Efficient Approaches for Homing Complex Network Services

by

Azzam Alsudais

B.Sc., King Saud University, 2011

M.Sc., University of Colorado Boulder, 2015

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2020

Committee Members:

Eric Keller, Chair

Shivakant Mishra

Sangtae Ha

Eric Rozner

Shankaranarayanan Puzhavakath Narayanan

Alsudais, Azzam (Ph.D., Computer Science)

Efficient Approaches for Homing Complex Network Services

Thesis directed by Professor Eric Keller

Network service providers (NSPs) offer a wide array of network services to their customers at a global scale. In recent years, NSPs have been migrating their infrastructures to a virtualized software-based one [8] enabled by the network functions virtualization (NFV) paradigm. One critical aspect in operating NFV-based network services is homing. Homing (or placement) of virtual network functions (VNFs) on cloud and network service provider (NSP) infrastructures is a crucial step in the orchestration of network services, involving complex interactions with the cloud, SDN and service controllers. Large NSPs process thousands of homing requests submitted by their customers on a daily basis. At such large scale, it is imperative that homing of network services is performed efficiently.

In this dissertation, and guided by our extensive discussions and collaboration with a Tier-1 NSP, we identify limitations and challenges across multiple layers of the homing stack. Starting at the bottom of the stack, we look at how it is extremely challenging to provision VNFs in a truly elastic manner – hindering the ability to efficiently manage them. At upper layers, we identify limitations with current approaches that are used for service and cloud controllers to aggregate data from end nodes. We analyze why such approaches are not efficient when deployed at large scale. Finally, we identify several dependency problems that result from deploying distributed instances of the homing service.

Accordingly, we design systems that efficiently address such challenges across the homing stack. Specifically, we design a novel stateless architecture for VNFs to provide true elasticity when deployed at NSP cloud sites – allowing VNFs to seamlessly scale and failover. In addition, we propose and design a peer-to-peer search service that offers real-time data retrieval at global scale in an efficient manner. We also design and evaluate a novel homing service that provides quality homing solutions while significantly reducing load on the service and cloud controllers. We extensively evaluate each of these solutions and demonstrate their efficiency in addressing the different challenges across the homing stack.

Dedication

To my amazing wife, Nora, and to my sweet kids, Fai and Mohammad. You have been the fuel that kept me going through this journey. I love you with all my heart.

Acknowledgements

I want to start by thanking my parents, Mohammed and Asma, and my parents in-law, Ibrahim and Maha, for their continuous prayers and support. Your encouragements kept me going.

I would like to also thank my amazing wife, Nora. I cannot express how much I appreciate all the love and support you have given me. You supported me in every way possible, and I am grateful that I got to do this with you on my side. Thank you.

Many thanks to my advisor, Eric Keller. I truly appreciate all the guidance, help, and support you provided me with throughout my journey. I have learned a lot from you for which I will ever be grateful. Working closely with you is what I am going to miss most about my PhD. Thank you.

I would like to thank my committee, Shivakant Mishra, Sangtae Ha, Eric Rozner, and Shankaranarayanan Puzhavakath Narayanan. You provided me with great constructive feedback and enlightening questions that helped me complete this dissertation.

I am fortunate to have collaborated with a great team. I would like to thank Bharath Balasubramanian, Zhe Huang, Shankar P. Narayanan, and Rick Schlichting for that amazing opportunity. I specifically would like to extend my gratitude to Shankar. You were like a mentor to me. Thank you.

I would also like to thank my lab mates, Mohammad Hashemi, Greg Cusack, Marcelo Abranches, Maziyar Nazari, Karl Olson, Sepideh Goodarzy, Dwight Browne, and my former lab-mate and friend Murad Kablan. You all helped my in one way or another. I really enjoyed our weekly meetings, and I am glad I got to work next to such smart people.

Finally, I want to thank King Saud University for offering me a full scholarship throughout my masters and doctoral studies.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Network service providers (NSPs) offer a wide range of services to their customers, including: software-defined WAN (SD-WAN), 5G network slicing, virtual private networks (VPN), and virtual customer premises equipment (vCPE). Typically, this is on a global scale – for example, a Tier-1 provider like AT&T and Verizon service over hundreds of thousands of corporate customers (and tens of millions of individual customers) across thousands of edge data centers and points of presences (PoPs) [9, 140]. In order to greatly reduce the time to provision and manage network services for customers, as well as decrease the overall operational expenditure (OpEx), these NSPs are moving to virtualized software-based appliances (known as network functions virtualization, or NFV, in industry) [41]. One of the critical steps in operating NFV-based network services is homing, which effectively determines where to place a collection of virtual network functions (VNFs) that make up a given customer's homing (service) request.

Homing, however, is a challenge to realize in practice in an efficient manner. Shown in Figure 1.1 is a high level illustration of the homing infrastructure. Homing requests come in to a homing service (step ❶), which then formulates a set of possible ways to home the request and uses the northbound API of a collection of cloud and service controllers (step ❷) to determine what practical options there are and how efficient they are. To fulfill those requests, those controllers use a southbound API (step ❸) to get information from micro-data centers, on the current state of the system. Then, after evaluating the collected information, the homing service compiles a solution for each submitted request, which is then, passed down to the NFV framework (step ❹) for provisioning. Four fundamental problems emerge from this architecture, which this dissertation seeks to address in a bottom-up manner.

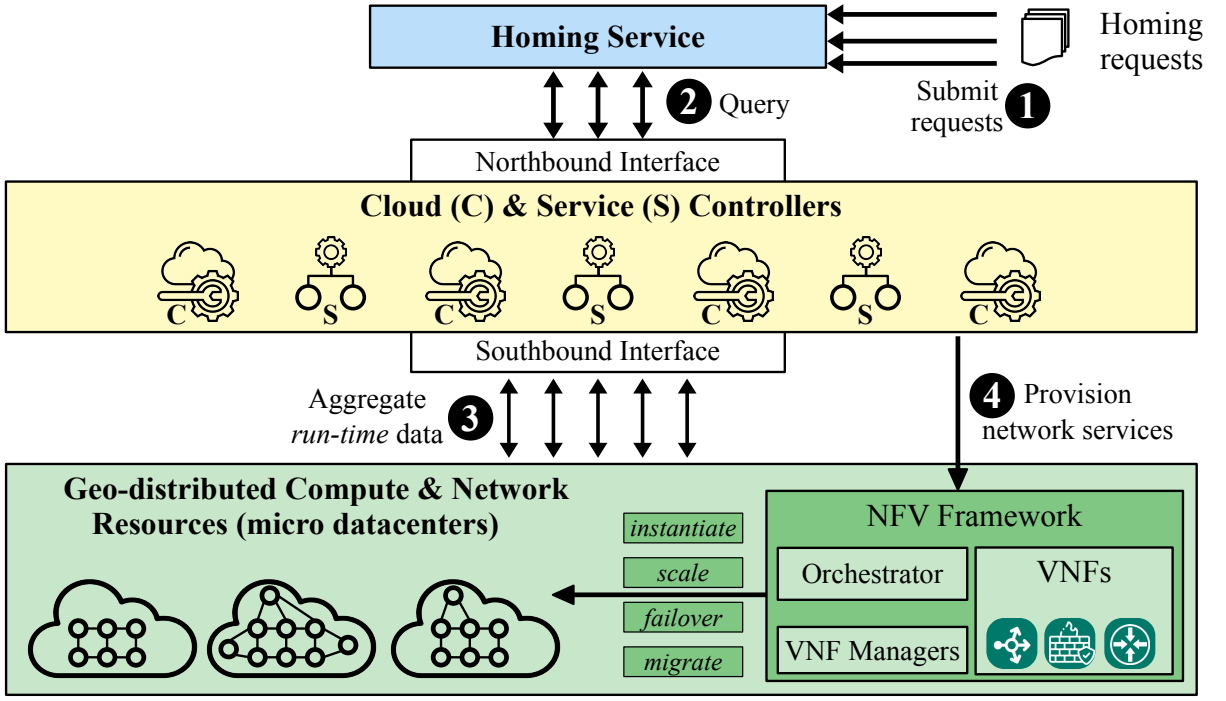


Figure 1.1: High-level view of the homing infrastructure showing numbered steps of the flow of processing a homing request at each layer of the stack: (1) homing requests get submitted by NSP customers, (2) the homing service will query the NSP controllers to fetch feasible resource candidates, and subsequently (3) the controllers will issue numerous queries to the end nodes to collect run-time information. (4) the homing service provides its recommendations (solution) for each homing request, which then are passed down to the NFV framework for provisioning.

(1) How can the NFV framework provision network services in an elastic and agile manner that utilizes resources and allows flexible management without interrupting service?

(2) How can controllers collect resource information in real-time given the geo-distributed nature of the NSP homing infrastructure?

(3) How can we gain information to make informed homing decisions without placing significant (querying) load on the controllers?

(4) How can we account for dependencies and handle resource contention and optimize resource sharing while we process simultaneous homing requests?

Each of these questions raises issues with each of the steps in Figure 1.1 in a bottom-up manner. In the remainder of this chapter, we provide a brief overview of the homing service (§1.1), and describe the

challenges and proposed solutions for each of the questions that this dissertation seeks to address (§1.2-1.5).

1.1 Overview of VNF Homing

In this section, we provide a brief overview of the homing service and describe how it works. We also describe the underlying infrastructure on top of which the homing service operates.

1.1.1 What is Homing?

Homing (or placement) of virtual network functions (VNFs) on cloud/network service provider infrastructures is an essential part of the orchestration of network services. NSPs offer homing as part of a suite of networking solutions to their customers, and a number of network automation and management frameworks [103, 107] provide the homing service in one shape or another – with many global NSPs driving the effort to develop and deploy such frameworks [104]. Due to the large geo-distributed scale of NSP infrastructures, this process consumes both time and resources by having to query numerous service and cloud controllers to find the best candidates to home VNFs while optimizing for the service request's objective(s).

The homing service, in a nutshell, provides recommendations for where to place VNFs, also known as network demands (firewall, gateway, loadbalancer, etc), submitted in the form of homing service requests by NSP customers – typically enterprises wishing to offload their networking functionality to the NSP. Each homing request consists of three main blocks.

- ***Demands*** (VNFs): a block that specifies the VNFs and the types of resources that are suitable to home those demands, such as: cloud resource (instantiate new instance of the VNF), or service resource (use slices of existing shared instances).

- ***Constraints***: a block that specifies how to evaluate resource candidates to determine their feasibility to home a given demand(s) – e.g., a firewall should be within close proximity to the customer's location. A service can have any number of constraints, and there can be some services that do not have any constraints.

- ***Objective functions***: a block that specifies the objectives for which the homing service should optimize (distance to customer, latency, resource utilization, provisioning cost, etc).

To summarize, the high level flow of a homing request is illustrated in Figure 1.1, and can be described as follows. Upon receiving a homing request (step ❶), the homing service will parse the request, which entails fetching a set of initial candidates for each of the demands in the request from the NSP inventory – mainly including high-level and static information about those candidates. The homing service, then, starts to evaluate the constraints for each of the demands. Constraint evaluation may require run-time information to be available, which in turn, triggers queries (step ❷) to be sent to the corresponding controllers – causing the controllers to fetch (step ❸) such run-time information directly from the end nodes (candidates). After evaluating the constraints, the homing service determines what candidate should *home* which demand guided by the objective function(s) of the request. Finally, the homing service provides its recommendations (solution) for each submitted homing request. Those recommendations are, then, passed down (step ❹) to the NFV framework for allocation and provisioning.

1.1.2 The Homing Infrastructure

NSP infrastructures typically consist of thousands of globally distributed edge micro datacenters and points of presence (PoPs) [79, 98], which can be operated by either the NSP or other third-party providers (e.g., Amazon EC2 [5], Microsoft Azure [64]) to expand the NSP footprint to reach more customers. Each of those edge datacenters consists of tens or hundreds of **compute** and **service** resources – which are treated as **candidates** to home network demands of a given customer's service request. Those resources are the ultimate source of information that the homing service needs to have in order to make informed decisions as to where to place VNFs[1] .

Cloud (compute) resources can be used to instantiate new VNF instances while service resources are managed and configured by the NSP to provide common functionality (e.g., cloud gateways) and can be shared and reused across multiple homing requests. In other words, a cloud resource can be selected to home

[1] Henceforth, we use the terms demand and VNF interchangeably.

a network demand (VNF) if that cloud resource has enough capacity (and meets all service constraints) to instantiate a new VNF instance. Likewise, a service resource can be selected to home a network demand if that service resource has enough capacity and is feasible to provide a slice to home the demand at hand.

The NSP deploys multiple controllers to use and manage those resources. Cloud controllers (C) are responsible for managing compute resources, including cloud and host level resources, covering NSP and third-party operated clouds. On the other hand, service controllers (S) manage existing network service instances, some of which can be shared across customers (*e.g.,* cloud-level gateway). These controllers offer application programming interfaces (APIs) to various network applications (including the homing service). The APIs can be consumed to configure or query the underlying resources managed by such controllers. The homing service consumes those northbound APIs to query controllers for the underlying resources to decide where to place each of the demands at hand.

When the demands of a given homing request are homed, the NFV framework (as well as the corresponding controllers) will monitor the status of live VNFs. Depending on the status of the VNF (as well as the host on which they run), some actions can be performed, including: scale in/out, failover, migrate, etc.

1.2 Provisioning Elastic Network Services

As we noted before, we follow a bottom-up approach to address and solve problems with each of the steps in Figure 1.1. In this section, we describe (§1.2.1) how network services are provisioned (step ❹), show why it is a challenge (§1.2.2) to provision truly elastic VNFs that can be easily managed (scaling, failover, etc), and describe our solution to address the challenges of efficiently implementing this step (§1.2.3).

1.2.1 Overview

Traditionally, network functions (firewall, router, loadbalancer, etc) had been deployed using hardware-based appliances that are highly optimized for each specific NF. This meant that when network operators wanted to roll out a new service, they had to manually plug in a hardware equipment and configure it in a way that satisfies service requirements. Nowadays, however, network service providers (NSPs) have moved

to software-based network services – driven by the NFV paradigm. That is, network functions are now virtualized (hence referred to as VNFs), and can run on commodity servers. This means that cloud-like technologies can be applied in such context to manage VNFs in a way that is more agile than before – bringing down the time to provision a new service from months to probably days or even hours. As such, NSPs have been leveraging NFV technologies to deploy large-scale and complex network services, and offer such services to customers who wish to offload their network processing from local offices to the NSP's large-scale network.

When the homing service provides its solution to a given homing request, the solution is then passed down to the NFV framework. This framework contains images of the VNFs that are available to provision network services. When the NFV framework receives a solution to a homing request, it will: (1) pull up the corresponding VNF images that make up the service, (2) deploy those VNFs (typically packaged in containers or VMs), and (3) configure the network such that designated traffic traverses the VNFs in an order that satisfies the service model. When the VNFs are deployed, the NFV framework enters a monitoring cycle where it monitors, detects, and reacts to any alarms (*e.g.,* scale out VNFs under high traffic load).

1.2.2 Challenge: seamless management of VNFs

After deploying VNFs, and when the NFV framework detects that a certain VNF needs to be scaled out, it instantiates a new instances of the VNF and redistributes traffic across the two instances. Doing so without incurring significant service disruption is extremely challenging. That is, due to the internal state that those VNFs keep track of (*e.g.,* connection setups, per-flow counts, etc), a newly instantiated instance of the VNF will not have access to such internal information, and thus could drop packets belonging to legitimate traffic. The same challenge also persists for other management primitives (failover, migrate, etc).

Unsurprisingly, this problem has gained much research interest in recent years. Recent works [124, 133] have proposed state checkpointing and input logging and replaying to statefully scale and failover. However, both approaches suffer from either high latency [124] or long recovery times [133]. We have concluded that a clean-slate solution is needed to deal with the problems at its root: the tight coupling of state and processing.

1.2.3 Stateless Network Functions: breaking the tight coupling of state and processing

To efficiently address the challenge of stateful and seamless management of VNFs, we take a clean slate approach in which we redesign network functions such that internal state is decoupled from the processing logic. To do this, we store the state on a remote replicated datastore that any VNF processing instance can access. We achieve high performance through optimized processing pipelines coupled with accelerated data-plane technology [67]. Our solution is capable of providing seamless failover and scaling for VNFs while minimizing overhead when compared to other solutions. Therefore, NSPs can leverage our design to provide such critical capabilities to uphold better service-level agreement (SLA) guarantees.

1.3 Collecting Real-time and Dynamic Resource Information at Large Scale

As we noted before, we follow a bottom-up approach in which we address and solve problems with each of the steps in Figure 1.1. In this section, we describe (§1.3.1) how controllers collect dynamic and run-time resource information (step ❸) – such information is needed to make informed homing decisions, and show why it is a challenge to implement this step in an efficient and scalable manner (§1.3.2). We, then, describe our proposed solution to address the challenges in implementing this step (§1.3.3).

1.3.1 Overview

Service and cloud controllers, as stated before, need to collect various resource (run-time) information to answer homing queries. Even a simple homing query (e.g., *"can a shared firewall instance X home the current demand?"*) will trigger numerous interactions with multiple controllers. For instance, when a service controller receives such query, it needs to: (1) query the firewall VNF manager (VNF-M) to check its capacity, (2) check with the cloud site hosting that firewall instance to determine whether there is enough bandwidth (at the site level) to serve the new demand, and (3) check with the NSP inventory to determine whether the customer requesting the demand is eligible to use it and whether the requested slice is within the customer's quota.

Mainly, there are two ways in which controllers can obtain and collect that information. First, re-

sources (or the end nodes) can frequently **push** their status updates to their corresponding controllers. Those controllers, then, keep the latest copy of that information in a local data store. Upon receiving homing queries, the controllers simply query their local data store to get the latest resource information. On the other hand, controllers can **pull** the information directly from the end nodes.

1.3.2 Challenge: scalability vs. data freshness

A push-based approach helps provide instantaneous responses to homing queries since responses are fetched from a local data store. However, this information can be stale and out-of-date – leading to incorrect behavior and resulting in errors in the homing process. One way to improve data *freshness* is to configure the update frequency at which the end nodes push their updates. However, this increases the workload the controllers need to handle, which as we show in Chapter 3, limits controller's scalability. Our own measurements of OpenStack (a widely used cloud management platform [111]) show that its scalability is limited due to the fact that nodes frequently communicate with the controllers (through a message queue like RabbitMQ [123]) to synchronize their state. Our observation is supported by findings performed by the OpenStack community [113]. Likewise, pull-based approaches are generally not considered scalable, as the controller needs to query many nodes simultaneously, and the synchronized responses coming back from the end nodes can result in server overload, or problems such as TCP incast [24].

1.3.3 FOCUS: scalable and efficient search service to process homing queries

The main challenge in implementing the data collection step in real time arises from substantial load being placed on controllers to process homing queries. To this end, we design FOCUS: a scalable search service that can be leveraged by NSPs to improve the step of processing queries between controllers and the end nodes and make it more efficient.

The main design intuition behind FOCUS is to offload the task of query processing to the end nodes. We achieve this by leveraging peer-to-peer (p2p) techniques to form the end nodes (resources) into groups based on their attribute values. Each group represents a range of attribute-value pair (e.g., nodes with available virtual CPUs between 10 and 20). This is coupled with having a group representative push meta-

data about group memberships to the corresponding controller, which then uses this group-level information to decide where to route homing queries. We show that FOCUS can significantly lower the load on the controllers by up to 90%, allowing them to use the freed-up resources to do more critical tasks – doing so while answering geo-distributed queries in a timely manner (within 1 second) .

1.4 Reducing Homing Query-load on Controllers

We now describe an overview of step ❷ of the homing process in Figure 1.1, where the homing service communicates with the controllers through their northbound API to query the underlying infrastructure (§1.4.1). We show why this step is challenging, given the large number of homing queries sent for each request and the global scale of an NSP infrastructure (§1.4.2), and propose a novel incremental approach to make the problem more tractable (§1.4.3).

1.4.1 Overview

The main job of the homing service is to provide recommendations for what resources to use for which requested demand (VNF) for each of the homing requests submitted to it by the NSP customers. To provide optimized homing recommendations, the homing service needs to have visibility over the underlying infrastructure to help make informed homing decisions by selecting the "best" possible resource candidates. This means that the homing service needs to query hundreds (if not thousands) of globally-distributed controller instances to get the most up-to-date information cloud and service resources. As mentioned before, for each of the demands of a homing request, there can be mainly two types of resource candidates: cloud (to instantiate a new instance of a VNF) and shared service (to reuse an existing instance of a VNF by requesting a "slice" of it). For a demand that requires cloud candidates, the homing service will need to query cloud controllers. However, for a demand that needs service candidates (e.g., a slice of shared firewall), the homing service will query both service and cloud controllers. The reason that cloud controllers are queried in this case is that the homing service considers them as potential candidates (to create a new instance of the requested shared VNF) in case it cannot find feasible service candidates.

After querying the corresponding controllers to get the information it needs, the homing service

evaluates the **constraints** of the homing request at hand. Those constraints tell the homing service what a feasible candidate is for each of the requested demands. After evaluating those constraints, the homing service will determine what resource candidate to select as the "best" candidate for each of the demands according to the logic of its optimization algorithm (greedy, random, exhaustive, etc).

1.4.2 Challenge: gaining visibility of the infrastructure requires intensive querying

The main challenge in providing enough visibility to the homing service (so that it can make informed decisions) is that we need to extensively query the underlying infrastructure to obtain up-to-date information. After analyzing logs of a homing service running in production of a large NSP, we observed that the homing process is query-intensive. Specifically, for a given homing request, querying the required data sources (cloud and service controllers) can take more than 800 seconds. When looking at the latency of individual queries by their data source type, we observed that a single query to either service or cloud controllers can take more than 1000 and 120 milliseconds, respectively. Viewed differently, this means that a single homing request can consume 800 seconds of controller-time. In such physically-constrained and limited-resource infrastructures [14, 21], one needs to carefully optimize the use of controller time and resources. As such, network and cloud operators often place a limit on the number of queries the homing service can make. Even though FOCUS reduces the southbound load on controllers, it would still be burdensome overhead, in the first place, to receive all those queries for each individual homing request at the northbound API.

1.4.3 *StepNet*: incremental approach to homing

Motivated by our observations, we seek to address this question: can we provide optimized homing recommendations without having to issue that many queries, and consequently, reduce the amount of controller resources the homing service needs to consume?

To this end, we design *StepNet*, a homing service that is built on top of the Homing and Allocation Service of the Open Network Automation Platform (ONAP) [103]. *StepNet* (described in detail in Chapter 4) adopts an incremental approach to homing where we incrementally explore the search space to evaluate potential candidates – the part that triggers queries to be sent to collect information needed to evaluate

constraints against a set of candidates to determine their feasibility.

The main intuition behind designing *StepNet* is that we limit the number of resource candidates that the homing service needs to evaluate, and correspondingly, limit the number of queries to be issued to the controllers – saving their time and processing power to perform more critical tasks (provision, configure, monitor, etc). The main challenge that arises from such design decision is how can we reduce the number of queries (i.e., visibility of the underlying infrastructure) without sacrificing quality of the homing recommendations. To achieve this, we leverage the objective function(s) of each homing request to help rank resource candidates. We, then, use those ranks to let the homing service evaluate only the top candidates. Recall that evaluating the feasibility of a candidates oftentimes requires issuing many queries to various controller instances. We show that *StepNet* can significantly reduce query cost by up to 92% for more than half of the 1200 requests we have evaluated while providing higher quality homing recommendations.

1.5 Accounting for Dependencies across Homing Requests

Now, we describe how dependencies between homing requests occur, and what problems and inefficiencies they present. We describe an overview of the problem (§1.5.1), show why it is a challenge to come up with a solution (§1.5.2), and then describe our solution that we designed to solve this problem (§1.5.3)

1.5.1 Overview

As stated before, homing is a lengthy process that is mainly dominated by query time. Given that NSPs serve a large number of customers and the time it takes to serve a single homing request is on the order several minutes, it becomes apparent that processing homing requests in a sequential manner means the homing service can quickly be overwhelmed with too many homing requests that are waiting to be served. To solve this, NSPs deploy multiple distributed instances of the homing service [101], each of which processes a subset of homing requests – a configuration that helps increase the overall throughput of the homing service.

1.5.2 Challenge: dependencies across homing requests cause problems

In our design of *StepNet*, we were solving for the base problem of reducing query-load on service and cloud controllers for each homing request, while assuming that homing requests are processed sequentially. Now, we relax that assumption and address the challenges introduced when we have distributed homing instances processing homing requests in parallel.

Specifically, there are three challenges that arise from having distributed instances of the homing service, all of which stem from dependencies and overlaps between multiple homing requests. First, multiple homing requests could issue redundant queries – placing unnecessary load on the controllers. Second, parallel homing instances could compete for the same underlying resources. This may lead to resource contention, and causes some homing requests to be re-solved – resulting in wasted resources and delayed service. Third, homing requests can ask for provisioning shared resources (*e.g.,* shared gateway, firewall, etc). Doing so in parallel leads to creating many unneeded shared resources as multiple homing instances may recommend creating multiple instances of shared resources at different locations – incurring extra provisioning cost.

1.5.3 *StepNet*+: accounting for dependencies between homing requests

To address dependencies between homing requests, we extend our original design of *StepNet*, with novel mechanisms that efficiently handle such dependencies. Specifically, *StepNet*+ adopts two main design decisions that allows it to account for dependencies: (1) centralized query caching – which helps reduce query redundancy through a centralized cache that is shared between distributed instances of the homing service, and (2) coordinated homing decisions – which decouples the decision making functionality from local homing instances, and instead coordinate such decisions through a centralized controller.

With these design decisions in place, we propose two novel techniques that help reduce resource contention and consolidate shared resources. First, we propose trace-driven mechanism that predicts whether a given resource (candidate) is likely able to accommodate more than one homing request. Second, we perform online consolidation of shared resources, where the controller consolidates shared resources at

solving time across multiple instances of the homing service. Further, we optimize our design with a novel multi-criteria batching that maximizes cache locality and minimizes the number of shared resources.

1.6 Outline and Contributions

In summary, we make the following contributions, which also show the outline for the rest of this dissertation.

- In chapter 2, we design and evaluate a novel architecture (StatelessNF) for virtual network functions (VNFs) in which they can be seamlessly managed (scaling, failover, migration, etc). We achieve this through decoupling the internal state of VNFs from the processing logic – making the VNF processing instances stateless. StatelessNF can seamlessly scale, failover, and migrate VNFs without incurring high performance overhead. By leveraging the StatelessNF architecture, NSPs can efficiently manage and better utilize their infrastructures by seamlessly instantiating and take down VNF instances without interrupting service.

- In chapter 3, we design a novel search service (FOCUS) for service and cloud controllers to find and aggregate data in real time. FOCUS leverages peer-to-peer techniques to manage resources and end nodes in an efficient manner that alleviate load on controllers managing those resources. FOCUS while aggregating data in a timely manner. FOCUS can easily integrate into existing systems through a well-defined API, and operates efficiently at large scale.

- In chapter 4, we propose a new homing service (*StepNet*) that adopts an incremental approach, which reduces query-load on the service and cloud controllers while providing high-quality homing solutions. We built an extensive trace-driven emulation framework that can emulate the behavior of controllers running in production networks of NSPs. Our trace-driven evaluation shows that *StepNet* reduces up to 92% of the query-load typically placed on the controllers to collect run-time information.

- In chapter 5, we design an extended homing service (*StepNet+*) that handles dependencies between

homing requests when there are distributed instances of the homing service. *StepNet+* can significantly reduce redundant queries and resource contention, and consolidate shared resources in an efficient manner across multiple homing requests. Our trace-driven evaluation demonstrates the benefits of our novel techniques that *StepNet+* adopts.

Chapter 2

Elastic and Resilient Homing Infrastructure with Stateless Network Functions

In this chapter, we present Stateless Network Functions, a new architecture for network functions virtualization, where we decouple the existing design of network functions into a stateless processing component along with a data store layer. In breaking the tight coupling, we enable a more elastic and resilient network function infrastructure. Our StatelessNF processing instances are architected around efficient pipelines utilizing DPDK for high performance network I/O, packaged as Docker containers for easy deployment, and a data store interface optimized based on the expected request patterns to efficiently access a RAMCloud-based data store. A network-wide orchestrator monitors the instances for load and failure, manages instances to scale and provide resilience, and leverages an OpenFlow-based network to direct traffic to instances. We implemented three example network functions (network address translator, firewall, and load balancer). Our evaluation shows (i) we are able to reach a throughput of 10Gbit/sec, with an added latency overhead of between 100μs and 500μs, (ii) we are able to have a failover which does not disrupt ongoing traffic, and (iii) when scaling out and scaling in we are able to match the ideal performance.

2.1 Introduction

As evidenced by their proliferation, middleboxes are an important component in today's network infrastructures [134]. Middleboxes provide network operators with an ability to deploy new network functionality as add-on components that can directly inspect, modify, and block or re-direct network traffic. This, in turn, can help increase the security and performance of the network.

While traditionally deployed as physical appliances, with Network Functions Virtualization (NFV),

network functions such as firewalls, intrusion detection systems, network address translators, and load balancers no longer have to run on proprietary hardware, but can run in software, on commodity servers, in a virtualized environment, with high throughput [61]. This shift away from physical appliances should bring several benefits including the ability to elastically scale the network functions on demand and quickly recover from failures.

However, as others have reported, achieving those properties is not that simple [48, 124, 125, 133]. The central issue revolves around the state locked into the network functions – state such as connection information in a stateful firewall, substring matches in an intrusion detection system, address mappings in a network address translator, or server mappings in a stateful load balancer. Locking that state into a single instance limits the elasticity, resilience, and ability to handle other challenges such as asymmetric/multi-path routing and software updates.

To overcome this, there have been two lines of research, each focusing on one property[1] . For failure, recent works have proposed either (i) checkpointing the network function state regularly such that upon failure, the network function could be reconstructed [124], or (ii) logging all inputs (*i*.e., packets) and using deterministic replay in order to rebuild the state upon failure [133]. These solutions offer resilience at the cost of either a substantial increase in per-packet latency (on the order of 10ms), or a large recovery time at failover (e.g., replaying all packets received since the last checkpoint), and neither solves the problem of elasticity. For elasticity, recent works have proposed modifying the network function software to enable the migration of state from one instance to another via an API [48, 73, 125]. State migration, however, takes time, inherently does not solve the problem of unplanned failures, and as a central property relies on affinity of flow to instance – each rendering state migration a useful primitive, but limited in practice.

In this chapter, we propose **stateless network functions** (or StatelessNF), a new architecture that breaks the tight coupling between the state that network functions need to maintain from the processing that network functions need to perform (illustrated in Figure 2.1). Doing so simplifies state management, and in turn addresses many of the challenges existing solutions face.

Resilience: With StatelessNF, we can instantaneously spawn a new instance upon failure, as the new

[1] A third line, sacrifices the benefits of maintaining state in order to obtain elasticity and resilience [39].

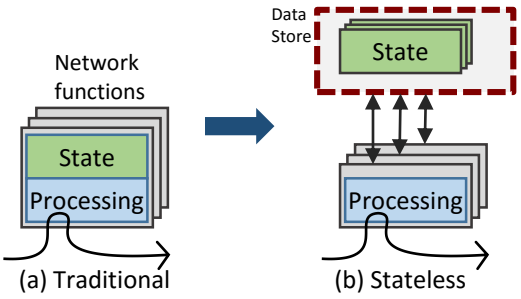


Figure 2.1: High level overview showing traditional network functions (a), where the state is coupled with the processing to form the network function, and stateless network functions (b), where the state is moved from the network function to a data store – the resulting network functions are now stateless.

instance will have access to all of the state needed. It can immediately handle traffic and it does not disrupt the network. Even more, because there is no penalty with failing over, we can failover much faster – in effect, we do not need to be certain a network function has failed, but instead only speculate that it has failed and later detect that we were wrong, or correct the problem (*e*.g., reboot).

Elasticity: When scaling out, with StatelessNF, a new network function instance can be launched and traffic immediately directed to it. The network function instance will have access to the state needed through the data store (*e*.g., a packet that is part of an already established connection that is directed to a new instance in a traditional, virtualized, firewall will be dropped because a lookup will fail, but with StatelessNF, the lookup will provide information about the established connection). Likewise, scaling in simply requires re-directing any traffic away from the instance to be shut down.

Asymmetric / Multi-path routing: In StatelessNF each instance will share all state, so correct operation is not reliant on affinity of traffic to instance. In fact, in our model, we assume any individual packet can be handled by any instance, resulting in an abstraction of a scalable, resilient, network function. As such, packets traversing different paths does not cause a problem.

While the decoupling of state from processing exists in other settings (*e*.g., a web server with a backend database), the setting of processing network traffic, potentially requiring per packet updates to state, poses a significant challenge. A few key insights and advances have allowed us to bring this new design to a reality. First, there have been recent advances in disaggregated architectures, bringing with it new, low-latency and resilient data stores such as RAMCloud [106]. Second, not all state that is used in

network functions needs to be stored in a resilient data store – only dynamic, network state needs to persist across failures and be available to all instances. State such as firewall rules, and intrusion detection system signatures can be replicated to each instance upon boot, as they are static state. Finally, network functions share a common pipeline design where there is typically a lookup operation when the packet is first being processed, and sometimes a write operation after the packet has been processed. This not only means there will be less interaction than one might initially assume, but also allows us to leverage this pattern to optimize the interactions between the data store and the network function instances to provide high performance.

We describe how four common network functions, can be re-designed in a stateless manner. We present the implementation of a stateful firewall, an intrusion prevention system, a network address translator, and a load balancer. Section 2.3 discusses the remote memory access. Section 2.6 discusses our utilization of RAMCloud for the data store, DPDK for the packet processing, and the optimized interface between the network functions and data store. Section 2.7 presents the evaluation: our experiments demonstrate that we are able to achieve throughput levels that are competitive with other software solutions [48, 125, 133] (4.6 Million packets per second for minimum sized packets), with only a modest penalty on per-packet latency (between 100us and 500us in the 95th percentile, depending on the application and traffic pattern). We further demonstrate the ability to seamlessly fail over, and scale out and scale in without any impact on the network traffic (as opposed to substantial disruption for a traditional design). Of course, the stateless network functions approach may not be suitable for **all** network functions, and there are further optimizations we can make to increase processing rates. This work, however, demonstrates that there is value for the functions we studied and that even with our current prototype, we are able to match processing rates of other systems with similar goals, while providing both scalability and resilience.

2.2 Motivation

Simply running virtual versions of the physical appliance counterparts provides operational cost efficiencies, but falls short in supporting the vision of a dynamic network infrastructure that elastically scales and is resilient to failure. Here, we illustrate the problems that the tight coupling of state and processing creates today, even with virtualized network functions, and discuss shortcomings of recent proposals.

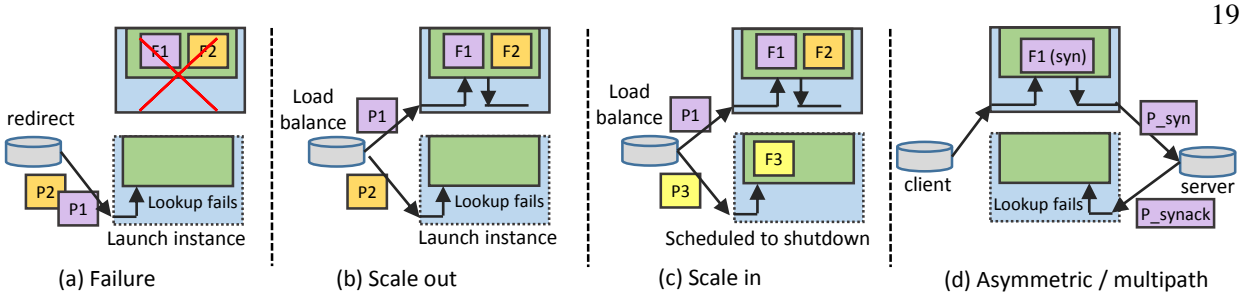


Figure 2.2: Motivational examples of traditional network functions and the problems that result from the tight coupling of state to the network function instance. State associated with some flow is labeled as F (*e*.g., F2), and the associated packets within that flow are labeled as P (*e*.g., P2).

First, we clarify our definition of the term "state" in this particular context. Although there is a variety of network functions, the state within them can be generally classified into (1) static state (*e*.g., firewall rules, IPS signature database), and (2) dynamic state, which is continuously updated by the network function's processes [46, 125]. The latter can be further classified into (i) internal instance specific state (*e*.g., file descriptors, temporary variables), and (ii) network state (*e*.g., connection tracking state, NAT private to public port mappings). It is the dynamic network state that we are referring to that must persist across failures and be available to instances upon scaling in or out. The static state can be replicated to each instance upon boot, so will be accessed locally.

2.2.1 Dealing with Failure

For failure, we specifically mean crash (as opposed to byzantine) failures. Studies have shown that failures can happen frequently, and be highly disruptive [121].

The disruption comes mainly from two factors. To illustrate the first factor, consider Figure 2.2(a). In this scenario, we have a middlebox, say a NAT, which stores the mapping for two flows (F1 and F2). Upon failure, virtualization technology enables the quick launch of a new instance, and software-defined networking (SDN) [19, 92, 108] allows traffic to be redirected to the new instance. However, any packet belonging to flows F1 or F2 will then result in a failed lookup (no entry in the table exists). The NAT would instead create new mappings, which would ultimately not match what the server expects. This causes all existing connections to eventually timeout. Enterprises could employ hot-standby redundancy, but that

doubles the cost of the network.

The second factor is due to the high cost of failover of existing solutions (further discussed below). As such, the mechanisms tend to be conservative when determining whether a device has failed [99] – if a device does not respond to one *hello* message, does that mean that the device is down, the network dropped a packet, or that the device is heavily loaded and taking longer to respond? Aggressive thresholds cause unnecessary failovers, resulting in downtime. Conservative thresholds may forward traffic to a device that has failed, resulting in disruption.

Problem with existing solutions

Two approaches to failure resilience have been proposed in the research community recently. First, pico replication [124] is a high availability framework that frequently checkpoints the state in a network function such that upon failure, a new instance can be launched and the state restored. To guarantee consistency, packets are only released once the state that they impact has been checkpointed – leading to substantial per-packet latencies (*e*.g., 10ms for a system that checkpoints 1000 times per second, under the optimal conditions).

To reduce latency, another work proposes logging all inputs (*i*.e., packets) coupled with a deterministic replay mechanism for failure recovery [133]. In this case, the per-packet latency is minimized (the time to log a single packet), but the recovery time is high (on the order of the time since last check point). In both cases, there is a substantial penalty – and neither deals with scalability or the asymmetric routing problem (discussed further in Section 2.2.3).

2.2.2 Scaling

As with the case of failover, the tight coupling of state and processing causes problems with scaling network functions. This is true even when the state is highly partitionable (*e*.g., only used for a single flow of traffic, such as connection tracking in a firewall). In Figure 2.2(b), we show an example of scaling out. Although a new instance has been launched to handle the overloaded condition, existing flows cannot be redirected to the new instance – *e*.g., if this is a NAT device, packets from flow F2 directed at the new instance will result in a failed lookup, as was the case with failure. Similarly, scaling in (decreasing

instances) is a problem, as illustrated in Figure 2.2(c). As the load is low, one would like to shut down the instance which is currently handling flow F3. However, one has to wait until that instance is completely drained (*i*.e., all of the flows it is handling complete). While possible, it is something that limits agility, requires special handling by the orchestration, and highly depends on flows being short lived.

Problem with existing solutions

The research community has proposed solutions based on state migration. The basic idea is to instrument the network functions with code that can export state from one instance and import that state into another instance. Router Grafting demonstrated this for routers (moving BGP state) [73], and several have since demonstrated this for middleboxes [47, 48, 125] where partitionable state can be migrated between instances. State migration, however, takes time, inherently does not solve the problem of unplanned failures, and as a central property relies on affinity of flow to instance (limiting agility).

2.2.3 Asymmetric / Multi-path Routing

Asymmetric and multi-path routing can cause further challenges for a dynamic network function infrastructure: Asymmetric and multi-path [116] routing relates to the fact that traffic in a given flow may traverse different paths, and therefore be processed by different instances. For example, in the scenario of Figure 2.2(d), where a firewall has established state from an internal client connecting to a server (SYN packet), if the return syn-ack goes through a different firewall instance, this packet may result in a failed lookup and get dropped.

Problem with existing solutions

Recent work proposes a new algorithm for intrusion detection that can work across instances [88], but does so by synchronizing processing (directly exchanging state and waiting on other instances to complete processing as needed). Other solutions proposed in industry strive to synchronize state across middleboxes [76] (*e*.g., HSRP [83]), but generally do not scale well.

2.3 How Network Functions Access State

The key idea in this chapter is to decouple the processing from the state in network functions – placing the state in a data store. We call this stateless network functions (or StatelessNF), as the network functions themselves become stateless, and the statefulness of the applications (*e*.g., a stateful firewall) is maintained by storing the state in a separate data store.

To understand the intuition as to why this is feasible, even at the rates network traffic needs to be processed, here we discuss examples of state that would be decoupled in common network functions, and what the access patterns are.

Table 2.1 shows the network state to be decoupled and stored in a remote storage for four network functions (TCP re-assembly is shown separate from IPS for clarity, but we would expect them to be integrated and reads/writes combined). As shown in the table, and discussed in Section 2.2, we only decouple network state.

We demonstrate how the decoupled state is accessed with pseudo-code of multiple network function algorithms, and summarize the needed reads and writes to the data store in Table 2.1. In all algorithms, we present updating or writing state to the data store as *writeRC* and reads as *readRC* (where RC relates to our chosen data store, RAMCloud). Below we describe Algorithms 1 (load balancer) and 2 (IPS). We also provide the pseudo-code for a stateful firewall (Algorithm 3), TCP re-assembly (Algorithm 4), and NAT (Algorithm 5) for reference.

For the load balancer, upon receiving a TCP connection request, the network function retrieves the list of backend servers from the remote storage (line 4), and then assigns a server to the new flow (line 5). The load for the backend servers is subsequently updated (line 6), and the revised list of backend servers is written into remote storage (line 7). The assigned server for the flow is also stored into remote storage (line 8), before the packet is forwarded to the selected server. For a data packet, the network function retrieves the assigned server for that flow, and forwards the packet to the server.

Algorithm 2 presents the pseudo-code for a signature-based intrusion prevention system (IPS), which monitors network traffic, and compares packets against a database of signatures from known malicious

Algorithm 1 Load Balancer

```
1:  procedure PROCESSPACKET(P: TCPPacket)
2:      extract 5-tuple from incoming packet
3:      if (P is a TCP SYN) then
4:          backendList ← readRC(Cluster ID)
5:          server ← nextServer(backendList, 5-tuple)
6:          updateLoad(backendList, server)
7:          writeRC(Cluster ID, backendList)
8:          writeRC(5-tuple, server)
9:          sendPacket(P, server)
10:     else
11:         server ← readRC(5-tuple)
12:         if (server is NULL) then
13:             dropPacket(P)
14:         else
15:             sendPacket(P, server)
```

Algorithm 2 IPS

```
1:  procedure PROCESSPACKET(P: TCPPacket)
2:      extract 5-tuple, and TCP sequence number from P
3:      if (P is a TCP SYN) then
4:          automataState ← initAutomataState()
5:          writeRC(5-tuple, automataState)
6:      else
7:          automataState ← readRC(5-tuple)
8:          while (b ← popNextByte(P.payload)) do
9:              // alert if found match
10:             // else, returns updated automata
11:             automataState ← process(b, automataState)
12:         writeRC(5-tuple, automataState)
13:         sendPacket(P)
```

Algorithm 3 Firewall

```
1:  procedure PROCESSPACKET(P: TCPPacket)
2:      key ← getDirectional5tuple(P, i)
3:      sessionState ← readRC(key)
4:      newState ← updateState(sessionState)
5:      if (stateChanged(newState, sessionState)) then
6:          writeRC(key, newState)
7:      if (rule-check-state(sessionState) == ALLOW) then
8:          sendPacket(P)
9:      else
10:         dropPacket(P)
```

Algorithm 4 TCP Re-assembly

```
1:  procedure PROCESSPACKET(P: TCPPacket)
2:      extract 5-tuple from incoming packet
3:      if (P is a TCP SYN) then
4:          record ← (getNextExpectedSeq(P), createEmptyBufferPointerList())
5:          writeRC(5-tuple, record)
6:          sendPacket(P)
7:      else
8:          record ← readRC(5-tuple)
9:          if (record == NULL) then
10:             dropPacket(P);
11:         if (isNextExpectedSeq(P)) then
12:             record.expected ← getNextExpectedSeq(P)
13:             sendPacket(P)
14:             // check if we can send any packet in buffer
15:             while (bufferHasNextExpectedSeq(record.buffPtr, record.expected)) do
16:                 P ← readRC(pop(record.buffPtr).pktBuffKey)
17:                 record.expected ← getNextExpectedSeq(p)
18:                 sendPacket(P)
19:             writeRC(5-tuple, record)
20:         else
21:             // buffer packet
22:             pktBuffKey ← getPacketHash(P.header)
23:             writeRC(pktBuffKey, P)
24:             record.buffPtr ← insert(record.buffPtr, p.seq, pktBuffKey)
25:             writeRC(5-tuple, record)
```

Algorithm 5 NAT

```
1:  procedure PROCESSPACKET(P: Packet)
2:      extract 5-tuple from incoming packet
3:      (IP, port) ← readRC(5-tuple)
4:      if ((IP, Port) is NULL) then
5:          list-IPs-Ports ← readRC(Cluster ID)
6:          (IP, Port) ← select-IP-Port(list-IPs-Ports, 5-tuple)
7:          update(list-IPs-Ports, (IP, Port))
8:          writeRC(Cluster ID, list-IPs-Ports)
9:          writeRC(5-tuple, (IP, Port))
10:         extract reverse-5-tuple from incoming packet plus new IP-port
11:         writeRC(reverse-5-tuple, (P.IP, P.Port))
12:     P' ← updatePacketHeader(P, (IP, Port))
13:     sendPacket(P')
```

Network Function	State	Key	Value	Access Pattern
Load Balancer	Pool of Backend Servers	Cluster ID	IP List	1 read/write at start/end of conn.
	Assigned Server	5-Tuple	IP Address	1 read/write at start/end of conn. 1 read for every other packet
Firewall	Flow	5-Tuple	TCP Flag	5 read/write at start/end of conn. 1 read for every other packet
NAT	Pool of IPs and Ports	Cluster ID	IP and Port List	1 read/write at start/end of conn.
	Mapping	5-Tuple	(IP, Port)	1 read/write at start/end of conn. 1 read for every other packet
TCP Re-assembly	Expected Seq Record	5-Tuple	(Next Expected Seq, Keys for Buffered Pkts)	1 read/write for every packet
	Buffered Packets	Buffer Pointer	Packet	1 read/write for every out-of-order packet
IPS	Automata State	5-Tuple	Int	1 write for first packet of flow, 1 read/write for every other packet

Table 2.1: Network Function Decoupled States

threats using an algorithm such as Aho-Corasick algorithm [2] (as used in Snort [129]). At a high-level, a single deterministic automaton can be computed offline from the set of signatures (stored as static state in each instance). As packets arrive, scanning each character in the stream of bytes triggers one state transition in the deterministic automaton, and reaching an output state indicates the presence of a signature.

The 5-Tuple of the flow forms the key, and the state (to be stored remotely) simply consists of the state in the deterministic automaton (e.g., an integer value representing the node reached so far in the deterministic automaton). Upon receiving a new flow, the automata state is initialized (line 4). For a data packet, the state in the deterministic automaton for that flow is retrieved from remote storage (line 7). The bytes from the payload are then scanned (line 8). In the absence of a malicious signature, the updated state is written into remote storage (line 12), and the packet forwarded (line 13). Out-of-order packets are often considered a problem for Intrusion Prevention Systems [146]. Similar to the Snort TCP reassembly preprocessor [129], we rely on a TCP re-assembly module to deliver the bytes to the IPS in the proper order.

For the load balancer, we observe that we require one read for each data packet, and at most one additional read and write to the remote storage at the start and end of each connection. For the IPS, we observe that we require one write to the remote storage to initialize the automata state at the start of each connection, and one read and one write to remote storage for each subsequent data packet of the connection. Table 2.1 shows similar patterns for other network functions, and Section 2.7 analyzes the performance

impact of such access patterns, and demonstrates that we can achieve multi Gbps rates.

2.4 Overall StatelessNF Architecture

At a high level, StatelessNF consists of a network-wide architecture where, for each network function application (*e*.g., a firewall), we effectively have the abstraction of a single network function that reliably provides the necessary throughput at any given time. To achieve this, as illustrated in Figure 2.3, the StatelessNF architecture consists of three main components – the data store, the hosts to host the instances of the network function, and an orchestration component to handle the dynamics of the network function infrastructure. The network function hosts are simply commodity servers. We discuss the internal architecture of network function instances in Section 2.5. In this section, we elaborate on the data store and network function orchestration within StatelessNF.

2.4.1 Resilient, Low-latency Data Store

A central idea in StatelessNF, as well as in other uses of remote data stores, is the concept of separation of concerns. That is, in separating the state and processing, each component can concentrate on a more specific functionality. In StatelessNF, a network function only needs to process network traffic, and does not need to worry about state replication, etc. A data store provides the resilience of state. Because of this separation, and because it resides on the critical path of packet processing, the data store must also provide low-latency access. For our purposes, we assume a data store that does not need support for transactions, but we anticipate exploring the impact of network functions that may require transactions as future work. In this chapter, we choose RAMCloud [106] as our data store. RAMCloud is a distributed key-value storage system that provides low-latency access to data, and supports a high degree of scalability.

Resilient: For a resilient network function infrastructure, the data store needs to reliably store the data with high availability.

This property is common in available data stores (key value stores) through replication. For an in-memory data store, such as RAMCloud [106], the cost of replication would be high (uses a lot of RAM). Because of this, RAMCloud only stores a single copy of each object in DRAM, with redundant copies on

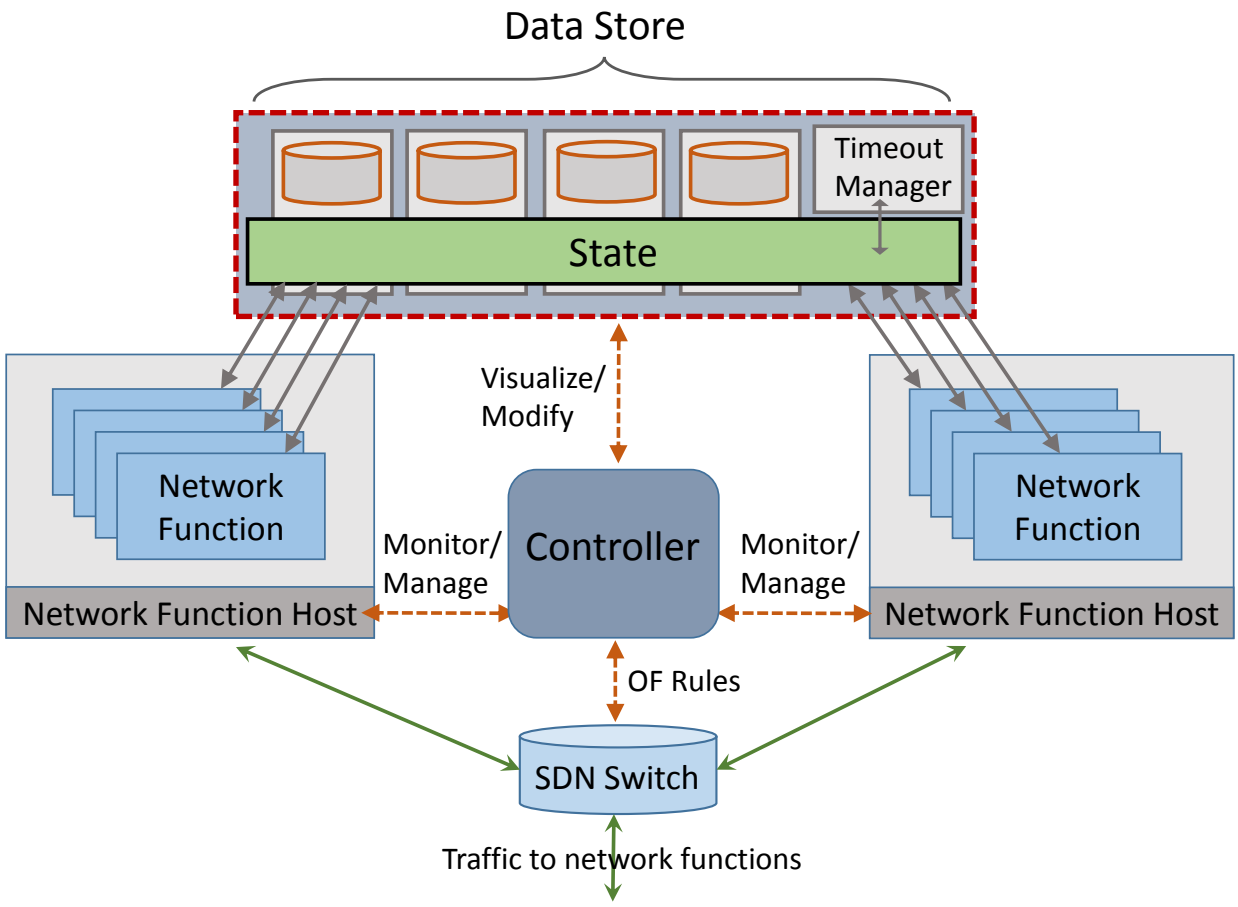


Figure 2.3: StatelessNF System Architecture

secondary storage such as disk (on replica machines). To overcome the performance cost of full replication, RAMCloud uses a log approach where write requests are logged, and the log entry is what is sent to replicas, where the replicas fill an in-memory buffer, and then store on disk. To recover from a RAMCloud server crash, its memory contents must be reconstructed by replaying the log file.

Low-Latency: Each data store will differ, but RAMCloud in particular was designed with low-latency access in mind. RAMCloud is based primarily in DRAM and provides low-latency access (6μs reads, 15μs durable writes for 100 bytes data) at large-scale (*e*.g., 10,000 servers). This is achieved both by leveraging low-latency networks (such as Infiniband and RDMA), being entirely in memory, and through optimized request handling. While Infiniband is not considered commodity, we believe it has growing acceptance (*e*.g., Microsoft Azure provides options which include Infiniband [65]), and our architecture

does not fundamentally rely on Infiniband – RAMCloud developers are working on other interfaces (*e*.g., RoCE [127] and Ethernet with DPDK), which we will integrate and evaluate as they become available.

Going beyond a key-value store: The focus of data stores is traditionally the key-value interface. That is, clients can read values by providing a key (which returns the value), or write values by providing both the key and value. We leverage this key-value interface for much of the state in network functions.

The challenge in StatelessNF is that a common type of state in network functions, namely timers, do not effectively conform to a key-value interface. To implement with a key-value interface, we would need to continuously poll the data store – an inefficient solution. Instead, we extend the data store interface to allow for the creation and update of timers. The timer alert notifies one, and only one, network function instance, for which the handler on that instance processes the timer expiration.

We believe there may be further opportunities to optimize StatelessNF through customization of the data store. While our focus in this chapter is more on the network-wide capabilities, and single instance design, as a future direction, we intend to further understand how a data store can be adapted to further suit the needs of network functions.

2.4.2 Network Function Orchestration

The basic needs for orchestration involve monitoring the network function instances for load and failure, and adjusting the number of instances accordingly.

Resource Monitoring and Failure Detection: A key property of orchestration is being able to maintain the abstraction of a single, reliable, network function which can handle infinite load, but under the hood maintain as efficient of an infrastructure as possible. This means that the StatelessNF orchestration must monitor resource usage as well as be able to detect failure, and adjust accordingly – *i*.e., launch or kill instances.

StatelessNF is not tied to a single solution, but instead we leverage existing monitoring solutions to monitor the health of network functions to detect failure as well as traffic and resource overload conditions. Each system hosting network functions can provide its own solution – *e*.g., Docker monitoring, VMWare

vcenter health status monitoring, IBM Systems Director for server and storage monitoring. Since we are using Docker containers as a method to deploy our network functions, our system consists of an interface that interacts with the Docker engines remotely to monitor, launch, and destroy the container-based network functions. In addition, our monitoring interface, through ssh calls, monitors the network function resources (cores, memory, and SR-IOV cards) to make sure they have enough capacity to launch and host network functions.

Important to note is that failure detection is different in StatelessNF than in traditional network function solutions. With StatelessNF, we have an effectively zero-cost to failing over – upon failure, any traffic that would go through the failed instance can be re-directed to any other instance. With this, we can significantly reduce the detection time, and speculatively failover. This is in contrast to traditional solutions that rely on timeouts to ensure the device is indeed failed.

Programmable Network: StatelessNF's orchestration relies on the ability to manage traffic. That is, when a new instance is launched, traffic should be directed to the instance; and when a failure occurs or when we are scaling-in, traffic should be redirected to a different instance. With emerging programmable networks, or software-defined networks (SDN), such as OpenFlow [92] and P4 [19], we can achieve this. Further, as existing SDN controllers (*e*.g., ONOS [13], Floodlight [144], OpenDaylight [108]) provide REST APIs, we can integrate the control into our overall orchestration.

2.5 StatelessNF Instance Architecture

Whereas the StatelessNF overall architecture provides the ability to manage a collection of instances, providing the elasticity and resilience benefits of StatelessNF, the architecture of the StatelessNF instances are architected to achieve the deployability and performance needed. As shown in Figure 2.4, the StatelessNF instance architecture consists of three main components – (i) a packet processing pipeline that can be deployed on demand, (ii) high-performance network I/O, and (iii) an efficient interface to the data store. In this section, we elaborate on each of these.

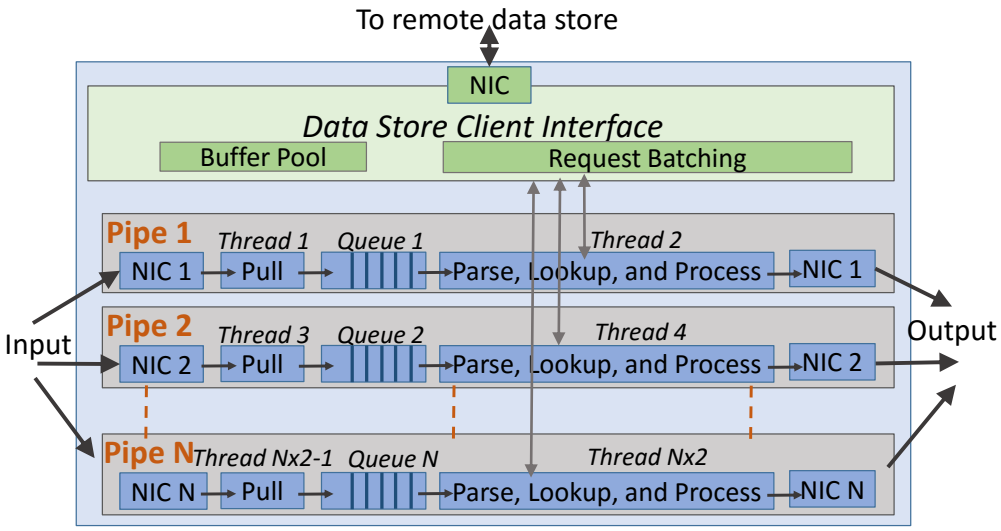


Figure 2.4: Stateless Network Function Architecture

2.5.1 Deployable Packet Processing Pipeline

To increase the performance and deployability of stateless network function instances, each network function is structured with a number of packet processing pipes. The number of pipes can be adaptive based on the traffic load, thus enabling a network function with a better resource utilization. Each pipe consists of two threads and a single lockless queue. The first thread is responsible for polling the network interface for packets and storing them in the queue. The second thread performs the main processing by dequeuing the packet, performing a lookup by calling the remote state interface to read, applying packet processing based on returned state and network function logic, updating state in the data store, and outputting the resulting packet(s) (if any).

Network function instances can be deployed and hosted with a variety of approaches – virtual machines, containers, or even as physical boxes. We focus on containers as our central deployable unit. This is due to their fast deployment, low performance overhead, and high reusability. Each network function instance is implemented as a single process Docker instance with independent cores and memory space/region. In doing so, we ensure that network functions do not affect each other.

For network connectivity, we need to share the physical interface among each of the containers (pipelines). For this, we use SR-IOV [66] to provide virtual interfaces to each network function instance.

Modern network cards have hardware support for classifying traffic and presenting to the system as multiple devices – each of the virtual devices can then be assigned to a network function instance. For example, our system uses Intel x520 server adapters [68] that can provide up to 126 virtual cards with each capable of reaching maximum traffic rate (individually). For connectivity to the data store, as our implementation focuses on RAMCloud, each network function host is equipped with a single Infiniband card that is built on the Mellanox RDMA library package [93], which allows the Infiniband NIC to be accessed directly from multiple network function user-space applications (bypassing the kernel). As new interfaces for RAMCloud are released, we can simply leverage them.

2.5.2 High-performance Network I/O

As with any software-based network processing application, we need high performance I/O in order to meet the packet processing rates that are expected. For this, we leverage the recent series of work to provide this – *e*.g., through zero copy techniques. We specifically structured our network functions on top of the Data Plane Development Kit (DPDK) architecture [67]. DPDK provides a simple, complete framework for fast packet processing.

One challenge that arises with the use of DPDK in the context of containers is that large page support is required for the memory pool allocation used for packet buffers and that multiple packet processing pipes (containers) may run simultaneously on a single server. In our case, each pipe is assigned a unique page filename and specified socket memory amount to ensure isolation[2] . We used the DPDK Environment Abstraction Layer (EAL) interface for system memory allocation/de-allocation and core affinity/assignment procedures among the network functions.

2.5.3 Optimized Data Store Client Interface

Perhaps the most important addition in StatelessNF is the data store client interface. The importance stems from the fact that it is through this interface, and out to a remote data store, that lookups in packet processing occur. That is, it sits in the critical path of processing a packet and is the main difference between

[2] After several tests, we settled on 2GB socket memory for best performance.

stateless network functions and traditional network functions.

Each data store will come with an API to read and write data. In the case of RAMCloud, for example, it is a key-value interface which performs requests via an RPC interface, and that leverages Infiniband (currently). RAMCloud also provides a client interface which abstracts away the Infiniband interfacing.

To optimize this interface to match the common structure of network processing, we make use of three common techniques:

Batching: In RAMCloud, a single read/write has low-latency, but each request has overhead. When packets are arriving at a high rate, we can aggregate multiple requests into a single request. For example, in RAMCloud, a single read takes 6μs, whereas a multi-read of 100 objects takes only 51μs (or, effectively 0.51μs per request). The balance here, for StatelessNF, is that if the the batch size is too small, we may be losing opportunity for efficiency gains, and too long (even with a timeout), we can induce higher latency than necessary waiting for more packets. Currently, we have a fixed batch size to match our experimental setup (100 objects), but we ultimately envision an adaptive scheme which increases or decreases the batch size based on the current traffic rates.

Pre-allocating a pool of buffers: When submitting requests to the data store, the client must allocate memory for the request (create a new RPC request). As this interface is in the critical path, we reduce the overhead for allocating memory by having the client reuse a preallocated pool of object buffers.

Eliminating a copy: When the data from a read request is returned from the data store to the client interface, that data needs to be passed to the packet processing pipeline. To increase the efficiency, we eliminate a copy of the data by providing a pointer to the buffer to the pipeline which issued the read request.

2.6 Implementation

The StatelessNF orchestration controller is implemented in Java with an admin API that realizes the implementation of elastic policies in order to determine when to create or destroy network functions. At present, the policies are trivial to handle the minimal needs of handling failure and elasticity, simply to allow us to demonstrate the feasibility of the StatelessNF concept (see Section 2.7 for elasticity and failure

experiments). The controller interacts with the Floodlight [144] SDN controller to steer the flows of traffic to the correct network function instances by inserting the appropriate OpenFlow rules. The controller keeps track of all the hosts and their resources, and network function instances deployed on top of them. Finally, the controller provides an interface to access and monitor the state in the data store, allowing the operator to have a global view of the network status.

We implemented three network functions (firewall, NAT, load balancer) as DPDK [67] applications, and packaged as a Docker container. For each, we implemented in a traditional (non-stateless) and stateless fashion. In each case, the only difference is that the non-stateless version will access its state locally while the stateless version from the remote data store. The client interface to the data store is implemented in C++ and carries retrieval operations to RAMCloud [106]. The packet processing pipes are implemented in C/C++ in a sequence of pipeline functions that packets travel through, and only requires developers to write the application-specific logic – thus, making modifying the code and adding new network function relatively simple. The data store client interface and the packet processing pipes are linked at compile time.

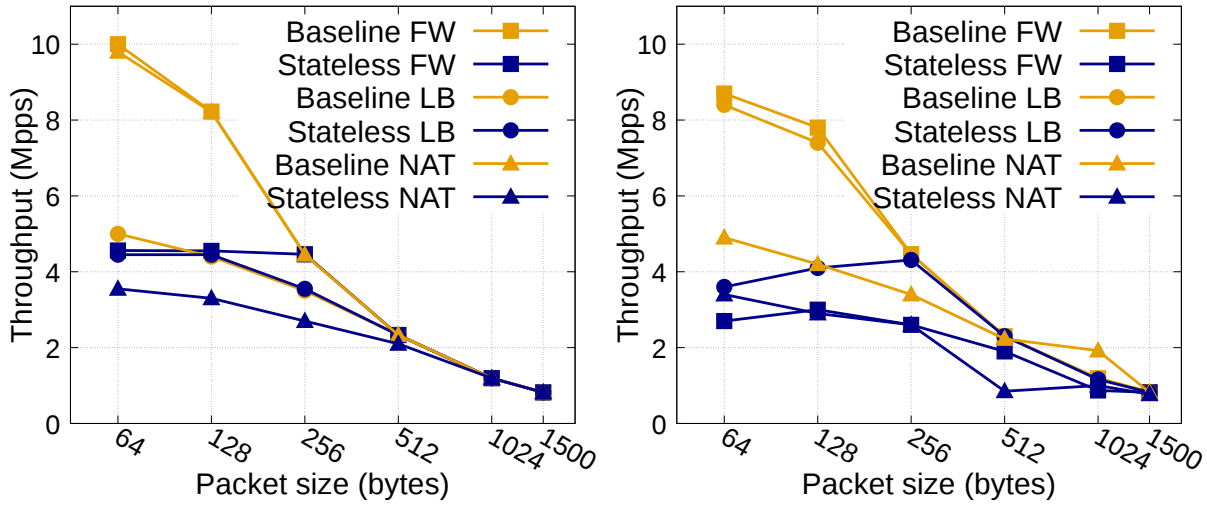
2.7 Evaluation

This section evaluates the network functions performance, the recovery times in failure events, and the performance impact when scaling in/out with the proposed stateless architecture.

2.7.1 Experimental Setup

Our experimental setup is similar to the one depicted in Figure 2.3. It consists of six servers and two switches. Two servers are dedicated to hosting the network function instances. These two servers are connected via Infiniband to two other servers hosting RAMCloud (one acting as the RAMCloud coordinator, and the other server storing state), and are connected via Ethernet to a server acting as the traffic generator and sink (not shown in Figure 2.3). The last server hosts the StatelessNF controller which orchestrates the entire management. Specifically, we use the following equipment:

- Network Function hosts: 2 Dell R630 Servers [31]: each has 32GB RAM, 12 cores (2.4GHz), one



(a) Long flow case

(b) Short flow case

Figure 2.5: Throughput of different packet sizes for long (a) and short (b) flows (*i*.e., flow sizes >1000 and <100, respectively) measured in the number of packets per second.

Intel 10G Server Adapter with SR-IOV support [68], and one 10G Mellanox InfiniBand Adapter Card [93].

- RAMCloud: 2 Dell R720 Servers [32], each with 48GB RAM, 12 cores (2.0GHz), one Intel 10G Server Adapter [68], one 10G Mellanox InfiniBand Adapter Card [93].
- Traffic generator/sink: 1 Dell R520 Servers [30]: 4GB RAM, 4 cores (2.0GHz), 2 Intel 10G Server Adapters [68] .
- Control: 1 Dell R520 Servers [30]: 4GB RAM, 4 cores (2.0GHz) to run StatelessNF and Floodlight controllers.
- SDN Switch: OpenFlow-enabled 10GbE Edge-Core [38].
- Infiniband Switch: 10Gbit Mellanox Infiniband switch between RAMCloud nodes and the network function hosts [94].

2.7.2 StatelessNF Performance

2.7.2.1 Impact of needing remote reads/writes

It is first critical to understand the performance of the RAMCloud servers as they may be a performance bottleneck, and limit the rates we can attain. Our benchmark tests reveal that a single server in RAMCloud can handle up to 4.7 Million lookup/sec. For write operations, a single server can handle up to 0.7 Million write/second.

The performance of a network function therefore heavily depends on the packets sizes, the network function's access patterns to the remote storage, and the processed traffic characteristics: For example, while a load balancer requires three write operations per flow, a firewall requires five write operations per flow. As such, whether traffic consists of short flows (e.g., consisting of only hundreds of packets), or long flows (e.g., comprising tens of thousands of packets), these differences in access patterns can have a significant impact on the network function performance. In particular, short flows require many more writes for the same amount of traffic. We consequently distinguish three cases for the processed traffic: long, short, and average in regards to the size and number of flows. The long case consists of a trace of 3,000 large TCP flows of 10K packets each. The short case consists of a trace of 100,000 TCP flows of 100 packets each. Finally for the average case, we replayed a real captured enterprise trace [26] with 17,000 flows that range in size from 10 to 1,000 packets. In each case, we also varied the packet sizes to understand their impact on the performance. We used Tcpreplay with Netmap [136] to stream the three types of traces.

Figure 2.5 shows the throughput of StatelessNF middleboxes compared to their non-stateless counterparts (which we refer to as baseline) with long and short flows of different packet sizes. For minimum sized packets, we obtain throughputs of 4.6Mpps. For small sized packets (less than 128 bytes), the gap between stateless and non-stateless in throughput is due to a single RAMCloud server being able to handle around 4.7 Million lookups/sec. In contrast, in the baseline, all read and write operations are local. We highlight that such sizes are used to test the upper bound limits of our system.

As packets get larger in size (greater than 128 bytes), the rates of stateless and baseline network functions converge. The obtained throughputs are competitive with those of existing elastic and fail resilience

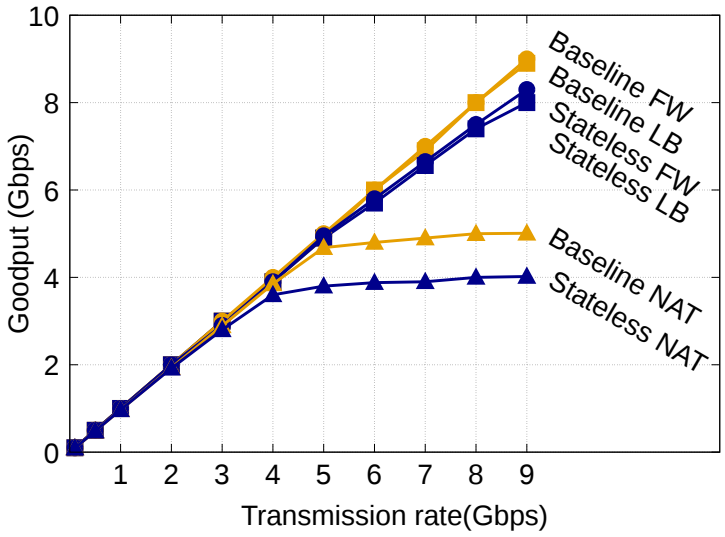


Figure 2.6: Measured goodput (Gbps) for enterprise traces.

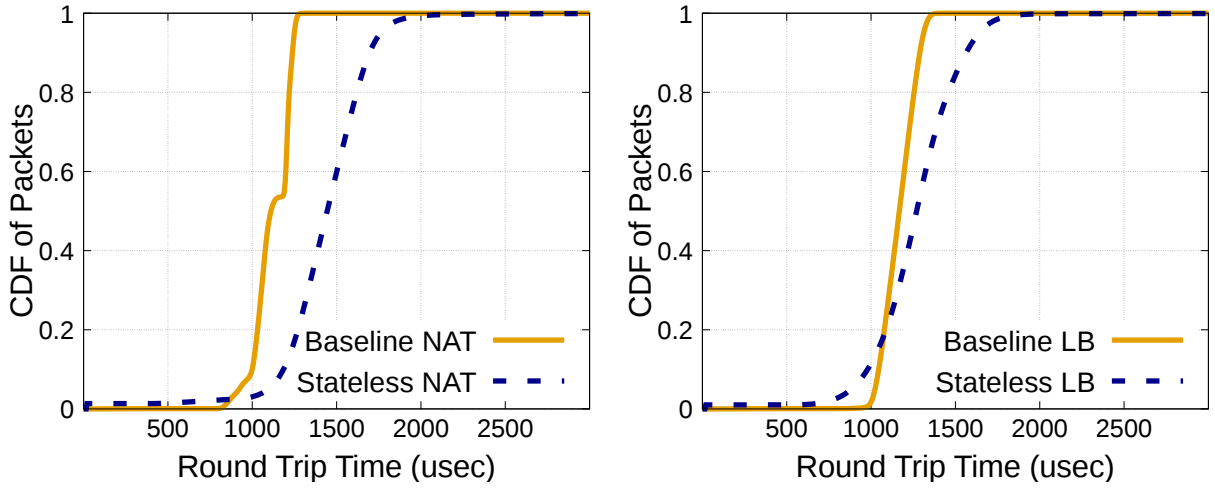
software solutions [48, 125, 133]. To understand the performance of stateless network functions with real traces, we increase the rate of the real trace to more than the original rate at which it was captured[3] , and analyze the achievable throughput. Since the packet sizes vary considerably (80 to 1500 bytes), we report the throughput in terms of traffic rate (Gbit/sec) rather than packets/sec. Figure 2.6 shows that the statelessNF firewall and loadbalancer have comparable performance than their baseline counterpart. The stateless NAT reaches a limit that is 1Gbps lower than the non-stateless version. Finally, we also observe that the performance of the NAT are several Gbps lower than the firewall and load balancer. This is due to the overhead of IP header checksum after modifying the packet IP addresses and port numbers.

2.7.2.2 Latency

The interaction with the remote storage can increase the latency of each packet, as every incoming packet must be buffered until its lookup operation is completed. To evaluate the delay increase, we compared the round-trip time (RTT) of each packet in the stateless and baseline network functions. We timestamp packets, send the traffic through the network function which resends the packets back to the initial host.

Figure 2.7 shows the cumulative distribution function (CDF) for the RTT of packets traversing the

[3] The rates of enterprise traces we found vary from 0.1 to 1 Gbit/sec



(a) NAT (b) Load Balancer

Figure 2.7: Round-trip time (RTT) of packets.

NAT and load balancer[4] . In the 50th percentile, the RTT of StatelessNF packets is only 100μs larger than the baseline's for the load balancer and NAT, and in the 95th percentile the RTT is only 300μs larger. The added delay we see in StatelessNF is a combination of read misses (which can reach 100μs), preparing objects for read requests from RAMCloud, casting returned data, and the actual latency of the request. These numbers are in the range of other comparable systems (*e*.g., the low-latency rollback recovery system exhibited about a 300μs higher latency than the baseline [133]). Further, while the reported latency when using Ethernet (with DPDK) to communicate with RAMCloud is higher than Infiniband (31.1μs, read 100B, and 77μs write 100B), it is still less than the average packet delays reported with StatelessNF system (65us, 100us, and 300us for firewall, load balancer, and NAT respectively). Given the actual network traversals of requests can occur in parallel as the other aspects of the request, we believe that the difference in latency between Ethernet with DPDK and Infiniband can be largely masked. We intend to validate as future work.

[4] The RTT for firewall (both stateless and baseline) showed similar trend to load balancer with a better average delay (67μs less).

2.7.2.3 IPS Analysis

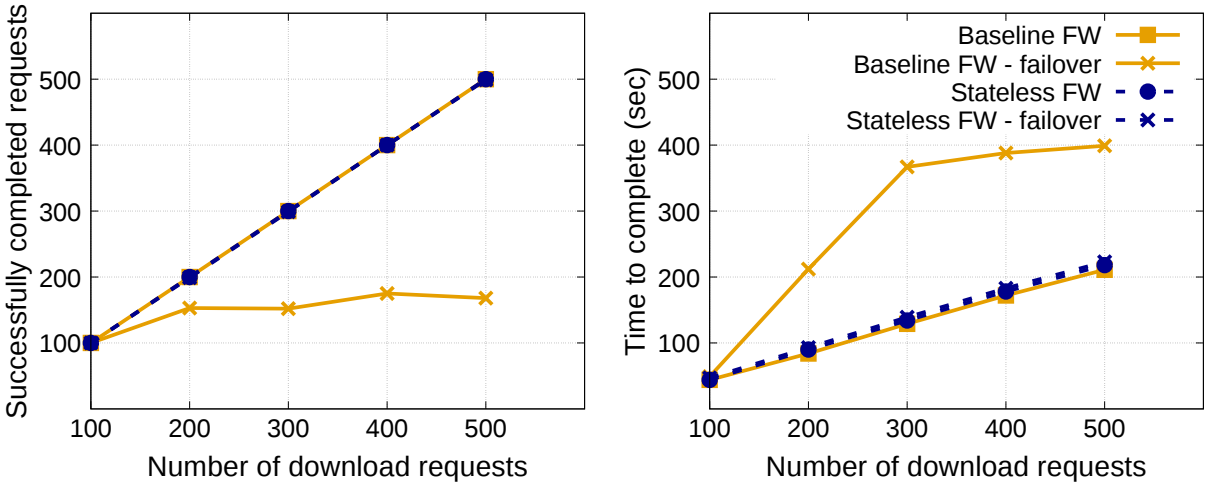
This section analyzes the impact of decoupling the state for an IPS. The overall processing of an IPS is more complex (Section 2.3) than the three network functions we previously analyzed. However, its access patterns to the remote storage is only incrementally more complex.

To analyze the impact of decoupling automaton state into remote storage, we implemented an in-line stateless network function that emulates a typical IPS in terms of accessing the automaton state and performs a read and write operation for every packet. For comparison, we run Snort [128] as an in-line IPS, and streamed real world enterprise traces through both instances: Snort and our stateless emulated IPS. The stateless emulated IPS was able to reach a throughput of 2.5Gbit/sec while the maximum performance for the Snort instance was only 2Gbit/sec. These results show that for an IPS, the performance bottleneck is the internal processing (*e*.g., signature search), and not the read/write operations to the remote storage.

2.7.3 Failure

As we discussed in Section 2.3, in the case of failover, the instance we failover to can seamlessly handle the redirected traffic from the failed instance without causing any disruption for the traffic. To illustrate this effect, and compare to the traditional approach, we performed a number of file downloads that go through a firewall, and measured the number of successful file downloads and the time require to complete all of the downloads in the following cases: 1) baseline and stateless firewalls with no failure; 2) baseline and stateless firewall with failure where we redirect traffic to an alternate instance. In this case, we are only measuring the effect of the disruption of failover, as we assume a perfect failure detection, and simulate this by programming the SDN switch to redirect all traffic at some specific time. If we instrumented failure detection, the results would be more pronounced.

Figure 2.8 shows our results where we downloaded up to 500 20MB files in a loop of 100 concurrent http downloads through the firewall. As we can see, the baseline firewall is significantly affected by the sudden failure because the backup instance will not recognize the redirected traffic, hence will drop the connections, which in turn results in the client re-initiating the connections after a TCP connection timeout.[5]



(a) Completed requests

(b) Time to complete requests

Figure 2.8: (a) shows the total number of successfully completed requests, and (b) shows the time taken to satisfy completed requests.

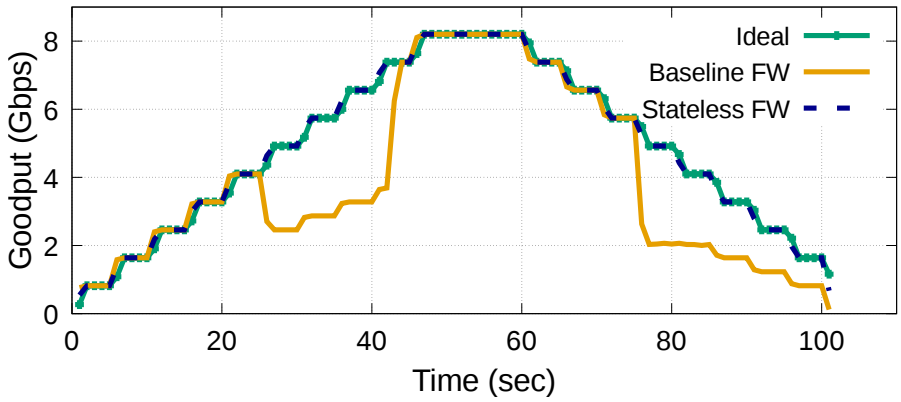


Figure 2.9: Goodput (Gbps) for stateless and baseline firewalls while scaling out (t=25s) and in (t=75s).

Not only was the stateless firewall able to successfully complete all downloads, but the performance was unaffected due to failure, and matched the download time of the baseline firewall when it did not experience failure.

2.7.4 Elasticity

In this chapter, we claim that decoupling state from processing in network functions provides elasticity, where scaling in/out can be done with no disruption to the traffic. To evaluate StatelessNF's capability of scaling in and out, we performed the following experiment: we streamed continuous traffic of tcp packets while gradually increasing the traffic rate every 5 seconds (as shown in Figure 2.9), keep it steady for 5 seconds, and then start decreasing the traffic rate every 5 seconds. The three lines in Figure 2.9 represent: the ideal throughput (Ideal) which matches the send rate, the baseline firewall, and the stateless firewall. The experiment starts with all traffic going through a single firewall. After 25 seconds, when the traffic transmitted reaches 4Gbit/sec, we split it in half and redirect it to a second firewall instance. Then after 25 seconds of decreasing the sending rate, we merge the traffic back to the first firewall instance.

As Figure 2.9 shows, the stateless firewall matches the base goodput. That is because the newly added firewall already has the state it needs to process the redirected packets, and therefore does not get affected by traffic redirection. On the other hand, with the baseline firewall, once the traffic is split, the second firewall starts dropping packets because it does not recognize them (*i*.e., doesn't have state for those flows). Similarly, upon scaling in, the firewall instance does not have the state needed for the merged traffic and thus breaks the connections.

2.8 Discussion

The performance of our current prototype is not a fundamental limit of our approach. Here we discuss two aspects which can further enhance performance.

Reducing interactions with a remote data store: Fundamentally, if we can even further reduce the interactions with a remote data store, we can improve performance. Some steps in this direction that we intend to pursue as future work include: (i) reducing the penalty of read misses by integrating a set membership structure (*e*.g., a bloom filter [18]) into the RAMCloud system so that we do not have to do a read if the data is not there, (ii) explore the use of caching for certain types of state (read mostly), and (iii) explor-

[5] We significantly reduced the TCP connection timeout in Linux to 20 seconds, from the default of 7200 seconds.

ing placement of data store instances, perhaps even co-located with network function instances, in order to maintain the decoupled architecture, but allowing more operations to be serviced by the local instance and avoiding the consistency issues with cache (remote reads will still occur when the data isn't local, providing the persistent and global access to state).

Date store scalability We acknowledge that we will ultimately be limited by the scalability of the data store, but generally view data stores as scalable and an active area of research. In addition, while we chose RAMCloud for its low latency and resiliency, other systems such as FaRM [35] (from Microsoft) and a commercially available data store from Algo-Logic [63] report better throughput and lower latency, so we would see an immediate improvement if they become freely available.

2.9 Conclusions and Future Work

In this chapter, we presented stateless network functions, a novel design and architecture for network functions where we break the tight coupling of state and processing in network functions in order to achieve greater elasticity and failure resiliency. Our evaluation with a complete implementation demonstrates these capabilities, as well as demonstrates that we are able to process millions of packets per second, with only a few hundred microsecond added latency per packet. We do imagine there are further ways to optimize the performance and a desire for more network functions, and we leave that as future work. We instead focused on demonstrating the viability of a novel architecture which, we believe, fundamentally gets at the root of the important problem.

Chapter 3

FOCUS: Scalable and Low-cost Search over Highly Dynamic Geo-distributed State

The ability for controllers to search for nodes that match certain criteria, based on potentially highly dynamic information, is a critical need in many distributed systems in general, and in the homing infrastructure in particular. With the ever-evolving NSPs infrastructures to accommodate the increase in dynamic customer demand, existing systems which typically implement a custom solution based around message queues where nodes push status to a central database, are ill-suited for this purpose. In this chapter, we present FOCUS, a general and scalable service which easily integrates into existing and emerging systems to provide this fundamental capability. FOCUS utilizes a gossip-based protocol for nodes to organize into groups based on attributes and current value. With this approach, nodes need not synchronize with a central database, and instead the FOCUS service only needs to query the sub-set of nodes which have the potential to positively match a given query. We show FOCUS's flexibility through an operational example of complex querying for homing virtual network functions (VNFs), and illustrate its ease of integration by replacing the push-based approach in OpenStack's placement service. Our evaluation demonstrates a 5-15x reduction in bandwidth consumption and an ability to scale much better than existing approaches.

3.1 Introduction

Many distributed systems need the ability to find a node, or a set of nodes, whose attributes match some criteria. A prime example is in cloud management systems, where admins need to identify nodes which satisfy certain properties, such as those that have low CPU utilization, to make scheduling/migration decisions. With the emergence of applications such as edge cloud computing, and Network Service Provider

(NSP) deployments [25, 103, 117] of Virtual Network Functions (VNFs) [41] modern systems are becoming more geographically distributed with more autonomous control within each regional domain. This introduces new challenges around scalability and the need to obtain node attributes directly from the nodes themselves.

Existing approaches, such as those used in cloud management platforms like OpenStack [111], Kubernetes [78], and Mesos [59] can not currently be used beyond the boundaries of a single site because their architectures were not designed for these new requirements. This is because they utilize either a *push* or *pull*-based approach to obtain node information, and neither is sufficient. With *push*-based approaches (used in OpenStack), the nodes periodically **push** their current state through message queues [123] to a central database. Fundamentally, this leads to the centralized database being out of sync with the state as held at the end nodes. Further, as we show in Section 3.3, this approach has limited scalability, requiring applications to work around these limitations (introducing various trade-offs in data freshness, operational complexity and search overhead). In *pull*-based systems, such as used in Google's Borg [141], the controller polls nodes for their current state on demand. This allows for the end nodes to serve as the definitive source of information, but results in expensive communication and ultimately not scalable. While the need for finding nodes that are geo-distributed is recognized as important (e.g., by the OpenStack community [109]), the fact is existing systems simply do not support it.

In this chapter, we introduce FOCUS, a scalable service providing timely search across geo-distributed nodes with varied and highly dynamic state. Its design is inspired by scalable peer-to-peer (p2p) systems such as BitTorrent [17]. In particular, central to FOCUS is a gossip-based system where nodes (geographically distributed over the wide area) form groups based on attributes and geographic proximity, which then allows FOCUS to perform directed queries to only the nodes which have the potential to positively respond to the query. We couple this with a query interface which allows FOCUS to be easily integrated into existing applications and support a wide range of complex queries. We demonstrate its flexibility by considering the operational query requirements of a deployed, complex system to instantiate VNFs in an NSP network (Section 3.5.2). We demonstrate its ease of use by replacing equivalent (but not scalable) functionality within OpenStack, for its VM placement service (Section 3.9).

In summary, this chapter describes FOCUS – a novel distributed service for finding nodes, with the following key technical contributions:

- FOCUS provides a query interface which can be easily integrated into larger systems. We demonstrate this by showing how FOCUS can handle complex queries in a production deployment of an NFV service, and replacing OpenStack's message queue based node finding system for placement with a FOCUS-based solution.
- We introduce an approach which enables directed pulls through sortable attribute-based groups. This is scalable and enables end nodes to be the ultimate source of information, as only a subset of nodes might match a query.
- We integrate a gossip-based peer-to-peer coordination into a general distributed application service, which enables end nodes to self-form into groups. This, in turn, alleviates any load on a central component, which in turn, provides much greater scalability.
- We implement and evaluate FOCUS in a geo-distributed environment with 1600 simulated nodes, and show a bandwidth consumption reduction between 5x and 15x, when compared to various node-finding techniques.

In the rest of this chapter, we first motivate the need for a service like FOCUS through two real world examples (in Section 3.2). We then discuss the limitations of existing solutions in Section 3.3, before describing the architecture of FOCUS, first at a high-level (Section 3.4), then in detail (Sections 3.5, 3.6, 3.7). We then describe our implementation in Section 3.8 and integration into OpenStack in Section 3.9. We present our evaluation results in Section 3.10 and end with conclusion (Section 3.12).

3.2 Motivating Use Cases

Searching for nodes with highly dynamic state is a central task of a number of applications. In this section, we highlight two critical applications from our production systems that illustrate the need for a service like FOCUS.

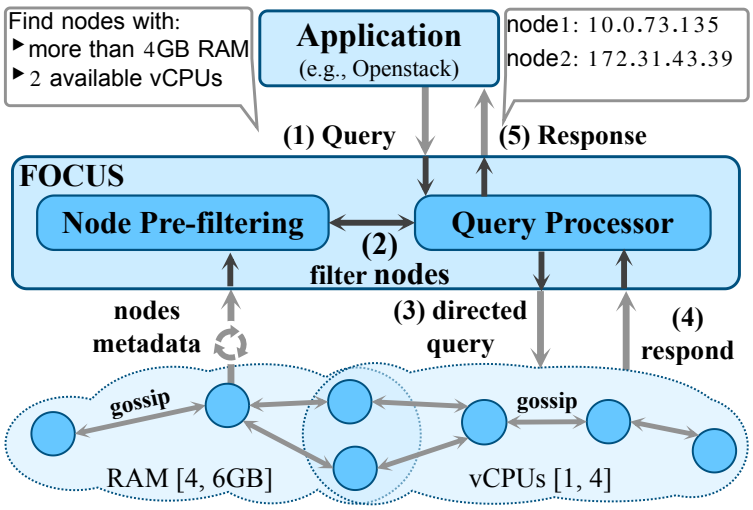


Figure 3.1: High-level overview of FOCUS where end nodes form p2p groups based on attribute values (e.g., memory, vCPUs) for scalable query processing. We describe the details of how FOCUS works in later sections (see Section 3.4 for an overview and Sections 3.5-3.7 for details).

3.2.1 Edge Cloud Management with OpenStack

OpenStack [111] is a cloud management system that helps deploy and maintain virtual infrastructures over large pools of compute, storage, and networking resources throughout a datacenter. To perform this task effectively, in several use-cases in OpenStack there is a need to find physical hosts (among potentially thousands in multi-site and edge deployments [109]) that satisfy certain criteria (exemplified by Table 3.1). For example, the VM Provisioning service will find physical hosts which have enough capacity (RAM, CPU, Disk, Network) to satisfy the needs of the virtual machine that is to be launched. The migration service, which is triggered externally, is an extension of placement and requires similar capabilities to find appropriate hosts. Currently, these services are built around OpenStack agents running on physical hosts pushing status via a message queue [123] to a central database, which limits OpenStack's scalability to barely support thousand hosts [112]. In Section 3.3, we discuss why this is limited in scalability. This limitation is exacerbated in multi-site OpenStack deployments, where multiple hierarchical OpenStack controllers are needed to handle the scale and geo-distribution of the deployment [109], introducing more operational complexity.

Clearly, there is a crucial need for a scalable service like FOCUS, that can be queried for physical hosts (**nodes**) matching constraints. In Section 3.9, we describe how we integrated FOCUS with Open-

Use Cases	Query Examples
VM Provisioning / Live Migration	Get hosts that meet new/migrated VM resource requirements
Verify Service Status	Get hosts by service type (e.g., compute, scheduler, etc)
Tenant Usage Reports	Get hosts belonging to a project ID
Hot Spot Detection	Get active/idle hosts

Table 3.1: Example queries from OpenStack source code [110]

Stack, seamlessly replacing its message-queue-based functionality for placement. Further, with a system like FOCUS, we could go beyond just placement tasks and perform periodic monitoring efficiently (*e*.g., find hosts with a high cache miss rate, indicating that VMs should be migrated). In addition, when using FOCUS within a multi-site deployment of OpenStack, FOCUS can provide a full view of the system (across all sites), reducing the operational complexity of OpenStack imposed by gathering and maintaining node information across all geo-distributed sites.

3.2.2 NFV Automation for Geo-distributed Network Services

The Open Network Automation Platform (ONAP) [103] enables Network Service Providers (NSPs) to automate and support the lifecycle management of complex virtual network functions [41]. Given a network service chain consisting of multiple network functions that traffic must traverse (*e*.g., a firewall and a load-balancer), a key task is to instantiate the chain. This could involve launching new virtual network functions (VNFs), or re-purposing existing VNFs that provide the required service. This task is performed by the ONAP 'homing' service (currently deployed in our production network) that often needs to find sites or service instances that satisfy complex service chain requirements involving a combination of the queries such as the ones in Table 3.2. There could be hundreds and potentially thousands of geo-distributed sites (for edge use-cases) that constitute an NSP's customer premises, central offices and full-fledged datacenters.

To avoid complexity, the homing service currently finds candidate sites and services by executing the above mentioned queries sequentially to various central inventories that only maintain static information (*e*.g., site/service attributes in Table 3.2). Following this step, it hands off the actual task of instantiation to separate processes within the chosen sites (which performs another set of queries, more similar to Open-

Category	Query Examples
Sites	Get all service provider-owned cloud sites
Services	Get services of type vGateway, vDNS
Site attributes	Sites within 100 miles of a given location and support SRIOV with KVM version 22
Service attributes	vGateways that have the VLAN Tag for the matching customer VPN ID
Site capacity	Sites that have a certain tenant quota, available upstream bandwidth, vCPU, available Memory
Service capacity	vDNS that can support 10000 resolutions/second, vCDNs that has a hot cache for Customer Y

Table 3.2: Example queries in an operational ONAP deployment.

Stack). While the homing service is currently constrained to just static properties, many customers aspire to dynamic properties (*e*.g., site/service capacity).

FOCUS can be a simple replacement for the inventory-sourced querying in the homing service, where each site and service is a 'node' in FOCUS (described in Section 3.5.2). This would enable complex combination of the queries shown in Table 3.2 supporting both static and dynamic properties. Alternatively, ONAP's architecture is largely driven by the need to handle a large scale – managing network services across hundreds, soon to be thousands, of sites, where each site has hundreds or thousands of servers. As such, it makes sense to separate functions – the homing service deals with site/service-level constraints and a cloud-level service like OpenStack handles host-level constraints. With FOCUS, we could rethink the architecture wherein the homing service performs both functions, using FOCUS to optimize the search over all hosts/services across sites in a scalable manner.

3.3 Limitations of Existing Systems

To motivate how FOCUS should be architected, we first examine the way node finding is commonly built into systems today (based around message queues) and why we believe this is not a great match for this purpose. We then discuss some architectural alternatives which could (at a glance) be a solution, but have significant shortcomings as well.

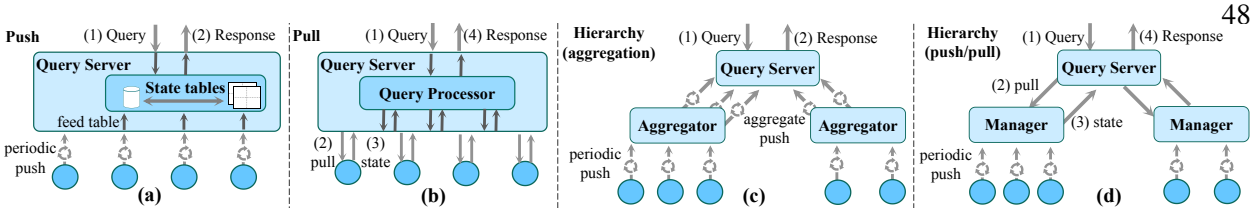


Figure 3.2: Various alternate architectural designs that can be used for node finding.

3.3.1 Node Finding with Message Queues

As illustrated in Figure 3.2a, the approach is broadly characterized by the nodes periodically pushing information about their current status (attributes and current values) to a central database through a message queue. This allows the query processing server to respond to any query to find a node by simply querying the database.

As an example system that is built like this, OpenStack has agents that run on each compute node (usually one per physical host). These nodes each produce a few messages per second containing their current state (*e*.g., number of VM instances, available memory, disk, CPUs, etc) through RabbitMQ [123] (the default messaging queue of OpenStack). A process within OpenStack consumes this information from RabbitMQ and feeds the information into a database.

To quantify the scalability limitation of message queues, we deployed a VM on Amazon EC2 [5], dedicated to run a RabbitMQ server with 8GB of RAM and a CPU with 4 virtual cores (each 2.4GHz), and we used 5 other VMs to host simulated producers (nodes). In each run, we had 100 consumers consuming from 100 queues to which the producers push their messages (we found this to be the most effective way to consume data). Each producer was sending five 1KB messages per second to the server (mimicking OpenStack hosts' behavior). We measured message latency and CPU usage of the RabbitMQ process 30 seconds into the tests.

Figure 3.3 shows the latency (left y-axis) and CPU usage (right y-axis) when we varied the number of producers from 1K to 8K. As shown, RabbitMQ hits its scalability limit around 6k nodes, and crossed over 50% CPU utilization as early as 2k nodes. While one can argue that adding more RabbitMQ servers can scale the solution, we argue that this not only will consume more resources, but it will also complicate

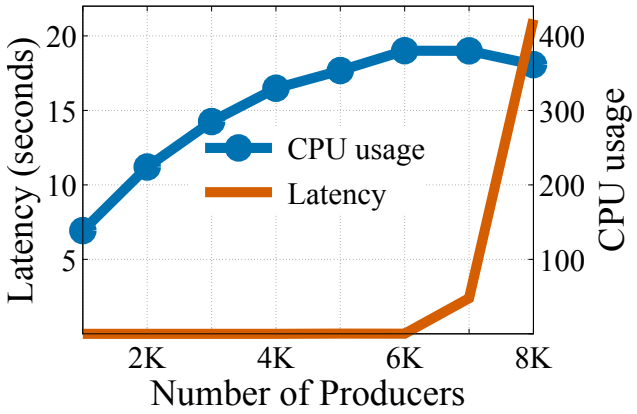


Figure 3.3: RabbitMQ test showing latency of messages and CPU usage of the RabbitMQ process while varying the number of producers (i.e., nodes).

the management of RabbitMQ (multiple RabbitMQ servers need to synchronize through distributed consensus [60, 105]). Such model puts too much emphasis on the messaging queue, making it a bottleneck and a single point of failure. We, therefore, conclude that message queues are not the most efficient means for finding nodes.

3.3.2 Alternate Architectures

Before presenting our gossip-based approach, it is worth considering some alternatives.

3.3.2.1 Pull

A first alternative is to pull information from the nodes in response to a query, rather than have the nodes periodically push information. This is illustrated in Figure 3.2b. When a query comes in, the server can poll the nodes for their current state. The nodes would then send a response to the server, which would process the responses and form a response to the node finding query.

Pull-based approaches are generally not considered scalable, as the server needs to query many nodes simultaneously, and the synchronized responses coming back from the nodes can result in server overload, or problems such as TCP incast [24].

3.3.2.2 Hierarchy

It is natural to assume that we could just add hierarchy to address the limitations of push-based message queues or simple pull-based approaches. Here, we consider two approaches to hierarchy and conclude that neither is ideal.

Aggregating: Rather than N nodes all sending to a single central server, we can introduce a layer of nodes that simply aggregate the data (as illustrated in Figure 3.2c). We note that this approach reduces the event rate (number of messages) at the central server, but does not reduce bandwidth consumption, nor does it reduce the event rate at the database.

Sub-setting: Rather than push all the way to the central server, we could effectively divide the infrastructure into subsets that each are designed with the nodes all pushing to their subset manager (as illustrated in Figure 3.2d). Then a central server would query (pull) each of the subset cloud managers anytime a query comes in. This has two key problems. First, this solution partitions the infrastructure, which as has been argued before, is not ideal as crossing the partition boundaries are an added challenge [51]. Second, this inherently increases management complexity – we are now running and managing several cloud managers as opposed to just a single one.

3.4 FOCUS Architectural Overview

FOCUS, as illustrated in Figure 3.1, is a system which provides a service to systems that need to find a set of nodes which have certain attributes. Overall, we have two main objectives: (i) serve as a general purpose service for node finding across many applications, and (ii) efficiently scale both in terms of performance and operational complexity. In this section, we highlight the key design/architectural aspects that help achieve this and then elaborate in subsequent sections.

Integrable Query Interface: In order to be useful to applications, we need FOCUS to have an interface that is easy to integrate and powerful enough to cover a variety of applications' needs. We provide a simple REST API in which a query contains attributes and the range (or specific) values to match. Further, we demonstrate the richness by illustrating the queries needed in a production deployment of NFV services.

Query Processing with Directed Pulling: As explained in Section 3.3.2, simple pull-based approaches do not scale beyond a small set of nodes. Yet, at the same time, pulling provides the ability to have the most up-to-date information. To balance between the goals of the scalability (both in terms of performance, and in terms of operational management) and supporting applications with dynamically changing nodes, we introduce directed pulling in FOCUS. Specifically, in our solution, we pre-filter nodes and only send pull requests to nodes which **have the potential** to positively match the query.

Gossip-based Node Coordination: To realize directed pulling, we use a gossip-based approach to group nodes based on attribute and value. Crucially, the nodes themselves organize into groups and gossip with each other in order to determine when group membership should change. Then, if needed, they communicate with the central FOCUS node, and change groups. This distributes load from the central server to all nodes, and enables more decentralized decision making. To answer a query, FOCUS simply needs to know which groups a given node is part of (or, said in the inverse, which nodes are part of which groups). Note that, with this approach, we avail all the benefits of the sub-setting approach in Section 3.3.2 without incurring any of the operational complexity of maintaining multiple managers.

3.5 Integrable Query Interface

A key goal of FOCUS is to be useful across a broad range of distributed applications – *i*.e., it has a query interface that is easily integrated, and rich enough to support the needs of applications. In this section, we first discuss the abstractions, then describe a real-world example.

3.5.1 Abstractions

In this section, we provide a high-level overview of the abstractions provided by FOCUS. As depicted in Figure 3.1, an application can specify constraints for the nodes it wants to find, and FOCUS will efficiently query the nodes and return nodes (out of possibly thousands) that satisfy the constraints.

Node Attributes: Nodes have attributes that can be described as either *static* or *dynamic*. Values of *static* attributes do not change (e.g., number of CPU cores) while values of *dynamic* attributes can and do change over time (e.g., free memory). For multi-site environments, the nodes within a given site inherit the

global attributes of that site. For example, a node representing a host in a cloud site will contain not only host attributes such as available CPU and memory, but will also inherit attributes such as "US-East" which describes the cloud site's geographic region.

Query Structure: Queries are attribute-oriented, meaning that each application issuing a query should specify the attributes and their desired values. A query structure contains a list of *queryable* attributes, and for each attribute there are the following fields: *name*, *upper bound value*, *lower bound value*, *limit*, and a *freshness* parameter. The attribute *name* is used to describe the attribute of interest to the requester application. The *upper bound* and *lower bound* values are used to support lesser/greater than operations. If an exact match is needed, then both bounds should be of the same value. The *limit* specifies the maximum number of responses to be returned. And finally, the *freshness* field can be specified in terms of milliseconds (a value of zero means the response must be as close to real time as possible to guarantee extremely fresh results). We note that this is one version of a query structure, and there are multiple versions that FOCUS supports for other attribute types (e.g., location, text-based attributes, etc).

3.5.2 Example Queries used in VNF Homing

To illustrate the use of FOCUS, here we consider an operational example of the VNF homing service described in Section 3.2.2. In this example, we specifically present the case which matches today's use (first searching for sites and services, and then performing instantiation), rather than re-architecting the solution which could be enabled by FOCUS (searching for physical hosts and services across sites and deploying a service chain in a one step process).

Figure 3.4 shows the homing requirements of a virtual Customer Premises Equipment (vCPE) [96] network service, that provides residential broadband connectivity. Figure 3.4a shows the layout architecture, connecting the residence to the vG (virtual gateway) hosting infrastructure at the Service Provider Edge (PE). Here, the bridged residential gateway (BRG) is the vCPE located at the residential customer premises, while the vG Multiplexer (vGMux) is a shared network function at the PE that maps layer-2 traffic between a subscriber's BRG and its unique vG, ensuring traffic isolation between customers.

Homing the vCPE service requires finding a slice of an existing vGMux instance and finding a suitable

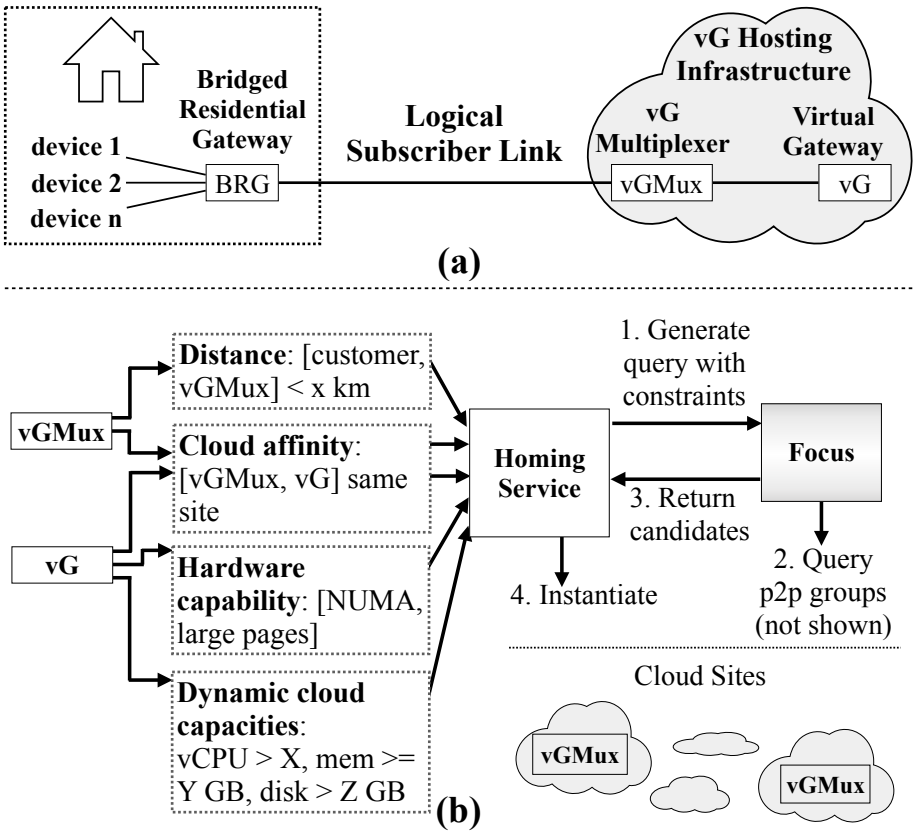


Figure 3.4: VNF homing: an apt use-case that illustrates the use of FOCUS for homing the residential virtual Customer Premises Equipment (vCPE) service [96] in ONAP [103].

cloud site for spinning up a new instance of the vG. Figure 3.4b shows the homing policies (or constraints) that drive the selection of an optimal vGMux and the corresponding PE site to host the vG for a given customer. While the first two constraints are relatively static, the hardware capabilities of a cloud site may change as new host aggregates are added, and instantaneous site capacities may vary at even shorter time scales since resources are typically shared among multiple services and customers. As shown in Figure 3.4b, FOCUS is a perfect fit for this problem wherein those constraints can be expressed as a query to FOCUS which will return a set of candidate vGMux instances and sites (FOCUS 'nodes') that satisfy all constraints.

3.6 Query Processing with Directed Pulling

In this section, we discuss the key concept of grouping nodes based on their attribute values, how FOCUS is able to pull from the right subset of nodes, and key optimizations for FOCUS's query processor.

Attribute-based Grouping: As demonstrated in section 3.5, the state of each node is described in the form of attributes (e.g., CPU utilization, free memory and disk, location, etc). Naturally, grouping nodes based on their attribute values makes it possible to filter out many nodes that cannot satisfy certain queries. Driven by our query structure, we group nodes based on attribute values that are within a specific range. For example, in Figure 3.1, there are two groups of nodes – one for nodes with free RAM of 4 to 6GB, and another for nodes with 1-4 virtual CPUs. Note that a node can be in multiple groups simultaneously (e.g., having 5GB of RAM and 2 vCPUs). In section 3.7, we discuss how nodes form such groups in a dynamic manner, adapting to changing attribute values.

Query Conjunctions through Sorted Pulls: Having pre-filtered nodes and prior (coarse-grained) knowledge of the current state of each node in the system, it is possible for FOCUS to direct queries to only those nodes that have the potential to satisfy the queries. Specifically, when FOCUS receives a query, it parses the query and sends it to the corresponding groups that satisfy the query conditions. The members of the group, then, will respond with their current state. For instance, consider a query to retrieve nodes with 4GB of RAM. FOCUS will send the query **only** to the group that has 4 to 6GB of free RAM (exemplified in Figure 3.1).

Multi-attribute/constraint queries, if not handled well, can undermine the advantages of pre-filtering and attribute-based grouping. That is, if a query containing too many constraints for different attributes is sent to every single group of nodes that correspond to each attribute, then this can quickly degenerate to the case where the query is sent to every single node in the system. Instead, FOCUS sends the query to the smallest group that corresponds to one of the query's attributes (mechanism described in Section 3.7). Then, the nodes within that group can answer to all constraints in the query. This narrows down the scope of nodes to which a query must be sent even further.

Optimizations: In addition to sending queries to the smallest group, we further optimize our querying with a cache to store query responses along with a timestamp of when they were fetched. Checking the cache is the first step in processing a query. As described in section 3.5, each query has a freshness parameter, which is checked against responses fetched from the cache. Should cached responses not qualify for the query freshness or in the event of a cache miss, the query will be sent to the appropriate group. Moreover,

under heavy-load conditions, and after determining what groups to which the query must be sent, FOCUS has the mechanism to delegate to the querying application the act of actually sending the query to the nodes. As a result, the load on FOCUS is alleviated, making it more lightweight and scalable. However, responses for those delegated queries will not traverse FOCUS, and consequently will not be cached.

3.7 Gossip-based Node Coordination

Our attribute-based groups are p2p groups that implement the gossip protocol [29], through which each node gossips membership information with only a few members of the group, who in turn gossip with other nodes until convergence in reached. The gossip channel is also used to disseminate queries received from the FOCUS server. In this section, we describe how nodes form those groups and how FOCUS is able to maintain information that is later used to process queries.

Dynamic Groups Management: Upon registering with FOCUS, a node reports its current attributes and the corresponding values. In return, FOCUS will provide entry points into the appropriate groups for the node to join. Each of the node's dynamic attributes corresponds to a specific p2p group based on the attribute value[1] . When there are no existing groups that suit the new values of a node, FOCUS will instruct that node to start a new group and, in turn, will be an entry point for future nodes. In addition to providing entry points, group suggestions from FOCUS also contain group ranges. A group range is used by nodes to detect when it is necessary to move to other groups (when new values fall outside the group range). Based on predefined attribute value cutoffs, FOCUS decides the range of each attribute-based group.

Further, in order to keep groups from growing indefinitely, which as we show (in section 3.10) has an impact on query latency, FOCUS keeps track of how many members are in a group. When a group size exceeds a certain threshold, FOCUS will fork groups by suggesting new groups to new nodes. We note that a group that tracks an attribute spanning multiple geographic locations (e.g., free RAM) could be formed disregarding the geographic locations of the nodes in the group, which could degrade performance. However, we can seamlessly split groups when they exceed certain geographic thresholds (like maximum

[1] Static attributes are maintained in the FOCUS distributed data-store (described in Section 3.8) and therefore do not need to be managed via groups.

distance among nodes) by treating them as separate attributes tied to location. For example, if a group containing nodes that have more than 4GB of free RAM traverses locations across say Texas and California, we simply create two groups: (nodes with more than 4GB of free RAM in Texas) and (nodes with more than 4GB of free RAM in California). Accordingly, when FOCUS processes a query that does not specify a location (e.g., *get nodes with greater than 4GB of RAM*), FOCUS will query groups with nodes having more than 4GB of RAM in all locations (or until it satisfies the *limit* parameter in the query as described in Section 3.5.1), aggregate the results, and return them to the query initiator.

Group Member List through Representatives: FOCUS maintains a list of member nodes for each group to serve as entry points for other nodes, to enable queries, and to perform operations like forking the group based on size, geography, etc. Even if this list is relatively stale, as long as this list includes some reachable, live member, FOCUS can pull the latest attribute values for the group (explained later in this section). To obtain this list, FOCUS randomly selects a small (configurable) number of nodes in each group to be the representatives and asks them to periodically upload the group member list. Since modern gossip protocols exchange and construct member lists, representatives nodes need to do minimal additional work in uploading this information. Further, the randomized representative node selection ensures that the workload is distributed evenly. In a group that has high churn rate, more representative nodes and/or more frequent updates are required. Finding the optimal upload frequency for different systems is an important aspect of future work.

Load-balanced Query Routing within p2p Groups: To receive a query response from a p2p group, FOCUS can send the query to any member of the group which, in turn, will gossip the query with other members of the group. Query responses from group members, however, will be directly sent to the member who originated the query in order to allow fast query processing. A key design decision of FOCUS is that the load must be distributed across all nodes. Further, FOCUS needs to be resilient to failure of nodes in the attribute groups. Therefore, every time a query needs to be sent to one of the p2p groups, FOCUS randomly picks a different group member, as opposed to just sending them to the representatives described above. In section 3.11, we describe future work to address the efficiency/communication trade-offs in this design.

Since node coordination is gossip-based, convergence can be relatively slow and if a query is sent to

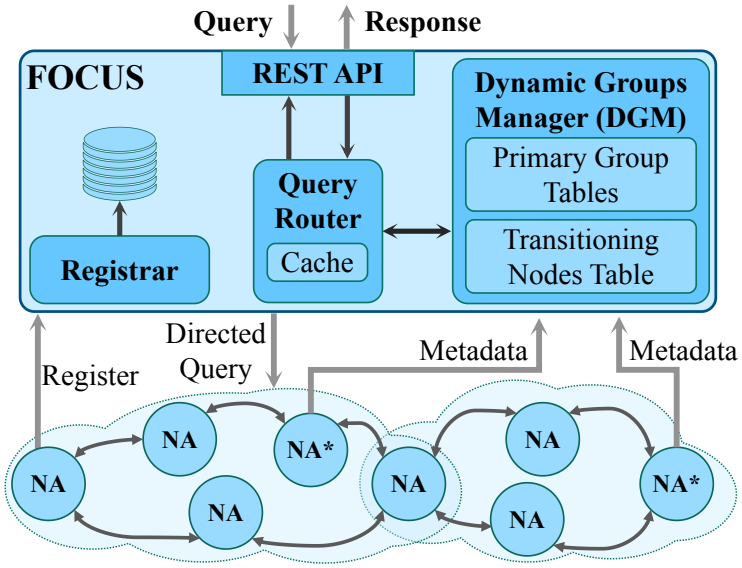


Figure 3.5: Internal design of FOCUS, showing node agents (NA) while group representatives (denoted with *) push their groups' metadata to FOCUS.

a group immediately after a new node has joined, the new node may not receive the query. To solve this problem, FOCUS maintains a table in its data-store that tracks nodes that are transitioning between groups or are entering a new group (when they ask FOCUS for group suggestions). To ensure inclusiveness, FOCUS also includes nodes in this table when processing a query.

3.8 Implementation

In this section, we describe the internal design of FOCUS (Figure 3.5) and provide details of its implementation. Specifically, we implemented FOCUS in Java with 3.1K lines of code (1.9K LoC for the FOCUS service and 1.2K LoC for the node agent). We used *Apache Cassandra* [80] as our service data store, *Eclipse Jetty* [37] as our web server, and *HashiCorp's Serf* [56] as our p2p fabric. First, we describe each component of the FOCUS service and then describe the node agent.

3.8.1 FOCUS Service

Each of the three main components of FOCUS (Registrar, Dynamic Groups Manager (DGM), and Query Router) exposes its services through a REST API that is hosted on a *Jetty* server. The input and

output of each API call is JSON-formatted [70].

3.8.1.1 The Registrar

The Registrar listens for node registration requests. Each request contains certain information about the new node, including: node IP address, a list of attribute-value pairs, and the port through which the node can receive commands and queries from FOCUS. The Registrar stores the new node information at the FOCUS database, which is backed by a Cassandra cluster to provide resiliency and fault tolerance. For each *static* attribute, the Registrar creates a table containing: node ID, attribute value, and a timestamp field. We also use an additional field to store all other attribute-value pairs for each node. For instance, if the new node has the following *static* attributes (*arch*:*x86*, *cores*:*8*), then an entry at the table for the *arch* attribute will look like the following.

node ID	**arch**	**attributes**	**timestamp**
IP address	x86	{cores:8}	time value

Storing other attributes in each attribute table makes it much more efficient to query the database. That is, to perform a query with multiple attributes, we just need to query one table, the table with the lowest number of entries. These tables are also updated by the DGM when it learns new information.

3.8.1.2 The Dynamic Groups Manager (DGM)

Deterministic Group Naming: Choosing consistent group IDs is crucial when referring new nodes to groups as well as when routing queries to the desired group. The DGM, using a deterministic group naming function, constructs group names using an attribute cutoff. For instance, if the *disk* attribute cutoff is set to 10, then a group named *disk.10GB* will contain nodes that have between 10 and 20 GB of free disk space. Our deterministic group naming function accepts an attribute-value pair, and returns the corresponding group name.

Group Tables: The DGM keeps track of groups by storing them in a *primary* key-value lookup table, which frequently gets synchronized with the Cassandra data-store. We note that since group information

is essentially maintained by the groups themselves, failure recovery of the DGM comes naturally. That is, when the DGM fails and a new one is instantiated, group representatives will send their corresponding group information, which the new DGM uses to populate its *primary* group tables. As discussed in section 3.7, information about nodes transitioning between groups will be kept in a temporary table until they appear in one of the groups updates.

3.8.1.3 The Query Router

In order to separate the load between the northbound API (consumed by querying applications) and the southbound API (consumed by nodes), we bind the Query Router to a different port than the DGM. It runs a process that has a cache table in memory, which is checked every time a query is received. For queries with only *static* attributes, the Query Router will get the corresponding values directly from the database. Otherwise, it will send the query to the corresponding group after consulting the DGM. To prevent FOCUS from indefinitely blocking on queries, FOCUS uses a configured timeout after which the query processing will abort.

3.8.2 Node Agents

Our node agent consists of two light-weight processes: a node manager and a p2p agent, both of which run on every node in our system. The node manager is responsible for communicating with the FOCUS service for managing node registration and requesting group suggestions. And the p2p agent (which runs a Serf client [56]) is responsible for connecting to other p2p agents for each of the node's attribute groups, one group per attribute.

Node Manager: The node manager has three tasks. (*i*) It runs OS commands that collect resource (attribute) information (CPU usage, free RAM and disk, etc). (*ii*) It handles communication with the FOCUS service. (*iii*) It provides an interface for receiving queries and commands (e.g., representative election) from the FOCUS service. When a node receives a query, the node manager will gossip the query (via its p2p agent) to the members of its group and gets the response back.

p2p Agents: Each node runs a separate Serf agent for each group it joins. When a node requests group

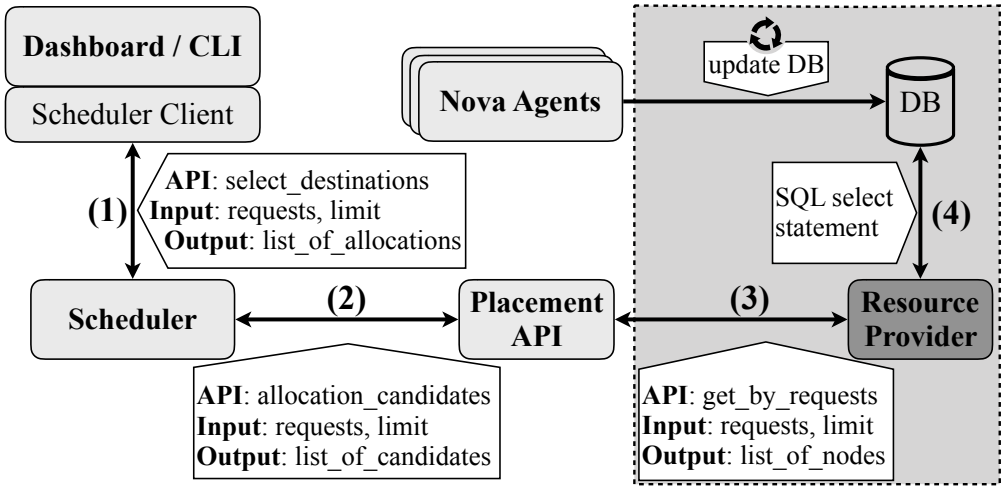


Figure 3.6: Order of API calls within OpenStack for provisioning a VM instance. The shaded area shows where FOCUS is integrated by replacing the object that queries the database with a FOCUS client that queries the FOCUS service.

entry information from FOCUS, it will get a list of entry points for each group. Each entry point consists of the IP address and the port number at which the Serf agent of the node is listening. The requesting node can use this information to join the p2p group. In the case of first node to register for a group, FOCUS will let the node know there are no entry points; consequently, the node will start a Serf agent and immediately let FOCUS know about its Serf binding port so that future nodes can join. Note that, the Serf agent is configurable with a set of parameters, including: the number of neighbor nodes to gossip with (gossip fanout) and a gossip interval parameter. In our implementation of the FOCUS node agent, we set the fanout to 4 nodes and the gossip interval to 100 milliseconds[2] . This setting achieves a balance between overhead on the node agents and query performance.

3.9 OpenStack Integration

In this section, to demonstrate FOCUS's usability, we provide a detailed overview of how we integrated FOCUS into OpenStack's VM placement service (using OpenStack version 3.15.0 and Nova version 18.0.0), thereby providing a much more scalable solution compared to RabbitMQ (as described in Section 3.3). First, we provide an overview of how OpenStack finds nodes for new VM placements (and live

[2] This allows a 400-node group to reach convergence in as little as 0.6 sec.

migration), which is illustrated in Figure 3.6.

Finding Nodes for VM Placement: OpenStack follows a model that resembles the one in Figure 3.2a, where Nova compute nodes running on each physical host periodically push their state updates to a centralized database through RabbitMQ, containing information about current capacities (cpu, ram, disk, etc) and virtualization-specific information (number of installed VMs, number of vCPUs, etc). Each VM placement request object takes the following form.

```
struct{ int limit, dict resources}
```

The `limit` field is used to limit the number of nodes in the response. The dictionary of `resources` contains the minimum required resources for the requested VM image. Such resources typically are: `RAM` (specified in Megabytes), `Disk` (specified in Gigabytes), and `VCPUs` (an integer value specifying the number of required virtual CPUs).

When a VM placement request is issued, the following steps take place (in accordance with steps in Figure 3.6). **(1)** A scheduler client (mainly used by the dashboard or CLI) will call the scheduler API `select_destinations` by passing the requested resources and a limit for the desired number of placement candidates. **(2)** The scheduler, in turn, will verify the request and then call the Placement API GET method `allocation_candidates`, which returns a list of placement candidates so that the scheduler can issue commands to the desired candidates to spawn a new VM. Upon receiving a placement request, the Placement service **(3)** calls the Resource Provider's `get_by_requests` method, which **(4)** queries the database and returns a list of candidates.

Integrating FOCUS: The `allocation_candidate` class of the placement service makes an indirect call to the database in order to fetch the available hosts and their state. The following line of code makes the request.

```
cands = rp_obj.AllocationCandidates.get_by_requests(requests,limit)
```

We replace this particular functionality, corresponding to the shaded box in Figure 3.6 for steps (3) and (4) with a FOCUS-based solution. Specifically, we replace the above call to the central database with the following single call to FOCUS.

```
cands = fc_obj.query(requests,limit)
```

Where `fc_obj` is an instance of a class that we implemented to handle queries from OpenStack to FOCUS. Currently, it supports placement queries, and adding support for other queries merely requires adding more functions to this class.

Augmenting FOCUS's Node Agents: We augmented our node agents (that now run on the physical hosts running the Nova compute agents) with the `libvirt` virtualization library [84] in order to gather resource information. Our node agent interfaces with `libvirt` and connects to the QEMU hypervisor [122] to gather the required information. This addition to our node agent resulted in less than 100 lines of additional code to the original node agent code. Although our current integration connects to the QEMU hypervisor, we can easily integrate with other hypervisors (Xen [137], KVM [87], VMWare ESX [142], etc) that `libvirt` supports.

3.10 Evaluation

In this section, we evaluate the performance of FOCUS and compare it against different node finding approaches. Our evaluation of FOCUS answers the following questions:

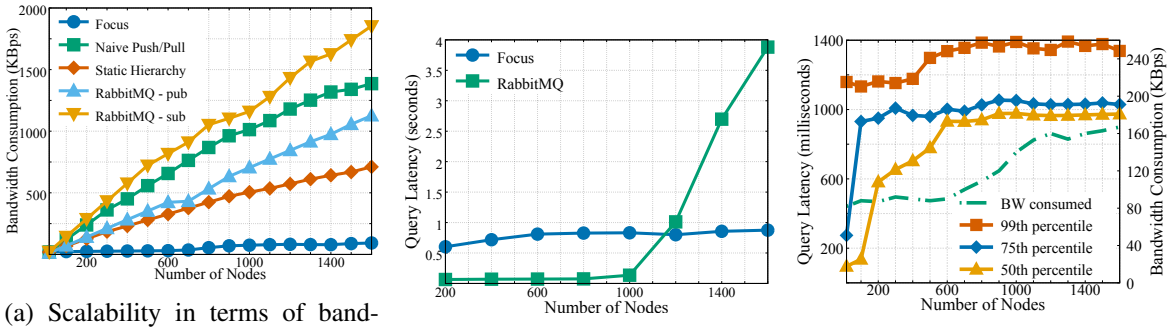
(1) How does FOCUS scale compared to other solutions?

(2) How efficiently does FOCUS perform with real-world query traces?

(3) What are the FOCUS benchmarks with respect to group size, and overhead on the node agents?

3.10.1 Testbed Setup

To evaluate the performance of FOCUS, we deployed it on Amazon's EC2 [5] and to simulate geo-distributed nodes, we chose four different EC2 regions in North America: Ohio, Canada, Oregon, and California. In each region, we instantiated 8 VMs each with 16GB of memory and 4 vCPUs to host our FOCUS node agents. Since multiple node agent programs are consolidated onto the same VM in our ex-



(a) Scalability in terms of bandwidth consumption of the query server for multiple node finding approaches.

(b) Average query latency (*s*) for FOCUS and RabbitMQ, when processing 40 queries per second.

(c) Query latency (*ms*) for real-world queries with nodes deployed across 4 geographical regions.

Figure 3.7: FOCUS's performance for different metrics when compared to other systems and when processing real-world queries.

periments, we introduced a randomness factor[3] to the node agents which they use to change their attribute values so that they do not report the same information of the VM on which they run. Each node agent reported 4 attributes: *CPU usage*, number of available *vCPUs*, free *RAM_MB*, and free *DISK_GB* space. The attribute-based group cutoffs were as follows: {CPU usage: 25%, vCPUs: 2, RAM_MB: 2048MB, disk: 5GB}. This means that nodes with CPU utilization between zero and 25% will be in the same group, and nodes that have 1 to 2 virtual CPUs will be in the same group, and so on.

3.10.2 FOCUS vs. Existing Systems

Bandwidth Consumption: In this experiment, we evaluate FOCUS's scalability by measuring the bandwidth consumption at the query server. We also compare our results with the following node finding approaches. (*i*) Naive push and pull, where node state is either frequently pushed from the nodes (Figure 3.2a) or pulled on-demand (Figure 3.2b). (*ii*) Static hierarchy (Figure 3.2d), where the number of state managers is 16.[4] We also compare against (*iii*) RabbitMQ with two configurations (publish and subscribe), where nodes either periodically publish information (pub) or subscribe for queries (sub) and then respond. The query/update frequency is 1/second.

[3] The randomness factor depends on the attribute value range. E.g., the value for *cpu usage* can be randomly assigned from 0 to 100.

[4] We chose 16 because that was the average number of group representatives that are in charge of reporting group information to FOCUS.

Figure 3.7a shows that FOCUS consumes less bandwidth than other systems. For instance, when the number of nodes reaches 1600, FOCUS can eliminate up to 86%, 92%, 93%, and 95% of the communication between the server and nodes when compared to static hierarchy, RabbitMQ (pub), naive push/pull[5] , and RabbitMQ (sub), respectively. This shows that FOCUS, with attribute-based grouping and directed pulling, can scale much better than other approaches.

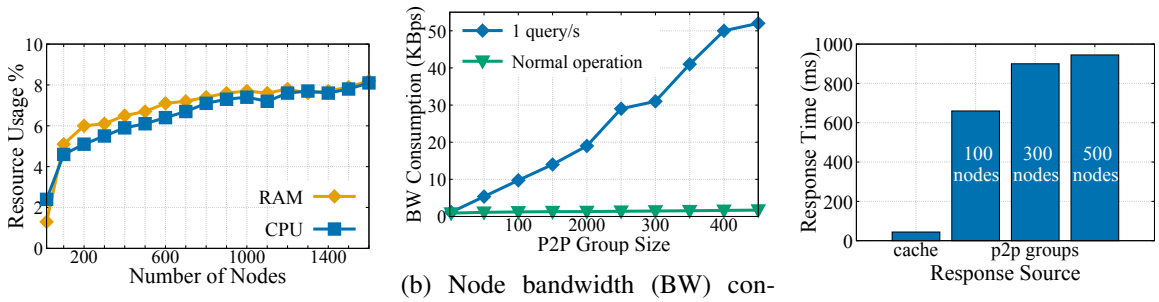
Query Processing Latency: Figure 3.7b shows the average query latency for FOCUS when compared to RabbitMQ while processing 40 queries per second. Note that FOCUS was deployed according to the setup described earlier in section 3.10.1; however, our RabbitMQ deployment was in one region of EC2 where we ran a RabbitMQ server and multiple simulated producers. Up to 1K nodes, RabbitMQ shows faster responses. However, after 1K nodes, RabbitMQ could not scale, while FOCUS's latency stays relatively constant. This is because instead of sending queries to all nodes, FOCUS used directed pulling to send queries only to the corresponding p2p groups.

3.10.3 Query Latency for Real-world Traces

In order to get a sense of how well FOCUS can perform in real-world deployments, we replayed a cloud trace from the Chameleon cloud testbed [22], containing OpenStack KVM events. The trace contains over 75K VM placement events over the course of 10 months. Those events provide resource requirements, which we parsed into our queryable attribute object (described in section 3.5), and then replayed those queries to FOCUS. We replayed the traces at an accelerated rate (15,000x faster) to test how well FOCUS performs under heavy loads. All queries were sent to the p2p groups, and the cache was disabled for this experiment.

Figure 3.7c shows the latency per request for the 50th, 75th, and 99th percentile of queries (left y-axis). The results show that there is a steady increase in the response latency until the number of nodes reaches about 600 nodes, after which the latency stays relatively constant. We monitored the FOCUS service at each run, and found that after 600 nodes, the average group size did not significantly increase ($\approx$150 members per group), whereas the number of groups kept increasing. This highlights the benefit of using an

[5] Naive push and pull showed identical results; hence, merged into one line.



(a) CPU and RAM usage of the FOCUS server while processing queries at 40 queries per second.

(b) Node bandwidth (BW) consumption under two conditions: normal operation and query processing every second.

(c) Query response latency for different response sources: cache or p2p groups of different sizes.

Figure 3.8: Various microbenchmarks showing different aspects of FOCUS's performance.

attribute-based grouping.

3.10.4 Microbenchmarks

In this section, we provide microbenchmarks of various aspects that impact FOCUS's performance.

Resource Usage of the FOCUS Server While replaying real cloud traces (discussed in section 3.10.3), we measured the CPU and RAM usage of the FOCUS server. Figure 3.8a shows the resources used by the FOCUS server while we increase the number of nodes. The results highlight that the FOCUS server is not resource-hungry, even when there are more than 1.5K nodes that are part of the FOCUS system. Note that the VM on which the server ran had 4 virtual CPUs and 16GB RAM.

Overhead on Node Agents: Figure 3.8b shows the bandwidth consumption at one of the nodes in a p2p group under two conditions: normal operation (exchanging membership information) and query processing (1 query/s). Even though that in our earlier experiments the average size of a group did not exceed 150 members, it is worth measuring the impact of having groups with hundreds of members. The results in the figure suggest that during normal operation, the bandwidth consumed is negligible (under 2KBps), even for groups with more than 400 members. This shows that even when deployed in a cross-site setting, FOCUS does not impose significant overhead on the nodes. When processing queries every second, the bandwidth consumed by the node is less than 10KBps for groups with 100 nodes and about 50KBps for groups with 400 nodes. We note that FOCUS picks a random member every time it sends a query to a group;

hence, eliminating any excessive overhead on one specific node.

Latency vs. p2p Group Size: Since queries are gossiped by the p2p nodes, query processing times depend mainly on how fast a p2p group can converge. In this experiment, we measure the query latency with respect to the source of the query response from which it was served. In our design of FOCUS, query responses are either fetched from a local cache or pulled from the p2p groups. For responses from the p2p groups, we measured the start and end times at the p2p node that received the query. Figure 3.8c shows two key points. First, fetching responses from the cache (takes 45ms) significantly reduces the query processing time by an order of magnitude. Second, even when a p2p group contains hundreds of nodes, the response is still under one second. Note that we used the same gossip configuration discussed in section 3.8.2.

3.11 Discussion and Future Work

In this section, we discuss some of the open issues in FOCUS and potential future directions of work.

Deciding the Right Group Ranges: Determining the "right" group ranges (section 3.7) is critical for FOCUS's performance as biased groups could form and harm FOCUS's ability to efficiently answer queries. Our design allows system operators to configure group ranges, allowing them to use the method of their choice (static, random, heuristic, trace-driven, etc). In the future, we will augment FOCUS with a default mechanism driven by machine learning techniques (trained with traces) to decide appropriate group ranges.

Faster Query Processing: In our evaluation (Figure 3.7b), even though FOCUS scales better than RabbitMQ, FOCUS's query processing takes longer than what RabbitMQ takes for nodes less than 1200. This is mainly attributed to FOCUS's use of a small number of nodes as its "fanout" gossip factor (e.g., number of neighbors that a node directly talks to), leading to a slow convergence. Another alternative (and faster) approach is to broadcast the query to all members of the group by configuring the fanout factor to be N, where N is the number of nodes in the group. This is a trade-off between resource usage of a node (bandwidth consumption used to gossip with other members) and query processing latency. In the future, FOCUS can provide the option to configure each group's fanout factor, which when set to a high value, will be of great use for time-sensitive applications.

Further, as future work, we first wish to explore materialized views in FOCUS by creating specific p2p

groups representing frequently issued queries. We wish to extend this concept by supporting event triggers – change in node state will automatically update the materialized view. Finally, we also wish to provide translation/normalization functions to deal with the potential heterogeneity in data sources of FOCUS.

3.12 Conclusion

In this chapter, we address a fundamental problem in large-scale distributed systems – finding nodes that match certain criteria and present FOCUS, a novel scalable search service for finding nodes. This is a challenging problem because of the scale of the nodes and the highly dynamic nature of their attributes. Current approaches to this problem that typically involve nodes pushing status to a centralized database through a message queue simply do not scale. Naiive hierarchical push/pull solutions impose unsuitable trade-offs in accuracy of the results and overhead of node finding. On the other hand, FOCUS uses a novel hybrid approach in which we maintain p2p groups of nodes based on their attribute values or state. We illustrate FOCUS's broad applicability in real-world systems such as OpenStack and VNF homing in the Open Network Automation Platform (ONAP). Our evaluation confirms the superior scalability of FOCUS over existing approaches.

Chapter 4

StepNet: Incremental Approach to Homing Complex Network Services

Homing or placement of virtual network functions on cloud and network service provider (NSP) infrastructures is a crucial step in the orchestration of network services, involving complex interactions with the cloud, SDN and service controllers. Traditionally, homing involves a laborious off-line process where Network Service Providers (NSPs) hand-craft service-specific homing heuristics, and pre-provision resources based on expected service load. This service-specific approach does not scale well as more services are deployed, since different services have very different set of requirements or constraints. While pre-provisioning leads to conservative over-allocation of resources, repeated querying of the various controllers (e.g., to check customer eligibility or capacity) consumes significant amount of time and resources at the controllers. We replace this traditional homing process with *StepNet*, a compositional homing framework[1] , which allows service designers to mix and match constraints to construct instances of the homing problem simply and quickly, enabling greater agility of service creation and evolution. *StepNet* adopts an incremental approach to querying that provides near optimal homing solutions, while reducing the cumulative time spent by all of the data sources responding to queries for each homing request (query cost). Our evaluation with production traces from a Tier-1 NSP shows a reduction in query cost of 92% for over 50% of the requests.

In this chapter, we explicitly assume homing requests are handled sequentially and solve for the base problems: service evolvability and per-request query burdensome on controllers. In chapter 5, we relax that assumption and address the challenges introduced when we have distributed homing instances.

[1] *StepNet* is a homing service that is built on top of the Homing and Allocation service (HAS) of the Open Network Automation Platform (ONAP). As such, *StepNet* leverages many of the components that are part of HAS.

4.1 Introduction

Network Service Providers (NSPs) offer fundamental networking capabilities such as managed dedicated internet connectivity, Wide Area Networks, Virtual Private Networks, Voice Over IP, and Secure Cloud Connectivity. A critical step in provisioning these services for their customers is identifying either optimal locations for creating the network elements of the network service or reusing existing service instances (among several thousands providing the required capabilities) that can be shared among services. This process, referred to as the **Homing problem**[2] , is performed based on a wide variety of constraints influenced both by the customer Service Level Objectives (SLOs) [53] and the NSP such as network latency, bandwidth capacity, and infrastructure capabilities.

Figure 4.1 shows an overview of the topology and homing requirements of a virtual Customer Premise Equipment (vCPE) residential broadband service, a simple but illustrative real-world network service offered by many NSPs. vCPE connects a residence to the vG at the Service Provider Edge (PE). The Bridged Residential Gateway (BRG) is the vCPE located at the residential customer premises, while the vGMux is a shared network service at the PE that maps layer-2 traffic between a subscriber's BRG and its unique vG, ensuring traffic isolation between multiple customers. In the homing terminology, the vCPE comprises of two **demands** - vG and vGMux, and two types of **candidates** - (i) PE cloud sites (called 'cloud' candidates), where new vG instances can be created, and (ii) existing instances of the vGMux service (called 'service' candidates), which can be shared by the new vCPE instance with other subscribers. The goal for homing the vCPE service, is to drive the selection of a close PE site to host the vG for a given customer, where an existing shared vGMux provides the required cross-connect capability. Finally, the homing requirements for the vCPE service are defined through five different constraints as shown in the figure.

Traditionally, NSPs have viewed homing as a constraint-based mapping of **resources** to **requirements** that has been explored in works on virtual machine (VM) placement [43, 71, 82, 95, 120, 135], virtual network function (VNF) placement [49, 97, 119, 145], capacity planning [20], facility location [44, 45] and replica placement in datastores [72, 90]. However, our detailed analysis of the service requirements and

[2] Henceforth the word **Homing** refers to the Homing of network services

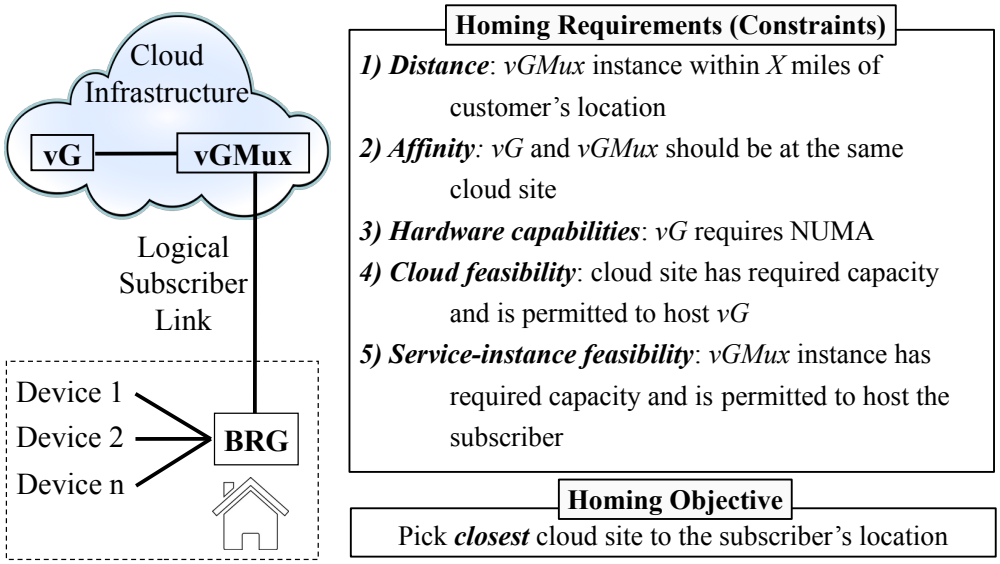


Figure 4.1: Homing a Residential Broadband service.

production homing traces of a Tier-1 NSP, reveal several challenges with this traditional approach as the number of network services and their operational complexity increases. In the next few paragraphs, we describe the two most significant challenges among them.

Evolving Service Requirements. Network services are complex with widely varying requirements that change as the service evolves. For example, the vCPE residential broadband service is typically offered with many pricing tiers, SLAs, and add-on features (e.g., added security) that alters the service requirements quite significantly. Developing hand-crafted heuristic optimization models for each service (and its variants) is a time-consuming and complex offline process with several cycles of testing and validation (order of months). Further, even simple updates to an existing service may often result in an entirely new formulation (e.g. linear to non-linear constraints) that will necessitate a significant amount of time before deployment. Clearly, this runs counter to the goal of evolving and maintaining systems in an agile manner.

Aggregating Data during deployment. The homing service requires extensive interactions with tens to hundreds of SDN, Cloud, and Network Service controllers in order to identify a feasible placement decision. This results in repeated queries to the controllers to check for run-time factors such as customer eligibility to use certain service instances or the availability of capacity in a certain cloud-site. Since these controllers are primarily responsible for running and managing the life-cycle of these network services,

NSPs rate-limit these repetitive homing related queries to the controllers, which limits the scale of the homing system as a whole. Our analysis of homing traces of a Tier-1 NSP showed that, if left unchecked, these queries would often exceed the allowed rate at the controllers (twice the rate for $\sim$50% of the time) and further, these controllers cumulatively spent more than 400 seconds per homing request to answer these queries for $\sim$50% of the homing requests. While pre-provisioning resources may help limit the need for such querying, in practice, this results in massive over-provisioning and requires repetition when the services evolve.

We address these challenges through a novel system, *StepNet* that is based on two key innovations. First, we introduce a novel compositional framework where homing instances of a service can be described through a declarative template that consists of **compositional blocks** through which a service designer can specify service requirements, including constraints and homing objectives. These abstractions have clearly defined structures, functional behavior, and APIs. These compositional blocks can be mixed and matched by service designers to create new homing requests with considerable ease as their customer requirements evolve.

Second, we introduce an online incremental approach to the homing problem which is designed to minimize the number of queries made to the controllers, while maintaining good solution quality. It does this through two key approaches: (i) rather than evaluating the entire solution space, we start with a small set of potential solutions, ordered based on the objective value, and gradually expand this set until a good solution is found, (ii) we sequentially evaluate the feasibility of constraints, performing the least expensive (in terms of queries) first, such that we can prune candidates before needing to evaluate the more expensive constraints.

We evaluate *StepNet* guided by production traces and 12 network services obtained from a Tier-1 NSP. Using the compositional framework, we generated over 1200 variants of these services supporting varied constraints and demands with just a few days of work. We demonstrate agile service evolution by supporting new run-time objectives and common heuristics for optimization with a few hundred lines of code. Next, we show that the incremental approach adopted by *StepNet*, reduces the cumulative time spent by all of the data sources responding to queries for each homing request (*i.e.,* query cost) by $>$ 1,000 seconds for 50% of

homing requests, and by $>$ 10,000 seconds for 20% of homing requests, while maintaining close to optimal solution quality.

In summary, this chapter makes the following contributions:

- Highlighting critical challenges in homing gleaned from our detailed analysis of the service requirements and the production homing traces of a Tier-1 NSP (§4.2).
- A novel system, called *StepNet*, that addresses these challenges through a compositional framework (§4.3)[3] for designing homing instances and an incremental approach to solving them (§4.4).
- Extensive evaluation using production traces, proving *StepNet*'s efficacy (§4.6).

4.2 Background and Motivation

Typically, the process of homing in NSPs involves two distinct phases - (1) an offline "design" phase when the network service-specific homing heuristics are built and an estimated set of resources are pre-provisioned for homing an anticipated number of service instances, and (2) a "run-time" phase where the actual homing decisions are made for each service instance as they are created by the service orchestrators. We now describe the current approaches adopted by NSPs for homing, highlighting the challenges observed with these approaches based on interactions and operational experience with a Tier-1 NSP.

4.2.1 Design Phase: Challenge of Evolving Services

In the offline or design phase of the homing process the service developer generates the service model, *i.e.,* service demands, constraints and objectives described in §4.1, and develops service-specific optimization models and heuristics for homing the service, drawing from several works on VM and VNF placement [43, 49, 55, 74, 82, 95, 97, 119, 120, 135, 145]. These works typically formulate the placement problem using integer linear programming (ILP) or mixed integer linear programming (MILP), and propose tailor-made heuristics to relax and solve the problem for a specific use-case (*e.g.,* 5G network slicing).

[3] We note that, *StepNet* is built on top of the homing and allocation service (HAS) of ONAP [101], which implements the compositional part of the framework. The compositional design is included in this chapter because of mutual authorship of the work that describes both *StepNet* and HAS.

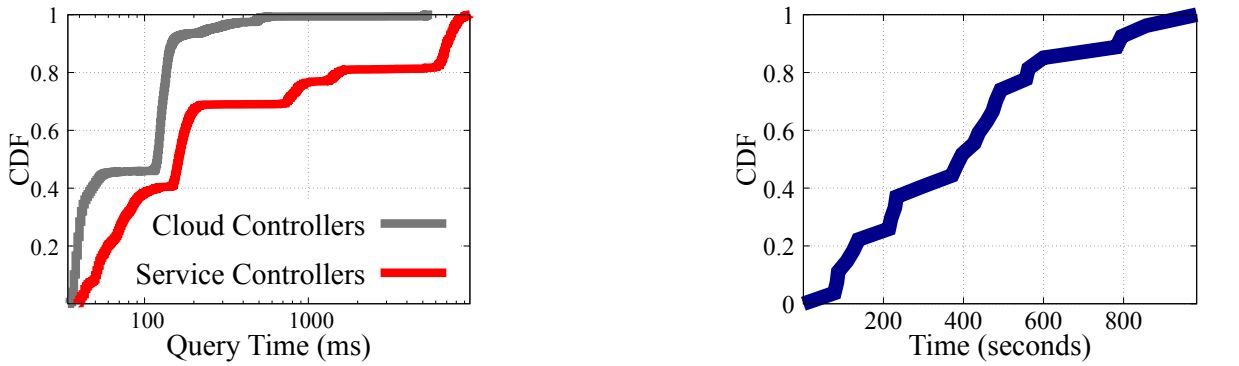
Our discussions with a Tier-1 NSP revealed that this approach of developing service-specific heuristics, validating and testing them before rolling them out to production, requires a significant Time To Market (TTM) (order of months) and resources. Further, services are often updated, and variants of them are created. For example, consider the case when a NSP wants to provide a new price-tiered offering or add a new firewall function for some subscribers. This requires additional constraints or demands that affect both the heuristic optimization models (*e.g.,* from a linear filtering constraint to a quadratic coupling constraint) and the underlying formulation of the homing heuristic. The new heuristic has to go through the entire cycle of testing and validation before production deployment, requiring a considerable amount of time. Clearly, the current approach compromises both evolvability and maintainability of network services, negating the benefits of virtualization.

Work in the peripheral space of optimizing software-defined networks seeks to simplify the optimization formulation process. For instance, SOL [58] and Chopin [57] provide a limited set of high-level APIs (*e.g.,* add link capacity constraint) to software-defined networking (SDN) applications to efficiently manage network resources. SDN applications, then, need to consume those APIs, and the framework will model those high-level API calls as LP/ILP programs, and then solve them to find the best solution. However, these works assume complete knowledge when performing the optimization, so the addition or change of one service would require re-doing the whole optimization. VNF placement approaches [49, 55, 74, 97, 119, 145] have the same disadvantage.

As evidenced above, the evolutionary nature of network services makes it impractical to design a specific model and heuristic for each possible service (and its variants). To address this problem, we present a compositional framework for homing in §4.3 where new **demands** and **constraints** can be added with little, or no development efforts, different heuristics can be utilized with no change to the service model, and service models can be added or changed without affecting existing services.

4.2.2 Run-time phase: Challenge of Aggregating Data

The online phase of homing begins when a new network service instance needs to be created by a service orchestrator, which invokes the homing service along with instance specific run-time inputs like



(a) Individual query latency by type of controller.

(b) Total Time spent by controllers per homing request.

Figure 4.2: Homing queries analysis for a week-long trace of a Tier-1 NSP.

Subscriber ID, Subscriber Location, *etc*. The homing service retrieves the pre-built models and heuristics from a model repository, and aggregates the data required by the models to run the heuristic on a solver like CPLEX [62]. The homing recommendations are then returned to the orchestration workflow that creates and configures these network elements based on these recommendations.

As we can see, a key aspect of the online phase is aggregating the data required for solving the homing heuristics. One approach used in practice by NSPs involves estimating the resources required for deploying an anticipated number of service instances and then reserving capacities for the tenant in the cloud. These pre-provisioned resources are inventoried in a centralized location, ready to be allocated through their respective Cloud and Service controllers when new service instances are created. This approach, however, suffers from two major problems: (i) reserving resources based on expected load across instances of the service often results in tremendous over-allocation, and (ii) as services evolve, this step needs to be performed repeatedly.

An alternative involves collecting the data on demand, which requires querying various **data-sources**, which include different inventories (for relatively static information), and multiple cloud and service controllers (for dynamic information). For example, consider the constraints for the vCPE service (Figure 4.1). Relatively static information such as **Distance**, **Affinity** and **Hardware capability** can be maintained in inventories/databases. However, the **cloud feasibility** constraint, evaluated at the cloud controller is inherently

dynamic, and requires run-time parameters like subscriber information (*e.g.,* customer-key) and service instance information (*e.g,* tenant-id) to evaluate whether a given cloud instance can support creating a new network element of the service for the subscriber. Similarly, the **service-instance feasibility** constraint, evaluated at the service controller, requires run-time querying to evaluate whether an existing vGMux service instance has sufficient capacity/resources, and authenticated to support the new vCPE service for the given subscriber.

These queries place significant burden on the cloud and service controllers, which are primarily responsible for life-cycle management of the cloud and service instances. For instance, a SDN Controller that manages flow control needs to perform topology and configuration operations like injecting routes into the network based on operator specified rules, besides performing periodic management and monitoring activities for the services (*e.g.,* BGP connections). These controllers, which are primarily designed for service life-cycle management as opposed to large-scale query processing, are easily over-burdened with a large number of repeated queries coming from the homing service. Hence, network operators typically rate-limit such queries coming from external services. Further, it is impractical for the NSP to re-design these controllers to process larger query volumes since they often depend on third party software. For example, the NSP we worked with built their SDN controller on top of the ODL controller [108].

To quantify the burden of queries at the controllers, we analyzed a week long trace of all homing related queries issued to all cloud and service controllers in a Tier-1 NSP. We make the following observations. First, the homing service would send queries at a rate at least twice the allowed rate for 22% and 44% of the time, to the least and most loaded controllers (*w.r.t.,* homing queries) respectively. To cope with this rate, the controllers queue those queries – resulting in unwanted delays. Second, both cloud and service controllers take a significant amount of time to process and respond to those queries. Figure 4.2a shows the CDF of the query latency observed at those controllers. As the graph shows, service controllers' query latency was more than 1 second for 20% of the queries. Third, as a result of both heavy query processing as well as query queuing, solving a single homing request takes a long time. Figure 4.2b shows the CDF of the total time spent by the controllers per homing request. The graph shows that about 50% of the homing requests required more than 400 seconds of controller's time across all controllers. With NSPs creating thousands

of service instances every day, querying the service and cloud controllers for every new service instance is prohibitively expensive and time-consuming.

We address this challenge through an incremental approach in §4.4, where we query instantaneous capacity and other required information during the run-time phase to avoid the wastage of pre-provisioning, but at the same time ensure a far reduced burden of querying at the controllers.

4.3 *StepNet* - A Compositional homing Framework

As described in §4.2.1, traditional homing approaches require significant changes in the underlying heuristic optimization models when the service composition or homing constraints evolve, because optimization models have tight dependencies across the demands, constraints and objective functions. In this section, we describe how we address this challenge through our novel compositional homing framework, *StepNet*. Through this framework, homing instances of a service can be described through a declarative template that consists of **compositional blocks** which are abstractions that specify demands, candidates, constraints, objective functions, data-sources and heuristic optimization algorithms (terms defined in §4.1, §4.2). These abstractions have clearly defined structures, functional behavior and APIs. Our notion of **composability** is that, as long as homing instances are created by mixing and matching these compositional blocks, our incremental approach (see §4.4) can provide a solution. This enables service designers to create new homing requests with considerable ease as their customer requirements evolve.

Standardized Compositional Behavior. The composition blocks, specifically the constraints and objective functions, have standardized interfaces and pre-established functional behaviors. For instance, all constraints are designed as plugins exposing a common interface, `solve(homing-context, candidate-set, data-sources)`, where the homing-context is an object that captures the current state of the homing request being processed including the demands, constraints, and input parameters. The candidate-set is an input set of candidates which are found feasible until the point when the current constraint is invoked. The data-sources specify what data sources need to be queried to collect information required to evaluate the constraint. All constraints exhibit a consistent filtering behavior, which eliminates zero or more candidates that do not meet the constraint's requirements, and returns a subset (not necessarily a strict subset) of the in-

put candidate-set. Similarly, all objective functions expose the interface `compute(optimization-goal, normalization-function, cost-function)`. The optimization-goal can be to minimize/maximize, while the cost-function can represent different metrics like latency, utilization, dollar costs, *etc*, while the normalization-function allows joint optimization with multiple metrics like latency and utilization by normalizing these values.

Library of Composition Blocks. We distill our detailed study of production services to create a library of composition blocks [102], through which a service designer can obtain common demands, constraints and objective functions. For example, our library contains the candidate types of 'cloud-region' and 'service-slice' and the associated inventories. The service designer simply needs to mention the VNFs that correspond to these demands. Similarly, the library also contains the constraints for zone and capacity and only the instance specific details like the demands and the actual bandwidth (among others) need to be specified. The library elements for the objective function and heuristics operate in a similar way.

Homing Template. To enable service designers to specify various compositional blocks, we provide a declarative template inspired by OpenStack's Heat template [114], appropriately extended to support the composition blocks described below. Listing 1 shows the structure of a homing specification for the example of the vCPE service described in Figure 4.1. The `RUN-TIME PARAMETERS` block captures the run-time inputs required for homing, like subscriber information and authentication keys that are typically used to evaluate the constraints (*e.g.,* vpn-key is used in cons-C). The `DEMANDS` block represents the network elements of the service, the candidate types for each of these elements and the inventory source from which these candidates can be drawn. The `CONSTRAINTS` block lists the constraints, their parameters and the specific **demands** to which the constraint applies. Some constraints are pertinent to a specific demand (*e.g.,* bandwidth capacity required by a VNF), while some constraints span multiple demands (*e.g.,* distance between two VNFs of the service). The `OBJECTIVE-FUNCTION` block specifies the target metric optimization like dollar costs, latency, *etc*. Except for the `DEMANDS` block, the other composition blocks are optional. For instance, some real-world network services do not have any hard constraints, but require optimality of an objective function metric. Finally, the heuristic algorithm that invokes these constraints and objective functions to meet the service homing requirements, can be selected as a part of the composition. For instance,

Listing 1 Structure of a Homing template instance

```
RUN-TIME PARAMETERS:
    instance-id    : 'id of current service instance being created'
    heuristic      : 'Best-fit'
    customer-id    : 'id'
    vpn-key        : 'vpn-k1'
DEMANDS:
    VNF-X:
        cand-type  : 'cloud-region'
        inventory  : 'cloud-controller'
    VNF-Y:
        cand-type  : 'service-slice'
        inventory  : 'network-srvc-provider'
CONSTRAINTS:
    cons-Z:
        demands    : [VNF-X, VNF-Y]
        type       : 'zone'
        zone       : 'cloud-region'
        qualifier  : 'same'
    cons-C:
        demands    : [VNF-Y]
        type       : 'capacity'
        resources  : {'BW'   :   '1Gbps'}
        subscriber-info: {'id'   :   customer-id,
                          'key'  :   vpn-key}
OBJECTIVE-FUNCTION:
    f1:
        f-type   :        'cost'
        demands  :        [VNF-X, VNF-Y]
```

the same homing template instance shown in Listing 1 can be used with either a heuristic best-fit search algorithm or a shortest-path search algorithm[4] .

To illustrate how our compositional framework allows service evolution, consider the example of vCPE service (Figure 4.1) providing an added Firewall (vF) function such that the cloud site hosting the vF has sufficient capacity and the vF is co-located with the vG. The vF would be a new **demand** in the `DEMANDS` block, and the co-location requirement for the vF can be easily added by modifying the **Affinity** constraint to include the vF along with the vG and vGMux. The capacity requirement would be a new **cloud feasibility** constraint added to the `CONSTRAINTS` block. This compositional framework is able to support over 1000 instances of production services (see §4.6) with varying demands and constraints ranging from traditional use-cases like Wide Area Networks and Private Virtual Networks and even futuristic network services such as 5G Network Slicing [27].

4.4 Incremental Approach to homing

As described in §4.2.2, traditional homing approaches place substantial burden on the controllers that need to be queried for *run-time* information as part of the homing process. We propose an incremental approach that minimizes the queries made to the controllers while maintaining a reasonable level of solution quality, by leveraging two independent dimensions:

- **Objective-based candidate ranking:** we limit the number of candidates against which the constraints shall be evaluated by incrementally increasing the set of candidates used until a quality solution is found (*i.e.,* we incrementally explore the overall search space of candidates).
- **Cost-based ordered constraint evaluation:** we order constraints such that the least expensive constraints are evaluated first. In doing so, we can more quickly prune out candidates that are not feasible, and only evaluate the most expensive constraints when we know the rest of the constraints are feasible.

In this section, we first describe an overview of the incremental approach (4.4.1), and then we expand on the

[4] This template serves as an example and is not intended to be an exhaustive description of all possible composition blocks.

Algorithm 6 Incremental Algorithm

Input:

r : instance of a homing request consisting of: demands (VNFs), constraints, objective function(s), and a set of initial candidates for each demand
p : instance of a heuristic algorithm (e.g., best-fit, exhaustive)
$step$: incremental step (candidate subset) size
tol_thresh : local solution tolerance threshold

```
1:  procedure FINDSOLUTION(r, p, step, tol_thresh)
2:      RANKCANDIDATES(r)
3:      best = null                                   ▷ keep track of best solution
4:      tolerance = 0                                 ▷ used for stopping_condition
5:      Start with empty available-candidates for each demand
6:      while True do
7:          if stopping_criteria then
8:              break
9:          for each d ∈ r.demands do
10:             increment available-candidates[d] by step
11:         sol = p.solution(r, available-candidates)   ▷ triggers constraints evaluation
12:         if sol = null then
13:             step = step * 2
14:             go to 6
15:         step = original size
16:         if first solution or sol is better than best then
17:             best ← sol
18:             tolerance = 0
19:         else                                      ▷ sol did not improve solution quality
20:             tolerance ++
21:     return best
```

two key concepts that enable it: objective-based candidate ranking (4.4.2) and cost-based ordered constraint evaluation (4.4.3).

4.4.1 Incremental Approach Overview

The incremental approach, in a nutshell, iteratively and carefully expands the search space until it satisfies certain stopping criteria. We highlight our incremental approach in Algorithm 6. The main procedure, FINDSOLUTION, is called for each homing request (an instance of the template presented in Listing 1) and should return a solution (if any). In order to obtain the initial set of potential candidates[5] to pass into FIND-

[5] This step takes place before the the incremental algorithm is invoked.

SOLUTION for each of the demands, *StepNet* queries multiple data-sources, including the NSP inventory and cloud controllers to determine what candidates can be used by each demand. We treat this as a fixed (minimal) cost for each homing request. To enable the incremental approach, we first rank the candidates (§4.4.2) for each of the demands (line 2) in a way that favors "good" candidates that have a higher chance of yielding quality solutions. The second step, which is triggered by the `solution` call in line 11, is to evaluate the constraints in a cost-based order (§4.4.3), from most to least expensive, to enable early elimination of candidates.

Incremental Search Space Exploration. For each demand, we start with an empty set of *available-candidates* (line 5) – the set of candidates that should be ready for constraint evaluation. In each iteration, then, we increment this set by a $step$ (line 10), which is set to some percentage of the whole set of initial candidates (*e.g.,* 2%).

We note that in each iteration, we include the union set of *available-candidates* of all previous iterations plus the current one such that $\mathsf{C}_{(i,d)} = \mathsf{C}_{(i-1,d)} \bigcup \mathsf{C}'_d$ where $\mathsf{C}_{\mathsf{i,d}}$ represents the *available-candidates* set for demand d in iteration i and C'_d represents the new candidates added in iteration i. This is necessary since we cannot discard candidates from previous iterations as it will break dependencies imposed by pairwise constraints. Such constraints could force us to choose one candidate for demand X in one iteration, and choose another candidate for demand Y in a different iteration.

Optimizer Independence of the Incremental Approach. After populating the *available-candidates* set for each of the demands in r, we pass r and *available-candidates* to the optimizer instance (line 11) where it evaluates the constraints for those *available-candidates*. By making the optimizer agnostic to how candidates are made available, the network operator can plug in any optimization algorithm (*i.e.,* optimizer instance) without modifying the logic of how candidates should be added to the *available-candidates* set. All the optimizer does is evaluate the constraints for *available-candidates* (triggered by the `solution` call on line 11), and according to its logic, decide what candidates to choose for which demand.

Adaptive Steps. Assigning a small value to the step size ($step$) helps reduce the number of queries to be sent by reducing the number of candidates against which the constraints are evaluated. However, when there are no feasible candidates in the first few iterations, the incremental algorithm can take a long time to

find the first solution. To avoid this, we make the $step$ adaptive such that each time the optimizer is not able to find a solution (line 12), we double the $step$ size (line 13). When *any* solution is found, we restore the original $step$.

Terminating the Incremental Loop. After getting a solution sol (line 11), the incremental algorithm evaluates it, and decides whether to accept it (lines 17-18) or to tolerate it by incrementing the $tolerance$ counter (line 20 – when sol is not "better" than the best solution so far, by comparing their objective values). The incremental algorithm terminates when certain $stopping\_criteria$ are met such as when $tolerance$ exceeds tol_thresh, a tolerance threshold that can be specified by the operator. Another metric that can be incorporated into the stopping criteria is an upper limit on the number of queries that can be made. We describe our stopping criteria in §4.6.

4.4.2 Objective-based Candidate Ranking

Since we incrementally increase the set of candidates available to be evaluated until we find a good solution (calculated on line 11 in Algorithm 6), we can reduce querying if we can find a good solution in as few iterations as possible. This requires that the candidates in early iterations are "good" candidates, so we do not have to proceed to later iterations. To rank the candidates, we leverage the objective function in the following manner. First, we construct a tree where each level in the tree consists of the lists of candidates for a given demand (where each demand is its own level). Each node is connected to the nodes in the next level down (for the next demand), with an edge weight that is set to the added cost of including the given candidate for that demand in the solution according to the objective function. For demands that are not part of the objective function, edge weights are all set to the same value for that demand. A solution path, then, is one candidate from each level in the tree. The rank of the candidate is based on the best solution-path value among each of the solutions paths that cover that candidate.

While the objective function does not indicate whether the candidate is a feasible candidate, it does indicate whether it could be a **good** candidate. We use this to determine how to incrementally increase the set of candidates used in a given iteration. This, in turn, reduces the number of candidates we need to test for feasibility, which is the more expensive part of homing. Interestingly, limiting the set of candidates to only

those with the best objective values can actually improve the solution quality of the optimization heuristic (see §4.6), because the incremental approach increases the chance of selecting a better local optimum.

Note that, objective-based candidate ranking in itself does not incur much query overhead since most services that we found in the Tier-1 NSP do not optimize (*i.e.,* their objective functions) for *run-time* information.

4.4.3 Cost-based Ordered Constraint Evaluation

We choose to evaluate constraints in a sequential, ordered manner to eliminate additional queries for candidates once we know they are infeasible. Unlike *static* constraints (location proximity, zone, *etc*), evaluating *run-time* constraints (instance-feasibility, network latency, *etc*) against a set of candidates triggers queries to be sent to collect *run-time* information for those candidates. Clearly, *static* constraints are the least expensive to evaluate since they do not require queries, and evaluating them first helps prune the set of candidates before getting to evaluate *run-time*, and therefore more expensive constraints.

When the *static* constraints indicate that a candidate is still feasible, we need to start evaluating the *run-time* constraints. In §4.2.2 we saw that different queries have different costs, and as such, we still would like to order the constraints. Determining how expensive a *run-time* constraint is to evaluate cannot be directly derived from the constraint properties. So, instead, we propose using past query logs to help calculate a real-time cost for each of the constraints that *StepNet* supports. We assign the rank (or cost) of a constraint as follows: $\mathsf{C}(cons) = l * f$, where l and f are the median latency and frequency of queries triggered by constraint $cons$ in past logs within a recent time window, respectively. Performing this offline task periodically allows our constraint ranking function to adapt to continuous changes in the network as well as adapt to new constraint types. To bootstrap the process when logs are not available, we assign equal ranks to *run-time* constraints, and after solving a number of homing requests, we re-evaluate these ranks.

4.5 Implementation

Figure 4.3 shows an overview of our implementation of *StepNet*, marked with path of a homing request (see Listing 1) as it traverses through the different components of *StepNet*. The homing request (1)

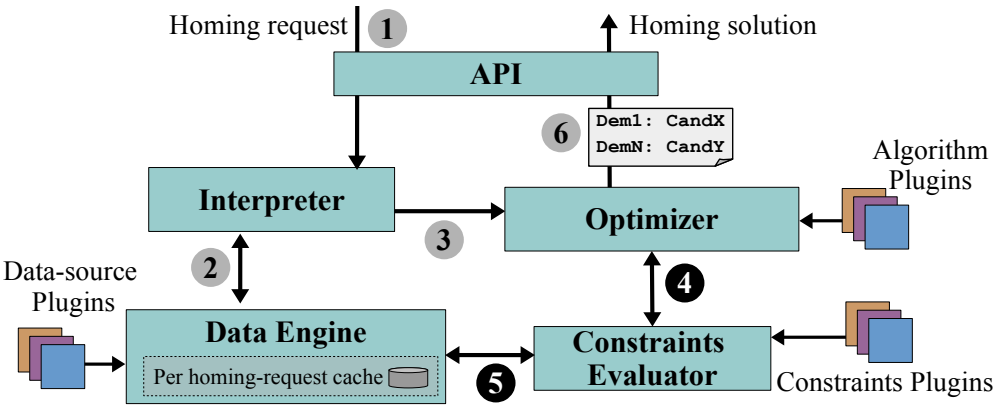


Figure 4.3: High-level view of the implementation of *StepNet*. The steps in black represent our incremental approach.

arrives at the homing API component, which performs basic validation of the inputs. The *Homing Interpreter* component, interprets the homing requests and associates the constraints (both per-demand and cross-demand constraints) to the corresponding demands on which they are applied. This step 2 also includes querying the **data-sources** through the *Data Engine*, to identify a potential set of **candidates** corresponding to each **demand**. We note that querying the universe of candidates is a standard look up type query, which is not expensive – hence, this part is excluded from our measurements in the evaluation.

At this point, the homing request is ready to be processed 3 by an optimizer for identifying an optimal **candidate** for each **demand**. The *Homing Optimizer* component executes the optimization algorithm on the interpreted homing request, and 4 evaluates the constraints as a part of this process guided by the algorithm. Note that evaluating the constraints typically require multiple queries being issued to the **data-sources** through the *Data Engine* 5. Once the "optimal" set of candidates are identified by the *Homing Optimizer*, the solution is sent back 6 to the *Homing API*, which responds back with the homing recommendations.

The constraints, algorithms, and data-sources are all implemented as plugins, and can be composed as presented in our design §4.3. We implemented *StepNet* in Python with about 10K lines of code. Our current implementation includes plugins for 10 constraints, 3 algorithms, 3 data source types, and 7 objective functions. This implementation of *StepNet* is now being deployed in the production network of a Tier-1 NSP.

4.6 Evaluation

In this section, we validate our key contributions of the compositional homing framework (§4.3), and the incremental approach (§4.4) by answering these two key questions: (i) can we easily compose homing requests for various network services using *StepNet*'s *compositional blocks*? And (ii) does the incremental approach significantly reduce the query cost while retaining quality of solution?

4.6.1 Experimental Setup

Trace-driven Evaluation. We collected 7 consecutive days of longitudinal homing traces from the production logs of a tier-1 NSP, which includes the homing requests, query parameters, query responses and latency observed for the queries sent to the different controllers. We used data collected from the trace to comprehensively emulate various aspects of a homing service running in production by following the same distribution found in the trace for the following aspects: feasibility/capacity responses (*i.e.,* whether a cloud site has enough capacity to home a given demand), query latency, geographical location density (*i.e.,* how many cloud sites per country). With our emulation framework, we emulate a total of 2400 cloud sites and service instances (*i.e.,* potential candidates), enabling us to evaluate *StepNet* at a large scale.

4.6.2 Supporting Flexible Service Composition

The traditional approach to homing, where new optimization models and heuristics are designed for each new service offering, incurs significant effort that runs counter to the requirements of maintainability and evolvability of services in NSP infrastructures (§4.2). One of the main goals of *StepNet* is to easily accommodate new network services by allowing service designers to compose homing requests for such services by mixing and matching different constraints and objective functions through our compositional framework.

Easy Service Composition. To demonstrate *StepNet*'s ability to accommodate various services, we obtained 12 service models from a tier-1 NSP, ranging from simple ones (1 demand, 1 constraint) to more complex ones (6 demands, 42 constraints). In a few days, the service designers could generate the homing

requests for those services using the template in Listing 1. Further, for each of the 12 services, then, we generate 100 instances that have different *run-time* parameters (required capacity, customer location, *etc*), while ensuring that we retain the same *service model* including demands, constraints, and constraint types as observed in production. This enabled us to create 1200 homing request instances with which we can test the incremental approach.

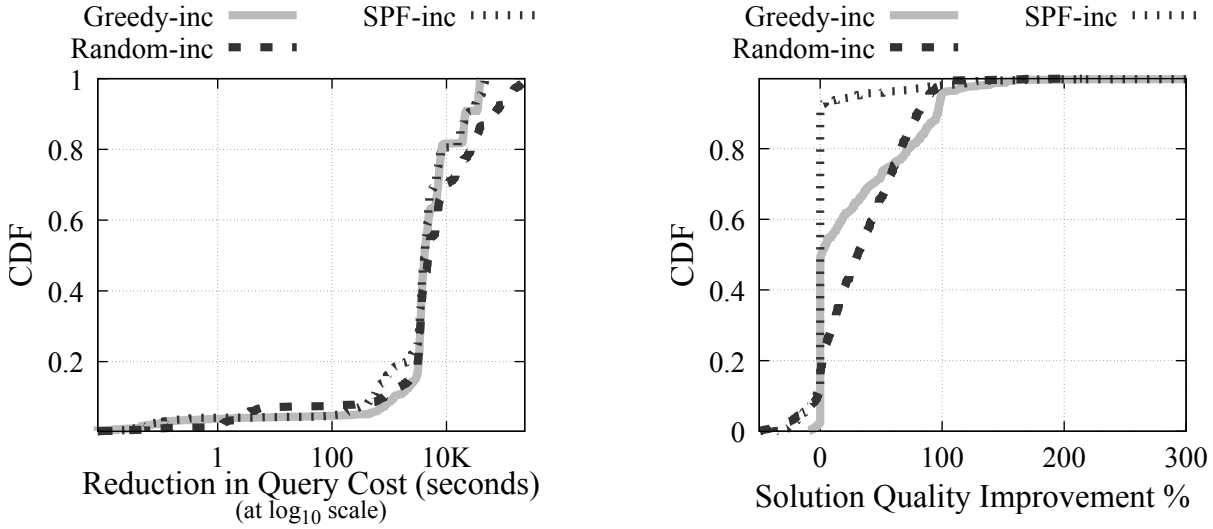
Supporting Service Evolution. Our base implementation of *StepNet* could support all the constraints required for the 12 services. However, a *run-time* objective function (minimize resource utilization) required by one of the services was not originally part of our implementation. With only 78 lines of code, we were able to implement this function along with the necessary data hooks to query cloud controllers for resource utilization information. This highlights the ease with which *StepNet* can adapt to evolving service/business requirements.

Supporting Pluggable Optimization Heuristics. Since our incremental approach is decoupled from the specific optimization heuristic, contrary to existing solutions that design a service-specific heuristic for each use-case/service, we can plug-in a variety of optimization heuristics through our compositional framework (line 11 in Algorithm 6). To demonstrate this aspect, we wrote two basic heuristic algorithms: *random* and *shortest-path first* (**SPF**), as well as a state-of-the-art optimization heuristic: backtracking best-fit (**BACBF**).

The **Random** heuristic randomly assigns a feasible *candidate* to a given *demand*. The **SPF** heuristic, on the other hand, is more comprehensive. It exhaustively identifies the best candidate-demand mappings that optimize the objective value based on an implementation of Dijkstra's algorithm [33]. The **BACBF** heuristic is a modified version of the best-fit optimization heuristic augmented with a backtracking ability. Best-fit has proven very effective in recent placement optimization studies [34, 50, 145]. For details on how we implement each of these optimization heuristics we refer the reader to [4].

4.6.3 Reducing Query Cost with the Incremental Approach

We evaluate *StepNet*'s incremental approach to answer the following questions: (1) how much cost is the incremental approach able to reduce? (2) does the incremental approach impact solution quality (*i.e.,*



(a) Reduction in **query cost**

(b) Improvement in **solution quality**

Figure 4.4: Results of running 1200 homing requests. This shows that the incremental approach not only reduces **query cost** (left) for all three heuristics, but it does so while improving **solution quality** (right).

objective value)? (3) is there a benefit to objective-based candidate ranking? and (4) is there a benefit to cost-based constraint ordering?

To answer these questions, we ran a set of experiments evaluating *StepNet* on CloudLab [36]. We use the three heuristics described in §4.6.2 with two configurations: with the incremental approach described in §4.4 (where we let the incremental approach decide what candidates the heuristic should see – denoted with **inc**) and without it (where we pass all potential candidates to the heuristic – denoted with **all**). Doing so yields six heuristic variants: **Random**-all, **Random**-inc, **SPF**-all, **SPF**-inc, **BACBF**-all, and **BACBF**-inc. Since the **SPF**-all configuration is the most comprehensive, we treat its solution for a given homing request as a baseline to measure how good the solutions of other heuristic configurations are. Moreover, we timeout **SPF**-all (and for that matter all heuristics) after 30 minutes of processing time, and use the best solution it could calculate at that point.

Incremental algorithm parameters. For the incremental versions of the heuristics, we set the step size ($step$) to 2% of the initial candidate set (while enabling adaptive steps), and we used a minimum of 5% improvement of any two successive solutions as our tolerance threshold (tol_thresh). We also used a

relaxed $stopping\_criteria$ throughout our experiments that terminates the incremental algorithm if either: $tolerance > 1$, processing timeout (30 minutes) is reached, or the total number of queries exceeds 2K queries. We generated these parameters by experimenting with a range of values and selected ones that yielded the best outcome. We note that we enable ordered constraint ranking for both incremental and non-incremental versions in our evaluation.

How well does the incremental approach work?

We ran experiments with the 1200 homing requests, that we generated for the 12 services described earlier, through all six configurations described above. For each homing request, we measure two key metrics: ***query cost*** and ***solution quality***. The *query cost* is the cumulative time spent by all data sources in responding to queries issued for a given homing request, which represents the "load" placed on the corresponding controllers. The *solution quality* is a percentage value that shows how close a given heuristic's objective value to the baseline's (**SPF**-all) objective value. A value greater than 100% indicates that a solution is better than the baseline.

Does the incremental approach reduce *query cost*? Figure 4.4a shows reduction in **query cost** with the incremental approach over the non-incremental approach for all the heuristics. The X-Axis shows the reduction in **query cost**, while the Y-Axis shows the CDF of the fraction of 1200 homing requests. From the graph, we make several observations. First, the incremental approach reduces the **query cost** across all three heuristics, highlighting the benefits of our incremental approach. Next, the incremental approach was able to reduce the **query cost** when compared to the non-incremental heuristics by more than 1K seconds for about 80% of homing requests (and by more than 10K seconds for 20% of homing requests). Finally, the incremental approach was able to (not shown in Figure 4.4a) reduce 60% of the **query cost** for 80% (and 95% for 20%) of the homing requests.

Does the incremental approach impact *solution quality*? Figure 4.4b shows the CDF of the percentage improvement in **solution quality** with the incremental approach over their non-incremental counterparts for all the three heuristics. The incremental approach improves the **solution quality** for the majority

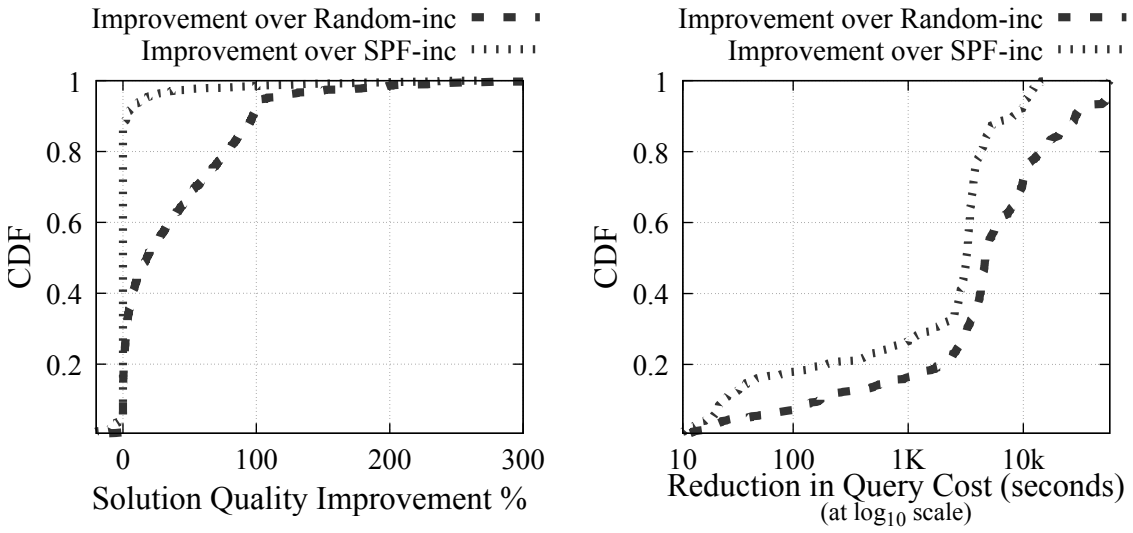
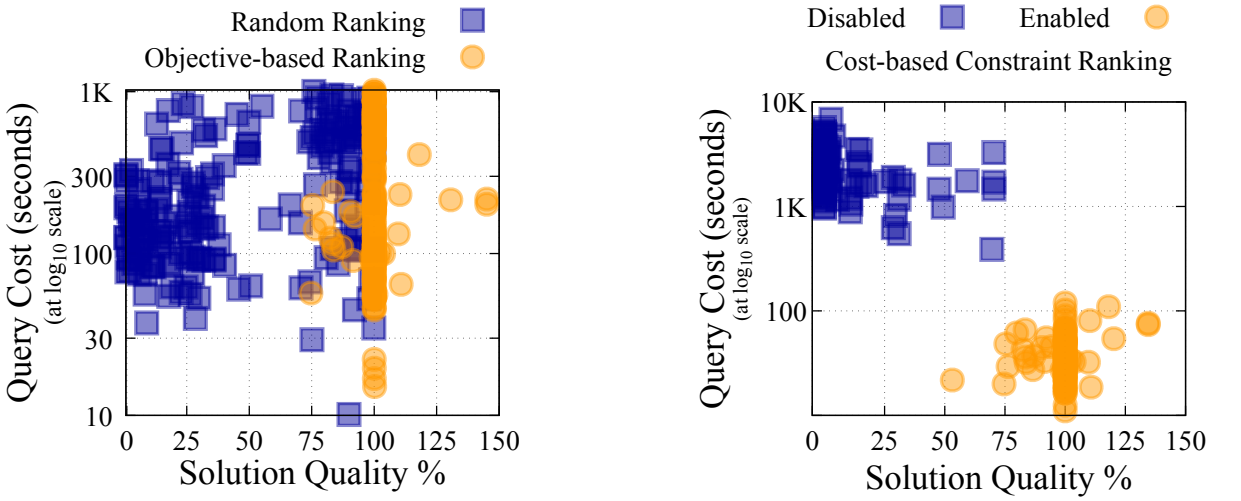


Figure 4.5: **BACBF**-inc provides the highest **solution quality** (left) and lowest **query cost** (right) when compared to **Random**-inc and **SPF**-inc.

of cases for the **BACBF** (50%) and **Random** (70%) heuristics, and provides comparable **solution quality** for the **SPF** heuristic. The benefits primarily come from the incremental algorithm's ability to intelligently rank candidates by favoring those that could lower the objective value (for minimizing objective functions) while limiting the search space. This, in turn, helps the heuristics terminate at a better local optimum. For a small fraction of homing requests, the incremental approach degrades the **solution quality** by at most 43% due to its limiting the candidate search space. We argue that this is a reasonable trade-off especially when considering the substantial reductions in **query cost**.

Comparing the different heuristics. While *StepNet* is designed to accommodate different heuristic algorithms, we now compare the three heuristics that we use in our evaluation. Figure 4.5a, shows the CDF of the percentage improvement in **solution quality** with **BACBF**-inc over **Random**-inc and **SPF**-inc, while Figure 4.5b shows the corresponding reduction in **query cost**. As observed from the figures, **BACBF**-inc provides substantial **query cost** reductions when compared to the other two heuristics. Interestingly, **BACBF**-inc was able to maintain the **solution quality** (compared to **SPF**-inc) or provide a higher quality solutions (compared to **Random**-inc) for a vast majority of homing requests. For less than 5% of requests,



(a) Objective-based candidate ranking vs. random ranking

(b) Enabling/disabling cost-based constraint ranking

Figure 4.6: Key features of the incremental approach (using **BACBF**-inc): objective-based candidate ranking (a) and cost-based constraint ranking (b), provide significant benefit to **solution quality** and **query cost**.

BACBF-inc yielded lower quality solutions (41%), when compared to **SPF**-inc[6] .

Objective-based vs. random candidate ranking. We now compare **BACBF**-inc for 100 homing requests for the same service with two configurations of candidate ranking: (1) random candidate ranking, and (2) objective-based candidate ranking. Figure 4.6a shows a scatter plot with the **solution quality** along the X-Axis and **query cost** along the Y-Axis. A value greater than 100% for the **solution quality** indicates a better solution, while a value less than 100% indicates a degradation in the **solution quality**. As observed from the graph, while objective-based ranking does not provide significant value in terms of **query cost**, it provides significantly better (3x for 50% of requests) **solution quality** (X-Axis).

Benefits of cost-based constraint ordering. Finally, we run 100 homing requests through two configurations of the **BACBF**-inc heuristic: (i) with the cost-based constraint ranking and (ii) without constraint ranking. We select one of the 12 services that had the most number of constraints – 42 constraints spread across 6 demands of the service. To highlight the penalty of incorrect ranking, we reversed the ordering of constraints for the latter configuration (ii). Doing so provides us with a lower bound for parallel constraint

[6] Since the **BACBF**-inc heuristic clearly performs the best, henceforth, for simplicity and clarity we only present results highlighting its performance. The trends for others were similar.

evaluation (*i.e.,* the same candidate set is evaluated by all constraints).

Figure 4.6b shows a scatter plot with the **query cost** along the Y-Axis and **solution quality** along the X-Axis. We observe that the **query cost** increases by at least one order of magnitude, when cost-based constraint ranking is disabled. Further, this leads to significant degradation in **solution quality** (higher objective values). In theory, constraint-ordering should not affect the final outcome of an algorithm and hence the quality of the solution. However, in our experiments, evaluating query-intensive constraints (*i.e.,* with cost-based ranking disabled) in the initial stages of constraint evaluation causes the incremental algorithm to hit the time-out limit of 30 minutes.

4.7 Conclusion

Homing of VNFs is a crucial component of the life-cycle management of complex network services provided by NSPs. Traditional approaches to homing are often service-specific and do not easily accommodate service evolution, which critically affects the maintainability of services. Further, a homing service that provides homing recommendation for thousands of service instances each day requires limiting repetitive and resource intensive queries to downstream data sources. Our compositional homing framework, *StepNet*, was designed to solve these challenges and caters to a growing number of services. Our results show that our incremental approach is able to provide good quality solutions, while reducing query costs by 92% for half of the homing requests when compared to non-incremental approaches.

Chapter 5

Accounting for Dependencies between Parallel Homing Requests with *StepNet*+

As has been demonstrated so far, homing network services is a query-intensive and lengthy process. To cope with increasing demand, NSPs deploy the homing service such that multiple instances of the service are running simultaneously to increase the throughput of the overall system by processing as many requests in parallel as possible. In chapter 4, we addressed the base challenges of service evolvability and per-request query burdensome on controllers while assuming homing requests are processed sequentially. In this chapter, we extend our solution in chapter 4, and relax this assumption to address dependency problems resulting from deploying distributed instances of the homing service. That is, dependencies between homing requests raise new challenges, including: query redundancy, resource contention, and unoptimized resource sharing. In this chapter, we shed light on these problems that are a direct result of dependencies across homing requests, describe why current approaches do not suffice, and extend our *StepNet* design to address those problems. To this end, we present *StepNet*+, an extended homing service that can: (*i*) eliminate up to 70% of queries across multiple homing requests, (*ii*) reduce chances of resource contention, and (*iii*) optimizes resource sharing by up to 45% when compared to the baseline approach.

5.1 Introduction

Homing complex network services often involves issuing numerous queries – making the process of serving each homing request very lengthy and time consuming (on the order of several minutes – see Figure 4.2b). To cope with high demand (thousands of service requests/day), NSP operators often deploy the homing service (§4.4) as a replicated service with multiple instances. In such setting, the homing service is

deployed in containers that can be scaled in and out. The main benefit of using such deployment is that when demand (*e.g.,* number of homing requests arriving at the service) increases, the operators can seamlessly instantiate new instances of the service. When deployed, each homing service instance processes a different homing request to ensure isolation between different running instances.

Although such deployment works well for the purpose of increasing the homing service's throughput, a number of new problems arise from processing homing requests in parallel. In this chapter, we focus on three challenges that stem from deploying parallel instances of the homing service.

Redundant queries. As we have motivated in §4.2, the number of queries should be reduced as much as possible to alleviate load on the service and cloud controllers. However, when homing requests are processed in parallel, the homing service could collectively issue redundant queries that otherwise should have been avoided. Although our incremental approach (§4.4) significantly reduces query cost, it does so for a per-request basis (as opposed to across multiple requests).

To address this, we extend our design of *StepNet* (dubbed *StepNet+*) to eliminate as many query redundancies as possible. We achieve this by batching homing requests based on how much query overlap they have, and use a logically centralized cross-instance query cache that all homing service instances check before actually sending those queries. Our design ensures that no queries will be sent more than once for each batch of homing requests.

Resource contention. When multiple homing requests are processed simultaneously, there can be an overlap between the resources that they select to home their demands. If not handled, this can result in resource contention. That is, when two or more homing requests select the same resource (cloud or shared service instances) to home their demands, that resource might not be able to serve *all* those requests. This results from the fact that when a homing service instance queries the corresponding controllers to check capacity/eligibility, it gets a green light to use that resource. However, when other *parallel* homing instances query the same resource, they will get the same response. Through our discussions with a tier-1 NSP, we learned that this problem occurs oftentimes in practice. This leads to failing to provision resources for many homing requests, causing those failed requests to be re-solved again – leading to significant delays and wasted resources.

To address this, we augment our homing service with a mechanism that uses past query traces to *"predict"* whether a given resource can accommodate more than one demand. If we predict that a given resource is not able to serve extra demands, then any other requests interested in this resource will not be allowed to select it. Although this prediction mechanism is not guaranteed to be correct all the time, it does provide a *"best-effort"* solution that can mitigate the problem.

Unoptimized resource sharing. To utilize their infrastructures, NSPs offer shared network services (VNFs) to their customers. Such shared VNFs provide connectivity, among other functionalities, while ensuring isolation between multiple customers. When the homing service receives a homing request (r) that specifies its need of a shared resource (*e.g.,* cloud gateway, firewall, etc), the homing service will look for existing instances that can satisfy r's constraints. If it does not find any existing instances, then it will recommend creating a new instance at a cloud site that optimizes r's objective function (*e.g.,* close proximity to customer location). This can quickly result in creating redundant shared resources. Due to the provisioning cost of those shared resources as well as resource utilization, NSPs try to avoid creating new instances of those shared resources unless necessary.

To address this, we make homing service instances more cooperative with respect to how they select shared resources. This entails that we modify how those instances assign resources to the demands of the request they are solving. This enables us to consolidate as many shared resources as possible, resulting in significant provisioning cost savings (up to 45% reduction).

In summary, we make the following contributions in this chapter.

- We present an analysis of different problems and inefficiencies that stem from mishandling dependencies between homing services (§5.2).
- We design and evaluate *StepNet+*, an extended homing service that efficiently handles homing requests dependencies (§5.3).

5.2 Background and Motivation

In our design of *StepNet* in Chapter 4, we proposed an incremental approach to homing such that we can significantly reduce the load on controllers that the homing service incurs when processing homing requests. As presented in Chapter 4, *StepNet* treats homing requests on an individual basis. This works well if the homing service processes requests sequentially such that the results of one homing request are accounted for when processing subsequent requests (*e.g.,* newly created shared firewall instance can be reused by future requests). Due to the high load of homing requests that NSPs receive (on the order of a few thousands per day), coupled with the fact that homing is a lengthy and time-consuming process, NSP operators typically deploy multiple instances of the homing optimization service to increase its throughput. Each parallel homing service instance, then, processes a different request to ensure isolation between those parallel instances.

There are three major problems that result from this parallel deployment. First, redundant queries can be issued by those parallel homing service instances – something to be avoided to reduce the load on controllers even further. Second, parallel homing service instances could compete for the same set of underlying resources, causing resource contention. This results in re-solving those conflicting requests – causing delays and wasted resources. Third, when parallel homing instances process requests that are asking for some type of shared resource (*e.g.,* firewall, cloud gateway, etc), multiple optimizer instances could recommend creating new instances of those shared resources at different locations, whereas a more optimal recommendation would be to create as few as possible of those shared resources. In the subsequent sub-sections, we discuss each in more detail.

5.2.1 Query Redundancy

As we have motivated in §4.2, homing a service request often involves complex interactions with cloud and service controllers to query for *run-time* information. The homing service needs such information to evaluate *run-time* constraints such as current capacity of a certain cloud site to home a given demand. For each homing request, the homing service sends hundreds of queries (if not more). Cloud and service

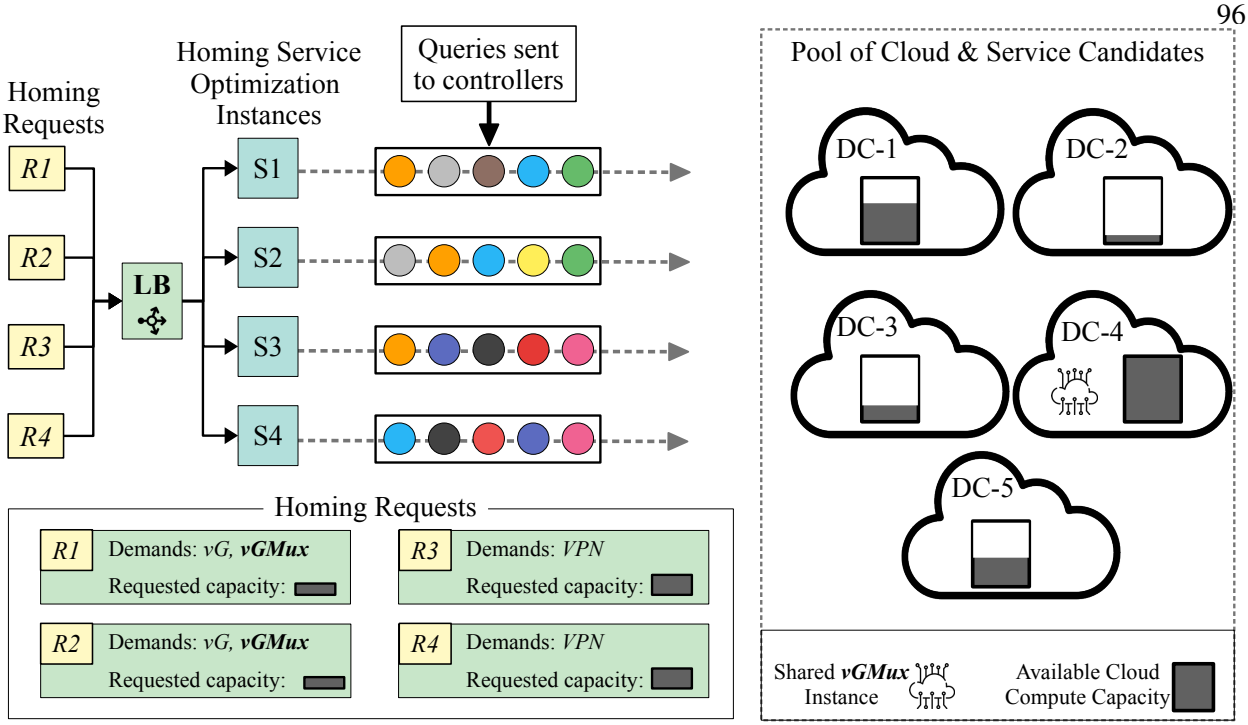


Figure 5.1: Example of running multiple homing service instances to process homing requests in parallel. Requests *R1* and *R2* are instances of the vCPE service (see Figure 4.1), while requests *R3* and *R4* are instances of a VPN service.

controllers, as a result, place a limit on how many queries they should receive from the homing service. It is important that those controllers utilize their own resources to be able to manage the services and compute resources in their domain. Although our incremental approach (§4.4) works well for each individual homing request, we want to explore the possibility of extending it to reduce redundant queries across multiple homing requests.

Figure 5.1 shows how multiple optimization instances of the homing service process homing requests in parallel. Each homing request (*R1* - *R4*) is handled by a separate homing optimizer instance (S1 - S4). When each optimizer instance starts evaluating the *run-time* constraints for each of those requests, it sends a number of queries to the cloud and service controllers. As can be seen, some of these queries (*e.g.,* orange and blue queries) are issued by more than one of those processing instances. In practice, there exists great redundancy in queries. To quantify query redundancy, we analyzed 1200 homing requests, expanded out the queries each would perform, and found that approximately 75% of queries showed some degree

of redundancy (*i.e.,* were issued by more than one homing request)[1] . Specifically, we found that 40% of queries were issued more than 100 times, and 5% of queries were issued more than 400 times across all homing requests. The impact of this is unnecessary load being placed on the controllers.

5.2.2 Resource Contention

Resource contention, in the homing context, takes place when two or more homing requests select the same resource (*e.g.,* a cloud-site to instantiate a new VNF, or a shared instance of an existing VNF) to home one of their demands. For example, consider the homing requests *R3* and *R4* in Figure 5.1, where they both ask for instantiating a VPN service at a cloud site that is closest to their customers' locations (*e.g.,* both in the west-coast). When two homing optimizer instances (S3 and S4) start to process *R3* and *R4* in parallel, they will send queries to the cloud controllers to inquire about their capacity. After querying the cloud controllers, both S3 and S4 determine that cloud site (DC-1) is the best candidate to home the VPN demand of *R3* and R4. However, as can be seen, DC-1 does not have enough capacity to support both demands – resulting in resource contention. This leads to failure in provisioning (or reserving resources)[2] at least one instance – resulting in re-solving one of those two requests from scratch (which, in turn, leads to issuing more queries and delaying service provisioning). Our analysis of production traces of a homing service shows that $\approx 15\%$ of homing requests fail to provision their required resources for this reason–leading to service delays and unnecessary extra burden placed again on controllers by re-solving those requests.

Further, knowing whether a given candidate can home one more demand is operationally impractical. This is due to the black box nature of those controllers (many of which are 3rd-party). That is, they provide a very specific set of APIs that the homing service can consume. The capacity APIs (that determine whether a given candidate has enough capacity or not) are built to support only binary yes and no answers. This makes it extremely difficult to infer how much resource a given candidate has, and therefore, the homing service cannot perform cumulative capacity checks for multiple requests.

[1] Note that queries are sent only once for each homing request. This is achieved using *StepNet*'s per-request query cache (see Figure 4.3)

[2] The homing service has a resource reservation component whose job is to communicate with the corresponding controllers to reserve the required resources after a homing optimizer instance solves a given request.

5.2.3 Unoptimized Provisioning of Shared Resources

Recall that one of the identifiable characteristics of homing is that some network services (VNFs) can be shared across homing requests, and potentially across customers (*e.g.,* a shared firewall instance, or a vGMux instance as in the vCPE service). To better optimize resources and utilize the underlying infrastructure, network operators try to minimize the cost of provisioning such instances by deploying as few instance as possible to support as many requests as possible.

Consider the two homing requests *R1* and *R2* in Figure 5.1, where both are instances of the vCPE service belonging to different customers. When two optimization instances (S1 and S2) start procesing *R1* and *R2* in parallel, they will issue queries to the controllers to evaluate a set of candidates. In this case, the candidates for the *vG* demand are the cloud sites (DC-X), and the candidates for the *vGMux* demand include the existing service instance at DC-4, and all other cloud sites to instantiate a new instance of the service.

After evaluating the constraints (one of which is a capacity constraint for both *vG*'s and *vGMux*'s candidates), S1 and S2 will determine that using the existing *vGMux* instance at DC-4 is not feasible since there is not enough capacity to home the *vG* demand at that cloud site[3] . Therefore, S1 and S2 will pick another cloud site that has enough capacity such that it minimizes the distance between that site and the customers' locations in *R1* and R2. This can result in S1 selecting DC-2 as its solution for *R1*, while S2 selecting DC-5 as its solution for *R2*. In this case, *two* instances of the vGMux VNF will be provisioned at these two cloud sites. Although these are feasible solutions, they result in wasted resources since two instances of the *vGMux* VNF will be provisioned. A more utilized solution, however, would be to select DC-3 for both *R1* and *R2*, and provision only one instance of *vGMux* that both *R1* and *R2* share. As can be noticed, assuming independence of simultaneous homing requests can lead to the creation of many redundant shared resources. This is exacerbated by the fact that a typical homing service processes hundreds of daily homing requests.

Our discussions with a large NSP reveal that NSPs currently implement a queue that keeps track of homing recommendations (solved requests) that ask for creating those shared resources, and will limit the

[3] Recall that there is a colocation constraint in the vCPE service specifying that both *vG* and *vGMux* should be at the same cloud site.

number of shared resources to only one per cloud site. This provides only a partial solution to the problem since shared resources are not (should not be) limited to the boundaries of a cloud site. That is, a more optimal solution would be to consolidate multiple resources created in multiple neighboring cloud sites into one shared instance created in one of those sites. To quantify the importance of this issue, we have analyzed 12 service models that are offered by a tier-1 NSP, and learned that at least 70% of the demands of those 12 services require the use of shared VNF instances. And thus, if not handled with care, a naive solution can undermine the NSP's goal of utilizing their infrastructure resources.

5.3 Design of *StepNet+*

As we have motivated in §5.2, when the homing service does not account for homing request dependencies, it can lead to unwanted problems and inefficiencies. Specifically, mishandling those dependencies can result in placing unnecessary load on the cloud and service controllers by issuing redundant queries across homing requests. In addition, multiple homing request instances could compete for cloud and service resources – resulting in resource contention. Unoptimized resource sharing is also another problem that stems from mishandling dependencies between homing requests.

In this section, we present our design of *StepNet+*, a homing service that extends our original *StepNet* design (presented in Chapter 4) to account for dependencies between homing requests. Figure 5.2 shows our design of *StepNet+*. At a high-level, we adopt two main design decisions that enable us to handle homing request dependencies.

(1) **Centralized Query Caching** (§5.3.1): we describe how the use of a centralized cache can eliminate redundant queries across multiple homing requests. As opposed to using a per-request cache at each of the local optimizer instances, our centralized cache is shared by all optimizer instances.

(2) **Coordinated Homing Decisions** (§5.3.2): we propose an approach in which local optimizer instances coordinate homing decisions with a centralized controller, which addresses resource contention unoptimized shared resources through two novel techniques.

   (a) **Trace-driven Prediction for Resource Contention** (§5.3.2.1): we describe how we leverage

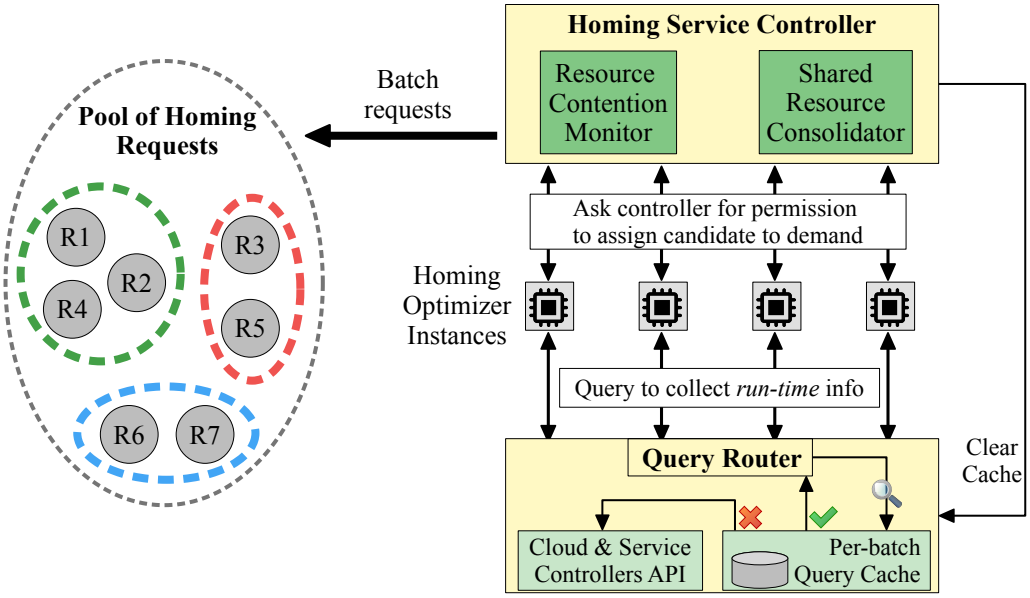


Figure 5.2: Design of *StepNet+* showing the homing service controller that is responsible for: (1) batching requests, (2) monitoring for resource contention, and (3) consolidating shared resources.

past query traces to predict whether a given candidate (resource) is able to home multiple demands.

(b) **Online Consolidation of Shared Resources** (§5.3.2.2): we propose an approach in which shared resources can be consolidated online (at time of solving). This allows us to minimize the total number of shared resources even across cloud-site boundaries.

Design optimizations: in §5.3.3, we also discuss how we optimize our proposed solution with a novel batching mechanism that maximizes cache locality, and minimizes the number of shared resources by scheduling *"similar"* homing requests to be processed together.

5.3.1 Centralized Query Caching

Our original design of *StepNet* (§4.5) uses a per-request local cache at each homing optimizer instance. To eliminate redundant queries across multiple homing requests, we propose decoupling the local cache from each local homing service instance, and have a centralized query cache that all homing service instances can access. Figure 5.2 shows a centralized cache at the homing query router (the component re-

sponsible for sending queries to the controllers). Whenever a homing optimizer instance wishes to send some queries (to evaluate *run-time* constraints), it sends a query request to the query router.

The query router, then, will compile a query signature for that query request. Such signature uniquely identifies each query. Then, the query router will check the query cache and fetch a response if it is able to find one. In case there is no response, the query router will send the query to the appropriate controllers using a well-defined set of APIs [100]. At the same time, the query router inserts an entry for that query in the query cache, with a response value set to *"pending"* to avoid sending the same query when other optimizer instances issue the same query request. After receiving a response, the query router will update the response value in the cache from pending to the actual response.

Having a centralized cache, however, presents a new challenge of when to evict cache entries to avoid having stale information. One solution is to assign per-entry timeouts that tell us whether a certain cache entry has expired or not. Setting conservative timeout values can lead to lowering the number of successful cache lookups as cache entries would be evicted very often. One can increase the timeout values to increase chances of successful cache lookups, but this can lead to information staleness. In §5.3.3, we discuss how our multi-criteria batching is able to maximize cache locality without leading to information staleness. After processing a batch of requests, then, the query cache is cleared by the homing controller (Figure 5.2) to start with a cleared cache for the next batch.

5.3.2 Coordinated Homing Decisions

The root cause of mishandling homing request dependencies is that homing decisions (solutions) are left entirely to each local homing optimizer instance (such as in Figure 5.1). That is, when a given homing optimizer instance (*e.g.,* S1) processes a homing request (*e.g., R1*), it will decide what candidates are assigned to which demands, guided by the objective function of *R1*. Figure 5.3 shows an example pseudocode of how the backtracking best-fit algorithm (**BACBF**)[4] is typically implemented. This shows (line 12) that the algorithm after evaluating the constraints, it assigns the best candidate to a given demand of

[4] Henceforth, we will use **BACBF** as a running example throughout this chapter, but the same intuition can be applied to other algorithms as well.

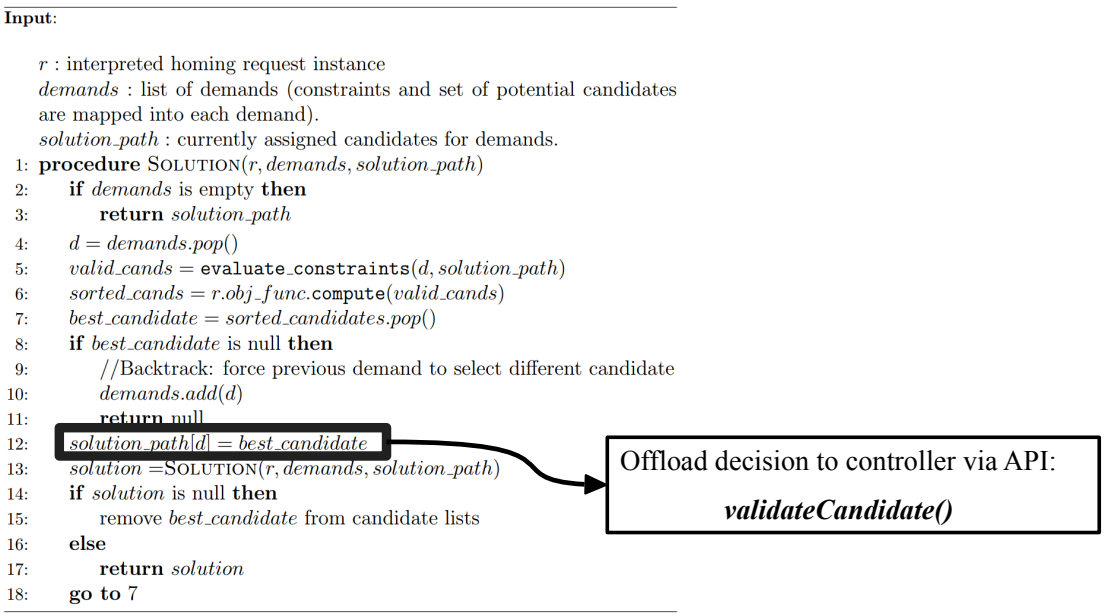


Figure 5.3: Example pseudocode of the Backtracking Best-fit (**BACBF**) algorithm. We offload the decision to assign candidates to demands (line 12) to the controller to allow it to monitor for resource contention as well as consolidate shared resources.

the request that it is processing. Leaving this decision to each instance running the algorithm is what causes problems like resource contention and unoptimized resource sharing. That is, each algorithm instance does not have the ability to know about other instances' solutions.

To handle homing request dependencies, we need to allow more coordination when it comes to assigning candidates to demands. To achieve this, we propose decoupling the decision of assigning candidates to demands for those local optimizer instances, and delegate this functionality to a centralized controller. Figure 5.2 shows a high-level design overview of *StepNet+*, in which we have a logically centralized controller whose main task (among others) is to coordinate between different homing optimizer instances. We achieve this by having the controller expose a well-defined (optimizer-agnostic) interface to the the underlying optimizer instances.

The optimizer instances, then, consume this interface whenever they wish to assign a candidate to a demand. Specifically, we replace line 12 in Figure 5.3 with a call to the controller (`validateCandidate()`,

Listing 2 Homing Controller's API to Consolidate Shared Resources and Monitor for Resource Contention

```
def validateCandidate(demand, best_cand, feasible_cands):
    if demand.type("shared_resource") and \
        best_cand is not service_instance:
        wait_for_other_requests(timeout)
        # for all other in-flight requests (in current batch)
        # that are interested in this shared_resource or until timeout
        chosen_candidate = consolidate_shared_resources(feasible_cands)
        # consolidates shared resources with other
        # requests in the current batch
        return chosen_candidate

    elif best_cand in requested_candidates:
        predictor = predict(best_cand)
        # predicts whether this candidate can
        # also accommodate this demand
        if predictor:
            return best_cand
        else:
            feasible_cands.remove(best_cand)
            return feasible_cands
    else:
        requested_candidates.increment(best_cand)
        return best_cand
```

shown in Listing 2), while passing the following pieces of information: the current demand (d), the best candidate (`best_cand`) that it wishes to assign to d, and a list of other feasible candidates that can also be assigned to d. Upon receiving this information, the controller will check for two issues: (1) whether the selected candidate (`best_cand`) can home d without resulting in resource contention, and (2) whether the selected demand (d) is looking for using a shared resource. Next, we elaborate more on how the controller is able to handle both cases.

5.3.2.1 Trace-driven Prediction for Resource Contention

Recall that when the homing service provides its recommendations (*i.e.,* solution) to a given homing request, it takes time for the service and cloud controllers to provision resources to home the demands of that request – leading to delays in reflecting changes for subsequent homing requests. Therefore, we

provide the controller with the ability to track what resources are being selected through a simple key-value lookup table (`requested_candidates` in Listing 2), with the key being the uniquely identifying `candidate-id` and the value being a list of demands that are interested in the same candidate so far. When the `validateCandidate()` API is triggered by one of the optimization instances (Figure 5.2), the controller will check whether the selected candidate exists in `requested_candidates` – *i.e.,* when other requests are also interested in the same candidate. If the candidate does not exist in the `requested_candidates` table, then the controller will add it to the table, and then allow the homing service instance that requested the candidate to use it. Otherwise, the controller will calculate the probability of the candidate's ability to home this extra demand.

Since the controller is not able to know the exact capacity of a certain candidate, it can only predict whether that candidate can support more than one demand at a time (*e.g.,* a cloud edge site that has a set of VMs that can home demands). One conservative approach would be to assume that whenever a candidate is selected by a given demand, then that candidate cannot accommodate more demands of other requests. This, however, is too conservative and can quickly result in missing on many valid candidates (like cloud sites that have enough resources to home several demands). Instead, we predict whether a given candidate can home those extra demands by looking at recent query responses from the cloud/service controller corresponding to that candidate.

For instance, if we see that a certain cloud site responds with *"yes"* to capacity queries most of the time for the past two days, then we can assume that it can accommodate more than one demand. This requires setting a threshold after which we predict that a given candidate can no longer home extra demands. As a starting point, we set the threshold to home two demands to 50% of past query responses being *"yes"*, and increase this threshold by increments of 10 as more in-flight demands accumulate. When the controller sees that query responses for that candidate do not meet the threshold, it will remove that candidate from the `feasible_cands` set and return this set to the homing optimizer instance that sent it. When the homing service instance sees that it did not receive the same candidate back from the controller, it will select the second best candidate from the set of `feasible_cands`, and so on.

Finally, the controller periodically updates its `requested_candidates` table when demands of

Prediction	Reality	Consequences
Yes	No	Requests will fail resource reservation, and will be re-solved (same as with no prediction).
No	Yes	Candidate will be removed from feasible candidates set of request, and request will be forced to select a different candidate.

Table 5.1: When our trace-driven capacity prediction does not match reality, two scenarios can occur. One is when we predict a certain candidate has enough capacity while it does not, and the other when we predict a certain candidate does not have enough capacity while it does.

those homing requests get provisioned (or fail to provision for that matter). This periodic check prevents the counts in our `requested_candidates` table to keep accumulating forever.

Correctness of Trace-driven Prediction. The best scenario for our prediction mechanism is when its prediction matches reality. However, it is possible that our prediction may not match reality. In that case, there can be two scenarios as depicted in Table 5.1. One is when we predict that a certain candidate has enough capacity to home an extra demand, whereas in reality it does not (first row in Table 5.1)–resulting in resource contention. This results in the same behavior of disabling the prediction mechanism and allowing demands to select the candidates they wish. To deal with this, we leverage a resource reservation mechanism that is already adopted as an intermediate solution. That is, after the homing service solves a certain request, it will pass the solution to a reservation component whose job is to reserve the required resources at each of the cloud/service candidates that are part of the request's solution. In case the corresponding controllers do not permit that reservation, the reservation will fail, and consequently, the request will be re-solved.

In the other scenario (second row in Table 5.1), we could predict a certain candidate does not have enough capacity to home that one extra demand, while in reality, it does. In this case, the request that asked for this candidate will not be able to use it since the prediction returned *"no"*. Consequently, that request will be forced to select a different candidate, which could negatively impact the objective value of the solution for that request. In rare cases, the predicted candidate may be the only feasible candidate for that request. In those cases, we can augment the local optimizer instances with an override primitive that allows them to override the controller's decision and use the requested candidate.

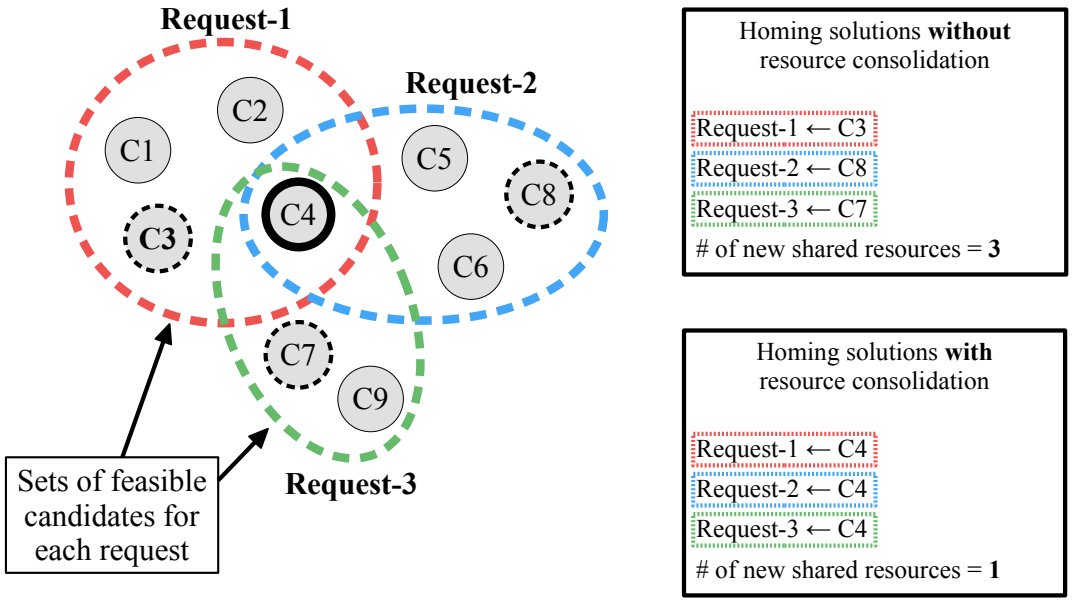


Figure 5.4: An example of how shared resource consolidation can minimize the number of new shared resource instances to be created to home the demands of in-flight requests.

5.3.2.2 Online Consolidation of Shared Resources

When the `validateCandidate()` API is triggered at the controller, another thing the controller checks is whether the demand it received through this API call is asking for a shared resource (*e.g.,* a slice of a gateway). In that case, it checks whether `best_cand` is a cloud-site – *i.e.,* meaning it would create a new instance of that shared resource at this cloud-site. In case `best_cand` is recommending creating a new instance of that shared resource, the controller will consolidate as many in-flight requests into as few new shared resource instances as possible. Specifically, the controller uses the list of `feasible_cands` of each in-flight homing request to find overlapping candidates for those resources.

Figure 5.4 shows two scenarios: with and without consolidating shared resources. When we do not consolidate shared resources, each of the three requests shown in the figure will select a different candidate to home their demands – leading the homing service to recommend creating a separate instance of that shared resource at each of these three cloud site candidates. Such result will pass the checks that NSPs currently have in place, where it consolidate shared resource instance if they are at the same cloud-site. A more optimized way, however, would be to look at the overlap in feasible candidate sets for those three

requests, and try to find a suitable candidate that works for all (or most) in-flight requests. The ideal case is to use one candidate (*e.g.,* C4) for all those in-flight requests, but in some cases, it can lead to using more than one – depending on how much overlap there is between those requests.

For shared resource consolidation to be effective, however, homing requests interested in the same shared resource need to be scheduled such that they are solved at the same time. This is critical since this step is performed online (while requests are being solved), and therefore, it would be useless if none of the in-flight homing requests share interest in the same shared resource.

In the next subsection, we discuss how we use multi-criteria batching to maximize the benefits of shared resource consolidation.

Consolidation Correctness. When resources are consolidated such that we are using the same cloud-site for a number of requests interested in a type of shared resource, there is a chance that an instance of that shared resource could not be created at that cloud-site (*e.g.,* not enough capacity). However, we believe that by having trace-driven prediction in place, we are minimizing the chances of that happening. In the worst-case scenario that this problem could occur, all requests, that had selected that cloud-site as their candidate to create an instance of a shared resource, will be re-solved. We believe this is an acceptable trade-off. That is, by consolidating many shared resources, it is an acceptable cost to pay to re-solve a small portion of homing requests.

5.3.3 Optimizing *StepNet*+ with Multi-criteria Batching

When homing requests are processed as they arrive (as in our original design of *StepNet*), our centralized query caching as well as shared resource consolidation mechanisms could be less effective. This is due to the fact that both mechanisms are time-sensitive, and require a number of requests to be simultaneously processed at the same time. That is, for both mechanisms to be effective, we need to fully utilize all running optimizer instances by scheduling homing requests such that we are solving as many homing requests as possible at any given time. One way to realize this is to process homing requests in batches. However, batching requests in the order in which they arrive may undermine the benefit of batching in this case. That is, when such strawman's approach batches homing requests based on the order in which they arrive, those

requests may not be related to one another (*w.r.t.,* the type of shared resources they ask for or the type of queries they issue). Therefore, there can be little benefit of batching requests using this approach.

To this end, we propose batching *"similar"* homing requests using a novel multi-criteria batching mechanism. The main intuition behind using multi-criteria batching (as opposed to a strawmans's approach) is that we would like to assign *"similar"* homing requests to the same batch. We achieve this by accumulating homing requests for a larger batching window (*e.g.,* one hour), and use our multi-criteria batching to batch similar requests together.

Processing *"similar"* homing requests in batches is useful in at least two ways. First, by processing similar homing requests in batches, we can coordinate across those requests at run-time as opposed to processing individual requests and dealing with the aftermath of mishandling requests' dependencies. Second, it allows us to cache queries for a number of requests (within a batch) without sacrificing information freshness since queries triggered by those homing requests take place within a certain (small) time window that is equal to the longest solving time of a request in that batch. Consequently, doing so allows us to handle two problems at the batch level: optimizing resource sharing and eliminating redundant queries. To this end, we use intuition about these two aspects to guide our batching process.

5.3.3.1 Batching Criteria

After accumulating homing requests for a given time window (*e.g.,* one hour), we need to divide these requests into multiple batches. Our analysis of production traces of a homing service running at a tier-1 NSP network reveals that the average rate of request arrivals at the homing service is close to 67 requests per hour, a number that is expected to grow as more services are virtualized and more NSP clients are served. Therefore, it is infeasible to put all requests within that time window into the same batch. Instead, we divide those requests into multiple batches based on two main criteria: (1) the type of shared resources that those requests are interested in, and (2) query overlap between different requests. Figure 5.5 shows how *StepNet+* uses those two criteria to assign homing requests into batches. Now, we elaborate more on how we define each criterion.

1) Type of Shared Resources. First, we look at what shared resources that those requests are inter-

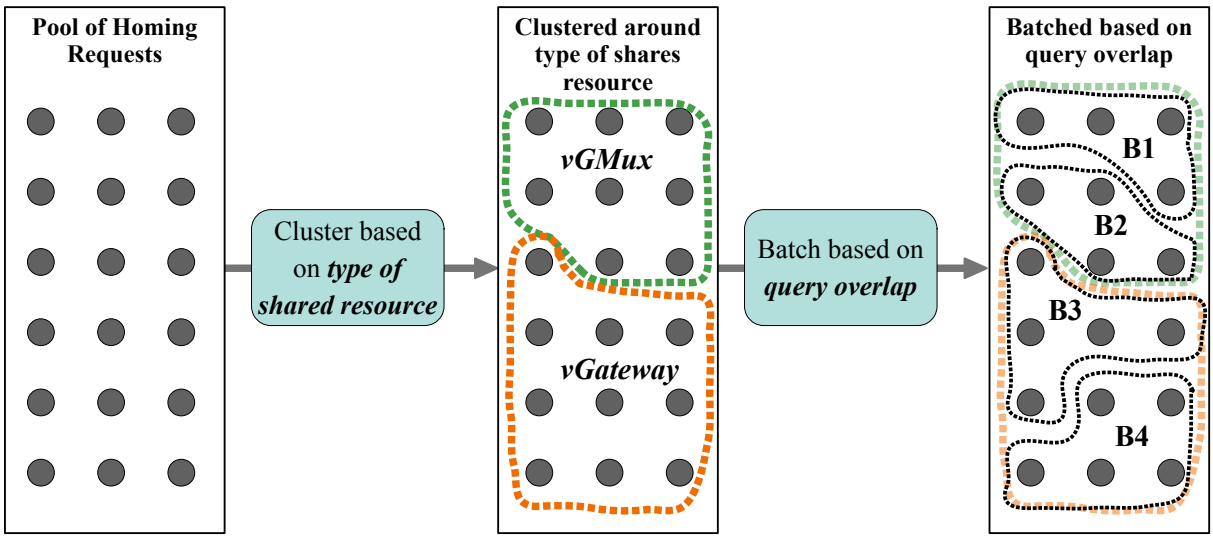


Figure 5.5: *StepNet+* adopts a multi-criteria batching approach, in which, the first step is to cluster requests based on the type of shared resources they are interested in, resulting in two clusters (*vGMux* and *vGateway*). Then, we batch requests within each cluster based on query overlap, resulting in 4 batches (B1-B4).

ested in, and use that as a first step to divide requests into multiple batches or clusters. For instance, vCPE requests interested in using a slice of a shared vGMux VNF (for reference, see Figure 4.1) can be in one cluster, and requests that are interested in using a slice of a shared firewall VNF will be put into a different cluster, and so on. Homing requests that do not ask for shared resources will be batched using the second criteria: query overlap. Using the type of shared resources as one batching criterion is extremely useful when it comes to consolidating shared resources for requests within the same batch.

Since there is only a handful of different types of shared resources, using this criterion alone could result in large batches. In addition, recall that our incremental approach (§4.4) does not evaluate all potential candidates. Rather, it evaluates the top candidates for each request. Because of this, if we randomly assign homing requests that are interested in the same shared resource into the same batch, we could easily miss our chance of consolidating shared resources for those requests. That is, because of the way the incremental approach works, two instances of the same homing request may not have an overlap in their candidates – rendering the task of consolidating shared resources impossible.

2) Query Overlap. To further improve our chances of eliminating redundant queries as well as consolidating shard resources, we further divide the requests in each of those clusters into multiple batches

based on their query overlap. To enable this, we first need to know what queries each homing request is likely to issue. Recall that queries are issued when the homing service evaluates *run-time* constraints that require *run-time* information. But, evaluating those constraints to know what queries they send, as have been demonstrated before, is prohibitively expensive. Fortunately, we know how each of those constraints construct their queries. That is, for a given set of candidates, and a homing solution path (see §4.3), the constraint evaluation function will issue a specific set of queries to the corresponding controllers. In our original design of *StepNet*, we used query signatures to uniquely identify queries so that we can cache them for each homing request. We leverage our knowledge about how those constraints construct those queries to pre-compile a list of queries each homing request is likely to issue.

Pruning the pre-compiled query set. Pre-compiling all potential queries that a homing request may issue can produce an exhaustive set of queries. At run-time when the homing service evaluates the constraints, however, only a smaller subset of those queries will be actually issued. This is due to the fact that our incremental approach already prunes the set of candidates (and, thus, the corresponding queries issued for those candidates) to a smaller set[5] . Further, exhaustively pre-compiling all potential queries can lead to misleading information. For instance, consider two vCPE homing requests that have the same set of initial candidates, but whose objective functions optimize for minimizing distance for two different locations (e.g., one in Europe and one in the US). Exhaustively pre-compiling the set of potential queries for those two requests could yield to producing the same set of pre-compiled queries, and determine that they can be in the same batch. At run-time, however, the incremental approach will evaluate only the top candidates for each request. This may result in not having any (or minimal) query overlap between those two requests.

To avoid an exhaustive pre-compilation and to further narrow down the set of potential queries, we leverage the same intuition we used to design our incremental approach (see Algorithm 6 in §4.4). That is, we first rank the candidates, and include only the top candidates in our query pre-compilation. We note that at this step, we include more candidates (*e.g.,* 3x) than the incremental approach's step size. For instance, if the incremental approach's step size is set to 2% of candidates for each of the demands, we include 6%

[5] In addition, some queries are iterative in nature (*i.e.,* depending in responses of previous queries, some new queries could be sent). We note that our current approach does not account for those queries.

of candidates in our query pre-compilation phase. Doing so allows us to include enough candidates (and, thus, queries to be issued for those candidates), and at the same time, avoid performing an exhaustive query pre-compilation.

After compiling and pruning the set of potential queries for all homing requests within the specified batching window, the controller batches those requests by looking at how much query overlap they have. That is, the controller will iterate over homing requests and assigned each to the batch that maximizes query overlap. In case there was not an overlap with any of the existing batches, the controller will create a new batch for this request. After doing so, the controller will iterate over batches and try to merge smaller batches together (e.g., batches that have one or two requests).

5.3.3.2 Batching Window and Batch Size

Window size. Our design does not rely on a fixed-size batching window (*i.e.,* the period of time in which we accumulate homing requests), instead we leave the decision of determining the batching window size to the operator as a configuration that is passed to the controller. However, our batching mechanism is more effective when the batching window is large enough to accumulate a large number of homing requests. Doing so allows the controller to be more effective in terms of how it clusters and batches those requests since it will be acting on a larger volume of information.

Batch size. Likewise, the batch size in our design can be passed to the controller as part of its configuration. Recall that shared resources offer *"slices"* of service as their unit of allocation. As part of our design guidelines, we recommend setting the batch size (how many requests can be assigned to a batch) to the number of slices a fresh instance of a shared resource can offer. We note that this number can be different for each type of shared resource.

5.4 Evaluation

In this section, we seek to evaluate the ability of *StepNet+* to eliminate query redundancy, help mitigate resource contention, and consolidate shared resources.

5.4.1 Implementation and Experimental Setup

We have implemented a prototype of *StepNet+*, building on top of our implementation of *StepNet* in Chapter 4 with approximately 620 lines of additional python code.

Our evaluation is done on an experiment we ran for 67 homing request instances (*i.e.,* average request rate per hour, as seen in production logs). We set our batching window to 60 minutes – *i.e.,* meaning we considered all 67 requests within the same batching window. Our batch size was set to 5 requests per batch, using the two criteria we described earlier: type of shared resources and query overlap. This resulted in having 14 batches of requests, where all requests in a given batch were evaluated by five homing services instances in parallel. Throughout our evaluation, we used our incremental approach with the backtracking best-fit algorithm (**BACBF**-in).[6] For other configurations, we used the same emulation framework (§4.6) to emulate cloud and service controllers. This also includes using production traces for our trace-driven prediction mechanism.

We compare our proposed solution with two other approaches: (1) **baseline** (our original solution, *StepNet*, as presented in Chapter 4 – no batching), and (2) **strawman** approach, where homing requests are batched and processed as they come in (no batching criteria). We note that for the strawman configuration, we also enable the same proposed techniques for handling resource contention as well as consolidating shared resources.

5.4.2 Eliminating Redundant Queries

Figure 5.6 shows the total number of queries that were sent to the controllers by each of the three configurations for all requests we evaluated. The left bar measures the total number of queries for the baseline (*StepNet*), where it resulted in sending close to 13K queries to the controllers. On the other hand, when requests were processed in random batches as they come in, the total number of queries dropped to approximately 6K queries for all requests (a 55% reduction when compared to the baseline). This highlights the benefits of using a per-batch query cache instead of a per-request query cache that is used in the baseline's configuration.

[6] Refer to §4.6 for more details.

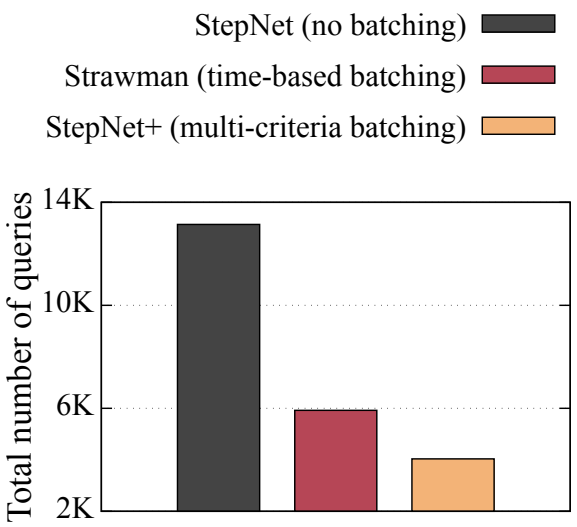


Figure 5.6: Total number of queries sent to the controllers for three approaches: (1) *StepNet* (baseline with no batching), (2) strawman (batch requests as they come in), and (3) *StepNet+* (using multi-criteria batching)

Further, the right bar shows the total number of queries for our multi-criteria batching, where it resulted in sending only 4K queries to the controllers – reducing the total number of queries by 70% and 30% when compared to the baseline and strawman approaches, respectively. This demonstrates the benefits of batching homing requests based on how much query overlap they have.

5.4.3 Shared Resource Consolidation

Now, we want to evaluate how effective shared resource consolidation is when we process homing requests in batches. Figure 5.7 shows how many new shared resource instances the homing service would recommend creating for all evaluated homing requests for each of the three configurations we evaluated. Note that for the baseline's configuration, if the recommendation for one request is to create a new shared resource instance at cloud-site X, and later on another request's recommendation happens to be the same, we count these as one new shared resource instance. This is along the lines of the current practice that NSPs have in place.

We can see in Figure 5.7 that the baseline recommended creating 31 new shared resource instances,

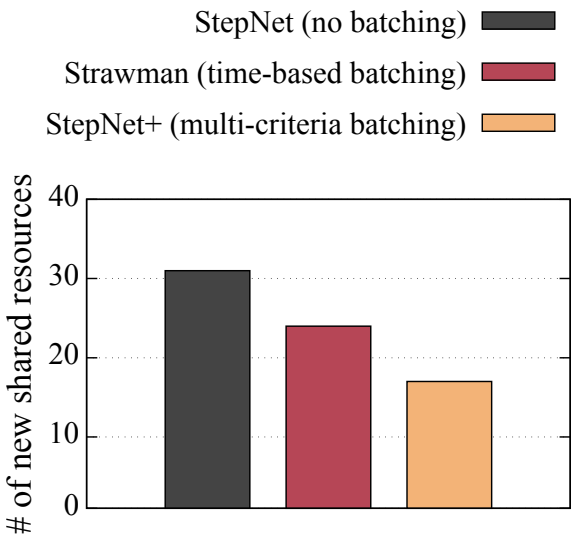


Figure 5.7: Number of new shared resource instances that each configuration have yielded for all evaluated homing requests. This shows *StepNet+* is able to reduce 45% and 29% of shared resource instances when compared to the baseline and strawman approaches, respectively.

amounting to half of the evaluated homing requests asking for new unique instances. When we introduce batching with the strawman, we can already see some improvement. The figure shows that the strawman's approach is able to reduce that number by 23%. Although this is a reasonable improvement, the figure shows that *StepNet+* is able to improve upon that even more bringing that number to only 17 (an improvement of 45% and 29% over the baseline and satrawman approaches, respectively).

The measurements from Figures 5.6 and 5.7 demonstrate the immense benefits of using multi-criteria batching to address homing requests' dependencies. These also show that even though batching alone (no criteria) can provide some benefit, it does not come close to the benefits that multi-criteria batching provides.

5.4.4 Reducing Resource Contention with Trace-driven Prediction

To measure the effectiveness of our trace-driven prediction mechanism, we ran an experiment in which we enabled trace-driven prediction for the strawman and *StepNet+* configurations described above. We configured our prediction mechanism such that it would enable a certain demand to use the requested candidate if past query traces returned *"yes"* to capacity queries for 80% of the time for the past 5 days.

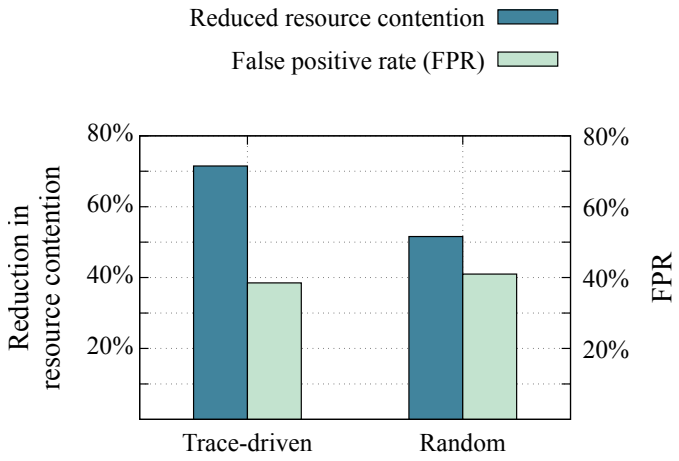


Figure 5.8: Our trace-driven prediction approach reduces resource contention by up 71% with a false-positive rate (predicting contention while there is not) of 38% on average. A random prediction approach reduces resource contention by only 52% with a slightly higher false-positive rate of 41% on average.

For each demand allowed to use a given candidate we add 10% to the threshold, and so on. We used production query traces to run this experiment. In order to reflect production scenarios, we flagged 15% (same percentage found in production logs) of the evaluated requests as requests that could cause resource contention. We, then, measure how many of those requests were forced to change their solution by the trace-driven prediction approach. We also compare our approach with a random approach that randomly predicts whether a given candidate can home one extra demand or not. We ran the experiment 10 times, and we report the average of these runs.

Figure 5.8 shows the reduction in resource contention (left y-axis) and false-positive rate (right y-axis)[7] each approach is able to provide while running the *StepNet+* configuration.[8] The figure shows that our trace-driven approach is able to reduce resource contention by 71% on average with a false-positive rate of 38%, while a random approach is able to reduce only 52% of resource contention with a false-positive rate of 41%. In case we predict contention while there is none (false positive), the homing requests are forced to select a different candidate to home their demands. We discussed the impact of this in §5.3.2.1. These results demonstrate the benefit of using query traces to predict resource contention. At the same time,

[7] A false positive is when we predict contention while there is not.

[8] Since our trace-driven prediction approach does nor rely on how requests are batched, the numbers for the strawman and *StepNet+* were highly similar.

we acknowledge that the false-positive rate is high, and we believe there is opportunity for improvement in the future.

5.5 Discussion and Future Work

Using Machine-learning approaches to improve the batching process. Although our multi-criteria batching shows promising results, we would like to investigate whether we can leverage a machine learning (ML) approach. Our current design uses fixed batching window and batching criteria to batch homing requests. An ML-based approach, on the other hand, could use dynamic batching window. That is, based on a trained model, it could predict that when it sees certain homing requests it could batch them (instead of accumulating requests for a longer period of time). In addition, we wish to study whether an ML-based approach can enhance batching accuracy even further.

In addition, using ML-based approaches could lead to better prediction when it comes to predicting resource contention. Our current trace-driven approach provides a starting point where we could extend it with ML-trained model that uses past query response traces, and at the same time, it enhances the prediction accuracy.

Synchronizing homing actions across parallel optimizer instances. Recall that when we prevent a certain demand from using a certain candidate (to prevent resource contention), such action could trigger a backtracking signal at the local homing service instance that processes that demand's request. This affects previous assignments of candidates to demands, which could be consolidated shared resource. Our current design permits such backtracking to take place locally. Doing so allows our design to be agnostic to the type of optimization heuristic that those homing service instances run. However, we wish to study the possibility of triggering such action across all other homing service instances to keep consolidated resources intact.

Cooperative homing decisions via voting mechanisms. When multiple homing optimization instances process homing requests in parallel, the homing controller is solely in charge of deciding what resource candidate can be selected by which optimization instances (*i.e.,* to prevent resource contention). Our current design deals with this problem on a first-come first-serve basis. However, this may not be the most optimal way to deal with resource conflict problems. Instead of leaving this decision completely to the

controller, we can give some control back to each optimization instance through voting approaches (such as Athens [10]). When each local optimization instance is augmented with a voting mechanism, it can *"negotiate"* with other parallel instances to resolve resource conflicts.

Impact on solution quality. Changing how the candidates get assigned to demands (to prevent resource contention or to consolidate shared resources) impacts the objective values of the corresponding homing requests. We wish to explore the impact of the controller's interference in this process, and correspondingly, plan on augmenting our controller with a mechanism to allow it to handle request dependencies while optimizing multiple requests' objective values.

Prediction accuracy of resource contention. Our evaluation shows that trace-driven prediction for resource contention is able to reduce a great deal of resource contention (71%), but it does so at a high cost of yielding a high false positive rate (predicting there is contention while there is not). In addition to exploring ML-based approaches for batching, we seek to enhance our prediction accuracy with similar approaches, where we can train a prediction model on production traces. We believe that our trace-driven prediction shows promising results, and that there are many opportunities to improve the prediction accuracy.

5.6 Conclusion

In this chapter, we present *StepNet+*, a homing service that accounts for dependencies between homing requests. It extends our original design of *StepNet* with two main design decisions. First, it batches requests to make dependency problems more tractable. Is uses these batches to eliminate query redundancy across multiple homing requests, as well as optimized shard resource usage across homing requests. Second, it coordinates homing decisions across multiple requests belonging to the same batch. This allows *StepNet+* to account for resource contention as well as optimize shared resource provisioning by consolidating multiple shared resource instances. Our evaluation shows that we can: eliminate up to 70% of redundant queries, and reduce the number of new shared resource instance by up to 45% when compared to the baseline. We believe that the design decisions we adopted in designing *StepNet+* provide a good start for a more optimized solution that we seek to pursue in the future.

Chapter 6

Related Work

This dissertation has touched on a variety of topics related to the homing problem, including: network function data-plane processing, cloud search services, and placement of network services in network service provider's infrastructures. In this chapter, we discuss highly related work to these topics, and shed light on some of the key differences that distinguish our work.

6.1 Stateless Network Functions

Beyond the most directly related work in Section 2.2, here we expand along three additional categories.

Disaggregation: The concept of decoupling processing from state follows a line of research in disaggregated architectures. [86], [85], and [115] all make the case for disaggregating memory into a pool of RAM. [54] explores the network requirements for an entirely disaggregated datacenter.

In the case of StatelessNF, we demonstrate a disaggregated architecture suitable for the extreme use case of packet processing.

Data plane processing: In addition to DPDK, frameworks like netmap [126] and Click [75] (particularly Click integrated with netmap and DPDK [12]) also provide efficient software packet processing frameworks, and therefore might be suitable for StatelessNF.

Micro network functions: The consolidated middlebox [131] work observed that course grained network functions often duplicate functionality as other network functions (*e*.g., parsing http messages), and proposed to consolidate multiple network functions into a single device. In addition, e2 [118] provides a

coherent system for managing network functions while enabling developers to focus on implementing new network functions. Each are re-thinking the architecture and complementary.

6.2 FOCUS

FOCUS is an extension of our earlier work [3] which introduced the idea to use p2p groups. In this chapter, we provide the complete architecture, implementation, evaluation and details of OpenStack integration.

Centralized Lookup Services: Node finding is a fundamental problem in any networking system. For example, name-based networking solutions such as the Intentional Naming System (INS) [1], Auspice [132, 138, 139], the Global Naming Service [81] or the geographic-addressing [52] in FocusStack [6] all propose to implement a centrally managed resolution service with nodes pushing updates to the centralized lookup service. Across many domains, either a push or a pull-based approach is used to enable the central service to satisfy the lookup – with the trade-offs having been carefully studied in [15]. In contrast, FOCUS follows a hybrid approach wherein a list of group members is periodically pushed to the centralized lookup service while to find the list of nodes satisfying a query, FOCUS uses a directed pull-based approach.

p2p Lookup Services: Node lookups are a fundamental part of p2p services. Systems like Gnutella [23] used a flooding protocol for information dissemination, Kademlia [91] uses a structured distributed hash table that allows node look up through structured IDs, and the gossip algorithm in [28] builds an unstructured p2p network and balances the update workload evenly among all the members in the group. The key differentiator in FOCUS is that a centrally managed system is dynamically managing the value-based p2p groups (based on Serf [56]) aiding nodes in joining and leaving the groups, and allowing for smaller p2p groups (reducing convergence time, and by extension, faster responses to queries).

Attribute-based Grouping: A key design decision in FOCUS is to group nodes in terms of attributes. This is similar in approach to publish-subscribe (pub-sub) systems [11, 42, 123, 143, 147], which also provide attribute-based grouping (e.g., channels and topics). But such systems, too, will not scale because nodes need to constantly publish or notify subscribers of their state through a global queue server (a bottleneck),

whereas in FOCUS we use a directed pull approach.

Cloud and Cluster Management: FOCUS's scalable and loosely-coupled design provides cloud and cluster management platforms (e.g., OpenStack [111], Google's Borg [141], Kubernetes [78], etc) with scalable search and a comprehensive view of the system with close to real-time information as compared to their push-based (OpenStack and Kubernetes) or pull-based approaches (Borg). Further FOCUS minimizes the resource usage of the controller. For instance, in order to scale Kubernetes to more than 500 nodes, the controller needs to have at least 36 vCPU cores and 60GB of RAM [77]. FOCUS's server, on the other hand, needs only 4 vCPU cores (an order-of-magnitude lower) and 16GB of RAM, out of which FOCUS utilizes only 10% to manage 1600 nodes (Figure 3.8a).

6.3 *StepNet*

There are two lines of work that, from a first glace, may seem highly relevant to our work on *StepNet*. However, there are key differences that distinguish *StepNet* from those works.

VNF and VM Placement: Numerous works in the areas of virtual network function (VNF) and virtual machine (VM) placement have looked at the placement problem in these two context [43, 49, 55, 74, 82, 95, 97, 119, 120, 135, 145]. These works develop service-specific optimization models and heuristics for for placing VNFs and VMs. These works typically formulate the placement problem using integer linear programming (ILP) or mixed integer linear programming (MILP), and propose tailor-made heuristics to relax and solve the problem for a specific use-case (*e.g.,* 5G network slicing).

SDN Optimization Frameworks: Work in the peripheral space of optimizing software-defined networks seeks to simplify the optimization formulation process. For instance, SOL [58] and Chopin [57] provide a limited set of high-level APIs (*e.g.,* add link capacity constraint) to software-defined networking (SDN) applications to efficiently manage network resources. SDN applications, then, need to consume those APIs, and the framework will model those high-level API calls as LP/ILP programs, and then solve them to find the best solution. However, these works assume complete knowledge when performing the optimization, so the addition or change of one service would require re-doing the whole optimization. VNF placement

approaches [49, 55, 74, 97, 119, 145] have the same disadvantage.

StepNet (and *StepNet+* by extension) differ from works in those areas in two unique characteristics. First, our *StepNet* framework caters to a variety of network services through a novel pluggable design. As we have shown in our evaluation (§4.6), *StepNet* is able to accommodate at least 12 service models through a set of constraint and objective function compositional blocks. Second, *StepNet* (and *StepNet+*) addresses the practical challenge of aggregating data needed to make informed homing decisions in an efficient manner through its incremental approach (§4.4).

Chapter 7

Future Work and Conclusion

7.1 Future Work

In this section, we discuss future directions that seem particularly promising. Specifically, there are two directions that we seek to pursue: improving our batching design of *StepNet+* with machine learning approaches, and enabling intent-based networking (IBN) with a novel system that interprets natural language to network configurations.

7.1.1 Improvements to *StepNet+*

StepNet+ uses multi-criteria batching to cluster and batch *"similar"* requests together. Such design has enabled us to account for dependencies between parallel homing requests, and has helped us minimize the impact of mishandling those dependencies (*e.g.,* redundant load placed on controllers, resource contention, and unoptimized resource sharing). Although our evaluation (§5.4) shows significant improvement, we wish to improve upon those results even more.

Instead of using a fixed set of criteria, we wish to employ some machine learning techniques to help us have better clustering. Specifically, our current design of *StepNet+* requires accumulating homing requests for a relatively large time window, and then cluster requests within that window such that similar requests are put into the same cluster. With an ML-based approach, however, we could train a model on a large set of homing requests (as well as query traces for those requests), and then use the trained model to decide when (and what) to batch.

In summary, we would like to augment our current design of *StepNet+* with machine learning to

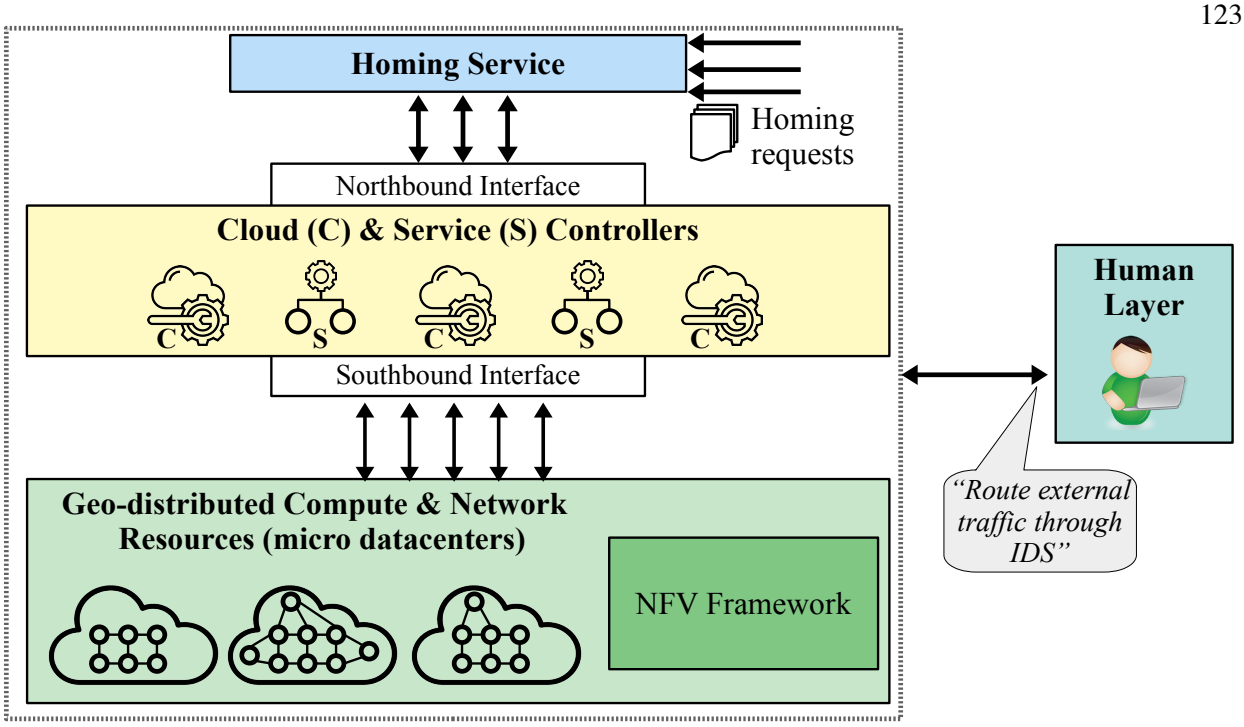


Figure 7.1: Interacting (configuring, querying, etc) with the network is challenging, and requires a decent level of knowledge on various networking languages and tools. We argue that it should be made simple by being able to perform various networking tasks using natural language.

improve its batching accuracy as well as its flexibility. There are different machine learning techniques that can be leveraged in this context. Specifically, there are different clustering techniques, such as: k-means [89] (or mini-batch k-means [130]), BIRCH (balanced iterative reducing and clustering using hierarchies) [148], and OPTICS (ordering points to identify the clustering structure) [7]. We wish to further study how we can leverage some of those techniques to better improve how *StepNet+* batches requests.

Finally, our evaluation of *StepNet* shows that our trace-driven prediction for resource contention is able to provide significant reductions for resource contention, but it does so at a relatively high cost of providing false positives. Such significant reductions highlight the benefit of performing trace-driven prediction. We seek to leverage this insight in training an ML-based approach with query traces to improve the accuracy of our prediction.

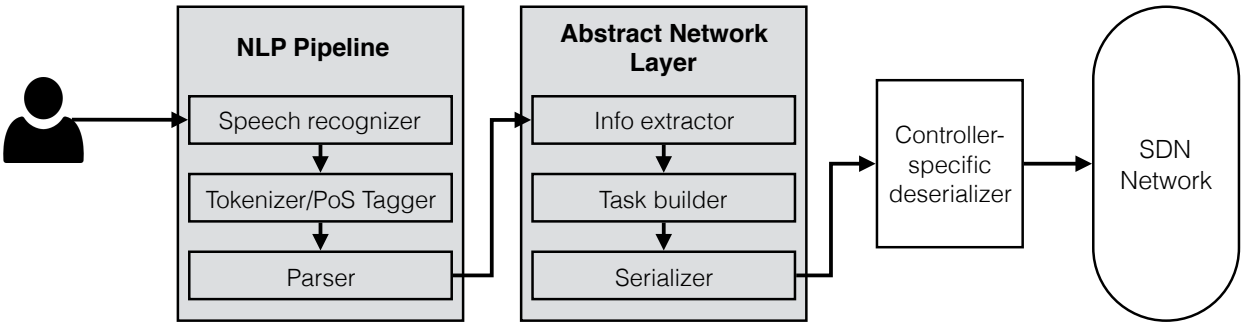


Figure 7.2: HeyNet Architecture

7.1.2 Enabling Network Management with Natural Language

Throughout this dissertation, we have looked at problems across the different layers of the homing stack (Figure 1.1), ranging from the data-plane layer (Chapter 2), traversing the control layer (Chapter 3), and moving up to the homing application layer (Chapters 4 and 5). One additional layer, that has recently gained great research interest [16,40,69], is the layer at which network operators interact (configure, manage, monitor, etc) with their networks. Shown in Figure 7.1 is a network operator interacting with the different layers of the network.

Network operators (or any individual for that matter) typically interact with their networks through the use of a variety of languages and configuration tools for each component of the network. That is, in order for one to manage a network, they need to learn a specific network configuration language. Whether that is in the form of configuring a specific SDN controller (a domain that has different languages for different controllers), or writing shell/python scripts to configure or monitor a certain equipment in the network. Having such heterogeneity in the tools and languages used to run, configure, and manage network systems is placing a significant burden on network operators and engineers. Even with NFV, what is becoming apparent is that configuring or managing network systems in general has become more challenging.

To this end, we identify a promising future direction in research through asking this question: *what if instead of configuring a network, we could talk to it, and it understood?* As a first step towards this, we designed a system we call HeyNet, which offers a natural language interface for users to interact with an SDN-operated network. HeyNet parses natural language input (*e.g.,* written English), constructs network

"tasks", which are translated into a format the network controller can understand (*e.g.,* OpenFlow). HeyNet achieves this through performing various steps in a processing pipeline that is depicted in Figure 7.2.

HeyNet introduces an abstract network layer that resides atop of existing network management solutions (*e.g.,* an SDN controller). Through this abstract layer (as shown in Figure 7.2), we abstract common network tasks (e.g., route flow x through a firewall-loadbalancer VNF service chain). When HeyNet receives a command/query in natural language from its users, it parses it leveraging natural language processing (NLP) techniques, and looks for keywords that can be used to identify the requested task. In the routing example mentioned, the keywords can be *"route"*, *"flow x"*, the word *"through"*, and the names of the VNFs (*firewall*, and *loadbalancer*). Using these keywords, HeyNet constructs a network task that is then translated to a format the network can understand (e.g., OpenFlow).

One of the main challenges in designing such systems is maintaining a reasonable level of accuracy. Our prototype of HeyNet is able to achieve 80% accuracy – meaning it is able to construct the correct tasks 80% of the time. To avoid any configuration errors, HeyNet can be configured to provide a verification mechanism for users to verify the constructed task is actually what they intended. This is especially important for critical tasks that change the network behavior/state. For other query-like tasks, they can be performed without resorting to user's help. We believe HeyNet offers a promising research direction that is worth pursuing to explore the extent to which it can operate.

7.2 Conclusion

In this dissertation, we identify limitations and problems with each of the four main steps of homing network services: (1) provisioning virtual network functions (VNFs) in an elastic fashion, (2) controller-node query processing in a scalable and real-time manner, (3) querying cloud and service controllers to obtain information needed to help make informed homing decisions, and (4) accounting for dependencies while processing homing requests in parallel. We design systems that overcome challenges and issues discovered in practice with each of these three steps. Specifically, the contributions of this dissertation are as follows.

First, we design a novel VNF architecture that decouples internal state from the processing logic – providing truly elastic and efficient VNF management primitives. Second, we design and implement FOCUS, a scalable and efficient search service that solves scalability issues with current approaches and processes geo-distributed queries in a timely manner. We achieve this by offloading query processing to the end nodes (a task that used to be performed by controllers) by leveraging peer-to-peer techniques to form the end nodes into groups based on their state. FOCUS also offers a well-defined API that integrates easily to existing frameworks.

Third, we design and implement *StepNet*, a homing service that adopts an incremental approach to homing that intelligently queries only "good" resource candidates – leading to significant reductions in query cost (by 92% for half of the 1200 homing requests we evaluated). Third, we identify two major problems that arise from dependencies between homing requests: redundant shared resources provisioning and resource contention. Finally, we extend our design of *StepNet* to handle dependencies between homing requests – reducing redundant queries and resource contention, and consolidating shared resources in an efficient manner.

Bibliography

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. ACM SIGOPS Operating Systems Review, 33(5):186–201, 1999.

[2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. Commun. ACM, 1975.

[3] Azzam Alsudais, Zhe Huang, Bharath Balasubramanian, Shankaranarayanan Puzhavakath Narayanan, Eric Keller, and Kaustubh Joshi. Nodefinder: scalable search over highly dynamic geo-distributed state. In 10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18), 2018.

[4] Azzam Alsudais, Shankaranarayanan Puzhavakath Narayanan, Bharath Balasubramanian, Zhe Huang, and Eric Keller. Iterative plugin design and implementation. `https://wiki.onap.org/display/DW/Iterative+Plugin+Design+and+Implementation`, 2020.

[5] Amazon Elastic Compute Cloud (EC2). `https://aws.amazon.com/ec2/`.

[6] Brian Amento, Bharath Balasubramanian, Robert J. Hall, Kaustubh R. Joshi, Gueyoung Jung, and K. Hal Purdy. FocusStack: Orchestrating Edge Clouds Using Location-Based Focus of Attention. In IEEE/ACM Symposium on Edge Computing, SEC 2016, Washington, DC, USA, October 27-28, 2016, pages 179–191, 2016.

[7] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. ACM Sigmod record, 28(2):49–60, 1999.

[8] AT&T. Setting the pace with our next-gen network. `https://about.att.com/innovationblog/121514settingthepace`, 2014.

[9] At&t colocation services. `https://www.business.att.com/content/dam/attbusiness/briefs/cloud-colocation-services-product-brief.pdf`, 2018.

[10] Alvin AuYoung, Yadi Ma, Sujata Banerjee, Jeongkeun Lee, Puneet Sharma, Yoshio Turner, Chen Liang, and Jeffrey C Mogul. Democratic resolution of resource conflicts between sdn control programs. In Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, pages 391–402, 2014.

[11] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E Strom, and Daniel C Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, ICDCS '99, pages 262–, Washington, DC, USA, 1999. IEEE Computer Society.

[12] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

[13] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN), pages 1–6. ACM, Aug 2014.

[14] David Bermbach, Frank Pallas, David García Pérez, Pierluigi Plebani, Maya Anderson, Ronen Kat, and Stefan Tai. A research perspective on fog computing. In International Conference on Service-Oriented Computing, pages 198–210. Springer, 2017.

[15] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: disseminating dynamic Web data. IEEE Transactions on Computers, 51(6):652–668, Jun 2002.

[16] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Net2text: query-guided summarization of network forwarding behaviors. In 15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18), pages 609–623, 2018.

[17] BitTorrent. `http://www.bittorrent.com`.

[18] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13:422–426, 1970.

[19] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev., 44(3):87–95, July 2014.

[20] Kurt M. Bretthauer. Capacity planning in manufacturing and computer networks. European Journal of Operational Research, 91(2):386 – 394, 1996.

[21] Maurantonio Caprolu, Roberto Di Pietro, Flavio Lombardi, and Simone Raponi. Edge computing perspectives: architectures, technologies, and open security issues. In 2019 IEEE International Conference on Edge Computing (EDGE), pages 116–123. IEEE, 2019.

[22] Chameleon Cloud: A configurable experimental environment for large-scale cloud research. `https://www.chameleoncloud.org/`.

[23] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like P2P systems scalable. In SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, pages 407–418, New York, NY, USA, 2003. ACM.

[24] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In Proc. ACM Workshop on Research on Enterprise Networking (WREN), 2009.

[25] Central Office Re-architected as a Datacenter (CORD). http://opencord.org/wp-content/uploads/2016/03/CORD-Whitepaper.pdf, Mar. 2016.

[26] Digital Corpora. Digital corpora. http://digitalcorpora.org/corp/nps/scenarios/2009-m57-patents/net/.

[27] FCC Technological Advisory Council. 5g network slicing whitepaper. https://transition.fcc.gov/bureaus/oet/tac/tacdocs/reports/2018/5G-Network-Slicing-Whitepaper-Finalv80.pdf, 2018.

[28] A. Das, I. Gupta, and A. Motivala. SWIM: scalable weakly-consistent infection-style process group membership protocol. In Proceedings International Conference on Dependable Systems and Networks, pages 303–312, 2002.

[29] Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, pages 303–312. IEEE, 2002.

[30] Dell. Poweredge r520 rack server. http://www.dell.com/us/business/p/poweredge-r520/pd.

[31] Dell. Poweredge r630 rack server. http://www.dell.com/us/business/p/poweredge-r630/pd.

[32] Dell. Poweredge r720 rack server. http://www.dell.com/us/business/p/poweredge-7520/pd.

[33] Edsger W Dijkstra. A note on two problems in connexion with graphs. Numerische mathematik, 1(1):269–271, 1959.

[34] Jiankang Dong, Xing Jin, Hongbo Wang, Yangyang Li, Peng Zhang, and Shiduan Cheng. Energy-saving virtual machine placement in cloud data centers. In 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pages 618–624. IEEE, 2013.

[35] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pages 401–414. USENIX Association, Apr 2014.

[36] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In Proceedings of the USENIX Annual Technical Conference (ATC), pages 1–14, July 2019.

[37] Eclipse Jetty. https://www.eclipse.org/jetty/.

[38] Edge-Core. 10gbe data center switch. http://www.edge-core.com/ProdDtl.asp?sno=436&AS5610-52X.

[39] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In USENIX Symposium on Networked Systems Design and Implementation (NSDI), Mar. 2016.

[40] Flavio Esposito, Jiayi Wang, Chiara Contoli, Gianluca Davoli, Walter Cerroni, and Franco Callegati. A behavior-driven approach to intent specification for software-defined infrastructure management. In 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), pages 1–6. IEEE, 2018.

[41] Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. `http://portal.etsi.org/NFV/NFV_White_Paper.pdf`, 2012.

[42] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. ACM Comput. Surv., 35(2):114–131, June 2003.

[43] Weiwei Fang, Xiangmin Liang, Shengxin Li, Luca Chiaraviglio, and Naixue Xiong. Vmplanner: Optimizing virtual machine placement and traffic flow routing to reduce network power costs in cloud data centers. Computer Networks, 57(1):179–196, 2013.

[44] Reza Zanjirani Farahani, Nasrin Asgari, Nooshin Heidari, Mahtab Hosseininia, and Mark Goh. Survey: Covering problems in facility location: A review. Comput. Ind. Eng., 62(1):368–407, February 2012.

[45] Reza Zanjirani Farahani, Maryam SteadieSeifi, and Nasrin Asgari. Multiple criteria facility location problems: A survey. Applied Mathematical Modelling, 34(7):1689 – 1709, 2010.

[46] Aaron Gember, Prathmesh Prabhu, Zainab Ghadiyali, and Aditya Akella. Toward software-defined middlebox networking. In Proceedings of the 11th ACM Workshop on Hot Topics in Networks, HotNets-XI, pages 7–12, New York, NY, USA, 2012. ACM.

[47] Aaron Gember-Jacobson and Aditya Akella. Improving the safety, scalability, and efficiency of network function state transfers. In Proc. 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox), Aug 2015.

[48] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In Proceedings of the ACM SIGCOMM. ACM, Aug 2014.

[49] Milad Ghaznavi, Aimal Khan, Nashid Shahriar, Khalid Alsubhi, Reaz Ahmed, and Raouf Boutaba. Elastic virtual network function placement. In Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on, pages 255–260. IEEE, 2015.

[50] Mostafa Ghobaei-Arani, Mahboubeh Shamsi, and Ali A Rahmanian. An efficient approach for improving virtual machine placement in cloud computing environment. Journal of Experimental & Theoretical Artificial Intelligence, 29(6):1149–1171, 2017.

[51] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In ACM SIGCOMM computer communication review, volume 39, pages 51–62. ACM, 2009.

[52] Robert J Hall. A geocast-based algorithm for a field common operating picture. In MILITARY COMMUNICATIONS CONFERENCE, 2012-MILCOM 2012, pages 1–6. IEEE, 2012.

[53] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. IEEE Communications Magazine, 53(2):90–97, 2015.

[54] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In Proceedings of the 12th ACM Workshop on Hot Topics in Networks (HotNets). ACM, Nov 2013.

[55] Davit Harutyunyan, Nashid Shahriar, Raouf Boutaba, and Roberto Riggio. Latency-aware service function chain placement in 5g mobile networks. In 2019 IEEE Conference on Network Softwarization (NetSoft), pages 133–141. IEEE, 2019.

[56] Serf: Decentralized Cluster Membership, Failure Detection, and Orchestration. `https://www.serf.io/`.

[57] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K Reiter, and Vyas Sekar. Intent-driven composition of resource-management sdn applications. In Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, pages 86–97, 2018.

[58] Victor Heorhiadi, Michael K Reiter, and Vyas Sekar. Simplifying software-defined network optimization using sol. In NSDI, pages 223–237, 2016.

[59] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[60] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In USENIX Annual Technical Conference (ATC), volume 8. Boston, MA, USA, 2010.

[61] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14, pages 445–458, Berkeley, CA, USA, 2014. USENIX Association.

[62] IBM. Ibm ilog cplex optimization. `http://www.cplex.com/`, 2020.

[63] Algo-Logic Systems Inc. Algo-logic systems. `http://algo-logic.com/`.

[64] Microsoft Inc. Microsoft Azure Cloud Computing Platform and Services. `https://azure.microsoft.com`.

[65] Microsoft Inc. Microsoft Azure Virtual Machines. `https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-a8-a9\protect\discretionary{\char\hyphenchar\font}{}{}a10-a11-specs/`.

[66] Microsoft Inc. Single-root IOV. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/single-root-i-o-virtualization--sr-iov-`.

[67] Intel. Data plane development kit. http://dpdk.org.

[68] Intel. Ethernet converged network adapter. http://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-x520-server-adapters-brief.html.

[69] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, Ronaldo A Ferreira, Lisandro Z Granville, and Sanjay G Rao. Deploying natural language intents with lumi. In Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, pages 82–84, 2019.

[70] JSON (JavaScript Object Notation). http://www.json.org/.

[71] Gueyoung Jung, Matti A. Hiltunen, Kaustubh R. Joshi, Rajesh K. Panta, and Richard D. Schlichting. Ostro: Scalable placement optimization of complex application topologies in large-scale data centers. In 35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015, pages 143–152, 2015.

[72] Magnus Karlsson and Christos Karamanolis. Choosing replica placement heuristics for wide-area systems. In 24th International Conference on Distributed Computing Systems, 2004. Proceedings., pages 350–359. IEEE, 2004.

[73] Eric Keller, Jennifer Rexford, and Jacobus Van Der Merwe. Seamless BGP Migration with Router Grafting. In Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pages 235–248. USENIX Association, Apr 2010.

[74] Nodir Kodirov, Sam Bayless, Fabian Ruffy, Ivan Beschastnikh, Holger H Hoos, and Alan J Hu. Vnf chain allocation and management at data center scale. In Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems, pages 125–140. ACM, 2018.

[75] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. ACM Transactions on Computer Systems (TOCS), 18(3):263–297, Aug. 2000.

[76] Jeff Kronlage. Stateful NAT with Asymmetric Routing. http://brbccie.blogspot.com/2013/03/stateful-nat-with-asymmetric-routing.html, March 2013.

[77] Building Large Clusters - Kubernetes. https://kubernetes.io/docs/setup/cluster-large/.

[78] Production-Grade Container Orchestration - Kubernetes. https://kubernetes.io.

[79] AT&T Labs. At&t edge cloud (aec) - white paper. https://about.att.com/content/dam/innovationdocs/Edge_Compute_White_Paper%20FINAL2.pdf, 2017.

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

[81] Butler W Lampson. Designing a global name service. In Proceedings of the fifth annual ACM symposium on Principles of distributed computing, pages 1–10. ACM, 1986.

[82] Kien Le, Ricardo Bianchini, Jingru Zhang, Yogesh Jaluria, Jiandong Meng, and Thu D Nguyen. Reducing electricity cost through virtual machine placement in high performance computing clouds. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, page 22. ACM, 2011.

[83] T. Li, B. Cole, P. Morton, and D. Li. RFC 2281: Cisco Hot Standby Router Protocol (HSRP), March 1998.

[84] libvirt: The Virtualization API. https://libvirt.org/.

[85] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In Proceedings of the 36th International Symposium on Computer Architecture (ISCA), Jun 2009.

[86] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In Proceedings of 18th IEEE International Symposium High Performance Computer Architecture (HPCA). IEEE, Feb 2012.

[87] Kernel Virtual Machine (KVM). https://www.linux-kvm.org/page/Main_Page.

[88] Jiefei Ma, Franck Le, Alessandra Russo, and Jorge Lobo. Detecting distributed signature-based intrusion: The case of multi-path routing attacks. In IEEE INFOCOM. IEEE, 2015.

[89] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, number 14 in 1, pages 281–297. Oakland, CA, USA, 1967.

[90] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. Fogstore: Toward a distributed data store for fog computing. In 2017 IEEE Fog World Congress (FWC), pages 1–6. IEEE, 2017.

[91] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In International Workshop on Peer-to-Peer Systems, pages 53–65. Springer, 2002.

[92] 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, Aug 2008.

[93] Mellanox. Infiniband single/dual-port adapter. http://www.mellanox.com/page/products_dyn?product_family=161&mtag=connectx_3_pro_vpi_card.

[94] Mellanox. Infiniband switch. http://www.mellanox.com/related-docs/prod_eth_switches/PB_SX1710.pdf.

[95] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In INFOCOM, 2010 Proceedings IEEE, pages 1–9. IEEE, 2010.

[96] David Minodier, Gregory Dalle, Dave Thorne, and Dave Allan. Network Enhanced Residential Gateway. https://www.broadband-forum.org/technical/download/TR-317.pdf.

[97] Hendrik Moens and Filip De Turck. Vnf-p: A model for efficient placement of virtualized network functions. In 10th International Conference on Network and Service Management (CNSM), pages 418–423, 2014.

[98] At&t expanding reach of networks. https://www.networkworld.com/article/2342130/at-t-expanding-reach-of-networks.html.

[99] Palo Alto Networks. Palo Alto Networks: HA Concepts. https://www.paloaltonetworks.com/documentation/70/pan-os/pan-os/high-availability/ha-concepts.html.

[100] AAI REST API Documentation. https://wiki.onap.org/display/DW/AAI+REST+API+Documentation+-+Dublin.

[101] Onap homing and allocation service. https://wiki.onap.org/pages/viewpage.action?\pageId=16005528.

[102] ONAP. Oof-has homing specification guide. https://wiki.onap.org/display/DW/OOF-HAS+Homing+Specification+Guide, 2020.

[103] ONAP. Open network automation platform (onap). https://www.onap.org/, 2020.

[104] Onap community. https://www.onap.org/home/community.

[105] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In USENIX Annual Technical Conference (ATC), pages 305–319, 2014.

[106] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), pages 29–41. ACM, Oct 2011.

[107] Open-O. Open-o. https://wiki.open-o.org/, 2020.

[108] OpenDaylight. The OpenDaylight Platform. https://www.opendaylight.org/.

[109] Openstack cascading solution. https://wiki.openstack.org/wiki/OpenStack_cascading_solution.

[110] OpenStack. OpenStack Nova. https://github.com/openstack/nova.

[111] Openstack: Open source software for creating private and public clouds. https://www.openstack.org.

[112] OpenStack Scalability Tests. https://docs.openstack.org/developer/performance-docs/test_results/1000_nodes/\index.html.

[113] Scale-out RabbitMQ cluster can improve performance while keeping high availability. https://www.openstack.org/assets/presentation-media/Scale-out-RabbitMQ-cluster-can-improve-performance-while-keeping-high-availability..pdf.

[114] Openstack. Openstack heat orchestration template. https://docs.openstack.org/heat/rocky/template_guide/hot_guide.html, 2020.

[115] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. SIGOPS Oper. Syst. Rev., 43(4), Jan. 2010.

[116] Christoph Paasch and Olivier Bonaventure. Multipath TCP. Communications of the ACM, 57(4):51–57, April 2014.

[117] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP), 2015.

[118] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP). ACM, Oct 2015.

[119] Akanksha Patel, Mythili Vutukuru, and Dilip Krishnaswamy. Mobility-aware vnf placement in the lte epc. In 2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), pages 1–7. IEEE, 2017.

[120] Jing Tai Piao and Jun Yan. A network-aware virtual machine placement and migration approach in cloud computing. In Grid and Cooperative Computing (GCC), 2010 9th International Conference on, pages 87–92. IEEE, 2010.

[121] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In Proceedings of the 2013 Conference on Internet Measurement Conference (IMC), Oct 2013.

[122] QEMU, the FAST! processor emulator. `https://www.qemu.org/`.

[123] Rabbitmq. `https://www.rabbitmq.com/`.

[124] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico replication: A high availability framework for middleboxes. In Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC). ACM, Oct 2013.

[125] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In Proceedings of the 10th USENIX Network System Design and Implementation (NSDI), pages 227–240. USENIX Association, April 2013.

[126] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[127] RoCE. RDMA over Converged Ethernet. `https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet`.

[128] Marty Roesch. Snort IDS. `https://www.snort.org`.

[129] Marty Roesch. Snort Users Manual 2.9.8.3. `http://manual-snort-org.s3-website-us-east-1.amazonaws.com/`.

[130] David Sculley. Web-scale k-means clustering. In Proceedings of the 19th international conference on World wide web, pages 1177–1178, 2010.

[131] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[132] Abhigyan Sharma, Xiaozheng Tie, Hardeep Uppal, Arun Venkataramani, David Westbrook, and Aditya Yadav. A global name service for a highly mobile internetwork. ACM SIGCOMM Computer Communication Review, 44(4):247–258, 2015.

[133] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-recovery for middleboxes. SIGCOMM Comput. Commun. Rev., 45(4):227–240, August 2015.

[134] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In Proceedings of the ACM SIGCOMM. ACM, Aug 2012.

[135] Vivek Shrivastava, Petros Zerfos, Kang-Won Lee, Hani Jamjoom, Yew-Huey Liu, and Suman Banerjee. Application-aware virtual machine migration in data centers. In INFOCOM, 2011 Proceedings IEEE, pages 66–70. IEEE, 2011.

[136] tcpreplay. Tcpreplay with netmap. `http://tcpreplay.appneta.com/wiki/howto.html`.

[137] The Xen Project, the powerful open source industry standard for virtualization. `https://www.xenproject.org/`.

[138] A. Venkataramani, A. Sharma, X. Tie, H. Uppal, D. Westbrook, J. Kurose, and D. Raychaudhuri. Design requirements of a global name service for a mobility-centric, trustworthy internetwork. In 2013 Fifth International Conference on Communication Systems and Networks (COMSNETS), pages 1–9, Jan 2013.

[139] Arun Venkataramani, James F. Kurose, Dipankar Raychaudhuri, Kiran Nagaraja, Morley Mao, and Suman Banerjee. MobilityFirst: A Mobility-centric and Trustworthy Internet Architecture. SIGCOMM Comput. Commun. Rev., 44(3):74–80, July 2014.

[140] Verizon corporation. `https://www.verizon.com/about/news/its-sweep-and-six-peat-win-verizons-network`, 2016.

[141] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In Proceedings of the Tenth European Conference on Computer Systems, page 18. ACM, 2015.

[142] ESXi — Bare Metal Hypervisor — VMware. `https://www.vmware.com/products/esxi-and-esx.html`.

[143] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a Replicated Logging System with Apache Kafka. Proc. VLDB Endow., 8(12):1654–1655, August 2015.

[144] Qing Wang, Geddings Barrineau, Ryan Izard, Jason Parraga, and Kuang-Ching Wang. Floodlight. `https://github.com/floodlight/floodlight/`.

[145] Ming Xia, Meral Shirazipour, Ying Zhang, Howard Green, and Attila Takacs. Network function placement for nfv chaining in packet/optical datacenters. Journal of Lightwave Technology, 33(8):1565–1570, 2015.

[146] Xiaodong Yu, Wu-chun Feng, Danfeng (Daphne) Yao, and Michela Becchi. O3fa: A scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection. In Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems (ANCS), Mar 2016.

[147] Distributed Messaging - zeromq. `http://zeromq.org/`.

[148] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. ACM sigmod record, 25(2):103–114, 1996.