Adaptive Load Balancing in P4: The Power of Promethee-Prometheus and Probes

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Mangesh Dalvi -112001010
Yukta Salunkhe -112001052

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**Adaptive Load Balancing in P4: The Power of Promethee-Prometheus and Probes**" is a bonafide work of **Mangesh Dalvi and Yukta Salunkhe**, **Roll.No: 112001010 and 112001052** carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr. Anish Hirwe
Assistant Professor
Department of Computer Science & Engineering
Indian Institute of Technology Palakkad

Acknowledgements

We hereby present our BTP project report on 'Adaptive Load Balancing in P4: The Power of Promethee-AHP and Probes'.

We would like to express our gratitute to our project mentor Dr.Anish Hirwe for giving us all the help and guidance we needed. We are grateful for the valuable support and suggestions provided by him, throughout the year.

We also thank our panelists for their suggestions on our project work.

Contents

List of Figures

Abstract

The rapid evolution of technologies such as Industrial IoT, cloud computing, AI, and machine learning has propelled businesses into the digital age, driving significant growth in network bandwidth. However, this progress has introduced challenges in terms of performance and efficiency of network devices and applications running on it. Moreover the traditional data planes are constrained by fixed functions and limited protocol sets, which reduces its flexibility and adaptability. This inflexibility from the vendor side leads to lengthy and costly development processes. However the recent developments in data plane programmability have opened new avenues for customization, attracting attention from both researchers and industry professionals.

Our project delved into the investigation of designing in-network computation accelerated application, with a primary focus on exploring the capabilities of next-generation network devices' programmable data planes. We reviewed recently published research papers that were primarily focussed on improving the performance of application protocols and/or enhancing the rapid responsiveness of network devices to changes in the network environment. Our final goal was to ultimately deploy/implement some application in the data plane, with a purpose to increase throughput and reduce latency.

We could identify a few key strategies within the research papers we went through. One crucial aspect was the deployment of specialized hardware, including FPGAs and programmable ASICs, to effectively offload and accelerate data plane operations. These operations encompass critical functions such as packet forwarding and processing.

Research papers also shed light on the significance of kernel bypass techniques, such as SR-IOV, DPDK user space libraries, and Netmap. These techniques have been found to optimize network performance by eliminating bottlenecks associated with context switching, packet copying between kernel and user-space, per-packet interrupt overhead, etc.

Few papers also suggest the implementation of fast and efficient data structures. These data structures expedite data lookup and manipulation processes, contributing to improved performance. Furthermore, ongoing studies are exploring the potential of P4, a data plane programming language, due to its reconfigurability and flexibility, further enriching the landscape of network optimization. P4's flexibility and few network parameters can be used together for faster detection of network disruption and optimal rerouting.

P4s programmability and other hardware offload techniques motivated us to propose solution for traditional load balancing techniques and to overcome its limitations, especially targeting the virtualized environments. Our primary objective is finally to develop a dynamic load balancing solution that leverage the flexibility of P4 to distribute traffic efficiently across virtualized resources. The following sections will detail our findings on this research, with a particular emphasis on the load balancing solution we developed for virtualized environments.

Chapter 1

Introduction

The ever-growing demand for data and network services has spurred advancements in network technologies. These advancements strive to achieve several key objectives: increased throughput, improved performance, and reduced server load. One promising approach in this area is the concept of a programmable data plane.

In this report, we first aim to offer an overview of three pivotal research areas that we've explored: SmartNICs in Data Centers, Reliability and Failure Recovery, and Load Balancing and Congestion Management, to gain insights into enhancing network performance and efficiency. It begins with a comprehensive examination of Azure Accelerated Networking (AccelNet), an initiative by Microsoft Azure to optimize cloud-based services. AccelNet leverages custom Azure SmartNICs based on FPGA technology, offloading Software-Defined Networking (SDN) policies to reduce CPU core utilization, minimize latency, and deliver high-performance connections. The report also provides analysis of a research work that explored the requirements and bottlenecks associated with consensus algorithms. It introduces the "Consensus as a Service" approach, which implements the Paxos algorithm within the network's forwarding plane. Using P4 programmable data plane language and switch ASICs, this approach reduces latency and enhances throughput.

Further, we tried to explore the reconfigurability offered by P4- the data plane programmable language. The data plane, traditionally fixed in functionality, refers to the hardware and software components responsible for forwarding packets across a network. A programmable data plane, however, allows for customization of packet processing tasks. P4's components, including headers, parsers, match-action processing, tables, actions, and control programs, facilitate the separation of control and data planes through P4Runtime. This language is a powerful tool for optimizing network performance. The report also contains the code and P4 setup which we tried on for emulating ipv4-based forwarding of packets on a data plane.

Various rerouting techniques proposed by researchers following the implementation of fast failure detection mechanisms are also described. Additionally we also tried to look at different techniques through which we could dynamically chose an optimal path for rerouting traffic based on network conditions and specific network configurations.The advent of Quic, a novel transport protocol, has opened up new possibility for exploring and refining data-plane based rerouting techniques.

Lastly, the report underscores the necessity for dynamic load balancing within virtualized environments. As service traffic escalates, the capacity to horizontally scale across a pool of application servers becomes crucial for maintaining optimal performance. Traditional software-based load balancers are either static or work with a centralized controller. Such constraints are particularly problematic given the capability of modern networks to deliver billions of packets per second. Moreover, conventional load balancers often prioritize uniform request distribution, overlooking the importance of achieving a balanced distribution of server load. The dynamic nature of VM utilization and varying demand should be taken into consideration while balancing the load. Therefore, there is a need for a distributed and efficient solution that can perform weighted load balancing based on real-time metrics while minimizing CPU overhead.

In response to these challenges, we propose a dynamically adjustable traffic distribution, capable of delivering line-rate performance with minimal implementation complexity. By harnessing real-time metrics extracted from virtual machines (VMs), the proposed solution introduces two algorithms implemented using the P4 language. The parameters for the metrics can be customized by the network operators, according to their specific use cases, providing flexibility and adaptability in managing network traffic.

Initially, we would explain few key concepts that leverage the power of a programmable data plane, in this section. We will introduce SmartNICs (Network Interface Cards), the P4 (Protocol Independent Packet Processors) language, the Paxos algorithm, and techniques for connection recovery and rerouting, the intricacies of load balancing in virtual environments and the impact of various parameters on load balancing efficacy, before we explain our solution in further sections.

1.1 SmartNIC: For Programmable Data Planes

Traditional Virtualization and Performance Bottlenecks:

In traditional virtualized environments, all network traffic for VMs (Virtual Machines) is processed by the host's software switch (vSwitch). This involves copying packets between the host and VM memory, increasing processing overhead and reducing performance. This has many drawbacks such as:

- **Reduced Throughput:** Packet copying adds overhead, limiting the amount of data transferable.
- **Increased Latency:** Processing delays in the host software stack lead to higher latency (communication delays) experienced by VMs.
- **Higher CPU Utilization**: The extra processing burden increases CPU usage on the host.

Cloud providers offer network functionalities like security groups and virtual routing. These are typically implemented in the vSwitch on each host for scalability and simplicity. However, this approach can become complex for managing diverse and frequently changing network policies.

SR-IOV is one of the approach for Improved Performance. Single Root I/O Virtualization (SR-IOV) allows efficient sharing of physical network devices (like NICs) among VMs. It allows a single physical network interface card (NIC) to present multiple virtual NICs to virtual machines (VMs). This enables direct data exchange between VMs and external switches, bypassing the VM kernel and virtual functions (VF), thereby enhancing CPU utilization and reducing network latency.. VMs connect to virtual functions (VFs) exposed by the hardware, bypassing the host software stack entirely. This offers significant performance benefits. Direct access to hardware reduces processing overhead and increases data transfer speed, thus improving the throughput. Bypassing the host software stack lowers communication delays. Moreover, offloading processing to the hardware frees up the host CPU resources. While SR-IOV improves performance, it bypasses host SDN policies enforced by the vSwitch. This creates a challenge how to maintain essential network functionalities like security groups with SR-IOV enabled. This problem was tackled by Microsoft Azure in their paper Azure Accelerated Networking: SmartNICs in the Public Cloud[1]

Solutions like DPDK and SR-IOV have offered significant enhancements in terms of performance and control over network operations. However, the emergence of P4 introduces a paradigm shift, providing network engineers with unprecedented granularity and control over network device behavior. Unlike traditional fixed-rule networking paradigms, P4 empowers engineers to define packet processing logic with remarkable flexibility.

1.2 Programming Protocol Independent Packet Processors Language

P4 [2], which stands for ”Programming Protocol-Independent Packet Processors”, is a domain-specific programming language used to control the data plane of network devices like switches and routers.

- Programmable Data Plane: Unlike traditional network devices with fixed rules, P4 gives you the flexibility to define how packets are handled.
- Protocol Independence: The language itself is independent of specific network protocols, allowing you to write programs that work across different network environments.
- Unlike OpenFlow which defines a protocol for a central controller to configure forwarding rules on network devices, P4 focuses on data plane programmability. It allows to define the exact behavior of the data plane in network devices (switches, routers).
- It offers more flexibility and control compared to OpenFlow. Aprt from forwarding rules, one can also define how packets are parsed, manipulated, and even implement new protocols.

Few terminologies often used with P4:

- **P4 Architecture:** P4 architecture defines the blueprint for how programmers interact with the data plane of a network device. It acts as a middle layer, hiding the hardware specifics and providing a consistent way to write P4 programs. Common architectures include: V1Model and PSA. PISA and PSA, both are architectures for P4, but with key differences:
  PISA (Protocol Independent Switch Architecture): A single pipeline forwarding model with well-defined stages for parsing, matching, and actions. It’s simpler but less flexible.
  PSA (Portable Switch Architecture): A newer, more comprehensive architecture for P4-16 [3]. It defines a set of standard building blocks (packet_in, packet_out, meter, etc.) and allows for composability (combining these blocks). This enables writing programs for various network devices, not just switches.
- **P4 Versions:** P4-14: Introduced in 2014 [4], was heavily focused on the PISA (Protocol Independent Switch Architecture) model. PISA offered a single pipeline forwarding architecture for switches.
  P4-16 (P4-16 or P416): Released in 2017, it’s the latest version and offers more flexibility. P4-16 supports a wider range of architectures like PSA, allowing for more complex and composable programs.
- **BMv2 (Barefoot P4 Runtime)**: BMv2 is a versatile framework for building P4 targets (software or hardware implementations). It includes:

P4 Runtime: P4Runtime is used to define how the control plane interacts with the data plane, specifically for configuring P4 tables. Thus, P4 and P4Runtime together enable the separation of control and data planes, providing a standardized way to configure network behavior, which is particularly beneficial in SDN.
Simple Switch: A reference software switch implementation that demonstrates how to build a P4 target using BMv2.

- **P4 Pipeline:** The P4 pipeline refers to the series of stages a packet goes through in a P4-programmable switch. The specific stages and their order depend on the chosen architecture (V1Model, PSA). These stages typically involve:
  Parsing: Extracting information from the packet header.
  Matching: Comparing packet fields against programmed rules.
  Actions: Taking actions based on the match results (forward, drop, modify header).

Applications of P4:
Customizable Traffic Management: P4 allows for fine-grained control over how packets are forwarded, enabling features like traffic shaping, load balancing, and implementing new protocols.
Network Function Virtualization (NFV): P4 programs can be used to implement network functions (firewalls, load balancers, DDoS Attack Detection, Deep Packet Inspection, Network Telemetry, etc) in software on commodity hardware.

Limitations of P4:
Steep Learning Curve: Requires knowledge of P4 language, networking concepts, and potentially specific hardware architectures.
Limited Hardware Support while growing, and limited available online documentation.

P4's functionalities have also been researched for distributed environments, where it has garnered significant attention for its adaptability and performance-enhancing capabilities. Notably, P4 has been proposed for implementing consensus algorithms like Paxos, demonstrating its relevance in ensuring agreement among distributed network nodes. P4's programmable nature enables dynamic packet processing logic, making it an efficient tool for orchestrating distributed architectures. It can be leveraged to optimize communication protocols and enhance fault tolerance mechanisms in distributed systems.

1.3 Paxos Algorithm

Paxos is a consensus algorithm designed to achieve agreement among distributed network nodes. The Paxos algorithm proceeds in four phases: Phase 1 involves the Coordinator sending a prepare message to all acceptors, initiating the protocol and requesting preparation for a new proposal. In Phase 2, upon receiving a prepare message, a majority of acceptors respond with a promise message, assuring the Coordinator they will ignore requests with a lower round number and providing information on any previously accepted

values. Subsequently, if the proposer receives responses from a majority of acceptors, it sends an accept request, proposing a value for the current instance. Acceptors respond with acknowledgments, confirming acceptance of the proposal and ensuring consistent data values across replicas.
Lamport's 3 communication steps theorem is a fundamental result in distributed systems theory. It suggests that a minimum of three message exchanges is necessary to ensure safety in distributed consensus algorithms. With a minimum of three message exchanges, the algorithm can handle the network uncertainties (like delay, failure) more robustly. The first message proposes a value, the second acknowledges its receipt, and the third confirms its acceptance. P4xos[5] addesses the significant challenges of enhancing throughput and reducing protocol latency by using p4- programmable data plane language.

Research into P4, reveals another exciting application: detecting failures swiftly and rerouting paths faster within the data plane. As a language tailored for the data plane, P4 offers a unique advantage in detecting network failures promptly and rerouting paths seamlessly— directly within the network's forwarding plane. This approach aims to leverage P4's programmability to enhance fault tolerance, ensuring networks can swiftly respond to issues and perform better.

1.4 Connection Recovery and Rerouting

In the face of local failures, rapid restoration of connectivity becomes feasible through the swift deployment of either advanced fast failure detection mechanisms—such as those leveraging hardware-generated signals or robust unanswered hardware keepalive mechanisms. However in case of remote failures, the network have had to wait for the Internet to converge to get back the connectivity. But this convergence is very slow.

According to BLINK[6] it takes more than 100 seconds for half of the BGP peers to just receive the 1st BGP withdrawal message. The final withdrawal can take up to 5-10 minutes to propagate within all BGPs. Ideally, P4 implementation for fast rerouting should be efficient enough to catch the failure signal within the first re-transmission rounds. It should be also accurate enough to react to only major disruptive events while being ignorant to noise caused by congestion or other bogus flow. Research Papers like BLINK utilizes TCP retransmissions to identify remote link failures. Once the failure is detected, many different strategies are explained in papers regarding the rerouting of the flows.

Some researchers proposed to maintain the next hops list for each prefix in every p4 switch. If a failure occurs, the 2nd preferred next hop is chosen for traffic flow, until the BGP convergence occurs, and the primary next-hop is updated by the controller. But in this approach as the number of flows we are tracking increases, switch memory overhead increases. P4Neighbor[7] uses packet header to keep track of backup path. This approach has a low overhead over the memory in switch and in most cases it is reliable. The primary objectives of such recovery system involves:
1. Ensuring efficient recovery from failure for each switch-to-switch.

2. Using minimal extra storage space to store information about the backup paths.
3. The recovery from a link failure should be done without needing the controller to intervene.

The advent of P4-enabled rerouting directly within the forwarding plane has spurred our exploration into load balancing within virtual environments. Unlike static or centralized controller-based approaches, where every packet's decision relies on a centralized controller, P4's direct rerouting capability offers several advantages, particularly in dynamic virtual environments. P4's ability to reroute traffic directly within the forwarding plane allows for real-time adaptation to these changes without relying on external controllers. Additionally, traffic patterns to VMs can fluctuate unpredictably, leading to imbalanced server loads.

P4's programmability facilitates the integration of real-time monitoring capabilities directly into the data plane, enabling on-the-fly adjustments to traffic distribution and efficient decision-making based on up-to-date metrics.This enables load balancing in virtual environments to become more adaptive, efficient, and responsive to changing workload dynamics and server conditions. Earlier research has been made of implementing p4 for data center architecture, where Fat-tree like topology exist.

1.5 Dynamic Load Balancing

In modern datacenter networks, providing large bisection bandwidth and agility for diverse applications is crucial. To ensure smooth operations, it's crucial to have an efficient way to distribute this traffic across the network. Traditional approaches like Equal Cost Multi-Path (ECMP) load balancing, is widely used, but can suffer from poor load balancing due to hash collisions, lack of awareness of downstream congestion, sudden traffic changes due to part of network failure, network server imbalances, etc. To address this, recent efforts have explored centralized scheduling, local switch mechanisms, and host-based transport protocols, each with drawbacks in speed, optimality, or complexity.

A new approach, CONGA (Congestion Aware Balancing) [8], aims to overcome ECMP's [9] limitations by using link utilization information to balance load across paths. Unlike centralized schemes, CONGA operates in the data plane, allowing for rapid load-balancing decisions every few microseconds. However, implementing CONGA requires custom silicon and limits its scalability to topologies with a small number of paths, such as two-tier Leaf-Spine architectures as here the leaf switches need to store and determine the entire path, obviating the need to maintain forwarding state for a large number of tunnels(one for each path).

These techniques have two limitations. First, because switch memory is limited, they can only maintain a small amount of congestion-tracking state at the edge switches, and do not scale to large topologies. Second, because they are implemented in custom hardware,they cannot be modified in the field. To overcome these problems, HULA [10], a dataplane load-balancing algorithm was introduced. First, instead of having the leaf switches track congestion on all paths to a destination, each HULA switch tracks congestion for the best path to a destination through a neighboring switch. Second, they designed HULA for emerging programmable switches and programmed it in P4 to demonstrate that HULA could be run on such programmable chipsets, without requiring custom hardware. Hula's efficient line rate adaptability is because of probes carrying the important data in the forward plane, depending on which major routing decisions are taken directly in forwarding plane.

In the subsequent part of this section, we delve into our problem statement tailored for virtual environments. Unlike traditional data center architectures with fat-tree-like topologies, virtual environments present distinct challenges that necessitate a specialized approach to load balancing. Virtual environments are characterized by their dynamic nature, with VMs being provisioned, migrated, or decommissioned in response to changing workload demands. This dynamicity introduces complexities not present in traditional static data center architectures. Moreover, traffic patterns to VMs in virtual environments can vary significantly over time, leading to imbalanced server loads and potentially impacting performance and user experience.

1.6 Problem Statement

Current load balancing solutions rely on communication between a central controller and data plane, consuming CPU cycles on both ends. Moreover, traditional load balancing algorithms are not dynamic to adjust to the network traffic. So in order to improve efficiency and scalability, we propose a dynamic, load-aware load balancing algorithm that operates entirely within the data plane.

1.7 Motivation

In-line rate load balancing using P4 (Programming Protocol-independent Packet Processors) has become increasingly important and necessary these days due to several factors:

- Growing Network Traffic: With the proliferation of internet-connected devices, cloud services, streaming media, and other bandwidth-intensive applications, network traffic continues to increase exponentially. Traditional networking equipment may struggle to handle this growing volume of traffic efficiently.
- Demand for High Performance: In-line rate load balancing helps distribute traffic across multiple paths, reducing congestion and improving overall network performance.
- Load balancing helps optimize the utilization of these paths, ensuring that no single link becomes a bottleneck.
- Fault Tolerance and Redundancy.
- Dynamic Workload Distribution.
- Efficient Resource Utilization.

Load Balancers can be broadly classified into types: **Reactive and Proactive Load Balancers**:

Reactive load balancing operates in real-time, dynamically adjusting traffic distribution based on the current state of servers. This approach maximizes system performance by efficiently utilizing available capacity and quickly adapting to workload changes. While reactive load balancing offers fast reaction to traffic patterns and server health, it may introduce processing overhead and struggle with sudden spikes in traffic.

Proactive load balancing takes a predictive approach, making routing decisions based on anticipated future traffic patterns. It provides stability by preemptively distributing traffic, optimizing resource allocation, and helping maintain system performance. However, it may struggle to adapt quickly to sudden changes in workload and can add complexity to the load balancing process.

Reactive load balancing suits environments with fluctuating workloads, where real-time responsiveness is crucial, such as web applications or microservices architectures. Proactive load balancing is ideal for environments with stable traffic patterns, requiring stability and efficient resource allocation, such as enterprise applications or IoT systems.

Why in a Data Plane?

As traffic to a service grows, the ability to scale horizontally across a pool of application servers presents substantial performance benefits. Traditional load balancers usually ensure a uniform distribution of requests, which does not necessarily correspond to a uniform distribution of server load. Dataplane assures to deliver line-rate performance.

It gives network operators the ability to configure the logic based on which they want to direct their network load to various servers, also allowing them to add on other functionalities like Traffic Prioritization, Real time monitoring and analysis of network traffic, Traffic filtering by integration with security policies, etc. in this topology.

Load balancing in the data plane is essential for optimizing the performance, scalability, availability, and efficiency of applications and services, particularly in virtualized environments where VMs are utilized and traffic is routed through P4-based switches.

Traditional Load Balancing Algorithms:

Traditional load balancing algorithms are designed to distribute incoming requests or traffic across multiple servers or resources in a network. Some common traditional load balancing algorithms include Round Robin, Least Connections, and Weighted Round Robin. While these traditional load balancing algorithms have their advantages in simpler network environments, they may not be ideal for data center networks due to several reasons like scalability, flexibility, etc.

Traditional load balancing struggles with scalability due to reliance on centralized systems, lacks granularity in considering factors like server load or CPU utilization, and is inefficient in dynamic cloud environments. Modern solutions employ dynamic load balancing, application-aware routing, and predictive analytics to adapt to changing traffic patterns and integrate with automation tools, ensuring efficient resource utilization and scalability. Load balancing aims to distribute workload evenly across servers by dynamically adjusting based on factors like server load and response time, preventing overload and underutilization. Consideration of server metrics such as load and request latency enhances load balancing decisions.

The paper related to Effects Of Parameters To Load Balancers in Cloud Computing [11] discusses the impact of various parameters on load balancing in cloud computing environments. It highlights the increasing demands on cloud data centers due to a growing number of users and services, which can lead to overloaded servers, impacting performance and user satisfaction. The study underlines the importance of efficient load balancing algorithms and parameter optimization to minimize task completion times and ensure smooth operation in cloud environments. It emphasizes the critical role of "makespan" (runtime) in cloud data center performance. Makespan likely refers to the maximum completion time for tasks processed by virtual machines. The research suggests that focusing on effective load balancing algorithms is crucial for reducing makespan and improving overall cloud performance.

1.8 Objectives:

Our primary objectives include:

- Load Balance the network traffic and direct the requests to specific server (eg. VM).
- Minimize controller interaction with the data plane, reducing CPU cycles.
- Leverage P4's stateful memory to store connection information for TCP flows, eliminating the need for constant controller involvement.
- Offload load balancing logic to SmartNICs for hardware acceleration.

Benefits:

- Increased scalability by reducing controller overhead.
- Lower CPU utilization on both controller and data plane.
- Improved performance in terms of throughput and response time.

Chapter 2

Literature Survey

During our reserech phase, we delved into advancements in network infrastructure across three pivotal domains: **SmartNICs in Data Centers, Reliability and Failure Recovery, and Load Balancing and Congestion Management**. In the modern networking, these categories are extensively researched on for supporting efficient data transmission, network management, and application scalability. In this section we provide a brief overview of each paper and the solution proposed to tackle specific problem.

2.1 SmartNICs in Data Centers

2.1.1 SmartNICs and Data Centers: Azure Accelerated Networking

Problem Statement:

The public cloud is the backbone behind a massive and rapidly growing percentage of online software services. In the realm of Microsoft Azure's cloud infrastructure, these services utilize millions of processor cores, exabytes of storage, and petabytes of network bandwidth. Network performance, encompassing both bandwidth and latency, plays a vital role in the success of the majority of cloud-based tasks, with a particular emphasis on interactive customer-facing work loads.

Microsoft Azure has developed its own software-defined networks (SDN) in hosts. Leveraging host SDN packet processing capabilities in software necessitates additional CPU resources. Consuming CPU cycles for these services reduces processing power for customer VMs and escalates the overall cost of delivering cloud services. SR-IOV is proposed to overcome this problem but it gives direct access to NIC from VM by which they were bypassing SDN policies for the network flow as well.

Solution:

Cloud vendors selling Infrastructure-as-a-Service (IaaS) need private networks with customer supplied address spaces, virtual routing tables, scalable L4 load balancers, etc. But configurations of this change so frequently that it is hard to implement them in hardware. To address this issue Microsoft Azure is using Virtual Filtering Platform (VFP) which is cloud based programmable vSwitch and provides software defined networking (SDN) policies. Capability to adapt programmability for integrating new features, elevate the performance and efficiency of custom hardware, and establish deeper processing pipelines. All of these aspects are fundamental in enhancing the performance of individual data flows. Thus, FPGA was selected as a viable option for SmartNIC.

In AzureAccelerated Networking (AccelNet) host SDN stack is implemented on FPGA-based SmartNIC. AccelNet delivers near-native network performance in virtualized environments by offloading packet processing from the host CPU to SmartNIC. It combines the benefits of dedicated hardware performance with software programmability.

Generic Flow Tables (GFT) is a solution developed by AccelNet to address the challenge of SR-IOV and SDN policies. GFT is a match-action language that defines how to handle network packets for specific "flows" (data streams between VMs). Here's how GFT works:

- **Flow Matching:** When a new flow starts, GFT checks its table for a matching entry based on network characteristics (e.g., source/destination IP addresses). VFP Processing (if needed): If no entry exists, the flow is directed to the host's vSwitch (VFP) software.

- **SDN Policy Enforcement:** VFP applies all configured network policies (security groups, etc.) to the first packet of the flow.

- **GFT Entry Creation:** VFP creates an entry in the GFT table with the applied policies and delivers the processed packet.

- **Hardware Offload:** Subsequent packets in the flow are processed directly by the GFT hardware, ensuring both performance benefits of SR-IOV and policy enforcement from the vSwitch software.

GFT essentially bridges the gap between SR-IOV's hardware acceleration and the need for SDN policy enforcement in virtualized environments.

Conclusion:

In conclusion, the AccelNet system's control plane, facilitated by the Virtual Filtering Platform (VFP), orchestrates flow creation, deletion, and policy determination. This control plane seamlessly integrates with the data plane, which is offloaded to an FPGA SmartNIC. Through the Lightweight Filter (LWF) GFT abstraction, the SmartNIC presents itself as a single NIC supporting both SR-IOV and GFT functionalities.

The system design efficiently handles exception packets by routing them through the FPGA and VFP. Exception packets trigger flow creation tasks in the VFP, populating entries in the GFT to bypass SDN policies for subsequent packets in the same flow. The FPGA detects termination packets, prompting the deletion of corresponding flow rules from the GFT.

Significantly, AccelNet achieves remarkable performance metrics. It efficiently minimizes CPU core utilization for exception processing in the VM network datapath, reducing it to less than 1%. Additionally, FPGA latency overhead is less than 1μs compared to

SR-IOV NIC alone, achieving line rate performance and surpassing CPU cores alone. AccelNet consistently achieves line rate performance on individual connections and is scalable to speeds exceeding 100Gb. These results highlight the effectiveness and efficiency of AccelNet's architecture in optimizing network performance while minimizing CPU overhead and latency.

2.1.2 P4xos (P4 + Paxos)

Paxos (mentioned in Section 1), the consensus algorithm is developed to establish unanimity within a distributed network of nodes. In this process, one or more nodes suggest values to Paxos. Consensus is reached when a majority of the participating nodes in the Paxos system agree on a specific proposed value. This algorithm consist of 4 major components- Prepare, Promise, Propose and Accept. P4xos explores how programmable network switches can be leveraged to accelerate Paxos. By implementing Paxos in the forwarding plane of these switches, the authors achieve significant performance improvements. Their P4-based implementation shows a throughput increase of four orders of magnitude compared to traditional software implementations.

Problem:

Consensus algorithms are a necessary component of distributed systems to ensure agreement and fault tolerance, but their inherent communication, synchronization, and coordination requirements in distributed environments can introduce performance bottlenecks that affect latency, throughput, scalability, and overall system efficiency.

Solution -P4xos:

Consensus As A Service
To eliminate consensus as a performance bottleneck, the paper proposes an innovative approach of implementing the Paxos algorithm within the forwarding plane of the network. The central concept is to execute Paxos logic directly within switch ASICs without imposing additional assumptions on the network or using any additional network hardware. It operates through four key components: the Proposer initiates the process by suggesting values and encapsulating requests in Paxos headers; the Leader acts as an intermediary in some implementations, ensuring only one process submits a message and enforcing message order; the Acceptor receives proposals, validates them, and decides whether to accept them to facilitate consensus; and the Learner observes acceptor decisions passively to ensure agreement. The authors used the Libpaxos library and distributed processes across three machines and found that the leader was a bottleneck due to handling numerous network interrupts and data copying from kernel to user space, affecting overall performance. The implementation defines a Paxos header with fields like message type, round number, instance number, and value. These fields convey essential information about the message and its purpose in the Paxos consensus process. Consensus messages are encoded in custom 'paxos' header which is embedded in udp payload. P4xos provides separate libraries for proposers and learners. The proposer library offers a "Submit" function that sends a value

and waits for responses. The learner library receives votes from acceptors and triggers the "deliver" function upon receiving a majority vote. This function delivers the learned value and associated information to the application.

It therefore addresses the significant **challenges of enhancing throughput and reducing protocol latency** through these two key strategies:

- **Processing Consensus Messages in the Forwarding Plane**: Using P4xos, protocol latency can be reduced by enabling each network device to fulfill a specific role in achieving consensus (spine as leader, aggregate as acceptor, ToR switch as learner, and hosts as proposers). This reduces the number of network hops and latency by processing consensus messages within the forwarding plane as they traverse networks and hosts. [5] This eliminates two network traversals, resulting in a 3x latency improvement compared to standard Paxos.

- **Leveraging High-Performance Switch ASICs**: By utilizing high-performance switch Application-Specific Integrated Circuits (ASICs), the approach significantly boosts throughput, providing a more efficient alternative than relying on server hardware. Thus utilizing the switch ASICs would help network I/O bottlenecks associated with the software implementation.

Conclusion:

P4xos offers significant performance benefits compared to traditional software implementations. The hardware offloading enables it to process billions of Paxos messages per second, resulting in dramatically increased throughput and reduced latency. Additionally, the hardware implementation simplifies formal verification of the protocol, enhancing its reliability. Notably, P4xos achieves this improvement without requiring any additional network hardware, leveraging the capabilities of programmable switches.

2.2 Reliability and Failure Recovery

2.2.1 BLINK: Fast Connectivity Recovery Entirely in the Data Plane

BLINK[6] is a system designed to achieve fast recovery from network outages without relying on the control plane. It operates entirely within the data plane of network switches, enabling sub-second rerouting of traffic in case of disruptions. BLINK's key innovation lies in leveraging specific patterns in TCP (Transmission Control Protocol) behavior to detect failures. When a connection experiences an outage, TCP reacts by retransmitting packets repeatedly in an exponentially increasing timeframe. BLINK monitors these retransmissions across multiple data flows to identify large-scale disruptions. Therefore, the number of flows experiencing rerouting is monitored for remote failures rather than the number of retransmission. Overall the Blink's problem statement can be defined as below:

Problem:

Remote link failures cause BGP to propagate updates to all routers in the network to reflect the new topology. This process, known as BGP convergence, can take an average of 30 seconds or more, which can lead to significant delays and disruptions in network traffic.

Solution:

Implement the fast failure detection and rerouting mechanism solely on data plane- BLINK. Blink utilizes TCP retransmissions to identify remote link failures. This approach is based on the observation that TCP flows exhibit a characteristic pattern of repeated retransmissions of the same packets at exponentially increasing intervals upon link failures. It therefore monitors flows based on number of flows experiencing retransmissions and not just retransmissions.

- **Failure Detection:** Flow Selection: BLINK focuses on critical prefixes, monitoring a subset of flows (64 per prefix) using a hashing mechanism. It prioritizes active flows and ignores non-TCP traffic.
- **Sliding Window and Retransmission Patterns:** BLINK employs a sliding window to track the number of flows experiencing retransmissions within a specific timeframe (typically 800 milliseconds). It differentiates failure-induced retransmissions (consecutive packets) from congestion-induced ones (interleaved) by analyzing sequence numbers and timestamps.
- **Inferring Failures:** If a significant portion of monitored flows (usually more than half) exhibit this characteristic retransmission pattern within the window, BLINK infers a network failure impacting that specific prefix.
- **Swift Rerouting for Seamless Recovery:** Pre-configured Backup Paths: BLINK leverages pre-defined backup next-hop options for each prefix, configured by the network operator based on BGP routes and specific policies.
- **Active Monitoring of Backups:** To ensure reliable rerouting and avoid issues like blackholes or loops, BLINK actively probes these backup paths. It sends a small portion of traffic to each backup and analyzes the success rate. Only paths demonstrating successful traffic restart are considered viable for rerouting.
- **Blackhole and Loop Detection:** BLINK employs clever techniques to identify potential forwarding issues:
  Blackholes: By analyzing the percentage of successfully restarted traffic on a backup path, BLINK detects blackholes (paths leading nowhere).
  Forwarding Loops: It monitors for duplicate packets, a signature of packets endlessly traversing the same loop. This approach leverages the similarity between loop behavior and failure-induced retransmissions.

Conclusion:

Overall, BLINK offers a compelling solution for fast network recovery. It detects remote failure within the first re-transmission round. By operating entirely in the data plane, at line rate, Blink restores connectivity in O(s) time. The accuracy of its failure detection is more than 80%.

However, it's not without limitations:
Storage Requirements: Maintaining flow information and backup path lists can be storage-intensive on network switches.
BGP Convergence Reliance: BLINK relies on BGP convergence for primary path restoration, which can lead to a brief period of suboptimal routing.
DDoS Attack Vulnerability: Malicious actors could potentially simulate retransmissions to manipulate BLINK and reroute traffic for malicious purposes.
By understanding BLINK's strengths and limitations, network operators can effectively leverage it for faster and more resilient network infrastructures.

2.2.2 P4Neighbor: Efficient Link Failure Recovery With Programmable Switches

P4Neighbor[7] tackles link failures entirely within the data plane of network switches, aiming for fast recovery with minimal storage overhead. This approach contrasts with traditional control-plane based methods that require switches to store backup path information for all potential destinations. Such an approach becomes memory-intensive, especially in large networks.

Design Objectives:

- **Effectiveness**: Ensuring an efficient recovery from failure for each switch-to-switch.
- **Low storage overhead**: The switch needs minimal extra storage space to store information about the backup paths.
- **Fast**: Typically, the recovery from a link failure can be done without needing the controller to intervene.

Solution:

P4Neighbor, a data-plane based rerouting technique, utilizes a neighbor-based approach to determine backup paths, reducing switch memory usage compared to control-plane based methods.

Each switch independently calculates backup paths to its neighbors, embedding the backup path information in a packet's custom header. Upon link failure, the receiving switch extracts the backup path from the custom header and redirects the packet along the pre-determined route. Once the link recovers, the custom header is removed.

The receiving node selects the egress port based on the information provided in the packet header. Traffic rerouting occurs at line-rate, eliminating the need for control-plane intervention.

A single-bit field within each packet indicates the packet's current state:

- Normal State: A value of 0 indicates that the packet is in a normal state and should be forwarded according to the routing table.

- Recovery State: A value of 1 indicates that the packet is in a link failure recovery state, and the backup path should be inserted into the custom header.

Upon receiving a packet, if the packet is in a normal state, it is forwarded based on its destination address. The port associated with the destination path is checked. If valid, the packet is forwarded in its normal state. If not, the packet's state is changed to recovery, the backup path is calculated, and added to the packet's custom header. The custom header functions as a stack data structure, with the second added value positioned immediately after the first, and so on. The packet is then retransmitted to the ingress port of the same switch, with the one-bit field indicating that the packet is in the recovery state.

Backup Path Calculation Algorithm

In networks, ensuring redundancy requires calculating backup paths. The traditional method stores all possible backup paths at each switch, leading to high memory usage (n(n-1) rules for n switches). The neighbor-based approach tackles this by only storing backup paths for immediate neighbors (2e rules for e edges), leveraging pre-computed shortest paths (Floyd-Warshall). This significantly reduces switch memory consumption.

Conclusion:

P4Neighbor offers a compelling solution for link failure recovery by minimizing switch memory usage through a neighbor-based approach. It offers a significant reduction in forwarding entries, ranging from 57.9% to 84.5%, compared to conventional destination-based recovery mechanisms. However, it might lead to a slight increase in hop count -an increase in the number of hops from 1.08 upto 1.98 hops, the benefits in terms of speed and storage efficiency make it a valuable technique for network operators in large-scale deployments.

2.3 Load Balancing and Congestion Management

2.3.1 HULA: Hop-by-hop Utilization-Aware Load balancing Architecture

Existing techniques for congestion-aware load balancing, like CONGA, have limitations. These limitations come from the hardware they rely on. Firstly, switches have limited memory, so CONGA-like techniques can only track congestion for a small number of paths at the edge of the network. This makes them unsuitable for large networks with many paths. Moreover, since these techniques are implemented in custom hardware that can't be easily modified. This lack of flexibility makes it difficult to adapt them to new requirements or network conditions. To address these limitations, HULA introduces a data-plane load-balancing algorithm designed to work within the constraints of switch hardware.
First, instead of having the leaf switches track congestion on all paths to a destination, each HULA switch tracks congestion for the best path to a destination through a neighboring switch Second, they designed HULA for emerging programmable switches and programmed it in P4 to demonstrate that HULA could be run on such programmable chipsets, without requiring custom hardware. HULA is therefore, Scalable to large topologies, Adaptive to network congestion, Proactive path probing and to network failure, Programmable in P4.

Solution:

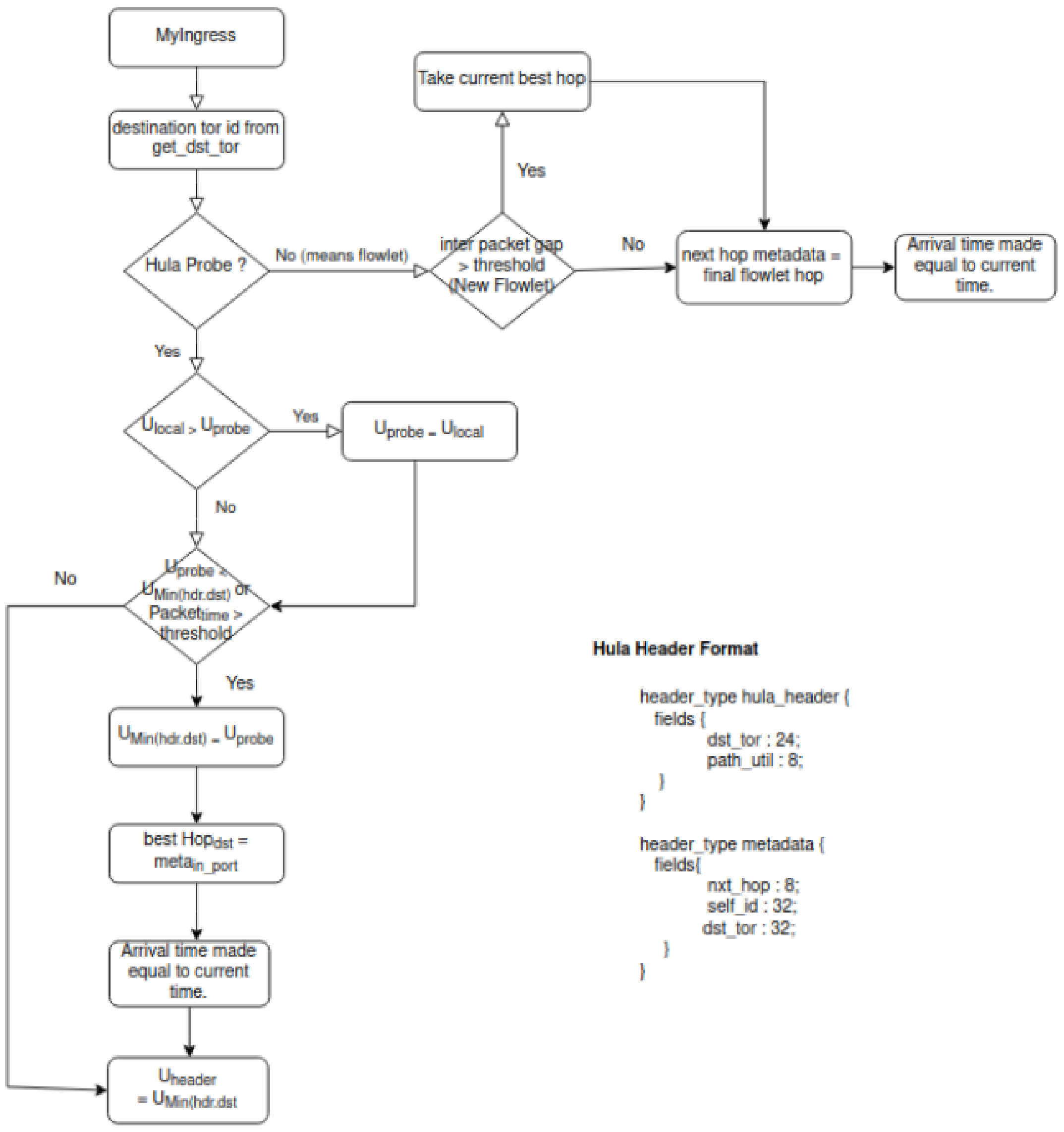
- Hula switch tracks congestion for the best path to a detsination through a neighbouring switch, instead of leaf switch tracking all paths.
- Each Hula switch only maintains congestion state for the best next hop per destination, and not all paths to destination.
- Use of special probes to gather global link utilization information.
- Stores the information related to best next-hop per destination and sends its view of best downstream path to other upstream switches.
- Information related to link utilization is propagated similar to Distance-Vector like propagation.

(Hula Logic is shown in Fig. 2.1).

Limitations:

One requirement for HULA is device homogeneity. In particular, HULA assumes that all switches in the network are P4 switches while CONGA requires custom hardware. In either case, however, the heavy operational burden involved in upgrading all the devices in a network makes deploying these applications difficult. It is not uncommon for modern data centers to have tens of thousands of machines. Upgrading all of these at once may be prohibitively expensive and operationally risky. In Dolphin: Dataplane Load-Balancing in Programmable Hybrid Networks [12], the authors try to explore the extent to which the benefits of dataplane time scale load-balancing can be realized after upgrading only a small fraction of the network.

Hula Logic


MyIngress
destination tor id from get_dst_tor
Hula Probe ?
No (means flowlet)
inter packet gap > threshold (New Flowlet)
Yes
Take current best hop
No
next hop metadata = final flowlet hop
Arrival time made equal to current time.
Yes
$U_{local} > U_{probe}$
Yes
$U_{probe} = U_{local}$
No
$U_{probe} < U_{Min(hdr.dst)}$ or $Packet_{time} >$ threshold
No
Yes
$U_{Min(hdr.dst)} = U_{probe}$
best $Hop_{dst} = meta_{in\_port}$
Arrival time made equal to current time.
$U_{header} = U_{Min(hdr.dst}$

Hula Header Format

```
header_type hula_header {
  fields {
          dst_tor : 24;
          path_util : 8;
    }
}

header_type metadata {
  fields{
          nxt_hop : 8;
          self_id : 32;
          dst_tor : 32;
     }
}
```

Fig. 2.1 Hula Logic

HULA is therefore, Scalable to large topologies, Adaptive to network congestion, Proactive to network failure because of probes and Programmable in P4.

2.3.2 CONGA: Distributed Congestion-Aware Load Balancing for Datacenters

Introduction:

Datacenters handle a massive amount of data traffic supporting a variety of applications. To ensure smooth operations, it's crucial to have an efficient way to distribute this traffic across the network. Traditional load balancing methods like ECMP often struggle to handle network imbalances and sudden traffic changes, especially when parts of the network fail. This can lead to congestion and delays. CONGA is a new load balancing approach designed specifically for datacenters. It works within the network itself and has several key advantages.

- **Congestion-Aware:** Actively monitors congestion levels across the network.
- **Fast Reaction:** Redistribute traffic in microseconds, ensuring quick responses.
- **Easy to Use:** Uses existing network technologies and doesn't require complex changes.
- **Works with Any Protocol:** It doesn't care what kind of data (TCP, UDP, etc.) is being sent, and it won't require changes to existing protocols like TCP.

How CONGA Works:

Leaf Switches: The bulk of CONGA's work happens in leaf switches (network switches at the edge of the datacenter fabric.

Congestion Tracking DREs: Each fabric link uses a Discounting Rate Estimator (DRE) for simple congestion measurement.

Local vs. Remote Metrics: Leaf switches keep track of both local congestion on their own uplinks and **remote** congestion further down the path toward the destination.

Feedback from Destination: Destination leaf switches send congestion information back to the source switches, helping them understand the full path's congestion picture.

Load Balancing Decisions: When a new flowlet (a subset of a larger data flow) starts, the source leaf switch picks the path with the least congestion, combining both local and remote congestion information.

Flowlet Table: Leaf switches track active flowlets to ensure packets in the same flowlet follow the same path, avoiding packet re-ordering.

Key Parameters:

Discounting Rate Estimator (DRE) :
The DRE is a simple module used to measure the load on a network link. It maintains a register, X, that follows these rules:
For each packet sent, X is incremented by the packet's size in bytes. Periodically (every T_{dre}), X is decreased by a factor of α (between 0 and 1): $X \leftarrow X \times (1 - \alpha)$.
This register's value, X, is roughly proportional to the traffic rate, R, on the link: $X \approx R \cdot \tau$, where $\tau = T_{dre}/\alpha$. Essentially, the DRE acts as a low-pass filter on packet arrivals with a

rise time of τ. To get the congestion metric, $X/C\tau$ is quantized to 3 bits (where C is the link speed).
The DRE algorithm shares similarities with the Exponential Weighted Moving Average (EWMA). However, the DRE offers two advantages:
Simpler implementation: Requires only one register, compared to EWMA's two.
Faster response: Reacts immediately to traffic bursts while still remembering past traffic patterns.

Limitations:

Communication Overhead: CONGA's distributed nature and information exchange requirements can introduce additional communication overhead within the network.

Scalability Challenges: In extremely large or highly dynamic networks, CONGA's decentralized approach may potentially impact performance.

Chapter 3

Working with P4

3.1 Hands on in P4 (IP based forwarding)

In this section, we describe how a basic p4 program for IP based forwarding can be loaded on p4 switch, using mininet.

3.1.1 Functionalities

- Parsing Ethernet headers and IPv4 headers from packet.
- Find the destination from the IPv4 routing table.
- Update source and destination MAC addresses in packet.
- Decrement time to live field in the header while forwaring.
- Set the egress port.

3.1.2 Pseudo P4 code snippets

Parser

```
Function PacketParser(packet in InPacket, out headers Header, inout metadata
    MetaData, inout standard metadata t StandardMetadata):
        Start
            transition ParseEthernet;
        ParseEthernet
            InPacket.extract(Header.ethernet);
            transition select(Header.ethernet.etherType) ;
                TYPE IPV4: ParseIPv4;
                default: accept;
        ParseIPv4
            InPacket.extract(Hdr.ipv4);
            transition accept
```

Ingress Control

```
Function IngressControl(inout headers PacketHeader, inout MetadataMeta,
    inout standard metadata t standard metadata):
        action DropPacket
            mark to drop(standard metadata);
        action forwardIPv4
            standard metadata.egress spec = egressPort;
            PacketHeader.ethernet.srcAddr = PacketHeader.ethernet.dstAddr;
            PacketHeader.ethernet.dstAddr = destinationAddr;
            PacketHeader.ipv4.ttl = PacketHeader.ipv4.ttl - 1;
        table ipv4Lookup
            key = { PacketHeader.ipv4.dstAddr: lpm; };
            actions = { forwardIPv4; DropPacket; NoAction; };
            size = 1024;
            default action = DropPacket ();
        apply
            if (PacketHeader.ipv4.isValid()) {
                ipv4Lookup.apply();
            }
```

3.1.3 Create topology using mininet

Mininet allows the creation of a realistic virtual network, running real kernel, switch and application code, on a single machine. We have a used python p4utils to create mininet network. Below is the code snippet for various operations.

```
from p4utils.mininetlib.network_API import NetworkAPI
net = NetworkAPI() # Creating network
net.addP4Switch('s1') # P4 Switch
net.addHost('h1') # Adding host
net.setP4Source('s1',p4code.p4') # configuring P4 code
net.addLink('s1', 'h1') # adding link
net.setIntfPort('s1', 'h1', 1) # adding link
```

We created the following topology using mininet.

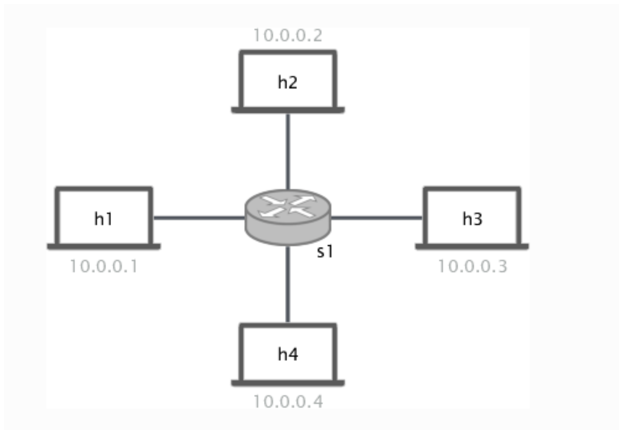


Fig. 3.1 Network Topology

3.1.4 Control Plane Configuration

After setting up a functional network topology with configured P4 switches, the next step is to populate the data plane with forwarding information to enable connectivity. To initiate the Thrift client and establish a connection to s1, it's necessary to have the IP address and port information for its Thrift server. In our scenario, the IP address is 127.0.0.1, and the port is 9090 (the default port).

CLI Command to start Thrift server

```
simple_switch_CLI --thrift-port 9090 --thrift-ip 127.0.0.1
```

Add entry to a match table

Table entry format

```
table_add <table name> <action name> <match fields> => <action parameters>
```

Added following entries in table MyIngress

```
table_add MyIngress.ipv4_forward {h.ipv4} => {h.mac} {port of link h->s}
```

This is how we can load and run a p4 program on p4-switch using mininet. Now, through mininet-cli we can xterm into any of the device of our topology. We can then, successfully send packets from host h1 to host h2 via switch S1.

Chapter 4

Load Balancing in Data Plane

P4, data plane programming language, is emerging as a powerful tool for network management. Unlike fixed hardware, P4 allows to program the switch itself for custom algorithms, giving you more control and flexibility. This programmable nature also allows for stateful load balancing, ensuring packets from a client consistently reach the chosen VM. Furthermore, P4 programs run directly on the switch hardware, bypassing software and reducing latency. However, P4 is still a developing technology with limitations in hardware support and programming complexity. Despite these challenges, P4 offers a powerful and adaptable approach to load balancing for the future of networks.

4.1 Design Challenges:

- Dynamic Weight Adjustment: Implementing dynamic weight adjustment mechanisms to adapt to changing workload conditions poses a challenge. The load balancer must dynamically adjust weights based on real-time resource utilization metrics while minimizing overhead and ensuring smooth operation.

- Static or Controller Dependent Dynamic Updates: Traditional hardware load balancers offer limited flexibility. Heavy reliance on the controller for sending and received updates can become a bottleneck if the controller is overloaded. VMs should take responsibility for reporting their metrics, potentially reducing the controller's workload. Updates might be faster as VMs can directly communicate with the host, potentially reducing communication overhead.

- Failure Handeling: Implementing proactive failure handling mechanisms to detect and respond to VM failures poses a challenge. The load Balancer should not route any new traffic to the failed VM, inorder to ensure availability of the system.

- Programmability: Developing hardware load balancers is cumbersome. Their fixed functionality ("one-size-fits-all") requires network operators to deploy them as it is. Any desired modifications or new features necessitate waiting for the next hardware revision, often years away.

- Flowlet load balancing: In dynamic environments, where workload conditions constantly change, static allocation of flows to virtual machines (VMs) may not remain optimal over time. Therefore, a mechanism for dynamically reassigning flows based on real-time metrics is essential to ensure efficient resource utilization. This can be done by setting a timeout threshold for each flow.

Following we discuss our approach towards implementing an efficient load balancing algorithm in virtualized envoronment.

The approach is to distribute traffic to VMs, based on some metrics collected from VMs (metrics like cpu utilization, storage space, network etc.) This load balancing algorithm is implemented over a p4 switch. The p4 switch's one interface would be connected to clients, where the incoming request from clients would reach. The other interfaces of the switch would be connected to VMs. The virtual p4 switch would distribute the traffic to the better VM, according to the load balancing algorithm. The approach has 4 major tasks to be solved, apart from the important task of finding appropriate load balancing algorithms.

4.2 Approach

4.2.1 NATing:

We created a topology as specified above in our system and loaded the p4 switch with a basic ip based forwarding algorithm. We were successfully able to forward the packets between client and vms. We could also forward tcp traffic between the client and vms. Then our major task was to implement a NATing algorithm within the p4 switch. This is needed, since the client would only be communicating with a VIP (virtual IP address) which is the ip of the switch's interface connected to the client. Switch will further load balance the requests and forward the clients' packets to a DIP: ip of the appropriate vm. The vms are therefore behind the nat. The approach here was to change the destination ip and mac addresses (src + dst) according to the vm whenever a request comes from client side, and change the source ip and mac address (src + dst) to that of host, whenever a pkt comes from vm. We implemented this in match action tables, as required for the p4 program. We were able to send icmp as well as tcp traffic to the vm behind nat by using switch's ip address (switch has virtual ip (vip) address), from the interface connected to the client.

Following is the table and the corresponding match actions, to identify the direction of the packets for NATing:

1. For VM to Client:

```
action change_src(bit<32> src_ipv4_addr, bit<48> src_mac_addr,
    bit<32> dst_ipv4_addr, bit<48> dst_mac_addr){

    hdr.ipv4.srcAddr = src_ipv4_addr;
    hdr.ipv4.dstAddr = dst_ipv4_addr;
    hdr.ethernet.srcAddr = src_mac_addr;
```

```
        hdr.ethernet.dstAddr = dst_mac_addr;
    }
```

2. For Client to VM:

```
    action change_dst(bit<32> dst_ipv4_addr, bit<48> dst_mac_addr, b
        it<48> switch_interface_mac){

        hdr.ethernet.srcAddr = switch_interface_mac;
        hdr.ipv4.dstAddr = dst_ipv4_addr;
        hdr.ethernet.dstAddr = dst_mac_addr;
    }

    table get_addr {

        actions = {change_dst; change_src, drop;}
        key = {  standard_metadata.egress_spec: exact; }
        size = 512;
    }
```

However getting the tcp traffic flow between client and vms was trickier, since the connection was supposed to be done and maintained between client and one of the vm (which is the better one in terms of metrics at the moment the first packet was received from the flow). This leads to 2nd task- of maintaining connection:

4.2.2 Maintaining Connection:

Once the first packet comes from the client, and is directed to one particular vm, then all other subsequent packets from the same client should be directed to the same vm, regardless of what the load balancing algorithm suggests as the best vm, at that time. In order to do this, we would need to maintain a table in the switch, which tells if the packet from the same flow was received earlier. So for this we are planning to make use of stateful memory units in p4 like the registers, to store the entries related to each flow and the corresponding vms assigned. In order to uniquely identify each flow, we would calculate the FlowId, which is the csum16 hash of 5-tuples (with one of the vm as destination). Thus whenever a packet enters the switch, it will calculate the flow ID, and check if any entry for the connection is already present in the connection table, if present it will forward the packets, to the vm assigned. If the entry is not present for the flowID, our load balancing algorithm would find the best vm, and add an entry corresponding to this flow in the connection table. Moreover, if a connection is terminated by the client, it is p4s responsibility to remove the entry corresponding to that vm from the switch's memory.

Following is the code snippet for calculating the flowID and assigning a VM to the flow:

```
bit<16> flow_id;
hash(flow_id, HashAlgorithm.csum16, (bit<16>)0, {
    hdr.ipv4.srcAddr,
    hdr.ipv4.dstAddr,
    hdr.tcp.srcPort,
    hdr.tcp.dstPort,
    hdr.ipv4.protocol},(
    bit<16>)65535);
}

// after applying load balancing algorithm, the egress_spec would
// be set equal to the port number of the appropriate VM
flowId_to_is_assinged.write((bit<32>) flow_id, 1);
flowId_to_port_assinged.write((bit<32>) flow_id,
                (bit<9>) standard_metadata.egress_spec);
```

Since p4 does not support key value storage and lookups can not be done based on key, the entries can only be stored in the form of an array, with indices. P4 provides the way of modifying any element in the array at any index, through read-write operation. Another way this can be done was by constructing a match action table with the flow id as key and value as vm. But this would need the controllers' interference. If we in future, decide to deploy this on smartNic, the better approach would be to minimize the controllers interference, in order to reduce the cpu cycles. Thus we decided to go with the 1st approach, of storing the entries in the array form.
With this approach, the optimal way for lookup into the array is by using the concept of bloom filters.[13]

4.2.3 Breaking Flows into Flowlets:

Considering the fact that all flows would not remain active for all the time, we could remove the entry of a flow, for which the last packet was sent more than tf (flowlet threshold) time back. [6] This not only helps to minimize the entries stored in the connection table, but also helps for better load balancing of the flows having less frequent packet traversal. It mitigates the problem of handling the traffic from such flows by only one vm permanently, as the utilization of it might go up higher in later time periods, and thus may not be optimal for such short flows. Thus flows would be divided into flowlets, its connection would be terminated and entry would be removed from the connection table, if no packet is seen to travel in either direction for more than flowlet threshold ($t_f lowletThreshold$) time.

Following code snippet depicts how the timestamp for the last seen packet for each flow is stored:

```
time_t cur_time = standard_metadata.ingress_global_timestamp;
flowId_to_last_updated.write((bit<32>) flow_id, cur_time);
```

4.2.4 How would the metrics reach the p4 switch?

In order to dynamically balance load, based on performance metrics of vms, we need the switch to get the data from each vms. We thought of 2 approaches for this:

- VMs will send the probes periodically to the host, with a probe header containing the data related to its performance metrics. This can be done as a startup program, which will run in vm as a background process, for the time VM is kept on.

  Code snippet depicting how the probe packet is parsed by the p4-program:

  ```
  header probe_t {
      bit<4>   id;
      bit<4>  cpu_percent;
      bit<4>  memory_percent;
      bit<4>  link_util;
  }
  transition select(hdr.ipv4.protocol) {
      42: parse_probe;
      6: parse_tcp;
      default: accept;
  }
  state parse_probe {
      packet.extract(hdr.probe);
      transition accept;
  }
  ```

- Using tools like prometheus [14] in host and gathering the important metrics from VMs. This data would then be analyzed by the host and the entry corresponding to the best vm (having minimum utilization) would be added by the controller into the p4 program. This could be done periodically. This way, whenever a client makes a request, it will be directed to the min utilized vm as per the view of the P4 switch, during that time period. PromQL is used to extract data from prometheus. The queries are shown in Chapter 7.

Both of the above approaches, comes with its own overhead. The main Load Balancing logic for both are explained in the next chapters.

4.2.5 Proposed Load Balancing Algorithms:

The 2 load balancing algorithms are discussed in further chapters:

- Probe Enhanced Weighted Round Robin Load Balancing in P4 (Chapter 6).
- Promethee-Prometheus Driven Load Balancing Using P4 (Chapter 7).

4.2.6 Handling Edge Cases in Load Balancing:

- **Idle Timeout Mechanism:**
  Consider the scenario where a packet from a flow is received after $T_{flowThreshold}$ time. If the difference between the current timestamp and the last updated timestamp for that flow is greater than $T_{flowThreshold}$, then we sent reset packets to both server and client. This is done using clone_packet functionality of p4 switch. We also remove the connection entry from p4. This is implemented, to better load balance the occasional-flows. The server which was better at the time of allocation, may not still be the better one for handling the flow. Therefore idle timeout mechanism is used to age out the older unused connection entries. This also helps in regaining the register memory used for stale entry storage. This mechanism helps identify and redirect older flows (idle flows) to different VMs, promoting better resource utilization and performance optimization. We have considered $T_{flowThreshold}$ to be very large (of the power of 10^3s). This is randomly chosen. It primarily depends on the application running on the server.

  Following code snippet shows that, if the packet is not received from a flow for $t_{flowThreshold}$ time, then we terminate the session, by sending reset packets both to server and client. Clone of the packet is used to send 2 packets from switch- one to client and other to sender. Cloned packet is directly forwarded to egress pipeline, along with original packet.

```
#define PKT_INSTANCE_TYPE_INGRESS_CLONE 1

action clone_packet(){
    const bit<32> REPORT_MIRROR_SESSION_ID = 500;
    clone(CloneType.I2E, REPORT_MIRROR_SESSION_ID);
}

time_t last_time;
flowId_to_last_updated.read(last_time, (bit<32>) flow_id);

if(cur_time - last_time > TIME_THRESHOLD) {

    clone_packet();
    hdr.tcp.rst = 1;
    flowId_to_port_assigned.read(standard_metadata.egress_spec,
                                (bit<32>) flow_id);

    flowId_to_is_assigned.write((bit<32>) flow_id, 0);
    flowId_to_port_assigned.write((bit<32>) flow_id, 0);
}
```

```
In egress pipeline check:
    if(standard_metadata.instance_type ==
                PKT_INSTANCE_TYPE_INGRESS_CLONE){

        set reset bit 1
        hdr.tcp.rst = 1;

        //swap src, dst -> ip, mac, port
        bit<48>temp;
        temp = (bit<48>)hdr.ipv4.dstAddr;
        hdr.ipv4.dstAddr = hdr.ipv4.srcAddr;
        hdr.ipv4.srcAddr = (bit<32>)temp;

        temp = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = hdr.ethernet.srcAddr;
        hdr.ethernet.srcAddr = temp;

        temp = (bit<48>)hdr.tcp.dstPort;
        hdr.tcp.dstPort = hdr.tcp.srcPort;
        hdr.tcp.srcPort = (bit<16>)temp;

        //egress port 1 : sending reset to client
        standard_metadata.egress_spec = 1;
    }
```

- **Failure Handling:**
  In case a VM fails, we would not receive the real time metrics from the VM, indicating a potential failure or outage, the load balancer immediately ceases to assign new requests or flows to that VM. This proactive approach helps prevent routing traffic to compromised or malfunctioning VMs, ensuring high availability and reliability of the system. The threshold time $T_{probeThreshold}$ depends on the average response time of the server, and thus also depends on the application running on the server. It should ideally be some x times the average response time. This is needed, so that the load balancer does not mistakenly identify the VM as failed, if the probe gets delayed due to congestion. The exact value should be determined experimentally. We currently have assumed the $T_{probeThreshold}$ = 2*probeFreq (= 10s, considering 5s as the ProbeFreq).

Following code snippet depicts, how to detect a failed VM. Moreover, if the failed VM is detected, we resubmit the packet to ingress pipeline, to assign the next available VM.

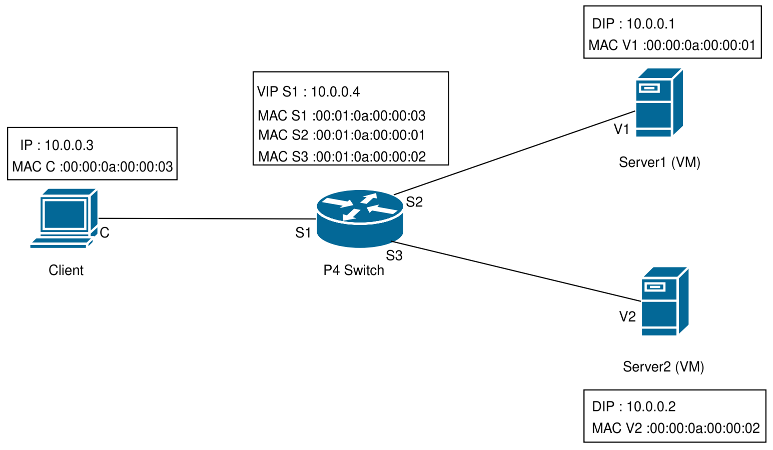
```
time_t cur_time = standard_metadata.ingress_global_timestamp;
time_t last_update_time;
```

```
probe_last_updated.read(last_update_time, (bit<32>) _cur_server_id);

if((cur_time-last_update_time) > PROBE_TIMEOUT) {
    has_failed.write((bit<32>)_cur_server_id, 1);
    check_failed = 1;
}

if(check_failed == 1) { //check if the next VM is availble
    resubmit_preserving_field_list((bit<8>)1);
}
```

4.3 Here's a basic Setup for Load Balancer in mininet

4.3.1 Our Topology

This network was created using Mininet for emulation purposes. Once we finalize the design and testing, we will try it on an actual virtual machine.

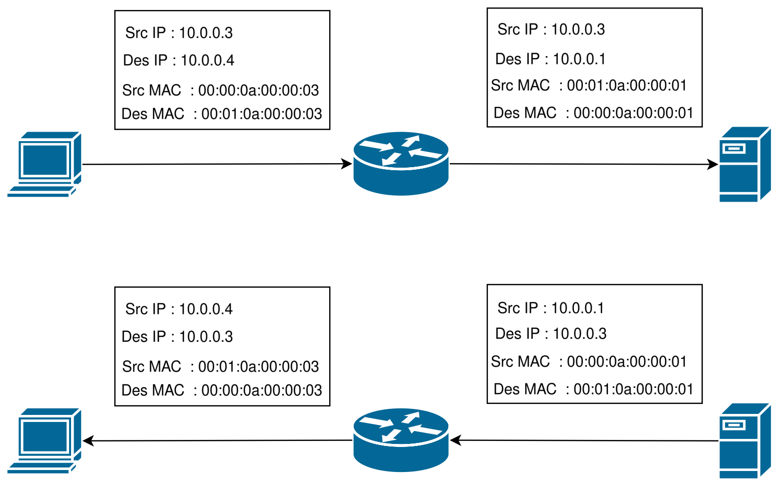


4.3.2 NATing the servers from external clients

From the client's perspective, assigned the VIP to S1 as follows by running following command in client.

```
sudo arp -s 10.0.0.4 00:01:0a:00:00:03
```

Then natting is done as per following figure.


Src IP : 10.0.0.3
Des IP : 10.0.0.4
Src MAC : 00:00:0a:00:00:03
Des MAC : 00:01:0a:00:00:03
Src IP : 10.0.0.3
Des IP : 10.0.0.1
Src MAC : 00:01:0a:00:00:01
Des MAC : 00:00:0a:00:00:01
Src IP : 10.0.0.4
Des IP : 10.0.0.3
Src MAC : 00:01:0a:00:00:03
Des MAC : 00:00:0a:00:00:03
Src IP : 10.0.0.1
Des IP : 10.0.0.3
Src MAC : 00:00:0a:00:00:01
Des MAC : 00:01:0a:00:00:01

4.3.3 Sending ICMP packets

ICMP relies on the IPv4 layer. So, we calculated the IPv4 checksum as follows in P4 switch.

```
update_checksum(
hdr.ipv4.isValid(),
    { hdr.ipv4.version,
      hdr.ipv4.ihl,
      hdr.ipv4.diffserv,
      hdr.ipv4.totalLen,
      hdr.ipv4.identification,
      hdr.ipv4.flags,
      hdr.ipv4.fragOffset,
      hdr.ipv4.ttl,
      hdr.ipv4.protocol,
      hdr.ipv4.srcAddr,
      hdr.ipv4.dstAddr
    },
hdr.ipv4.hdrChecksum,
HashAlgorithm.csum16);
```

4.3.4 TCP connection between client and server

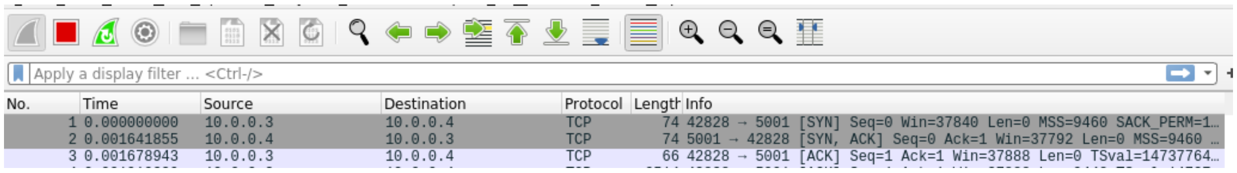
The L4 payload length was calculated and stored in the switch's metadata for inclusion in the checksum calculation.

```
meta.l4_payload_length = hdr.ipv4.totalLen - (((bit<16>)hdr.ipv4.ihl)*4);
```

TCP checksum [15] is calculated as follows in P4 switch.

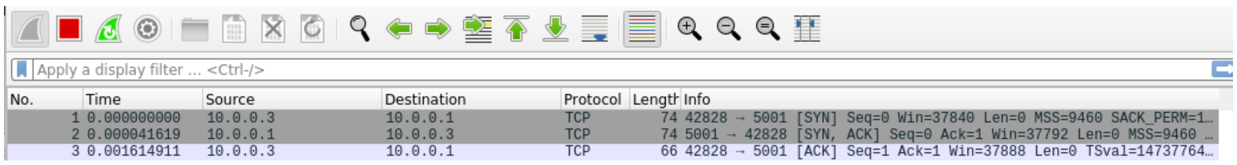
```
update_checksum_with_payload(
hdr.tcp.isValid(),
    {   hdr.ipv4.srcAddr,
        hdr.ipv4.dstAddr,
        8w0,
        hdr.ipv4.protocol,
        meta.l4_payload_length,
        hdr.tcp.srcPort,
        hdr.tcp.dstPort,
        hdr.tcp.seqNo,
        hdr.tcp.ackNo,
        hdr.tcp.dataOffset,
        hdr.tcp.res,
        hdr.tcp.cwr,
        hdr.tcp.ecn,
        hdr.tcp.urg,
        hdr.tcp.ack,
            hdr.tcp.psh,
            hdr.tcp.rst,
            hdr.tcp.syn,
            hdr.tcp.fin,
            hdr.tcp.window,
            16w0,
            hdr.tcp.urgentPtr
        },
        hdr.tcp.checksum,
        HashAlgorithm.csum16);
```

Following is the TCP handshake on wireshark. We sent TCP traffic from client to server using linux iperf command.



```
No.   Time          Source      Destination  Protocol Length Info
    1 0.000000000   10.0.0.3    10.0.0.4     TCP         74 42828 → 5001 [SYN] Seq=0 Win=37840 Len=0 MSS=9460 SACK_PERM=1…
    2 0.001641855   10.0.0.4    10.0.0.3     TCP         74 5001 → 42828 [SYN, ACK] Seq=0 Ack=1 Win=37792 Len=0 MSS=9460 …
    3 0.001678943   10.0.0.3    10.0.0.4     TCP         66 42828 → 5001 [ACK] Seq=1 Ack=1 Win=37888 Len=0 TSval=14737764…
```

Fig. 4.1 Client interface connected to switch



```
No.   Time          Source      Destination  Protocol Length Info
    1 0.000000000   10.0.0.3    10.0.0.1     TCP         74 42828 → 5001 [SYN] Seq=0 Win=37840 Len=0 MSS=9460 SACK_PERM=1…
    2 0.000041619   10.0.0.1    10.0.0.3     TCP         74 5001 → 42828 [SYN, ACK] Seq=0 Ack=1 Win=37792 Len=0 MSS=9460 …
    3 0.001614911   10.0.0.3    10.0.0.1     TCP         66 42828 → 5001 [ACK] Seq=1 Ack=1 Win=37888 Len=0 TSval=14737764…
```

Fig. 4.2 Server interface connected to switch

Actual VM-server and client setup is explained separately in each of the corresponding chapters of the two Load Balancing approaches.

Chapter 5

Probe Enhanced Weighted Round Robin Load Balancing in P4

5.1 Overview

This Load Balancing algorithm meant for virtualized environment, operates within the p4 switch, managing resource allocation across multiple virtual machines (VMs) to optimize system performance and ensure efficient utilization of computing resources.
The load balancer collects probes from VMs at regular intervals. These probes include CPU, memory, and link utilization metrics, encapsulated in a custom header and transmitted to the switch. The load balancer calculates a weighted mean based on pre-determined weights, normalized to a range of 0-12. Dynamic weights adjust according to resource usage. It employs Weighted Round Robin load balancing for request distribution and includes an idle timeout mechanism to optimize resource usage. Additionally, the load balancer stops assigning requests to failed VMs.

5.2 Flow Chart

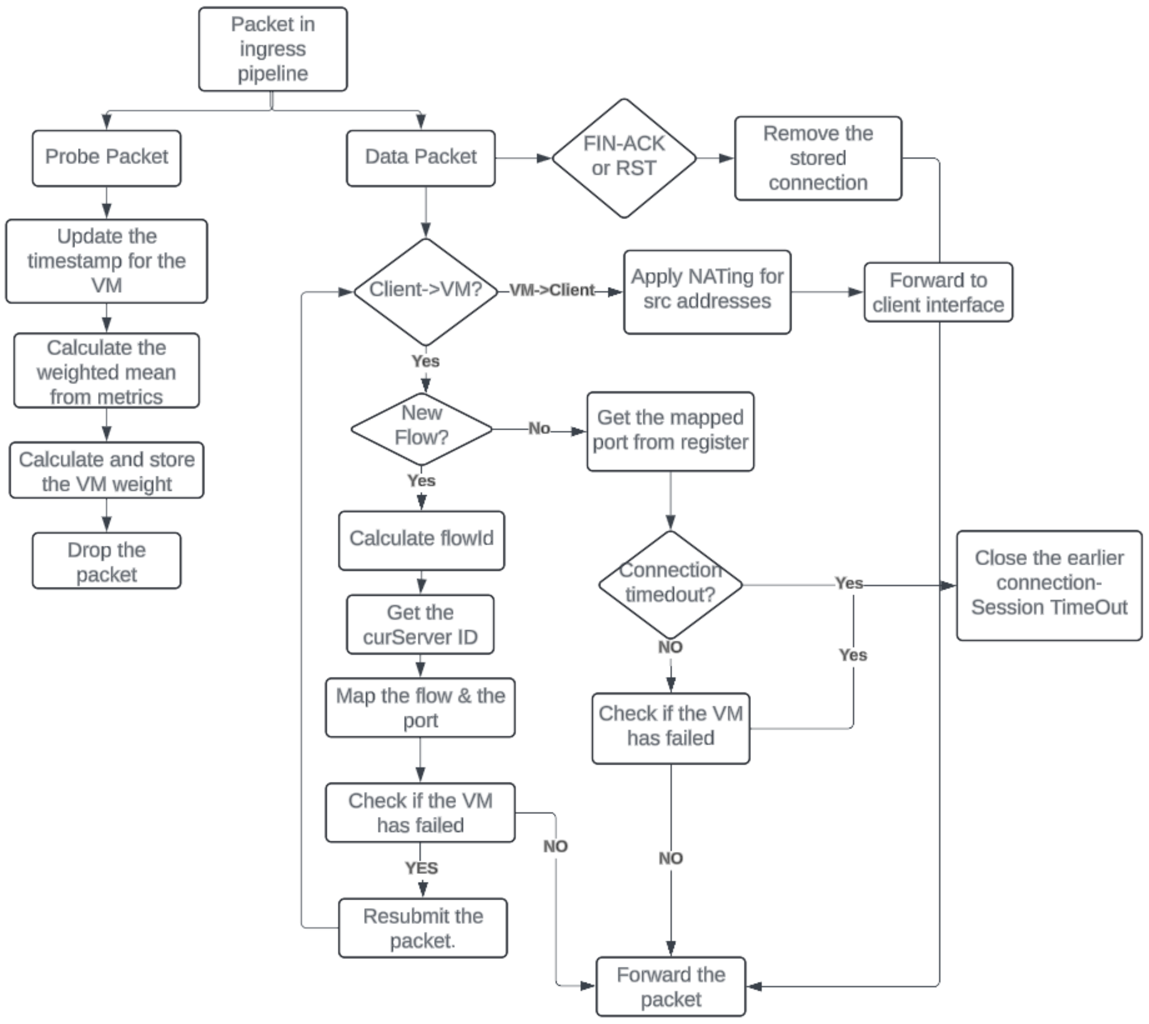
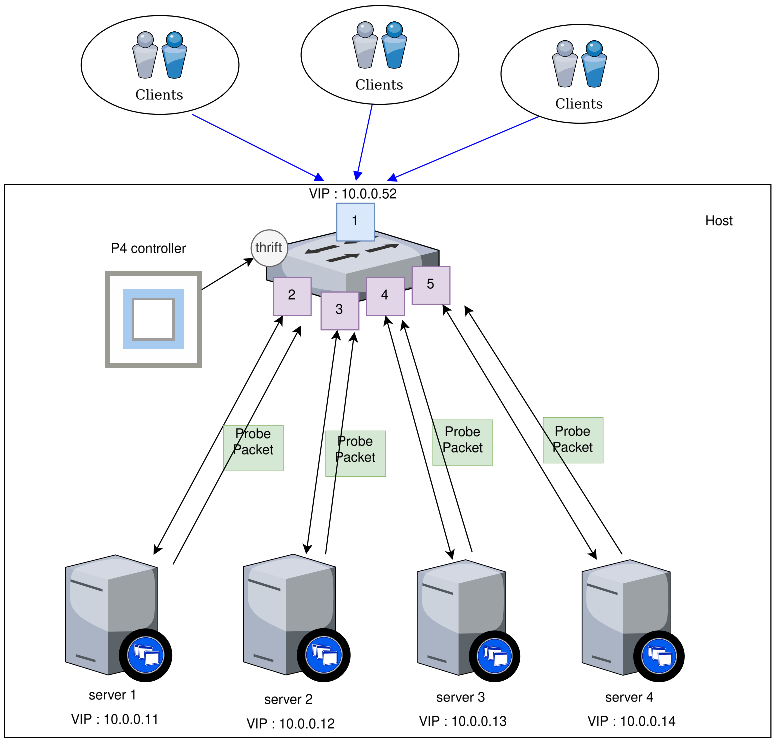

Packet in ingress pipeline
Probe Packet
Data Packet
FIN-ACK or RST
Remove the stored connection
Update the timestamp for the VM
Client->VM?
VM->Client
Apply NATing for src addresses
Forward to client interface
Calculate the weighted mean from metrics
Yes
New Flow?
No
Get the mapped port from register
Calculate and store the VM weight
Yes
Drop the packet
Calculate flowId
Connection timedout?
Yes
Close the earlier connection-Session TimeOut
Get the curServer ID
NO
Yes
Map the flow & the port
Check if the VM has failed
Check if the VM has failed
NO
NO
YES
Resubmit the packet.
Forward the packet

Fig. 5.1 WorkFlow

5.3 Implementation

5.3.1 Setup



5.3.2 Probe with Custom Header

A probe packet is a packet of size (60 bytes) which contains a custom header along with the normal Ethernet and IP headers. The protocol field of IP header is set to 0x42 for the probe packets. The custom header comprises of 4 fields of 4 bits each:

- **Id:** Each VM has a unique id, which is sent along with each probe originating from that VM. It is used to identify the VM from probe packet.

- **CPU (in range of 10)** left unutilized in VM.
  The CPU is responsible for processing incomming requests, encoding streams, managing network connections and other tasks. By monitoring CPU utilization, the load balancer can identify servers that are experiencing high processing demands and distribute incoming requests to less burdened servers to ensure smooth functioning of servers.

- **Memory (in range of 10)** left unutilized in VM.
  Memory (RAM) is crucial for storing and accessing data required for streaming operations, such as buffering content, caching frequently accessed data, and managing connections. High memory utilization may indicate that the server is reaching its capacity to store and manage data effectively. By monitoring memory utilization, the load balancer can ensure that requests are distributed evenly among servers with

available memory capacity, preventing performance degradation due to memory exhaustion.

- **Link Utilization (in range of 10)** which is untilized.
  The network link between the server and clients carries the video data packets. High link utilization can indicate that the network link is congested, potentially leading to packet loss, latency, and reduced quality for clients. By monitoring link utilization, the load balancer can identify congested links and route traffic to less congested paths or servers, ensuring optimal network performance for clients.

How are these metrics calculated?

Python's psutil library is used to calculate CPU utilization, memory utilization, and network I/O (bytes sent and received) on each of the servers. Running psutil inside a VM provides visibility into the resource usage within the context of the virtualized environment. psutil employs the 'times' system call to retrieve CPU times (idle, user, system, interrupt, and softirq) in clock ticks. The difference in these values between measurements is divided by the total clock ticks in a second and the number of cores to compute CPU utilization. For memory usage information, psutil employs the /proc/meminfo file (Linux) or system-specific APIs (other OSes). psutil utilizes the /proc/net/dev or /sys/class/net interfaces (depending on the OS) to access network device statistics. This typically includes total, free, used, buffers, cached, and swap memory. The difference in bytes sent and received between measurements provides the network traffic information for the specified link (here the link connecting the VM to the switch is considered). This sent and received bytes over a period of time(30s each) is used along with the link bandwidth to calculate the link utilization. Therefore, all the three metrics are collected in percent and sent through probes(in range of 0-10).

5.3.3 Weighted Mean Calculation and Normalization

The Probes are sent after every $T_{ProbeFreq}$ sec (referred as probe frequency), periodically by each of the server VMs. Once the probes reach the P4 switch, the custom header is parsed and the metrics are extracted by p4. They are then multiplied by their corresponding weights, and weighted mean is calculated. The weights for the metrics can be calculated using AHP algorithm (explained in section 6.2.2). Weights assigned are link utilization: 6, CPU: 3, Memory: 1. This weighted mean lies in range of 0-100, which is then normalized to a range of 0-12, in order to provide finer granularity in distributing traffic among servers. This normalized mean is used to weigh the respective VM. The weight is updated for the VM, each time the probe for that VM is received. This way, the weights assigned to each VM are dynamic in nature, meaning they can change over time based on the fluctuations in resource utilization. This dynamic adjustment allows the load balancer to adapt to changing workload conditions and optimize resource allocation accordingly.

5.3.4 Weighted Round Robin Load Balancing

The weights for VMs are considered to be proportional to the capacity of VM, i.e. the connections that can be made to the VM. For each incoming flow, a weighted round robin is implemented in switch using 2 registers, one (cur_server_id) which stores the id of the VM to which the next flow is to be assigned, and the second (vm_counter) stores the capacity of the VM, the first register points to. For example, suppose the weight of vm2 is 4, and the values of registers are cur_server_id = 2, vm_counter = 1, then the incoming four connections would be sent to vm2, with the vm_counter value changing to 2,3,4 each time the connection is assigned. After that, the cur_server_id would shift to vm3, and so on.

What if weight comes out to be 0? If weight of any VM is 0, no flow is assigned to that VM. The cur_server_id value is incremented and the packet is resubmitted to the ingress pipeline until it finds a valid VM (one whose weight is >0).

5.3.5 Probe overhead

The probes from the VMs should be sent frequently enough so that the load balancers perform optimally in the virtualized environment, with varying resource utilization. However, the frequency should not be too high to overwhelm the network.

The link bandwidth was found from the /sys file system in the VMs. the /sys/class/net/<interface name> has a speed file, which states the maximum bandwidth that interface can store. Each of the VM interfaces supported 1000Mbps bandwidth. Considering this as our bandwidth, and 5 as probe frequency, the probe overhead for each link connecting switch to the VM can be calculated as:

$$\begin{aligned} \text{Overhead} &= \frac{\text{ProbeSize} * 100}{\text{ProbeFreq} * \text{LinkBandwidth}} \\ &= \frac{60 * 8 * 100}{5 * 1000 * 1024 * 1024} \\ &\approx 0.00001\% \end{aligned}$$

Overall, this load balancer effectively orchestrates resource allocation and workload distribution across the virtualized environment, leveraging dynamic weighting mechanisms and proactive failure handling to optimize performance and ensure seamless operation under varying workload conditions. Following are few of the important observations made regarding this load balancer.

5.4 Evaluation

5.4.1 Setup

Our evaluation was conducted on a PC equipped with an 11th Gen Intel i5-1135G7 processor (8 cores) and 16 GB of RAM. We used VirtualBox to create four virtual machines (VMs). Each VM hosted a TCP server implemented using the Python socket library. These servers were configured to listen for new connections on port 5007. Upon receiving a message from a client, the server responds with a 5 MB data.

Client and Testing Methodology:

Each client transmits a 250-byte string to the server at a regular interval of one second and the server responds to it by sending back 5MB data. This transmission pattern was maintained for a duration of 30 seconds per client instance. To simulate a gradual increase in network load, clients were connected in intervals of 0.5 seconds. Upon completion, each client calculated and recorded its average, minimum, and maximum response times to a file.

VM Specifications:

Each virtual machine was configured with Ubuntu 16.04.7 LTS (Xenial Xerus) as the operating system. We allocated 2405 MB of base memory and a single processor to each VM, with an execution cap of 100%.

5.4.2 Observations

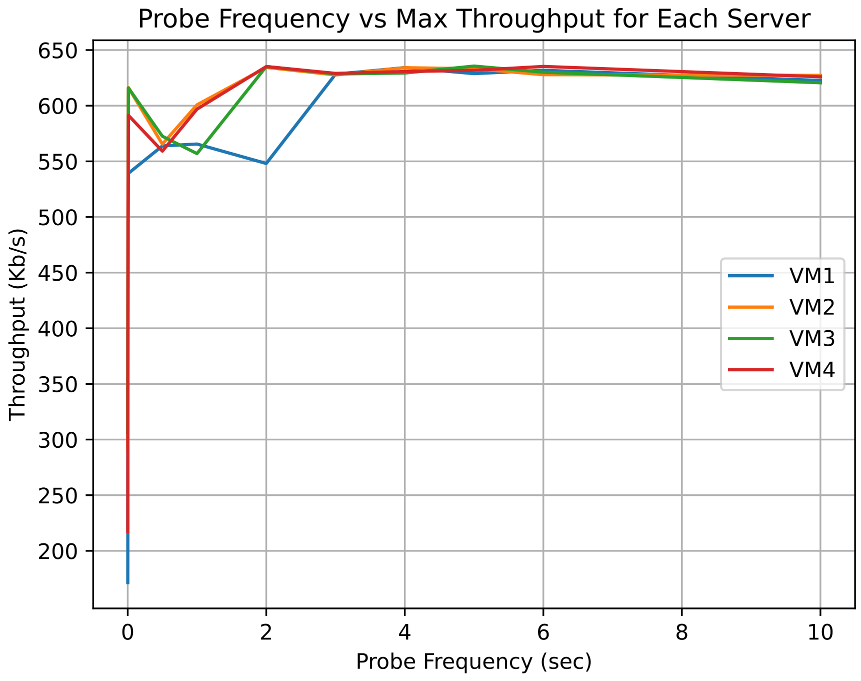


Fig. 5.2 Probe Frequency vs Server Throughputs

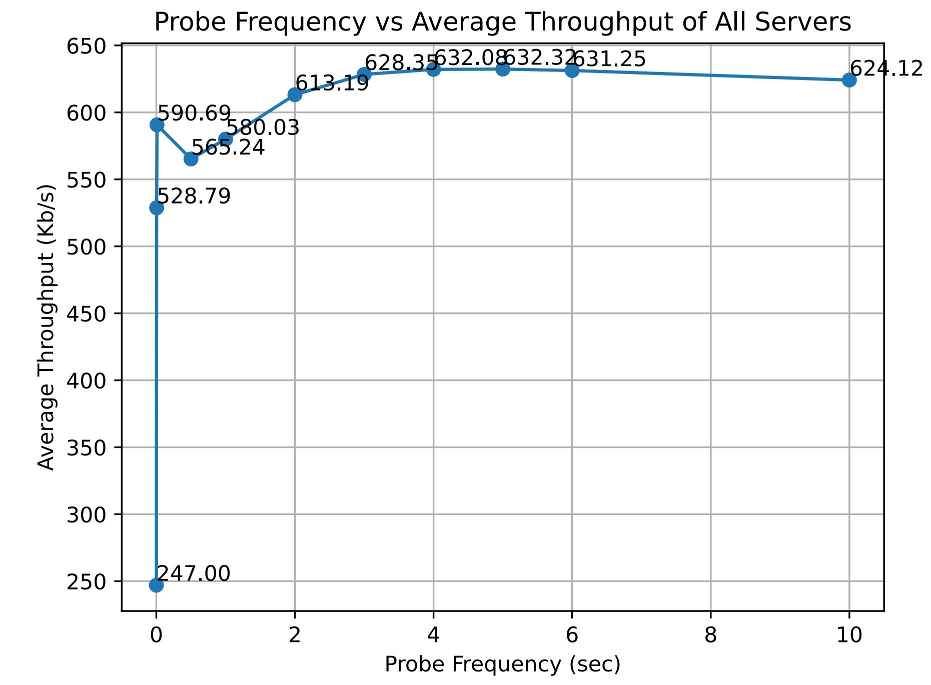


Fig. 5.3 Probe Frequency vs Average Throughput

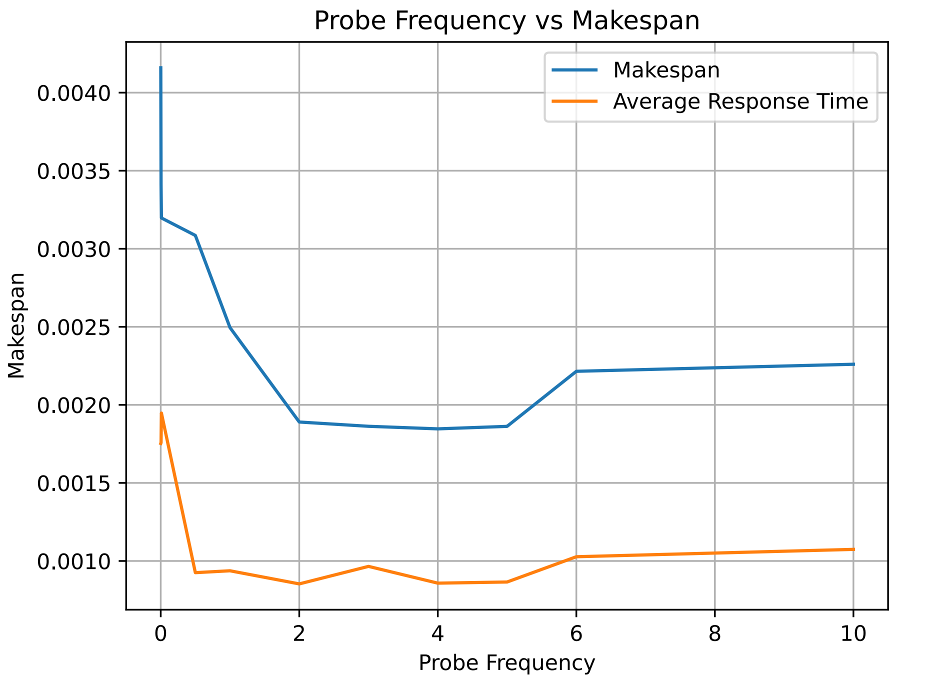


Fig. 5.4 Probe Frequency vs Makespan

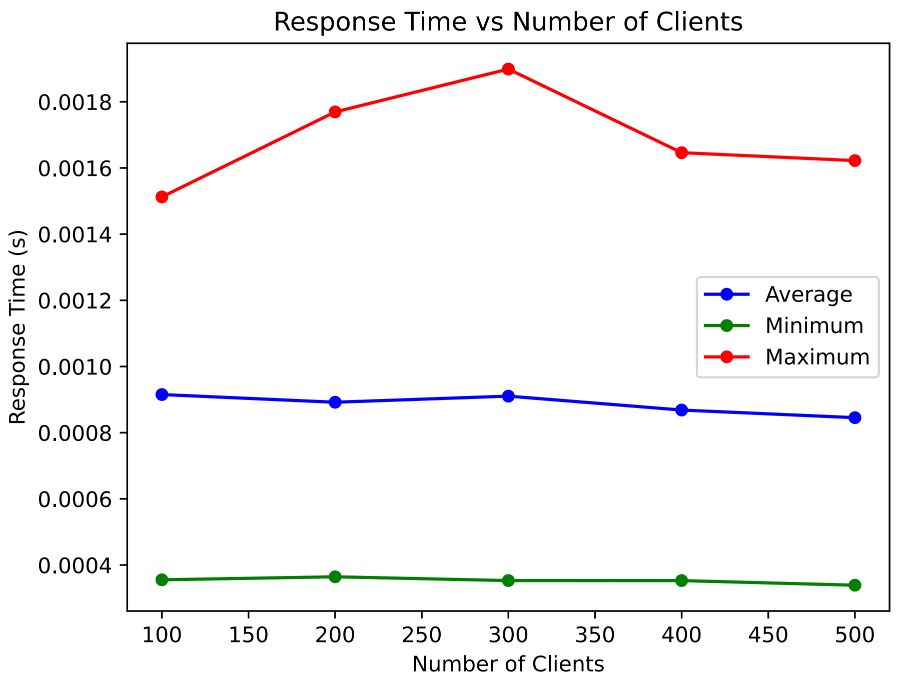


Fig. 5.5 Response Time vs Clients

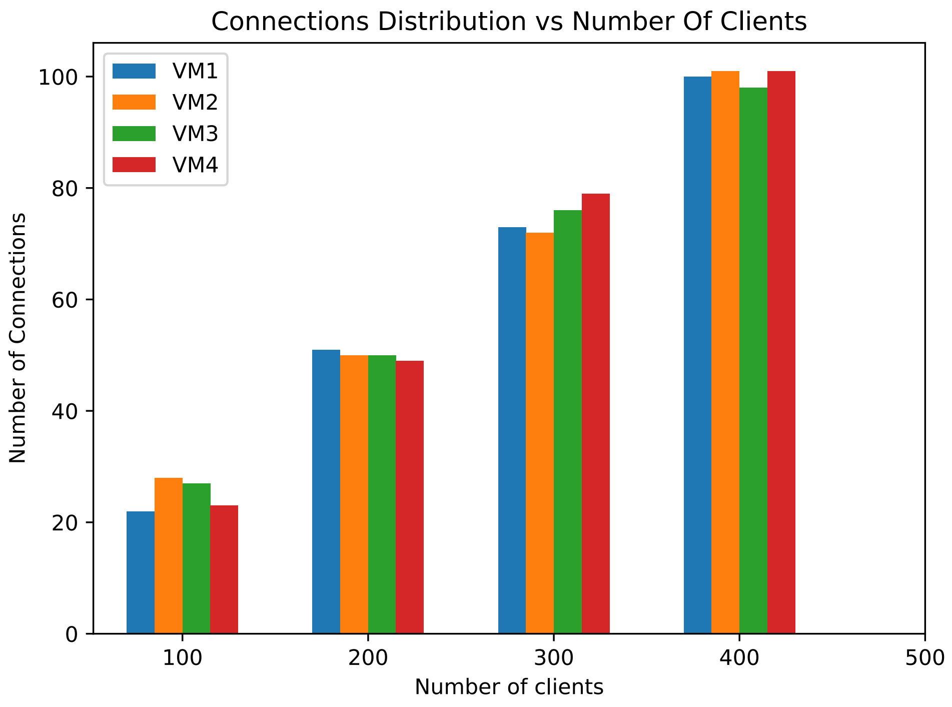


Fig. 5.6 Connections vs Clients

- As is visible from Figure 5.3, the throughput (without considering the traffic due to probes) is less for probe frequencies less than 1, however it increases and becomes roughly constant for all the probe frequencies greater than 4, with maximum of 632.31875 Kb/s for probe freq of 5 seconds. Thus, we can clearly see, that with more number of probe packets, the network can get congested, and therefore decrease the throughput.

- However we cannot keep the probe frequency too high, inorder to perform better load balancing. An optimal probe frequency would be the one which is the lowest among the time, where the network does not get overwhelmed, and therefore provides better throughput. We have chosen the probe frequency of 5 sec for further experiment.

- Makespan refers to the maximum time it takes for a single client request to be fully handled and a response returned. This would include: time spent traversing the load balancer, time to select the least loaded server, time for the request to reach the VM, time for the VM server to process the request and time for the response to travel back through the load balancer to the client. Figure 5.4, shows the graph between the Makespan and the probe frequency. Makespan is less for higher probe frequencies. A shorter makespan directly translates into faster response times from the clients' perspective. This is crucial for maintaining a responsive and satisfying user experience.

- Figure 5.5, plots the average, max and min response time as the number of client connections increase. We varied the number of connections in the range of 100-500. There doesn't seem to be much variation in the average response time of server even when the number of client connections were increased till 500.

- Figure 5.6, depicts the distribution of client connections by the load balancer, when varying number of clients connect to the server. During the experiment, no stress was loaded on any server, that is the reason for the connections to have got distributed more or less equally.

- If we put stress on one of the VM, and generate TCP traffic using iperf, the VM with stress, takes up less connections than the rest of the VMs. This validates our load balancing approach.

Next, considering the fact that the entire virtualized environment is set up in the same host machine, we thought of using prometheus tool to gather the VMs system metrics, instead of the VMs sending metrics through probes. Our setup and approach is discussed in the next section.

Chapter 6

Promethee-Prometheus Driven Load Balancing Using P4

6.1 Overview

A load-balancing approach for virtualized environments that takes into consideration the dynamic nature of resource utilization. This method integrates the Promethee II [16] algorithm and Analytic Hierarchy Process (AHP) [17] to dynamically select the optimal virtual machine (VM) based on real-time cpu, memory and network metrics by using prometheus node exporter. The system leverages P4-enabled switch to track connection states (SYN, FIN, RST), facilitating intelligent and proactive load distribution to enhance network performance and resource utilization.

The Analytic Hierarchy Process (AHP) is used to determine weights for decision criteria. It is used to break down decision-making problems into a hierarchy of criteria and alternatives. Through pairwise comparisons among criteria, AHP calculates weights representing their relative significance. This method is ideal for handling both qualitative and quantitative factors, and it calculates a consistency ratio to validate the reliability of the weights.

The Promethee II algorithm, used for ranking VMs, is a multi-criteria decision-making method used for ranking alternatives (VMs), making it suitable for load balancing. Promethee evaluates alternatives based on their performance across multiple criteria (cpu, memory and network metrics in our case). The Promethee II approach surpasses a simple weighted mean using AHP because it offers a more comprehensive decision-making process. While a weighted mean linearly aggregates scores, Promethee considers preference functions that allow for non-linear relationships between criteria differences and preference intensity. Additionally, Promethee's outranking principle evaluates alternatives based on their overall strengths and weaknesses across all criteria. This goes beyond a simple weighted average, providing a more robust and flexible method for determining the best virtual machine for load balancing.

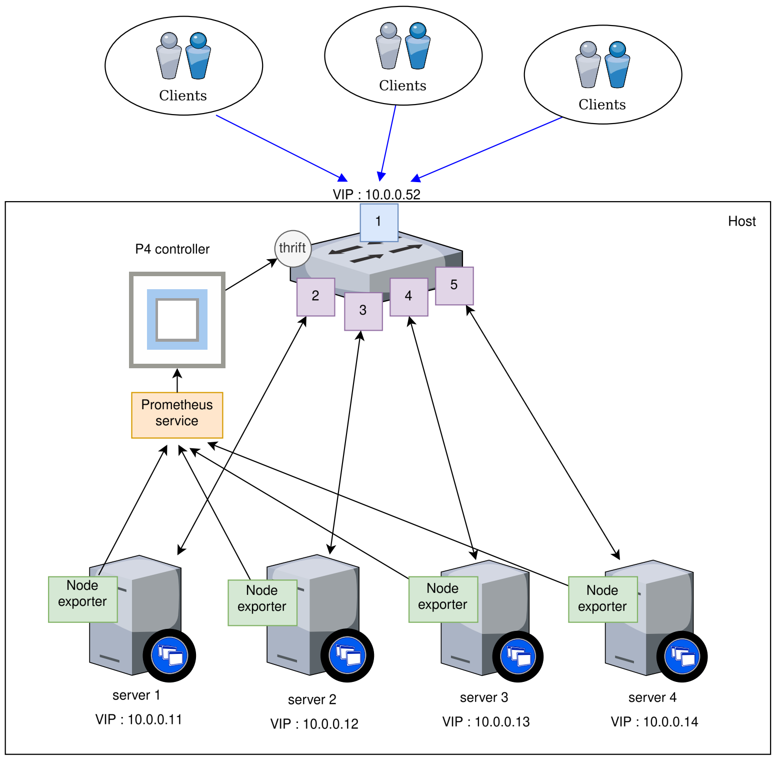
Prometheus is a monitoring tool, to collect system metrics from virtual machines (VMs).

To facilitate communication between Prometheus and individual VMs, we use port forwarding, a technique that tunnels data from a specific VM port to a designated port on the host machine. This enables efficient retrieval of VM metrics by Prometheus for further analysis.

The combination of Promethee and AHP proves particularly advantageous for load balancing. In a P4 switch, we have a **min_utilized** register that indicates the port associated with the virtual machine (VM) possessing the lowest current resource consumption. Upon the arrival of new flows, the switch set the egress port according to the value stored in this register. The register's value is dynamically updated by a controller that leverages the Promethee II algorithm to determine the most suitable VM for load balancing.

6.2 Implementation

6.2.1 Setup



A Node Exporter transmits data from each connected virtual machine (VM) to a Prometheus service through a single interface by utilizing port forwarding. For instance, on Server 1, a Node Exporter instance running on port 9100 forwards its data to port 9501 on the host. This is configured within the Prometheus configuration file (prometheus.yml). The following job configuration is added:

```
scrape_configs:
  - job_name: 'First_VM'
    metrics_path: /metrics
    static_configs:
      - targets: ['127.0.1.1:9501'] # This is our VirtualBox VM
```

With this setup, a P4 controller running on the host can periodically retrieve data from the Node Exporter service within each VM (port 9100). Each VM connects to a P4 switch via a separate interface, and the switch links to external clients.

6.2.2 Promethee II and AHP algorithm

The Promethee II algorithm has been implemented using Python's pymcdm library, and the AHP algorithm has been implemented using the ahpy library. For the Promethee II implementation, the standard **usual** preference function provided by pymcdm was utilized. We have used following judgment matrix to calculate weight using AHP by considering streaming service as the usecase. Therefore, We have prioritized utilisation network, followed by CPU, and then memory.

-	**cpu_load**	**ram_usage**	**network_usage**
cpu_load	1	4	1/2
ram_usage	1/4	1	1/6
network_usage	2	6	1

we obtained consistency ratio (CR) of 0.009 which is less than 0.1, indicating that the judgment matrix is consistent.

6.2.3 P4 Controller

The Python prometheus-api-client library was used to determine CPU load, RAM usage, and network usage (transmitted bytes). This was accomplished by formulating queries using **PromQL (Prometheus Querying Language)**. These queries enable the extraction of relevant metrics from the Prometheus.

```
Queries used

cpu load : avg_over_time(100 - (avg by(instance)(irate(
    node_cpu_seconds_total{mode="idle"}[1m])) * 100)[1m:])
# Explanation: Calculates the average CPU utilization percentage
over a 1-minute window, aggregated over all instances, by determining
the percentage of time the CPU was not in idle mode.

ram usage : (1 - (node_memory_MemFree_bytes + node_memory_Cached_bytes
    + node_memory_Buffers_bytes) / node_memory_MemTotal_bytes) * 100
# Explanation: Calculates the percentage of actively used RAM
by determining the ratio of non-free memory (used, cached, and
buffered) to total memory.

Network transmitted bytes : ((rate(node_network_transmit_bytes_total[1m])
    * 8) / (1000 * 1024 * 1024)) * 100
```

```
# Explanation: Calculates the average network transmission rate in megabytes
per second (MB/s), over a 1-minute window, adjusted for consistency of units.
```

The process periodically retrieves metrics from a Node Exporter at intervals of **tp seconds**. Subsequently, the Promethee II algorithm is applied to rank VMs according to available resources. The controller then updates the **min_utilised** register of a P4 switch, identifying the most suitable VM. This procedure make sure that subsequent flows are routed to the VM possessing the greatest available resources. We minimize writes in switch by conditionally updating min_utilized only when least utilized VM has changed.

What should be value of tp ?
The value of the polling interval (tp) must be carefully selected. A small tp could lead to the controller consuming excessive CPU cycles, potentially slowing down the switch. Conversely, a large tp could hinder the load balancing process from accurately reflecting real-time conditions and could delay effective responses to VM failures.

6.2.4 Visualize the metrics

We integrated Grafana and Prometheus with the node exporter to facilitate runtime monitoring of individual virtual machines. This provided insights into resource utilization and system health.

6.3 Observation

We have used same setup for evalution as mentioned in 5.4.1.

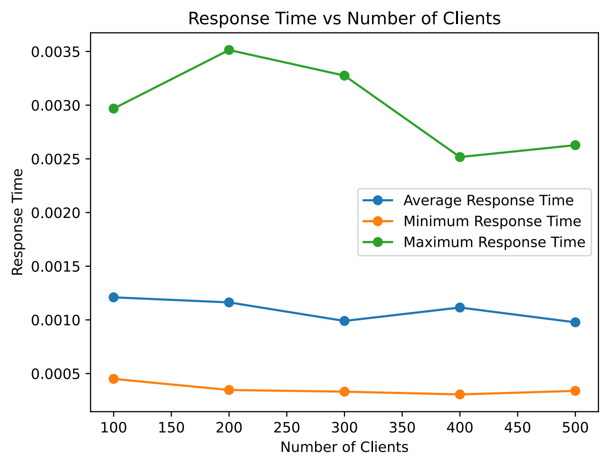


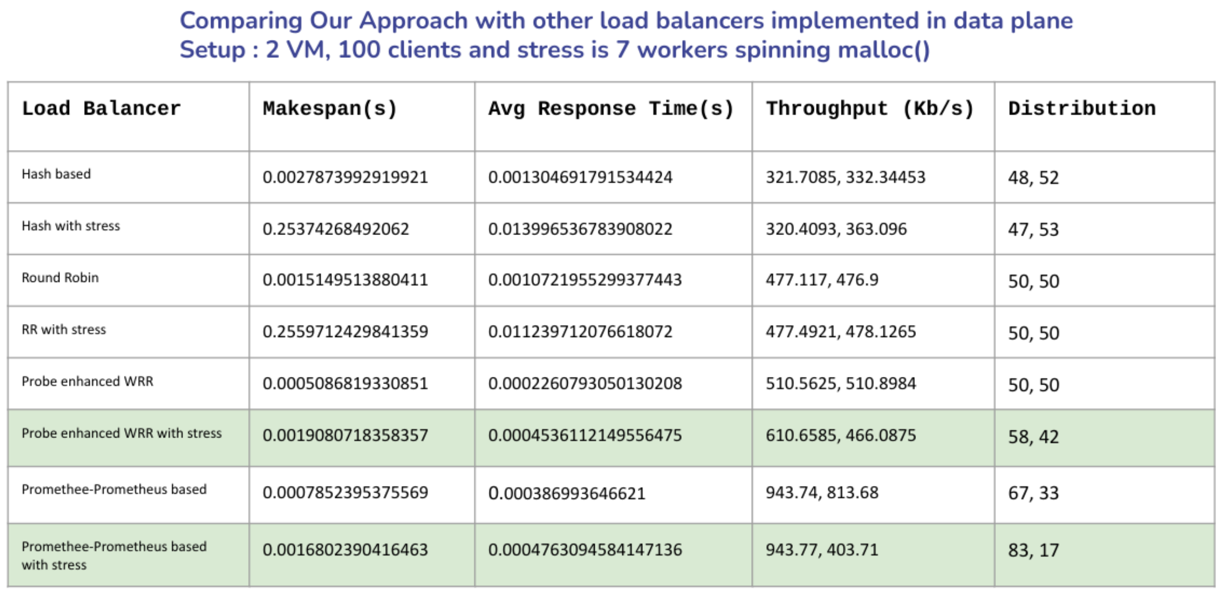
Fig. 6.1 Response Time vs Number of Connections

Figure 6.1 indicates a slightly elevated average response time compared to the previous approach. This increase may be attributed to the Prometheus server sharing resources with the virtual machine under evaluation. We expect that testing within a fully isolated environment should yield better results.

Chapter 7

Conclusion & Future Work

7.1 Final Observations



Comparing Our Approach with other load balancers implemented in data plane
Setup : 2 VM, 100 clients and stress is 7 workers spinning malloc()

Load Balancer	Makespan(s)	Avg Response Time(s)	Throughput (Kb/s)	Distribution
Hash based	0.0027873992919921	0.001304691791534424	321.7085, 332.34453	48, 52
Hash with stress	0.25374268492062	0.013996536783908022	320.4093, 363.096	47, 53
Round Robin	0.0015149513880411	0.0010721955299377443	477.117, 476.9	50, 50
RR with stress	0.2559712429841359	0.011239712076618072	477.4921, 478.1265	50, 50
Probe enhanced WRR	0.0005086819330851	0.0002260793050130208	510.5625, 510.8984	50, 50
Probe enhanced WRR with stress	0.0019080718358357	0.0004536112149556475	610.6585, 466.0875	58, 42
Promethee-Prometheus based	0.0007852395375569	0.000386993646621	943.74, 813.68	67, 33
Promethee-Prometheus based with stress	0.0016802390416463	0.0004763094584147136	943.77, 403.71	83, 17

Fig. 7.1 Comparing various Load Balancing algorithms

From Figure 7.1, it is evident that both algorithms perform better in terms of load distribution and average response time under varying loads on VMs. Therefore, these algorithms are adaptive and load-aware for load balancing.

7.2 Conclusion

Our research explored the implementation of dynamic load balancing techniques in the data plane using P4 switch. We demonstrated that P4-based load balancing offers significant advantages over traditional application-layer approaches, including:

- **Speed & Granularity:** Fast decision-making with fine-grained control at packet level, reducing latency compared to application-layer balancing.
- **Scalability & Efficiency:** Scales effectively within the network infrastructure, distributing load across multiple VMs without overhead. It minimizes complexity and overhead by integrating directly into switch hardware.
- **Flexibility & Resilience:** P4's programmability enables custom load balancing algorithms tailored to network needs, enhancing flexibility. Additionally, it provides resilience against failures without relying on external controllers.

Real-time metric-based load balancers effectively distribute traffic in virtualized environments. Network operators can customize weight parameters for VMs to optimize performance for their specific use cases. Offloading load balancing algorithms to smart NICs has the potential to further improve efficiency by reducing CPU cycles required on the host.

Challenges and Learnings:
Setting up the P4 environment and integrating it with a virtual environment presented initial challenges. Creating a targeted testing environment was also complex.

7.3 Future Work

We plan to re-evaluate our algorithms in an isolated environment where VMs are pinned to dedicated CPU cores, ensuring isolation from host resources, thereby isolating servers and clients. This will allow us to analyze the impact of resource contention on load balancing performance.

Overall, this BTP project provided us with valuable insights into advanced networking concepts, particularly those related to faster and more efficient network management. We could explore many advance technologies and varying networking environments. P4 offers a dynamic and adaptable platform for network operators to manage and modify different network functions implemented in data planes. Load balancing strategies in P4, is one such approach towards improving performance. Offloading these tasks to smart NICs presents a promising avenue for further research and validation of our proposed solutions.

References

[1] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[2] T. P. L. Consortium, "p4-spec." [Online]. Available: https://github.com/p4lang/p4-spec

[3] ——, "P416 language specification." [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[4] ——, "The p4 language specification." [Online]. Available: https://opennetworking.org/wp-content/uploads/2020/10/p4-110.pdf

[5] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé, "P4xos: Consensus as a network service," *IEEE/ACM Transactions on Networking*, 2020.

[6] T. Holterbach, E. C. Molero, M. Apostolaki, E. Z. A. Dainotti, C. S. D. S. Vissicchio, U. London, L. Vanbever, and E. Zurich, "Azure accelerated networking: SmartNICs in the public cloud," 2019. [Online]. Available: https://www.usenix.org/system/files/nsdi19-holterbach.pdf

[7] S. X. Jiarui Xu and S. M. I. Jin Zhao, "P4neighbor: Efficient link failure recovery with programmable switches," 2021. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9319403

[8] S. D. R. V. K. C. A. F. V. T. L. G. F. M. R. P. N. Y. G. V. M. Mohammad Alizadeh, Tom Edsall, "Conga: Distributed congestion-aware load balancing for datacenters," 2014.

[9] C. C. Jin-Li Ye and Y. H. Chu, "A weighted ecmp load balancing scheme for data centers using p4 switches," 2018. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8549549&tag=1

[10] C. K. A. S. J. R. P. U. . N. M. C. Naga Katta*, Mukesh Hira†, "Hula: Scalable load balancing using programmable data planes," 2016.

[11] T. C. Hung and N. X. Phi, "Study the effect of parameters to load balancing in cloud computing," 2016.

[12] C. A. Andrew Or, Matheus V. X. Ferreira, "Dolphin: Dataplane load-balancing in programmable hybrid networks," 2021. [Online]. Available: https://matheusvxf.github.io/files/manuscripts/dolphin.pdf

[13] A. A. D. Author: Kamila Souckov", "Load aware -l4 load balancing algorithm." [Online]. Available: https://github.com/AnotherKamila/sdn-loadbalancer/blob/master/report/report.pdf

[14] SoundCloud, "Prometheus," https://prometheus.io, Prometheus, 2012.

[15] ederollora, "Checksum calculation when adding payload to syn packet," 2021. [Online]. Available: https://forum.p4.org/t/checksum-calculation-when-adding-payload-to-syn-packet/96

[16] B. M. J.P. Brans, Ph Vincke, "How to select and how to rank projects: The promethee method," *European Journal of Operational Research*, 1986. [Online]. Available: https://www.researchgate.net/publication/4756250_How_to_select_and_how_to_rank_projects_The_PROMETHEE_method_European_Journal_of_Operational_Research_14_228-238

[17] H. Taherdoost, "Decision making using the analytic hierarchy process (ahp); a step by step approach," *Research and Development Department, Hamta Business Solution Sdn Bhd, Malaysia*, 2017. [Online]. Available: https://www.researchgate.net/publication/4756250_How_to_select_and_how_to_rank_projects_The_PROMETHEE_method_European_Journal_of_Operational_Research_14_228-238