B.Sc. in Computer Science and Engineering Thesis

Cluster Based Distributed State Management of Network Function

Submitted by

Fardin Hossain
201705038

Nishat Farhana Purbasha
201705067

Saif Ahmed Khan
201705110

Supervised by

Dr. Rezwana Reaz

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka, Bangladesh

May 2023

CANDIDATES' DECLARATION

This is to certify that the work presented in this thesis, titled, "Cluster Based Distributed State Management of Network Function", is the outcome of the investigation and research carried out by us under the supervision of Dr. Rezwana Reaz.

It is also declared that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Fardin Hossain
201705038

Nishat Farhana Purbasha
201705067

Saif Ahmed Khan
201705110

ACKNOWLEDGEMENT

We are thankful to our honorable supervisor, Dr. Rezwana Reaz, our family and our peers for all the assistance they have provided us with. This huge undertaking would not have been possible without their help.

Dhaka
May 2023

Fardin Hossain

Nishat Farhana Purbasha

Saif Ahmed Khan

Contents

List of Figures

List of Tables

ABSTRACT

Network function is a software application or service that performs a specific task within a computer network. We propose a cluster based distributed state management system for network function that supports dynamic and efficient flow and state migration information from one network function to another. We hereby present three novel mechanisms to ensure the efficient functioning of stateful NFs. Firstly, multithreaded network functions are introduced for faster packet processing and efficient utilization of resources. Secondly, we propose a cluster-based NF state replication scheme to reduce bandwidth requirements during flow migration. States are replicated among all cluster members continually, which results in efficient state migration when a flow migration takes place without overloading the network. Finally, an enhanced cluster-based load balancing approach is presented to improve the system's load distribution. This approach makes sure that no node is being over-utilized, preventing a single node from becoming a bottleneck. The proposed mechanisms work together seamlessly to maintain strong consistency on global state updates without any central dependency. Experimental evaluations show that the proposed system has faster processing power and reduced state migration overhead compared to existing systems.

Chapter 1

Introduction

Network Funtions (NFs) [1] are at the heart of computer networks. They are software or hardware entities designed to perform network specific tasks. With advancements in virtualization technologies, Network Function Virtualization (NFVs) [2] has transformed the way network functions are implemented by substituting dedicated hardware with virtual machines (VMs) that operate on physical servers. This transition provides numerous notable benefits, such as scalability, flexibility, resilience, and cost-effectiveness.

Within the NFV framework, the processing of packets is dispersed among various instances of virtual network functions. In this article, we will use the terms ”NF instance” or ”NF” to refer to a software-based generic network function running on a virtual machine node. A stateful NF is an NF in which its current packet processing relies on information from previously handled packets. These past information of the packets are stored as states in the NF to allow them to perform their intended tasks successfully.

Creating and maintaining these NFs are a challenging task and often these instances are in distributed machines, that is, they reside in different physical servers. Managing the abstraction of their physical separation involves state sharing, state migration among the NF instances and also handling failure events.

State-sharing among NFs has been made possible by OpenNF [3] thanks to a centralized controller. An NF alerts the central controller when its state changes, and the controller then broadcasts the new status to all other NFs. State migration is required during scaling events if states are not shared by all NFs. A state migration technique provided by OpenNF guarantees lossless state transfers while maintaining their order. To deal for NF failures, replication is used. However, there is a chance of a single point of failure when relying on the central controller for state synchronization and sharing.

To avoid migration completely, we introduce StatelessNF [4] where states are kept on a remote server that is available to all NFs involved. Neither typical nor scaling events necessitate migration as a consequence. Due to the requirement for remote state access for each packet processing, such an approach introduces inefficiency. The remote server must also be replicated.

S6 [5] uses yet another migration avoidance method. All NFs have access to the distributed object space where states are stored via S6. In S6, all NF instances are able to access some states, but not all of them do. This makes it possible for one NF to remotely access a state object held by another NF. However, for states with significant read activity, they might be exported to the NF making the request. S6 does not totally eliminate the requirement for state movement as a result. It's essential to remember that the current architecture of the S6 does not support NF failure.

The replication of NF states can be used to guarantee fault tolerance in the presence of NF failures. Leading projects that use the replication strategy to mitigate NF failure include Pico replication [6], FTMB [7], and REINFORCE [8]. For each NF involved in these activities, a replica NF is present, and states are transferred from the active NF to the replica NFs. However, elastic scaling is not taken into account in the aforementioned approaches. When elastic scaling and load balancing are used, explicit state migration becomes necessary because states are not shared by all replicas.

The DEFT (Distributed Elastic and Fault Tolerant NFs) [9] system covers both elastic scalability and fault tolerance. DEFT does not employ centralized storage like StatelessNF, but each NF has a state manager for local states and Consensus for global updates to reduce the burden of state migration. Only updates to global states are required to be transferred during normal operation. Every NF has a backup that receives the exact same packets as the primary does. State migration overhead upon failure is decreased as a result. However, there is still a high migration overhead during elastic scaling.

Our system builds upon DEFT and fills in some of its gaps. We use a stateful NF system to reduce the significant overhead costs associated with elastic scaling of network functions. We also present a method for improving throughput and lowering latency by ensuring faster processing of packets.

1.1 Design Goals

Faster Packet Processing:

- Aiming to increase the processing speed of incoming packets, this paper proposes unique strategies or optimizations.
- To speed up request processing, this can entail looking into effective algorithms, parallelization techniques, or hardware acceleration.

Faster Release of Batches:

- Increase the rate at which the output buffer is filled with processed data.
- The objective is to design methods that increase the rate at which the output buffer is filled with processed data.

Reducing Migration Overhead

- When dynamically scaling resources, such as adding or removing instances or nodes, the objective is to minimize the performance impact.
- This could entail creating effective state replication systems, enhancing load balancing algorithms, or reducing the impact of resource reconfiguration.

1.2 Our Contributuion

In this study, we contribute by proposing a technique to aid in quicker packet processing and lower migration costs during elastic scaling. The majority of prior research has focused on enhancing processing speed or decreasing migration overhead independently, without considering the interaction between these two crucial elements. While DEFT [9] does address this, it comes with a high migration overhead during scaling event.

Our method utilizes multithreading techniques to speed up packet processing substantially, as well as the effective discharge of finished packets into the output buffer. We take advantage of numerous threads' capability by using parallel computing, allowing concurrent packet processing. This concurrency enables processing operations to be completed more quickly, resulting in decreased latency and increased throughput while still maintaining per-flow packet order.

In addition, our system includes clustering methods to enable effective state sharing between clusters. By enabling the grouping of NF instances, clustering enables state sharing between cluster members. The sharing of states mean that NF instances can quickly access and synchronize state data, avoiding overhead and the need for too aggressive replication.

1.3 Properties of Our System

- **Local States:** Local states or states associated with a single flow are handled locally by the NF's storage.

- **Parallel Processing:** The NF now houses several threads as opposed to the single thread packet processing unit employed by DEFT. A single flow has its own distinct input buffer to maintain the per-flow packet order and its own distinct processing thread to ensure significantly faster packet processing.

- **Load Balancing:** An enhanced load balancer measures cpu utilization of the NFs and redistributes traffic to idle instances using a customized cluster based load balancing technique.

- **Global States:** Global states are those that are updated by several flows. All clones share the same global states.

- **Lack of a Centralized Control:** By conducting a consensus among the copies, global states are updated across all of them. The global update order of states is not serialized by a single central entity.

- **Strong Consistency:** Global state updates are observed by all replicas in the same sequence, indicating that global state updates are strongly consistent across replicas. Consensus determines the order, which may differ from the order in which the matching packets arrive from the switch.

- **Eventual Consistency:** Other cluster members already have the status of the NF whose traffic is to be migrated away during the scaling event, but they do not have access to the in-flight packets that are kept in the input buffer while the migration is being performed. The new NF where the flow is to be transferred receives these in-flight packets. Members of the cluster are considered to follow eventual consistency during this time.

Chapter 2

Preliminaries

We will define the most frequently used terminology throughout the dissertation in this chapter.

2.1 Network Function

A Network Function (NF) is a functional building block within a network infrastructure that possesses clearly defined external interfaces and exhibits well-defined functional behavior. In practical terms, Network Functions commonly manifest as network nodes or physical appliances [1]. NFs are special software or hardware components that have specific roles within a network. They are responsible for handling tasks like managing network traffic, implementing network protocols, and offering different network services. Some examples of NFs include routers, firewalls, load balancers, intrusion detection systems, and other network devices.

2.2 Network Function Virtualization (NFV)

Network Function Virtualization (NFV) is an innovative technology introduced by The European Telecommunications Standards Institute (ETSI) in 2012. It makes clever use of virtualization technology to enable the provisioning of NFs such as load balancers, switches, firewalls, Domain Name Server (DNS) and more [2]. These NFs are implemented as software-based services, decoupled from dedicated hardware, and hosted on virtually making use of general-purpose servers, storage, and switches.

By transforming traditional network appliances into virtual network appliances, NFV offers numerous benefits. It minimizes the need for installing, maintaining, and acquiring specialized and or dedicated hardware at client premises which, in turn, leads to reduced costs and power consumption. Additionally, the adoption of NFV creates a new market that fosters the development of innovative businesses in the network services domain.

2.3 Software-defined Networking (SDN)

Software-Defined Networking (SDN) revolutionizes network management and configuration by centralizing control and enabling enhanced network agility and flexibility [10]. They do this via separation of the control plane from the data plane. SDN empowers organizations to automate crucial network management tasks such as traffic routing and load balancing. This automation not only improves overall network performance but also reduces the likelihood of errors stemming from manual configuration.

While NFV and SDN are related concepts and often deployed together, they address different aspects of network architecture and functionality [11]. NFV focuses on the virtualization of network functions, enabling flexible deployment and scalability of services. SDN, on the other hand, focuses on centralizing and programmatically controlling the network, providing flexibility and agility in managing network behavior. The use of script-based simulations in our research provided a practical means of evaluating SDN concepts related to load balancing, packet flow redirection, and other network functions.

2.4 State

In order to maintain network connections, a Network Function (NF) requires state information to be stored. Most NFs are designed to be stateful, meaning they can remember important network information as they process packets. For instance, a router consults with its routing table which stores information about various networks before sending a packet to next hop. However, managing the state information can be a daunting task for NFs. A firewall, for example, needs to keep track of all processing history to decide whether a packet should be allowed to pass through. If it fails to block any unwanted packet through it, the state information will reflect a faulty current state. This highlights the challenge of managing and updating network state information, despite its value for NFs.

In our system we have introduced two types of state:

1. Per-flow State: State that is associated with a particular flow is called a per-flow state. As a single flow is always directed towards a specific NF until any migration occurs, per-flow states do not need to be shared among other NFs. That is why it is often called local state.

2. Global State: State that is updated by multiple NFs is called a global state. NFs concerned with any global state must share it with one another.

In our system, we are mainly concerned about the sharing of local states when migration occurs due to overloading of an NF. In that case, updating of local states might need stalling through out the process of migration.

2.5 State Update

The update of state can be broadly categorized in two classes:

1. Commutative Update: Updates that can be performed in any order without changing the end result. In other words, the order in which commutative updates are applied does not affect the final state. For example, each packet updates a state by adding numbers. If three packets A, B and C contains -1, +4 and +2 respectively, then the total sum will always be +5 regardless of the sequence of packets arrived.

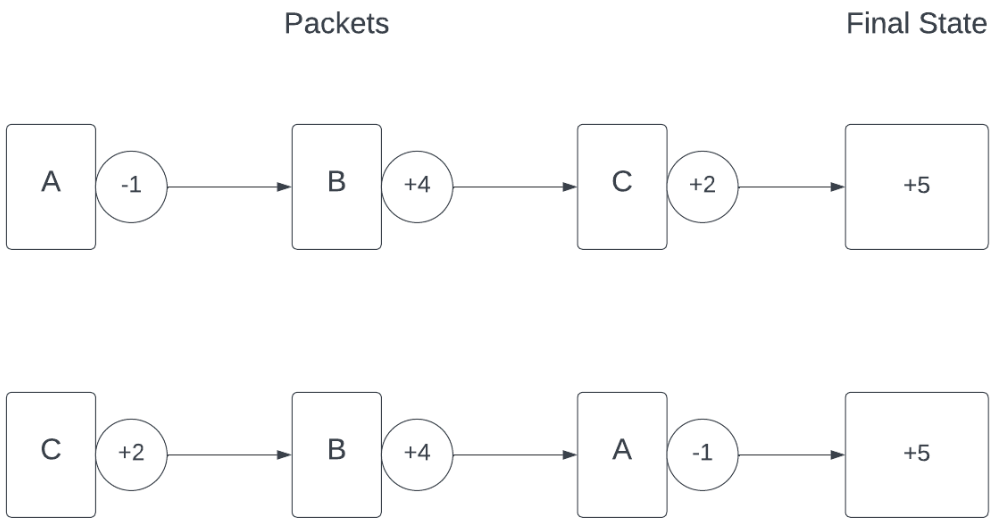


Figure 2.1: Commutative State Update

2. Non-commutative Update: Updates that must be performed in a specific order to achieve the desired result. The order in which non-commutative updates are applied can affect the final state. In case of previous example, If the packets A, B and C contains -1, x4 and +2 respectively, the sum will be -2 in case of arriving sequence of packets $A \to B \to C$ and +4 in case of arriving sequence of packets $A \to C \to B$.

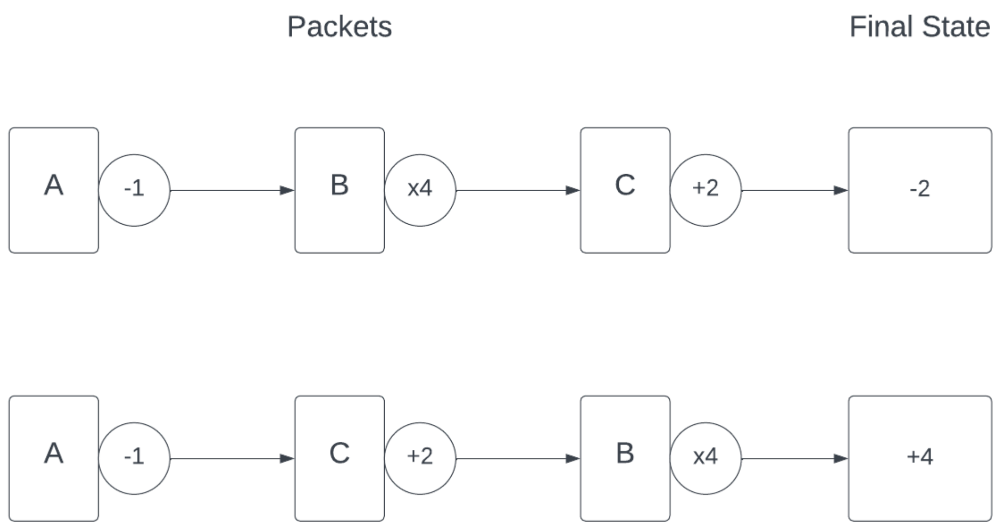


Figure 2.2: Non-commutative State Update

2.6 State Migration

State migration refers to the process of transferring the state information of a system from one location to another. In distributed systems, state migration is used to ensure fault tolerance and load balancing. In our system, we are concerned about load balancing due to overloading. if one node in a distributed system becomes overloaded, state migration can be used to transfer some of the state information to a less loaded node. This helps to ensure that the system remains stable and can handle increasing loads without degrading performance.

2.7 Consistency

Three categories of consistency models were taken into account in our design. These paradigms are additionally employed in distributed systems, such as distributed shared memory architectures or distributed data storage. As well as having varying degrees of efficiency and consistency trade-offs, many consistency models set distinct rules for the apparent order and visibility of updates.

1. Eventual Consistency: We refer to a situation as eventual consistency where some states across several NFs do not need to be consistent right away but will be consistent given enough time. Suppose we have two NFs, NF1 and NF2, and three updates that need to be reflected in both NFs: A, B, and C. This will be demonstrated using an example. $A \rightarrow B \rightarrow C$ are the updates that NF1 receives. Although NF2 might not immediately receive

this update, eventual consistency ensures that, given enough time, NF2 will eventually receive update $A \to B \to C$ and will match NF1 in terms of consistency.

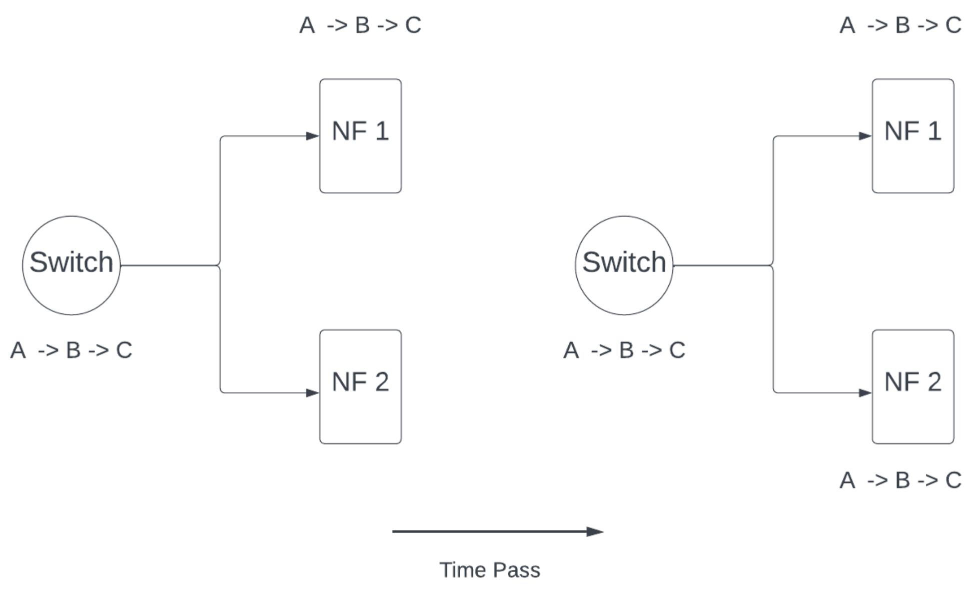
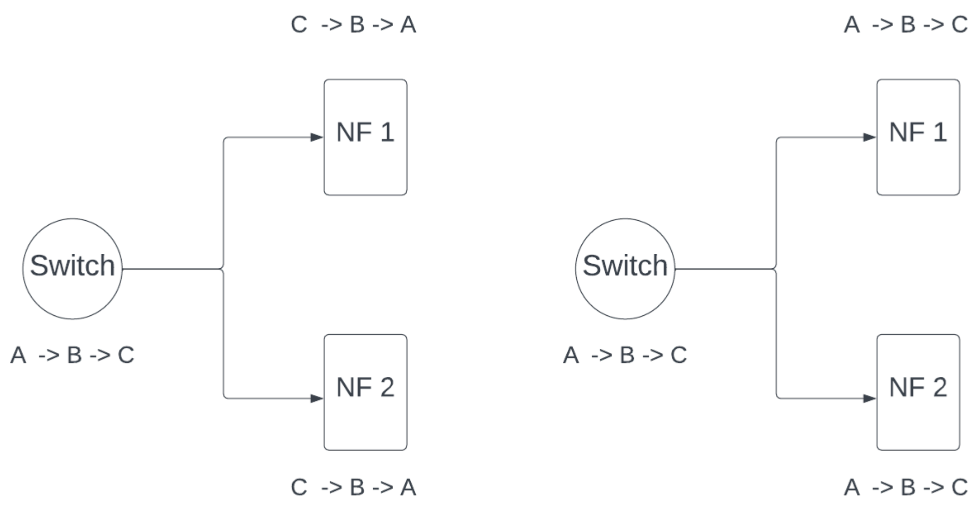


Figure 2.3: Eventual Consistency

2. Strong Consistency: If all updates seen by the parallel nodes or network functions are in the same order, a protocol is said to offer strong consistency. Suppose we have two NFs in our system, NF1 and NF2. The switch's update sequence is $A \to B \to C$. Now, if NF1 perceives the update order as $C \to B \to A$, then NF2 must also reflect this specific update order.



(a) Strong Consistency (b) Strict Consistency

Figure 2.4: Strong and Strict Consistency

3. Strict Consistency: Strict consistency refers to the consistency protocol that requires that NFs process packets in the same sequence that they were delivered by the switch. The strongest consistency paradigm is strict consistency. The packets must be processed by NFs in the exact same order if the switch's packet order is $A \rightarrow B \rightarrow C$.

2.8 Cluster

A cluster of network functions refers to a group of network function instances that work together to provide a specific network service. These network functions can include firewalls, load balancers, routers, and other devices that are used to manage and optimize network traffic.

The main goal of creating a cluster of network functions is to improve the performance and availability of the network service. By distributing the workload across multiple network function instances, a cluster can handle a larger number of requests and provide better scalability. In addition, the cluster can provide redundancy and failover capabilities, which help to ensure that the network service remains available even in the event of hardware or software failures.

2.9 Multithreading

Multithreading is a programming concept that allows a program to perform multiple tasks concurrently. In other words, it enables a single program to execute several different threads of execution, each running independently and simultaneously, within a single process. Each thread can perform its own set of operations and share the same resources, such as memory and CPU time, with the other threads.

In distributed systems, multithreading is used to improve the performance and scalability of applications by running on multiple machines or nodes. Tolerating unpredictable communication latency is also one of the main attractions of using this mechanism [12]. In our system, it is used to process packets from different flows in different threads simultaneously inside one NF.

2.10 Load Balancing

Load balancing is a technique used in computer networking and distributed systems to distribute workloads across multiple resources to improve overall system performance and avoid overloading individual resources [13]. It increases scalability, and better resource utilization.

In our proposed system, load balancers act as intermediaries between stamper and NFs, receiving incoming requests and routing them to the most appropriate NF based on a set of predefined rules or algorithms. These rules can take into account factors such as resource availability, network congestion, or CPU utilization times.

2.11 Network Function Replication

Replication of NF is a technique that is widely used in different network architectures to ensure fault tolerance property. The main approach of the technique is to create replicas of each NF where one is active NF and the others are passive NFs. The active NF is called the primary NF, and the passive ones are called secondary NFs. [14,15] In these papers, every primary NF has a replica or backup NF. In our system, we have also maintained one backup NF for each primary NF.

The primary NF processes incoming packets and updates network state information. In our context, it replicates its state information to the backup NF after each batch of packets. If the primary NF fails, the backup NF takes over as the primary NF and continues processing packets after the last processed batch of packets.

2.12 Elastic Scaling

Elastic scaling is a technique used to dynamically add or remove the resources allocated to a network with the change in incoming packet load [16]. In our scenario, adding or removing new instance of NF is denoted as elastic scaling.

In our scenario, we try to avoid elastic scaling as much as possible since we have presumed that cost of elastic scaling is much higher than state migration after each batch. We only apply this technique when all the Nfs from all clusters are overloaded. In that case we are left with no option but elastic scaling.

Chapter 3

Literature Review

3.1 Elastic State Management

3.1.1 State Migration Strategy

OpenNF [3]: OpenNF is a control plane architecture that provides efficient, coordinated control of both internal NF state and packet forwarding state to allow quick and safe reallocation of flows across NF instances. OpenNF simultaneously achieves 3 goals of addressing race condition, bounding overhead and accommodating a variety of NFs with minimal changes. OpenNF contains a Northbound API and a Southbound API for maintaining state management and migration operations. To manage events and export or import states between NFs, the controller uses the southbound API, which offers a standard NF interface. The move, copy, and share operations on NF states are supported by the northbound API. While copy and share procedures are used to replicate states across multiple NF instances, move transfers the state and input traffic for a group of flows from one NF instance to another during scaling events. When state consistency across NFs is not necessary or eventual consistency is desired, the copy operation, which copies states from one NF instance to another, can be utilized. When stringent or strong consistency between NF instances is desired, share operation is utilized. Loss-free and order-preserving state migration procedures are offered by OpenNF and are built around a central controller. During state sharing, the central controller is in charge of preserving the sequence of state updates across the NF. The central controller is alerted when an NF modifies any state, and it then duplicates the changed state across all NFs.

3.1.2 State Migration Avaoidance Strategy

StatelessNF [4]: StatelessNF is designed by decoupling the existing design of network functions into a stateless processing component along with a data store layer. The states are conserved in an independent data store layer in their system. There is not any traffic affinity to an instance. Both per-flow and per-packet distribution methods are available for traffic. No state migration is necessary during normal and scaling events because states are accessible to all NFs. They do not require a failure detection system that is highly strict because there is essentially little cost for them to fail over. Although their decoupled nature offers straightforward solutions, the frequent distant state access results in a substantial increase in latency. Additionally, the data store, StatelessNF's main building block, functions as a single point of failure. It must therefore be low-latency and resilient. StatelessNF offers a number of optimizations to reduce access to external data servers. Instead of maintaining static states on a remote store that users may access remotely, they replicate static states (such as firewall rules and IPS signature databases) on each NF during startup. Stateless also makes an effort to reduce communication with remote servers by utilizing the state access patterns of NFs.

3.2 Fault Tolerant State Management

Pico Replication [6]: Pico replication is a framework for middleboxes or NFs geared towards the objective of high availability (HA). A framework for middleboxes or NFs called pico replication is designed with high availability (HA) in mind. It operates according to flow. Three crucial contributions were made by Pico Replication: (i) an order of magnitude improvement in replication performance over current system-level HA solutions (ii) system-level support for middlebox HA that enables custom per-flow replication with transparent failure recovery (iii) and a middlebox aware HA policy framework that enables various dynamically adaptive policies. From a high level, Pico Replication is made up of four main parts: the SDN Controller, the State Management Module, the Packet Management Module, and the Replication Module. The first three modules are connected to each middlebox/NF. Packet replication methods are not used by Pico Replication. In conjunction with PMM, it processes the packets in batches, ensuring that only the right packets get processed and saved. It collaborates with SMM to guarantee that the states are updated and saved appropriately. The state is replicated by RM and assigned to the appropriate VM. The replica VM is finally added to an SDN, where a controller turns it on. Pico utilizes checkpointing to recover from failures. Following completion of checkpointing, PMM releases the packets. Backups restore to the most recent saved checkpoint in the event of failure. The fact that this approach to state management provides for consistency with little expense is one of its main benefits.

FTMB [7]: The primary objective of FTMB was to develop a framework or structure for NFs that could accommodate a wide range of NFs without requiring too many modifications (generality) and maintain a very low level of overhead during routine operations while ensuring a successful and accurate recovery from failure. FTMB is a multi-threaded architecture with shared (global) and local (per-flow) state. They use packet-replay with infrequent checkpointing as their failure recovery model. At the packet level, logs are preserved. A packet is released once it has been processed. It is necessary to perform checkpointing infrequently in order to ensure that the recovery procedure is effective. During checkpointing, snapshots of the system state are stored, and the packet logs up to that point are no longer required. The system is brought up to date to the most recent checkpoint during a failure, and the packets are replayed to restore the system to its previous condition.

3.3 Elastic and Fault Tolerant State Management

DEFT [9]: DEFT tries to address the issue of a distributed state management system being both elastic and fault tolerant. DEFT supports strong consistency on global state updates. It achieves that by having a consensus of NFs on global updates. As DEFT achieves strong consistency through a consensus, no centralized state storage is necessary. Thus, DEFT mitigates the issue of a single point of failure for centralized server. Local state updates are done by an NF's own state manager. DEFT ensures a primary-backup pair for every NF pair. A primary NF and its secondary NF communicate and share information with each other through Two-Phase Commit. DEFT also contains a Failure Detection Unit(FDU) that detects failure. Upon failure, the FDU assigns the secondary NF as the new primary NF.

Chapter 4

Methodology

In this chapter we will discuss the design and methodology of our architecture.

4.1 Architecture and Components

We present a high-level design of our architecture in Figure 4.1 . In our system, a node may contain multiple NFs, and each NF works either as a primary NF or as the backup of a primary NF. We call the later ones secondary NF.

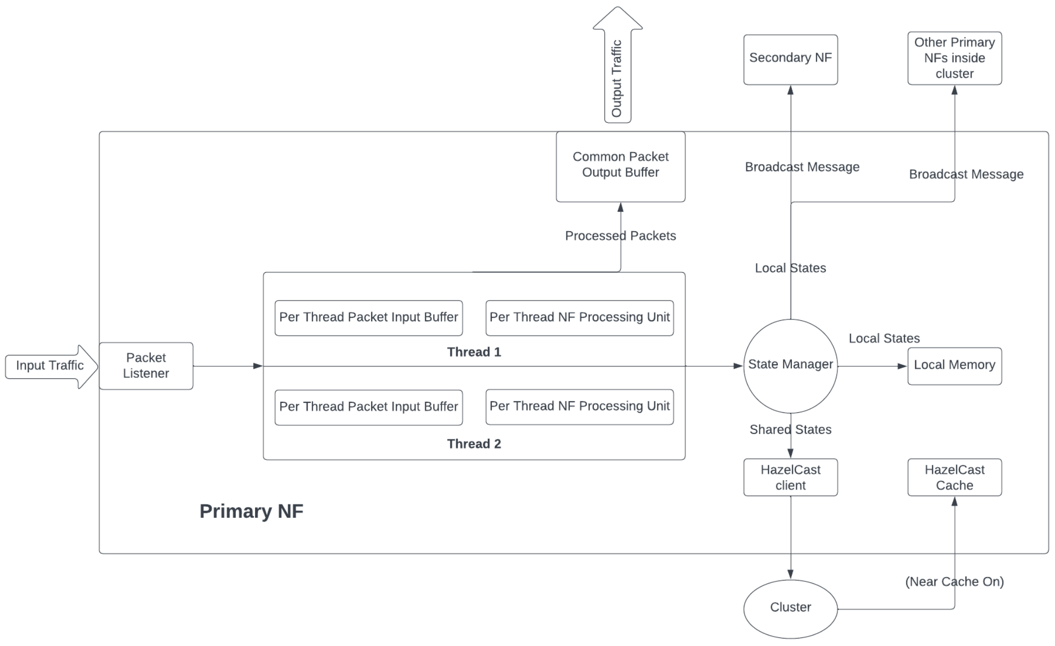


Figure 4.1: System Architecture

4.1.1 Network Topology

Our network topology is shown in fig. 4.2. The SDN controller is connected to the two switches and the load balancing unit. All the input packets from the client come through switch-1 and then go through the stamper unit. The stamper unit assigns a packet number to each packet from each flow and forwards them to the load balancing unit(LBU). The LBU then forwards them to switch 2. After that switch 2 forwards the packets to designated primary NF and its secondary NF. We will discuss the reasoning behind our system's design in the upcoming sections.

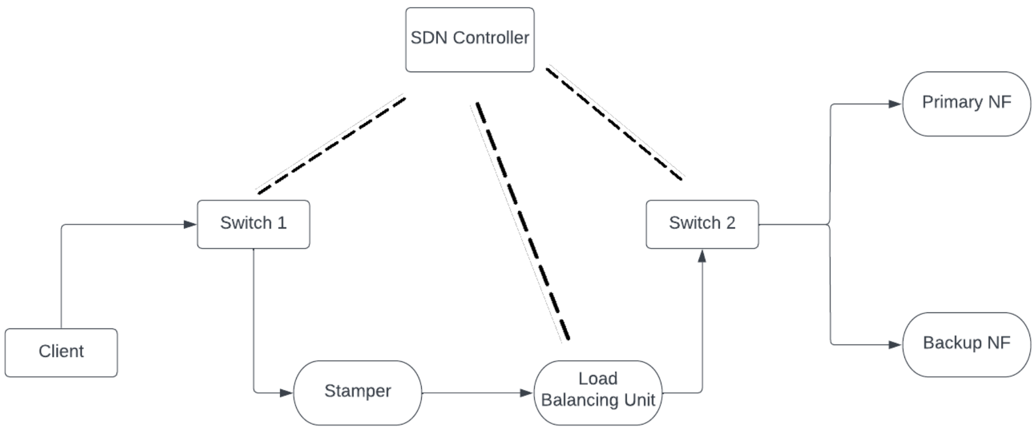


Figure 4.2: Network Topology

4.1.2 Failure Detection Unit(FDU)

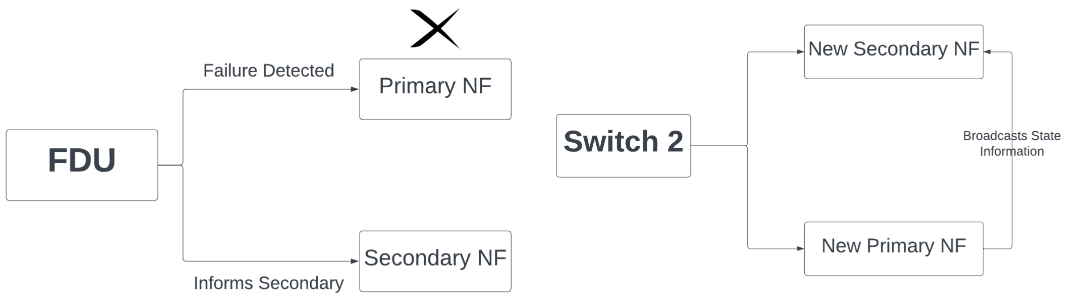
Our system has a failure detection unit(FDU) which detects the primary NF failure. We have inherited this concept from DEFT [9].

The operation of FDU is shown in fig. 4.3. The primary function of the FDU (Failure Detection Unit) is to regularly assess the status of both primary NFs and their corresponding secondary NFs by sending periodic ping requests. Each NF instance responds with its current state. In the event of a primary NF failure, the FDU unit triggers a notification to the secondary NF, instructing it to assume the role of the primary NF and take over its responsibilities.

4.1.3 Buffers

In our system, every NF has 2 types of buffers to store packets:

- Input Buffer: Inside a primary NF, each thread has its own input buffer and processing unit. Before allowing packets to enter the processing unit, the input buffer in the primary



(a) FDU informs Secondary

(b) New Primary NF sends states to New Secondary

Figure 4.3: Operations of a Failure Detection Unit

NF stores them. As long as the processing unit is busy processing packets or if the packet arrives out of order and has not yet had its turn to enter the processing unit, it remains in the input buffer. The input packets travel from the input buffer to the processing unit. In the secondary NF, the input buffer stores the input packets until the primary NF instructs it to discard them.

- Output Buffer: Inside a primary NF, there is a common output buffer for all the threads. The processed packets are held in the primary NF's output buffer until they are released. The packets are released from the output buffer by the primary NF once its atomic commitments with its secondary NF are complete. Until this secondary NF becomes the new primary, the output buffer in the secondary NF is inactive.

4.2 Normal Operation

In this section, we will discuss in detail the working mechanism of our system in normal operations. Until any flow migration events happen, packets from the same flow will always be delivered to the same NF. Different flows will be distributed among multiple NFs.

4.2.1 Flow Initialization

At first, there are no flow rules implemented in any of the switches in fig. 4.2. The controller adds a flow rule to the switch so that when packets begin arriving at switch-1, they are forwarded to the stamper. These packets are then forwarded to load balancing unit(LBU) by the stamper once the payload has been given a special number by the stamper. The stamper module's given number essentially functions as a network packet flow counter. The LBU then determines the destination NF of a particular flow and sends this information to SDN controller. The controller adds a rule to switch-2 so that when switch-2 receives the stamped packet, switch-2 will send a duplicate packet to the matching secondary NF.

4.2.2 Input Packet Buffering

To enforce strict consistency between the primary and the related secondary NF is our main goal. To accomplish this, we need the packets to be processed by the primary and secondary NFs in the same order. However, a packet may be reordered on the route from the switch to the NF. We won't be able to achieve strong consistency if the order of packet processing is inconsistent between the primary and secondary NFs. Each packet is given a number by the stamper module, which enables us to determine whether a packet arrived out of order.

We have different threads for different flows in our architecture, as shown in fig. 4.1. Each thread consists of an input buffer and a processing unit. A packet of a particular flow would first enter the designated input buffer before moving on to the processing unit. To ensure that the order in which packets are processed matches the order in which switches send them, each NF will maintain a nextExpectedPktId variable for each flow. The NF would place the packet inside the input buffer of that particular flow rather than processing it if the packetId of the packet received by the NF was not equal to nextExpectedPktId of that flow. The nextExpectedPktId, which will be incremented upon subsequent packet reception, will be equal to packetId. If space is available in the output buffer, the NF will select this packet from the input buffer and begin processing it. The NF would continuously check to see if the input buffer included the packet whose packetId was equal to the variable nextExpectedPktId of that flow.

So the overall condition to process a packet of a particular flow by the primary NF:

- The value of the packet's packetId assigned by the switch has to be equal to the NF's nextExpectedPktId for a particular flow.
- There has to be room in the output buffer for the packet.

4.2.3 Packet Processing

Inside the packet processing unit, a packet of a particular flow may update both the local or global states. In our system, we have mainly focused on the local state information and its migration. The update of global state is similar as DEFT [9]. For local updates, our system achieves strict consistency.The packets are processed by the primary NF in the exact same sequence as the switch sends them. The output buffer holds onto every packet that leaves the processing unit and does not instantly release it.

4.2.4 Output Packet Buffering

A packet will be stored in the output buffer by primary NF after being processed and having its states updated. The output buffer would wait until a batch of packets with consecutive packetIds was formed. The release of the packets happens after the formation of a batch.

4.2.5 State Information Migration

After the construction of a batch, the primary NF broadcasts the updated state information to its secondary NF and other primary NFs inside the same cluster, as we can see in fig. 4.1. Then the output buffer releases the packets and upon instruction, the secondary NF discards these packets from its input buffer. So after processing each batch, all the NFs inside same cluster become consistent i.e. they all have the updated state information of all the flows that go through that particular cluster.

4.3 Flow Migration Operation

The flow migration situation occurs when any primary NF becomes overloaded. In that case, we have developed an enhanced cluster based load balancing algorithm to tackle this situation. This algorithm is shown in fig. 4.4.

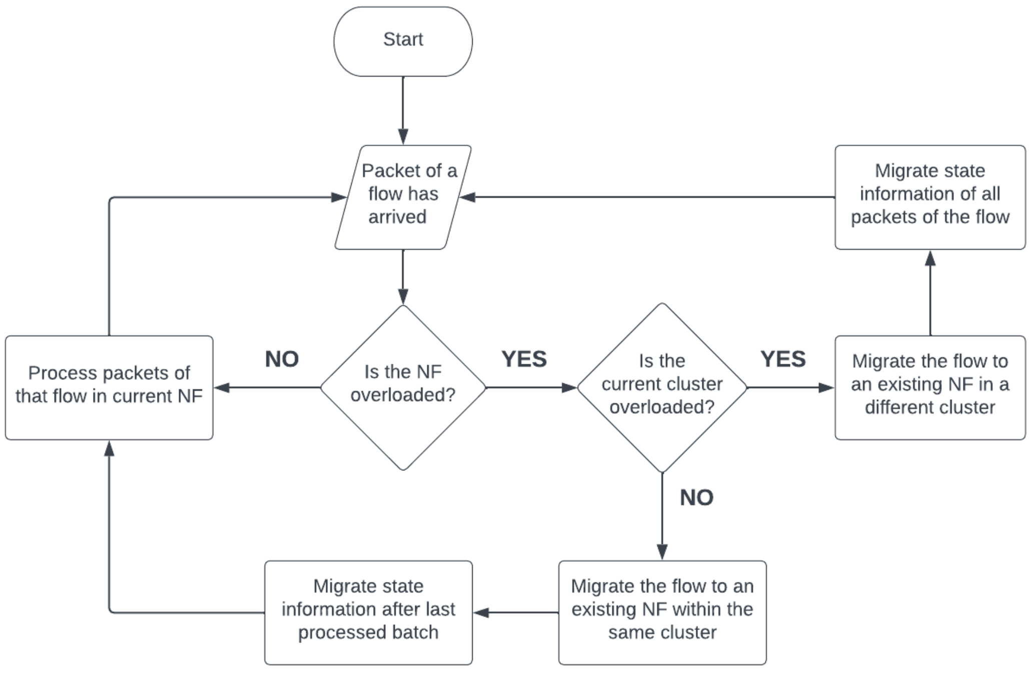

Start
Packet of a flow has arrived
Migrate state information of all packets of the flow
Process packets of that flow in current NF
NO
Is the NF overloaded?
YES
Is the current cluster overloaded?
YES
Migrate the flow to an existing NF in a different cluster
NO
Migrate state information after last processed batch
Migrate the flow to an existing NF within the same cluster

Figure 4.4: Flow Chart of Cluster-Based Load Balancing Algorithm

Upon selection of the destination NF for the flow to migrate from the overloaded NF, two cases can arrive.

- Destination NF in same cluster: In this case, the overloaded NF starts buffering packets from that flow. Then the LBU selects the destination NF. The overloaded NF sends the updated state information after previous batch to the destination NF. Upon receiving this information, SDN controller updates the flow rule at switch 2. After that, overloaded NF forwards the buffered packets to the destination NF.
- Destination NF in different cluster: As the destination NF is in another cluster, it does not have any information about the states updated by the flow. Thus the overloaded NF needs to send the state information updated by all the packets till migration event to the destination NF. Rest of the steps are similar.

4.4 Failover Operation

We inherit this concept from DEFT [9]. In our system, we will consider two types of failures:

1. NF Failure
2. Node Failure

We must address several of our architectural presumptions before moving on to failure recovery:

- The FDU unit continuously monitors for any NF failure. Because of this, this component is solely responsible for NF failure detection.
- The primary NF is not housed on the same physical node as the associated secondary NF.
- The primary NF and the associated secondary NF do not fail at the same time.

4.4.1 Failure Recovery of NF

The NF failure recovery is illustrated in fig. 4.3. The secondary NF takes over role and continue packet processing if the primary NF fails. The FDU unit notifies the appropriate secondary NF B after acknowledging the primary NF A's failure. The new primary and secondary NFs are then designated as NF B and NF C, respectively. In order for the new secondary NF to be consistent with the new primary NF, NF B must now migrate its states to NF C through broadcasting. Noting that duplicate packets are always forwarded to the relevant secondary NF, all the packets that were meant for NF A but could not be processed due to failure are likewise buffered at NF *B*. So we would not experience any packet loss in this scenario. Now, NF *B* needs to process these packets and migrate the states. The SDN controller then changes the flow rule at switch 2 so that the flow is now forwarded to NF *B* and the duplicate packets are forwarded to new secondary NF *C*.

Chapter 5

Implementation

In this chapter, we discuss about the implementation of our proposed architecture. We used Docker container to simulate our network, Hazelcast clients as Network Functions and Packet Sender as UDP packet generator.

5.1 Docker

Docker [17] provides a lightweight and portable way to package and run applications along with their dependencies, making them easy to deploy across different environments. It supports end-hosts, switches, routers and links. Containers are isolated environments that bundle an application and its dependencies together, and Docker employs containerization technology to construct them.A read-only blueprint or template for building containers is known as a Docker image. The operating system, runtime, libraries, and dependencies are all included, along with everything else required to run a program.A text file called a Dockerfile is a collection of instructions for creating a Docker image. The base image to be used is specified, files are copied into the image, environment variables are set, packages are installed, commands are run, and the container is configured. In the docker container, we can define how many network function instances we need, how many backups we need, the stamper module and the consensus module.

5.2 Network Functions

Different network functions have various methods for accessing and updating local and global states. So, in order to evaluate different configurations seen in a wide range of NFs, we created a prototype of a generic network function with configurable parameters. Batch size and global update rate are the variables that can be changed.

How frequently we communicate with the backups and the members within own cluster is indicated by the batch size. The global update rate regulates how frequently global updates (and consequently consensus) are required inside a batch of packets. Each NF has the ability to dynamically allocate threads for each flow so that packets of different flows can be processed in parallel.

5.3 Stamper

Each packet passing through it requires a number added by the stamper module. This action is known as stamping. The module maintains track of how many packets are sent through each flow. The subsequent packet of flow s would be stamped with the number $\mathsf{n+1}$ if, for instance, n total packets of flow s had been processed by the stamper module. For the sake of uniformity, this is required. An NF determines if its $\mathsf{nextExpectedPktId}$ and the packet's $\mathsf{packetId}$ match when it gets a packet. If not, the specific packet has likely arrived out of sequence and cannot be handled right now.

5.4 Primary and Backup

Every primary NF has a backup NF. To be resilient against failure of a single host, the primary and backups are hosted on different hosts. Primary and backup NF store connection information(IP, port) of each other. After each batch of packets is processed, the primary sends the necessary state information of the processed batch to its backup NF. In case of NF failure, the backup NF becomes the new primary and a corresponding backup NF is assigned.

5.5 Buffers

There are two types of buffer in each NF : input buffer and output buffer. There can be multiple input buffers, because every thread of packet processing contains its own input buffer. On the other hand, there is only one output buffer in the whole NF. In order to maintain the packets in FIFO(First In First Out) order, we need a queue data structure. Our input buffers follow the rule of producer-consumer problem [18]. Hence, Python's Queue [19] module was used. Both buffers' upper bound sizes are adjustable variables.

5.6 Global Update Unit

For our consensus module, we used Hazelcast [20]. Hazelcast is a free and open-source In-Memory Peer-to-peer data grid that is straightforward, scalable, quick, and redundant. The Raft consensus protocol is used to provide linearizability even when there is a failure.

A cluster is made up of all the nodes in the system. No master node means there is no single point of failure. Nodes can be easily added to and removed from the cluster. A new node immediately discovers the cluster when it is added to it.

The NF itself connects to the cluster via the Python client. The distributed dictionary (map) is one of the distributed data structures offered by the Python client. They offer the dictionary data structure in both synchronous and asynchronous forms. As for the accuracy of our system, we used the synchronous version of the dictionary because we cannot move on until the consensus is reached. Additionally, we can enable the client's ”near-cache” feature at the risk of reading outdated data. The client will always have a copy of the changes in local memory thanks to this feature, which will enable quicker reading in the future.

To store our global states, we employed a distributed key-value scheme. The Python client alters the corresponding key-value and starts the consensus mechanism whenever an NF needs to update certain global states. All of the NFs in the cluster will receive and apply the update in the same order when the procedure is finished.

5.7 Packet Generation

We used the Packet Sender [21] application to generate UDP packets and run the experiments. In the packet sender application, the desired size and rate of packets can be set very easily. Each instance of the packet sender application generated a new flow of packets.

Chapter 6

Experiments and Results

6.1 Results of Multithreading

At first, let us take a look at the results of multithreading experiments.

6.1.1 Latency vs Number of Threads

The following parameters were used in this experiments:

Parameter	Value
Total Packets Processed	4000
Packet Rate	100 pkts/second
Batch Size	30
Number of Network Functions	1

Under these model parameters, we get the following results.

As we can see from 6.1, per packet latency decreases as the number of threads increases. It is because multiple threads enables separate input buffers and packet processing units for each flow. As a result, packets of every flow can be processed independently. Moreover, multiple flows do not share a common input buffer. So packets of different flows do not have to wait longer to enter its intended input buffer. In conclusion, the more threads we have in an NF, the faster the processing will be and as a result, overall latency will decrease with the increase of threads.

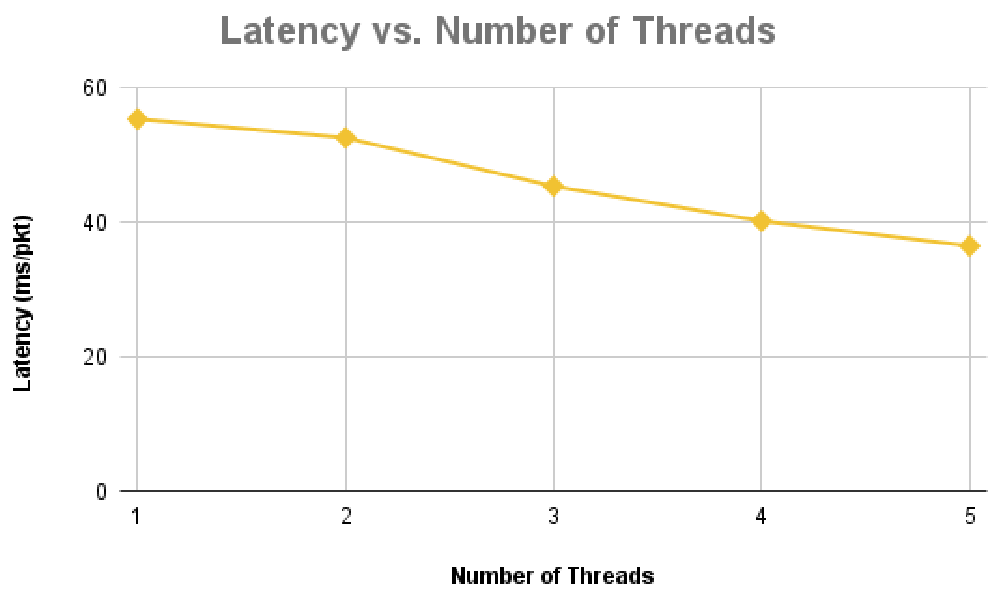


Figure 6.1: Latency vs Number of Threads

6.1.2 Throughput vs Number of Threads

The following parameters were used in this experiments:

Parameter	Value
Total Packets Processed	4000
Packet Rate	100 pkts/second
Batch Size	30
Number of Network Functions	1

Under these model parameters, we get the following results.

As we can see from 6.2, throughput increases as the number of threads increases. This is because increased number of threads reduces the dependency between different flows and packets of multiple flows can be processed in parallel. So much higher number of packets can be processed in less time which results in increased throughput of the system.

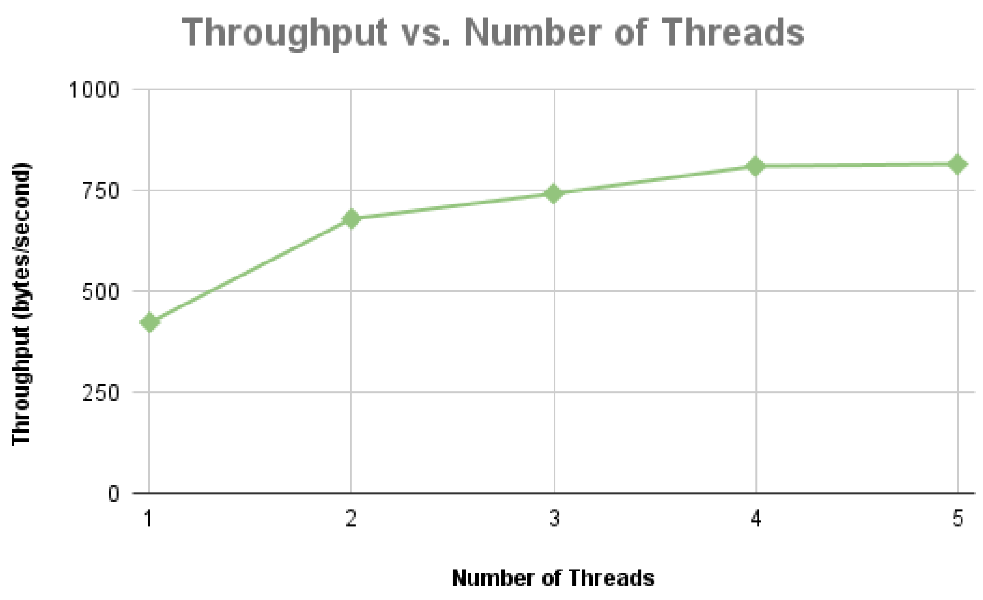


Figure 6.2: Throughput vs Number of Threads

6.2 Effects of Clustering

Now we observe the performance of the system with the introduction of clustering.

6.2.1 Throughput vs Batch Size

The following parameters were used in this experiments:

Parameter	Value
Total Packets Processed	4000
Packet Rate	100 pkts/second
Number of Cluster	3
Number of Network Functions in each cluster	4
Number of Network Functions in total	12

We can observe the following results under these parameters.

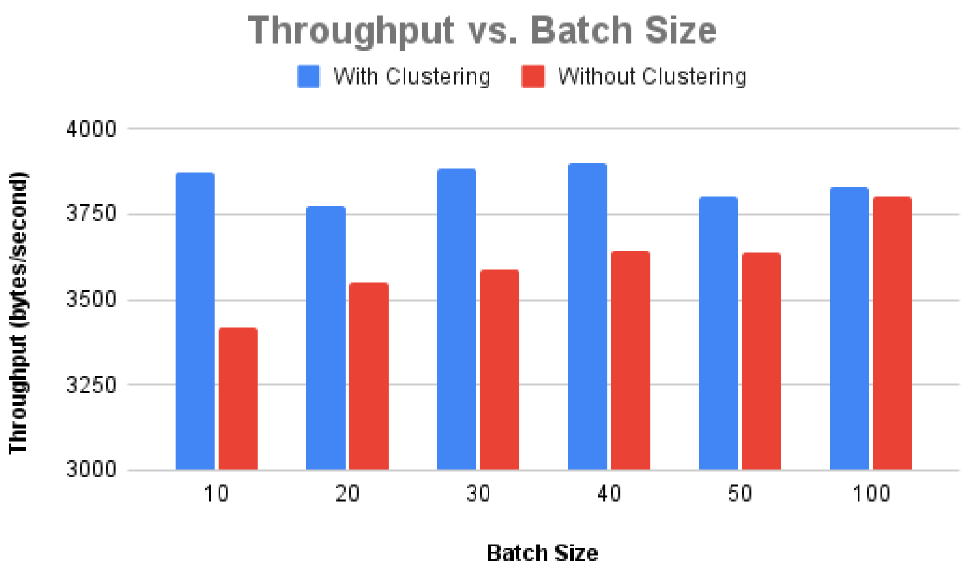


Figure 6.3: Throughput vs Batch Size

It is clear that a system with clustering has significantly higher throughput than a system with no clustering. The reason behind this is when there is no clustering, an NF has to send state information to all the other NFs in the system after each batch. For example, in our experiment, there were 12 NFs in total and so, after each batch, an NF has to send state information to 11 other members. This incurs a high overhead. On the other hand, an NF in the clustered system has to send state information to only 3 other NFs after each batch. Thus, the overall packet processing time is much higher in the non-clustered system and this holds true no matter how much we vary the batch size.

6.2.2 Throughput vs Packet Rate

The following parameters were used in this experiments:

Parameter	Value
Total Packets Processed	4000
Batch Size	30
Number of Cluster	3
Number of Network Functions in each cluster	4
Number of Network Functions in total	12

We can observe the following results under these parameters.

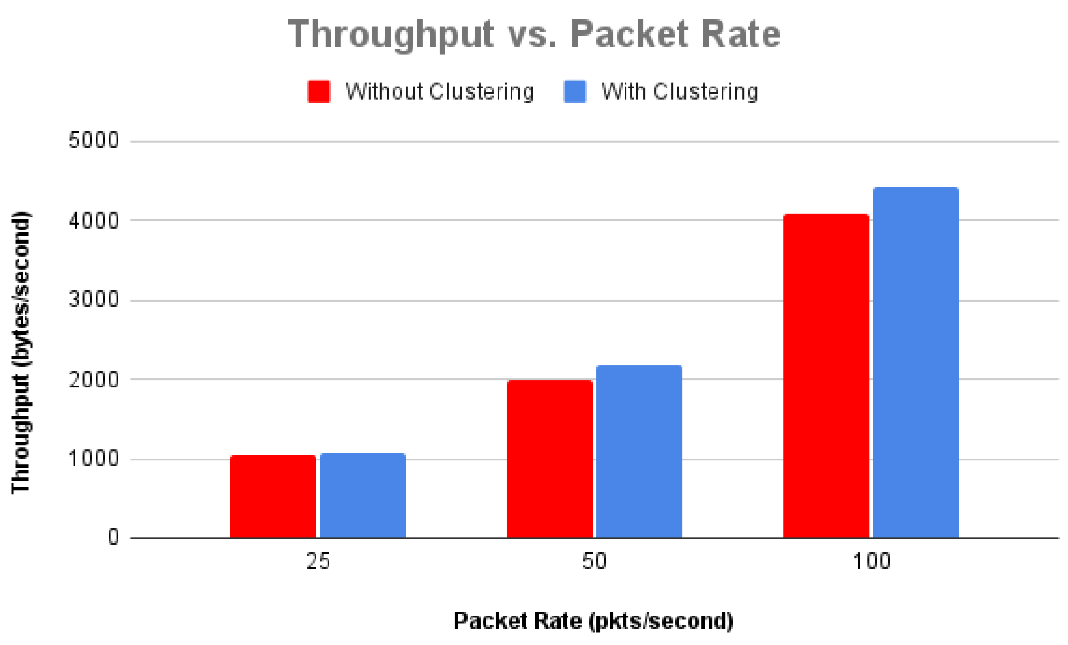


Figure 6.4: Throughput vs Packet Rate

Again, a clustered system shows higher throughput than a non-clustered system. In every packet rate, we can see this trend. The difference of throughput between these systems becomes more evident as we increase the packet rate. Moreover, packet rate plays a significant role in the throughput for both systems. As we can see, throughput increases for both systems as we keep increasing the packet rate. It is because faster arrival of packets ensure quicker insert into the input buffer.

6.3 Results of State Migration Overhead in Clustering

For this experiment, we consider a 2-cluster system where each cluster contains 2 NFs. NF-1 and NF-2 are located in Cluster-1 and NF-3 and NF-4 are located in Cluster-2.

The following parameters were used in this experiments:

Parameter	Value
Batch Size	30
Packet Rate	50 pkts/second
Number of Cluster	2
Number of Network Functions in each cluster	2
Number of Network Functions in total	4

Under these model parameters, we get the following results. In the following graphs, a blue bar indicates the overhead while sending state information to an NF's own cluster member, a yellow bar indicates the overhead while receiving state information from own cluster member and a purple bar indicates the overhead while receiving state information from an NF outside own cluster.

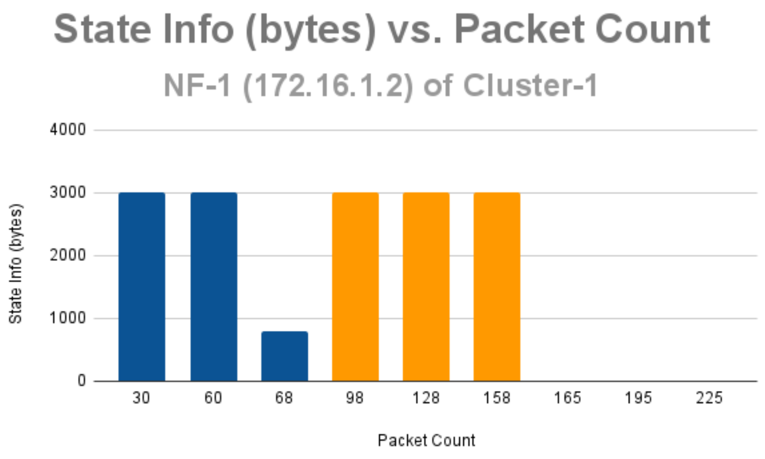


Figure 6.5: State info for NF-1

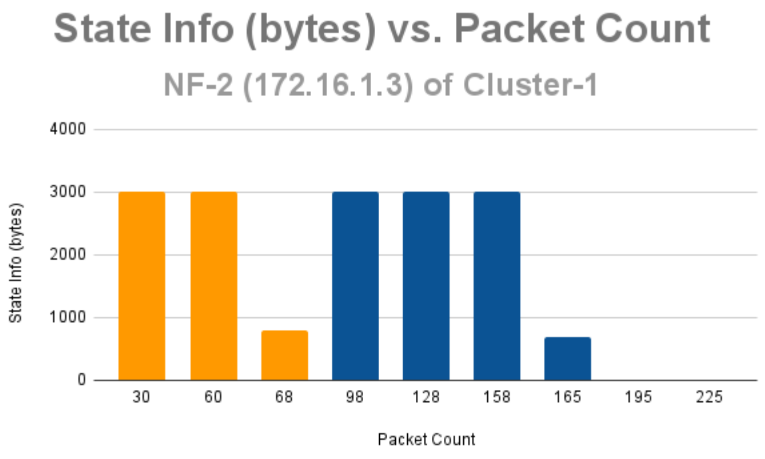


Figure 6.6: State info for NF-2

From 6.5, we can see that when NF-1 processes packets of Flow-1, it sends Flow-1's state information to NF-2 after every batch (30 packets). After 68 packets, a migration scenario occurs. As state information of 60 packets has already been sent to NF-2, the state information of only 8 packets is necessary to be sent to NF-2. Now Flow-1 is migrated to NF-2 and so, NF-1 continues to receive state information of Flow-1 from NF-2 after every batch processing.

Similarly, from 6.6, we can see that at first NF-2 receives state information of Flow-1 from NF-1 after every batch. After 68 packets, Flow-1 is migrated to NF-2 and so NF-2 continues to send state information to NF-1 after every batch. At 165 packets, an inter-cluster migration event occurs and Flow-1 now needs to be migrated to NF-3 of Cluster-2. As state information of 158 packets has already been sent to NF-1, NF-2 needs to send state information of only 7 packets to NF-1.

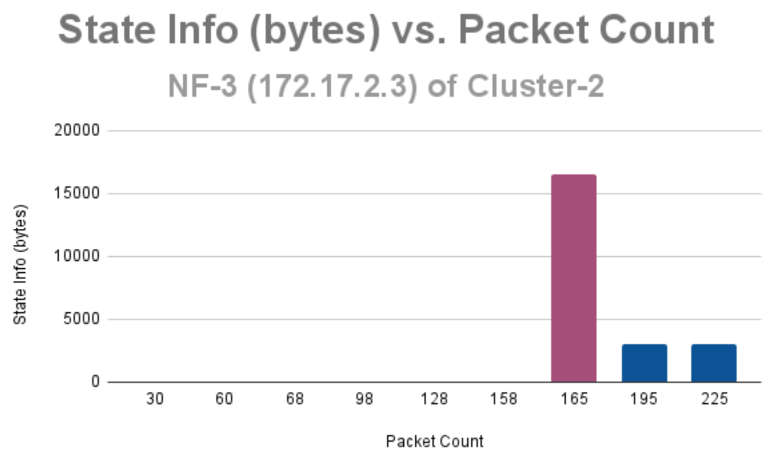


Figure 6.7: State info for NF-3

Finally, from 6.7, we can see that NF-3 receives state information of all 165 packets processed so far because NF-3 is in a different cluster and it has no information about Flow-1. After that, NF-3 continues to operate normally.

In conclusion, an intra-cluster flow migration incurs significantly low migration overhead than an inter-cluster flow migration. Hence, implementing clustering is very benificial as it significantly reduces intra-cluster migration overhead.

6.4 Results of Load Balancing

In this section, we observe how the cluster-based load balancing algorithm affects the performance of the system.

6.4.1 Latency vs Packet Rate

The following parameters were used in this experiments:

Parameter	Value
Total Packets Processed	4000
Batch Size	30
Number of Cluster	3
Number of Network Functions in each cluster	3
Number of Network Functions in total	9

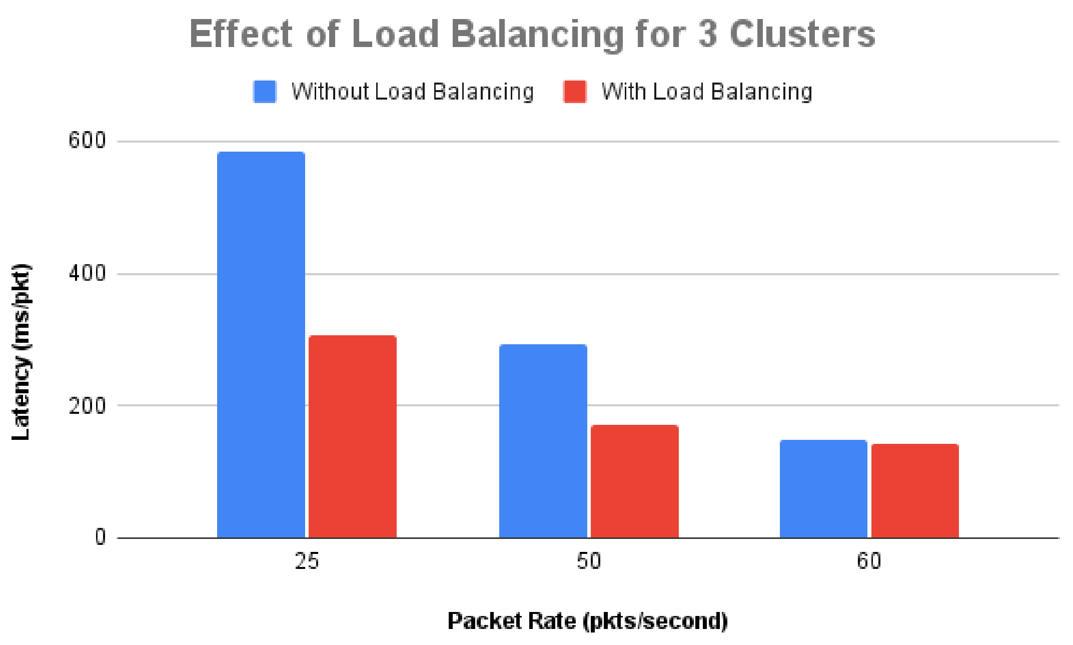


Figure 6.8: Effect of Load Balancing for 3 Clusters

Under these model parameters, we get the following results.

A system with cluster-based load balancing has significantly lower latency than a system without load balancing. When load balancing is implemented, no NF is overloaded for long. As soon as an NF is overloaded, flows are migrated to another NF. So NFs are utilized properly and they can process packets faster. As a result, the overall latency of the system decreases in the presence of cluster-based load balancing.

6.5 Summary

From these experiments, firstly, we can see that applying multithreading for parallel flow processing decreases the latency and increases the throughput of the system. Secondly, a cluster-based system provides better throughput than a non-clustered variant, while also ensuring reduced migration overhead. Finally, adopting a cluster-based load balancing algorithm results in increased efficiency of the system.

Chapter 7

Future Work

There are a number of directions for future research that can further improve the capabilities and performance of the system, even though our current work has made great progress in tackling the issues of faster packet processing, less migration overhead, and effective state management. We discuss prospective research areas in this section that could lead to advances.

1. **Cluster-Based Global States:** Future research can look into the viability and advantages of cluster-based global states rather than relying on network-wide global state updates. With this strategy, the network is divided into clusters, with each cluster keeping its own global state repository. We may be able to lessen the cost of network-wide synchronization and increase the system's scalability by localizing global state changes. Exploring effective mechanisms for cluster-based global states may result in higher fault tolerance and more efficient resource management.

2. **Comparative Analysis of Custom Load Balancer:** It would be beneficial to compare our custom load balancer to other load balancing methods that are currently in use. Evaluation of elements including performance, scalability, fault tolerance, and resource consumption may be part of this comparison study. We may learn about our load balancer's advantages, disadvantages, and potential areas for improvement by comparing it to other cutting-edge systems. This research can direct future improvements and allow for the creation of load balancing techniques that exceed current methods.

3. **Using Various Consensus Algorithms:** Future study can investigate the use of various Consensus Algorithms to Reach Consensus on Global State Updates. Consensus algorithms are essential in distributed systems because they ensure participant consistency and agreement. We can assess alternative consensus algorithms' applicability, effectiveness, and resilience in the context of elastic scaling and fault tolerance by integrating them into the state management system, such as Paxos or Raft. This investigation may open

the door to state management methods that are more reliable and effective in dynamic network situations.

4. **Achieving Strict Consistency:** Investigating methods to ensure strict consistency of global states in the absence of a centralized controller and during elastic scaling events is a key area for future research. This calls for the creation of novel protocols and algorithms that provide synchronized state updates across NF instances, avoiding conflicts or inconsistencies in the global state. We can improve the system's correctness and dependability even in highly dynamic and dispersed contexts by tackling this problem.

5. **Comparative Analysis of Performance:** Performance is a key concern in such systems, even though ours address all the lacking features of the than the majority of cutting-edge systems in terms of design. To guarantee our system's feasibility and competitiveness in real-world circumstances, it becomes essential to compare its performance to that of existing solutions.

Chapter 8

Conclusion

In this study, we have developed a thorough state management system that tackles the issues of quicker packet processing, low migration overhead during elastic scaling, and effective resource usage. Our system integrates enhancements including multithreading for faster packet processing, intelligent load balancing for efficient resource use, and clustering methods for efficient state sharing between clusters. Our technology meets the expectations of achieving faster packet processing, which improves the responsiveness and throughput of the networking infrastructure. While simultaneously addressing the issue of migration overhead during elastic scaling, we have ensured smooth scaling events with little interference to ongoing flows. Our method uses clever tactics to manage in-flight traffic, preventing packet loss and guaranteeing reliable packet delivery. The future works section has outlined possible directions for additional research, such as achieving strict consistency of global states, improving state management strategies during elastic scaling events, comparing load balancing tactics, and investigating alternative consensus algorithms. These potential directions for further research will aid in the continued creation of more durable and effective solutions in the industry.

References

[1] "What is a network function - sdxcentral,"

[2] A. Alwakeel, A. Alnaim, and E. Fernández, "A pattern for a virtual network function (vnf)," pp. 1–7, 08 2019.

[3] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," *ACM SIGCOMM Computer Communication Review*, 08 2014.

[4] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless Network Functions: Breaking the Tight Coupling of State and Processing," 03 2017.

[5] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, (USA), p. 299–312, USENIX Association, 2018.

[6] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proceedings of the 4th annual Symposium on Cloud Computing*, pp. 1–15, 2013.

[7] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, *et al.*, "Rollback-recovery for middleboxes," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 227–240, 2015.

[8] S. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu, "Reinforce: Achieving efficient failure resiliency for network function virtualization-based services," *IEEE/ACM Transactions on Networking*, vol. PP, pp. 1–14, 02 2020.

[9] M. M. Shahriyar, G. Saha, B. Bhattacharjee, and R. Reaz, "DEFT: Distributed, Elastic, and Fault-tolerant State Management of Network Functions," *Undergraduate thesis of Dept of Computer Science and Engineering, BUET.*

[10] T. A. Srinivas, A. Donald, G. Thippanna, M. Kousar, and T. Murali, "Nfv and sdn: A new era of network agility and flexibility," *International Journal of Advanced Research in Science, Communication and Technology*, pp. 482–493, 02 2023.

[11] Papavassiliou, "Software defined networking (sdn) and network function virtualization (nfv)," *Future Internet*, vol. 12, p. 7, 01 2020.

[12] T. Marsl, Y. Gao, and F. Lau, "A study of software multithreading in distributed systems," 12 1995.

[13] W. Ma, J. Beltran, Z. Pan, D. Pan, and N. Pissinou, "Sdn-based traffic aware placement of nfv middleboxes," *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, pp. 528–542, 2017.

[14] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," pp. 227–240, 04 2013.

[15] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," 10 2013.

[16] R. Szabo, M. Kind, F.-J. Westphal, H. Woesner, D. Jocha, and A. Csaszar, "Elastic network functions: opportunities and challenges," *IEEE Network*, vol. 29, no. 3, pp. 15–21, 2015.

[17] "Docker 23.0." `https://docs.docker.com/engine/release-notes/23.0/`.

[18] S. N. Mehmood, N. Haron, V. Akhtar, and Y. Javed, "Implementation and experimentation of producer-consumer synchronization problem," *International Journal of Computer Applications*, vol. 975, no. 8887, pp. 32–37, 2011.

[19] "Python Queue." `https://docs.python.org/3/library/queue.html`.

[20] "Hazelcast." `https://github.com/hazelcast/hazelcast`.

[21] "Packet Sender." `https://packetsender.com/`.

Appendix A

Result Tables

Data for various graphs in this paper are given below along with their figure number.

A.1 Table for Latency vs Number of Threads

Number of threads	Latency (ms/pkt)
1	55.372
2	52.599
3	45.395
4	40.227
5	36.528

Table A.1: Data for figure 6.1

A.2 Table for Throughput vs Number of Threads

Number of threads	Throughput (ms/pkt)
1	424.5608
2	680.8253
3	742.922
4	810.796
5	815.686

Table A.2: Data for figure 6.2

A.3 Table for Throughput vs Batch Size (Cluster vs No Cluster)

Batch Size	Throughput with Cluster (ms/pkt)	Throughput without Cluster (ms/pkt)
10	3871.31978	3420.3238
20	3775.057	3551.51
30	3886.5038	3590.37
40	3898.136	3645.6754
50	3803.9816	3637.2304
100	3831.2319	3800.7672

Table A.3: Data for figure 6.3

A.4 Table for Throughput vs Packet Rate (Cluster vs No Cluster)

Packet Rate	Throughput with Cluster (ms/pkt)	Throughput without Cluster (ms/pkt)
25	1081.4915	1051.6907
50	2173.2047	1988.6341
100	4408.8377	4082.9092

Table A.4: Data for figure 6.4

A.5 Table for Migration Overhead (NF-1)

Packet Count	State Information (bytes)
30	3000
60	3000
68	800
98	3000
128	3000
158	3000
165	700
195	0
225	0

Table A.5: Data for figure 6.5

A.6 Table for Migration Overhead (NF-2)

Packet Count	State Information (bytes)
30	3000
60	3000
68	800
98	3000
128	3000
158	3000
165	700
195	0
225	0

Table A.6: Data for figure 6.6

A.7 Table for Migration Overhead (NF-3)

Packet Count	State Information (bytes)
30	0
60	0
68	0
98	0
128	0
158	0
165	16500
195	3000
225	3000

Table A.7: Data for figure 6.7

A.8 Table for the Effect of Load Balancing

Packet Rate	Latency with Load Balancing (ms/pkt)	Latency without Load Balancing (ms/pkt)
25	307.743	583.2177
50	171.452	291.975
100	142.386	147.3667

Table A.8: Data for figure 6.8