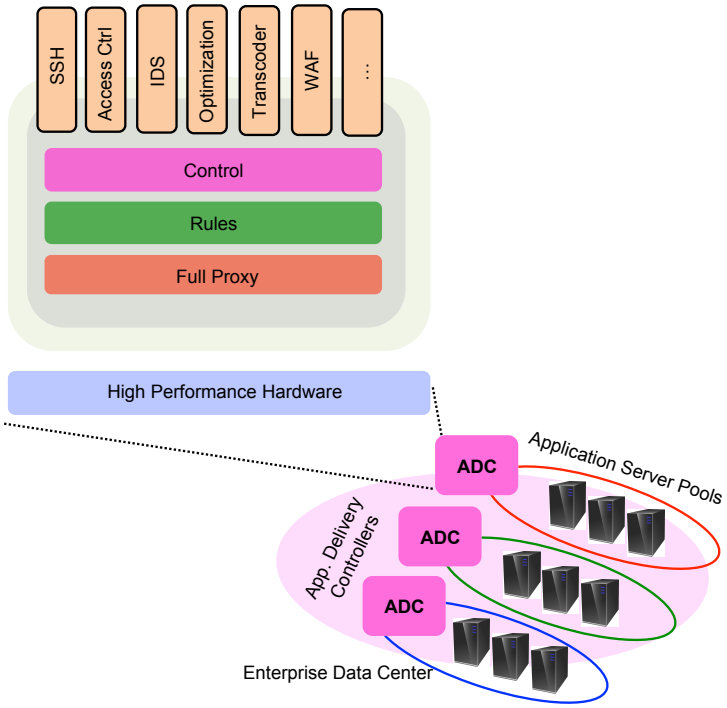

Figure 2.11: Application Delivery Controller Deployment in an Enterprise Datacenter

ADCs are great except that they are proprietary solutions and hence expensive. They are monolithic and hence can only scale vertically (by adding a new ADC), thus incurring high marginal costs to scale the deployment at the ADCs performance boundaries. Also, both ADP processing (owing to the diversity in application-layer data processing) and APR processing (depending on the complexity of the rule) is non-deterministic, making it difficult to virtualize ADC platforms with some notion of resource fairness (except for static partitioning) and hence difficult to share across logically isolated application deployment contexts.

– **Cloud Datacenter Environments:** One of the key barriers to migrating enterprise applications to the cloud is the unavailability of application delivery services (middleboxes) in cloud datacenters. All that cloud providers offer in the way of

application delivery services is a very basic load balancing service (e.g. elastic load balancing or ELB in Amazon EC2 [121]). There are two approaches to address the issue of providing application delivery services in cloud datacenters: a) Under-the-Cloud (UtC), and b) Over-the-Cloud (OtC).

* **Under-the-Cloud (UtC) Approach:** As shown in Fig. 2.12, in the UtC approach, the Cloud Service Provider (CSP) may provide support for application delivery services by leasing out special ADC-like hardware appliances to its tenants. However, the problem with this approach is that it is extremely difficult to virtualize ADCs among multiple tenants and hence tenants would need to lease dedicated ADC resources. This may not be economical, especially as the applications scale-up and down dynamically.

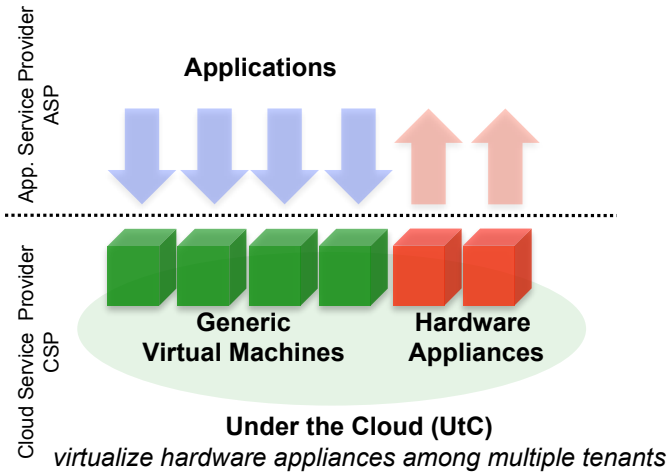

Figure 2.12: Under-the-Cloud (UtC) Application Delivery Services [101]

* **Over-the-Cloud (OtC) Approach:** As shown in Fig. 2.13, in the OtC approach, the ASP deploys virtual appliances on generic virtual machines leased from the cloud provider. Though this seems to be a more plausible solution economically, it introduces new deployment challenges. Virtual appliances are resource constrained and hence the monolithic ADC-like deployment model is

not possible. Application deployments need to be supported by multiple virtual appliances, each providing a specific service. This leads to management issues. To add to this problem, the ASP does not have access to the network infrastructure of the cloud datacenter. Hence, the ASP cannot depend on the ad-hoc network configuration techniques used in enterprise datacenters to deploy virtual appliances in a cloud datacenter.

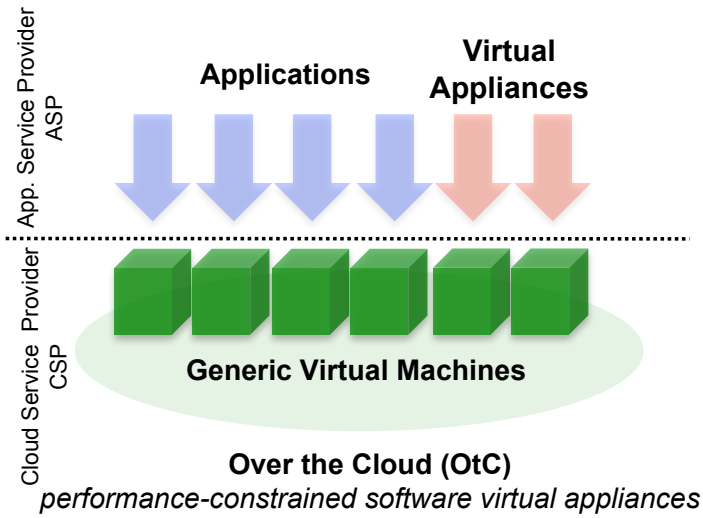

Figure 2.13: Over-the-Cloud (OtC) Application Delivery Services [101]

Therefore, to deploy virtual appliances in cloud datacenters, the ASP needs to be able to explicitly route application traffic through these virtual appliances. This problem may be addressed either, through an UtC approach, or an OtC approach. In the UtC approach, the CSP may provide new interfaces through which the ASP may configure workflows to chain virtual appliances and application servers. Thus, the ASP may specify the policies in the control plane and the CSP will be responsible for enforcing these policies in the data plane. In the OtC approach, the ASP will be responsible for both, specifying and enforcing the deployment policies over each application-level component, including virtual appliances and application servers. It may be noted that these workflows need

to be configured dynamically to handle variable load over multiple virtual appliances and application servers and at the same time need to adhere to session affinity constraints (application messages belonging to the same session need to be mapped to the same workflow).

– **Multi-Datacenter Environments:** Modern Internet-scale applications benefit from geographically distributed deployments to be able to serve their global user base and also to ensure high availability. Applications may be replicated and partitioned across different datacenters. Such deployments present new application delivery challenges in routing user requests dynamically to the proper application instance. We will discuss three separate application deployment scenarios in this category.

  * **Hybrid Deployments:** Hybrid deployments, which we also call failover deployments, represent the most common use of cloud computing at present. In this environment, the cloud acts like a failover infrastructure to the enterprise data center. Under conditions such as extreme or unexpected usage spikes, failures and planned maintenance, the enterprise datacenter may migrate or replicate some of its application servers to the cloud. Since this is a temporary arrangement, most likely for the duration of the churn event, during this time application traffic may be indirected (or bounced) through the middlebox infrastructure in the enterprise datacenter (Fig. 2.14). The scope of this solution is limited to intermittent failure and scaling events that can be solved by simply throwing in more hardware to the problem. Also, the solution assumes that even under such extreme circumstances (except for planned maintenance), all the middleboxes as well as the network infrastructure of the enterprise datacenter is still available.

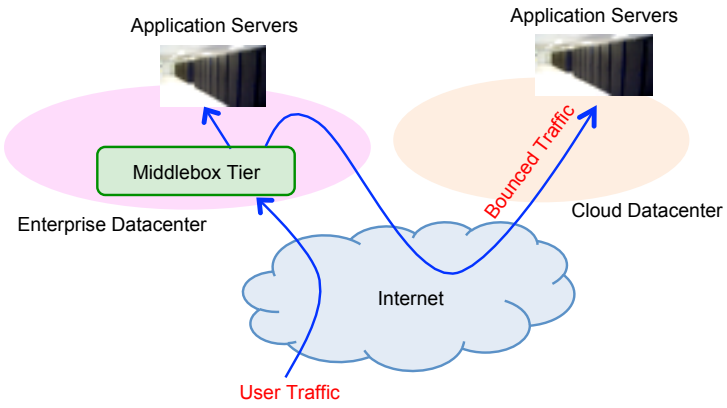

Figure 2.14: Hybrid (private-public) Application Deployment

* **Multi-Enterprise Datacenter Environments:** Most large ASPs like Google, Facebook, Microsoft, etc. operate multiple globally distributed datacenters. For managing these deployments, these ASPs need to deploy Application Policy Routing(APR) services as global services outside the datacenters. These services include global load balancing, fault-aware routing, context routing and content-based routing. To provide such distributed APR services, large ASPs like Google operate application level (layer 4-7) proxies (most probably) at all major network POPs with the Googles WAN (Fig. 2.15) infrastructure [49, 50]. Accesses to Googles applications are intercepted at the nearest POP and intelligently routed to a Google datacenter that is best suited(in terms of some metric) to serve the request. However, managing such a distributed APR service infrastructure incurs significant capital and operational expenditures.

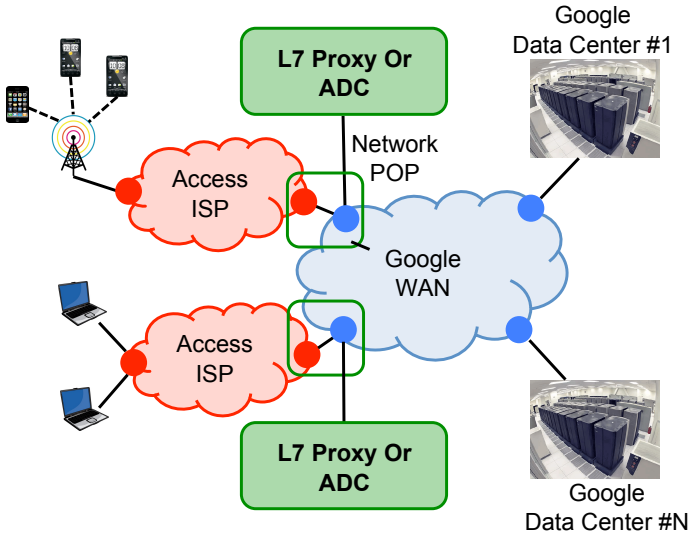

Figure 2.15: Multi-Enterprise Datacenter Application Deployment

* **Multi-Cloud Datacenter Environments:** As cloud computing advances, it is expected that different cloud platforms will eventually converge over standardized access APIs (across different cloud technologies) and business models (across separate Cloud Service Providers) to provide a globally distributed and automatically accessible computing platform for application deployment (Fig. 2.16). This will provide new opportunities to ASPs to have global presence without the need to own and operate their own datacenters. Also, ASPs may dynamically optimize their application deployments based on different deployment parameters including cost, access patterns and failures, by instantiating new application replicas/partitions and bringing down existing ones. However, for these smaller ASPs, although cloud computing provides them with the opportunity to match the globally distributed deployments of larger ASPs, it is prohibitively expensive for them to own and operate a distributed APR services infrastructure. A feasible alternative would be to outsource APR services to third-party providers such as ISPs that can provide it as a shared service to the ASPs.

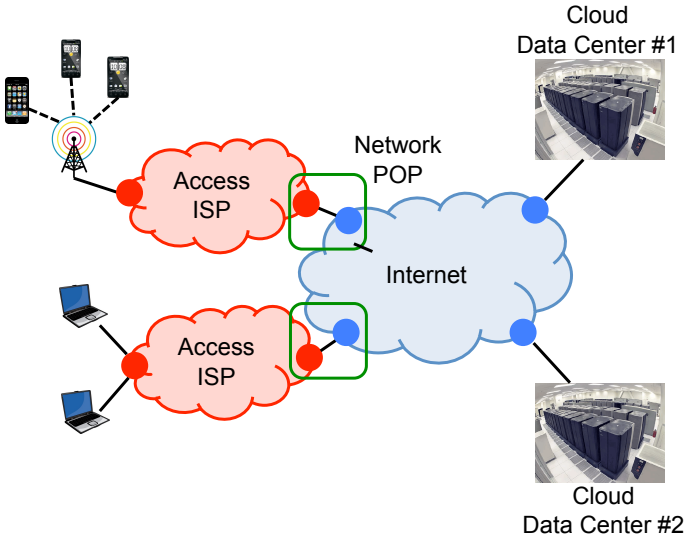

Figure 2.16: Multi-Cloud Datacenter Application Deployment

We summarize the observations in this section through a subjective plot (Fig. 2.17) of opportunity (for application deployment) vs. difficulty (of application delivery) for the various application deployment scenarios. The goal is to motivate our claim that a generic platform for application delivery is required for the Internet.

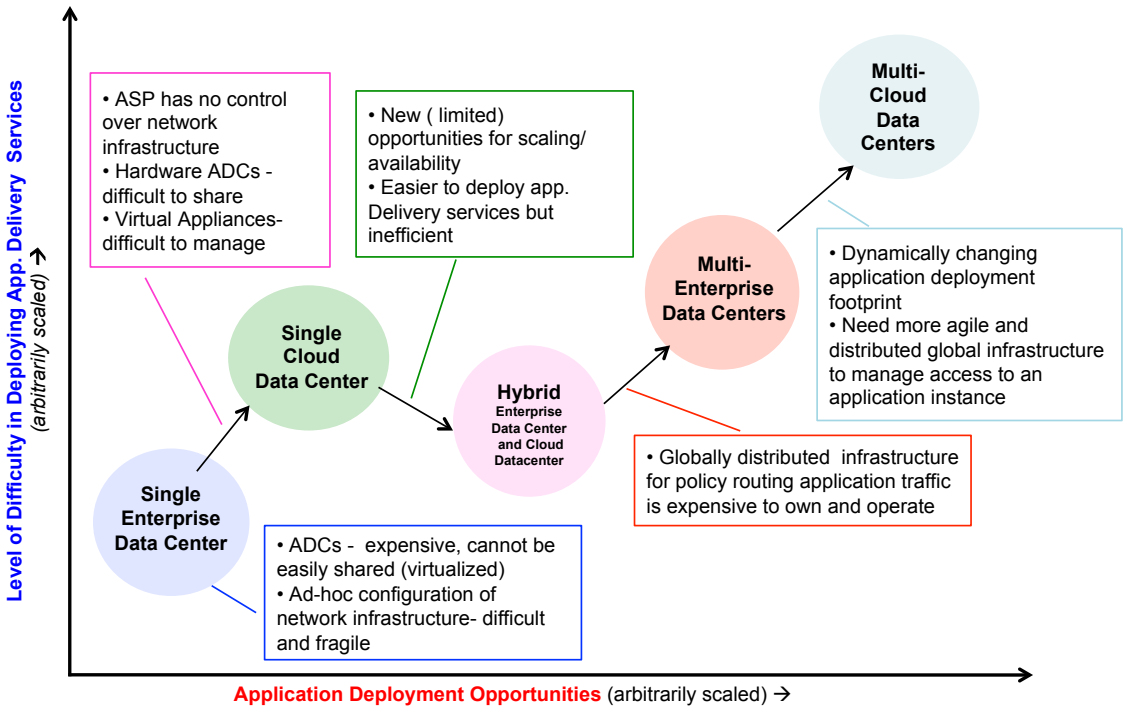

Figure 2.17: Subjective Plot of Opportunity-vs-Difficulty

Fig. 2.17 is a subjective plot (based on loose comparisons) and is not meant to make quantitative claims based on any particular metric (in both the axes) and is also not properly scaled. The x-axis represents the opportunities that different application deployment environments provide. The subjective parameters that are used to represent opportunity of deployment include:

- **Dynamicity:** The opportunity to dynamically optimize the deployment for cost, performance, energy, etc.
- **Availability:** The opportunity to ensure availability in the event of infrastructure failures, including computing and networking failures, through mechanisms such as service mobility, distributed service replicas. dynamic scaling, session handoff, etc.
- **Adaptability:** The opportunity to adapt to usage patterns, both in terms of load and location.

– **Innovation:** The opportunity to innovate by reducing risk; by having low barrier for entry of new ideas and introduce changes to existing ideas without having to commit to long term investments.

The y-axis represents the level of difficulty in delivering applications across these different deployment scenarios. The single enterprise datacenter environment is considered to be the basis for comparing other deployment scenarios. It represents having the least application deployment opportunities and also the least difficulty in delivering them through dedicated ADCs. It is not at the origin since it provides at-least more deployment opportunities than traditional third-party hosting (note: not computing) environments and has more difficulty in deploying application delivery services than an environment that does not use any application delivery services. To summarize, the key points in favor of the need of a generic transport for application delivery, such as OpenADN, are as follows:

– OpenADN will provide an explicit interface for service composition to allow ADP services to be deployed more easily by allowing them to be independently instantiated off-path and included in the deployment through explicit configuration. Also, the APR and service composition services would allow applications themselves to be designed as separate application service components and dynamically composed to suit a certain application context.

– By generalizing the application delivery transport, the need for expensive, specialized hardware-based ADC solutions can be avoided by allowing ADP services to be implemented and deployed as separate components. Component based ADP is also more economical in terms of scaling a particular ADP service component, and hence more preferable, as long as the underlying infrastructure provides mechanisms to easily manage the complexity (through the APR and service composition services) as a result of moving away from the convenient centralized solution based on monolithic ADCs.

– In single cloud datacenter environments, OpenADN will also make deploying resource-constrained, component based ADP services over virtual appliances easier by placing them in separate dynamically scaling (or elastic) groups (managed through the APR service interface) and composing them through the service composition interface.

- OpenADN will support multi-datacenter and multi-cloud environments by providing a generic APR service interface, allowing these services to be easily outsourced to third-party providers such as ISPs, CDNs (Content Delivery Networks) and CSPs. There are two requirements to achieve this goal: a) the ASP needs to be able to control and configure the APR services to suit its particular deployment requirement even though the service is instantiated over a third party provider, and 2) the third party provider should not need to have access to application data to provide these services in order to maintain the privacy requirements of most modern applications. These requirements present two specific design issues that OpenADN needs to address.
- For hybrid deployments, OpenADN will remove the need to indirect application traffic through the enterprise datacenter by, a) providing APR services in the network by third-party providers (Point 4), and b) making ADP services available in the cloud (Point 3).
- Each of these deployment scenarios will benefit from an interface that will allow the ASP to specify and access the specific data transport services (QoS), and provided by third party providers such as ISPs and CSPs. OpenADN will provide a standard interface between the ASP and the network service provider to achieve this.

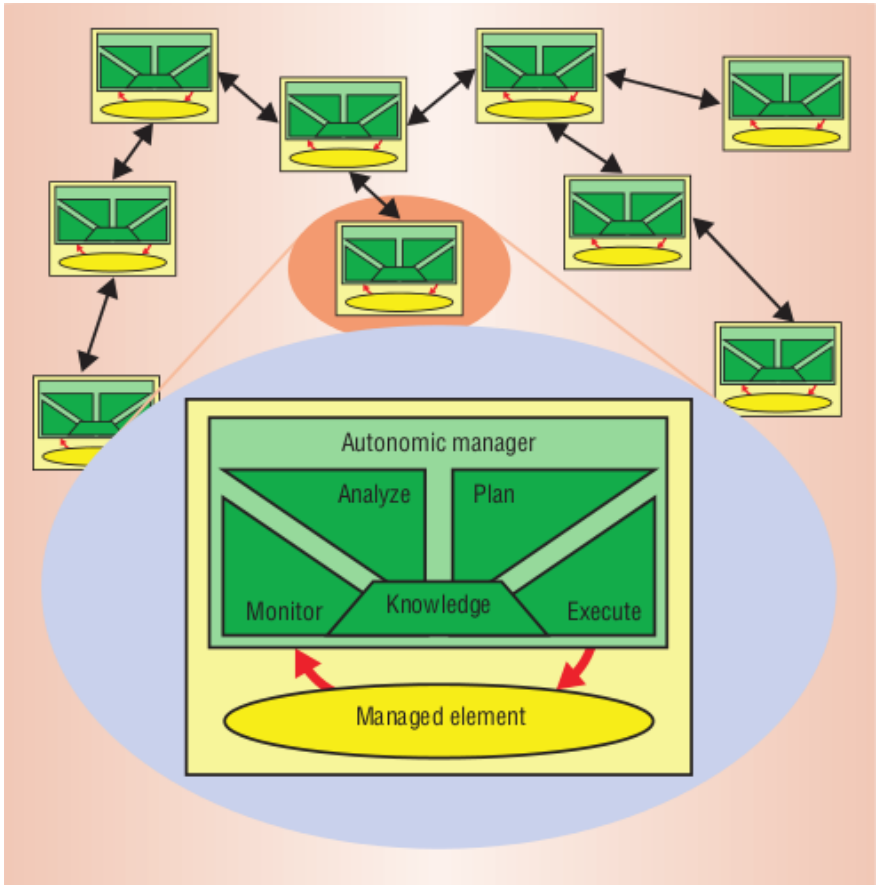

Figure 2.18: IBM autonomic Computing: Structure of an Autonomic Element [67]

- **Middleware:** In 2001, IBM published a manifesto introducing a new paradigm of system design called autonomic computing [67]. It was based on the observation that the sheer scale and complexity of modern large-scale software systems was a huge deterrent to the progress of IT in empowering a wide-range of business processes. Deploying and managing such systems was turning out to be a mammoth task even for the most skillful IT workforce. Under these circumstances, the natural solution was to try and design more autonomic system architectures. These systems would be able to self-configure based on high-level policies, self-optimize to improve performance and efficiency under extremely dynamic operating conditions including variable load, network connectivity and power constraints, self-heal against local and global failures, and self-protect against configuration errors, cascading failures and/or directed attacks. Fig. 2.18

shows a high-level framework for designing autonomic systems published by IBM research. The idea is to attach an autonomic manager with each node (where a node may represent a hardware resources such as a CPU, storage, etc. or a software resource such as a database service, web service etc.), turning it into a managed element. The IBM manifesto spurred a new hype wherein system designers started labeling and attributing self-* capabilities to their systems. However, due to the lack of a standard open framework, autonomic computing has evolved through vendor specific solutions such as IBM Websphere [55] adding several of these capabilities into their middleware product.

The concept of autonomic computing was proposed to provide infrastructure-level support to the deployment and management of large-scale system integration platforms. Since the early 1990s, system designers have been trying to design generic integration platforms that would make it easier to build large-scale software systems by integrating separate software pieces written in different languages, running on different processes/hosts, distributed across different geographical and administrative boundaries, and each solving separate point problems. In 1991, the Object Management Group (OMG) released version 1 of the Common Object Request Broker Architecture (CORBA) [52] which provided a software platform for interoperability among distributed objects through a language and location independent interface for Remote Method Invocation (RMI). It may be safely said that CORBA was one of the key initiators of the Service-Oriented Architecture (SOA) paradigm although other SOA-flavored technologies existed before (e.g. Microsoft Object Linking and Embedding (OLE) in 1990) and roughly during the same period (e.g. Microsoft COM [80] and DCOM [81]). CORBA was the leading open standards effort at the time while the others such as COM and DCOM were proprietary. However, CORBA never reached the level of popularity and adoption that was expected of it since the standard was extremely complex and bloated resulting in many incomplete, non-compliant and buggy implementations. The SOA story changed significantly around 1998 with the increasing popularity of the XML data/document format combined with Microsoft proposing the Simple Object Access Protocol (SOAP). SOAP is an XML-based protocol for SOAP-based services to be able to exchange information and integrate. Another significant development in this area happened around 2000, when Roy Fielding proposed a new architectural style for designing distributed software systems called REST (Representational State Transfer)

[42]. REST is the generalized architectural style behind the Hypertext Transfer Protocol or HTTP (also co-designed by Roy Fielding along with Tim-Berners Lee in 1996) that lies at the heart of the World-Wide Web (WWW). REST-ful services, preferably using the JSON (JavaScript Object Notation) data-interchange format provide a much simpler and lightweight service integration platform than the SOAP/XML combination. Although REST-ful service architectures are more restrictive and less flexible than SOAP-based architectures, it often suffices most use-case requirements. For example, REST over HTTPS is often good enough for most SOA use-cases as compared to the more elaborate (and better) security provided by SOAP + WS-Security (Web Services Security - a security extension of SOAP). The SOAP vs. REST debate continues to polarize system architects, but it is clearly not the objective of this discussion to address it. The key point is that no matter which system integration platform is selected, the actual deployment and management of the system is subject to the eight fallacies of distributed computing, out of which seven fallacies were stated by Peter Deutsch in 1994 and the eighth was added by James Gosling in 1997; both working for Sun Microsystems. The fallacies state the common assumptions often made by system architects while designing distributed systems which come back to haunt them in the long run. These include:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

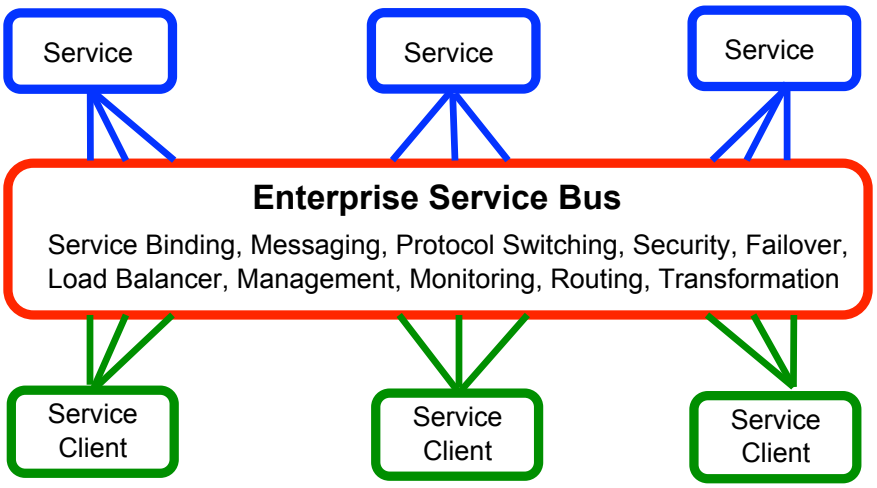

Figure 2.19: Schematic Representation of an Enterprise Service Bus (ESB)

Taking care of these issues at the infrastructure level through an intelligent service deployment and management environment would alleviate much of the pain for system architects. Such an infrastructure platform needs to be mostly generic and yet capable of optimizing for application-specific use-cases. Autonomic computing was an effort in this direction. What emerged in the process, as we have already discussed before, is a bunch of autonomic computing capabilities in vendor-specific middleware platforms. In the context of distributed service-oriented architectures, middleware platforms, generically called the Enterprise Service Bus or ESB (Fig. 2.19) provide support for service orchestration. The two key functions that the ESB provides include *message routing* and *message transformation.* Message routing involves routing messages between different independent services based on application-level content and context. Message transformation is required to make these services interoperable. The output from one service may need to be transformed to make it a suitable input to another service in the workflow. Therefore the backbone of an ESB is a *message brokering* subsystem. There are mainly two types of message brokers - centralized and distributed. Let us briefly discuss them here in order to be able to understand their weakness and appreciate the motivation behind the design choices of AppFabric.

– **Centralized message brokers:** As shown in Fig. 2.20, a centralized message broker intercepts all messages between any two services in the system, transforms the messages if required and routes them to the appropriate next hop. This

solution although very simple to implement and deploy suffer from the standard problems of any centralized architecture. The centralized broker may be the single point of failure for the entire system. Also, it may be the performance bottleneck during high load situations. These problems may be addressed to some extent by using standard techniques such as hot-standbys and deploying the the centralized broker as a cluster of smaller devices rather than one single device. However, this solution has severe limitations when serving application use-cases that are naturally distributed such as Internet-of-Things, Cyber-Physical Systems, online games, mobile apps, virtual worlds,etc. Indirecting each message through a centralized entity would be highly inefficient.

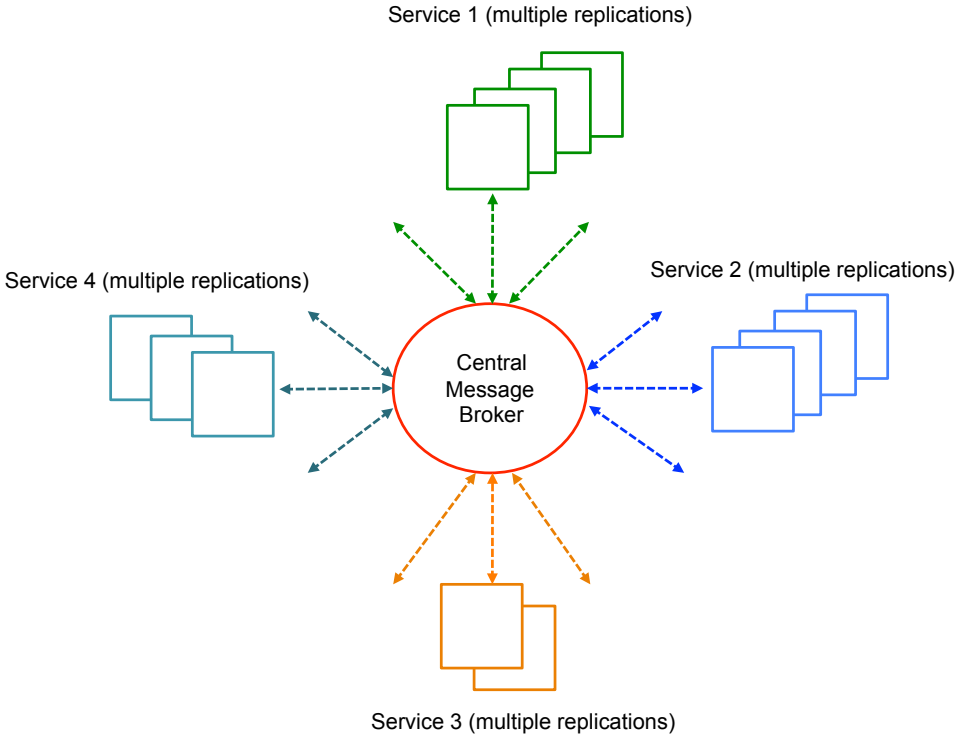

Figure 2.20: Centralized Broker Architecture

– **Distributed message brokers:** As shown in Fig. 2.21, a distributed message broker system. Again, as is standard for any distributed vs. centralized argument, the distributed system is much more resilient although much more difficult to configure, control and debug. Also, current distributed message broker solutions

are mostly limited to a few high-performance broker nodes that serve hundreds(and sometimes thousands) of services. Also, they are not generally designed to dynamically scale-out as required, both in number as well as location.

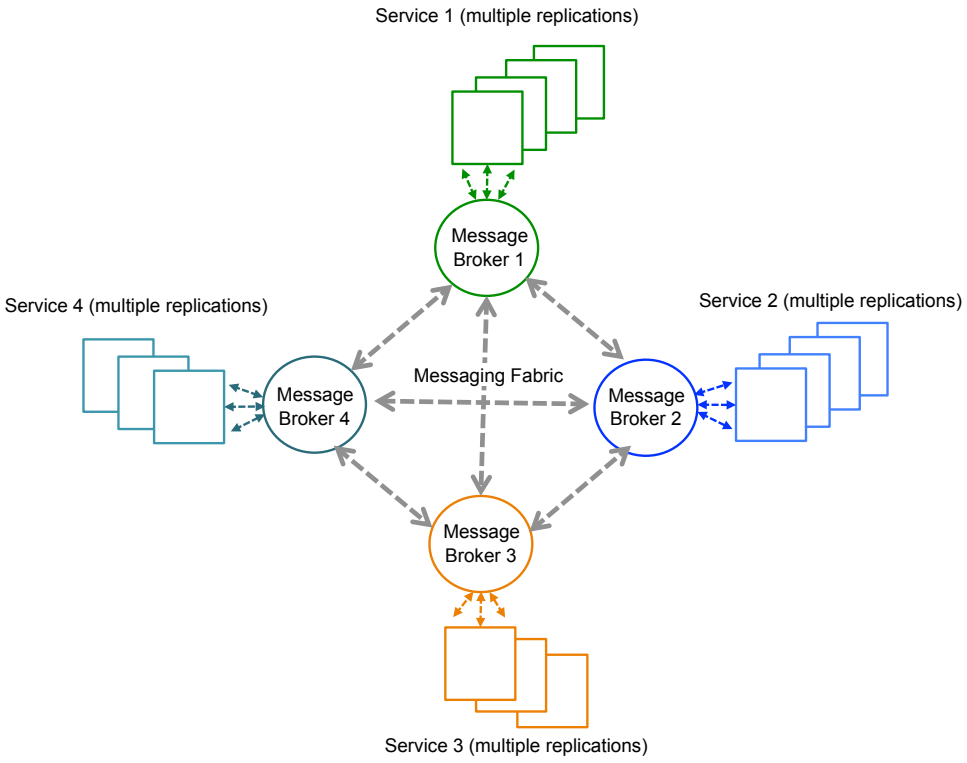

Figure 2.21: Distributed Broker Architecture

In the rest of this thesis, we will see how AppFabric addresses some of these limitations of modern middleware architectures and evolves it to make it suitable for next generation application delivery.

2.4 Next-generation Internet

AppFabric is the result of integrating many different ideas that were conceived in the cradle of next generation Internet research funded by the National Science Foundation (NSF) [91] as part of its FIND (Future Internet Design) [92] program. Therefore, it is imperative

that we provide a brief overview of some of the most prominent and promising ideas on the next generation Internet design with specific focus on how our own work on the Internet 3.0 [64, 105, 104] architecture gradually matured into the design of AppFabric.

In this section, we will first discuss a family of architectures called ID-locator split architectures which, according to us, is one of the major architectural limitations of the current Internet and also is one of the key design principles behind our Internet 3.0 architecture. Then we will discuss the key elements of the Internet 3.0 architecture followed by brief discussions on each of the four projects supported by NSF's Future Internet Architecture (FIA) program which was a followup to the FIND program. For a more detailed survey on future Internet design, please refer to [107].

2.4.1 ID-locator split architectures

The current Internet is faced with many challenges including routing scalability, mobility, multihoming, renumbering, traffic engineering, policy enforcements, and security because of the interplay between the end-to-end design of IP and the vested interests of competing stakeholders which lead to the Internets growing ossification. The architectural innovations and technologies aimed at solving these problems are set back owing to the difficulty in testing and implementing them in the context of the current Internet. New designs to address the major deficiency or to provide new services cannot be easily implemented other than by step-by-step incremental changes.

One of the underlying reasons is the overloaded semantics of IP addresses. In the current Internet, the IP addresses are used as session identifier in transport protocols such as TCP as well as the lo- cater for routing system. This means that the single IP address space is used as two different namespace for two purposes, which leads to a series of problems. The Internet Activity Board (IAB) workshop on routing and addressing [79] reached a consensus on the scalable routing issue and the overloaded meaning of IP ad- dresses. It urged further discussion and experiments on decoupling the dual meaning of IP addresses in the long-term design of the next generation Internet. Currently, there are several proposals for ID-lo- cater

split, but most of them cannot provide a complete solution to address all the challenges including naming and addressing, routing, mobility, multihoming, traffic engineering, and security.

One of the most active research groups of IRTF (Internet Research Task Force) is RRG (Routing Research Group) [61], where there is an on-going debate on deciding which way to go among several ID-locater split directions. One possible direction is called coreedge separation (or Strategy A in Herrins taxonomy [76]) which tries to keep the de-aggregated IP addresses out of the global routing tables, and the routing steps are divided into two levels: the edge routing based on identifier (ID) and the core routing based on global scalable locaters. Coreedge separation requires no changes to the end-hosts. Criticisms to this direction include difficulty in handling mobility and multihoming, and handling the path-MTU problem [76]. In some solutions, the weird ID-based routing in the edge also makes some purist believe that it is a short-term patch rather than a long-term solution. Typical solutions include LISP [40] , IVIP, DYNA, SIX/ONE, APT, TRRP (all form [62]). This coreedge separation can be deemed as decoupling the ID from locater in the network side, which is an intuitive and direct idea for the routing scalability issue and relatively easy to deploy, but not good at solving the host mobility, host multihoming, and traffic engineering. Other recent RRG proposals include: 2- phased mapping, GLI-split, hIPv4, Layered Mapping System (LMS), Name Overlay (NOL), Name-Based Sockets, Routing and Addressing in Next-Generation EnteRprises (RANGER), and Tunneled Inter-do- main Routing (TIDR). They are related to this category from different aspects such as naming, addressing, sockets, encapsulation, mapping, and hierarchy.

The other direction is called ID locater split which requires globally aggregatable locaters to be assigned to every host. The IDs are decoupled from locaters in the end-hosts network stacks and the mapping between IDs and locaters is done by a separate distributed system. The proposals following this direction handle mobility, multihoming, renumbering, etc., well. However, they do require host changes and it may be hard to ensure compatibility with the current applications. Typical solutions include HIP [83], Shim6 [89], I3 [129], and Hi3 [88].

It is seen that these two directions have their own advantages and disadvantages, and it is hard to judge which one is right for the future Internet. Here we describe two example solutions (HIP and LISP) of these two directions, and after that we discuss our MILSA [98, 99, 97, 100] solution which combines the advantages of these two directions and avoids their disadvantages.

- **Host Identity Protocol:** Host Identity Protocol or HIP [83] is one of the most important ID locater split schemes which implements the decoupling of ID from locater in end-hosts. It has been under development in the HIP working group of IETF for couple of years.

  HIP introduces a new public keys based namespace of identifiers which enable some end-to-end security features. The new namespace is called Host Identity (HI) which is presented as a 128-bit long value called Host ID Tag (HIT). After the decoupling of HIs from IP addresses, the sockets are bound to HITs instead of IP addresses, and the HITs are translated into IP addresses in the kernel. HIP defines the protocols and architecture for the basic mechanisms for discovering and authenticating bindings between public keys and IP addresses. It explores the consequence of the ID locater split and tries to implement it in the real Internet. Besides security, mobility and multihoming are also HIPs design goals and are relatively easier to implement than the coreedge separation solutions. HIP supports opportunistic host-to-host IP-Sec ESP (Encapsulation Security Protocol), end-host mobility across IPv4 and IPv6, end-host multi-address multihoming, and application interoperability across IPv4/IPv6.

  However, for HIP, although the flat cryptographic-based identifier is useful for security, it is not human-understandable and not easy to be used to setup trust relationship and policies among different domains or organizations. It uses the current DNS system to do the mapping from ID to locater which is not capable of dealing with the mobility under fast handover situation, and multihoming. Specifically, mobility is achieved in two ways: UPDATE packets and rendezvous servers. First way is simple but it does not support simultaneous movement for both end-hosts. Rendezvous servers are better

but do not reflect the organizational structure (realm), and there is no explicit signaling and data separation in the network layer.

Moreover, HIP requires that all the changes happen in the end- hosts which may potentially require significant changes to the cur- rent Internet structure and could lead to compatibility issues for the existing protocols and applications.

- **Locator ID Separation Protocol:** Locater ID Separation Protocol or LISP is another important ID locater split scheme following the coreedge separation approach which implements the decoupling of ID from locater in the network side instead of the host side. It is being developed by the LISP working group of IETF.

  LISP is a more direct solution for routing scalability issue. LISP uses IP-in-IP packets tunneling and forwarding to split identifiers from locaters which eliminates the Provider Independent (PI) ad- dresses usage in the core routing system and thus enables scalability. The tunnel end-point routers keep the ID-to-locaters cache and the locater addresses are the IP addresses of the egress tunnel routers. The mapping from ID to aggregatable locaters is done at the border of the network, i.e., the tunnel end-point routers.

  LISP enables site multihoming without any changes to the end- hosts. The mapping from identifier to RLOC (Routing Locater) is performed by the edge routers. LISP also does not introduce a new namespace. Changes to the routers are only in the edge routers. The high-end site or provider core routers do not have to be changed. All these characteristics of LISP lead to a rapid deployment with low costs. There is also no centralized ID to locater mapping database and all the databases can be distributed which enable high mapping data upgrade rates. Since LISP does not re- quire current end-hosts with different hardware, OS platform and applications, and network technologies to change their implementations, the transition is easier compared to HIP. The requirements for hardware changes are also small which allow fast product delivery and deployment.

However, LISP uses PI addresses as routable IDs which potentially leads to some problems. In the future, it will be necessary to create economic incentives to not use the PI addresses, or to create an automatic method for renumbering by Provider Aggregatable (PA) addresses.

Obviously, there is a tradeoff between compatibility to the cur- rent applications and enabling more powerful functions. Since LISP does not introduce any changes to the end-host network stack, by design it cannot support the same level of mobility as HIP. The host multihoming issue is similar. Specifically, from design perspectives, LISP lacks support for host mobility, host multihoming, and traffic engineering. Some researchers argue that LISP is a short- term solution for routing scalability rather than a long-term solution for all the challenges listed in the beginning of this section.

- **Mobility and Multihoming supporting Identifier Locator Split Architecture (MILSA):** MILSA [98, 99, 97, 100] is basically an evolutionary hybrid design which has combined features of HIP and LISP, and avoids the disadvantages of these two individual solutions. Since there is still a debate regarding whether the ID locater split should happen in end-host side such as HIP or in network side such as LISP, it is hard to decide which is the right way to go at this point of time. Thus, MILSA is designed to be adaptive; it supports both directions and allows them to evolve to either direction in the future. By doing this, we can avoid the deployment risk at the furthest.

  Specifically, MILSA introduces a new ID sublayer into the net- work layer in the current network stack, i.e., it separates ID from locater in the end-host and uses a separate distributed mapping system to deliver fast and efficient mapping lookup and update across the whole Internet. MILSA also separates trust relationships (administrative realms) from connectivity (infrastructure realms). The detailed mechanisms on how to setup and maintain this trust relationship are presented in [99]. A new hierarchical ID space is introduced which combines the features of flat IDs and hierarchical IDs. It allows a scalable bridging function that is placed between the host realms and the infrastructure realms. The new ID space can be used to facilitate the setup and maintenance of the trust relationships, and the policy enforcements among different organizations. Moreover, MILSA implements signaling and data separation to

improve the system performance, efficiency, and to support mobility. Detailed trust relationship setup and maintenance policies and processes are also presented in MILSA.

Through the hybrid combination, the two approaches are integrated into one solution to solve all the problems identified by the IRTF RRG design goals [90] which include mobility, multihoming, routing scalability, traffic engineering, and incremental deployability. It prevents the Provider Independent (PI) address usage for global routing, and implements identifier locater split in the host to provide routing scalability, mobility, multihoming, and traffic engineering. Also the global routing table size can be reduced step by step through our incremental deployment strategy which is also one of the biggest MILSA advantages. Specifically, in MILSA, different deployment strategies can be implemented to gain fastest routing table size reduction considering the different incentives or motivations from both technical and non-technical aspects, i.e., the strategies make sure that each incremental deployment step of MILSA can pay off with reasonable and accept- able balance between costs and benefits. Different incentives such as scalability, mobility, and multihoming lead to different deployment models which have different effect in reducing the routing table size gradually.

2.4.2 Internet 3.0

The Internet 3.0 project [64, 105, 104] is a clean-slate architecture to overcome several limitations of the current Internet. The top features are: strong security, energy efficiency, mobility, and organizational policies. The architecture explicitly recognizes new trends in separate ownership of infrastructure (carriers), hosts (clouds), users and contents and their economic relationships. This will shape the services that the network can provide enabling new business models and applications.

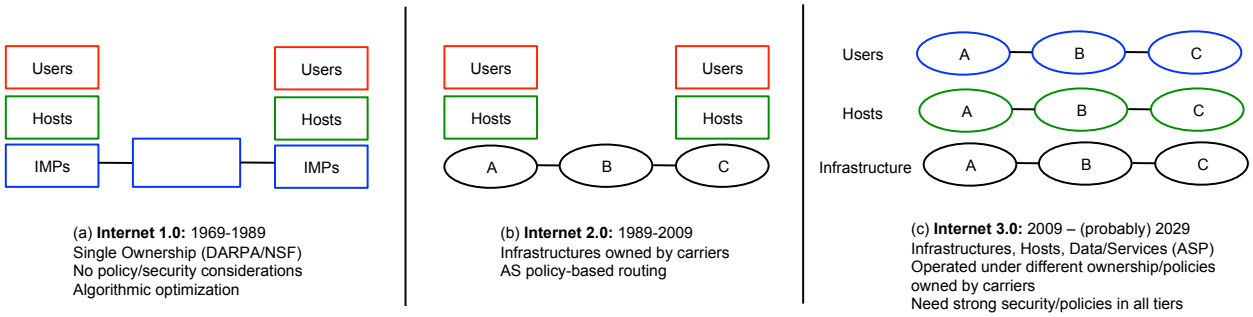

Figure 2.22: Internet Generations

As shown in Fig. 2.22, Internet 1.0 (approximately around 1969) had no ownership concept since the entire network was operated by one organization. Thus, protocols were designed for algorithmic optimization with complete knowledge of link speeds, hosts, and connectivity. Commercialization of Internet in 1989 led to multiple ownership of networking infrastructure in what we call Internet 2.0. A key impact of ownership is that communication is based on policies (rather than algorithmic optimization) as is seen in inter-domain (BGP) routing. The internals of the autonomous systems are not exposed. We are seeing this trend of multiple ownership to continue from infrastructure to hosts/devices (Clouds), users, and content. Internet 3.0s goal is to allow policy-based secure communication that is aware of different policies at the granularity of users, content, hosts, or infrastructure.

Cloud computing is an example of applications that will benefit from this inherent diversity in the network design. Hosts belonging to different cloud computing platforms can be leased for the duration of experiments requiring use of data (e.g., Gnome) to be analyzed by scientists from different institutions. The users, data, hosts, and infrastructures belong to different organizations and need to enforce their respective policies including security. Numerous other examples, related to P2P computing, national security, distributed services, cellular services exist.

Organization is a general term that not only includes employers (of users), owners (of devices, infrastructure, and content) but also includes logical groups such as governments, virtual interest groups, and user communities. Real security can be achieved only if such

organizational policies are taken into account and if we de- sign means of monitoring, measurement, and independent validation and enforcement.

Internet 1.0 was designed for host systems that had multiple users and data. Therefore, the hosts were the end systems for communication. Today, each user has multiple communication devices. Content is replicated over many systems and can be retrieved in parallel from multiple systems. The future user-to-user, user-to- content, machine-to-machine communications need a new paradigm for communication that recognizes this new reality and allows mobility/multihoming for users and content as easily as it does for devices. In this new paradigm, the devices (hosts) are intermediate systems while the users and content are the end-systems. The inclusion of content as an end-system requires Internet to provide new services (e.g., storage, disruption tolerance, etc.) for developing application specific networking contexts. There will be more intelligence in the network which will also allow it to be used easily to use by billions of networking-unaware users.

Internet 3.0 uses the term Realm to represent a trust domain such as an organization. All entities within a tier belonging to a single organization belong to a realm. The management and control plane of the realm, which we generically call Realm Manager (RM) enforces security and other organizational policies. These policies can be very general and may include security considerations, e.g., authentication and authorization. RMs also provide additional services such as ID-locator translation that allows objects to move without loosing connections and energy management services. RMs are part of the management and control plane and are active during the start phase of a communication. Once set up, the communication can continue in the data plane without intervention of the RMs.

Realms overlay entities with a discrete ownership framework. Ownership entails related security, administrative and management responsibilities. In the Three-tier Object Model (Fig. 2.23), the bottom tier infrastructure is owned by multiple infrastructure owners. The second tier of hosts is owned by individual users or different organizations such as DoE, DARPA, and Amazon. The third tier of users and data may belong to specific organizations

or individual users. Thus, realms represent logical division of entities into multiple ownership, trust, and management domains.

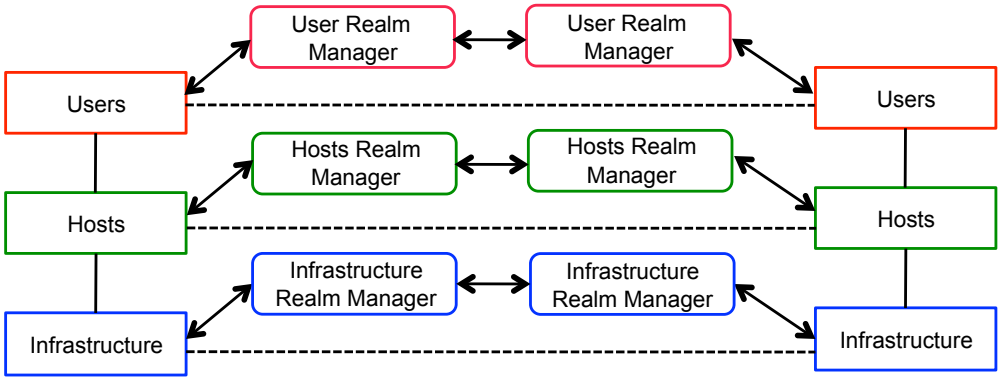

Figure 2.23: Organization of "Objects" in Internet 3.0

Explicit representation of ownership simplifies the security and policy framework design through more natural representation and enforcement of policies rather than conflating them with functionality as in the current Internet.

Realms advertise specific services through Objects. Objects encapsulate the complexities of resource allocation, resource sharing, security and policy enforcements, etc., and expose a standard interface representing capabilities (in standardized abstract parameters) and fixed or negotiable policies.

Objects provide services. They may use services of other objects to provide their own services. Also, a service may consist of an aggregation of objects, e.g., end-to-end transport service. The aggregated objects may belong to the same or multiple owner- ships. Thus, object composition in Internet 3.0 lies at the basis of the policy and security framework of the architecture.

Like real organizations, realms are organized hierarchically. The hierarchy is not a binary tree since a realm can have two or more parents, i.e., an organization can be a part of several

higher-level organizations and can have several lower-level sub-organizations. Note that the concepts of objects and realms are recursive. An object may comprise a group of objects. Thereby, a realm or a group of realms could be treated as an object and provide a service.

When a realm advertises an object, it keeps complete control over how that object is managed internal to the realm. Inside the realm, the realm members may delegate responsibilities to other objects. This allows the objects to go to sleep for energy saving. It allows specialized services in the realm that can be used by other objects. For example, all packets leaving a realm may be signed by a realm signer that assures that the packets originated from that realm although the source of the packet was not authenticated. In some applications, this type of assurance is sufficient and useful in accepting or discarding the packet.

Each object has an ID and a locator. The ID is unique in the realm and is assigned by the RM. The locator is the ID of the object in the lower tier. Thus, the locator of data is the set of IDs of hosts on which the data resides. The locator of the host is the set of IDs of infrastructure points of attachments to which the host is connected. This separation of ID and locators among multiple tiers is unique and is the basis for allowing independent mobility of users over hosts and hosts over infrastructure. It is also the basis for multihoming of users (a user with multiple host devices such as a smart phone, a laptop, and a desktop).

At the infrastructure tier, the object abstraction framework is implemented through a management and control plane connecting Internet POPs installed with a special Internet 3.0 node called the context router. Fig. 2.24 presents a highly simplified POP design where each AS has a border router that connects to the POP, enhanced with the context router. The context router has two key functions:

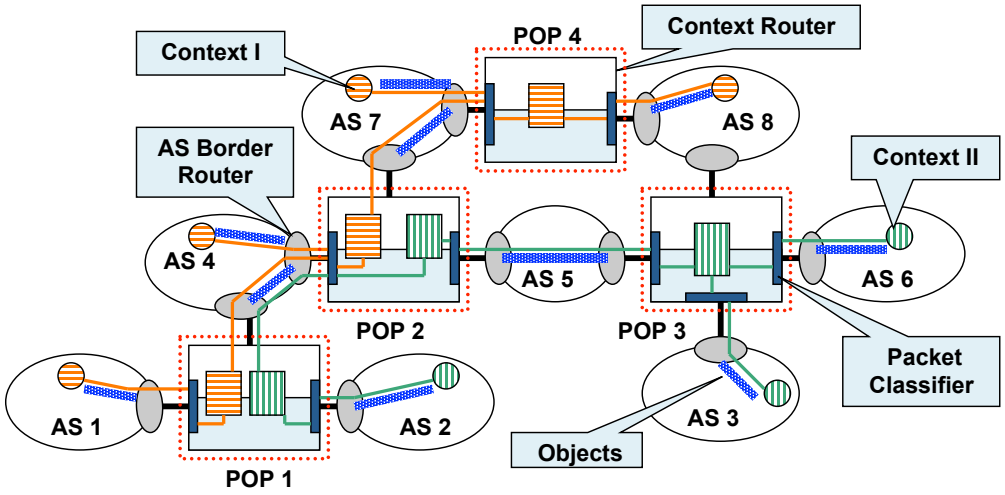

Figure 2.24: Internet 3.0 POPs enhanced with Context Routers

- It maintains a behavioral object repository advertised by the participating ASs and makes them available for lease to application contexts.
- It leases programmable objects provisioned over packet processing hardware resources such as SRAMs, DRAMs, and network processors. This allows application contexts to set-up their own packet processing contexts at POPs (shown as the hatched and dotted contexts).

Fig. 2.25 presents a high-level overview of the context router design. A context router needs to have multiple virtualized contexts advertised as programmable objects. A hypervisor is responsible for creating and controlling these programmable objects. There is a base context called the context 0 that hosts the object store. Participating ASs advertise their objects at the POP and they are stored at the context 0 of the context router. Also, the context 0 participates in the inter-infrastructure realm management plane and stores AS level connectivity maps. It runs a brokering protocol that allows application contexts to query, block, lease and release objects from the object store. A secure software switch allows inter-context communications, mostly to allow application contexts to be able to communicate with the context 0.

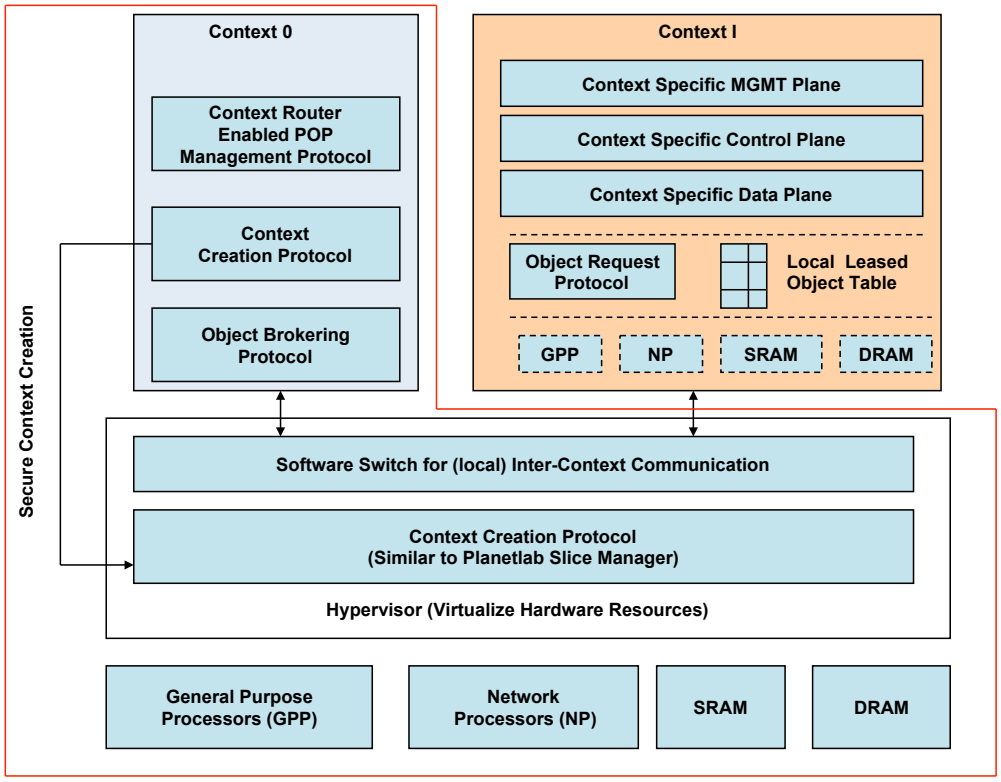

Figure 2.25: Context Router Design

At the host tier,programmable objects are provisioned over compute resources consolidated over end-user personal compute resources, private and public cloud computing and storage re- sources, Content Delivery Network (CDN) storage resources, server farms, grid resources, etc. The mechanisms for sharing common compute resources across multiple application contexts may vary from virtualization techniques achieving near perfect isolation and providing strong deterministic performance guarantees to traditional operating system based resource allocations based on global optimization and fairness considerations. Similar to the infrastructure realm, Internet 3.0 allows complete autonomy to host realms to choose the specific mechanisms for allocation of compute resources to application contexts. Also, it provides a common object abstraction interface that allows host resources to be shared across multiple ownerships over a policy negotiation plane. However, unlike the infrastructure realm which was marked by a physical realm boundary, host realms could have physical as well as logical boundaries. Behavioral objects are pro- visioned similar to the Software-as-a-Service (SaaS) or Platform- as-a-Service (PaaS) paradigms in cloud computing. Security and other services

may be advertised as behavioral objects which advertise the service in terms of abstracted parameters such as security level, etc. An application context should be able to choose the required level of security without worrying about how the end- to-end security service is being orchestrated across the different host realms. The underlying federation mechanism requires considerable efforts in standardization.

Thus, Internet 3.0 is an overarching architecture for the next generation Internet. It identifies the key design basis and defines primitives that shall allow the next generation Internet to be diversified. It significantly improves upon the one-suit fits all paradigm of the current Internet and allows each application context to be able to fully program and optimize its specific context.

2.4.3 Future Internet Architectures (FIA) projects

FIA [41] was the followup to NSFs FIND program. While the FIND program funded a very broad set of research ideas on next generation Internet design, the goal of the FIA program was to consolidate these diverse ideas and move forward towards actually creating a complete top-down architectural framework for the next generation Internet. As part of the program, four research projects were funded with each project granted an approximate amount of $10 million. Here we will provide a brief overview of these four projects to give the reader a flavor of the different ideas that are being considered to evolve the Internet design to serve next generation applications. AppFabric also has a very similar goal and hopefully the discussion in this section will be able to better elucidate the strengths and weaknesses of the AppFabric idea.

- **Named Data Networking:** The Named Data Networking (NDN)[84] project is led by the University of California, Los Angeles with participation from about 10 universities and research institutes in the United States. The initial idea of the project can be traced to the concept of content-centric networks (CCNs) by Ted Nelson in the 1970s. After that, several projects such as TRIAD [21] at Stanford and DONA[69] from the University of California at Berkeley were carried out exploring the topic. In 2009 Xerox

Palo Alto Research Center (PARC) released the CCNx project led by Van Jacobson, who is also one of the technical leaders of the NDN project.

The basic argument of the NDN project is that the primary usage of the current Internet has changed from end-to-end packet delivery to a content-centric model. The current Internet, which is a client-server model, is facing challenges in supporting secure content-oriented functionality. In this information dissemination model, the network is transparent and just forwarding data (i.e., it is content-unaware). Due to this unawareness, multiple copies of the same data are sent between endpoints on the network again and again without any traffic optimization on the networks part. The NDN uses a different model that enables the network to focus on what (contents) rather than where (addresses). The data are named instead of their location (IP addresses). Data become the first-class entities in NDN. Instead of trying to secure the transmission channel or data path through encryption, NDN tries to secure the content by naming the data through a security-enhanced method. This approach allows separating trust in data from trust between hosts and servers, which can potentially enable content caching on the network side to optimize traffic. Fig. 2.26 is a simple illustration of the goal of NDN to build a narrow waist around content chunks instead of IP.

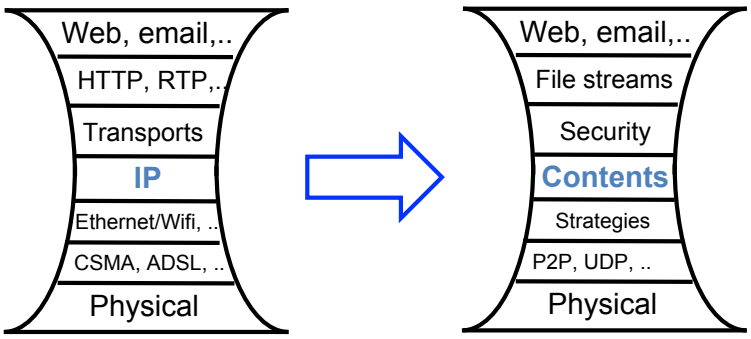

Figure 2.26: The new "narrow waist" of NDN (*right*) compared to the current Internet (*left*)

NDN has several key research issues. The first one is how to find the data, or how the data are named and organized to ensure fast data lookup and delivery. The proposed idea is to name the content by a hierarchical name tree which is scalable and easy to retrieve. The second research issue is data security and trustworthiness. NDN

proposes to secure the data directly instead of securing the data containers such as files, hosts, and network connections. The contents are signed by public keys. The third issue is the scaling of NDN. NDN names are longer than IP addresses, but the hierarchical structure helps the efficiency of lookup and global accessibility of the data. Regarding these issues, NDN tries to address them along the way to resolve the challenges in routing scalability, security and trust models, fast data forwarding and delivery, content protection and privacy, and an underlying theory supporting the design.

- **MobilityFirst:** The MobilityFirst [82] project is led by Rutgers University with seven other universities. The basic motivation of Mobility-First is that the current Internet is designed for interconnecting fixed endpoints. It fails to address the trend of dramatically increasing demands of mobile devices and services. The Internet usage and demand change is also a key driver for providing mobility from the architectural level for the future Internet. For the near term, MobilityFirst aims to address the cellular convergence trend motivated by the huge mobile population of 4 to 5 billion cellular devices; it also provides mobile peer-to-peer (P2P) and infostation (delay-tolerant network [DTN]) application services which offer robust- ness in case of link/network disconnection. For the long term, in the future, MobilityFirst has the ambition of connecting millions of cars via vehicle-to-vehicle (V2V) and vehicle-to-infra- structure (V2I) modes, which involve capabilities such as location services, georouting, and reliable multicast. Ultimately, it will introduce a pervasive system to interface human beings with the physical world, and build a future Internet around people.

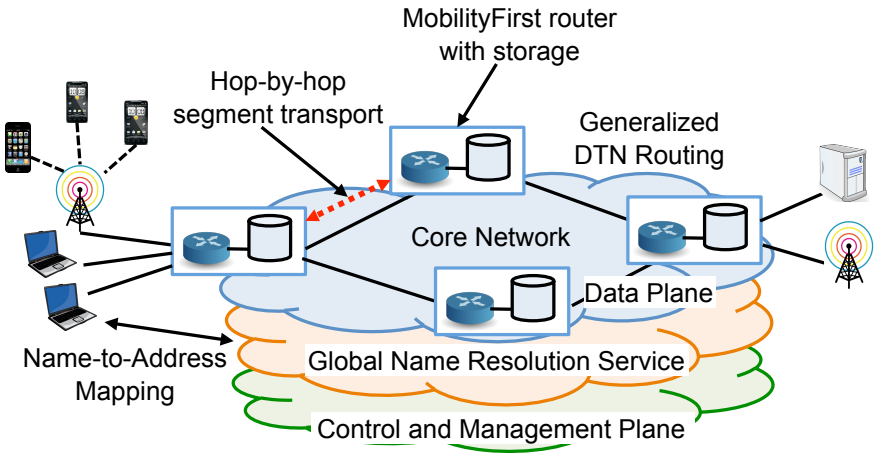

Figure 2.27: MobilityFirst Architecture

The challenges addressed by MobilityFirst include stronger security and trust requirements due to open wireless access, dynamic association, privacy concerns, and greater chance of network failure. MobilityFirst targets a clean-slate design directly addressing mobility such that the fixed Internet (see Fig. 2.27) will be a special case of the general design. MobilityFirst builds the narrow waist of the protocol stack around several protocols:

- Global name resolution and routing service
- Storage-aware (DTN-like) routing protocol
- Hop-by-hop segmented transport
- Service and management application programming interfaces (APIs)

The DTN-like routing protocol is integrated with the use of self-certifying public key addresses for inherent trustworthiness. Functionalities such as context- and location-aware services fit into the architecture naturally.Some typical research challenges of Mobility- First include:

- Trade-off between mobility and scalability
- Content caching and opportunistic data delivery
- Higher security and privacy requirements Robustness and fault tolerance

- **NEBULA:** NEBULA [85] is another FIA project focused on building a cloud-computing-centric network architecture. It is led by the University of Pennsylvania with 11 other universities. NEBULA envisions the future Internet consisting of a highly available and extensible core network interconnecting data centers to provide utility-like services. Multiple cloud providers can use replication by themselves. Clouds comply with the agreement for mobile roaming users to connect to the nearest data center with a variety of access mechanisms such as wired and wireless links. NEBULA aims to design the cloud service embedded with security and trustworthiness, high service availability and reliability, integration of data centers and routers, evolvability, and economic and regulatory viability. NEBULA design principles include:
  - Reliable and high-speed core interconnecting data centers
  - Parallel paths between data centers and core routers
  - Secure in both access and transit
  - A policy-based path selection mechanism
  - Authentication enforced during connection establishment

  With these design principles in mind, the NEBULA future Internet architecture consists of the following key parts:
  - The NEBULA data plane (NDP), which establishes policy-compliant paths with flexible access control and defense mechanisms against availability attacks
  - NEBULA virtual and extensible networking techniques (NVENT), which is a control plane providing access to application- selectable service and network abstractions such as redundancy, consistency, and policy routing
  - The NEBULA core (NCore), which redundantly interconnects data centers with ultra-high-availability routers

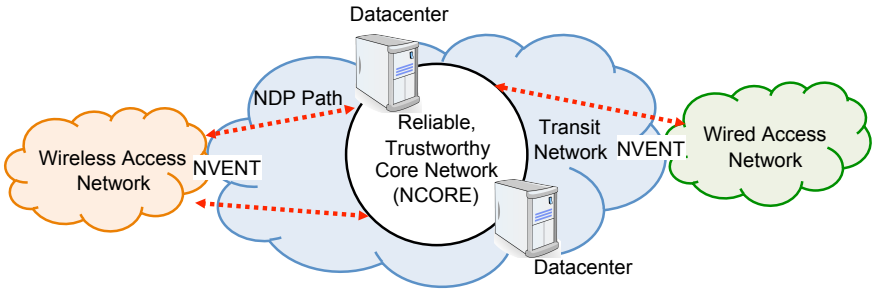

Figure 2.28: NEBULA Architecture

NVENT offers control plane security with policy-selectable network abstraction including multi-path routing and use of new networks. NDP involves a novel approach for network path establishment and policy-controlled trustworthy paths establishment among NEBULA routers. Fig. 2.28 shows the NEBULA architecture comprising the NDP, NVENT, and NCore, and shows how they interact with each other.

- **eXpressive Internet Architecture (XIA):** Expressive Internet Architecture (XIA) [141] is also one of the four projects from the NSF FIA program, and was initiated by Carnegie Mellon University collaborating with two other universities. As we observe, most of the research projects on future Internet architectures realize the importance of security and consider their architecture carefully to avoid the flaws of the original Internet design. However, XIA directly and explicitly targets the security issue within its design. There are three key ideas in the XIA architecture:
  - Define a rich set of building blocks or communication entities as network principals including hosts, services, contents, and future additional entities.
  - It is embedded with intrinsic security by using self-certifying identifiers for all principals for integrity and accountability properties.
  - A pervasive narrow waist (not limited to the host-based communication as in the current Internet) for all key functions, including access to principals, interaction among stakeholders, and trust management; it aims to provide interoperability at all levels in the system, not just packet forwarding.

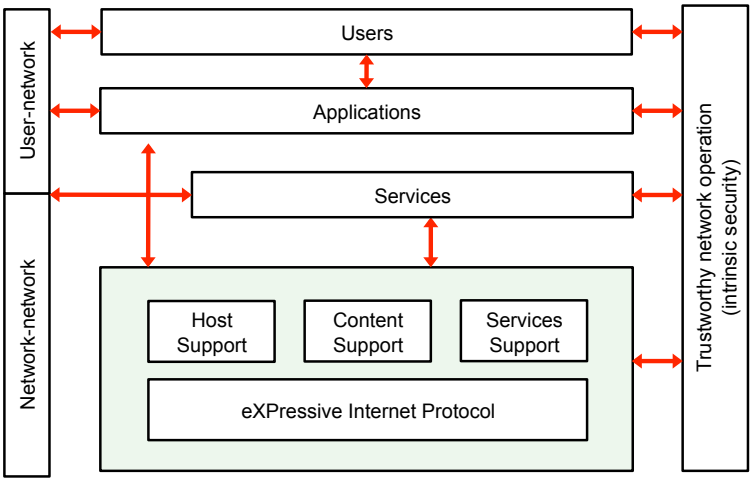

Figure 2.29: XIA Components and Interactions

The XIA components and their interactions are illustrated in Fig. 2.29. The core of the XIA is the Expressive Internet Protocol (XIP) supporting communication between various types of principals. Three typical XIA principal types are- content, host (defined by who), and service (defined by what it does). They are open to future extension. Each type of principal has a narrow waist that defines the minimal functionality required for interoperability. Principles talk to each other using expressive identifiers (XIDs), which are 160 bit identifiers identifying hosts, pieces of content, or services. The XIDs are basically self-certifying identifiers taking advantage of cryptographic hash technology. By using this XID, the content retrieval no longer relies on a particular host, service or network path. XIP can then support future functions as a diverse set of services. For low-level services, it uses a path-segment-based network architecture (named Tapa in their previous work) as the basic building block; and builds services for con- tent-transfer and caching and service for secure content provenance at a higher level. XIA also needs various trustworthy mechanisms and pro- vides network availability even when under attack. Finally, XIA defines explicit interfaces between network actors with different roles and goals.

2.5 Other Related Work

The design of AppFabric borrows from a large number of previous and current researches on network architectures. AppFabric extends some of these ideas and combines several disparate key concepts to realize its goals of an open network architecture for service/application delivery.

- **Content Distribution Networks (CDN):** Unlike CDNs, AppFabric is designed for highly dynamic data that cannot be cached; or, when the data owner wants to keep complete control of their data and the data generation processes and cannot trust the CDN with full view of the data, e.g., in medical, banking, and defense applications.

- **Active Networks:** In active networks [132], routers are required to execute code in packet headers. The idea was never adopted by industry because no service provider will allow others code to run on its routers thereby relinquishing control of its routers to others. AppFabric is specifically designed to avoid this flaw. ISPs are always in complete control of their network. ISPs own the policies and control the translation and implementation of ASP specified rules in the forwarding plane.

- **Rule-Based Forwarding (RBF):** In RBF [111], which is also designed to avoid the flaws of active networks, packets carry rules that are processed in the forwarding plane to determine an action on the packet. Four possible actions are defined. These are: 1) drop the packet, 2) forward the packet over the underlying IP forwarding plane, 3) invoke a local router function, and 4) update an attribute in the packet. Again, the users directly control the ISPs routers behavior which may not be acceptable to the ISP. In contrast, packets do not carry any rules in the AppFabric data plane. All they will have is the application meta-tag which will allow the ISPs to handle the packet as agreed with the ASP in the control plane. Second, in the AppFabric data plane, the forwarding actions may change as soon as the state of the application server or the network changes. In RBF the receiver entity (that specifies the rules) cannot change the way packets are processed until the communicating entity asks for the rules again or the rule lifetime ends. The third difference is that the AppFabric data plane allows specifying forwarding rules based upon application-level semantics such as packet flows, messages and application sessions while RBF is simply packet based. The flow-based

design allows a compute once, use many-times design for the forwarding plane. There are several other subtle differences between these two architectures. We believe that AppFabric is a simple, robust, flexible, scalable, and more easily implementable and deployable design.

- **Serval:** Serval [90] is another recent work addressing the problem of accessing geographically distributed services. Serval provides a point solution for accessing distributed services through a service router infrastructure. The service router infrastructure is similar to the requirement of outsourcing application-level routing in the AppFabric data plane. However, AppFabric provides 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 includes the application context into the switching abstraction. Other distinguishing features include support for switching across middlebox sequences both in the data and control planes and support for both sender and receiver policies.

- **Delegation Oriented Architectures:** Delegation based architectures such as DOA [139] proposed a mechanism for off-path, middlebox deployments. AppFabric provides a more fundamental primitives that will allow a more realistic approach to realize the requirements of application deployments.

- **ALTO:** More recently, the ALTO [56] working group at IETF is developing an application-layer traffic optimization service that will provide network information to applications to inform peer selection for P2P services, content delivery, mirror selection, etc. Network information provides only one set of parameters among others (cost of deployment, usage patterns, application replication and partitioning) that inform application delivery policies in AppFabric. The ALTO architecture provides a subset of the services provided by AppFabric.

  There have been several research efforts to motivate architectural changes to address the problem of adding explicit support for middleboxes into the Internet architecture. However, the non-standard network configuration techniques and support for forward and reverse proxies in HTTP somehow managed the show and curbed the motivation for adopting a general architectural change. We believe that in the context of cloud computing the problem is more urgent and different as application deployments move to third party infrastructures and have more dynamic and distributed deployments. 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. Also, none of them offer any discussion on the control plane; and how application level policies may be enforced on third-party infrastructures.

In this chapter, we provided some background that will help the reader appreciate the motivations that led to the different design choices in AppFabric. In the rest of this thesis, we will discuss the specific details of the AppFabric architecture, the design details and a proof-of-concept prototype implementation of the platform.

Chapter 3

AppFabric High-level Architecture

In this chapter we will present a high-level architecture of the AppFabric platform. The design of AppFabric has been motivated by the need to address some general issues and architectural deficiencies in current application delivery solutions. It is especially important that we address these issues now, in light of the recent advances in infrastructure virtualization technologies and a shift towards Software-Defined Infrastructures (SDI). We will discuss these issues and how AppFabric addresses them in more detail in this chapter.

The discussion in this chapter is divided in two parts. In the first part, we will discuss some of the high-level requirements that motivate our design. In the second part, we will discuss the high-level goals of the AppFabric architecture.

3.1 High-level Ideas

In this section, we will present some of the high-level requirements that motivate our design of AppFabric.

3.1.1 Horizontal integration Platform

Delivering modern applications is extremely complex and requires the integration of many different components. System engineers strive to achieve this extremely complex task through

a combination of manual ad-hoc techniques, automation tools and mostly proprietary middleware solutions. Most of these tools, techniques and software achieve vertical integration (across the same or similar type of component) leaving the deployment teams to figure out how to achieve horizontal integration. This gap is often the source of most fault-lines and missed opportunities. It is all the more pressing to address the issues of horizontal integration now, given that SDI enables new infrastructure usage capabilities and next generation applications need to leverage these capabilities to enable new use cases. The benefits of a horizontal integration platform, providing the proper abstractions through a well designed and well implemented API set is indisputable. In AppFabric, we strive to provide such a platform for application delivery. To give a flavor of what horizontal integration AppFabric is trying to achieve, consider the following(refer to Fig. 3.1):

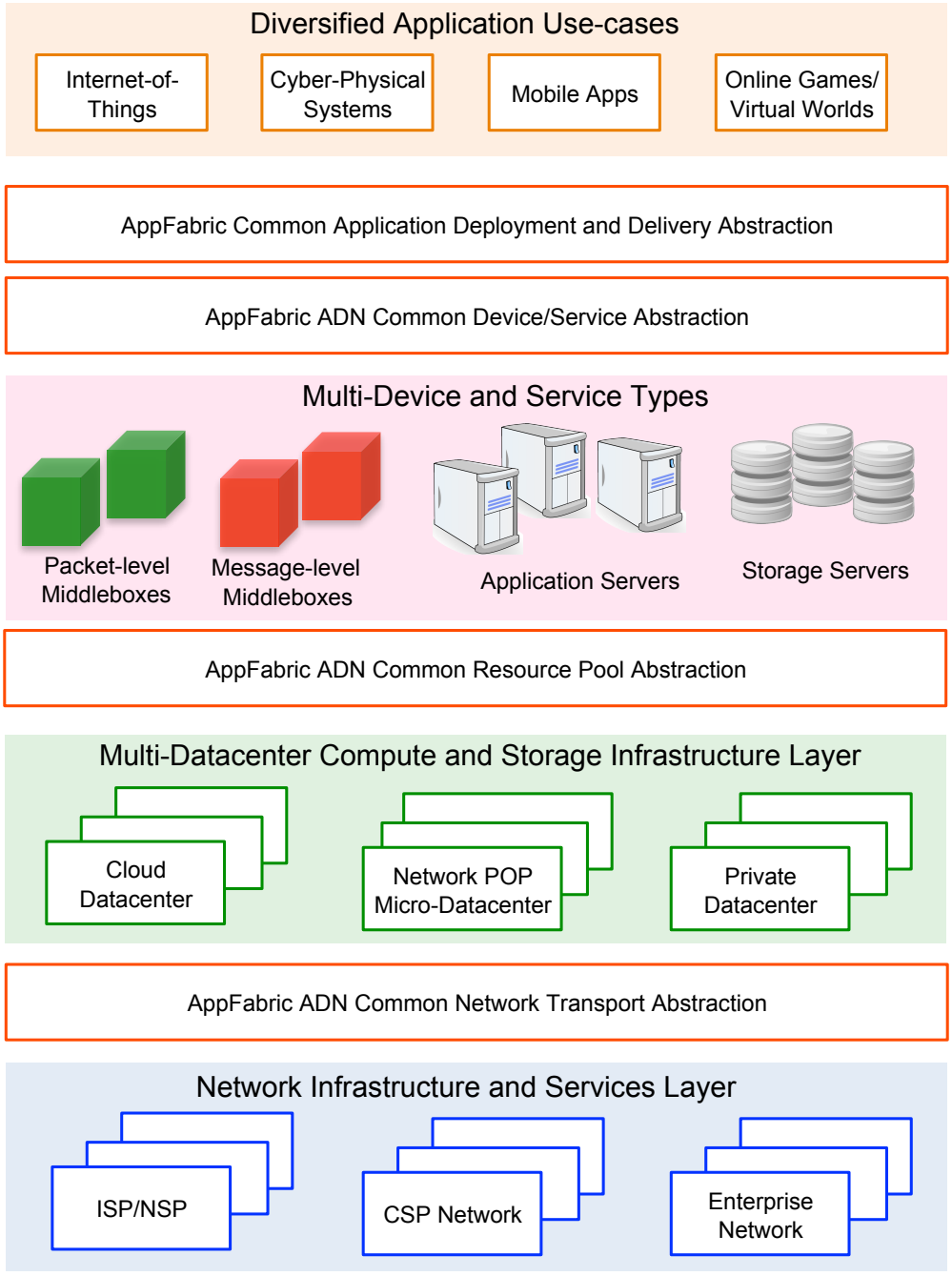

Figure 3.1: Schematic Representation of an Application Delivery Network (ADN)

- **Integration of network and compute infrastructures:** Application delivery depends on the capabilities of both the compute (including storage) as well as the network

infrastructures. However, traditionally these two infrastructure components have been largely kept separate. Current networks, even within the administration of the same Application Service Provider (ASP), for example in enterprise datacenters, are seldom application aware. Applications are sometimes designed to be network aware. For example most media transmission applications dynamically adapt to changing network conditions by using different media encoding techniques. However, this seems to be a reversal of roles from an architectural standpoint. Contrary to the original design goal, in this case the compute layer seems to be providing a service to the network layer to cope with its problems of failure or congestion by degrading itself. Such issues of tussle between the network and compute layers have existed since the very beginning and there have been many efforts to address them as well. In-fact network QoS has been the one of the most prolific areas of research in Computer Science in the past two decades [29], resulting in almost no real-world deployments. We think that the problem is that although we know the techniques of creating a differentiated network infrastructure for different types of applications, we do not have a good interface through which the application can explicitly (and dynamically) specify its requirements to the network; allowing the network to enable the required service for the application. SDI stands to change this situation. SDI virtualizes both compute and network resources alike and provides access to each of these different resource types through a service interface. AppFabric is designed to be the horizontal integration platform that sits on top of the SDI layer and translates the applications requirements (both in terms of compute and the network) to the capabilities offered by the SDI components.

- **Integration of different device types:** One of the problems with current application delivery environments is *device sprawl* enterprise datacenters house a variety of different types of devices including application servers, storage appliances, network equipment, and packet-level and message level middleboxes; each of them coming from many different vendors, providing different functionalities, having different deployment requirements and each having separate management and control interfaces. AppFabric provides a common data, management and control framework for all these different types of devices. Each of these different types of devices can connect to each other over a common AppFabric data plane, and can be monitored, managed and controlled through a common AppFabric control/management plane interface. Thus, AppFabric provides a horizontal integration platform across many different device types similar

to the way SDN seeks to provide horizontal integration across different switching-layer devices from many different vendors.

- **Integration of different transport mechanisms:** : In AppFabric, application and network services do not need to worry about how they communicate with each other. The platform intelligently deploys the services and provides a common interface for them to communicate with each other. Underneath this common interface, the platform will choose from different transport mechanisms including network sockets, inter-process communication (IPC), inter-thread communication and IP packet forwarding to provide the most efficient way to enable the communication.

- **Integration of multiple resource providers:** AppFabric also provides horizontal integration across multiple resource providers, providing different types of resources and implementing different SDI management stacks (e.g. OpenStack, CloudSack, OpenDayLight, etc.). Thus, application deployments over AppFabric can span across multiple Cloud Service Providers (CSPs) and Network Service Providers (NSPs), thus providing the perfect platform to deploy massively distributed application use cases.

Thus, one of the primary architectural goal of AppFabric is to cut across the different vertical components that make the current application delivery process extremely complex, fragile and unwieldy and simplify it as much as possible.

3.1.2 Separation of control and data planes

Control and data plane separation is an architectural style that has received much attention in the recent past in the context of discussions related to Software-Defined Networking (SDN). However, in the ensuing hype it has come to mean different things to different people. The key confusion emerges from the arbitrary understanding as to what is meant by the term ***separation*** in this context. Control, management and data planes have always been logically separate, and thus the different names. In the present context, however, the word separation entails two concepts:

- **Logically centralized control plane and distributed data plane:** The logical vs. distributed argument is not new and we have seen system design trends swing back- and forth across both these extremes. There are strong arguments both, in support of and against each side. Distributed systems are good for robustness (no-single point of failure) and incremental scalability (adding more resources in smaller increments). Also, generally, it is less expensive to build a powerful system from many less capable commodity parts than to build a single high capacity system from specialized hardware. However, just like any other aspect of system design, these advantages trade-off with the increased complexity in managing and controlling a distributed system. Centralized systems, on the other hand, are much easier to manage and control but have disadvantages in terms of robustness, incremental scalability and cost. However, many times, the very nature of the system deployment itself, force system architects to choose one over the other. Computer networks are a typical example where the system itself is inherently distributed. Therefore, while these computer network designs have been extremely robust and scalable, managing them has been extremely challenging. The complexity of distributed control algorithms has often been the bottleneck in introducing new services and evolving to new requirements, troubleshooting problems when the system does fail in-spite of the robustness, and dealing with configuration and algorithmic errors. SDN introduced the notion that there was actually no need to club the deployment of control and data plane functions together since they had very different requirements. The data plane that is responsible for the actual movement of the bits needs to be extremely robust and very efficient. The control plane on the other hand can afford to be much less efficient (although it needs to be equally robust). At the same time, there are enormous gains in centralizing the control plane and getting rid of complex distributed control algorithms. Therefore, while the data plane functions (packet forwarding) can be distributed, the control plane functions (computing the routing tables) can be centralized.

- **Decoupling the implementation of data plane and control plane:** This hybrid architectural notion of centralized control plane and distributed data plane was already proposed earlier in the context of centralized route servers [15] and MPLS path computation elements [9, 74]. SDN's contribution was to make a more compelling business case for it through the idea of completely de-coupling the implementation of the data plane and the control plane functions and defining a standard, open interface through

which a control plane element can talk to a data plane element and vice-versa. This has a tremendous impact in opening up the system for multi-vendor innovations by making sure that devices from multiple vendors can now be easily interoperate. For example, a network provider may now buy switches from multiple vendors and buy a controller from a completely different vendor, as long as they all implement this standard interface. Vender lock-ins has been one of the reasons for slow innovations in the networking industry and SDN has definitely helped in addressing this impasse.

Given this framework for control and data plane separation, we argue that it is equally relevant to an application delivery platform such as AppFabric as it is to SDN. Let us look at the specific requirements of each of these three planes in AppFabric in order to justify our design choice.

- **Need for a distributed data plane in AppFabric:** AppFabric is an application delivery platform and hence comprises of two key components - the **application** and the **network**. Modern applications, unlike large monolithic application deployments in the Mainframe-era, are mostly distributed. Let us look at the different contexts in which these application deployments are distributed:
  - *Service-Oriented Architecture (SOA) based applications:* The application world has been trying to evolve to a service-oriented application design architecture. In SOA, applications are composed of many loosely connected components called *services*. SOA contributes enormously to simplifying each stage of the application development process including planning, design, development, testing and maintenance. Apart from this, SOA enables sharing of data and capabilities among different application providers allowing innovative new use-cases through collaborations and partnerships and also opening new avenues for monetization of their resources. As obvious, SOA-based applications are inherently distributed.
  - *Moving towards commodity hardware:* Commodity server hardware has the same benefits to application delivery as commodity switching hardware has to SDN. Modern datacenters comprise of lots of cheap commodity servers replacing specially designed, high-performance servers. Doing so ensures high utility, lower capital and operation expenditures, easily migrating workloads to third-party

leased infrastructure and avoiding vendor lock-ins. Most modern application frameworks, such as Map-Reduce and Big-Data databases such as the Hadoop Distributed File System (HDFS) [12], are designed to promote this trend by distributing compute/storage across many low capability commodity servers.

– *Massively distributed application use-cases of the future:* Most modern and future application use-cases including Internet-of-Things (IoT), Cyber-Physical Systems, mobile apps, online gaming, and virtual worlds benefit from having a distributed geographical presence. Therefore, distributed application deployments will no longer be limited to many hosts in a datacenter but will require many hosts in many datacenters.

Note that this does not automatically imply that the AppFabric data plane should be distributed. As we mentioned earlier, AppFabric consists of two key components - the network and the application. Therefore, there are two considerations here that are important.

– The network is by its very nature distributed. Currently, both the control and data plane of the network are distributed. SDN is trying to advocate centralizing the control plane while keeping the data plane distributed.

– The backbone of distributed service-oriented architecture platforms is a middleware tier, generically called the Enterprise Service Bus (ESB)(described in Chapter. 2). It was also described in Chapter 2 that ESBs can be implemented either through a *centralized* or a *distributed* message broker subsystem. A centralized broker subsystem suffers from the usual drawbacks of centralized architectures such as introducing a single point of failure and performance bottleneck chokepoints. However, these issues can be easily mitigated through techniques such as adding redundancy in the form of a hot standby and operating a cluster of smaller nodes instead of a single large node. The bigger issue is that a centralized implementation of an orchestration platform is inefficient owing to the overhead due to indirection - every communication between any two entities will need to be in-directed through the orchestrator (single node or a cluster). This cost of indirection is especially unacceptable for application use-cases where the application deployment may be distributed across multiple datacenters. There is no

doubt that AppFabric requires a distributed solution. However, the current distributed broker subsystem solutions also have their own limitations. Firstly, they are difficult to configure, control and deploy. Secondly, most distributed ESB solutions are limited to a few high-performance message broker nodes that serve hundreds of services. Thirdly, we are not aware of any middleware platform that can dynamically scale and distribute, on demand, to accommodate the changes in user access patterns and distribution. Therefore, there is a clear motivation for AppFabric to adopt a *hybrid* architecture where the control plane is logically centralized and the data plane is distributed. Such an architecture will allow the message broker subsystem to be easily configured, controlled and deployed. Also, it avoids the inefficiencies (as a result of indirection in message routing in the data plane) of a centralized solution. Also, since the control plane is centralized, it is now possible to operate a *massively* distributed data plane, one that matches the deployment of the application services. As we will see in the next chapter, the message brokering subsystem in AppFabric does not consist of any special or separate broker nodes but is distributed across all the compute and storage nodes in the application deployment.

Therefore, both the application and the network architectures converge to a hybrid *centralized control plane, distributed data plane* architecture for better efficiency, more dynamism and scalability, and easier management and control. However, in AppFabric, the control plane is not exactly centralized (physically or logically) but is instead hierarchical (converge to a centralized entity slowly). The next point discusses the motivation behind this design choice more elaborately.

- **Need for a hierarchical control plane in AppFabric:** The idea of a centralized control plane overlooks the concerns of robustness and scalability that also affect the control plane, albeit to a much lesser extent than the data plane. Of these, the robustness issue can be easily solved by adding redundancy. For example, by adding a hot/cold standby. It is more challenging to address the scalability issue. Also, the problem is more severe if the data plane is dynamically programmed by the control plane, as was proposed in the original OpenFlow architecture [78, 46]. Any packet that could not be classified to one of the flow entries in the data plane would be forwarded to the controller that would then program the data plane nodes to handle this new flow.

The scalability of such an architecture is restricted in two ways. Firstly, it depends on the number of new flows that arrive at each instance. Secondly, the delay between the arrival of a new flow and the time it takes by the centralized controller to program the data plane to handle it, needs to be bounded. This problem of delay may not be much of an issue in datacenter environments, but certainly becomes much more relevant in wide-area network environments or multi-datacenter environments. We address it by treating the control plane as a sort of intermediary between the data plane and the management plane (as shown in Fig. 3.2); where the data plane is fully distributed with a flat structure (where all the nodes are peers) , the management plane is logically centralized, and the control plane is distributed but with a hierarchical structure. Each data plane node has a controller agent that reports to the local datacenter controller, that reports to a zonal (multiple datacenters in a given geographical region) controller and so on. Finally, at the top of the hierarchy is a central global controller that in turn talks to the centralized management plane element, discussed next. At each level in the hierarchy there is aggregation of information. For example, the local datacenter controller only reports the aggregated resource available at the datacenter to the zonal controller instead of reporting per-host resource availability.

- **Need for a centralized management plane in AppFabric:** The AppFabric platform is dynamic in the sense that it can adapt in the runtime to a set of pre-defined contexts. For example, it can allocate/release resources and replicate application workflows, on demand. However, unlike the OpenFlow architecture, it does not expect a new context (for example a new flow) to show up during runtime. All possible contexts need to be pre-specified (or negotiated) through a centralized management plane that then programs the control plane and data plane to handle it in the runtime. In the runtime, the role of the management plane is to oversee that the policies are being enforced properly and record non-compliance.

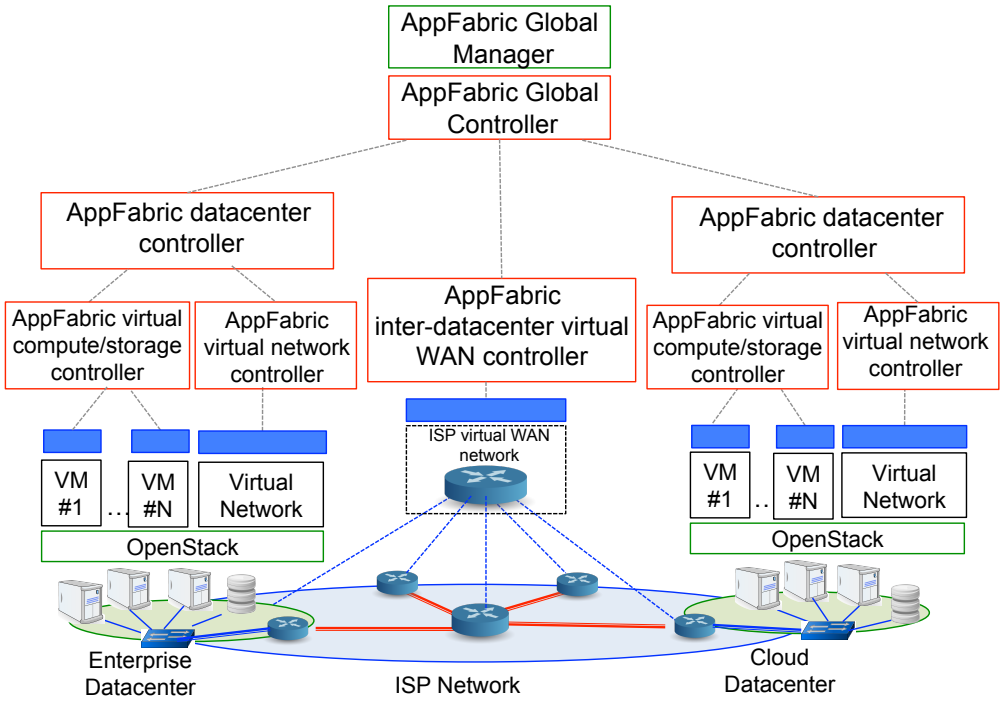

Figure 3.2: High-level Architecture of AppFabric showing the distributed data plane, hierarchical control plane and centralized management plane

Fig. 3.2 shows a high-level representation of the organization of the data plane, control plane and management plane of AppFabric. As can be seen in this figure, the AppFabric data plane is distributed over virtual machines and virtual networks. These virtual machines may be acquired from the enterprise-owned datacenters or leased from one or more cloud providers. They host different types of services including application services, storage services, packet-level middleobox services and message-level middlebox services. Similarly, the virtual network may be owned by the enterprise, as in the case of virtual network in enterprise datacenters or corporate WAN infrastructrues (such as the Google WAN [49, 50]) or, may be leased from a cloud provider (virtual network inside a cloud datacenter) or a ISP (virtual leased WAN services). Each of these data plane resources, either owned or leased is managed, controlled and programmed by the AppFabric control and management plane. The control plane is hierarchical, with a separate controller for each resource provider. Also, each of these controllers are further logically sub-divided based on whether they are controlling virtual machines (with different types of services) or a virtual network. These

local controllers, in turn, are managed by a global controller. Note that there may be more levels in the hierarchy. The decision on how many levels of controllers are needed is based on addressing the tradeoff between the advantages of having more levels (such as local decision making that reduces the delay between the control and data plane and localizing problems making it easier to isolate and debug failures) versus the disadvantages (such as increased complexity of managing the control plane itself as a result of more distributed control functions requiring synchronization). The management plane is centralized and we have already argued the rationale behind it in the preceding discussion. The details of each of these data, control and management plane entities will be provided in the chapters to follow.

3.1.3 ID/Locator Split

AppFabric provides a service-centric infrastructure platform to application deployments as opposed to the host-centric platform provided by the Internet. This means that a service is independently identifiable and addressable within the platform and is not coupled to a host. The service to host mapping is dynamic and maintained by the platform. The general architectural idea behind this is to try and achieve ID/locator split.

Over the past few years there has been a lot of research on ID/locator split architectures (*see the detailed discussion on this in chap. 2*), which intend to decouple a hosts locator (IP address) from it identity. The coupling of the ID to a locator is the reason behind a lot of problems faced by application deployments on the current Internet. We too have made some significant contribution to this research [105, 99, 97, 98, 100, 124, 125, 96, 103, 106]. However, decoupling the identity from a locator of a host still results in a host-centric architecture. A service is still bound to the hosts ID. The implication of this is that one can only specify and enforce policies at the host-level and all services that are running on that host need to automatically inherit those policies. Also, a service is not allowed to move to a new host if required; as may be required to optimize inter-service communication, service composition and handle host failures. To address this problem, we had proposed a multi-tier ID/locator split architecture called PONA- Policy-Oriented Network Architecture (Fig. 3.3). PONA proposed a three-tier architecture with users and services in the top tier, hosts forming the middle tier and the network in the bottom-tier. Network interfaces would be identified through a locator (IP or MAC) whereas the higher tier entities (hosts, services,

and users) would have globally unique identifiers. An ID to locator mapping plane between each tier would dynamically map the ID of an entity of a higher tier to the id of an entity in the tier below it (*e.g.* $Service\,ID \rightarrow Host\,ID \rightarrow Network\ interface\ \langle IP, MAC \rangle$). This would make each of these entities independent of each other while still maintaining the tiered dependency that is inherent in the architecture.

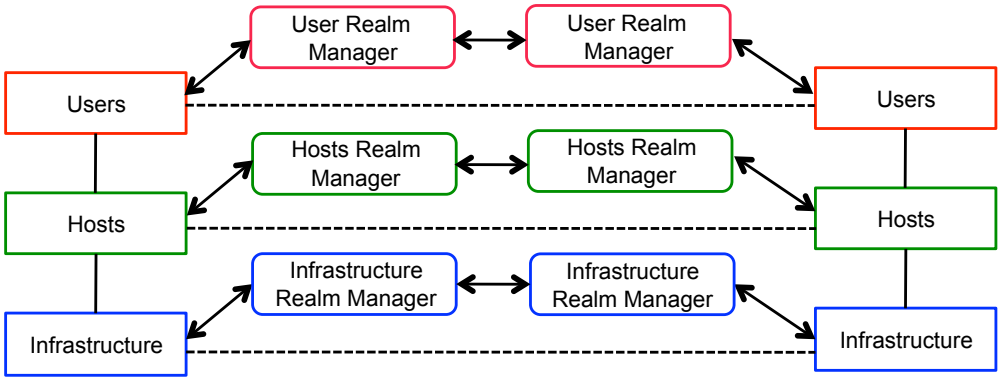

Figure 3.3: PONA is part of the Internet 3.0 Architecture (redrawn from Fig. 2.23)

The PONA solution was proposed as a clean-slate architecture for the next generation Internet. In AppFabric, we take much of PONAs ideas but instead of trying to change the Internets architecture natively (that is by changing the network stack) we provide a service-centric communication abstraction as an overlay over the original host-centric design.

3.2 High-level Goal

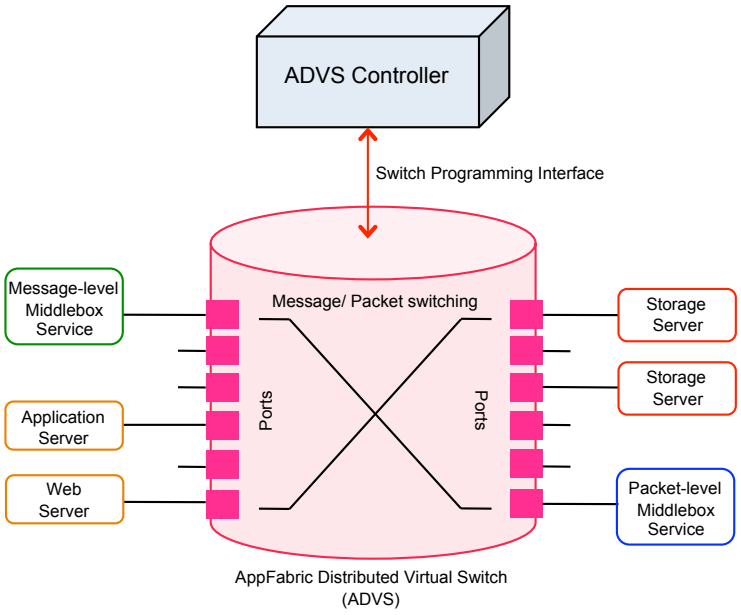

Figure 3.4: AppFabric Distributed Virtual Switch

The high-level goal of the AppFabric architecture is to design a AppFabric Distributed Virtual Switch (ADVS), as shown in Fig. 3.4. Some of the properties of this switch are:

- **Distributed:** The ADVS provides the abstraction of a single centralized switch to the user while the actual implementation is distributed across all the different components such as network nodes, middleboxes and application servers, that make up an application delivery network. The different components of the switch may be distributed across multiple different resource provider infrastructures across many different geographical sites.
- **Message/Packet level switching:** Both, message and packet level services should be able to connect to this switch and the switch should be capable of switching both messages and packets.
- **Programmable:** The switch should be programmable to allow the application to program it according to its requirement.

- **Application routing:** Unlike simple destination-based routing in IP, routing in the ADVS is based on application-level data and context. The destination of a message or a packet is determined dynamically at the switch based on application-level classification rules.

There are lots of design issues in designing the ADVS and we will discuss these issues and how we address them in Chapter. 4, Chapter. 5 and Chapter. 6.

In this chapter we presented a high-level architecture of AppFabric and tried to motivate some of the key design choices. In the rest of this thesis, we will provide a more elaborate and specific discussion on the design of the AppFabric data, control and management planes.

Chapter 4

OpenADN: The AppFabric Data Plane

In this chapter, we will discuss the architecture of OpenADN: the data plane of AppFabric. OpenADN is the abbreviation for Open Application Delivery Networking. An application delivery network or ADN is a network comprising of both, application-layer components and network-layer components, that are required for deploying and delivering applications. Application-layer components are those that need to access and act upon application-layer data whereas network-layer components are those that need to access only network layer (Layer 3 and below of the OSI stack). application-layer components may be further classified into message-level components and packet-level components. Message-level components process application messages for which they need to re-assemble network packets into application messages. Examples of message-level components include application servers such as web servers, database servers, etc. and message-level middleboxes such as Web-Application Firewalls (WAF), application transcoders, SSH off-loaders, etc. Packet-level components work at packet-level granularity but still need access to the application data carried in each packet. Examples of packet-level components include intrusion detection and prevention systems(IDS/IPS), packet scrubbers, WAN optimizers, etc. On the other hand, network-layer components perform only packet-level functions such as packet routing and forwarding and therefore need access to only Layer 3 (and below) packet headers. OpenADN, as the name suggests, is an open standard for creating, managing and controlling an application delivery network.

The discussion in this chapter is organized in two sections. In the first section, we will lay down the architectural requirements that OpenADN needs to satisfy and try to motivate these requirements. In the second section, we will discuss the OpenADN architecture in light of how it actually addresses these requirements.

4.1 OpenADN: Architectural Requirements

The OpenADN design implements the switching substrate of the AppFabric Distributed Virtual Switch (Fig. 4.1) or ADVS as discussed in Chapter. 3.

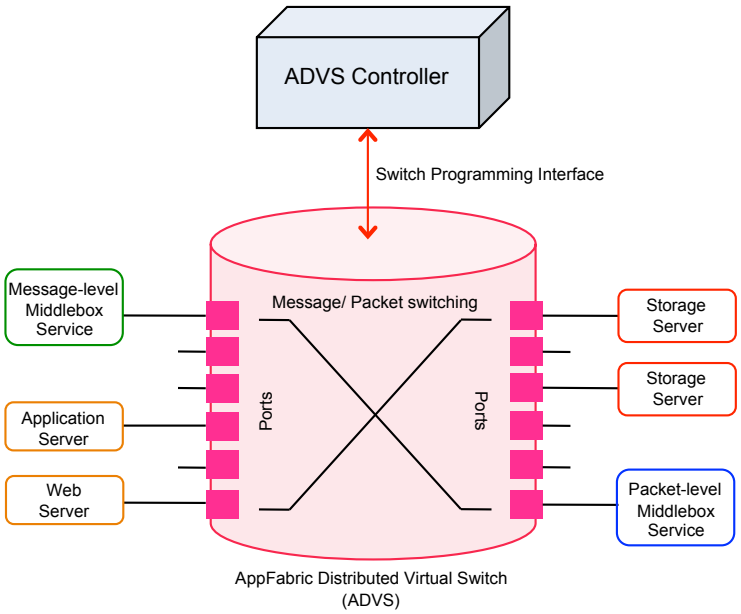

Figure 4.1: AppFabric Distributed Virtual Switch (reproduction of Fig. 3.4)

The design of the ADVS switching substrate has three high-level architectural requirements - **Integration, Programmability and Distribution**. Let us discuss each of these requirements in some more detail.

- **Integration:** OpenADN needs to implement the horizontal integration platform goals of AppFabric discussed in Chapter. 3. These include:

– **Integration of the compute and network infrastructures:** One of the key motivations behind creating an integrated application delivery network is to improve the co-operation between the network infrastructure and the application infrastructure so that future applications can be more network aware and future networks are more application aware. Traditionally, the network infrastructure has mostly been application-unaware. A lot of work has been done on network quality-of-service but none of it has found its way to actual deployment. The Internet today is still largely best-effort. Enterprises and applications that need any sort of transport guarantees can get it only thorough static pre-provisioning of resources between the fixed enterprise locations such as branch offices and the central head quarters. Another technique used by some applications is to create smart overlays that can optimize delivery over the underlying best-effort networks. However, the overlay traffic management policies often interact negatively with the underlay traffic management policies, creating tussles that are bad for both parties. Nonetheless, it is much easier for applications to get their desired quality of service by operating their own private networks rather than depend on a shared network infrastructure such as the Internet. In-fact, for this reason, some enterprises, with very distributed geographical footprints such as Google and Microsoft have moved towards bypassing most of the shared Internet and instead operate their own private networks; both, to connect their geographically distributed datacenters as well as to connect users to their applications. The problem with this approach is that it is not possible for smaller enterprises to take this approach owing to prohibitively high costs of owning and managing their own network infrastructures. Also, as enterprises move towards more and more shared computing platforms (cloud datacenter), they also need a shared network infrastructure that can satisfy their requirements. Here we not only mean shared WAN infrastructures but also shared datacenter networks inside a cloud datacenter that is shared by multiple tenants. We believe that there were two big hurdles in the way of creating application-aware shared network infrastructures:

  * Complex and distributed network control planes made it difficult to dynamically and optimally allocate/deallocate network capacity to traffic demands.
  * There is no standard interface through which applications can automatically and dynamically communicate their requirements to the network.

While the first problem has been solved (or will eventually be solved) by Software-defined Networking, OpenADN solves the second problem, partly, while the other part is solved by Lighthouse, which is is control and management plane of AppFabric.

– **Integration of different types of application-level device/service types:** An ADN comprises of different types of application-level devices and services including packet-level middleboxes, message-level middleboxes, and application servers. The ADVS device needs to provide a common switching substrate for each of these different device and service types.

  * **Challenges in integrating packet-level middleboxes with ADNs:** Middleboxes are a key component of modern application delivery environments providing essential security services (e.g. Intrusion detection and Prevention Systems (IDS), packet scrubbers, Web-application firewalls, etc.) and improving performance (e.g. load balancers, content caches, WAN optimizers, SSL off loaders, transcoders, etc.). However, in spite of their functional relevance, middleboxes are considered architecturally harmful. This is because middleboxes do not fit well in the end-to-end application delivery design of the current Internet architecture. The key issue is that there is no support for off-path deployment of packet-level middleboxes: The Internet Protocol or IP was designed to route packets between a source and a destination (not considering IP multicast). Initially, it had no support for explicit hop-by-hop forwarding. Therefore, it was not possible to deploy middleboxes off the IP routed path between the source and destination. Although, subsequently IP added source routing (strict and loose), it was rarely used. It had considerable security issues allowing a sender to easily spoof its address but still be able to receive the reply packets. Also, it required support from the intermediate IP routers. Instead, system administrators have used ad-hoc manual techniques for deploying middleboxes. These include, physically placing a middleboxes on-path, or manually tweaking path selection mechanisms of underlying protocols such as Layer 2 spanning tree protocol of Layer 3 OSPF protocol to make sure certain paths are always in-directed through the middlebox. However, such techniques are extremely prone to configuration errors or may not work well given the dynamic nature of routing algorithms. Moreover, the

task becomes exponentially harder when more than one middlebox needs to be deployed. Therefore, as shown in Fig. 4.2, current datacenter administrators mostly bundle all the middleboxes in the aggregation-layer. North-south traffic (packets travelling in and out of the datacenter) is forced to travel through these middleboxes while east-west traffic (between two servers) may be forcefully routed through the aggregation layer if required. This arrangement, however quirky and inefficient does work and there may not be enough motivation for it to change. In any case, there have been several proposals addressing this particular issue [139, 53, 65]. Our interest in revisiting the middlebox deployment issue concerns two recent developments; cloud computing and software-based appliances.

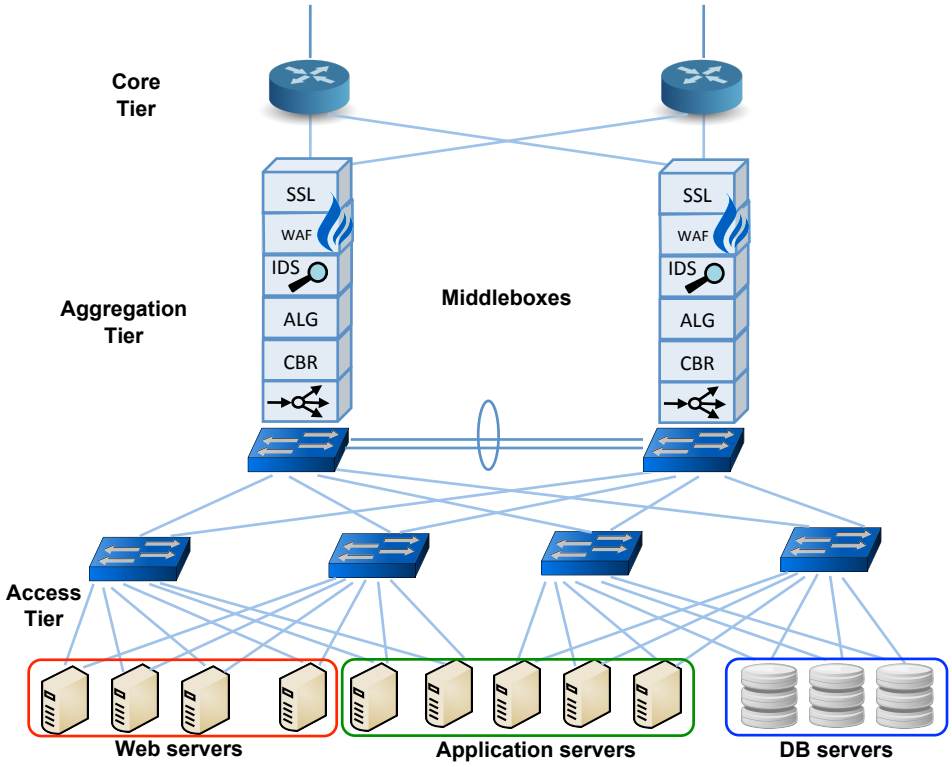

Figure 4.2: Middlebox Deployment in Enterprise Datacenters (reproduction of Fig. 2.10)

As applications migrate from enterprise datacenters to third-party cloud platforms, the Application Service Provider (ASP) no longer has direct access to the infrastructure layer. Therefore, the existing middlebox deployment

techniques no longer work. An alternative solution proposed by many cloud providers is making many of the middlebox functions available to applications as services (Fig. 4.3). This may work for some, but for most ASPs and especially for middlebox functions that need access to application-layer data, this may not be an option for security concerns. Another alternative is virtualizing the middlebox hardware appliance and leasing a virtual appliance to an ASP. One of the problems with this approach is that it is extremely difficult to virtualize a special-purpose hardware appliance while maintaining the performance advantage of having a special-purpose hardware in the first place. The bigger problem is that while the cloud allows applications to grow dynamically on commodity hardware resources, introducing special-purpose hardware could turn out to be a bottleneck to its scalability. A more plausible approach is software-based appliances that can be deployed on commodity hardware, as discussed next.

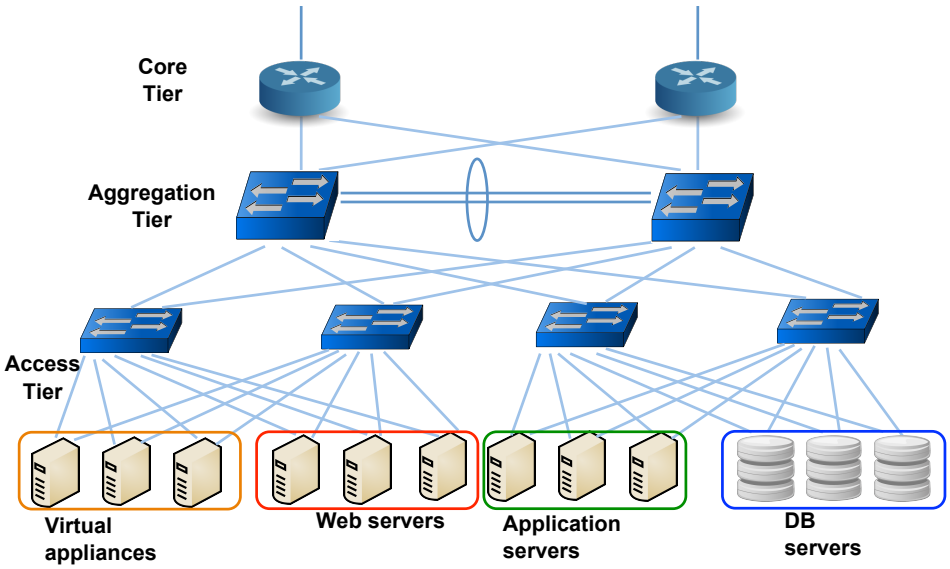

Figure 4.3: Middlebox Deployment in Cloud Datacenters

Appliance makers have responded to the general trend of shifting away from specialized hardware towards commoditized hardware by releasing software versions of their appliances (or, soft-appliance) that can run on commodity

hardware. Soft-appliances have a lot of advantages compared to their hardware counterparts. It allows deployments to easily scale-out (by adding more number of soft-appliances to the deployment) instead of having to scale-up (by increasing hardware capacity). It is more suitable for cloud-based application deployments where soft-appliances can be easily be deployed over a leased commodity host. This allows an ASP to freely choose any cloud provider instead of worrying about availability of certain types of appliances it needs for its deployment. However, given these advantages, they come with their share of inconveniences as well. With soft- appliances, it is no longer possible to deploy them transparently by sneaking them into the network path. They now need to be explicitly accommodated in application deployments; making the case for the need of off-path deployment of middleboxes more stronger.

* **Challenges in integrating message-level middleboxes and application servers with an ADN:** There are no standard methods for integrating message-level middleboxes with other application-layer components. Message-level middleboxes such as Web-application Firewalls (WAF) and transcoders need to be deployed as proxies that terminate a transport-layer (Layer 4 TCP/UDP) connection. They cannot be transparently sneaked into an application deployment, unlike packet-level middleboxes. To achieve this, they need to be integrated with other application-layer components such as application servers, storage servers, etc. This is difficult since the middlebox will need to implement different application-layer protocols and also sometimes understand and support higher-level application-layer constructs such as user sessions; thus making it extremely difficult to design such middleboxes as plug and play devices. The problems of scalability discussed in the context of hardware-based packet-level appliances also apply to hardware-based message level middleboxes. As a result of these problems, message-level middleboxes are generally made part of an ESB middleware platform that provides separate proxying and protocol translation support. However, middleware solutions have their share of deficiencies too, and are discussed next.
  Most complex application deployment environments use some type application platform support such as Oracles Weblogic[95] or IBMs Websphere[55]

that include service integration middleware solutions, generically called the Enterprise Service Bus (ESB) [19]. An ESB provides an indirection middleware layer through which loosely coupled service components can be integrated. The key role of an application delivery middleware is service orchestration; composing applications from many different service components, including application-layer services and message-level middlebox services. Some of these services may be part of the application logic, such as the application servers, web servers and database servers, while others such as protocol transformation and translation gateways may be part of the middleware itself. The logic for combining and composing these disparate set of services is programmed into the underlying application-layer routing (APR) layer that routes application messages based on both, the content of the message as well as the context of the message.

However, middleware-based solutions have several issues that make it unsuitable to serve as a general platform for next generation application delivery.

- ***First***, most middleware solutions are proprietary. This is not so much of an architectural issue as much as a political issue. Proprietary solutions do not generally like commoditization through standardization. Proprietary ESB solutions pack in much more than just the minimal support required to create, manage and control an ADN. This is why we claim that AppFabric is not an ESB. It does not provide any of the in-built services that a commercial ESB ships with. Rather it is a minimal platform for operating ADNs above it. The idea is to standardize this minimal platform so that a competitive eco-system of third-party provided services can develop around it. Presently, commercial ESB platforms also allow the application providers to design their own set of support services such as translation and transformation engines and plug them to the ESB platform. However, the proprietary APIs force the service designers to develop these services differently for different platforms.

- ***Second***, middlewares do not support integration of packet-level middlebox services.

- ***Third***, middlewares do not integrate the compute and network infrastructures.

- ***Fourth***, as was discussed in Chapter 2 , the backbone of most middleware solutions is a message-brokering subsystem. Also, as discussed in Chapter 2 , such message brokering subsystem could either be centralized or distributed. While a distributed brokering system seems to be the better choice for the application use-cases (massively distributed) for which AppFabric is being designed for, the centralized solution appeals through its simplicity in every aspect including configuration, control and debugging. Also, the distributed solution that we have presently are not suitable for massively distributed architectures.

Therefore, it is clear from the above discussion that we need a common integration layer that can integrate both message-level as well as packet-level services. Also, such a platform should be open and have support for virtualized, leased infrastructure that may belong to multiple providers and may be distributed across different geographical locations. While AppFabric provides this common platform to applications, its data plane architecture, OpenADN, needs to provide support for, 1) integrating both message-level as well as packet-level services, and 2) integrating compute, storage and network services; into an application delivery network and provide mechanism for orchestrating these services in application-specific ways .

* **Integration of multiple resource providers:** As already discussed before, an application delivery network (ADN) comprises of different types of devices. Also, AppFabric is designed for creating ADNs over virtual resources leased from many different providers. These virtual resources include both, computing resources (from cloud service providers) as well as networking resources (intra-datacenter network links from cloud providers and inter-datacenter network links from Network Service Providers). Each of these providers may run different management stacks including OpenStack, OpenDayLight, Floodlight, EC2, Eucalyptus, and CloudStack to manage their virtual resources.Therefore to cut across these different virtualization platforms,

device types, and ownership and administrative boundaries, a standard data plane needs to be defined. OpenADN has been designed to serve as the standard data plane that can integrate each of these different vertical components and provide a common data plane abstraction for deploying, controlling and managing different application delivery networks.

* **Integration of different transport mechanisms:** AppFabric provides a common platform for deploying different types of services that can either be owned by the Application Service Providers themselves (application-layer services, both packet-level as well as message-level services on virtual machines) or provided by third party Cloud Service Providers (virtual network services within a cloud datacenter) or Network Service Providers (e.g. WAN services between datacenters). The communication substrate connecting these different types of services is extremely diversified. For example, two message-level services deployed on the same virtual machine may efficiently communicate over Inter-Process Communication (IPC) transport whereas if they or on different hosts they will need to communicate over the network socket transport (network Layer 4 transport such as TCP, UDP, etc.). If the two hosts are in the same datacenter they may be connected through a Layer 2 or Layer 3 network. However, in clouds (as well as in enterprise datacenters), if the network needs to be partitioned into many separate isolated contexts (virtual networks) then the Layer 2 or Layer 3 network may be overlaid with some kind of virtual overlay network such as VLAN(Layer 2) or more recently VxLAN, STT or NVGRE which are all Layer 3 overlays. If the virtual machines are hosted on different datacenters, then they may need to be connected over a WAN link over a ISP network using plain IP or more sophisticated transport services such as MPLS. The key point is that the type of transport required to connect any two services is decided at runtime, when the services are actually deployed. Therefore, at design time, service developers need to be completely abstracted out from writing code for any particular type of transport. To do this, AppFabric needs to provide a common transport abstraction to which all service connect and the actual transport used underneath is bound to the connection at runtime. The AppFabric data plane, OpenADN, is responsible

for providing this abstraction of a common transport and dynamically bind it to an appropriate transport at runtime.

- **Programmability:** The ADVS switch needs to be programmable. Since an application delivery network comprises of both application-layer components and network-layer components, to provide a common standard for managing and controlling the data plane of such a network we need to provide abstractions that are suitable for specifying policies both for application-layer traffic as well as network-layer traffic. We define application-layer traffic as the flow of data between two application-layer entities; whereas network-layer traffic is defined as the flow of data between two network-layer entities. Therefore, although packet flows serve as a good abstraction for routing and forwarding network-layer traffic (through routers and switches) and is therefore used by OpenFlow, it is not appropriate for OpenADN since OpenADN needs to provide the proper abstraction for controlling and managing both, application-layer traffic as well as network-layer traffic. To manage application-layer traffic, OpenADN needs to factor in application-layer semantics into its data plane abstractions. More specifically, It should allow the data plane to be able to represent the following:
  - **Sessions:** A session refers to a logical connection between the user and the application. The exact definition of a a session is specific to the application-layer protocol in use. For example an HTTP session is different from a SIP (Session Initiation Protocol) session. Also, the start and end of a session may be explicitly signaled through special start and end messages (for example, the SIP INVITE and BYE messages) or it could be implicitly signaled thorough a timeout mechanism. The session represents an application context - type of user, user credentials, type of device, place of access, etc. and allows the application to specify context aware policies handle different sessions differently.
  - **Messages:** A message is the smallest, indivisible unit of transaction between a user and an application; and a session is made up of one or more message exchanges between them. Similar to sessions, the structure of a message (start and end of the message) is specific to the application protocol in use. A message is the basic unit of routing among message-level components within an application deployment including message-level middleboxes and application

servers. Message routing is based on forwarding policies specified over both, application context (session information) as well as application data (message content).

  * **Packet flows:** Each application-layer message becomes one or more packets (layer 3 PDUs) in the network. A set of packets that are related in some way and represent some sort of useful semantics are called flows. For example, all packets from a particular user may be subject to a particular security policy and represents a flow. This flow may still have many sub-flows. For example, different packets from the same user bound towards different application end-points may be subject to different policies. For, example, apart from a general security check on all packet from a user, all packets bound towards a server hosting confidential documents may require more stringent checks than packets bound for a server hosting general public information. Thus, flows are hierarchical. In OpenADN we allow the concept of hierarchical flows where the semantics of the hierarchy starts at the application-layer and goes all the way down to the packet-level. Note that this is different from the concept of flows in OpenFlow which also allows hierarchical flows through support for multiple chained flow tables[46]. The difference is that, in OpenADN, the packet are classified into flows based on application-layer information as against only network-layer information in the case of OpenFlow. OpenFlow does allow TCP port numbers to be used to classify packets into flows and thus tries to factor in some application-layer information, but using port numbers as a replacement for application-layer context and content information is quite restrictive.

- **Distribution:** ADVS provides a centralized switch abstraction to the user to which all the different ADN components are connected. It is much easier for the application administrator to control and manage the entire ADN topology thorough this centralized switch abstraction than to actually have to manage each of the distributed ADN components separately. However, this centralized switch abstraction is created over a distributed implementation where the switching function is distributed across all the components in the ADN. Therefore, ADVS is a virtual switch. Distributing the switching function allows ADVS to support a

highly distributed ADN topology without introducing any indirection inefficiencies, performance bottlenecks and a central point of failure.

These are some of the architectural requirements that the OpenADN architecture needs to address. In the next section, we will discuss the OpenADN architecture in light of these requirements.

4.2 OpenADN: Architecture

In this section, we will discuss the key elements of the OpenADN architecture and see how they address the architectural requirements discussed in the previous section. Fig. 4.4 shows a high-level schematic representation of the OpenADN design. Each OpenADN node has a control agent through which the AppFabric control plane (Lighthouse) manages and controls the data plane nodes. The details of the architecture of the control plane and the interface between the control and data planes will be discussed in Chapters 5 and 6. The control agent creates a number of virtual ADVS ports in each node as directed by the controller. Application services connect to the ADVS switch through these ports.

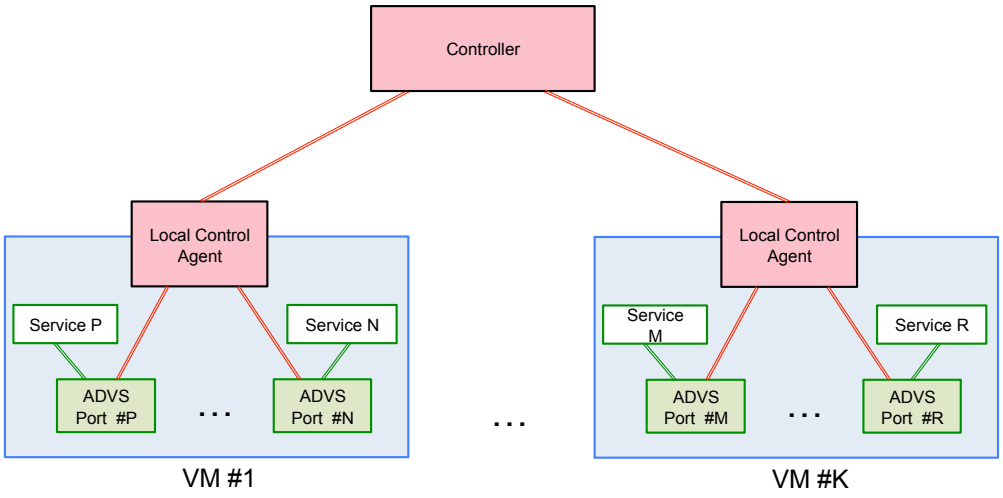

Figure 4.4: High-level Schematic Representation of the OpenADN Design

The general architecture of an ADVS port is shown in Fig. 4.5. Each port has a set of ingress and egress interfaces through which it connects to other ADVS ports in the same or other nodes, a service interface through which an application service connects to the port, and a control interface through which the control plane may program the port. Depending on the type of service, there are three types of ports.

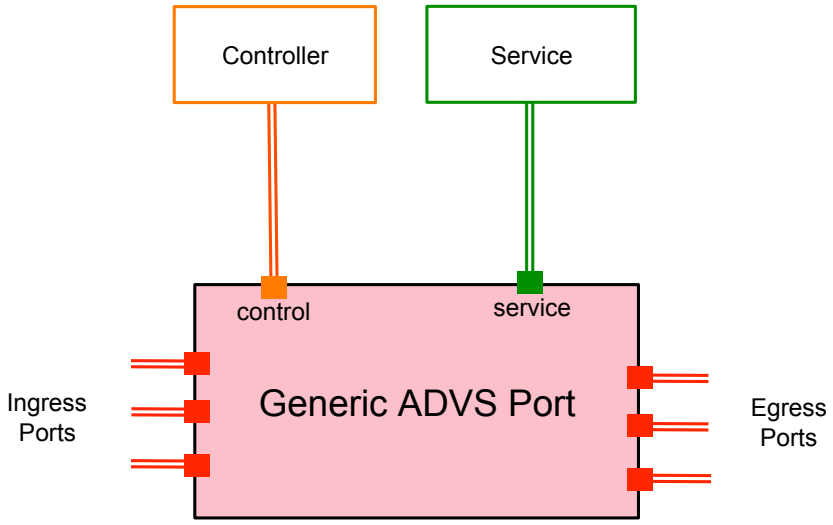

Figure 4.5: Generic ADVS Port

- **sPort:** A sPort or a service port connects a message-level service to the AppFabric platform.
- **tPort**: A tPort or a tunnel port connects a packet-level service to the AppFabric platform.
- **pPort:** A pPort or proxy port connects external third-party services and AppFabric-unaware internal services to the AppFabric platform.

The detailed architecture of each of these ports will be discussed in Chapter 6. For now, it is sufficient to understand that the control agent can dynamically spawn a service in the node, create an appropriate port for the service and connect the service to the port; all based on instructions from the Lighthouse control plane. Once a service is spawned and attached to an appropriate port, the service needs to be connected to all the neighboring services as

specified in the application workflow. This is done by setting up data **tunnels** to all the ports that host these neighboring services. However, this tunneling mechanism is not trivial. We will discuss more on this tunneling mechanism in the context of the how they address the OpenADN architectural requirements. This is discussed next.

- **Integration: Integration** is one of the central high-level themes of the OpenADN architecture. Let us see how OpenADN addresses the different aspects of integration that was discussed in the previous section. But before discussing any of the integration scenarios, we first need to discuss the different logical perimeters within an ADN.

  Just like any network, an application delivery network too has logical perimeters that we call *realms*. Here we discuss two types of realms that the OpenADN design needs to consider.

  - **AppFabric-aware entities and AppFabric-unaware entities:** As is obvious, introducing two new layers into the stack will cause interoperability issues with legacy components that may not migrate to the new architecture. We expect that ASPs that wish to use AppFabric and third-party providers that sell middlebox services to these ASPs will be willing to make their components AppFabric-aware. As we will see later, it is actually quite straightforward to make these components AppFabric-aware. However, it may be much more challenging to make the application user-access agents such as mobile apps, web browsers, etc. AppFabric-aware. Also, the application may want to interpose certain services that may not have yet added support for AppFabric.
  - **ASP-trusted entities and ASP-untrusted entities:** In an ADN, some of the entities may be trusted by the ASP while others may not be trusted by the ASP. Trust is generally linked to an asset. That is, trust is qualified and quantified based on whether an entity allows another entity access to its assets, how much it values these assets and how much access it allows. In our case, we qualify ASP trust based on two assets - Application data and ADN control plane policies. While restricting access to application data as a function of trust is intuitive, it may not be clear as to why we need to restrict access to control plane policies. The rationale behind this is that in a programmable data plane design such as OpenADN, the control plane policies being enforced by the data plane indirectly

reveal sensitive information about the application deployment state and could be easily misused to jeopardize the application. When we come to it, we will see that how, if not done carefully, ADN control policies may reveal a lot about the topology of the application deployment, the relative importance of the different services, what security checks the ADN performs, etc.

Based on this, we use the following two categories to classify entities:

* **Application-data Visible entities (ADV) and Application-data Blind entities (ADB)**: These are two mutually exclusive categories. An Application-data Visible entity or ADV is an entity that has access the application-layer data, either at the packet-level or message-level. On the other hand, an Application-data Blind entity or ADB is an entity that has no visibility into the application data. ADBs include third party(such as CSPs or ISPs) virtual infrastructures such as virtual switches, virtual routers, virtual WAN links, etc.
* **ADN Policy Visible entities(APV) and ADN Policy Blind entities (APB):** These are again two mutually exclusive categories. An ADN Policy Visible entity or APV is an entity that can be programmed with OpenADN control plane policies whereas an ADN Policy Blind entity or ADB cannot.

All ADN entities can be classified as belonging to one of the four classes constituted by these two categories. All ASP-owned application servers and middleboxes (both message-level as well as packet-level) belong to the class ADV and APV (that is, fully trusted). Completely untrusted third-party service to which the ASP does not want to communicate at all belong to the class ADB and APB (that is, completely untrusted). Between these two extremes, are the two partially-trusted scenarios of i) ADV but not APV, and (ii) APV but not ADV. An example of an entity belonging to the class ADV but not APV is an user-access agent (for example, mobile app., web browser, etc.) since they may not always run in a secure, ASP-trusted environment. However, if the user-access agent is running within a secure enterprise environment then it may be fully trusted. Similarly, third-party services to which the the ASP needs to communicate also falls in the same category. Examples of entities belonging the class APV but not ADV include CSP or ISP provided virtual resources including virtual switches, virtual routers, and virtual WAN links. ASPs do not provide these entities access to their

application data. However, they do provide them with some access to control plane policies. CSPs and ISPs may misuse this information but we (safely) assume that it is against their economic interests to do so. However, there may still be unintentional misuse of these policies by rogue compromised elements within them (insider attacks). But, this is a common security problem that may be posed for any situation; including rogue elements within the ASP-owned entities also. Therefore, insider attack scenarios are outside the scope of this discussion.

Out of the eight different classes that are possible by combining these three different categories (AppFabric aware/unaware, ADV/ADB ad APV/APB) some of the cases are trivial and does not need to be handled separately. These cases include:

- AppFabric-aware entities that are both ADB as well as APB (completely untrusted by the ASP) are not part of the ASPs ADN and therefore we do not need to worry about communicating with them.
- AppFabric-aware entities that are fully trusted (ADV and APV) do not need any special mechanism to be made part of the platform.
- AppFabric-unaware entities cannot be APV by definition of APV (entities that can be programmed with ADN control plane policies). Therefore, we can rule out two classes of entities - (AppFabric-unaware, APV, ADV) and (AppFabric-unaware, APV, ADB).

This leaves us with four classes of entities that need further consideration.

- **AppFabric Gateway Node(AGN):** An AppFabric Gateway Node or AGN is a special node designed to easily integrate the following classes of nodes into the ADN:
  - **(AppFabric-unaware, ADV, APB)** These are legacy entities that need not have any notion about the existence of AppFabric. The AGN is needed to mediate all communication between these entities and the ADN. Entities belonging to this class include external entities such as third-party services that the ADN needs to contact for certain services or user-access agents through which users connect to the application. The AGN proxies all application sessions between these entities and the application and provides application policy visibility for subjecting these application sessions to the ADN

control policies. Each of these legacy entities may speak different standard application-layer protocols such as HTTP, JSON, SIP, etc. or non-standard, application-specific protocols such as distributed gaming applications, virtual worlds, etc. The definition of an application session and application message may vary widely across these different protocols. Thus, the AGN needs to convert these application-specific representations of these data plane abstractions to a standard AppFabric platform specific representation over which the data plane may be programmed. To do so, the AGN terminates these application protocol specific sessions and creates standard OpenADN sessions. An OpenADN session comprises of multiple OpenADN messages with explicit session start and session end signaling messages. The controller may specify different session-level policies that are enforced on each message within the session (based on context information such as user type, access location, access device type, etc) as well as message-level policies that are enforced per-message based on the content of the message.

  * **(AppFabric-aware, ADV, APB)** These are AppFabric-aware entities that have application data visibility but are not programmable. There is a slight technicality in calling these entities AppFabric-aware that needs to be understood. These nodes are AppFabric-aware in the sense that they can deal with AppFabric-specific header fields in the application messages/packets but are not programmable by the AppFabric control plane. Examples of such entities include legacy middleboxes that can easily be programmed with new parsing and filtering rules for new types of message/packet headers but have not yet added support for being programmed by the AppFabric control plane with ADN traffic management rules and policies. The AGN acts as a proxy for such nodes and all the OpenADN control plane policies can be programmed into the AGN and the AGN enforces those policies on behalf of the node.

– **Layer 4.5 and Layer 3.5 shim layers**: OpenADN introduces two new shim layers into the network stack - Layer 4.5 and Layer 3.5.

  * **The OpenADN Layer 4.5:** The Layer 4.5 is a shim layer between the application layer and the end-to-end transport layer (TCP, UDP, etc.). It acts as the thin waist for connecting many different message-level entities. On

the top, it provides a common standard *OpenADN session* and *OpenADN message* abstraction across many different application level protocols. The Layer 4.5 message header carries session-level context information for every message. Also, it carries a *meta-tag* that encodes the per-message content information. The *meta-tag* allows the application data to be securely encrypted while still allowing entities that are ADB but APV to enforce control plane policies on the message such as routing and forwarding the message through a set of ADV entities. At the bottom, it provides a common transport between different message-level entities. For example, services on the same virtual machine may communicate efficiently over an Inter-process communication (IPC) transport whereas those on different virtual machines need to communicate over socket abstraction. Layer 4.5 can be programmed to select the most efficient transport between any two message-level services dynamically during runtime.

* **The OpenADN Layer 3.5**: While the Layer 4.5 shim layer provides the *OpenADN session* and *OpenADN message* abstractions, the Layer 3.5 shim header lying between Layer 4 and Layer 3 of the network stack, provides the abstraction of an OpenADN packet flow. the OpenADN packet flow is defined as a set of packets that are related through some application-layer semantics such as sessions and messages. The Layer 3.5 header carries this application-layer context (session and message information) in each packet such that packet-level entities that may not have visibility into the application data (ADB) but are APV can easily enforce the ADN control plane policies at packet-level granularity.

Therefore, the shim layers 4.5 and 3.5 allow the integration of entities that have access to the control plane policies (APV) but do not have access to the application data (ADB) into the ADN - (AppFabric-aware, ADB, APV). Note that these entities have to be AppFabric-aware.

– **Tunneling:** Now, the only class of entities left are (AppFabric-unaware, ADB, APB). These entities are not directly part of the ADN, but serve as the underlying infrastructure for physically connecting the different ADN entities. Example of such services include network infrastructure services providing QoS transport between any two ADN nodes. The infrastructure nodes in this case cannot be

expected to be AppFabric-aware. The usual technique of *tunneling* is used to create an ADN overlay over these legacy infrastructure nodes. *Tunneling* is a very important concept in the OpenADN architecture and we will discuss it in much more detail in the rest of this chapter.

Now that we have seen how different types of entities connect to the platform, let us see how OpenADN addresses the various integration aspects in the data plane.

– **Integration of message-level and packet-level application services into the ADN:** As already discussed (and as shown in Fig. 4.6), an ADN comprises of both message-level and packet-level application services. Typically in application deployments, message-level application services are orchestrated using some sort of middleware platform, while packet-level application services are considered part of the network and their deployment is part of the network management and control plane. In OpenADN, the deployment of both these types of entities are controlled and managed by a common control plane since both these types of services are hosted on commodity virtual machines (in the datacenters access tier) in cloud environments. We use nested tunneling to achieve this (Fig. 4.7). To illustrate nested tunneling, consider the simple AppFabric Service Workflow (ASW) shown in Fig. 4.6. Note that our goal is only to illustrate how nested tunneling works, and so, Fig. 4.6 shows only an uni-directional path through the different device types and the Application Routing component is really trivial in this example.

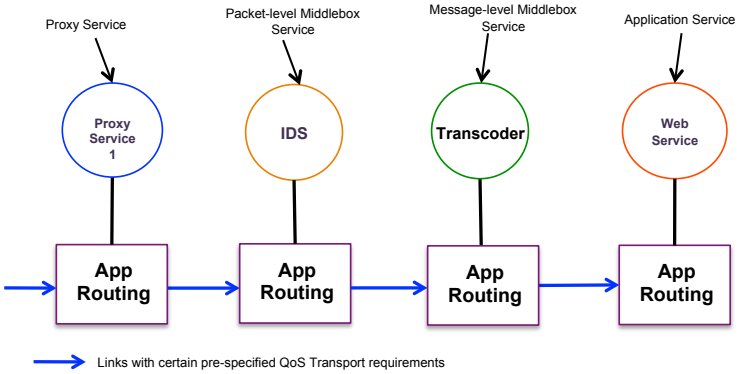

Figure 4.6: An Example AppFabric Service Workflow

To integrate message and packet level services in the ADN through nested tunneling, we create a Layer 4.5 tunnel between any two message-level services and this tunnel is transported over one or (possibly) more than one Layer 3.5 tunnels spliced end-to-end. The Layer 3. 5 tunnels connect intermediary packet-level services between the two message-level services. Layer 4.5 and Layer 3.5 are new shim layers introduced into the network stack by OpenADN, as discussed above. The services just connect to an appropriate OpenADN port (software port) that may be a message-level (Layer 4.5) port or a packet-level (Layer 3.5) port. A message-level port can have only one message level service connected to it while a packet level port can have many different packet-level services connected to it. A message level port acts as a message switch while a packet-level port acts as a packet switch that switch messages/packets arriving at the ingress tunnels to the proper egress tunnels based on application-layer (content and context based) switching rules programmed by the control plane. A message/packet arriving at the ingress tunnel head in a port is first sent to the service attached to it. In case of a message-level port that has only one message-level service attached to the port, this step is trivial. In the case of packet-level ports that may have more than one packet-level services attached to it, the port needs to decide which service the packet needs to be sent based on the application-layer switching rules specified over the contents of the Layer 3.5 OpenADN header fields. When the packet/message returns from the service, it is either switched to one of the egress tunnel heads or to another service attached to the packet-level port; again based on the application-layer switching rules programmed into the port by the controller over the contents of either, the Layer 4.5 header fields or the Layer 3.5 header fields.

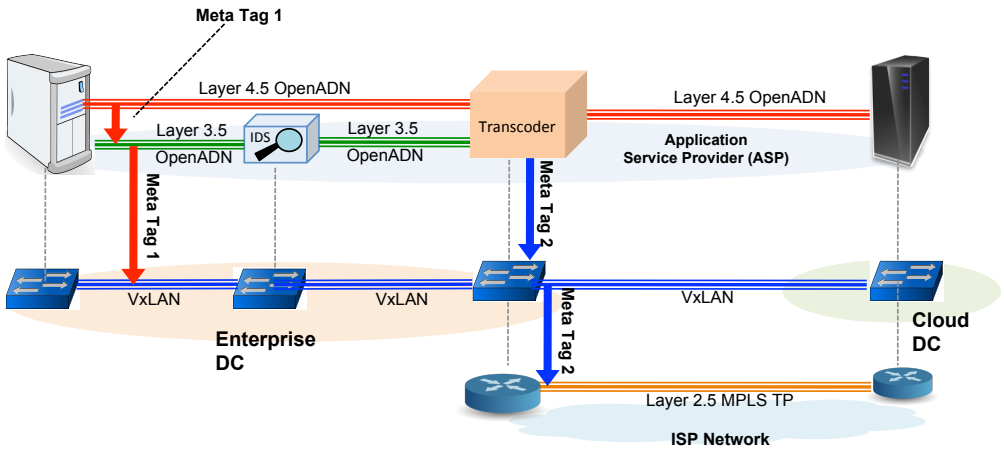

Figure 4.7: Configuration of Nested Tunnels in OpenADN for the AppFabric Service Workflow in Fig. 4.6

It may be noted that in the multi-hop scenario described above, the information about the last hop is implicit in the ingress interface at which the message/packet arrives in a port and therefore the message/packet does not need to explicitly carry this state. At an abstract level, the ADN looks exactly like a physical network where each node has fixed physical connections to a set of other nodes and thus the ingress interface implicitly carries the information about the last hop. The only difference is that in OpenADN, these connections are virtual, pre-established by the control plane, and can be dynamically changed as and when required.

– **Integration of the compute and network infrastructures:** The nested-tunneling mechanism is also used to create application-specific networks over third-party network infrastructures. The idea is to transport the OpenADN Layer 3.5 tunnels over Layer 3 tunnels such as VxLAN[77], STT[31] or NVGRE[127] between virtual machines and over layer 2 tunnels such as MPLS-TP over wide-area networks. These tunnels are created and operated by the infrastructure provider and not directly under the control of the ASP. However, the ASP may request the infrastructure provider to create a virtual network connecting its different application-layer entities such that there may be more than one network tunnels between each pair of VMs. Each of these tunnels may be programmed to provide different QoS services. At the ingress of the network tunnel, which could be a virtual switch in the hypervisor or a top of the rack switch, the application

packets could be classified and transported over the different tunnels based on the OpenADN layer 3.5 header. Alternatively, the infrastructure provider could create just one network tunnel between the two VMs and application-specific traffic shaping is done at the egress of the VM itself. This allows the traffic between two VMs to be prioritized based on the application context but the network tunnel itself has no priority over other tunnels in the network. Hence, this overlay mechanism of providing QoS could be more restrictive than the first technique in which the QoS is provided by the infrastructure natively. The important point to note is that both these techniques are equally plausible in the OpenADN architecture. Also note that the infrastructure nodes are ADB but can be programmed indirectly, as in the case where the QoS is provided natively by the infrastructure, to provide application-specific transport services using OpenADN Layer 3.5 header.

– **Integration of multiple resource providers:** The ADN may include infrastructure components (virtual machines, virtual switches, virtual appliances, etc.) leased from many different resource providers. OpenADN either has full control of these virtual resources as in the case of a virtual machine or programmatic control through a delegation interface as is the case of a virtual network. Full control means that the application can actually deploy and run the platform code in the resource whereas programmatic control means that the application can configure the resource (such as a virtual switch) but cannot run its own code over them. While integration of resources with full control is trivial (the OpenADN code runs on them directly), integrating resources with only programmatic support is a bit more trickier. Such entities represent the class (AppFabric-unaware, ADB, APB). To integrate these type of resources into an ADN, the AppFabric-aware nodes need to be able to implement the appropriate tunneling and signaling mechanisms. For example, suppose the application has two different VMs hosted in two different datacenters connected by an ISP network. Now suppose the ISP provides a virtual QoS guaranteed packet transport service over a Metro Ethernet of MPLS infrastructure to the application. In this case, OpenADN should be able to setup a Layer 4.5 or Layer 3.5 tunnel between the two VMs that is transported over the virtual packet transport service provided by the ISP. To do so, the VM tunnel endpoints need to have the proper drivers that allows them to properly signal and consume this third-party provided service.

- **Programmability:** Each of the OpenADN ports (pPort, sPort and tPort) are programmable. The control plane can dynamically spawn any number of OpenADN ports (and their associated services) in each data plane node through the node's control agent. It can also shutdown the service and its associated OpenADN port when no longer required.

  From the application routing policies specified by the application administrator through the management plane interface, the controller computes the message/packet forwarding rules for each OpenADN port. The controller distributes these forwarding rules to the appropriate ports. The OpenADN message or packet forwarding is a two-step mechanism. In the first step, the message/packet is classified based on application-level content/context and network-level header information to belong to one of the many application flow classes. In the second step, the forwarding rule for the appropriate flow class is applied on the packet/message.

  Since the OpenADN ports are virtual (software-based), they allow a lot of flexibility in terms of specifying the message/packet classification rules. For example, different applications may attach different semantic meanings to the definitions of messages and sessions in specifying the classification rules and thus application-level policies. Albeit, this flexibility comes at the cost of lower forwarding performance. However, the inefficiency of a software-based forwarding is expected to be ameliorated by the on-demand, scale-out property of the software-based solution. This flexibility is extremely important to build application-specific networks where the application can appropriately express and enforce application-level policy routing. This application-level forwarding overlay is automatically created by AppFabric over an underlying high-performance, standardized packet forwarding infrastructure.

- **Distribution:** Although OpenADN provides the abstraction of a single, centralized ADVS switch, in reality, each of the ADVS ports are distributed across the data plane nodes such that they may be on the same virtual machine, on different virtual machines in the same physical machine, on different virtual machines in the same datacenter, or on virtual machines distributed across different datacenters. To achieve this, AppFabric provides a common transport abstraction to the application services. Application services connect to the platform through the standard AppFabric socket interface. The AppFabric socket interface has two types - message interface for message-level services and packet interface for packet-level services. At runtime, the AppFabric socket

binds to an OpenADN port. The OpenADN port may connect to other OpenADN ports using the appropriate underlying transport including IPC, socket, IP and nested transports as described above. The same OpenADN port may use different transports depending on the service it wants to communicate with. The service developer is abstracted out of all this detail and the platform automatically binds to the most appropriate transport mechanism. Therefore, it is extremely easy to make any legacy service AppFabric. all one needs to do is replace the network sockets in the code with AppFabric sockets. The destination address that the socket needs to bind to is the local OpenADN port. The socket connects to the port either over IPC (since they are on the same host) or a packet raw interface (for packet-level services). The address of the port is provided as part of the service' environment parameters when the service is launched.

So, this concludes our discussion on the high-level architecture of the AppFabric data plane - OpenADN. Details of the actual implementation of OpenADN will be discussed in Chapter. 6.

Chapter 5

Lighthouse: The AppFabric Control and Management Plane System

In this chapter, we will discuss the architecture of Lighthouse: the control and management plane system of AppFabric. Lighthouse comprises of two subsystems - the control subsystem and a the management subsystem. The control subsystem consists of many different controller entities arranged in a hierarchy while the management subsystem consists of single centralized manager entity.

The hierarchical design of the control subsystem is a hybrid between completely distributed control plane architectures (for example, distributed routing protocols) and completely centralized control plane architectures as proposed by some of the initial Software-defined Network(SDN) designs. While distributed architectures are more scalable and also more resilient against failures and security threats, centralized architectures are simpler to design, manage and debug. The hierarchical design of Lighthouse control subsystem adopts the best of both these extremes. Unlike a central controller based architecture, It is more scalable since each controller at every level is responsible for only a fixed (relatively smaller) number of nodes (either data plane nodes or other controller nodes). It is also more resilient since failures and threats can be more effectively isolated and thus their effect can be minimized. However, It may not be as simple to design, manage and debug a hierarchical system as it is for a completely centralized controller design. The key issue is that each controller has a partial view of the system. this view gets wider (or more complete) as we move up the hierarchy till the global controller has the complete view of the system. However, a more complete view comes at the cost of additional delay. Therefore, to successfully design a hierarchical

controller system, it is extremely important to be able to properly factor the responsibilities across the different levels such that each controller can make local decisions based on its limited local view of the system while delegating the decisions requiring a more global view to the higher-level controllers. In AppFabric, this tradeoff between delay and accuracy is extremely relevant since AppFabric is designed for multi-datacenter environments where the virtual resources to be controlled by the control plane may be very widely distributed. In such environments communication delays are non-negligible and needs to be accounted for during the design itself.

On the other hand, the rationale behind the centralized design of the management subsystem is that management plane functions are often less time critical than control plane functions. To drive this point home, it is important to understand the differences between the control plane functions and the management plane functions. But, to be able to understand this difference in the context of the AppFabric architecture, first let us take a step back and revisit the high-level design objectives of AppFabric. As discussed in Chapter. 3, AppFabric sits between the virtual software-defined infrastructure layer and the application deployment layer. The virtual infrastructure may be provided by many different providers including cloud service providers (providing virtual compute, storage and intra-datacenter network) and network service providers (providing virtual inter-datacenter links and other WAN services). All such virtualized infrastructure providers expose a set of service APIs (application programming interfaces) through which their virtual resources can be requested, acquired/leased, programmed and released. On the other hand, Application Service Providers (ASPs) need to deploy and deliver their applications over these virtualized resources leased from multiple resource providers and continuously monitor and optimize the application deployment at runtime. Now, with these objectives, let us try to motivate the difference between the management and control plane functions.

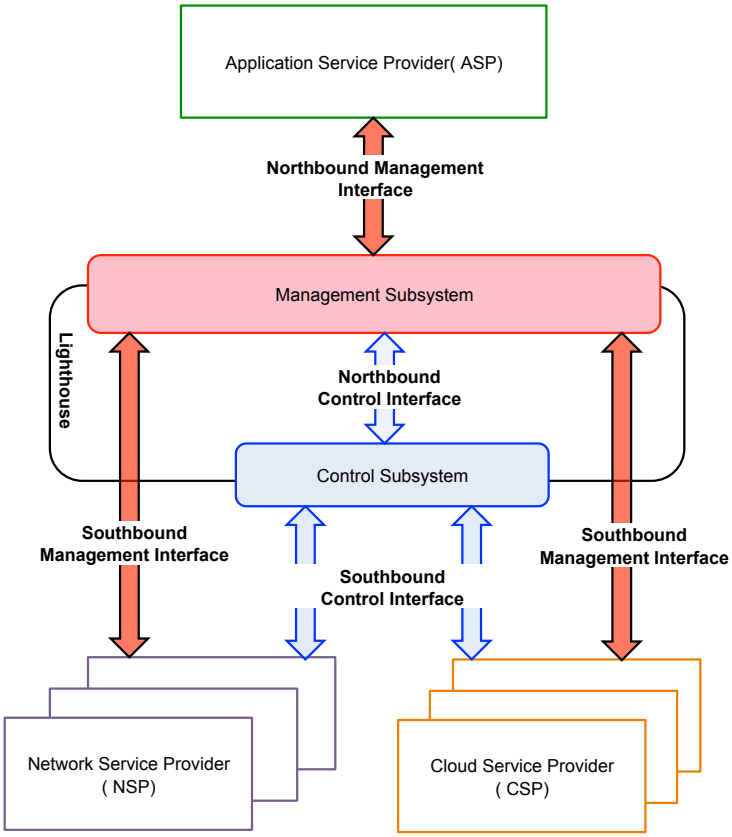

Figure 5.1: Lighthouse Interfaces

- **The Lighthouse management plane:** As shown in Fig. 5.1, the management plane exposes three interfaces - two external interfaces and one internal interface. The external interfaces are policy-based management interfaces between the platform and external entities including the ASP in the north and the ISPs and Cloud Service Providers (CSPs) in the south. The interface between the ASP and the platform is called the *northbound management interface* and the interface between the platform and the virtual infrastructure providers is called the *southbound management interface*. Through the northbound interface, the ASP may specify policies pertaining to how the application needs to be deployed and delivered to optimize different objectives such as better user experience, cost, efficiency, enhanced security, etc. We have already discussed the abstractions exposed through this interface in much detail in chapter. 1. Through the southbound interface, the Lighthouse management plane talks to the SDI management

plane of the virtual infrastructure providers. Through this interface it may dynamically acquire and release virtual resources from multiple providers.

The third interface is the internal interface between the management module and the control module. We will discuss it in the next point.

- **The Lighthouse control plane:** As shown in Fig. 5.1, the control plane has two interfaces - one internal interface and one external interface. One of these is an internal interface between the control plane and the management plane and is called is called the *northbound control interface*. The management plane compiles the ASP policies received through its northbound management interface into runtime enforceable rules and pass them to the control plane through this interface. Also, the control plane may request the management plane for more virtual resources, either additional resources from an existing provider (e.g. to scale-out the application deployment) or resources from a new provider (e.g. to distribute the application deployment to newer locations). The second interface is an external interface between the control plane and the virtual resource(s) such as virtual machines, virtual switch, virtual router, etc., acquired by the management plane and is called the *southbound control interface*. Through this interface the control plane programs the virtual resources in order to enforce the ASP's deployment and delivery policies during runtime.

Therefore, although as compared to traditional architectures, the AppFabric management plane is very dynamic, its functions are still much less time critical as compared to the control plane. This justifies our design choice of a centralized implementation of the Lighthouse management subsystem while the control subsystem is hierarchical. In the rest of this chapter we will discuss the Lighthouse architecture in more detail.

5.1 The Management Subsystem

As shown in Fig. 5.2, the management subsystem consists of a single entity called the *Lighthouse global manager*. It is the first and the only node in the application that has to be started by the administrator manually. The ASP may either run it on-premise or on a cloud

datacenter. However, given some of the sensitive information that this node has access to including billing information, user authorization/authentication and application policies and configurations, it is preferable to host it on-premise. The global manager exposes the *northbound management interface* thorough which the ASP administrators can specify the policies for deploying and delivering the application. The manager is connected to a database server from where it may fetch the virtual machine images of the different Lighthouse controllers and data plane nodes.

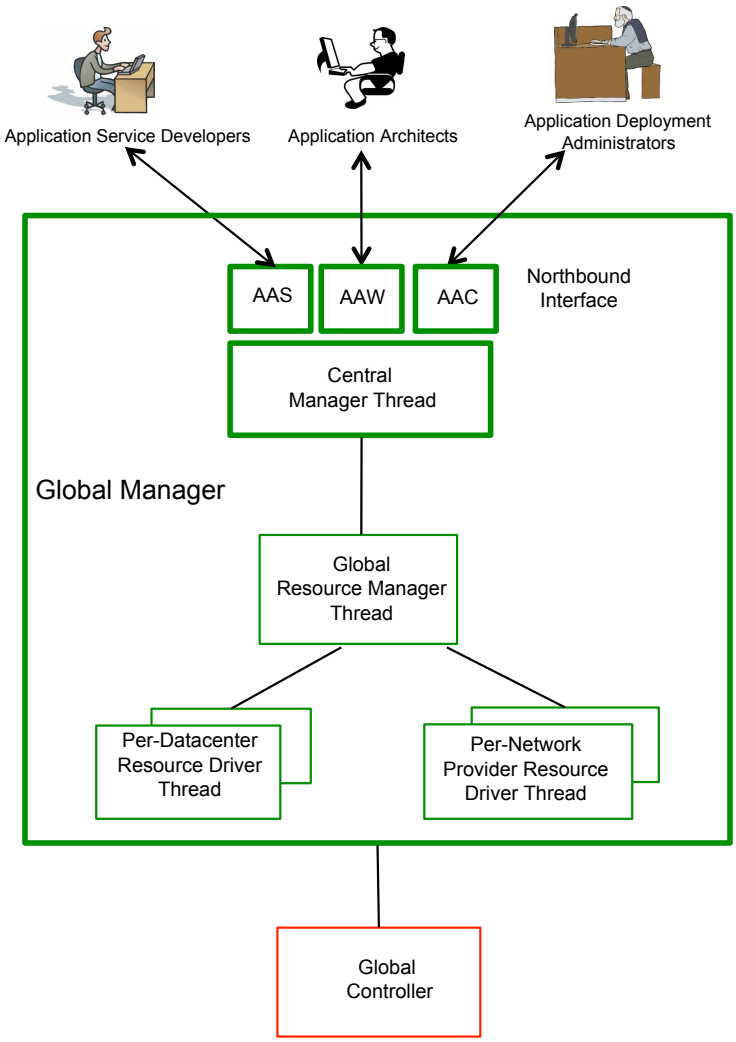

Figure 5.2: Schematic Representation of the Global Manager

The primary function of the global manager in the application runtime is to bootstrap the application deployment. But before discussing the bootstrap process, we need to explain the concepts of zones and sites (Fig. 5.3). In AppFabric application deployments, a large geographical region is divided into several sub-regions called *zones*. Each zone has several datacenters, also called *sites*. For example, as shown in Fig. 5.3 United States may be divided into 3 zones -US-E, US-W and US-S. Each zone independently manages the application deployment over the datacenters in that region. Also each datacenter is classified as 'EDGE,' 'CORE,' or 'EDGE/CORE.' The services in the application are also classified under the same labels. An 'EDGE' service needs to be close to the user access locations and are widely distributed. Examples of edge services include application gateways, firewalls, and data gathering and near-real time control functions in IoT-like use cases. It needs to be deployed on datacenter marked as either, 'EDGE', or 'EDGE/CORE.' A 'CORE' service is located at a somewhat centralized location relative to all the edge datacenters. it is expected that there will much fewer core data centers compared to the number of edge datacenters, with each core datacenter serving multiple edge locations. Examples of 'CORE' services includes aggregation services running business intelligence logic over the gathered data in IoT use-cases, centralized database for synchronizing the different edge services, etc. Note that in many cases the core datacenter could be an enterprise datacenter while the edge datacenters could be cloud datacenters or much more distributed and smaller micro-datacenters operated by ISP networks in their network points-of-presence (POPs). A 'CORE' service may be deployed either on a 'CORE' datacenter, or a 'EDGE/CORE' datacenter.

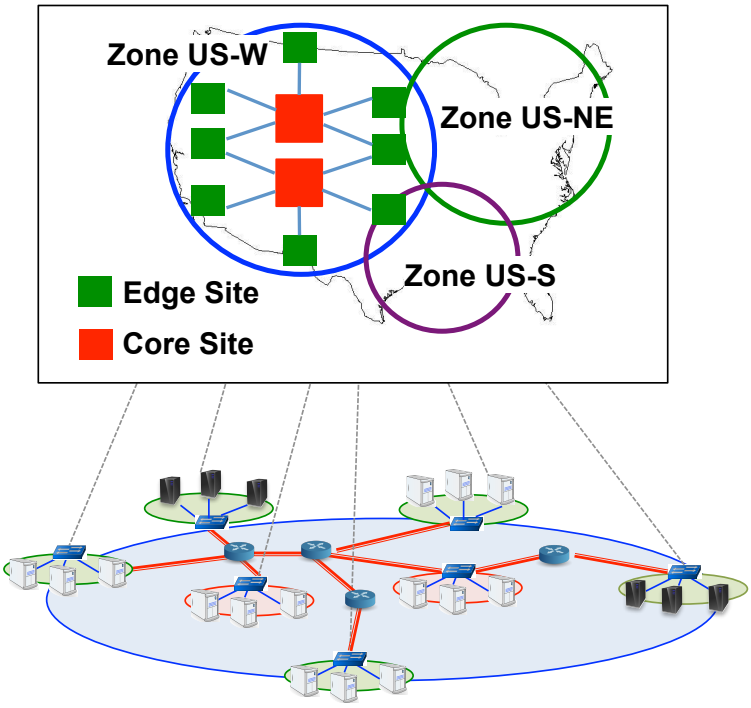

Figure 5.3: Sites and Zones

Now, let us look at the sequence of tasks performed by the global manager during the application bootstrap process.

- *Compile user-specified policy scripts:* The global manager accepts the policies for creating the application workflows (AAW) and the distributed application cloud (AAC) from the administrator through its northbound interface, verifies and validates the policies for conflicts and/or mis-configurations and then compiles and loads these policies into a local database.

- *Start the global controller and select cloud datacenter sites:* After loading the policies, it starts the global controller. Again the global controller may either be on-premise or on a cloud datacenter. Generally the manager will choose a location for the global controller such that it has (almost) the same latency from all possible datacenters (cloud as well as enterprise) that are candidates for the application deployment. The global controller in turn starts a workflow controller for each workflow in each zone. We will discuss more about this in our discussion on the global controller design. The global

manager, on the other hand, maintains a global database of different cloud/enterprise datacenter configurations including provider name, location, security (authorization and authentication) and access policies, billing information, and resource access and lease policies. It chooses (based on some policy) an initial set of datacenters in each *zone* from this list and acquires a pre-configured minimum number of virtual resources (compute, storage and network) from these datacenters. The initial number of resources acquired should be enough to launch at-least one instance of each workflow per zone. However, the ASP administrator may choose to override this default behavior and start multiple workflow instances during the bootstrap. This may be necessary to distribute the application geographically within the same zone. These virtual resources are acquired through the *southbound management interface*. The manager starts a separate resource driver thread for each datacenter. These resource driver threads acquire the necessary virtual resources by calling the appropriate APIs specific to the datacenter management stack (such as OpenStack, Eucalyptus, CloudStack, etc).

- *Start datacenter controllers:* After selecting the datacenter sites and acquiring the required virtual resources on them, it starts a datacenter controller for each site.

- *Launch virtual resources per datacenter:* Next, it launches the data plane nodes including AppFabric virtual machines and virtual switches in each of these selected datacenters.

- *Setup inter-datacenter links:* Just like the global database for cloud datacenters, it also maintains a global database of WAN providers that connect these datacenters. After bootstrapping the sites, it calls the APIs of WAN providers to allocate resources for connecting the different sites.

After bootstrapping the control plane and launching the resources for the data plane, it signals the global controller to start the workflow manager threads. The global controller starts the workflow managers; one per zone per workflow.

These 6 steps complete the initial bootstrapping function of the global manager. It then steps aside and lets the global controller take over the responsibility of actually launching and managing the application. During the application runtime, the global controller may

call the global manager from time-to-time to allocate more resources. These resources may be allocated from the already active sites or it may choose to launch new sites based on the policy specified by the ASP administrator. In our current implementation of AppFabric, our policy language is quite limited in what it can express. Future versions will incorporate much more richer policy specification languages that will allow the ASP administrator to have better control over the application deployment and delivery. The other role of the global manager during the runtime is to create reports and alarms for the ASP administrator based on the runtime logs exported by the control plane entities. Also, the global manager provides an interface to the ASP administrator to manually change the state of the application deployment (overriding the automatic behavior). For example, the ASP administrator may choose to shutdown certain sites or start new sites in order to handle exceptions or other planned or unplanned situations such as security attacks, system maintenance, etc.

5.2 The Control Subsystem

The Lighthouse control subsystem comprises of a set of controllers arranged in a hierarchical order. At the top of the hierarchy is a global controller that spawns a set of workflow manager threads. Each *zone* has a **global controller** which controls multiple **local datacenter/network controllers**. In the rest of this section we will discuss the functions of these two control plane modules in more detail.

5.2.1 The Global Controller

After the initial bootstrap, the central manager signals the central controller to start the workflow manager thread. The global controller (Fig. 5.4) will launch one workflow manager for each zone. Initially, each workflow manager will try to launch one workflow instance for that workflow. The steps in launching/removing a workflow instance thread include the following:

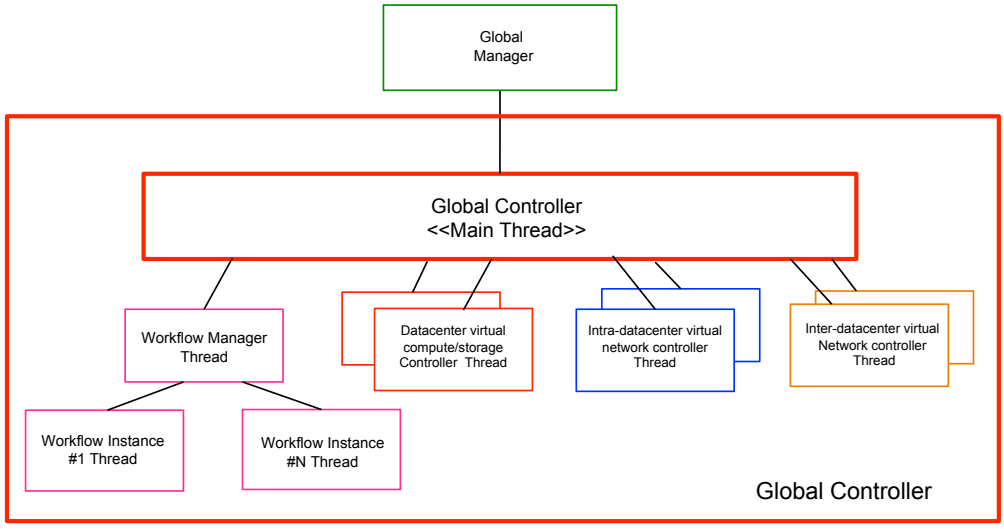

Figure 5.4: Schematic Representation of the Global Controller

- **Query for proxy port (pPort) resources:** The ingress and egress to every workflow instance has to be a proxy port or a pPort. The details of the pPort design will be discussed in Chap. 6. For now, let us look at the high-level schematic representation of the pPort as shown in Fig. 5.5. The pPort sits between the AppFabric Gateway Node (AGN) on the *external side* and AppFabric-aware services composing the AppFabric service workflow on the *internal side*. The external side represents external users and third party services that are assumed to be AppFabric-unaware and thus need the AGN to connect them to the platform. The internal side represents the AppFabric Service Workflow (ASW) comprising of a set of AppFabric-aware services. The pPort may serve as the ingress/egress for multiple workflow instances, as specified in the configuration. The constraint in getting resources for the pPort for the workflow ingress/egress is that it needs to be launched at a datacenter site marked as EDGE (Fig. 5.3). The global controller sends the resource query to each of the local datacenter/network controllers of EDGE datacenters. On receiving the replies from the datacenter controllers, the workflow manager selects one of them (potentially from a list of multiple datacenters) based on certain policies that may, for example require maximizing the extent of distribution of the proxies across the available edge datacenters. It then signals the selected datacenter to start a pPort and the associated AGN node.

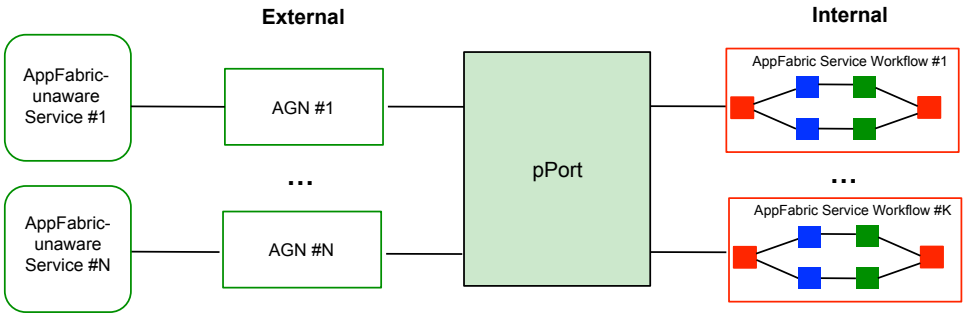

Figure 5.5: Schematic Representation of the pPort

- **Launch a workflow instance thread:** Once the pPort and the associated AGN node has been started, the central controller launches the first workflow instance thread. It then queries the datacenter application controllers, datacenter network controllers and inter-datacenter network controllers (Fig. 5.4) for resources to launch the workflow instance. Note that we have not yet introduced the concept of separate controllers for the application and the network. We will discuss it in the next section. For now, it is sufficient to understand that our controllers are logically separated into two sets of functions- one for directly controlling the OpenADN ports in the data plane (pPort, sPort, and tPort) and the other for managing virtual network resources (virtual switch, virtual WAN links) and virtual network services provided by a cloud or network provider. Once the workflow manager finds the resources to launch the workflow instance, it delegates the responsibility of actually launching, managing and controlling the workflow instance to the workflow instance thread.

- **Dynamically scale-up or scale-down:** The workflow manager constantly queries each active workflow instance for their load information. If it finds that the workflows are overloaded (actual algorithm will be discussed in our detailed design section), it will start a new workflow instance thread. Conversely, if it finds that the average load across all the workflows is less than a particular lower threshold, it starts de-commissioning the active workflows one at a time (again, the actual algorithm will be discussed in our detailed design section) till the average per-workflow load goes above the threshold.

- **Requesting additional resources from the global manager:** Whenever, any of the resource request queries fail, the controller requests the central manager to

allocate additional resources. On receiving such a request, the manager either requests additional resource allocations from the currently active resource providers, or chooses a new resource provider from its database and requests resources from it.

Apart from these, each workflow instance thread in the central controller performs the following functions:

- **Start workflow services:** The workflow instance thread launches all the services in the workflow and their associated ports (sPort or tPort) in parallel.
- **Configure Layer 4.5 and Layer 3.5 tunnels:** After launching the services and their associated ports, it configures the Layer 4.5 and Layer 3.5 tunnels across them.
- **Configure application-level routing and packet routing in the ports:** After setting up the tunnels, it configures application-level routing in the sports and packet routing in the tPorts. An application-level router consists of a message classifier (over the message content and context) and a forwarding rule while a packet router classifies packets based on the meta-tag carried either in the IP_OPTIONS field (for packets traveling between an sport and tPort in the same host) or in the Layer 3.5 tunnel header (for packets traveling between tPorts in separate hosts).
- **Attach workflow instance to proxy:** After successfully launching the workflow, it attaches the workflow to an ingress/egress proxy service (connected to a pPort). Note that the workflow instance thread is not responsible for launching/managing the proxy and is instead managed by the workflow manager. This is because the proxy is a shared resource that is shared among multiple workflow instances.
- **Make the workflow globally accessible:** After successfully attaching an ingress/egress proxy to the workflow instance, it registers the workflow instance with the name-server (currently we have our own name-server implementation but DNS can also be used). The entry is a mapping of the form $\langle Application\ Name \rightarrow IP\ address\ of\ proxy \rangle$. As you may have noted, all workflow instances connected to the same proxy has the same entry. This is intended. The name server implements a weighted round robin scheme and each duplicate mapping entry is considered a separate entry. For example, a proxy hosting n workflow instances will have n duplicate mappings n the name

server. Suppose, there is another proxy hosting m workflow instances. Now applying simple round robin, the first proxy will receive n/(n+m) fraction of the traffic while the second proxy will receive (m/n+m) fraction of the traffic; thus automatically the simple round-robin scheme actually implements a weighted round-robin. Load balancing across the workflow instances within the same proxy is done by the pPort.

5.2.2 The Local Controller

Each resource site leasing/contributing virtual resources to the AppFabric platform has a **local controller**. Therefore, whenever the global controller decides to launch a new deployment site (e.g. cloud/enterprise datacenter or micro-datacenters attached to a network POP), it needs to launch a local controller for managing/controlling the resources for that new site. Also, adding a new site to the application deployment may need adding new wide-area-networking resources to ensure the reachability of the new site from the existing sites. In that case, the global controller may need to launch a new inter-datacenter virtual WAN controller to control/manage these new network links if it involves a new network provider that is not already part of the ADN.

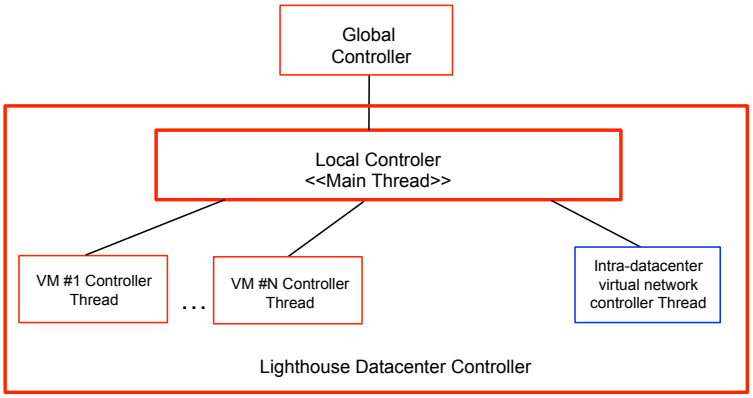

Figure 5.6: Schematic Representation of the Local Controller

As shown in Fig. 5.6, the design of the local controller is pretty simple. Its primary function is resource management. Each virtual machine that is launched by the platform runs a host controller agent called the cPort (as shown in Fig. 5.7). The cPort manages the AppFabric

platform ports in the hosts. The AppFabric platform ports are the interfaces through which services connect to the distributed AppFabric platform. There are three types of platform ports:

- **pPort:** The pPort has already been discussed in the previous sub-section.It connects AppFabric-unaware services to the platform.
- **sPort:** The sPort connects AppFabric-aware **message-level** services to the platform.
- **tPort:** The tPort connects AppFabric-aware **packet-level** services to the platform.

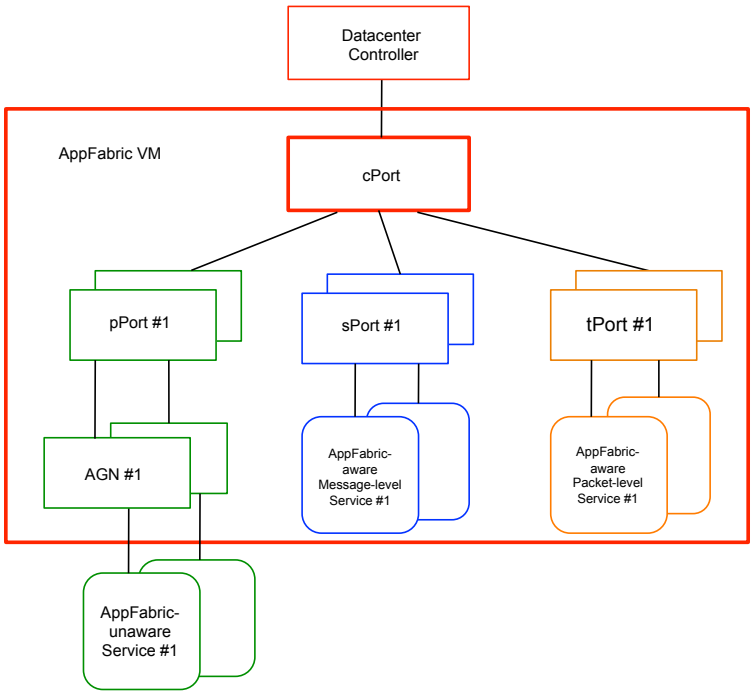

Figure 5.7: Schematic Representation of an AppFabric VM

Each of these ports are programmable. The cPort is the control plane agent in each virtual machine and is responsible for executing control plane instructions in the virtual machine. The instructions may include commands for launching a new service in the virtual machine and programming the corresponding platform port through which the service is attached to the platform; or replying to queries from the control plane regarding service liveness, resource

availability, load information etc.

When a virtual machine is launched by the global manager, it configures the address of the local controller as a parameter in its global environment. The virtual machine is pre-configured to launch the cPort at startup. The cPort reads the address of the local controller from its environment and sends a *registration message* to the local controller. On receiving this registration message, the local controller launches a new thread dedicated to managing the virtual machine. This way the local controller manages a pool of threads; each dedicated to managing a single virtual machine. The VM thread in the local controller synchronizes information on load, liveness, resource availability and other performance metrics with the host's cPort. Whenever the local controller receives any query from the global controller, it queries each of its VM threads to respond to the query. Also, it forwards commands for programming the virtual machine from the global controller to the appropriate VM thread which in turn passes it to the hosts cPort.

VMs are represented at the global controller based on their host IDs. Host IDs are flat identifiers. that unlike IP addresses do not change across host mobility. Also, these IDs are the cryptographic hash of a public key; allowing secure communication between the host and the control plane. The local controller keeps the mapping of the host ID to its locator, thus allowing the VM to migrate within the datacenter without the global controller needing to know about it. However, for inter-datacenter VM migrations, the global controller would require to update its Host ID to datacenter ID mapping.

Note that the current architecture is not very useful to mask local failures from the global controller as the global controller is exposed to individual VM identities. In the next version of the Lighthouse architecture, we plan to mask out this VM-level information from the global controller. The local controller will present the global controller with the abstraction of a single very large host and will be responsible for managing this abstraction by aggregating the resources over many different VMs.

Also note that the local controller behaves a bit differently in managing the network resources. This is because the platform can directly run its code inside a virtual machine. However, most virtual network resources (virtual switches, virtual routers, and virtual WAN links) will not allow the equivalent of the cPort and ASP-owned services to be launched over them. These virtual resources generally expose a configuration interface through which they may be programmed to some limited extent. The network controller module of the local controllers will directly access these interfaces to program the virtual network in application-specific ways as required by the ASP.

This brings us to the end of the high-level discussion on the architecture of the Lighthouse control and management plane design. The details of the actual implementation of this design will be presented in the next chapter (Chapter. 6). Note that this chapter presents the reference architecture of Lighthouse that eventually needs to be implemented. Our current prototype implementation does not implement the architecture fully but implements the key components and an extensible framework that will allow later releases to extend and add the missing components pretty easily.

Chapter 6

AppFabric Prototype

In this chapter, we will discuss the details of the AppFabric prototype implementation. The AppFabric prototype has been implemented in C and Python; with most of the control plane (Lighthouse) code implemented in Python while the data plane (OpenADN) is implemented in a mix of C and Python. The total size of the codebase is approximately 10,000 lines of code. As we will explain later in this chapter, the prototype has been implemented to allow future extensions to add new resource drivers, algorithms for resource acquisition, management and control, and new policies for deploying and delivering applications. Also, it may be noted that the current prototype does not implement the complete data plane and control plane architectures discussed in the previous chapters. We will explicitly mention what has not been implemented in the relevant portions of this discussion. The current code base may downloaded from:

`https://sites.google.com/site/applicationfabric/6-code-download`

6.1 High-level Design Issues

The AppFabric platform itself is a complex and distributed system. Therefore, before delving into the details of the implementation, let us first discuss some of the design challenges and how we address them in our implementation. Note that most of these challenges are generic to the design of most distributed systems; it is the solution that is system specific. Also

there could be many alternate solutions to each of these problems, however, we will try to motivate that our design choices are valid and acceptable for the case in point.

- **Enforcing modularity:** Modularity is a common goal with almost all system designs. It is basically a way to factor the system into a set of constituent components. In general, modularity helps achieve the following system properties:
  - **Readability and maintainability:** Code readability and maintainability is often an important goal, especially for large software projects. Modularity allows parts of a code to be developed and maintained separately and in parallel. It allows the code to be more easily maintained by restricting the effect of code changes to modules and more deterministically manage these changes.
  - **De-coupling fate:** A stronger notion of modularity is to address the problems of fate-sharing across modules. This requires that if a module fails, it should not be able to take down the whole system. Also, a malfunctioning module should not be able to directly corrupt the state of another module.
  - **Concurrency:** Modularity also introduces the idea of concurrency where each module may independently execute concurrently. This allows the system designer to make more efficient systems by partitioning his code into concurrently executable modules such that the system as a whole can perform many different tasks at the same time leading to better utilization of the hardware resources and ensuring that the system as a whole makes progress all the time.

  The most basic form of modularity is obtained through procedure (or function) calls. Procedures are modules within the same process. However, this is quite a weak notion of modularity that neither supports fate de-coupling nor concurrency. The procedure call graph is sequential and all procedures share the whole process address space. Therefore, a single malformed function can very easily corrupt the address space and the whole program can be brought down. Often a more stronger notion of modularity is required for most systems. Threads provide a stronger notion of modularity where each module runs in a separate thread within the same process. Threads share the process heap but each thread has its own stack. Therefore, threads support concurrent execution while not completely achieving fate-decoupling. The next level of modularity may be achieved by running each module in a separate process. This allows the system

to achieve all the three modularity goals. However, processes in the same host share fate in the sense that a rogue process with the right privileges may corrupt the hosts operating environment. Also, concurrency in this case is largely notional since each independent module still share the same hardware resources. Distributing the modules across different processes on different hosts provides a stronger notion of both fate-decoupling and concurrency.

Any sufficiently complex system employs all these different techniques for modularity and AppFabric is no different. In modularizing AppFabric, we made the following design choices:

- **Run platform modules and application services in separate address spaces:** The first level of modularity is enforced by always running platform modules and application services in separate processes. Therefore application services are designed as external modules that need to connect to the platform through an external communication interface. This is in contrast to an alternate implementation where the platform is provided as a special communication library that runs in the address space of the service and provides an intelligent communication substrate. Separating the platform and the application service allows the platform to independently handle failed application services without affecting other services.
- **May run platform modules in the same host within a single address space:** In each host, platform modules may run within the same address space (same process). Note the use of the word "may" in the preceding sentence. In our current implementation, all the sPort and pPort instances run in the same process whereas all the tPort instances run in a separate process. This is not a design requirement and is the case simply because we want to use the kernels network stack to interpose between message-level and packet-level communications. We could very well write our own version of the network stack and avoided this. Anyways, for platform modules running in the same process, concurrency is provided by running them in separate threads. Therefore, each port instance runs as a separate thread. This is to avoid the overhead of enforcing process level modularity; especially since it is not required within trusted modules. This also allows the platform to have some explicit control over scheduling the modules

(the ports in this case) instead of delegating it to the operating system. However, in our current implementation we do not do any explicit scheduling of the threads. These threads share the process heap and it provides threads with a cheap way to communicate with each other. However, in-order to avoid fate-sharing completely, the only way for threads to communicate inside the platform process is through messaging. Messaging is a less efficient communication mechanism between modules than shared memory but has excellent concurrency and fate-decoupling properties.

  - **No assumption can be made whether application services will be deployed on the same host or on separate hosts:** The platform requires that each application service be run as a separate process. But beyond that, it may not make any assumption whether they will be run on the same host or on separate hosts. When run on the same host, inter-process communication is the cheapest way to communicate whereas when run on different hosts the only way to communicate is through the network transport. However, this decision is taken dynamically during deployment time and hence the platform needs to provide a common communication interface to the service and automatically map it to the appropriate transport during deployment. This is exactly what the AppFabric Service Conduit (ASC) abstraction provides.

- **Synchronous vs. asynchronous bootstrap:** There are two options of bootstrapping distributed systems - synchronous and asynchronous. In a synchronous system, a strict bootstrapping order is maintained where one module is started only after another has started and so on. In an asynchronous bootstrap, each module may be started at any point in time. For a large distributed system like AppFabric involving multiple hosts in multiple datacenters, it would be highly inefficient to have a synchronous bootstrap. Also, an asynchronous bootstrap mechanism helps in making the system more dynamic. AppFabric implements asynchronous bootstrapping through a simple re-try mechanism for each connection request. If a node tries to connect to another node but fails, it re-tries after a certain interval. The interval may be fixed or adaptive. In-fact, it makes more sense to decrease the interval across successive re-trials since during the bootstrap stage the probability of the node being ready to accept the connection request becomes higher as more time passes.

- **Synchronous vs. asynchronous request/response messaging:** Another design choice that is often encountered by distributed system designers is whether to implement synchronous or asynchronous request/response messaging. In synchronous messaging each request is lock stepped with a corresponding response. In asynchronous messaging, the request response pairs are not lock stepped, thus allowing the system to handle many requests concurrently. AppFabric implements asynchronous request response messaging. Each request message is given an unique ID that is replayed in the response. This allows the sender to match a request to a response in case the request-response pairing discipline needs to be maintained. Also, each recipient maintains a message queue to queue the messages and process them in the order in which it received them.

- **Two-step vs. three-step transactions:** This is another important design choice that depends on the type of the transaction between two modules. As is the general rule, AppFabric applies three-step transaction process whenever two modules are negotiating certain parameters that requires either synchronizing or updating critical state information. As a very simple example; suppose that the central controller sends a request to a datacenter controller to start a service on any host on any port in the datacenter. The datacenter controller starts the service and sends back the host ID, host IP and port number for the service to the central controller. The central controller then acknowledges receiving this information so that the datacenter controller can delete this service-host mapping from its database since this mapping is of use only to the central controller (that keeps record of service deployments). For most other transactions that does not involve parameter negotiations or update of critical state, two-step transactions are enough.

- **Lazy updates:** The AppFabric control plane has two conflicting goals: scalability and dynamism. It needs to dynamically adapt the system to the needs of the application while at the same time be scalable to handle large application deployments. These goals are conflicting because to be dynamic the control plane has to keep an updated view of the whole system which involves either polling (from top down) or reporting (from bottom up) the state of each individual data plane node at very frequent intervals. Having a hierarchical control plane design helps to a certain extent. The other mechanism we apply is lazy updates. The way it works is that instead of reporting

system state at regular intervals, either the control plane queries the system before it needs to make a critical decision such as allocating more nodes, or the data plane nodes report an event based on a configuration; such as when the load crosses a particular pre-set threshold.

6.2 AppFabric Prototype: Structure

The AppFabric Prototype comprises of three key components:

- **Management-plane configurations:** The management plane configurations allow the ASP to specify policies for *deploying* and *delivering* their applications.
- **The OpenADN data plane:** Each data plane node (that is a virtual machine hosting a message-level or packet-level service) is part of a AppFabric Distributed Virtual Switch (ADVS) implementation. This ADVS can route and forward both, application messages as well as packets. Also, the distributed components of the switch may span across multiple datacenters. Also, the OpenADN data plane implementation exposes two types of interfaces for services to connect to the ADVS:
  - **AppFabric socket interface:** This allows newer AppFabric-aware services to connect to the ADVS.
  - **Proxy port and proxy service:** The proxy port (pPort) and the proxy service together allow legacy, non-AppFabric-aware services to connect to the ADVS.
- **The Lighthouse control plane:** The Lighthouse control plane is implemented through a set of hierarchical controller modules whose primary task involves configuring and executing ASP policies (specified through the management plane configuration files) in the OpenADN data plane.

Table. 6.1 shows the top level directory structure of the AppFabric code. In the next few sections, we will go into the specific details of each of these componeents

Directory	Sub-directory	Notes
AppFabric/containers		Container for the ASP message and packet level service implementations. Note that the *container* directory in each virtual machine image contains all the *services* that are part of the AppFabric Service Workflow. To deploy the application efficiently, the controller decides during runtime which service to fire on which virtual machine.
	OpenADN_proxy_container	The OpenADN proxy container hosts the OpenADN proxy services. Along with a proxy port (pPort), the proxy service allows AppFabric-unaware services to connect to the platform.
	OpenADN_service_container	Container for application services.
AppFabric/experiments		Experiment setups to validate the protoype.
	mininet_simulations	Code for running experiments to validate the code. Currently implements a Mininet driver. Will add drivers for other SDI stacks (e.g. OpenStack, EC2, OpenDayLight, etc.) in future releases.
AppFabric/platform		Contains the source code for the control and data plane implementations
	src/lighthouse	Control plane code.
	src/openadn	Data plane code.
	src/common	Common code for the control and data planes.

AppFabric/runtime		Contains runtime configurations and scripts.
	configurations	Configuration files through which the ASP specifies policies for *deploying* and *delivering* their applications.
	scripts	Scripts to fire the different data plane and control plane modules depending on what role the node is playing, either as a controller or a data plane node.

Table 6.1: Top-level Directory Structure of the AppFabric Code

6.3 AppFabric Prototype: Management Plane Configurations

In this section, we will discuss some of the configuration files that are exposed to the ASP for specifying application deployment and delivery policies. Most of the configuration files are written in plain XML, with a few exceptions that are written in an AppFabric-specific format. Note that these management plane configurations are pretty static and currently support very simple policy specifications. Therefore, XML seems to be a good choice to start with. However, as AppFabric evolves to support more complex policies, it may need to be replaced by a more expressive programming language. Currently, languages like Business Process Execution Language (BPEL) is the *de-facto* language used to specify business processes in SOA environments. Also, a breed of new declarative languages such as Frenetic[45] has evolved recently in the context of Software-Defined Networking for programming large, distributed networks. Standard scripting languages such as Python, Perl, Java Script, etc.

may also be used.

One may note that one of the goals of the the configuration interface design is to hide from the user as much of the details as possible. For example, message-level services, packet-level services and proxy services are three types of services that the application is composed of. These different types of services differ a lot in the way they are implemented and deployed but their configuration files look very similar. This is done on purpose to keep things as simple as possible for the application administrators.

6.3.1 Configuring the AppFabric Service Workflow (ASW)

The AppFabric Service Workflow (ASW) was discussed in chap. 1. Fig. 6.1 is a reproduction of Fig. 1.3.

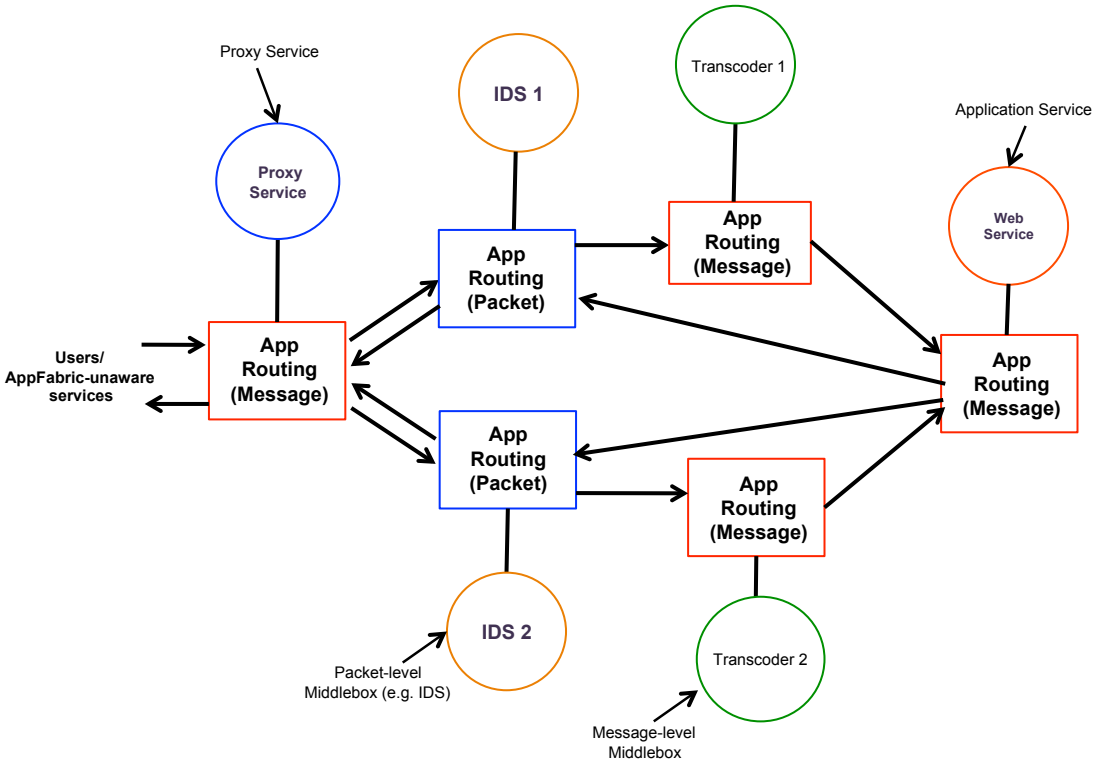

Figure 6.1: Schematic representation of an AppFabric Service Workflow. Reproduction of Fig. 1.3 from chap. 1

Now let us see how an application architect can configure the different components of an ASW.

- **Configuring the *service nodes*:** There are two types of service nodes in AppFabric - regular service node and and proxy service node. Note that a service node is not a service, but a service node hosts a service.
  - A regular service node hosts a service that may either implement some sort of application logic or a middlebox function.
  - A proxy service node hosts a **proxy** service. A proxy service sits at the boundary of **AppFabric-aware** and **AppFabric-unaware** entities. AppFabric-unaware entities may include legacy services that have not been updated yet to connect to the platform directly (through the **AppFabric socket abstraction) explained**

later, third-party services external to the ADN, or user agents such as web-browsers and mobile clients that connect to the ADN.

Listing. 6.1 lists the configuration of a service node. This configuration is stored in the file ∼**/AppFabric/runtime/configurations/workflowConfig.cfg**

Listing 6.1: Message-level Service Node Configuration

```
1
2  // Service Node configuration
3
4  <node−cfg>}
5          "node_type": "SERVICE_MSG",
6          "node_name":"Node1" ,
7          "in_ports": 3,
8          "out_ports": 3,
9          "service_name": "service_2",
10         "classifier": "test.cls",
11         "classifier_type": "regex_serial" }
12 </node−cfg>
```

Fig. 6.2 shows the configuration listed in Listing. 6.1. Note that this is the configuration of only a generic node that hosts a service. The configuration of the actual service and the way they fit into the overall architecture may vary across message-level, packet-level and proxy services and will be discussed later.

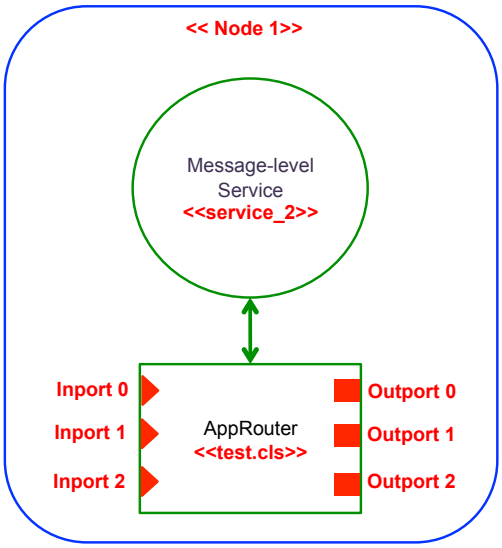

Figure 6.2: Service Node Configuration

Table. 6.2 explains the meaning of the different parameters.

attribute	**meaning**
node_type	Type of node - either, message-level service (SERVICE_MSG), packet-level service (SERVICE_PKT) or proxy service(PROXY)
node_name	Name of the node. Each node needs to have a unique name.
in_ports	Number of ingress ports in the attached App Router (refer to Fig. 6.1)
out_ports	Number of egress ports in the attached App Router (refer to Fig. 6.1)
service_name	Name of the service that the node hosts
classifier	Classifier file that has the classification rules to classify messages/packets. The classifier files are in the ∼/AppFabric/runtime/configurations/classifiers directory.
classifier_type	Many different classifier types are allowed. **classifier_type** indicates how to interpret or which module is used to interpret the **classifier**

Table 6.2: Attributes for a general node configuration and their meanings

- **Configuring the *service*:** Each service node hosts a service. These services need to be configured too. Listing. 6.2 lists the configuration of a service. This configuration is stored in the file ∼/**AppFabric/runtime/configurations/services.cfg**

Listing 6.2: Service Configuration

```
1
2  // Service configurations
3
4  <service_config>
5
6      <service>
7          <name> proxy_service_http </name>
8          <executable> proxyHTTPServer.py </executable>
9          <service_type> 2 </service_type>
10         <app_protocol> http </app_protocol>
11         <service_args> None </service_args>
12         <norm_res_index> 1.5 </norm_res_index>
13         <attrs>
14             <deployment_site> EDGE </deployment_site>
15             <deployment_type> shared </deployment_type>
16         </attrs>
17     </service>
18
19     <service>
20         <name> service_2 </name>
21         <executable> hw_server.py </executable>
22         <service_type> 0 </service_type>
23         <service_args> None </service_args>
24         <norm_res_index> 1.5 </norm_res_index>
25         <attrs>
26             <deployment_site> CORE </deployment_site>
27             <deployment_type> non-shared </deployment_type>
28         </attrs>
29     </service>
30
31     <service>
```

```
32          <name> service_P1 </name>
33          <executable> hw_server_packet.py </executable>
34          <service_type> 1 </service_type>
35          <service_args> None </service_args>
36          <norm_res_index> 3.0 </norm_res_index>
37          <attrs>
38              <deployment_site> CORE </deployment_site>
39          </attrs>
40      </service>
41
42  <service_config>
```

Listing. 6.2 lists three services; one of each type - message-level service, packet-level service and proxy service. The attributes in the service configuration are explained in Table. 6.3

attribute	**meaning**
name	Name of the service
executable	The name of the executable file that the service runs. The executables are fetched from the following directories: – **Application service (message-level/packet level):** ∼/AppFabric/containers/OpenADN_service_container/ – **Proxy service:** ∼/AppFabric/containers/OpenADN_proxy_container/
service_type	Type of the service – 0: Message-level application service – 1: Packet-level application service – 2: Proxy service (Message-level by default)
app_protocol **(only for proxy services)**	Application-level protocol that the proxy is proxying. Note that the proxy acts like a **gateway** converting different message-types and session definitions for different application-level protocols to the standard AppFabric message/session format.

service_args	Command line arguments for the service.
norm_res_index	Normalized resource index value. This value suggests how much resource the service is expected to consume normalized with respect to the service that consumes the minimum resources. The value of this index cannot be less than 1. It is used for coarse-grained resource planning across the system. The value may be determined by running separate benchmarking tests on the service.
attrs	Deployment attributes for the service. Currently, the configuration supports one attribute that determines whether the service should be deployed in a **CORE** datacenter, or a **EDGE** datacenter, or either of the two (**EDGE/CORE**.) We will explain what these terms mean in the next subsection (in the context of discussing how to configure the AppFabric Application Cloud (AAC).)

Table 6.3: Attributes for service configuration and their meanings

- **Configuring the *service graph*:** Listing. 6.3 lists the configuration of the service graph. This configuration is stored in the file ∼/**AppFabric/runtime/configurations/workflowConfig.cfg**.

Listing 6.3: Service Graph Configuration

```
1
2  // Workflow Graph
3
4  <graph>
5
6      Node0 [0] -> [0] Node1
7
8      Node1 [0] -> [0] Node0
9
10     Node1 [1] -> [0] Node2 [0] -> [1] Node1
11
12     Node1 [2] -> [0] Node3 [0] -> [0] Node4
```

```
13
14      Node4 [0] -> [2] Node1
15
16  </graph>
```

The corresponding service graph is shown in Fig. 6.3.

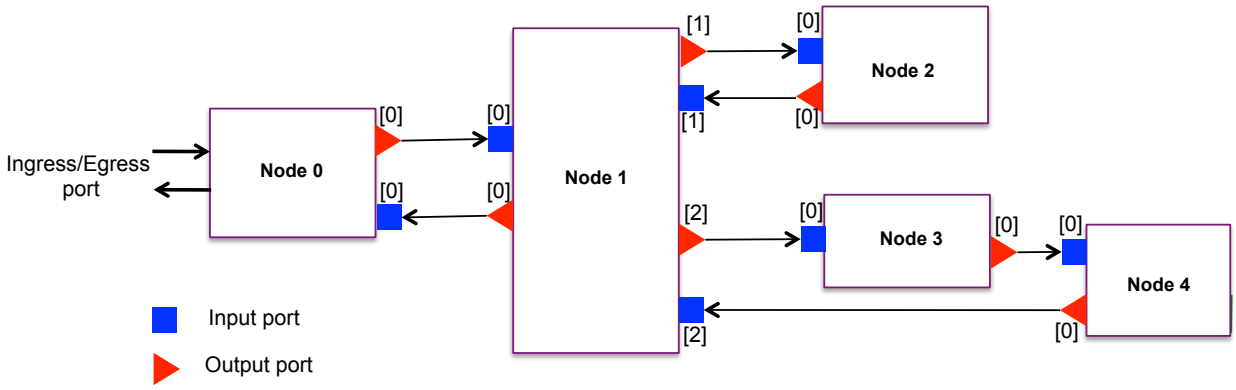

Figure 6.3: Service Graph corresponding to Listing. 6.3

This service graph (Fig. 6.3) describes the application that is then deployed based on the policies specified in the AppFabric Application Cloud (AAC) configuration described in the next subsection. However, Fig. 6.3 does not show the application router (**appRouter**)attached to the service. The application router configuration is discussed in the next point.

- **The Message Router Configuration:** Each message-level service has an attached application router that classifies the incoming message and applies routing rules to it. The AppFabric message routing is a two-step process:
  - **Step 1.** Dynamically determine the end-point (next message-level service).
  - **Step 2.** Dynamically determine the path to the end-point (packet-level services in-path).

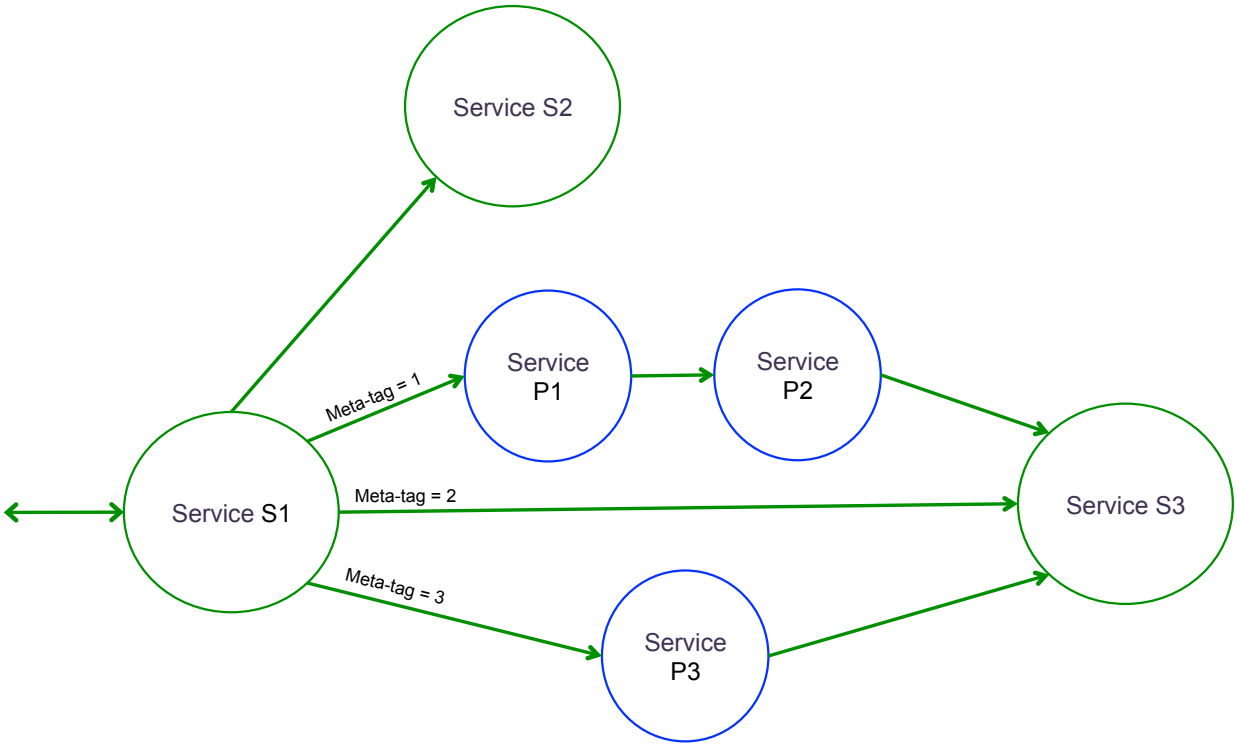

Figure 6.4: Message Routing

For example, as shown in Fig. 6.4, messages coming into service S1 may either be routed to service S2 and service S3; where all three services are message-level services. Further, messages to service S3 may be routed through three independent paths.

- **Path 1:** Through packet-level services P1 and P2.
- **Path 2:** Directly to S3.
- **Path 3:** Through packet-level service P3

Therefore, when a message comes in the application router attached to service S1, it classifies the message and determines whether the next-hop should be service S2 or service S3. If the next-hop is service S3, then it uses ***meta-tags*** to identify the different paths. The *meta-tag* is inserted in a layer 3.5 OpenADN shim header. Currently,the layer 3.5 *shim header* is implemented through setting the IP_OPTIONS field in each packet. In later versions of the implementation, a new shim layer may be implemented. It may be noted that even though the message routing is a two-step process, message

classification happens only once per message. As a result of that classification, both, the next hop message-level service as well as the packet-level path are determined. the reason for the inserting the *meta-tag* is that the packet-level services do not have access to the message-level application data and hence cannot be effectively classified. The *meta-tag* is therefore used to store the result of the message-level classification into every packet such that each packet of the message can be classified as belonging to an equivalence class as defined by the classification policies.

An example application router configuration is shown in the Listing. 6.4. This configuration can be found in *~/AppFabric/runtime/configurations/classifiers/test.cls*

Listing 6.4: The App-Routing Configuration for a Service

```
1
2  <?xml version="1.0"?>
3
4  <classifier_config>
5          <classifier in_port = "0">
6                  <rule id="1.1" pref= "1" >
7                          <hdr> ["APP_HDR", "path"] </hdr>
8                          <pattern> .html </pattern>
9                          <outport> 1 </outport>
10                         <tag> "A" </tag>
11                 </rule>
12
13                  <rule id="1.2" pref= "2" >
14                         <hdr> ["APP_HDR", "path"] </hdr>
15                         <pattern> .shtml </pattern>
16                         <outport> 2 </outport>
17                         <tag>"B" </tag>
18                  </rule>
19
20                 <rule id= "1.3" pref= "3" >
21                         <hdr> ["APP_HDR"] </hdr>
22                         <pattern> . </pattern>
23                         <outport> 2 </outport>
```

```
24                          <tag> "C" </tag>
25                  </rule>
26          </classifier>
27
28          <classifier in_port = "1">
29                  <rule id= "2.1" pref= "1" >
30                          <hdr> ["APP_HDR"] </hdr>
31                          <pattern> . </pattern>
32                          <outport> 0 </outport>
33                          <tag> "D" </tag>
34                  </rule>
35          </classifier>
36
37          <classifier in_port = "2">
38                  <rule id= "3.1" pref= "1" >
39                          <hdr> ["APP_HDR"] </hdr>
40                          <pattern> . </pattern>
41                          <outport> 0 </outport>
42                          <tag> "E" </tag>
43                  </rule>
44          </classifier>
45  </classifier_config>
```

attribute	**meaning**
classifier	**in_port** = input port number
rule	**rule_id** = rule identifier, **pref** = specifies the order of preference of matching to the rule. Note that the preference order is currently not used.
hdr	Message header fields over which the classification is performed
pattern	Pattern that is matched in the **hdr**
outport	Output Port
app_tag	Meta-tag (explained in the text following this table)

Table 6.4: Attributes for the classifier configuration and their meanings

Table. 6.4 explains the meaning of the different parameters. Note that we are aware of the fact that this form of specifying the classifier file is quite restrictive and not very useful. For example, currently it does not allow matching agains multiple header fields. This simple format was created simply to demonstrate the proof-of-concept and may not be useful to specify real-life message classification scenarios. However, the design is extendible; allowing more sophisticated **classifier** classes to be attached to the system.

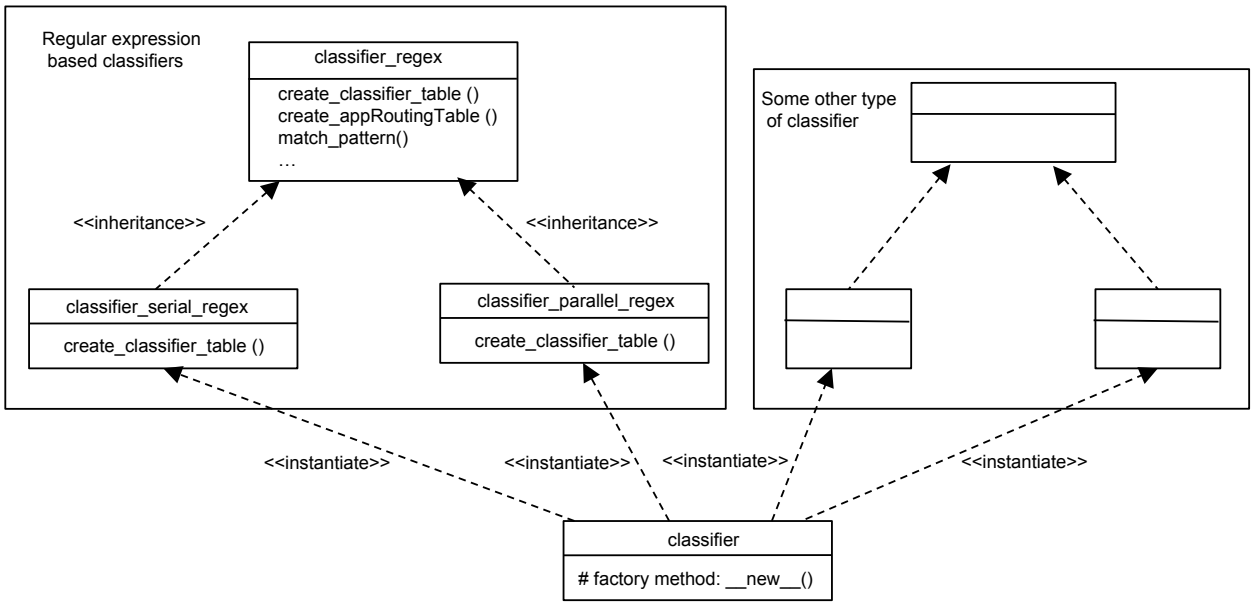

Figure 6.5: Class Diagram of the Classifier System

As shown in Fig. 6.5, the *classifier* class is a factory class (see Listing. 6.5) spawning the appropriate classifier objects of the classifier type specified in the configuration.

Listing 6.5: Classifier Class

```
1
2 class classifier :
3         # This is a factory class
4
5         def __new__(self, _classifierType):
```

```
6                   if _classifierType == "regex_serial":
7                           return classifier_serial_regex()
```

The service graph corresponding to the Fig. 6.4 is listed in Listing. 6.6. Note that in the listing we just list the nodes and not the services as shown in the figure. Services are drawn as circles and nodes are drawn as squares in all our representations. For this case assume that each service in Fig. 6.4 is hosted by the appropriate node and the mapping of service to node is as follows: S1 –> Node0, S2 –> Node1, S3 –> Node2, P1 –> Node3, P2 –> Node4, and P3 –> Node5.

Listing 6.6: Service Graph corresponding to Fig. 6.4

```
1
2  // Workflow Graph
3
4  <graph>
5
6      Node0 [0] -> [0] Node1
7
8      Node0 [1] -> P1 -> P2 -> [0] Node2
9
10    Node0 [2] -> [1] Node2
11
12    Node0 [3] -> P3 -> [2] Node2
13
14 </graph>
```

We will see in the next section, how this service graph is compiled and a running instance of this graph in the data plane is created by the control plane. Note that our configuration of the service graph does not yet allow specifying the network transport services such as quality of service paths yet. In the next version of AppFabric, we expect to add this feature as well.

6.3.2 Configuring the AppFabric Application Cloud (AAC)

After the application architect has designed the application and created the AppFabric Service Workflow (ASW), it is the responsibility of the deployment administrators to deploy the application. Fig. 6.6 shows a schematic representation of the application cloud. Listing. 6.7 provides the configuration of a typical application cloud. There are several deployment policies that the deployment administrators may want to specify.

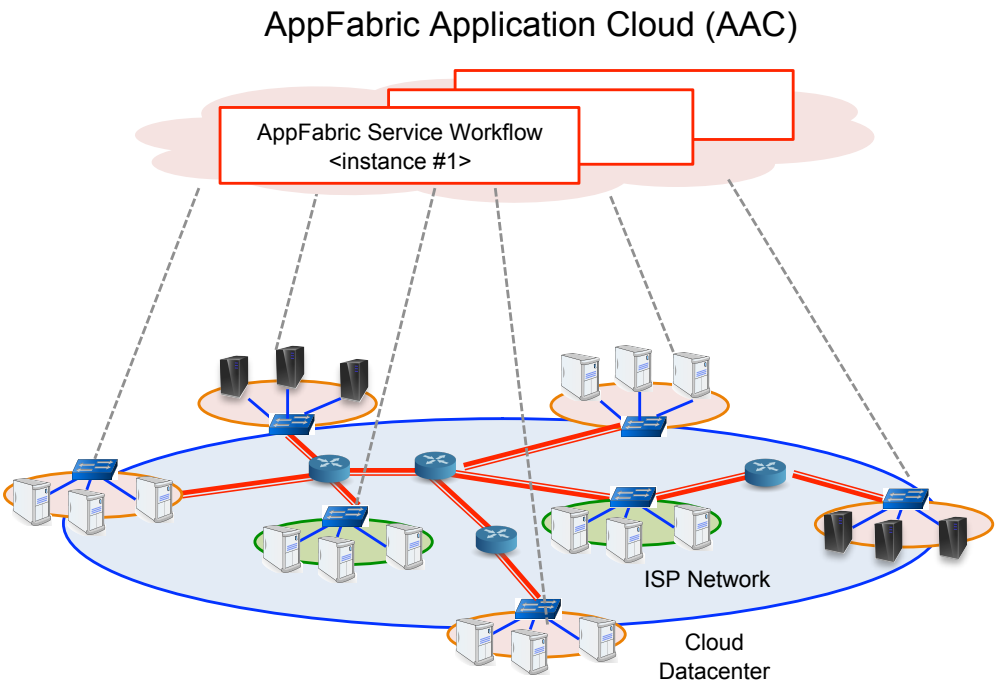

Figure 6.6: AppFabric Application Cloud(reproduction of Fig. 1.4)

Listing 6.7: AppFabric Application Cloud Configuration

```
1
2  <workflow-properties> {
3      "name": "ABC",
4      "resource_allocation_method": "greedy_max_2_site",
5      "avg_load_per_session": 5,
6      "deployment_sites":["US-E","US-W"],
7      "instance_capacity": 15,
```

```
8       "wf_per_proxy": 3,
9       "overload_notification_level": 0.5,
10      "scale_down_level":0.2 }
11  </workflow-properties>
```

Now, let us discuss in some detail what these policies are and how they are specified.

- **Policies for distributing the application across multiple datacenters distributed geographically:** AppFabric has been designed to support massively distributed application use-cases where the primary motivation for distributing the application is driven by the need to support different latency tolerances for the different services that compose the application. As an example, Internet-of-Things use-cases have many different distributed data collection and aggregation locations to support a wide range of functions varying between near-real time control to long-term business intelligence. For such applications, the topology of the application deployment (or, geographical footprint) is extremely important. Other benefits of having a distributed deployment such as fault tolerance (both in the application and in the infrastructure) and better resilience to security attacks follow naturally.

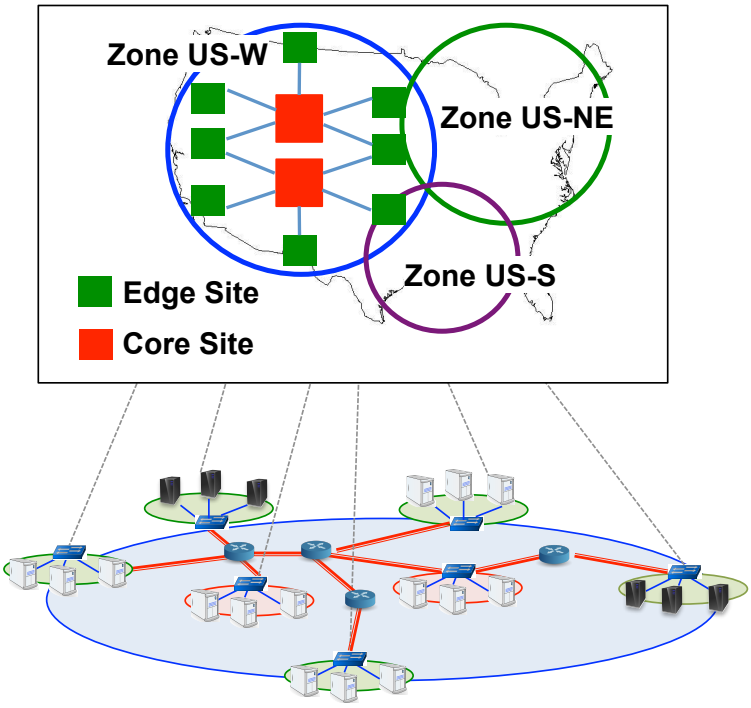

Figure 6.7: Sites and Zones (reproduced from Fig. 5.3)

In AppFabric, each service is associated with a **deployement_site** configuration specified as part of the service configuration (*see* Listing. 6.2). This attribute is used to place the service either in a **EDGE** datacenter or a **CORE** datacenter. Now let us revisit the concept of zones and sites for better clarity of the current discussion. As shown in Fig. 6.7, the deployment manager may divide a large geographical region into several **zones**. Each zone is independent of each other. For example, as shown in Fig. 6.7, the whole of United States may be divided into three zones - US-NE, US-S, and US-W. Each zone has many different **sites**. In our current implementation, sites are classified into two types - **CORE** and **EDGE**. The EDGE sites may be small micro datacenters attached to network POPs and operated by the ISPs. These massively distributed micro-datacenter infrastructure is no longer just a concept but is actually being deployed by many carriers such as AT&T and Verizon to drive Network Function Virtualization(NFV) and Internet-of-Things use cases. The CORE datacenters are relatively larger and more centrally located. Each CORE datacenter may support many EDGE datacenters. This distribution-aggregation architecture suits most of the application use-cases that we can envision at present. More intermediary levels may

be needed in the topology, especially when driving applications over very large *zones*. Our implementation supports only two levels presently and may be extended in the future to add more intermediary levels.

The **deployment_sites** attribute allows the administrator to list the different zones at which the application needs to be deployed. The application is started simultaneously at all the zones listed in this specification. We will see how the sites within a zone are selected in the following discussion.

- **Policies for acquiring the required resources:** AppFabric is designed to create an application delivery network (ADN) from resources either owned (enterprise networks and datacenters) or leased from many different resource providers (Cloud providers and ISPs). However, although AppFabric will dynamically decide (during runtime) which sites to deploy the application on, a list of all the possible sites from which this selection is made is provided in the configuration file *∼/AppFabric/runtime/configurations/sites.cfg*. Listing. 6.8 lists the configuration in this file.

Listing 6.8: Sites Configuration

```
1
2  <site_config>
3
4      <zone name="US-E">
5          <site name= "DC1">
6              <site_type> CORE/EDGE </site_type>
7               .... // authorization/authentication keys
8               .... // billing and other information
9              <site_addr> 10.10.1.0 </site_addr>
10         </site>
11
12         <site name= "DC2">
13             <site_type> CORE/EDGE </site_type>
14             .... // authorization/authentication keys
15             .... // billing and other information
16             <site_addr> 10.10.2.0 </site_addr>
17         </site>
```

```
18      </zone>
19
20      <zone name="US-W">
21          <site name= "DC3">
22              <site_type> CORE/EDGE </site_type>
23               .... // authorization/authentication keys
24               .... // billing and other information
25              <site_addr> 11.11.1.0 </site_addr>
26          </site>
27
28          <site name= "DC4">
29              <site_type> CORE/EDGE </site_type>
30               .... // authorization/authentication keys
31               .... // billing and other information
32              <site_addr> 11.11.2.0 </site_addr>
33          </site>
34      </zone>
35
36  </site_config>
```

We have already discussed most of these parameters. However, the **authorization/authentication keys** and the **billing and other information** are not part of the current implementation and will need to be added in future versions when AppFabric is tested on commercial platforms such as Amazon EC2, RackSpace. etc.

Now, let us see how the platform dynamically selects the sites from this list. The algorithm to make this selection is specified by the parameter **resource_allocation_method** in the workflow properties configuration (Listing. 6.7). Again, currently we implement a very simple greedy algorithm called the **greedy_max_2_site**. This algorithm greedily selects two sites, one EDGE and one CORE, among all the sites that has the required resources to run the EDGE and CORE services in the application service workflow. However, more complex algorithms may need to be designed based on policies for optimizing the cost of the deployment, the need for distributing the application to certain

geographical regions, etc. The implementation allows these extensions to be incorporated later. Listing. 6.9, which is in the file ∼/AppFabric/platform/src/lighthouse/globalc/selectDeploymentSite.py, shows the code snippet that allows the flexibility of attaching different methods of site selection to the platform. The implementation would be much more robust if it were implemented thorough the factory design pattern and is one of the small changes that the future version of the code should incorporate.

Listing 6.9: Selecting the algorithm for choosing deployment sites

```
1
2  def selectDeploymentSite(_siteList, _deployment_scenario):
3          if _deployment_scenario["WF_RESOURCE_ALLOCATION_METHOD"] == "greedy_max_2_site":
4                  flag, selected_coreSite, selected_edgeSite = res_allocation_greedy_max_2_site.select_site
5                                                                                                  (
6                                                                                                  _siteList,
7                                                                                                  _deployment_scenario
8                                                                                          )
9                  return flag, selected_coreSite, selected_edgeSite
```

- **Policies for scaling up and scaling down:** One of the policies the deployment administrators would like to specify is *how* and *when* to scale-up and down. Automatically scaling-up and down frees the administrator from continuously monitoring the system and dealing with intermittent periods of high/low load. The parameters that he may set to specify this are **instance_capacity, overload_notification_level and scale_down_level**.

  - **instance_capacity:** The maximum number of active user sessions that the a workflow instance can handle.
  - **overload_notification_level:** To account for the delay between signaling an overload condition and the time taken to spawn a new instance, the administrator may specify an overload notification level at which the system starts watching itself intelligently for overload situations and takes pre-emptive actions to avoid it.
  - **scale_down_level:** The deployment needs to scale down and free the resources when they are no longer needed. This parameter allows the administrator to set a scale down level; which is a measure of the load per workflow instance and the system scales down by shutting down workflow instances till the load per workflow goes above the set value of the scale_down_level parameter. Note that because of

intrinsic load_balancing in the system all the workflow_instances at any point of time has equal load.

One of the limitations of the current implementations is that it automatically starts the application with one instance per zone (as specified in the **deployment_sites** parameter). However, there must be more control on this. For example, the application should be allowed to start with more than one live instance and also the deployment administrator should be able to explicitly control how these instances are distributed. This is one of the most urgent features that need to be implemented in a future release of the platform.

- **Load Balancing:** The platform ensures that user sessions are automatically load balanced across each of the active workflow instances. Load balancing is intrinsically managed in the platform and no external policy interface is exposed to control this. In the future versions it may be useful to allow different load balancing policies to be specified explicitly as well.

- **Fault-tolerance:** The system should automatically detect service failures across workflows and automatically initiate repair efforts. This feature is partially implemented. We will see in the next section on OpenADN that the platform implements a elaborate **heartbeat** mechanism to detect and report failures. However, our current control plane implementation is not mature enough to handle these failures to provide reparative actions.

6.4 AppFabric Prototype: The OpenADN Data Plane

As discussed in chapter. 1, the main goal of the AppFabric architecture is to design and implement an AppFabric Distributed Virtual Switch (ADVS). The OpenADN data plane architecture is tuned towards achieving this goal. As shown in Fig. 6.8, designing the ADVS involves designing two key components:

- The interface between the services and the ADVS ports.

- The common message and packet switching substrate.

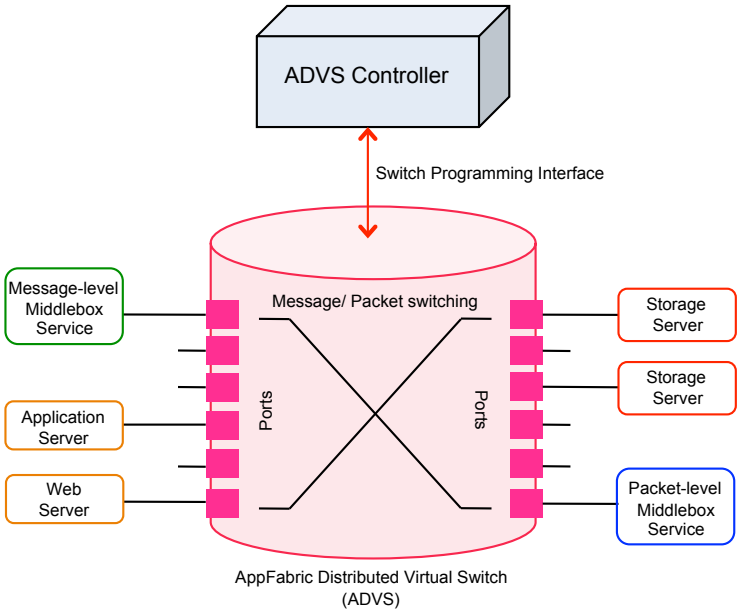

Figure 6.8: AppFabric Distributed Virtual Switch (reproduction of Fig. 3.4)

In this section we will discuss the details of these two components in the AppFabric prototype.

6.4.1 AppFabric Service Conduit (ASC) Abstraction

As shown in Fig. 6.9, the AppFabric Service Conduit or ASC is the interface between a service and the AppFabric platform. Let us discuss the two main communication channels provided by the ASC abstraction before delving into the various components that implement these abstractions in more detail in the rest of this sub-section.

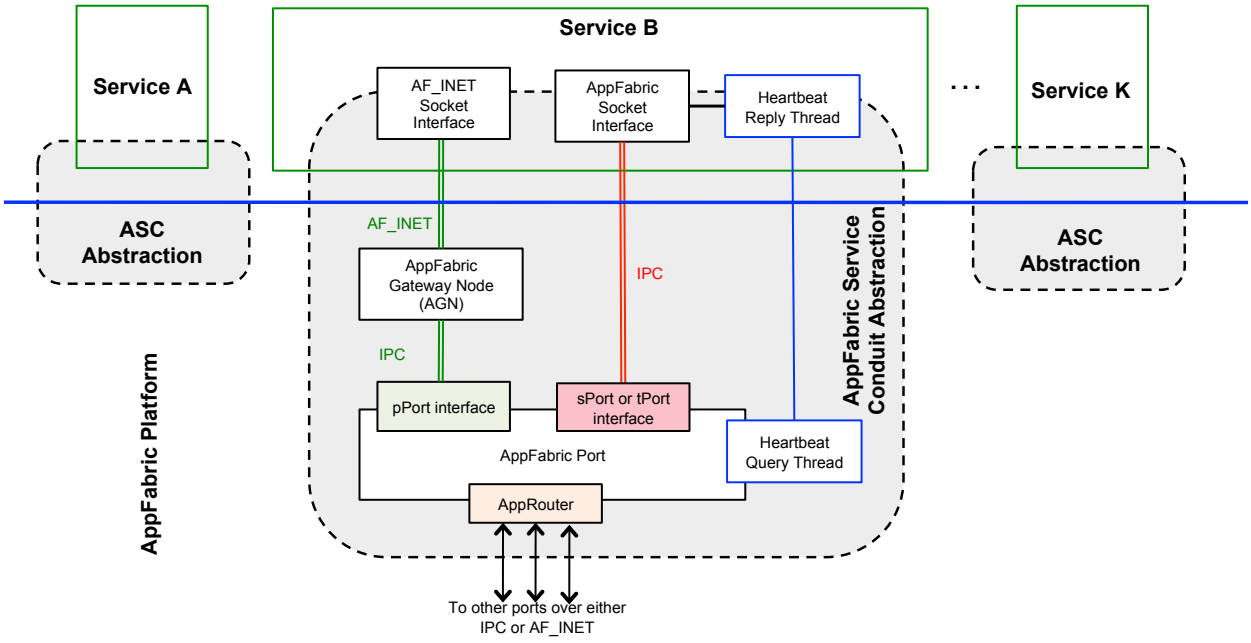

Figure 6.9: AppFabric Service Conduit Abstraction

- **Communication channel for AppFabric-aware services:** The communication channel between an AppFabric-aware service and the platform is created over an Inter-Process Communication (IPC) transport channel. The service-end (the end accessed by the service) of the channel is managed by the AppFabric socket abstraction which provides a message I/O interface from which the service may read and write application messages without having to worry about the actual destination of the messages. The platform-end (the end accessed by the platform) of the channel is managed by the AppFabric Port interface, either a service port (sPort) or a tunnel port (tPort). We will discuss the different types of ports and their architectures in more detail later in this section.

- **Communication channel for AppFabric-unaware services:**AppFabric allows legacy AppFabric unaware services to connect to the platform as well. These services do not implement the AppFabric sockets and instead communicate over the standard AF_INET sockets. To connect these services to the platform, two communication channels are spliced together at the AppFabric Gateway Node (AGN). One end of this channel uses standard AF_INET transport while the other end uses IPC. The AGN

is responsible for proxying the connection from the AppFabric-unaware service to the platform. Note that currently we assume that only legacy users and third-party service providers will have AppFabric-unaware services and therefore will need to connect through an AGN. At the platform-end, the channel between the AGN and the platform is terminated at a pPort (proxy port). We will discuss the architecture of the pPort in more detail later in this section.

Now let us look at the various components implementing the ASC abstraction in more detail.

- **The AppFabric socket abstraction:** The AppFabric socket is simply a wrapper around an Inter-Process Communication (IPC) mechanism. The service reads and writes at on end of the IPC pipe while the AppFabric port reads and writes from the other end. The service feels like it is opening a new AppFabric socket and communicating over it. The AppFabric socket automatically binds to an IPC channel pre-configured in the service's environment variable before it is started. Listing. 6.10 shows a typical sequence of calls inside an AppFabric Service to send and receive messages over the AppFabric socket.

Listing 6.10: Using the AppFabric Socket

```
1  // socketType: specifies if the socket is a message/packet socket
2
3  service_socket = appFabricSocket ()
4  service_socket.bind ()
5
6
7  // Send/Receive functions
8
9  msg = service_socket.recv_msg()
10 service_socket.send_msg(msg)
```

The **appFabricSocket()** call initializes the socket. The **bind()** call automatically binds the socket to an IPC channel. The address of the channel is automatically set into the service' environment when it is started. Another parameter that is set in the service' environment is the number of data threads servicing the socket. To the service, the socket is a single I/O interface. However, beneath this abstraction of a single I/O interface, it may be served by one or more data threads reading from and writing into the socket. This increases the effective bandwidth available at the

socket. The **send_msg/recv_msg** and the **send_packet/recv_packet** calls send and receive messages or packets. Notice that the same AppFabric Socket suffices for both message-level and packet-level services. The reason is that within the ASC components everything is treated as a message. The tPort in this case may receive packets from the hosts kernel stack but it converts these packets to OpenADN messages before delivering them to the service attached to it. The Appfabric socket library code can be found in ∼ /AppFabric/platform/src/openadn/appfabricSocketLib.

- **AppFabric Ports:** As we have already discussed, the platform-end of the communication channel is a port. There are three types of ports: sPort, tPort and the pPort. Let us discuss each of them separately in more detail.

  – **sPort:** The sPort or a service port connects to a message-level service such as an web server, storage server or a message level middlebox (e.g.Firewall, transcoder, etc.). As shown in Fig. 6.10, it has many different types of interfaces.

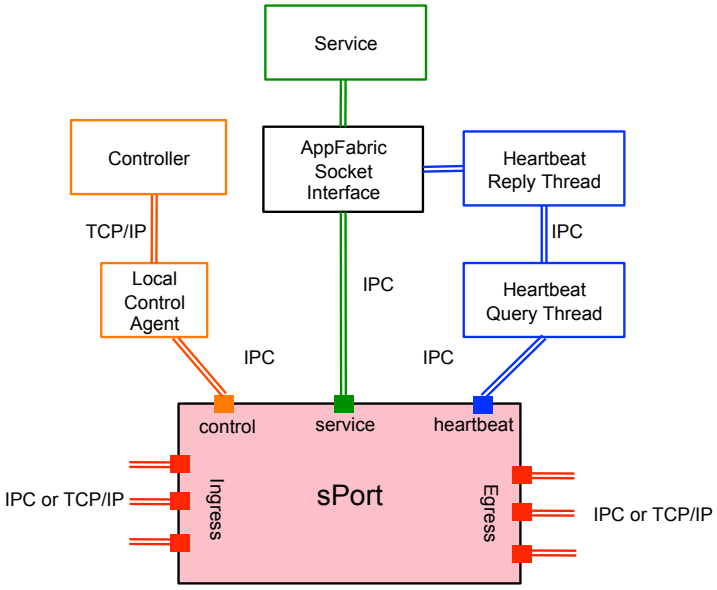

Figure 6.10: sPort - Service Port

The sPort implements the Layer 4.5 OpenADN tunnel discussed in chapter. 4. Each tunnel has an **ingress** and **egress** tunnel head. Messages are received in the sPort at an ingress tunnel head and sent out over the egress tunnel head.

Currently Layer 4.5 tunnels are implemented over TCP, although in the future other transport protocols may be used.

Messages received in the sPort are forwarded to the service attached to the port through the **service** interface . The service processes the message and sends it back to the sPort. Now the sPort classifies the packet based on the classification rules specified by the classifier configuration (Listing. 6.4). The classification determines the <**outport, meta-tag**> two-tuple. The **outport** indicates the next message-level service hop for the message. The **meta-tag** indicates the path through which the packets need to be routed through to reach the next message-level service hop. The meta-tag is included as an IP_OPTIONS field in each outgoing packet. This concept of dynamically determining the **destination** as well as the **path** has already been discussed in the last section (*see* Fig. 6.4).

The sPort is configured (tunnel setup and application routing classifiers/tables) thorough the **control** port which connects it to the local control agent that is in turn connected to a control plane controller.

Also, the sPort has an attached heartbeat thread that is connected through the **heartbeat** interface. The heartbeat thread constantly monitors the liveness of the AppFabric socket and the associated service.

– **pPort:** The pPort or proxy port connects message-level legacy services to the AppFabric platform through the AppFabric Gateway Node (AGN) as shown in Fig. 6.9. Fig. 6.11 shows the AGN architecture and its interface with the pPort.

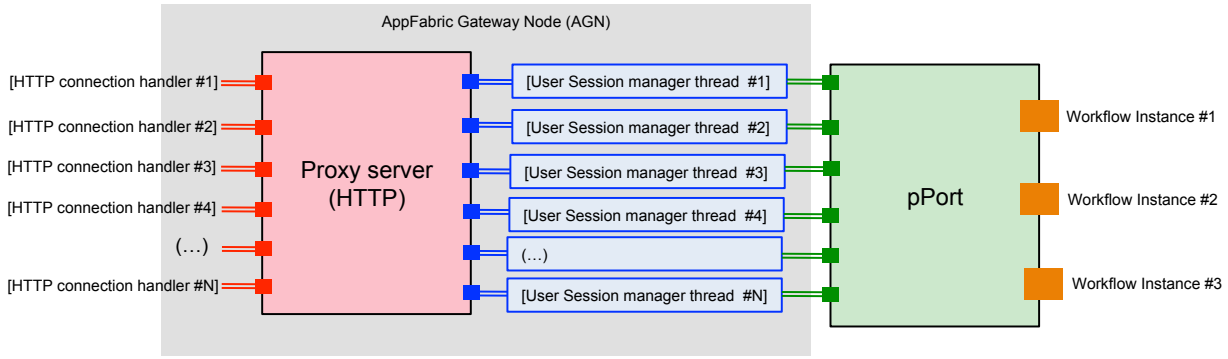

Figure 6.11: AppFabric Gateway Node - pPort interface

The AGN consists of two components - a **proxy server** and a **user session manager**. Users or third-party services speaking a legacy application-level protocol such as HTTP or JSON connect to the proxy server. The proxy server assigns a session manager thread to each connection. The session manager thread is responsible for handling all the transactions within the user session. Each application-level protocol has its own definition of a session. For example, HTPP 1.0 is a stateless protocol where each transaction is completely independent of each other. Any notion of a session has to be implemented separately at the application layer (for example, using application cookies). HTTP 1.1 introduces some notion of a session by allowing more than one transaction over the same established connection. Other protocols such as Session Initiation Protocol (SIP)[117] has very well defined concept of a session where the session start and send are explicitly signaled within the protocol. The session manager in the AGN allows the application architect to specify how different transactions from the same user may be classified into a session. For example, a simple rule could be to set an upper limit on the number of messages exchanged with a user or the time for which the connection is inactive to implicitly signal the end of a session. More robust methods such as identifying the context of a transaction from actually looking into the content of the messages exchanged and signaling the end of the session when the transaction is logically concluded could also be implemented. For example, a session could be signaled as concluded after the user successfully makes a payment on the items in his shopping cart. For protocols with explicit signaling for the beginning and the end of sessions, these explicit signaling messages may be used. The key point is that, in AppFabric, the application architect may define how the beginning and end of a session may be identified, either by looking at the content of the messages, tracking the context of message exchanges or using explicit application-layer signaling messages. Note that this definition of a session only applies to a live connection between the user and the application. The users state may be stored by the application in long term persistent storage to be used later when the user returns. But, when the user returns, his transactions would be part of another session and the stored user context can be used to drive the application-level routing policies within this new session. For example, a frequent shopper to a e-commerce portal may be allowed access to

certain discounted services as compared to a regular user. Therefore, whenever the user logs into the application, his state is fetched and based on it his transactions may be routed differently within the rest of the application service workflow.

The AGN is connected to a pPort. The pPort spawns a separate thread to handle each session manager thread. Each session manager thread is provided with an unique ID. This ID is copied into the Layer 4.5 OpenADN header of each incoming message to be able to identify the user connection in the message's return path (from the application to the user). Unlike the sPort, where each sPort hosts only one service, the pPort maybe shared among many different application workflow instances. This is because, the AGN nodes could be specialized hardware devices handling millions of users whereas the application and middlebox services may be hosted on commodity servers with limited capacity. Therefore, the AGN may need to be shared by multiple instances of the workflow to maintain resource balance. The job of the pPort is to automatically load balance user sessions across the workflow instances that are connected to it and route return messages to their appropriate session manager threads.

– **tPort:** The tPort or tunnel port allows attaching packet-level services to the application workflows. As shown in Fig. 6.12, unlike the sPort, tPort is a shared port that can be shared with more than one packet-level service and these services may belong to different workflow instances. As shown in the figure, the tPort has four types of interfaces.

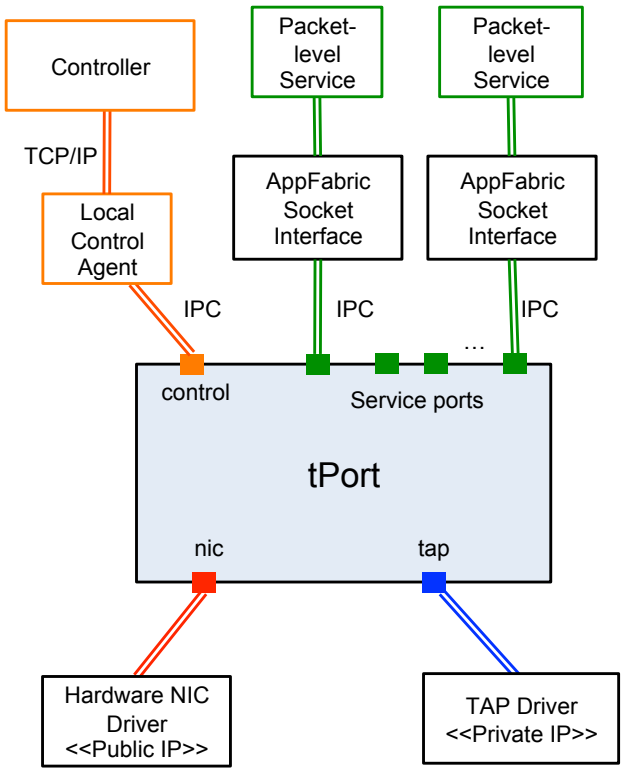

Figure 6.12: tPort Interfaces

* **control interface:** The control interface connects the tPort to the local control agent which in turn is connected to a controller node. The control plane configures/programs the tPort through this interface.
* **nic interface:** The tPort sends and receives packets from the hardware NIC device through the *nic* interface. The nic interface has a public IP address through which it can communicate with services in other hosts.
* **tap interface:** The tPort sends and receives packets from the virtual TAP device through the tap interface. The tap interface has a private IP address. The use of the TAP device and the private IP address an interesting architectural detail of the AppFabric design and will be discussed in more detail in the next subsection when we describe the common packet and message switching substrate.

* **service ports:** The service ports connect the tPort to one or more packet-level services. As mentioned earlier, the tPort may be shared between multiple packet-level services. These services may belong to different workflow instances. As shown in Fig. 6.13, the tPort implements 2-level switching. In the first level, packets are de-multiplexed based on their workflow instance ID. The workflow instance ID is also part of the the OpenADN layer 3.5 header which is currently carried in the IP_OPTIONS field. In the second stage, the packets are switched based on the **meta-tag** (also carried in the IP_OPTIONS field); between packet level services belonging to the same workflow instance.

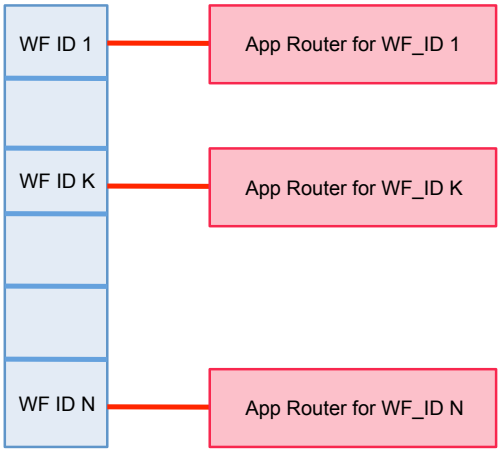

Figure 6.13: 2-level Switching in tPort

6.4.2 The common packet and message switching substrate

In the last subsection, we discussed the different types of **ports** (sPort, pPort and tPort) in AppFabric through which different types of services can attach to the platform. Apart from this, these ports also implement a distributed switching substrate for messages and packet that form the heart of the AppFabric Virtual Distributed Switch (ADVS) implementation discussed in Chapter. 3. The common packet and message switching substrate implementation makes extensive use of the ZeroMQ[143] distributed messaging library. ZeroMQ is basically a messaging library which hides almost all aspects of actual socket-level communication from the user programs. Therefore, we had to hack into the ZeroMQ source code to add support for packet tagging; that is add an IP_OPTIONS field to all packets being sent

out of a particullar socket connection. In this subsection, we will discuss how the AppFabric ports implement the common packet and message switching substrate.

The key mechanism for converging different types of services including application services, packet and message level middleboxes and network transport services onto a common switching substrate is **nested tunneling**. We discussed the high-level idea behind nested tunneling in Chapter. 4. In the current version of the AppFabric prototype (version 0.1), we do not implement the network tunnels (VxLAN or MPLS-TP). The architecture assumes that the network tunnels will be implemented; that is instantiated and configured; by the network provider. The network provider could be the cloud service provider in the case of a cloud datacenter network, or an ISP for a WAN network. The ASP does not need to implement these tunnels explicitly but rather needs to request them from the third party network providers and access them to send packets in the data plane. Therefore, implementing network tunnels in AppFabric involves extending the software-defined cloud management stacks such as OpenStack or CloudStack, or software defined network controllers such as OpenDayLight. By suggesting the APIs needed to implement AppFabric network tunnels, we hope to make useful contributions to the effort in defining define proper northbound and southbound interfaces to the software defined infrastructure space. These features will be part of our future extensions to the present AppFabric prototype.
In the current prototype implementation, we have been more interested in the design and implementation of two new tunneling mechanisms, the OpenADN Layer 3.5 tunnel and Layer 4.5 tunnel. These tunnels are unique to application delivery networks and is one of the major contributions of this work. Some of the features of the Layer 3.5 and Layer 4.5 OpenADN tunnels include:

- These tunnels are created, configured, and managed fully by the Application Service Provider.
- A common switching layer implementing Layer 4.5 and Layer 3.5 tunneling allows the convergence of message-level and packet-level services to be part of a commonly controlled infrastructure instead of the current state-of-the-art where message-level service deployments are managed by solutions such as middlewares and packet-level services(such as middleboxes) are managed by ad-hoc network management techniques.

- Switching messages and packets on this system of nested tunnels is application-aware. Each tunnel is associated with an application context and the switching mechanism dynamically establishes the application-level context of a message or packet and forwards it over the appropriate tunnel.

Fig. 6.14 shows the relationship between Layer 3.5 and Layer 4.5 tunneling in OpenADN. Note that Layer 3.5 OpenADN tunnels are a bit unique in the sense that it allows multiple-hops between the start and end points of the tunnel. Therefore, in Fig. 6.14, there is a single Layer 3.5 tunnel between Host 1 and Host 3 with Host 2 being an intermediary hop. New tunneling mechanisms such as VxLAN also implements a mechanism called VPath[138] which allows hop-by-hop forwarding within the tunnel.

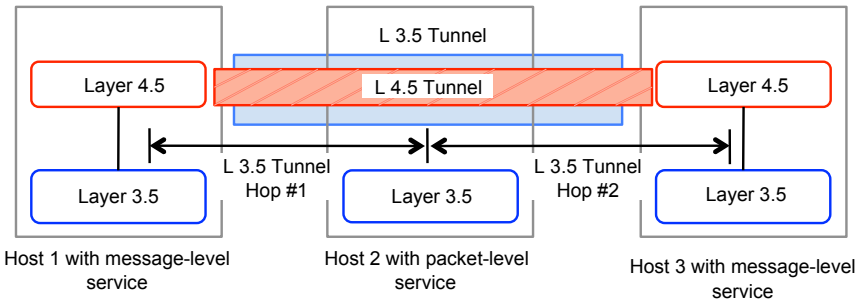

Figure 6.14: Layer 3.5 and Layer 4.5 Tunelling

Now, let us look at the details of how this tunneling mechanism is implemented by considering the following four cases.

- **Case 1: Between two message-level services on the same host (S1–> S2):** As shown in Fig. 6.15, service S1 sends out the message to its sPort, which is transported over the local IPC transport to the sPort of service S2. The Layer 4.5 tunnel is transported over IPC in this case and since there are no packet level entities, there is no Layer 3.5 tunnel.

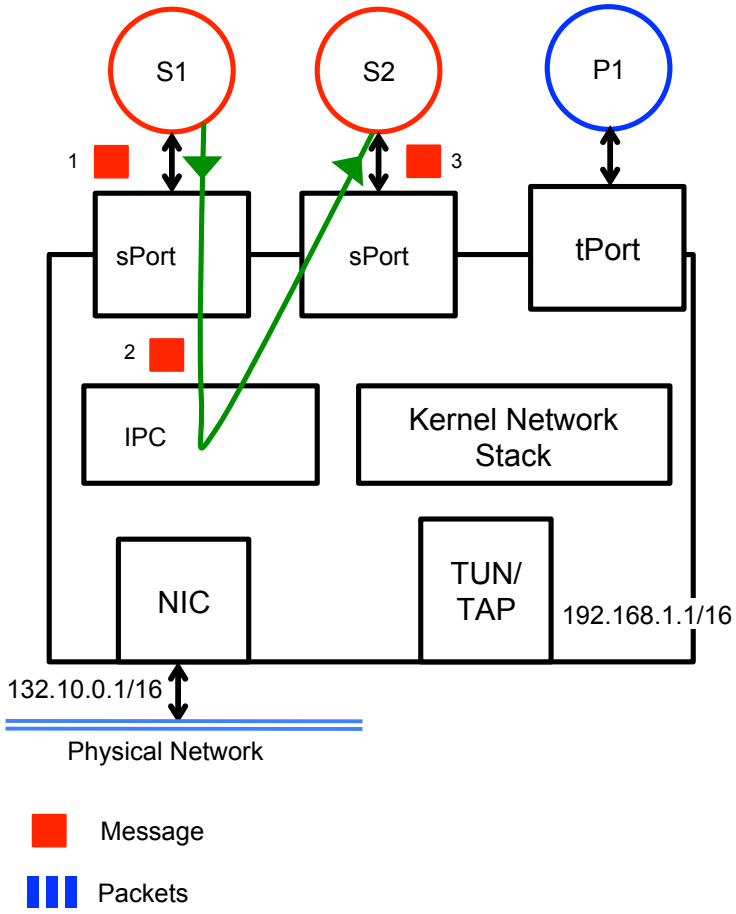

Figure 6.15: Case 1. Between two message-level services on the **same host**

- **Case 2: Between two message-level services with one or more intermediary packet-level services; all on the same host (S1–> P1 –> S2):** This case is shown in Fig. 6.16. The message-level services (S1 and S2) are attached to their corresponding sPorts while the packet-level service (P1) is attached to the tPort. Note that the tPort is shared and therefore if there would have been more than one packet level services, all of them would be attached to the same tPort.

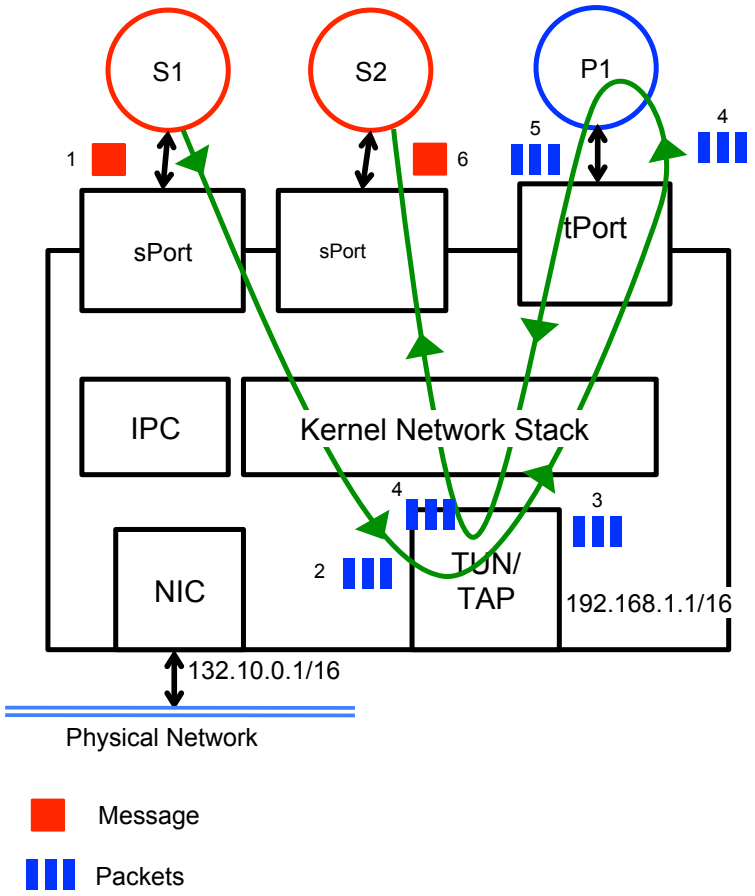

Figure 6.16: Case 2. Between two message-level services with one or more intermediary packet-level services

Now, before we explain the details of the nested tunneling mechanism, let us look at the addressing scheme. As shown in Fig. 6.17, each host has two interfaces, a physical NIC interface and virtual TUN/TAP interface. The physical NIC interface is used to communicate with other nodes and hence is assigned a externally routable public IP address (such as 132.10.0.1/16). Note that we use the term "external" instead of "global" since there may be infrastructure Network Address Translator (NAT) devices that can be used to create smaller domains within the datacenter and the addresses may need to be translated when communicating with hosts in a different datacenter. The TUN/TAP interface is used for local communication between services within the same host and hence provided with a private IP address (192.168.1.0/16). Actually the TUN/TAP can be viewed as a point-to-point device. In our case, one end of

our TUN/TAP device is attached to the host kernel (192.168.1.1/16) while the other end (192.168.1.254/16) is connected to a user-space program, which in our case is the **tPort**. The tPort is attached to both the TUN interface as well as the NIC, while sPorts are attached only to the TUN interface.

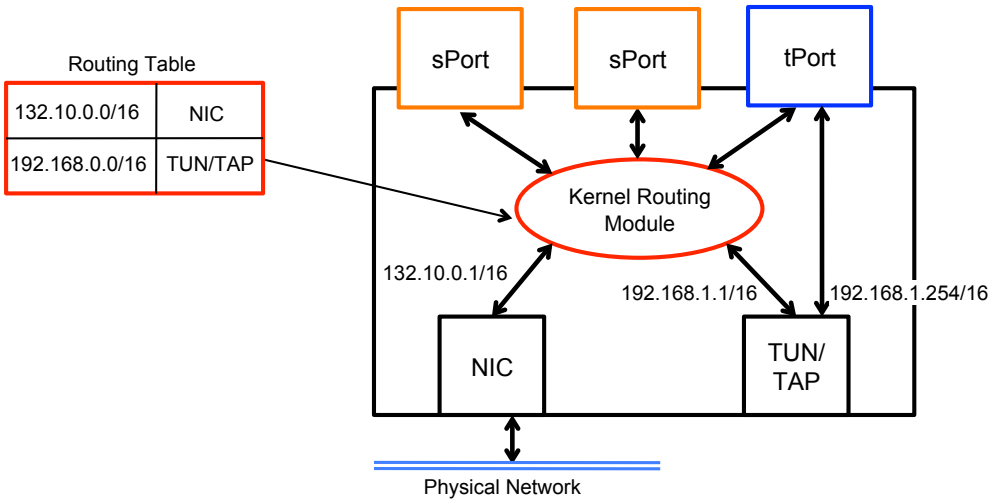

Figure 6.17: Host Interfaces

Therefore, the tPort acts like a local switch between the sPorts in the same hosts. And just like a switch, as shown in Fig. 6.18 it may host several packet level services through which it may forward the packets. The tPort uses the **meta-tag** in the OpenADN layer 3.5 header to determine which packet level services the packet needs to go through and in what order. For now consider the whole application, that is, all the services in the AppFabric Service Workflow, to be hosted in a single host. We will consider the case of multiple hosts in the next example.

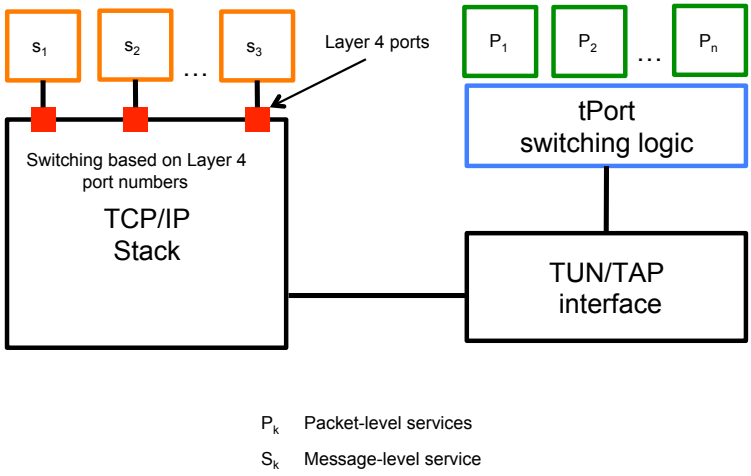

Figure 6.18: tPort Switch

So, Fig. 6.16 shows how message/packet forwarding is done for the service graph S1–> P1 –> S2. Service S1 sends out messages to its sPort (Step 1). The sPort classifies the message and determines that its bound towards service S2 through packet-level service P1. It sends out the message with the destination address set to 192.168.1.254 and the source address set to 192.168.1.1. The kernel network stack encapsulates the packet into an IP packet and may also need to fragment it. Based on the destination address the the kernel routing module will route the packets to the TUN/TAP interface (Step 2) which will forward the packets to the tPort (Step 3). The tPort will look at the OpenADN layer 3.5 header, and forward the packets to the packet-level service P1 attached to it (Step 4). Service P1 will processes the packets and send them back to the tPort. The tPort will swap the source and destination addresses of the packet (so the destination address is now 192.168.1.1) and send it back to the TUN/TAP interface. The TUN/TAP interface injects it back to the kernel network stack where the the IP packets are re-assembled (if required) and converted back to the message. The message is then delivered to service S2 by de-multiplexing over the TCP port number (Step 6).

- **Case 3: Between two message-level services on different hosts (S1–> S2):** As shown in Fig. 6.19, in the third case, a message is sent from service S1 to service S2; where service S1 and S2 are hosted on different hosts. Note that there are no intermediary packet-level services in the graph. The interesting point to note in this

example is that even though there are no packet-level services involved in this service graph, we still forward the packets through the tPort twice- once in the sending host and once in the receiving host. This is because message-level services in each host is attached to the local TUN/TAP interface which only has a local IP address that is valid only within the host. To communicate with an external remote host, the communication has to go through the physical NIC interfaces and hence the local addresses need to be converted to the externally routable addresses assigned to the NICs. Therefore, although it may be very inefficient (because the packets have to unnecessarily cross the *(*kernel-user space) boundary several times), this is the only way to nest OpenADN Layer 4.5 tunnels inside a OpenADN Layer 3.5 tunnels. To address the efficiency issue, in future version of the AppFabric prototype we plan to replace the user-space tPort implementation with a kernel module implementation.

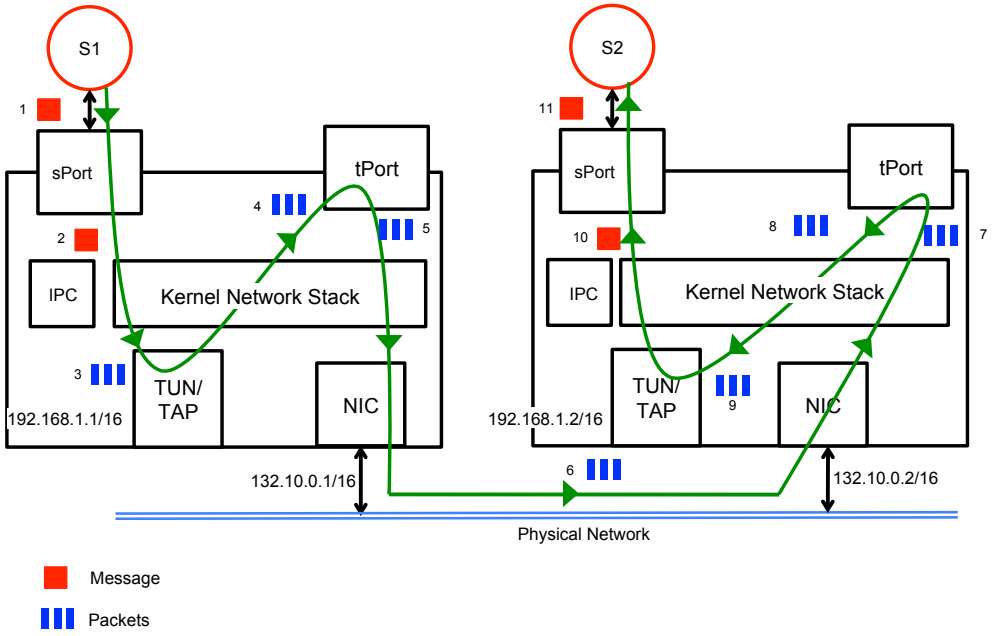

Figure 6.19: Case 3: Between two message-level services on **different hosts**

Now let us trace the steps in Fig. 6.19. Service S1 sends out a message to its sPort (Step1). The sPort classifies the message and determines that its bound towards service S2. Therefore it sends the message towards service S2 with the source address set to 192.168.1.1 and the destination address set to 192.168.1.2 (Step 2). The kernel network stack sends the message towards its destination over IP. Based on the destination

address the the kernel routing module will route the packets to the TUN/TAP interface (Step 3) which will forward the packets to the tPort (Step 4). The tPort determines the public IP addresses corresponding to the private IP address and translates the source address to 132.10.0.1 and the destination address to 132.10.0.2. It then injects the packets to the kernel network stack which forwards them to the NIC interface (Step 5). The packets are routed in the external network (Step 6) and delivered to the NIC interface of the host associated with the address 132.10.0.2. At the destination host, the tPort is the only service listening on the NIC interface and hence receives the packets from it in Step 7. The tPort has to re-translate the source and destination address of the packets to the private addresses (source = 192.168.1.1 and destination = 192.168.1.2) and inject them back into the network stack. The packets are converted back to the message and delivered to service S2 (Steps 8, 9, 10 and 11).

- **Case 4: Between two message-level services on different hosts thorough a packet level service(S1–> P1 –> S2):** Case 4 is shown in Fig. 6.20. Case 4 is very similar to Case 3 discussed above and hence the steps are very similar. The only difference is that the tPort in the sending host forwards the packets through service P1 before sending them out through the physical NIC. Also note that in the current prototype design, we are restricted to hosting all the packet-level services on the tPort of the sending host. In future versions, this restriction shall be removed with additional flexibility of hosting the packet-level services on a completely separate host. This will allow physical packet-level middlebox devices to be incorporated into the AppFabric platform if they decide to become AppFabric-aware. Currently, the AppFabric prototype only allows **virtual** packet-level middleboxes to be incorporated into the service graph.

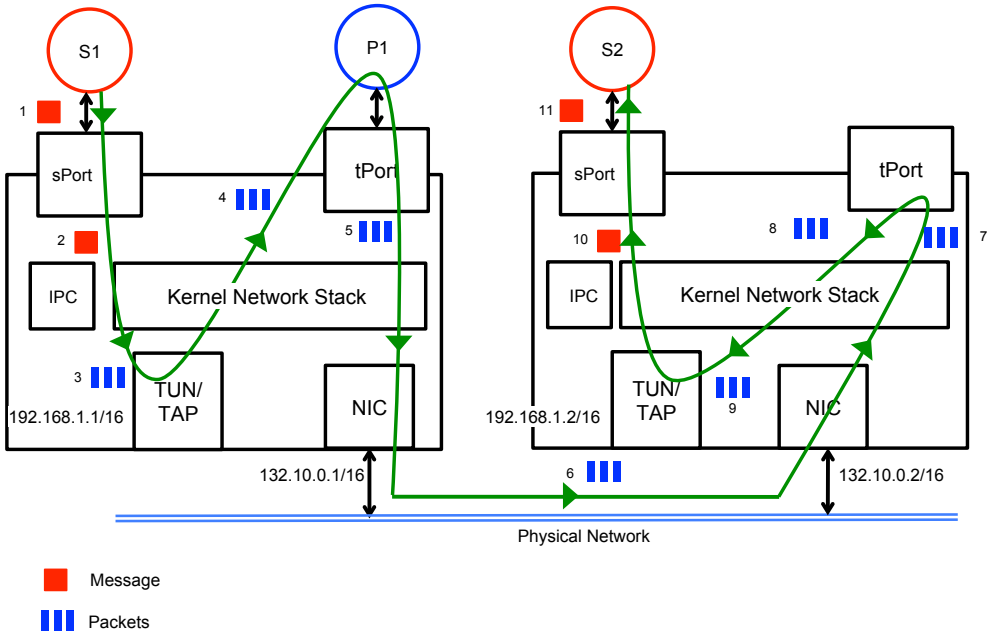

Figure 6.20: Case 4: Between two message-level services on **different hosts** thorough a packet level service

Fig. 6.21 shows the overall OpenADN architecture as it is currently implemented in the AppFabric protocol.

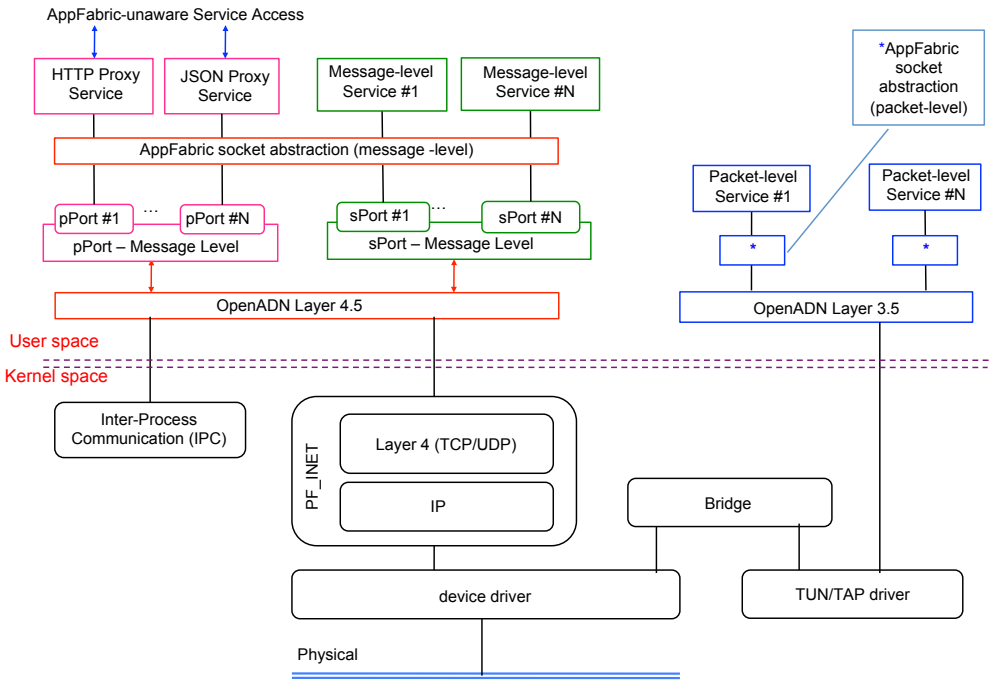

Figure 6.21: OpenADN: Overall Architecture (as implemented)

As already mentioned, implementing the tPort in user space is highly inefficient since the packets need to cross the kernel-user space boundaries several times. To address this issue, future versions of AppFabric needs to implement the tPort as a kernel module as shown in Fig. 6.22.

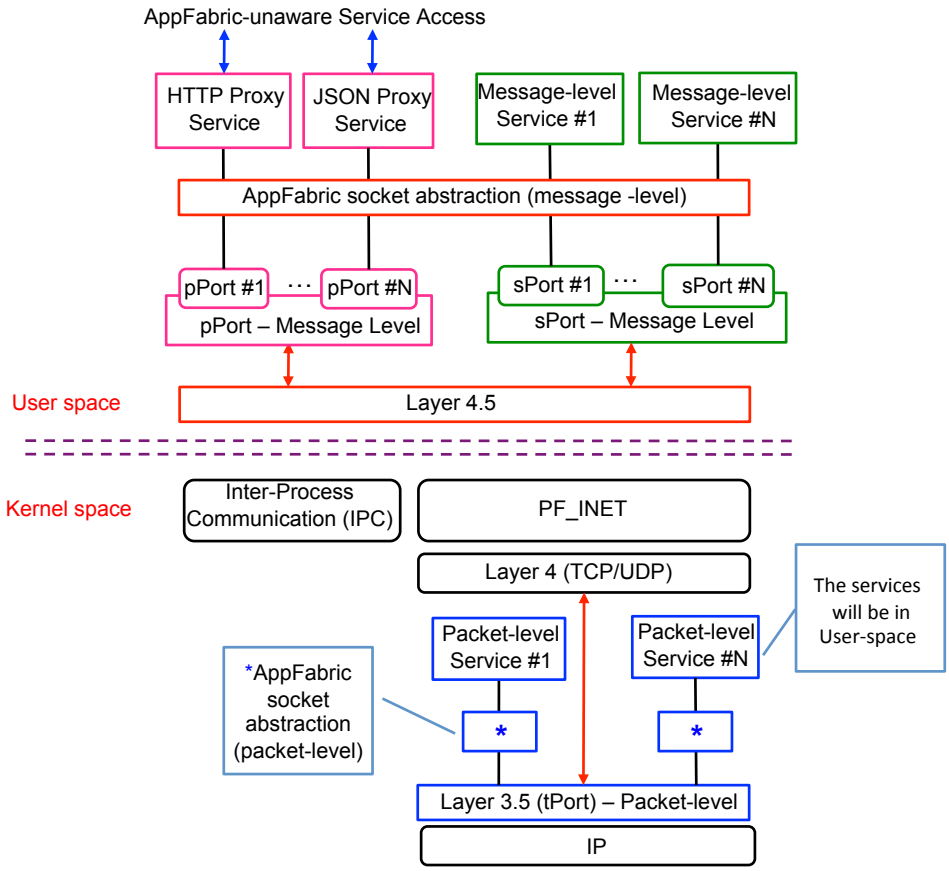

Figure 6.22: OpenADN: Overall Architecture (future implementation)

Also, in the current prototype implementation of the nested tunneling mechanism, we do not map the Layer 3.5 tunnels over VxLAN-like MAC-over-IP network tunnels or wide-area Layer 2.5 MPLS-TP tunnels. Mapping network tunnels into the OpenADN nested-tunneling mechanism is one of the future goals of the project.

6.5 AppFabric Prototype: Lighthouse Control/Management Plane

The architecture of the Lighthouse control and management plane was discussed in some detail in Chapter. 5. In this section we will discuss some of the implementation details of

Lighthouse in the current AppFabric prototype. As shown in Fig. 6.23, the Lighthouse control and management plane has a hierarchical structure. In our current prototype implementation we support only two-levels of hierarchy. However, additional levels may be added if required in future versions. Also, the current prototype does not implement the *virtual network controller* or the *virtual WAN controller*. Most of the management interfaces of Lighthouse has been discussed in the Section. 6.3. Here we will focus our discussion mostly on the control plane and management plane modules that implement these interfaces and also program the data plane nodes to enforce the policies specified by the application architects and administrators on the actual data plane traffic..

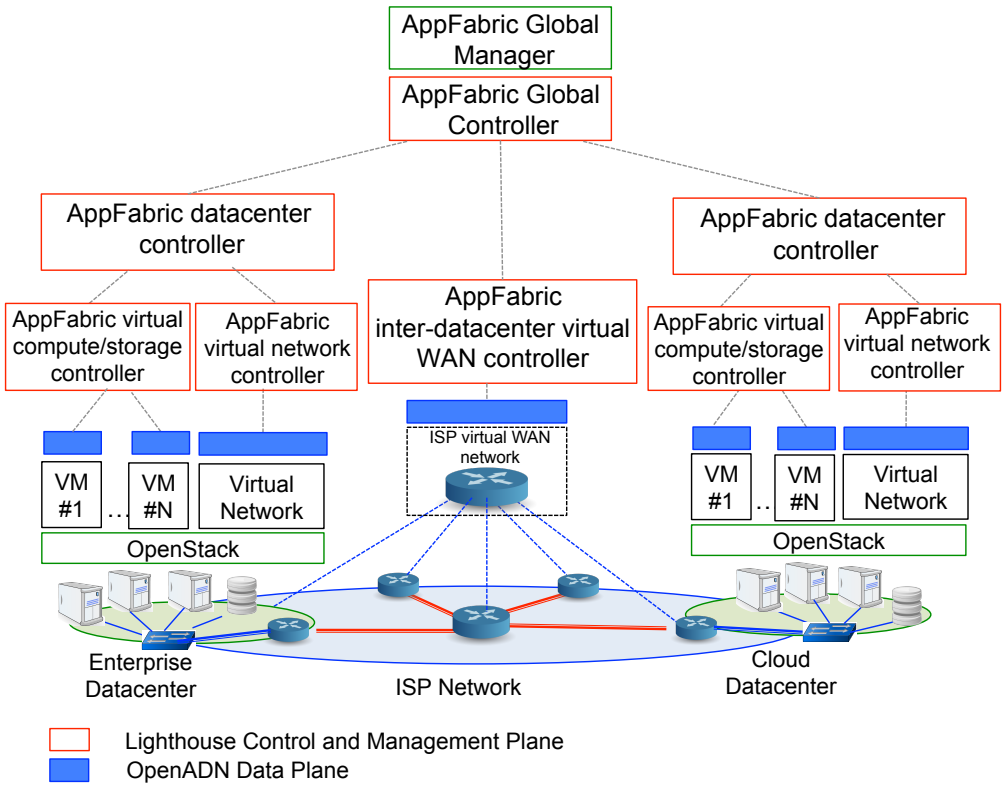

Figure 6.23: High-level Architecture of AppFabric showing the distributed data plane, hierarchical control plane and centralized management plane (reproduction of Fig. 3.2)

The Lighthouse code is written entirely in Python and is located at ~/AppFabric/platform/src/lighthouse. This directory has three subdirectories.

- **globalc:** The **globalc** directory contains the code for the global controller and the central manager modules.
- **localc:** The **localc** directory contains the code for the datacenter controller module.
- **ns:** The **fakeNameServer** directory contains the code for a *name service* implementation called the fake name server. The name server is like a DNS server mapping an application name (or url) to the locator (IP address) of an AppFabric proxy which has the required resources to handle the user session.

The **scripts directory** (∼/AppFabric/runtime/scripts) contains the scripts to start the different control modules. In the rest of this section, we will describe the code in each of these subdirectories on some more detail.

6.5.1 The Global Controller and Central Manager modules

Fig. 6.24 shows the class diagram of the global controller modules. The *global_controller_init.py* script is fired in the node that takes up the role of the global controller. This script (located in the scripts directory mentioned above) instantiates a *gc_controller* class object.

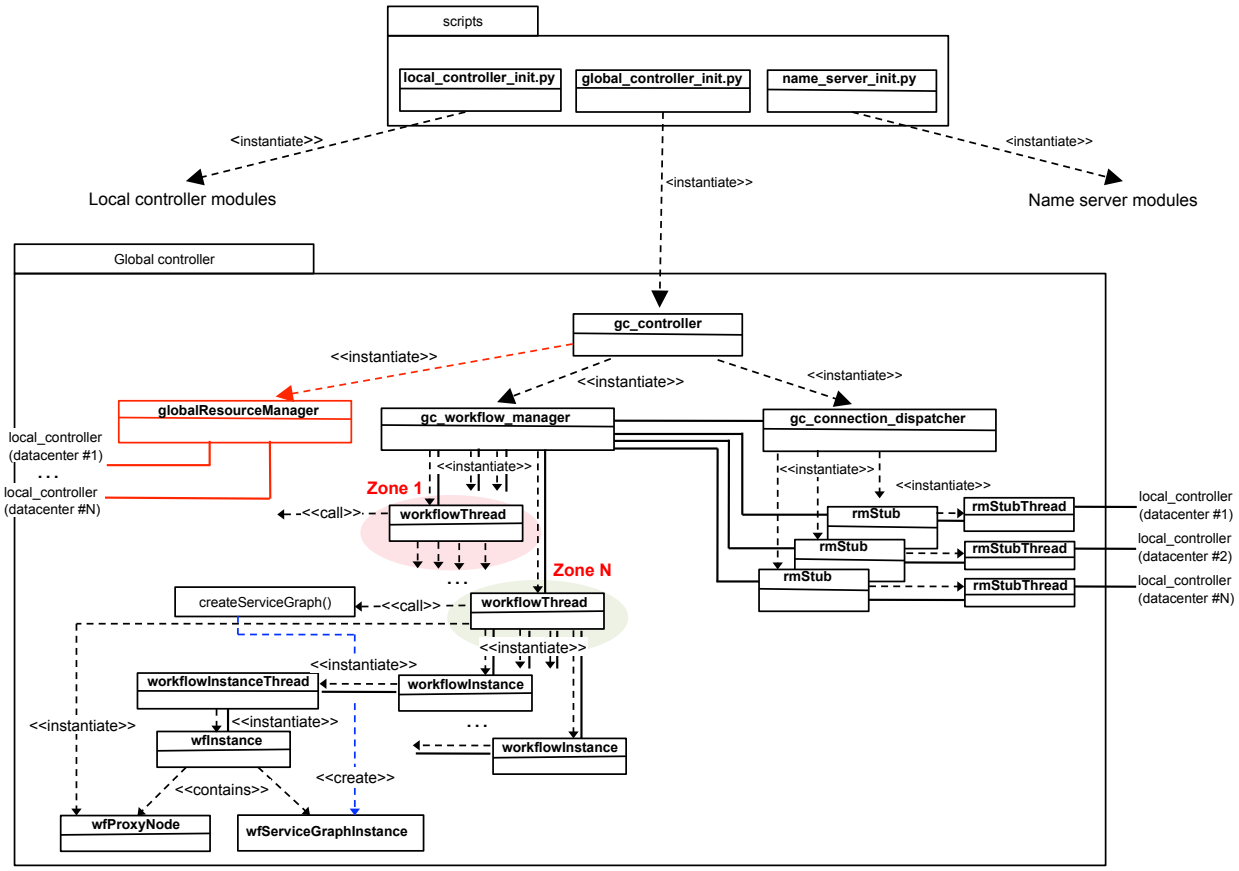

Figure 6.24: Class Diagram of the Global Controller

The *gc_controller* object instantiates three more objects:

- ***globalResourceManager Object:*** The *globalResourceManager* object talks to the resource manager modules in each local (datacenter) controller to request allocation of additional resource when required. This object has not been implemented in the current version of the code but has instead been replaced by the *mininet_driver* code in the ∼/AppFabric/experiments/mininet_simulations directory. The mininet_driver module allocates resources statically at the start of the application instead of dynamically allocating resource on-demand. In order to simulate on-demand allocation of resources, we allocate extra resources at the beginning of the application and manage this large resource pool to allow the application to grow and shrink dynamically. In future versions of the code, this resource manager subsystem will be fully implemented.

- ***gc_connection_dispatcher Object:*** The *gc_connection_dispatcher* object handles the communication between local datacenter controllers and the global controller. Whenever a new datacenter controller is started, it registers itself with the global controller. The gc_connection_dispatcher is forwarded these registration requests and it spawns a new *rmStub* object. The rmStub object acts as the local stub for each datacenter controller. The global controller accesses these stub objects locally whenever it needs to communicate with any datacenter controller; for example to pass control messages, make resource requests or query resource or application deployment state (load, liveness,etc.). These requests are either handled locally by the rmStub object or forwarded to the actual datacenter controller. The global controller oblivious to how this communication is handled. The rmStub object may need to continuously synchronize some critical state information such as those related to failure and performance with their corresponding datacenter controllers while it may resort to a more lazier query based approach for other state information such as to keep track of resource availability. For control messages that are sent to program the data plane nodes, the rmStub object will simply forward them to their corresponding datacenter controller who will in-turn forward them to the appropriate data plane nodes. Note that data plane objects are addressed in the global controller using their globally unique identifiers (UUID[75]) and not their IP addresses. The local controller is responsible for mapping the UUIDs to IP addresses for forwarding the messages to the appropriate data plane node over the IP network.

- ***gc_workFlowManager Object:*** The *gc_workflowManager* object is responsible for the deployment and runtime control of the AppFabric Service workflows. It spawns multiple *workflowThread* objects; one for each *zone* where the application is to be deployed and as specified by the application deployment administrator through the management plane interface. Each *workflowThread* object spawns multiple *workflowInstance* objects which are independent replicated instances of the service workflow. How and when these *workflowInstance* are spawned and destroyed; for instance triggered by overload/underload conditions, need to distribute the applications footprint, or handle failures in the existing active workflow instances; is again specified by the application deployment administrator through the management plane interface. Each workflowInstance needs a proxy node and a service graph instance. The proxy node is created by the *workflowThread* object and may be shared by one or more *workflowInstance*

objects. The *workflowThread* will spawn new proxy nodes when none are available for new *workflowInstance* objects. Also, the *workflowThread* calls the *createServiceGraph* function which creates the service graph instance for the workflow Instance.The creation of a service graph instance comprises of querying the appropriate datacenters for resource availability, choosing between multiple candidate datacenters that are both appropriate and have the required resources, reserving the required resources on the selected datacenters, starting the appropriate services on different datacenter nodes, and finally connecting these nodes over the common data plane communication substrate comprising of nested tunnels and application-level routing (discussed in the previous section).

Once a workflow instance is started it is also registered with the nameserver to be listed as an active application instance ready to receive user requests.

6.5.2 The Local Controller

Fig. 6.25 shows the class diagram of the local datacenter controller modules. The *local_controller_init.py* script is fired in the node that takes up the role of the local controller. This script (located in the scripts directory mentioned above) instantiates a *lc_controller* class object.

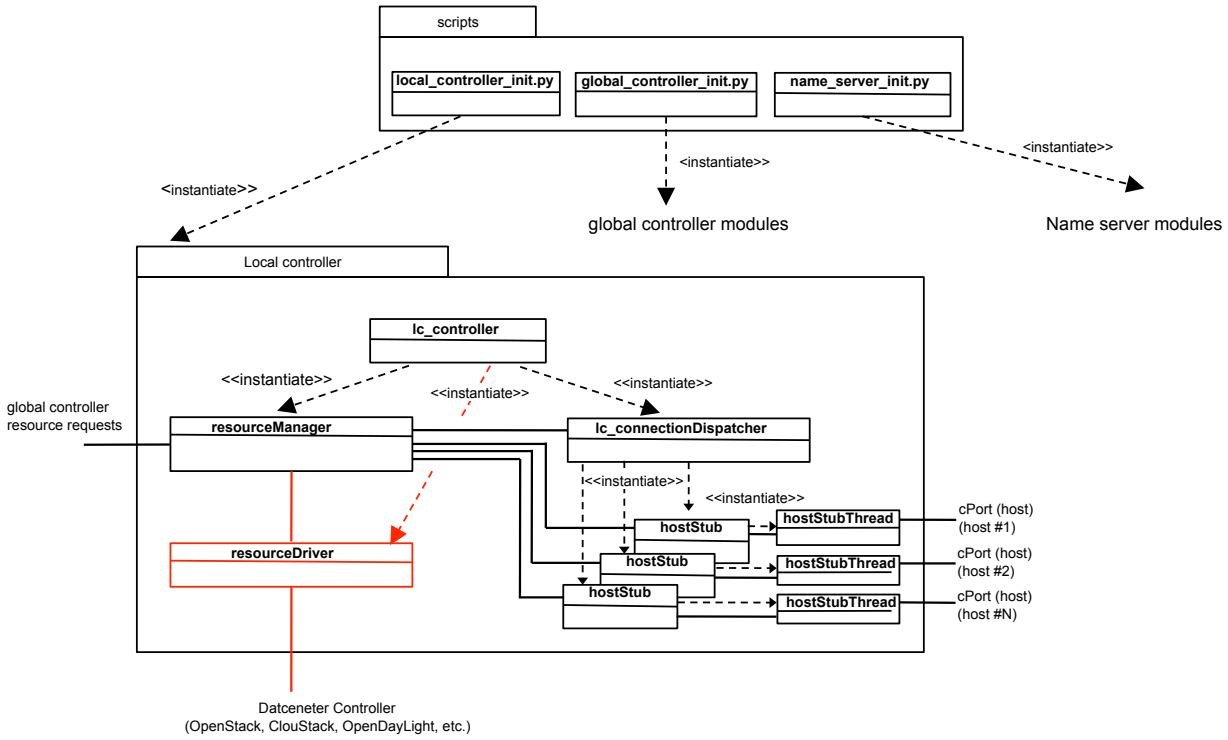

Figure 6.25: Class Diagram of the Local Controller

The *lc_controller* object instantiates two more objects:

- ***lc_connectionDispatcher Object:*** The *lc_connection_dispatcher* object handles the communication between local datacenter controllers and the data plane nodes. Whenever a new data plane node is started, it registers itself with the local datacenter controller. The lc_connection_dispatcher is forwarded these registration requests and it spawns a new *hostStub* object. The hostStub object acts as the local stub for each data plane node. The local controller accesses these stub objects locally whenever it needs to communicate with any data plane node; for example to pass control messages, query resource or application deployment state (load, liveness,etc.). These requests are either handled locally by the hostStub object or they are forwarded to the control agent (cPort) in the actual data plane node. The local controller is oblivious to how this communication is handled. The hostStub object may need to continuously synchronize some critical state information such as those related to failure and performance, with their corresponding data plane nodes while it may resort to a more lazier query

based approach to keep track of other state information such as resource availability. For control messages that are sent to program the data plane nodes from the global controller, the hostStub object will simply forward them to their corresponding data plane node cPort who will in-turn forward them to the appropriate data plane module (the appropriate OpenADN port in the node). The hostStub also maps the ID of the host to which all control plane messages from the global controller is addressed to the appropriate IP address. This allows data plane nodes to move within the datacenter; for example virtual machine mobility to optimize the resources of the datacenter . Inter-datacenter migration of virtual machines would require a handoff between the home (from where the virtual machine is migrating) and the remote (to which the virtual machine is migrating) datacenter controller. The handoff mechanism, currently not implemented in the current AppFabric prototype implementation, would need the *hostStub* object in the home datacenter controller to be destroyed and a new *hostStub* object to be instantiated at the remote datacenter controller. This would also require re-registering the host IDs at the appropriate *rmStub* objects in the global controller.

- ***resourceManagerr Object:*** The *resourceManager* object handles resource requests form the global controller. It keeps track of all the *hostStub* objects and their resource states. In future versions of AppFabric, the resource manager will also instantiate a *resourceDriver* object that will be capable of dynamically allocating more resources from the datacenter on-demand.

6.5.3 The Name Server

The third component of the Lighthouse control plane is the *name server* module. The code for this module is located at ∼ /AppFabric/platform/src/lighthouse/ns. We call our name server implementation *fake name server* since it does not comply with the standard DNS standards and may not be very useful since legacy user agents will not be able to query it. Future implementation of the name server must make it compatible with DNS.
Each workflow instance registers its workflow with the name server. The workflow is identified by an <**application name, workflow instance ID**>. The application name s similar to the concept of an **url** - a human readable name that the users want to connect to. The

nameserver keeps a mapping of all the active workflow instances corresponding to an application name and implements a simple round-robin mechanism to map the name to a workflow instance ID. However, the user is not returned the workflow instance ID, but instead returned the IP address of the proxy server for the workflow instance. Note that for a legacy user, this is no different from mapping an url to an IP through the DNS.
The name server also implements an interface with the global controller module through which a workflow instance may add, delete, activate, suspend its entry in the name server mapping. The difference between **suspend** and **delete** is that in the suspended state the workflow instance may still be returned to the user if all workflow instances corresponding to an application are overloaded and are hence suspended.

So, this concludes our discussion on the implementation details of the AppFabric prototype. There are many subtle implementation details that have been deliberately omitted from the discussion for simplicity. This documentation serves the purpose of providing some high idea on the structure of the code to developers who wish to modify/extend the design. The only way to learn the details is to dive into the actual code and parse it within the high-level framework provided in this subsection.

6.6 Prototype Validation

Now that we have discussed the design of the AppFabric prototype, let us try to validate some of its architectural claims. At this point we are not interested in studying the performance of the system since we have not yet got to the point of optimizing the performance of the different components. The purpose of the current prototype was to validate the architectural claims of AppFabric and demonstrate the viability and usefulness of such a design. Our performance evaluation is limited to running some micro-benchmarking tests in order to identify the components that attribute to the performance bottlenecks and suggesting improvements in future versions of the code.

This section is divided into two sub-sections. In the first section we will present the tests we performed to validate the architectural claims of AppFabric. In the second section we

provide the micro-benchmarking tests we performed to evaluate the relative performance of the platform.

6.6.1 Validating Functionality

To validate the functionality, we run AppFabric within an emulated mininet[72] environment. Mininet allows emulating a whole network ,running real kernel, switch and application code, within a single computer. We performed the following functionality validation tests.

- **Initial BootStrapping - Infrastructure:** Before we get to the configurations for the application deployment, let us take a look at the bootstrapping process of the platform. This step validates the claim that the platform automatically bootstraps and self configures itself and the only node that needs to be manually started is the resource manager that is responsible for allocating resources to the application from many different providers. The resource manager initiates the bootstrap process where it assigns a role to each node that is started. On startup, each node checks for the role it has been assigned by the resource manager and fires the appropriate bootstrap script to assume its role.

  In the first scenario, we emulate five datacenter networks each with five data plane nodes as shown in Fig. 6.26

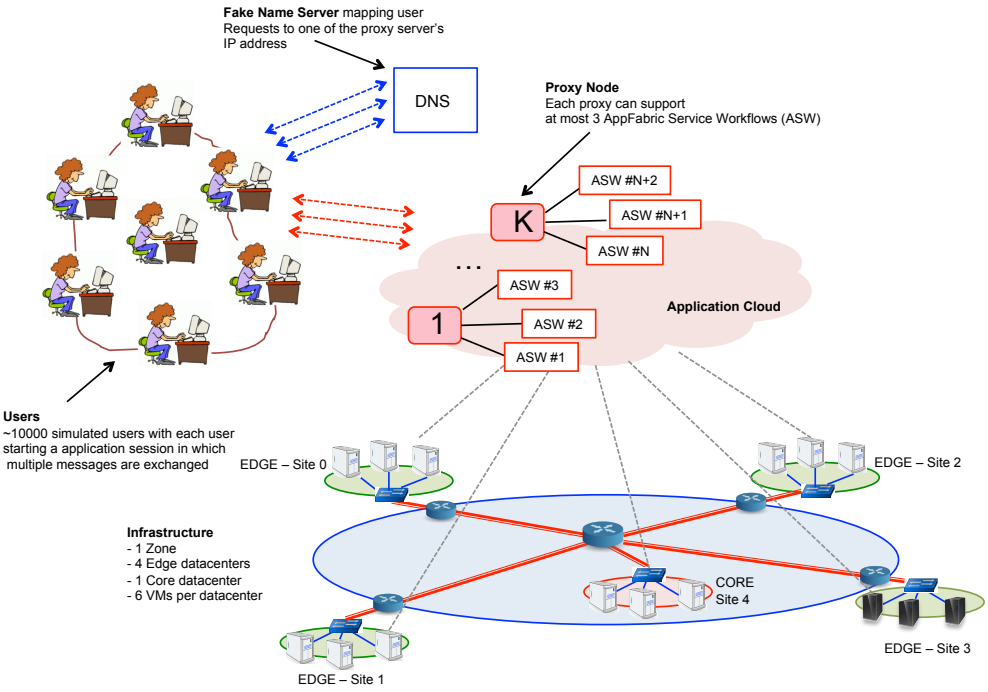

Figure 6.26: Emulation Scenario 1

Out of these five datacenters, four datacenters are EDGE datacenters that can host EDGE services only, while one datacenter is a CORE datacenter that can host CORE services. Listing. 6.11 shows the initial bootstrap messages when the system is started showing the infrastructure coming up.

Listing 6.11: Initial Bootstrap Messages - Infrastructure

```
1
2  adding global nameserver<FakeNS, 00:00:00:00:03:e8, 100.100.0.1>
3  adding global lighthouse controller <gc, 00:00:00:00:00:01, 10.10.0.1>
4  adding client host <client, 00:00:00:00:03:e9, 200.200.0.1>
5  adding lighthouse controller for site 1: <lc_s_1, 00:00:00:00:00:02, 10.10.1.1>
6  adding host to site 1: <h1s1, 00:00:00:00:00:03, 10.10.1.2>
7  adding host to site 1: <h2s1, 00:00:00:00:00:04, 10.10.1.3>
8  adding host to site 1: <h3s1, 00:00:00:00:00:05, 10.10.1.4>
9  adding host to site 1: <h4s1, 00:00:00:00:00:06, 10.10.1.5>
10 adding host to site 1: <h5s1, 00:00:00:00:00:07, 10.10.1.6>
11 adding lighthouse controller for site 2: <lc_s_2, 00:00:00:00:00:08, 10.10.2.1>
12 adding host to site 2: <h1s2, 00:00:00:00:00:09, 10.10.2.2>
13 adding host to site 2: <h2s2, 00:00:00:00:00:0a, 10.10.2.3>
14 adding host to site 2: <h3s2, 00:00:00:00:00:0b, 10.10.2.4>
15 adding host to site 2: <h4s2, 00:00:00:00:00:0c, 10.10.2.5>
```

```
adding host to site 2: <h5s2, 00:00:00:00:00:0d, 10.10.2.6>
adding lighthouse controller for site 3: <lc_s_3, 00:00:00:00:00:0e, 10.10.3.1>
adding host to site 3: <h1s3, 00:00:00:00:00:0f, 10.10.3.2>
adding host to site 3: <h2s3, 00:00:00:00:00:10, 10.10.3.3>
adding host to site 3: <h3s3, 00:00:00:00:00:11, 10.10.3.4>
adding host to site 3: <h4s3, 00:00:00:00:00:12, 10.10.3.5>
adding host to site 3: <h5s3, 00:00:00:00:00:13, 10.10.3.6>
adding lighthouse controller for site 4: <lc_s_4, 00:00:00:00:00:14, 10.10.4.1>
adding host to site 4: <h1s4, 00:00:00:00:00:15, 10.10.4.2>
adding host to site 4: <h2s4, 00:00:00:00:00:16, 10.10.4.3>
adding host to site 4: <h3s4, 00:00:00:00:00:17, 10.10.4.4>
adding host to site 4: <h4s4, 00:00:00:00:00:18, 10.10.4.5>
adding host to site 4: <h5s4, 00:00:00:00:00:19, 10.10.4.6>
adding lighthouse controller for site 5: <lc_s_5, 00:00:00:00:00:1a, 10.10.5.1>
adding host to site 5: <h1s5, 00:00:00:00:00:1b, 10.10.5.2>
adding host to site 5: <h2s5, 00:00:00:00:00:1c, 10.10.5.3>
adding host to site 5: <h3s5, 00:00:00:00:00:1d, 10.10.5.4>
adding host to site 5: <h4s5, 00:00:00:00:00:1e, 10.10.5.5>
adding host to site 5: <h5s5, 00:00:00:00:00:1f, 10.10.5.6>
Starting network
-------------

1. Starting Global Fake Nameserver (as a replacement of DNS) .... started

2. Starting Global Lighthouse Controller .... started


3. Starting Local Lighthouse Controllers:
       Starting Local Lighthouse Controller (EDGE Site)<Site= 1, 10.10.1.1> .... started
       Starting Local Lighthouse Controller (EDGE Site)<Site= 2, 10.10.2.1> .... started
       Starting Local Lighthouse Controller (EDGE Site)<Site= 3, 10.10.3.1> .... started
       Starting Local Lighthouse Controller (CORE Site) <Site= 5, 10.10.5.1> .... started


4. Starting hosts:

        Site:0:
              Starting host <Site = 1, Controller = 10.10.1.1, h1s1, 10.10.1.2> .... started
              Starting host <Site = 1, Controller = 10.10.1.1, h2s1, 10.10.1.3> .... started
              Starting host <Site = 1, Controller = 10.10.1.1, h3s1, 10.10.1.4> .... started
              Starting host <Site = 1, Controller = 10.10.1.1, h4s1, 10.10.1.5> .... started
              Starting host <Site = 1, Controller = 10.10.1.1, h5s1, 10.10.1.6> .... started

        Site:1:
              Starting host <Site = 2, Controller = 10.10.2.1, h1s2, 10.10.2.2> .... started
              Starting host <Site = 2, Controller = 10.10.2.1, h2s2, 10.10.2.3> .... started
              Starting host <Site = 2, Controller = 10.10.2.1, h3s2, 10.10.2.4> .... started
              Starting host <Site = 2, Controller = 10.10.2.1, h4s2, 10.10.2.5> .... started
```

```
64                Starting host <Site = 2, Controller = 10.10.2.1, h5s2, 10.10.2.6> .... started
65
66          Site:2:
67                Starting host <Site = 3, Controller = 10.10.3.1, h1s3, 10.10.3.2> .... started
68                Starting host <Site = 3, Controller = 10.10.3.1, h2s3, 10.10.3.3> .... started
69                Starting host <Site = 3, Controller = 10.10.3.1, h3s3, 10.10.3.4> .... started
70                Starting host <Site = 3, Controller = 10.10.3.1, h4s3, 10.10.3.5> .... started
71                Starting host <Site = 3, Controller = 10.10.3.1, h5s3, 10.10.3.6> .... started
72
73          Site:3:
74                Starting host <Site = 4, Controller = 10.10.4.1, h1s4, 10.10.4.2> .... started
75                Starting host <Site = 4, Controller = 10.10.4.1, h2s4, 10.10.4.3> .... started
76                Starting host <Site = 4, Controller = 10.10.4.1, h3s4, 10.10.4.4> .... started
77                Starting host <Site = 4, Controller = 10.10.4.1, h4s4, 10.10.4.5> .... started
78                Starting host <Site = 4, Controller = 10.10.4.1, h5s4, 10.10.4.6> .... started
79
80          Site:4:
81                Starting host <Site = 5, Controller = 10.10.5.1, h1s5, 10.10.5.2> .... started
82                Starting host <Site = 5, Controller = 10.10.5.1, h2s5, 10.10.5.3> .... started
83                Starting host <Site = 5, Controller = 10.10.5.1, h3s5, 10.10.5.4> .... started
84                Starting host <Site = 5, Controller = 10.10.5.1, h4s5, 10.10.5.5> .... started
85                Starting host <Site = 5, Controller = 10.10.5.1, h5s5, 10.10.5.6> .... started
86    5. Starting client host .... started
```

First, the resource manager adds resources from different sites to the global resource pool. In this simulation, it allocate the resources from Mininet through the mininet driver script. In future versions, it should start separate threads for each resource provider (cloud datacenter, ISP network, etc.) that will in turn spawn resource drivers capable of consuming the APIs of the datacenter's management software stack such as OpenStack or EC2 to negotiate and allocate the resources. The resource manager also allocates a specific role to each node and configures it accordingly. On starting, the node assumes its role as shown in the listing where we can see the different types of nodes getting started including the fake name server, the global controller, local datacenter controllers for each site and data plane nodes. Also in Step 5 (line 86 in Listing. 6.11), a client host is started. For our experiments, the client host simulates around 10,000 users, each starting a separate user session with the application.

- **Initial Bootstrapping: Platform Components:** After the infrastructure boots up, each infrastructure node assumes its specific role that has been assigned to it within the AppFabric platform and together run a distributed mechanism to automatically bootstrap the whole platform and self-configure based on the the policies specified by

the application administrator. All the platform components have been discussed in the previous sections and we will not repeat that discussion here.

Listing. 6.12 shows the different steps in the auto bootstrap process of the AppFabric platform. Note that this distributed bootstrapping mechanism has been designed to be completely **asynchronous**. This removes node-level dependencies in the proper functioning of the platform.

Listing 6.12: Initial Bootstrap Messages - Platform

```
1
2
3  gc: WFM: Initializing workflow manager
4  WFM: Starting workflow manager
5
6  WFT <ABC, US-E>: started
7  WFT <ABC, US-E>: connected to fake name server port
8  GC Controller:Workflow Manager(WFM) started...will start Global Connection Dispatcher(CD) now
9  Global CD: Intializing GC connection dispatcher
10 Global CD: initialized
11 Global CD: Starting gc connection dispatcher
12 Global CD: Ready to receive connection requests from Datacenter Controllerss
13 WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: started
14 WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: Requesting proxy Node
15 WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: Requesting proxy Node
16 WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: Requesting proxy Node
17 RM Stub <1dcc3c64-da20-4bc6-a51a-876603bba99b: 10.10.2.1, EDGE>: InitializedWFM:<site_2>: available resource = 0
18
19 RM Stub <9646d4b5-12a3-4702-b6ed-0066bb446e42: 10.10.3.1, EDGE>: InitializedWFM:<site_3>: available resource = 0
20
21 RM Stub <e5cafa30-f5a6-4166-b0fd-06e6acbaf739: 10.10.1.1, EDGE>
22
23 lc_s_1: Datacenter controller <US-E:site_1>: ....Starting resource manager (RM)
24 Datacenter controller <US-E:site_1>: Resource Manager(RM) started...will start Connection Dispatcher(CD) now
25 Datacenter controller <US-E:site_1>: Starting connection dispatcher...
26
27 lc_s_2: Datacenter controller <US-E:site_2>: ....Starting resource manager (RM)
28 Datacenter controller <US-E:site_2>: Resource Manager(RM) started...will start Connection Dispatcher(CD) now
29 Datacenter controller <US-E:site_2>: Starting connection dispatcher...
30
31 lc_s_3: Datacenter controller <US-E:site_3>: ....Starting resource manager (RM)
32 Datacenter controller <US-E:site_3>: Resource Manager(RM) started...will start Connection Dispatcher(CD) now
33 Datacenter controller <US-E:site_3>: Starting connection dispatcher...
34
35 gc: : InitializedWFM:<site_1>: available resource = 0
36
37 WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: Requesting proxy Node
38 WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: Requesting proxy Node
39
40 gc: WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: Requesting proxy Node
41
42 gc: WFM:<site_1>: available resource = 10000
43 sending proxy resource req to 10.10.1.1
44 WFM:<site_2>: available resource = 10000
45 WFM:<site_3>: available resource = 10000
46 WFM:<site_3>: available resource = 20000
47 WFM:<site_3>: available resource = 30000
```

```
48 WFM:<site_1>: available resource = 20000
49 WFM:<site_2>: available resource = 20000
50 WFM:<site_2>: available resource = 30000
51 WFM:<site_2>: available resource = 40000
52 WFM:<site_1>: available resource = 30000
53 WFM:<site_1>: available resource = 40000
54 WFM:<site_2>: available resource = 50000
55 WFM:<site_1>: available resource = 50000
56 WFM:<site_3>: available resource = 40000
57 WFM:<site_3>: available resource = 50000
58
59 WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: Requesting proxy Node
60
61 gc: WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: proxynode initialized, get resources for the rest
62 WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: Requesting resources
63
64 gc: WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: WFINSTANCE_WFT_GET_RESOURCES_REP failed
65
66 gc: WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: WFINSTANCE_WFT_GET_RESOURCES_REP failed
67
68 h1s1: Starting server, use Ctrl-C to stop
69
70 h1s1: HTTP PRoxy: Add threads to the threadpool ===== 10
71
72 h1s1: HTTP Proxy Num threads alloted ================== 0
```

In line 3, the global controller(gc) initializes a the workflow manager (WFM). The WFM spawns an initial workflow thread (WFT) in line 6 for application "ABC" in the zone "US-E." Note that in this experiment we have only one zone. In line 7, the workflow thread registers this application instance with the fake name server (replacement of the DNS mapping system in our prototype). The name server does not yet advertise this mapping. It will do so only when the WFT explicitly activates the mapping entry. The other thing the WFT does on being started is to get a proxy node allocated to itself (line 14) that will be the interface between the workflow services and the external users. A proxy node is shared between many WFTs each of which are replicated instances of the same application. The WFM is responsible for allocating a proxy node to each WFT. However, the WFM does not yet have the resources to start the proxy node and hence it is not able to allocate one to the requesting WFT. Instead of the WFM keeping state of this requests, it simply sends back a "REQUEST_FAILED" type message to the WFT. It is now the responsibility of the WFT to try again later. The WFT runs an exponential backoff mechanism to repeat its request instead of flooding the system with useless request messages. The WFM on the other hand is independently trying to get the resources required to allocate a proxy node.

On the other side, each datacenter controller boots up independently and tries to register itself with the *gc*. In the listing the local datacenter controllers may be identified as **lc_s_[site_num]**. the lc stands for "local controller" while the s_[site_num] represents the site number. line 23-25 shows the datacenter controller for the site 1 coming up and starting a connection dispatcher. Apart form registering with the *gc*, starting the connection dispatcher is the other job that the datacenter controller needs to do during bootstrap. The connection dispatcher allows data plane nodes to register with the data center controller.

In line 35, you can see the WFM making a resource request to site 1 (after the datacenter controller has registered with the **gc**). However, site 1 offers no resources to the WFM since none of its data plane nodes have registered their resources with it and so it does not have any available resources. The WFM would try other sites and repeat these requests later (based on an exponential backoff mechanism) till it gets the required resources. In the meantime, the WFT thread keeps on polling the WFM for a proxy node(lines 37-40).

In lines 42-57, we see the WFM flooded with resource updates from the different sites. Each data plane node in our experiment reports 1000 units of resources and hence the total resource available per-site is 5000. Currently, we implement a greedy mechanism for resource selection and hence, as can be seen in line 43, the WFM sends the request to allocate a data plane node to run the proxy service to the first site (site_1 in this case) that reports enough available resource.

In line 61, we can see that the proxy node has been initialized and now the WFM starts gathering the resources to deploy the other services within the workflow. In line 68, you can see that the proxy service has been started on node *h1s1* (host 1, site 1).

We will narrate the next part of the story in the next point that shows the steps in starting a workflow that involves starting the service on the different data plane nodes and configuring application-level routing policies into the platform ports through which the service connect to the platform.

- **Initial Bootstrapping- Starting a Workflow:** Once the required data plane nodes with enough resources to run the workflow has been identified, the next step is to actually start the application services on these nodes and setup the message and packet routing services. Listing. 6.13 lists these steps. In this listing we have replaced some of the output with "..." to fit the output within the width of the page.

Listing 6.13: Initial Bootstrapping - Starting a Workflow

```
1
2  gc: WFT <ABC, US-E, 1abf7081-58b0-410c-8f98-ce4648b093a4>: Starting services
3  h3s2: Hello! I am service_3:d9c2c88b-5455-40d0-8a9c-e8168d9b52ec
4  h4s2: Hello! I am service_4:89820cb5-79a2-414b-b2f4-6cec4a7ce230
5
6  h3s2: service3: connecting to appfabric socket
7
8  h2s2: Hello! I am service_2:cb15f190-9cff-495b-8622-2ea82edcfe52
9
10 h4s2: service4: connecting to appfabric socket
11
12 ....
13 ....
14
15 h3s2: hearbeatREP initialized
16
17 h4s2: hearbeatREP initialized
18
19 ....
20 ....
21
22 h3s2: heartbeat REQ thread started for service: service_3
23
24 h1s2: service5: connecting to appfabric socket
25
26 gc: WFT <ABC, US-E, ... >: Service <service_3> <d9c2c88b-5455-40d0-8a9c-e8168d9b52ec> started on host:10.10.2.4
27
28 gc: WFT <ABC, US-E, ... >: Service <service_4> <89820cb5-79a2-414b-b2f4-6cec4a7ce230> started on host:10.10.2.5
29
30 gc: WFT <ABC, US-E, ... >: Service <service_2> <cb15f190-9cff-495b-8622-2ea82edcfe52> started on host:10.10.2.3
31
32 gc: WFT <ABC, US-E, ... >: Service <service_5> <2ab20b69-16ea-4e0d-9fb9-77c55bf5896e> started on host:10.10.2.2
33
34 gc: WFT <ABC, US-E, ... >: WF attached to proxy node <proxy_service_http> on <10.10.1.2>
35
36 gc: WFT <ABC, US-E, ... >: setting up links
37
38 h4s2: Service Port: Adding ingress
39
40 h1s2: Service Port: Adding ingress
41
42 h3s2: Service Port: Adding ingress
43
44 h2s2: Service Port: Adding ingress
45
46 h2s2: Service Port: Adding ingress
47
48 h2s2: Service Port: Adding ingress
49
```

```
50  gc: WFT <ABC, US-E, ... >: service_2[2] --> [0]service_4
51
52  gc: WFT <ABC, US-E, ... >: service_4[0] --> [0]service_5
53
54  gc: WFT <ABC, US-E, ... >: service_2[0] --> [0]proxy_service_http
55
56  gc: WFT <ABC, US-E, ... >: service_2[1] --> [0]service_3
57
58  gc: WFT <ABC, US-E, ... >: proxy_service_http[0] --> [0]service_2
59
60  gc: WFT <ABC, US-E, ... >: service_3[0] --> [1]service_2
61
62  gc: WFT <ABC, US-E, ... >: service_5[0] --> [2]service_2
63
64  gc: WFT <ABC, US-E, ... >: App. Routing Table intialized in <service_5> <2ab20b69-16ea-4e0d-9fb9-77c55bf5896e>
65
66  gc: WFT <ABC, US-E, ... >: App. Routing Table intialized in <service_2> <cb15f190-9cff-495b-8622-2ea82edcfe52>
67
68  gc: WFT <ABC, US-E, ... >: App. Routing Table intialized in <service_4> <89820cb5-79a2-414b-b2f4-6cec4a7ce230>
69
70  gc: WFT <ABC, US-E, ... >: App. Routing Table intialized in <proxy_service_http> <026f1948-e8d5-477b-8831-04aff93e4057>
71
72  gc: WFT <ABC, US-E, ... >: App. Routing Table intialized in <service_3> <d9c2c88b-5455-40d0-8a9c-e8168d9b52ec>
73
74  gc: WFT <ABC, US-E, ...>: Sent activate WF message to proxy port
75
76  h1s2: number of active_wf_ports = 1
77
78  gc: WFT <ABC, US-E, ...>:
79  Workflow <5c23dc37-2636-48c7-aff5-948385d5c629><6d815b1d-831c-4b94-a6a0-373ed60c06a6> activated ......
80
81  h2s1: HTTP PRoxy: Add threads to the threadpool ===== 10
82
83  h2s1: number of active_wf_ports = 1
84
85  h2s1: HTTP Proxy Num threads alloted ================== 3
86
87  h2s1: Port= 0, Load = 1
88
89  h2s1: HTTP PRoxy: Add threads to the threadpool ===== 3
90
91  h2s1: Port= 0, Load = 2
92
93  h2s1: Port= 0, Load = 3
94  number of active_wf_ports = 1
95
96  h2s1: HTTP Proxy Num threads alloted ================== 3
97
98  h2s1: Port= 0, Load = 4
99
100 h2s1: Port= 0, Load = 5
101
102 h2s1: Port= 0, Load = 6
```

In line 2, the WFT gets to the job of starting the services for the workflow after the WFM has allocated it the required resources. After each service is initialized it connects to the AppFabric socket which opens a communication channel between the

service and the platform. Also, as shown in line 15, the AppFabric socket also starts a hearbeat reply service to reply to liveness queries from the platform. In line 34 the WFT attaches itself to one of the ports of the shared proxy service.

As already discussed in the previous sections, the AppFabric socket connects to and AppFabric port (pPort or a tPort). These ports are configured for:

- Setting up the ingress and egress links - example in line 38, 40, 42. (Note, the platform does not have an output for reporting the success of setting up the egress links)
- Setting up application routing policies in the ports - example in line 64, 66, 68.

After configuring the ports, the deployment of the workflow is complete. Lines 5–62 shows the setup service graph for the workflow. At this point the workflow is ready to serve user requests and a message to activate its mapping entry in the name server is sent and the workflow is activated in line 79. After this, the listing simply shows user session connecting to the workflow increasing the load on Port = 0 of the proxy service to which this workflow is attached. From this point on, the application instance (workflow) starts serving user requests and is automatically controlled by the runtime control poiicies.

- **Initial Bootstrapping- Multiple Zones** Listing. 6.14 shows the listing for an application deployment scenario with two zones - US-E and US-W. Note that the only difference from the listing for single zone deployment is that the global controller (or gc) launches a workflow managers (WFM) that launches two separate workflow threads; one for each zone. Therefore, the WFM is in control of all the application instances in all the zones but maintains separate data structures internally to distinguish between resource allocation and control policies in the different zones.

Listing 6.14: Bootstrapping messages for multi-zone deployment

```
1
2  ....
3  ....
4  gc: WFM: Initializing workflow manager
5  WFM: Starting workflow manager
6
7  WFT <ABC, US-E>: started
8  WFT <ABC, US-E>: connected to fake name server port
9  WFT <ABC, US-E, 25b58903-d802-41c4-9c0e-d4903cee62e5>: started
10 WFT <ABC, US-E, 25b58903-d802-41c4-9c0e-d4903cee62e5>: Requesting proxy Node
```

```
WFT <ABC, US-W>: started
WFT <ABC, US-W, de43b6b7-403d-4be3-a824-40236f54393f>: started
WFT <ABC, US-W, de43b6b7-403d-4be3-a824-40236f54393f>: Requesting proxy Node

....
....

WFT <ABC, US-E, 25b58903-d802-41c4-9c0e-d4903cee62e5>: Requesting proxy Node

gc: WFT <ABC, US-E, 25b58903-d802-41c4-9c0e-d4903cee62e5>: proxynode initialized, get resources for the rest

gc: WFT <ABC, US-E, 25b58903-d802-41c4-9c0e-d4903cee62e5>: Requesting resources

gc: WFT <ABC, US-E, 25b58903-d802-41c4-9c0e-d4903cee62e5>: Starting services

gc: WFT <ABC, US-W, de43b6b7-403d-4be3-a824-40236f54393f>: Requesting proxy Node

....
....
```

- **Application-level routing** We demonstrate application level routing through the following two outputs:
  - **Workflow comprising of message-level services only:** We created a simple service graph shown in Fig. 6.27.

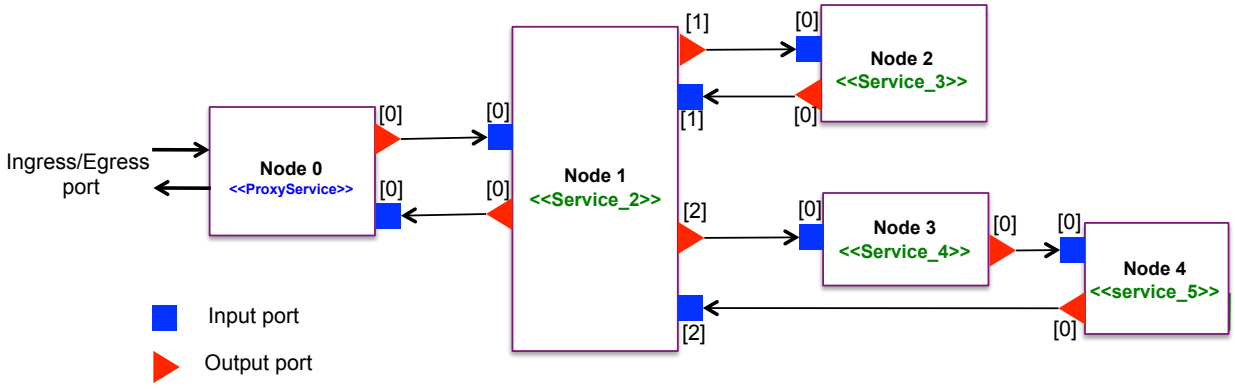

Figure 6.27: Service Graph to Demonstrate Application-level Routing

Listing. 6.15 shows the message received at the client demonstrating that the message from the same client for the same session may traverse two different paths (through different services) based content-based message classification.

Listing 6.15: Application-level routing output corresponding to service graph shown in Fig. 6.27

```
1
2  client: Sender:water.dhr.com , URL: /~card/guest.htm,
3  Response: Hello! I am service_2:cb15f190-9cff-495b-8622-2ea82edcfe52
4  Hello! I am service_4:89820cb5-79a2-414b-b2f4-6cec4a7ce230
5  Hello! I am service_5:2ab20b69-16ea-4e0d-9fb9-77c55bf5896e
6  Hello! I am service_2:cb15f190-9cff-495b-8622-2ea82edcfe52
7
8  client: Sender:water.dhr.com , URL: /~scottp/publish.html,
9  Response: Hello! I am service_2:cb15f190-9cff-495b-8622-2ea82edcfe52
10 Hello! I am service_3:d9c2c88b-5455-40d0-8a9c-e8168d9b52ec
11 Hello! I am service_2:cb15f190-9cff-495b-8622-2ea82edcfe52
```

– **Workflow comprising of both message-level services and packet level services:** Listing. 6.16 validates the application-level routing scenario in a mixed workflow constituting both, packet-level services as well as message-level services.

Listing 6.16: Application-level routing output showing a workflow consisting of message-level services interposed with packet-level services

```
1
2  client: Sender:sirius.develcon.com , URL: /~lowey/saskatoon/about_saskatoon/pictures.html,
3  Response: Hello! I am service_2:f471c087-11f4-4284-b242-fa34b60018d1
4  Hello! I am packet_service1 on host: e072cb1e-a217-4b87-9495-283350e658e0
5  Hello! I am service_3:5f4e6fe7-058d-47f5-b61f-c5e203689012
6  Hello! I am packet_service2 on host: df144359-a1e5-4cf5-a33d-a365e5ffe237
7  Hello! I am service_2:f471c087-11f4-4284-b242-fa34b60018d1
```

- **Dynamically creating new application instances to manage long user sessions**: Fig. 6.28 shows an experiment where we simulated long user sessions. Each user exchanges many messages within its session spanning a considerably long duration. The motive of this experiment was to create a condition where the application would be forced to rapidly replicate itself to serve new users. As shown in Fig. 6.28, the application starts with one application instance at time t = 0.The maximum capacity (in terms of number of user sessions) for each application session has been set to 15. The re-order level is set to 50 percent of the maximum capacity; that is around 8. The re-order level indicates when an application instance reports an overload condition. The re-order level should be set based on the delay of launching a new application instance such that the the new instance is activated before the existing instances get overwhelmed with user requests beyond their maximum capacity.

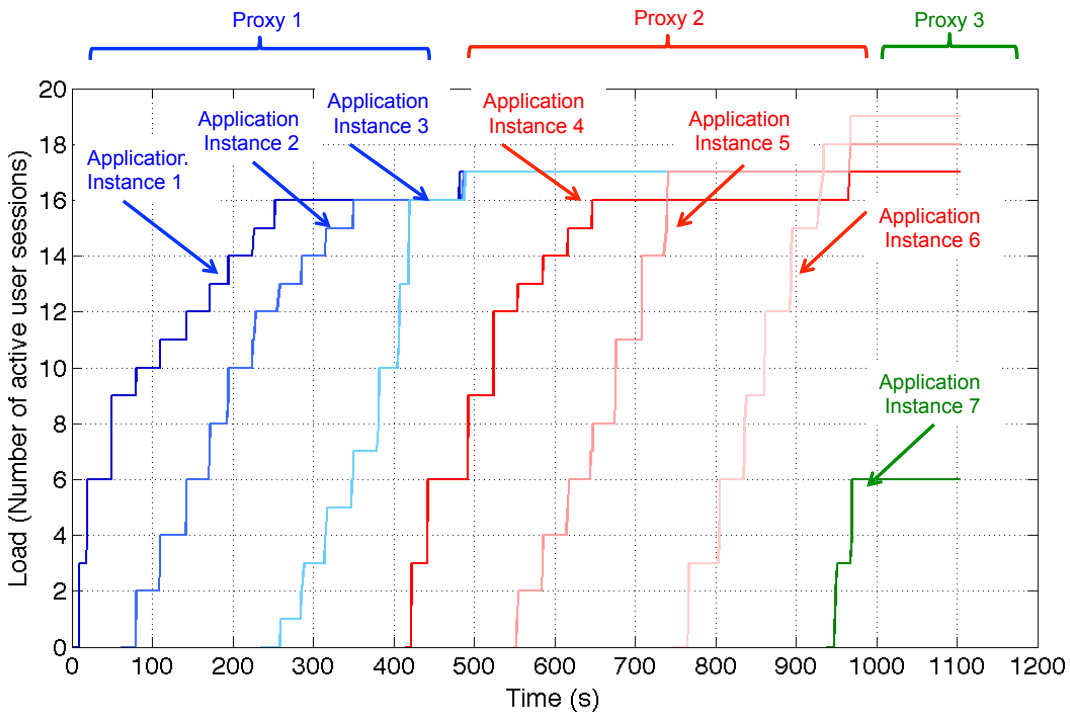

Figure 6.28: Dynamically Creating New Application Instances to Manage Long User Sessions

At t $\sim$ 50 secs; application instance 1 reports possible overload to which the controller reacts by spawning a new application instance (application instance 2) at t $\sim$ 90 secs. This process continues as more and more instances are launched. We already mentioned earlier in this chapter that the proxy service is a shared resource between multiple application instances. In this experiment, each proxy server is configured to be shared by at most three application instances. Fig. 6.28 also shows that the platform dynamically allocates new proxy servers along with spawning new application instances to accomodate the load. Another observable behavior evident from this experiment is that after reaching the maximum capacity, the application instance suspends itself - that is marks its mapping in the name server as suspended. However, even in the suspended state, it keeps serving new users if there are no other active instances in the system. Also, if there are more than one suspended instances, the additional load is balanced between all the suspended instances. This ensures that if for some reason the system has not been able to launch new instances in a timely fashion the application

does not become completely un-responsive. To prevent the application from serving new users completely, its mapping needs to be deleted from the name server.

- **Dynamically creating and destroying application instances for short user sessions**: This experimental setup is similar to the previous setup except that in this case the user sessions are short. This experiment is designed to show the dynamic behavior of the platform as the number of users in the system vary over time. Similar to the previous experiment, the maximum capacity (in terms of number of user sessions) for each application session has been set to 15. The re-order level is set to 50 percent of the maximum capacity; that is around 8. As shown in Fig. 6.29 :

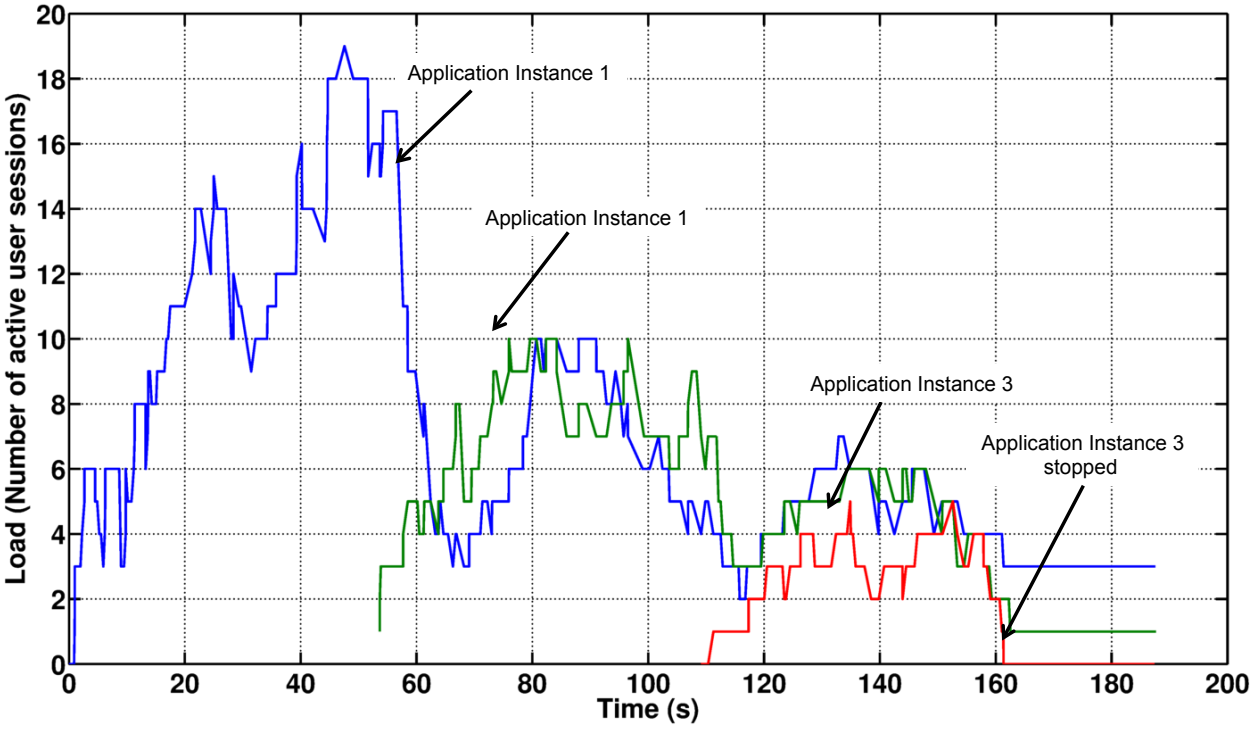

Figure 6.29: Dynamically Creating and Destroying Application Instances to Manage Short User Sessions

  - **At t = 0 sec:** The first application instance is started.
  - **At t $\sim$ 15 secs:** Load $>$ 8, application instance 1 reports of a a possible overload in the future. The controller reacts to this by automatically starting a second application instance. All lad is balanced between these two instances. Note that

the mechanism to predict a future overload has to take care of local perturbations (due to temporary bursts). there are several algorithms to do this. In our current implementation, we have implemented a specific weighted time wait mechanism that may be replaced by more robust predictors in the future.

- **At t $\sim$ 110 secs:** Similarly a third application instance is started.
- **At t $\sim$ 161 secs:** When the average load drops below a pre-specified level (under-load), the platform stops application instance 3. Application instance 3 stops accepting new user sessions and after it has served all the existing users connected to it, it may free all the resource allocated to it. We have implemented a chaining mechanism for spawning and destroying application instances wherein the most recent application instance is dropped during overload. We realize that this may not be always the case, and in future versions, the policy to be used to launch or remove application instance will be specified by the application administrator. Also note that the system works on presumptive estimates of overload/under-load conditions to account for the delays in dealing with them.

6.6.2 Performance Benchmarking

AppFabric makes heavy use of the ZeroMQ communication library and hence its performance is limited by the performance of ZeroMQ. ZeroMQ provides benchmarking tests for throughput and latency that provide a good estimate to the best possible performance that we may achieve in AppFabric. As mentioned earlier, the AppFabric code has not been tuned for performance. Therefore, the results reported in this section are only intended to give us some idea about the combined effect of the AppFabric platform overhead and the un-optimized code on the overall performance of the system. As of now, it is not possible for us to estimate the individual contributions of each of these factors, however, we can still provide some objective insights based on the observations. Each of these tests were performed on commodity server hardware with Intel Xeon 3.2 GHz processors (dual core), 4GB RAM, 16 K L1 cache, and 2048 K L2 cache.

- **Latency tests:** The first set of tests measure the latency in terms of the round-trip times (RTT) for processing each message. The setup for the tests shown in Fig. 6.30 is

very simple. A client sends a message to the application that comprises of a workflow of two message-level services (S1 and S2) chained together. We measure the RTT of the message traversing the service chain and returning to the client. We capture three scenarios :

- **Inproc:** In this scenario, S1 and S2 are started as two separate processes on the same host with the IPC as the underlying transport.
- **TCP-1 Host:** In this scenario, S1 and S2 are started as two separate processes on the same host with TCP as the underlying transport.
- **TCP-1 Host:** In this scenario, S1 and S2 are started as two separate processes on the two separate hosts with TCP as the underlying transport.

While in the first two scenarios the latency is mostly contributed by the limitation of the host configuration (CPU, RAM and Cache architecture), the latency in scenario three is mostly contributed by the underlying transport (TCP over 1Gbps Ethernet link).

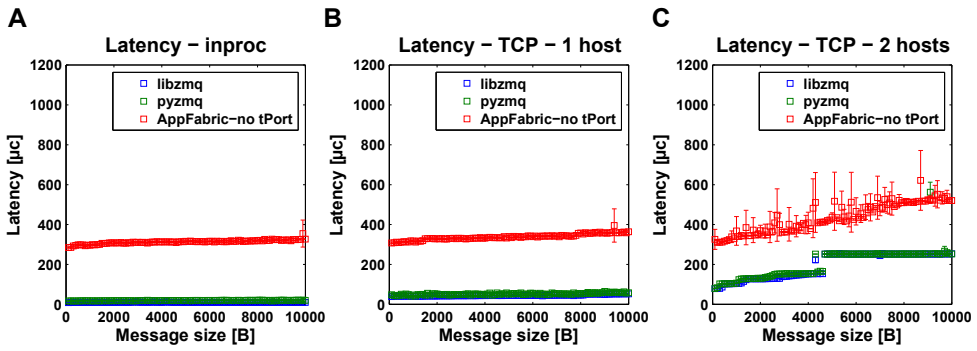

Figure 6.30: Latency Tests - Different Scenarios

In Fig. 6.30, each graph has three plots; one each for **libzmq**, **pyzmq**, and **AppFabrc**. The libzmq is the C library for ZeroMQ, the pyzmq provides the python language bindings for the libzmq APIs and AppFabric is the AppFabric code on top of pyzmq. Also, **no tPort** for the AppFabric represents that the application workflow comprises only of message-level services and does not have a packet-level port involved.

The results show that changing the message size from 100 Bytes to 10000 Bytes does not have a significant effect on any of the plots across all the three scenarios. This

is an expected result for latency measurements which tend to be a function of header processing and queueing overheads. This is exactly the reason why we see AppFabric adding an additional latency of 250 microseconds to each message over libzmq and pyzmq. AppFabric has significant application-level routing overhead where application-level headers are parsed and matched against classifiers to make a forwarding decision. Currently, the classifiers used in AppFabric are highly un-optimized implementations of regex-based classifiers, Future versions can use more optimized classifier implementations to get the AppFabric latency closer to the optimal (pyzmq latency).

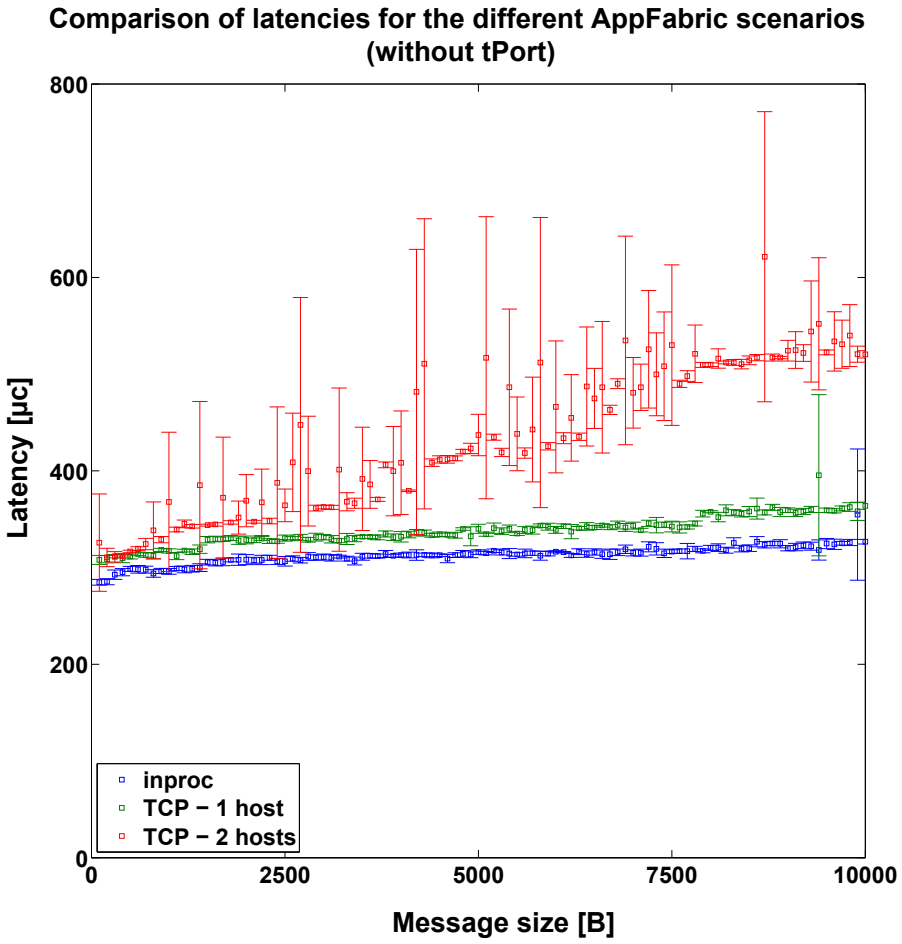

Figure 6.31: Latency Tests - Relative Contribution of the Different Transports Across the Three AppFabric Scenarios

Fig. 6.31 shows the relative contribution of the underlying transport across the three scenarios for AppFabric. As expected, the TCP 2-Host scenario has more latency than both the 1-Host scenario owing to the latency introduced by the network link. Also, increasing message sizes does not affect the 1-Host scenarios as much as the 2-Host scenario owing to the **transmission delays** being a function of the message size and the bit rate of the network link (in this case 1 Gbps Ethernet).

Next, we measured the latency for the scenario where the message has to go through packet-level services interposed between message-level services. We only compare the 2-Host case for these measurements. Also, for these measurements, the message was not actually passed through a packet-level service attached to a tPort so that we could realistically compare it with a 2 service sPort only scenario. The messages do hit the tPort twice (once for each host - tPort tunnel endpoints). Fig. 6.32 shows the comparison of the 2-Host scenario without tPort with the 2-Host scenario with tPort.

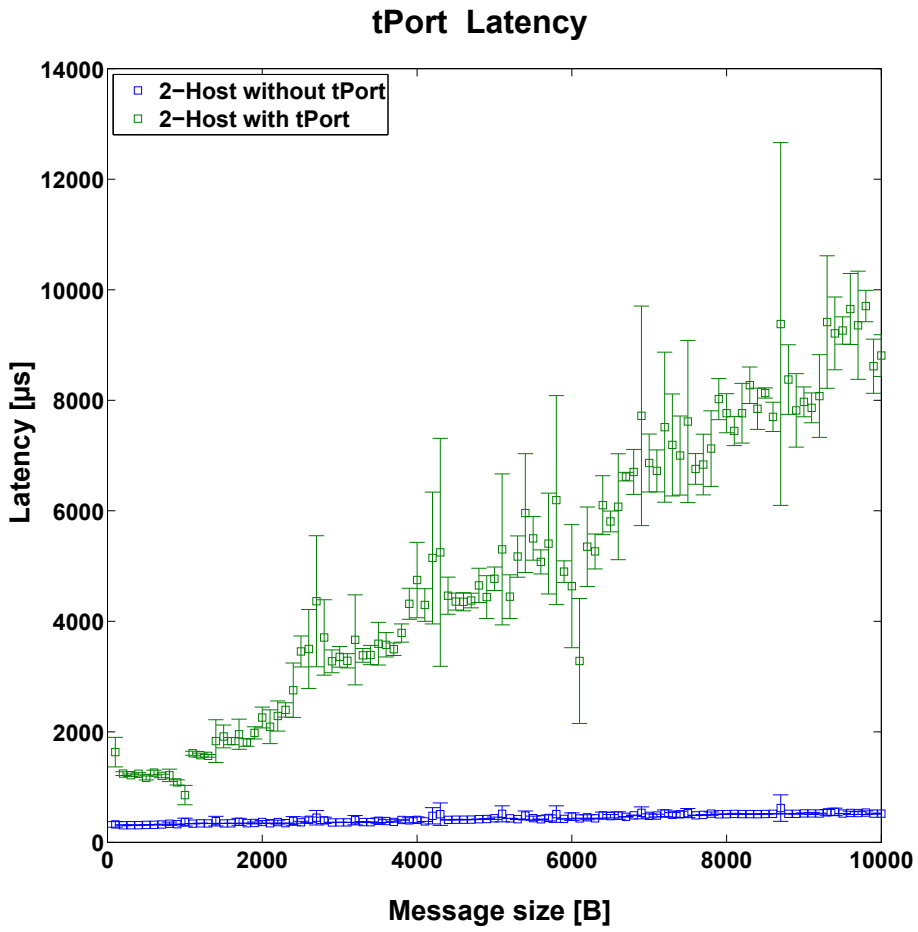

Figure 6.32: Latency Tests - Comparison of sPort vs tPort

As can be seen in Fig. 6.32. introducing the tPort has a severe negative impact on the latency. Again this is due to the highly un-optimized implementation of the tPort in the current code. In the current implementation, the packets have to cross the kernel-userspace boundary four times (twice for each tPort) which introduces significant overheads. Note that in Fig. 6.32 the latency for the tPort scenario increases with increasing message size, clearly indicating that copying the message multiple times is the reason for the lower performance. Implementing a zero-copy mechanism will significantly improve the latency for the tPort as indicated by the significantly smaller difference in the latency values for lower message sizes. Therefore, In the future we are looking towards a kernel implementation of the tPort with packet-level services

connecting to the kernel tPort switch using shared memory. This zero-copy implementation will improve the tPort performance significantly.

- **Throughput tests:** The second set of tests measure the throughput in terms of the Msg/sec that can be processed by the system. The setup for the tests shown in Fig. 6.33 is similar to the latency test setup. A client sends messages to the application that comprises of a workflow of two message-level services (S1 and S2) chained together. In this case, we measure the throughput at S2 in terms of the number of messages/sec that the system could sustain after allowing the system to reach equilibrium. Also, similar to the latency tests, we capture three scenarios :
  - **Inproc:** In this scenario, S1 and S2 are started as two separate processes on the same host with the IPC as the underlying transport.
  - **TCP-1 Host:** In this scenario, S1 and S2 are started as two separate processes on the same host with TCP as the underlying transport.
  - **TCP-1 Host:** In this scenario, S1 and S2 are started as two separate processes on the two separate hosts with TCP as the underlying transport.

Similar to the latency tests, the throughput tests also indicate that the AppFabric platform needs to be further optimized to match the benchmark results.

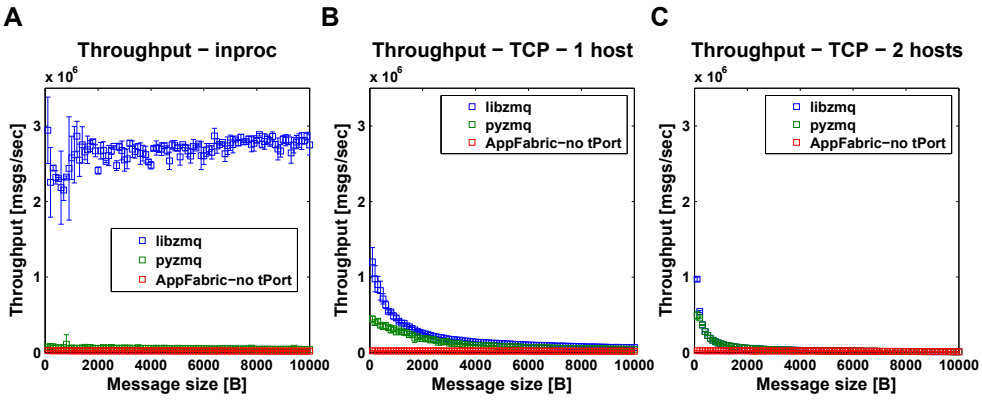

Figure 6.33: Throughput Tests - Different Scenarios

Again, similar to the latency tests, in Fig. 6.33, each graph has three plots; one each for **libzmq**, **pyzmq**, and **AppFabrc**. The libzmq is the C library for ZeroMQ, the

pyzmq provides the python language bindings for the libzmq APIs and AppFabric is the AppFabric code on top of pyzmq. Also, **no tPort** for the AppFabric represents that the application workflow comprises only of message-level services and does not have a packet-level port involved.

Some interesting observations from the graphs presented in Fig. 6.33 are as follows:

- In the first graph, the libzmq throughput is orders of magnitude greater than both pyzmq and AppFabric indicating that the python binding introduces significant overheads in interprocess communication as a result of which the AppFabric performance is also affected.
- For the TCP 1-Host scenario, the libzmq performance quickly deteriorates with increasing message size while the pyzmq and the AppFabric throughputs almost remain constant. This is because, while the performance of the TCP stack limits the performance of the libzmq(which quickly fills the available bandwidth), libzmq and AppFabric never really manage to fill the available bandwidth and hence their performance is not affected with increasing the message size to the extent of that of libzmq.
- The TCP 2-Host scenario is very similar to the TCP 1-Host scenario except that the limiting bandwidth is now 1Gbps.

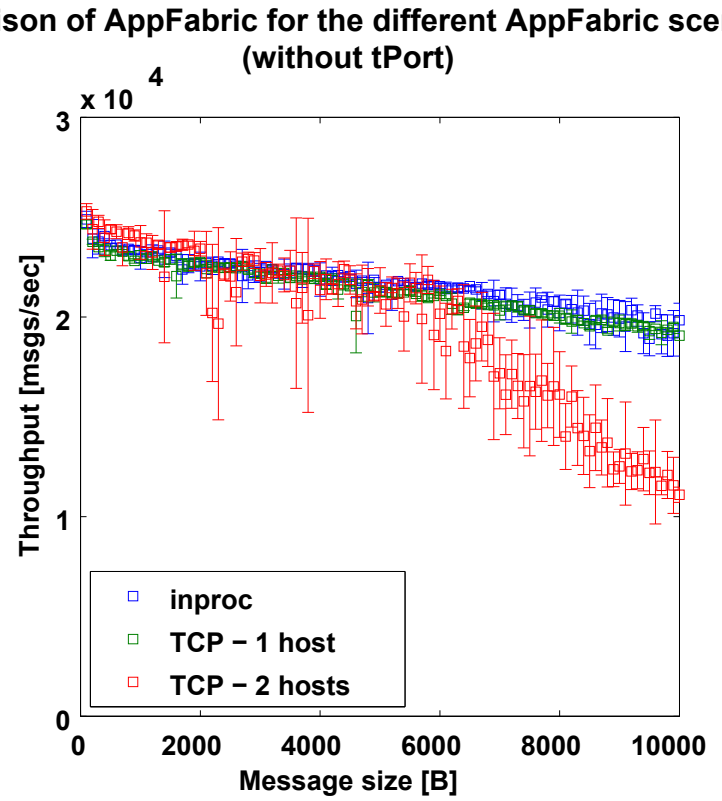

Figure 6.34: Throughput Tests - Relative Contribution of the Different Transports Across the Three AppFabric Scenarios

Fig. 6.34 shows the relative contribution of the underlying transport to the throughput across the three scenarios for AppFabric. As expected, the TCP 2-Host scenario has more latency than both the 1-Host scenario owing to the latency introduced by the network link. Also, increasing message sizes does not affect the 1-Host scenarios as much as the 2-Host scenario owing to the transmission delays being a function of the message size and the bit rate of the network link (in this case 1 Gbps Ethernet).

Next, we measured the throughput for the scenario where the message has to go through packet-level services interposed between message-level services. Again, similar to the latency measurements, we only compare the 2-Host case for these measurements. Also, for these measurements, the message was not actually passed through a packet-level service attached to a tPort so that we could realistically compare it with a 2 service

sPort only scenario. The messages do hit the tPort twice (once for each host - tPort tunnel endpoints). Fig. 6.35 shows the comparison of the 2-Host scenario without tPort with the 2-Host scenario with tPort.

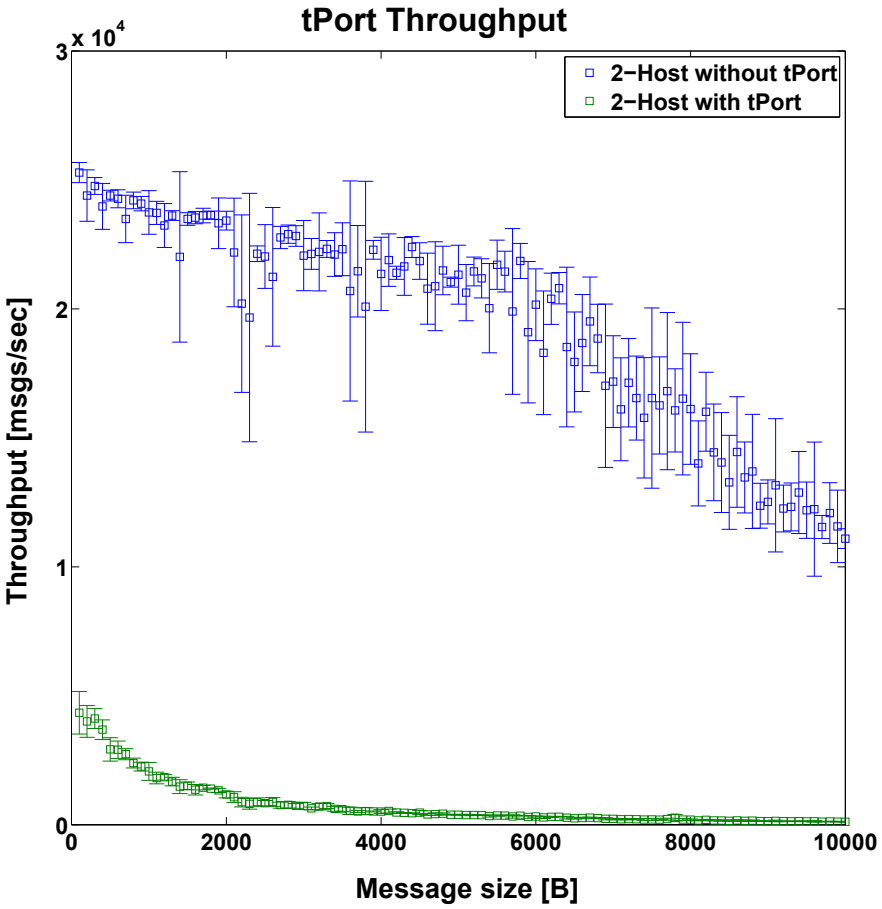

Figure 6.35: Throughput Tests - Comparison of sPort vs tPort

Again, similar to the latency tests, it can be seen from Fig. 6.35 that introducing the tPort has a severe negative impact on the throughput of the system. Similar to the latency tests, this is also expected to be because of the user-space implementation of the tPort causing messages to have to cross the kernel-userspace boundary four times (twice for each tPort) which introduces significant overheads. However, unlike the latency test results, this cause is not very evident from the graph in Fig. 6.35. This is

because for throughput of the system is limited by the width of the least throughput link in the system which is not identifiable from this graph. This will require running further micro-benchmarking tests on the throughput of the system with the tPort. These micro-benchmarking tests have not been conducted as part of the present work and will be undertaken in the future. But nonetheless, it may be safely assumed that zero-copy methods, which is expected to improve the latency significantly, will also play a major role in improving the throughput as well.

These benchmarking tests clearly suggest that there is a lot of room for optimizing the performance of the current AppFabric implementation and also provide some insights into some of the probable causes underlying these performance issues.

In this chapter, we discussed the AppFabric prototype implementation in much detail. As indicated several times, although the present implementation serves as a good proof-of-concept for validating the architectural design claims made in this thesis, there is a lot of scope for both design improvements and performance optimizations. These improvements may be expected to be part of the AppFabric platform as the architecture matures and evolves to serve real application use-cases.

Chapter 7

Summary and Future Work

In this thesis we presented the architecture, design and prototype implementation details of AppFabric. AppFabric is a platform for automatically deploying and delivering massively distributed and extremely dynamic applications over next generation software defined infrastructures. We believe that AppFabric makes very timely and extremely relevant contribution that will open up a vast space of new and exciting application use-cases that were not possible to be deployed before. Internet-of-Things, Cyber-Physical Systems, online gaming, virtual worlds, mobile apps are just a few application examples that will benefit tremendously from a platform such as AppFabric. We expect AppFabric to be as disruptive as operating systems were to the wide-scale adoption and usage of computers or the app model of open APIs was to the success of modern smart phones. Like these two, AppFabric is also a "platform" that provides generic abstractions making it much easier to leverage the dynamism and flexibility of modern software-defined infrastructures. Additionally, AppFabric allows applications to easily create and manage their application deployments over virtual resources leased from many different providers distributed across many geographical sites. In this sense, AppFabric enables a two-sided market with Application Service Providers (ASPs) on one side and Infrastructure providers such as Internet Service Providers (ISPs) and Cloud Service Providers (CSPs) on the other. This role of AppFabric as an economic platform increases its chances of its success as a disruptive, new technology.

This thesis does not represent the culmination of the ideas that led to the development of AppFabric but rather marks the beginning of a new discussion. The work to improve and build upon the initial framework proposed in this thesis will continue over the next few

years. The idea of AppFabric has been funded and appreciated both by government research funding agencies as well as the industry. There seems to be a lot of interest around the idea already and we are pretty certain that this interest will only grow over the next few years. We believe that the success of AppFabric will depend on the simplicity and clarity of its architectural foundations, the robustness of its design and implementation and the generality of its exposed interfaces. We will make sure that we meet each of these criteria as we move forward towards the possible adoption of AppFabric in the real-world.

The current architecture and design of AppFabric has been covered quite extensively in this thesis.

- In **chapter 1,** we tried to motivate the case for AppFabric by presenting the benefits of having a platform that can easily and automatically manage the deployment and delivery of massively distributed application use-cases over virtual infrastructure leased from many different providers distributed across many different geographical sites.
- In **chapter 2,** we provided an extensive background on current and past research efforts that has in some way, directly or indirectly, motivated the design of AppFabric. The diversity and the quantity of the research presented in this chapter indicates the usefulness of AppFabric in that it converges many different ideas and makes them all part of a single platform.
- In **chapter 3,** we provided the high-level ideas behind the overall architectural framework of AppFabric. There are two key ideas discussed in this chapter.
  - The basic control framework of AppFabric is designed to create an integrated application delivery network (ADN) comprising of all the components required to deliver modern distributed applications including local and wide area network routing, forwarding and QoS transport services, middlebox services (including both, packet and message level middleboxes) providing security, optimizing performance and improving the efficiency of application deployments and application-level services implementing application logic including web servers, application servers and storage servers; and place them under a single control structure that can

program and administer each of these different components together to optimize the application delivery process.

  - The basic management framework of AppFabric is designed to accommodate many different software defined infrastructure stacks and negotiate with many different resource providers to acquire virtual resources from many geographically distributed sites and present the abstraction of a resource pool where each resource is represented by some attributes such as cost or latency from another resource etc. In the application runtime, these attributes are used by the platform to make a decision on resource selection.

- In **chapter 4,** the data plane architecture of AppFabric called OpenADN or Open Application Delivery Network is presented. OpenADN is the distributed and programmable data plane of the converged application delivery network or ADN controlled by the AppFabric control plane. The key idea behind the design of OpenADN is nested tunneling and programmable tunnel endpoint ports.

- In **chapter 5,** the control and management plane architecture of AppFabric called Lighthouse is presented. Lighthouse has a hierarchical design that enables it to effectively control and manage a massively distributed data plane. Also, Lighthouse interfaces with the application architects and administrators on one side (northbound interface) and with the OpenADN data plane nodes on the other side (southbound interface). Through the northbound interface, the application architects and administrators are allowed to specify the application's deployment and delivery policies. These polices are compiled by Lighthouse and it uses them to program the data plane nodes through the southbound interface.

- In **chapter 6,** we discussed the details of the implementation of the AppFabric prototype implementation. The AppFabric prototype has been implemented in C and Python. The present version has around 10,000 lines of code. Several parts of the AppFabric architecture have not yet been implemented and they have been clearly mentioned and documented in this chapter to serve as a reference for the future development. Only those parts of the architecture has been implemented that seem to be challenging and novel and hence require validation.

As we already mentioned, this thesis marks the beginning of further research on AppFabric-like platform. Given its present appeal, its usefulness to drive future innovations and the clear economic motivations to its adoption, we are more than hopeful that it will be a successful endeavor and will contribute to significantly changing our lives in the future. The present work on AppFabric may be extended in several different directions, including:

- **Migrate the platform to run on real multi-cloud deployment environments:** Currently, the AppFabric development branch is tested only within emulated lab environments. The next step forward is certainly to run it on real multi-cloud deployment environments and deal with some of the issues of failures, latency, security, and interoperability which are difficult to emulate within a lab environment.

- **Run real application workloads on the platform:** Currently we run only experimental workloads to test and debug the platform services. In the future, we would need to run real application workloads on the platform. Many IoT use-cases present a massively distributed application deployment scenario that may be mapped to run over AppFabric. Another example is mapping distributed game deployment engine such as Colyseus[11] to run over AppFabric. Running real application workloads over AppFabric will help extend the architecture design of AppFabric to expose more relevant APIs and implement newer services.

- **Making the platform secure:** The current implementation of AppFabric does not have any notion of platform security built into it. While this may be acceptable for experimental systems, our vision is to have a much greater impact than just confining it to be just a prototype design. Therefore, the first step is to make the platform secure. This involves both, securing the code against vulnerabilities as well as securing the communication between the different distributed components in the platform. Also, the platform needs to provide secure APIs for services to connect and communicate with the platform.

- **Performance optimization:** As shown in our micro-benchmarking results in Chapter. 6, the current implementation of AppFabric is highly un-optimized for high-performance environments. Also, the platform introduces significant overheads that may also be optimized by more careful and stricter architectural design reviews.

- **Running on multiple kernel versions and exposing APIs with many different language bindings:** Currently the AppFabric code has been tested to run only on Linux kernel version 3.0 and needs to support newer (and some older) kernel versions moving forward. Also, currently the AppFabric socket library exposes only Python-based APIs and hence only Python-based services can connect to the platform services presently. Creating more language bindings for the socket APIs will allow more general use of the platform by removing this restriction.

- **GUI-based management interface:** GUI-based management interfaces are easier for system administrators to work with. Currently, AppFabric does not provide a GUI-based management interface. System administrators have to edit xml-based configuration scripts. Future releases of AppFabric should provide a GUI-based management console for system administrators to easily manage the system. Also, the GUI should allow the administrator to monitor the runtime state of the system by allowing him to create centralized views. Currently, the administrator has to manually process the distributed logs to get this information.

- **Setting up more efficient and relevant sensors across the distributed platform components:** Currently, AppFabric is setup with a minimal heartbeat sensor to monitor the liveness of the attached services. This is clearly not enough for a distributed and dynamic system of the size and complexity of AppFabric. Several other sensors monitoring the platform components for failures, performance issues, security attacks, etc. need to be in place.

The list presented above enumerates some of the tasks that are in the immediate roadmap of AppFabric development. Once we are able to create a minimal framework to add these extensions to the current design, the goal is to open source the project to create a community of developers who can then take the project forward. Ultimately, we hope that AppFabric will successfully evolve to be the platform for deploying and delivering next generation, massively distributed applications over multi-cloud environments.

References

[1] IEEE Std. 802.1Q-2005. Virtual bridged local area networks. *IEEE Standards*, 2005.

[2] IEEE Std. 802.1Q-2011. Media access control (mac) bridges and virtual bridged local area networks. *IEEE Standards*, 2011.

[3] IEEE Std. 802.1Qbg-2012. ieee standard for local and metropolitan area networks media access control(mac) bridges and virtual bridged local area networks amendment 21: Edge virtual bridging. `http://standards.ieee.org/getieee802/download/802.1Qbg-2012.pdf`, 2012. [Online; accessed 17-June-2014].

[4] Dan Aloni. Cooperative linux. In *Proceedings of the Linux Symposium*, volume 2, pages 23–31, 2004.

[5] AMD-V. Amd virtualization. `http://www.amd.com/en-us/solutions/servers/virtualization`. [Online; accessed 17-June-2014].

[6] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. *Resilient overlay networks*, volume 35. ACM, 2001.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[8] Array networks. `http://www.arraynetworks.com/`. [Online; accessed 17-June-2014].

[9] Jerry Ash. Path computation element (pce) communication protocol generic requirements. *rfc 4657*, 2006.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[11] Ashwin R Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A distributed architecture for online multiplayer games. In *NSDI*, volume 6, pages 12–12, 2006.

[12] Dhruba Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[13] Brocade. http://www.brocade.com/index.page. [Online; accessed 17-June-2014].

[14] Stephen F Bush. Active virtual network management protocol. In *Parallel and Distributed Simulation, 1999. Proceedings. Thirteenth Workshop on*, pages 182–192. IEEE, 1999.

[15] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 15–28. USENIX Association, 2005.

[16] Brian Carpenter and Scott Brim. Middleboxes: Taxonomy and issues. Technical report, RFC 3234, February, 2002.

[17] Jeffery Case, Mark Fedor, Martin Schoffstall, and C Davin. A simple network management protocol (snmp), 1989.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[19] David Chappell. *Enterprise service bus.* " O'Reilly Media, Inc.", 2004.

[20] Hung-Bing Chen and Calvin Y Liu. Network attached storage, March 11 2008. US Patent D563,994.

[21] David R Cheriton and Mark Gritter. Triad: A new next-generation internet architecture. 2000.

[22] Ludmila Cherkasova and Jangwon Lee. Fastreplica: Efficient large file distribution within content delivery networks. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[23] Susanta Nanda Tzicker Chiueh and Stony Brook. A survey on virtualization technologies. *RPE Report*, pages 1–42, 2005.

[24] Cisco. Cisco one platform kit(onepk). http://www.cisco.com/c/en/us/products/ios-nx-os-software/onepk.html. [Online; accessed 17-June-2014].

[25] Citrix systems. http://www.citrix.com/. [Online; accessed 17-June-2014].

[26] Citrix. Netscalar application delivery controller. [Online; accessed 17-June-2014].

[27] Citrix. xenmotion. http://knowcitrixx.wordpress.com/xenserver-6/xenmotion/. [Online; accessed 17-June-2014].

[28] Apache cloudstack. `http://cloudstack.apache.org/`. [Online; accessed 17-June-2014].

[29] Jon Crowcroft, Steven Hand, Richard Mortier, Timothy Roscoe, and Andrew Warfield. Qos's downfall: at the bottom, or not at all! In *Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS: What have we learned, why do we care?*, pages 109–114. ACM, 2003.

[30] Yi Cui, Baochun Li, and Klara Nahrstedt. ostream: asynchronous streaming multicast in application-layer overlay networks. *Selected Areas in Communications, IEEE Journal on*, 22(1):91–106, 2004.

[31] B Davie, J Gross, et al. A stateless transport tunneling protocol for network virtualization (stt)(version 4), 13-sep-2013.

[32] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[33] Jeff Dike. *User mode linux*, volume 2. Prentice Hall Englewood Cliffs, 2006.

[34] Avri Doria, Ram Gopal, Hormuzd Khosravi, Ligang Dong, Jamal Salim, and Weiming Wang. Forwarding and control element separation (forces) protocol specification. *rfc 5810*, 2010.

[35] Amazon ec2. `http://aws.amazon.com/ec2/`. [Online; accessed 17-June-2014].

[36] Rob Enns, Martin Bjorklund, and Juergen Schoenwaelder. Netconf configuration protocol. *rfc 6241*, 2011.

[37] ETSI. Nfv whitepaper. 2012. [Online; accessed 17-June-2014].

[38] Eucalyptus. `https://www.eucalyptus.com/`. [Online; accessed 17-June-2014].

[39] F5 networks. `https://f5.com/`. [Online; accessed 17-June-2014].

[40] Dino Farinacci, Darrel Lewis, David Meyer, and Vince Fuller. The locator/id separation protocol (lisp). 2013.

[41] Nsf future internet architecture project. `http://www.nets-fia.net/`. [Online; accessed 17-June-2014].

[42] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[43] Project FloodLight. Floodlight openflow controller. http://www.projectfloodlight.org/floodlight/. [Online; accessed 17-June-2014].

[44] Bryan Ford and Janardhan R Iyengar. Breaking up the transport logjam. In *HotNets*, pages 85–90, 2008.

[45] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291. ACM, 2011.

[46] Open Networking Foundation. Openflow switch specification version 1.3.2. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.2.pdf, 2013. [Online; accessed 17-June-2014].

[47] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[48] Garth A Gibson and Rodney Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

[49] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 15–28. ACM, 2007.

[50] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. The flattening internet topology: Natural evolution, unsightly barnacles or contrived collapse? In *Passive and Active Network Measurement*, pages 1–10. Springer, 2008.

[51] IETF Working Group. Content delivery networks interconnection (cdni). https://datatracker.ietf.org/wg/cdni/. [Online; accessed 17-June-2014].

[52] Object Management Group. Corba. [Online; accessed 17-June-2014].

[53] Saikat Guha and Paul Francis. An end-middle-end approach to connection establishment. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 193–204. ACM, 2007.

[54] Joel Halpern and J Hadi Salim. Forwarding and control element separation (forces) forwarding element model. Technical report, RFC 5812, March, 2010.

[55] IBM. Ibm websphere. [Online; accessed 17-June-2014].

[56] IETF. Application layer traffic optimization. http://datatracker.ietf.org/wg/alto/charter/. [Online; accessed 17-June-2014].

[57] IETF. Interface to the routing system working group. http://datatracker.ietf.org/wg/i2rs/charter/. [Online; accessed 17-June-2014].

[58] IETF. Network virtualization overlays working group. https://datatracker.ietf.org/wg/nvo3/. [Online; accessed 17-June-2014].

[59] IETF. Open shortest path first igp. http://datatracker.ietf.org/wg/ospf/charter/. [Online; accessed 17-June-2014].

[60] Intel-VT. Hardware-assisted virtualization technology. http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/hardware-assist-virtualization-technology.html. [Online; accessed 17-June-2014].

[61] IRTF. Routing research group. https://irtf.org/concluded/rrg. [Online; accessed 17-June-2014].

[62] IRTF. Routing research group wiki. http://trac.tools.ietf.org/group/irtf/trac/wiki/RoutingResearchGroup. [Online; accessed 17-June-2014].

[63] IRTF. Software defined networking working group. https://irtf.org/sdnrg. [Online; accessed 17-June-2014].

[64] Raj Jain. Internet 3.0: ten problems with current internet architecture and solutions for the next generation. In *Military Communications Conference, 2006. MILCOM 2006. IEEE*, pages 1–9. IEEE, 2006.

[65] Dilip A Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 51–62. ACM, 2008.

[66] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[67] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[68] Virtual ethernet bridging. http://www.ieee802.org/1/files/public/docs2008/new-dcb-ko-VEB-0708.pdf. [Online; accessed 17-June-2014].

[69] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A data-oriented (and beyond) network architecture. *ACM SIGCOMM Computer Communication Review*, 37(4):181–192, 2007.

[70] Linux KVM. Kvm live migration. http://www.pcisig.com/specifications/iov/single_root/. [Online; accessed 17-June-2014].

[71] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[72] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[73] Layer 7 technologies. `http://www.layer7tech.com/`. [Online; accessed 17-June-2014].

[74] J-L Le Roux. Path computation element communication protocol (pcecp) specific requirements for inter-area mpls and gmpls traffic engineering. *rfc 4927*, 2007.

[75] Paul J Leach, Michael Mealling, and Rich Salz. A universally unique identifier (uuid) urn namespace. 2005.

[76] T Li. Preliminary recommendation for a routing architecture. *draft-irtf-rrg-recommendation-02 (work in progress)*, 2009.

[77] Mallik Mahalingam, D Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. Vxlan: A framework for overlaying virtualized layer 2 networks over layer 3 networks. *draftmahalingam-dutt-dcops-vxlan-01. txt*, 2012.

[78] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[79] David Meyer, Lixia Zhang, Kevin Fall, et al. Report from the iab workshop on routing and addressing. Technical report, RFC 4984, 2007.

[80] Microsoft. Component object model (com). [Online; accessed 17-June-2014].

[81] Microsoft. Distributed component object model (dcom). [Online; accessed 17-June-2014].

[82] Mobility first. `http://mobilityfirst.winlab.rutgers.edu/`. [Online; accessed 17-June-2014].

[83] Robert Moskowitz, Pekka Nikander, Petri Jokela, and Thomas Henderson. Host identity protocol. *RFC5201, April*, 2008.

[84] Named data networking. `http://named-data.net/`. [Online; accessed 17-June-2014].

[85] Nebula. `http://nebula-fia.org/publications.html`. [Online; accessed 17-June-2014].

[86] F5 Networks. Big-ip. [Online; accessed 17-June-2014].

[87] Juniper Networks. Understanding edge virtual bridging for use with vepa technology. `http://www.juniper.net/techpubs/en_US/junos12.3/topics/concept/bridging-edge-virtual-bridging-understanding.html`. [Online; accessed 17-June-2014].

[88] Pekka Nikander, Jari Arkko, and Börje Ohlman. Host identity indirection infrastructure (hi3). In *Proc. Second Swedish National Computer Networking Workshop (SNCNW), Karlstad, Sweden*, 2004.

[89] Erik Nordmark and Marcelo Bagnulo. Shim6: Level 3 multihoming shim protocol for ipv6. Technical report, RFC 5533, June, 2009.

[90] Erik Nordström, David Shue, Prem Gopalan, Robert Kiefer, Matvey Arye, Steven Ko, Jennifer Rexford, and Michael J Freedman. Serval: An end-host stack for service-centric networking. In *NSDI*, pages 85–98, 2012.

[91] NSF. The nation science foundation. [Online; accessed 17-June-2014].

[92] NSF. Nsf-find. [Online; accessed 17-June-2014].

[93] Eiji Oki, Tomonori Takeda, JL Le Roux, and A Farrel. Framework for pce-based inter-layer mpls and gmpls traffic engineering. *rfc 5623*, 2009.

[94] Openstack. `https://www.openstack.org/`. [Online; accessed 17-June-2014].

[95] Oracle. Oracle weblogic. [Online; accessed 17-June-2014].

[96] Jianli Pan, Raj Jain, and Subharthi Paul. A novel incrementally-deployable multi-granularity multihoming framework for the future internet. In *Global Communications Conference (GLOBECOM), 2012 IEEE*, pages 2659–2664. IEEE, 2012.

[97] Jianli Pan, Raj Jain, Subharthi Paul, Mic Bowman, Xiaohu Xu, and Shanzhi Chen. Enhanced milsa architecture for naming, addressing, routing and security issues in the next generation internet. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–6. IEEE, 2009.

[98] Jianli Pan, Raj Jain, Subharthi Paul, and Chakchai So-In. Milsa: A new evolutionary architecture for scalability, mobility, and multihoming in the future internet. *Selected Areas in Communications, IEEE Journal on*, 28(8):1344–1362, 2010.

[99] Jianli Pan, Subharthi Paul, Raj Jain, and Mic Bowman. Milsa: a mobility and multihoming supporting identifier locator split architecture for naming in the next generation internet. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–6. IEEE, 2008.

[100] Jianli Pan, Subharthi Paul, Raj Jain, and Xiaohu Xu. Hybrid transition mechanism for milsa architecture for the next generation internet. In *GLOBECOM Workshops, 2009 IEEE*, pages 1–6. IEEE, 2009.

[101] Embrane Architecture White Paper. Powering virtual network services. [Online; accessed 17-June-2014].

[102] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[103] Subharthi Paul, Raj Jain, and Jianli Pan. An identifier/locator split architecture for exploring path diversity through site multi-homing-a hybrid host-network cooperative approach. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–5. IEEE, 2010.

[104] Subharthi Paul, Raj Jain, and Jianli Pan. Multi-tier diversified architecture for the next generation internet. In *Proceedings of Cloud Computing and virtualization Conference (CCV 2010), Singapore*, 2010.

[105] Subharthi Paul, Raj Jain, Jianli Pan, and Mic Bowman. A vision of the next generation internet: A policy oriented perspective. In *BCS Int. Acad. Conf.*, pages 1–14, 2008.

[106] Subharthi Paul, Jianli Pan, and Raj Jain. A future internet architecture based on deconflated identities. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–6. IEEE, 2010.

[107] Subharthi Paul, Jianli Pan, and Raj Jain. Architectures for the future networks and the next generation internet: A survey. *Computer Communications*, 34(1):2–42, 2011.

[108] Pci sig. `http://www.pcisig.com/home/`. [Online; accessed 17-June-2014].

[109] R Perlman, D Eastlake, D Dutt, S Gai, and A Ghanwani. Routing bridges (rbridges): Base protocol specification. *rfc 6325*, 2011.

[110] Radia Perlman. *Interconnections: bridges, routers, switches, and internetworking protocols*. Pearson Education India, 2000.

[111] Lucian Popa, Norbert Egi, Sylvia Ratnasamy, and Ion Stoica. Building extensible networks with rule-based forwarding. In *OSDI*, pages 379–392, 2010.

[112] A Linux Foundation Collaborative Project. Opendaylight. `http://www.opendaylight.org/`. [Online; accessed 17-June-2014].

[113] Mika Raento, Antti Oulasvirta, Renaud Petit, and Hannu Toivonen. Contextphone: A prototyping platform for context-aware mobile applications. *Pervasive Computing, IEEE*, 4(2):51–59, 2005.

[114] Ahmad Rahmati and Lin Zhong. Context-for-wireless: context-sensitive energy-efficient wireless data transfer. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 165–178. ACM, 2007.

[115] Redback networks - a subsidiary of ericsson. `http://www.ericsson.com/`. [Online; accessed 17-June-2014].

[116] Remote differential compression. `http://msdn.microsoft.com/en-us/library/aa372948(v=vs.85).aspx`. [Online; accessed 17-June-2014].

[117] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, Eve Schooler, et al. Sip: session initiation protocol. Technical report, RFC 3261, Internet Engineering Task Force, 2002.

[118] Peter Saint-Andre. Extensible messaging and presence protocol (xmpp): Core. *rfc 6120*, 2011.

[119] Peter Saint-Andre. Extensible messaging and presence protocol (xmpp): Instant messaging and presence. *rfc 6121*, 2011.

[120] Vyas Sekar, Sylvia Ratnasamy, Michael K Reiter, Norbert Egi, and Guangyu Shi. The middlebox manifesto: enabling innovation in middlebox deployment. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 21. ACM, 2011.

[121] Amazon Web Services. Elastic load balancing. [Online; accessed 17-June-2014].

[122] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[123] PCI SIG. Single root i/o virtualization. `http://www.linux-kvm.org/page/Migration`. [Online; accessed 17-June-2014].

[124] Chakchai So-In, Raj Jain, Subharthi Paul, and Jianli Pan. Virtual id: a technique for mobility, multi-homing, and location privacy in next generation wireless networks. In *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*, pages 1–5. IEEE, 2010.

[125] Chakchai So In, Raj Jain, Subharthi Paul, and Jianli Pan. Future wireless networks: key issues and a survey (id/locator split perspective). *International Journal of Communication Networks and Distributed Systems*, 8(1):24–52, 2012.

[126] WU Song and JIN Hai. A survey of storage virtualization. *Mini-micro Systems*, 24(4):728–732, 2003.

[127] M Sridharan, K Duda, I Ganga, A Greenberg, G Lin, M Pearson, P Thaler, C Tumuluri, N Venkataramiah, and Y Wang. Nvgre: Network virtualization using generic routing encapsulation. *IETF draft*, 2011.

[128] Randall R Stewart and Qiaobing Xie. *Stream control transmission protocol (SCTP): a reference guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[129] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 73–86. ACM, 2002.

[130] Jon Tate, Fabiano Lucchese, and Richard Moore. *Introduction to storage area networks*. IBM Corporation, International Technical Support Organization, 2005.

[131] Technavio. Global application delivery controllers market in datacenters 2009-2013. `http://www.technavio.com/content/global-application-delivery-controllers-market-datacenters-2009-2013`. [Online; accessed 17-June-2014].

[132] David L Tennenhouse, Jonathan M Smith, W David Sincoskie, David J Wetherall, and Gary J Minden. A survey of active network research. *Communications Magazine, IEEE*, 35(1):80–86, 1997.

[133] Joe Touch and Radia Perlman. Transparent interconnection of lots of links (trill): Problem and applicability statement. *rfc 5556*, 2009.

[134] Jonathan S Turner, Patrick Crowley, John DeHart, Amy Freestone, Brandon Heller, Fred Kuhns, Sailesh Kumar, John Lockwood, Jing Lu, Michael Wilson, et al. Supercharging planetlab: a high performance, multi-application, overlay network platform. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 85–96. ACM, 2007.

[135] VMware. Understanding full virtualization, paravirtualization and hardware-assist. `http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf`. [Online; accessed 17-June-2014].

[136] VMware. vmotion. `https://www.vmware.com/products/vsphere/features/vmotion.html`. [Online; accessed 17-June-2014].

[137] vmware esx. Vmware esx and vmware esxi. `http://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf`. [Online; accessed 17-June-2014].

[138] Enabling service chaining on cisco nexus 1000v series. `http://www.cisco.com/c/en/us/products/collateral/switches/nexus-1000v-switch-vmware-vsphere/white_paper_c11-716028.html`. [Online; accessed 17-June-2014].

[139] Michael Walfish, Jeremy Stribling, Maxwell N Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *OSDI*, volume 4, pages 15–15, 2004.

[140] Wilkipedia. `http://www.wikipedia.org/`. [Online; accessed 17-June-2014].

[141] xpressive internet architecture. `http://www.cs.cmu.edu/~./xia/`. [Online; accessed 17-June-2014].

[142] Lily Yang, Ram Dantu, T Anderson, and Ram Gopal. Forwarding and control element separation (forces) framework. Technical report, RFC 3746, April, 2004.

[143] Zeromq. `http://zeromq.org/`. [Online; accessed 17-June-2014].

[144] Ying Zhu, Baochun Li, and Jiang Guo. Multicast with network coding in application-layer overlay networks. *Selected Areas in Communications, IEEE Journal on*, 22(1):107–120, 2004.

Vita

This is just a sample of what to do in a vita

Subharthi Paul

Degrees	B.S. Magna Cum Laude, computer Science, May 1988 M.S. Computer Science, December 1990 D.Sc. (or Ph.D.) Some Department, May 2007
Professional Societies	Association for Computing Machines The Touring Society The Free Software Foundation
Publications	Student, I. D. (2005). LaTeX document class for Sever Institute, *The LaTeX J.* **10**(4): 323–336. Student, I. D. (2005). More LaTeX wisdom, *Another LaTeX J.* **42**(7): 100–101.

June 2014

Note: Use month and year in which your degree will be conferred.