Implementing and Evaluating Prism in the NS3 Network Simulator

Yumiao Sui

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

This thesis presents the implementation and evaluation of Prism in the NS3 simulator, because, although Prism is a novel approach that eliminates the problems with traditional proxy systems, it is hard to experiment in a real testbed at a scale due to the need for a large number of machines and high bandwidth networks. We describe our implementation of Prism in the NS3 network simulator in detail, and report experimental results with up to 32 nodes in simulation.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Yumiao Sui*)

Acknowledgements

I gratefully acknowledge my advisor, Professor Michio Honda, for his professional advice, guidance and encouragement. Although I feel that this sentence alone could encapsulate all my thanks, I'd like to take this opportunity to share some thoughts that I usually don't get to express.

I was always a person without direction, unsure of what I truly wanted to pursue. My advisor, in stark contrast, is always optimistic, full of energy, and extremely passionate about his work. His qualities not only gave me the direction but also empowered me.

Despite being aware of my shortcomings, he never ceased to encourage me. There were countless times when I wanted to give up, but each time, thinking of his encouragement and support, I would pick myself up and choose to keep going.

Particularly when I first started this project, I had almost no foundational knowledge related to it. I can't imagine how I could have reached this point today without his careful guidance and selfless support.

In summary, he made me realize that no matter how great the difficulties and setbacks are, as long as you have firm belief and the right guidance, you can overcome them. Therefore, I would like to take this moment to offer him my most sincere thanks and respect.

Table of Contents

Chapter 1

Introduction

Prism [1] is a new approach to enabling scale-out systems, which dynamically adjust the service capacity by adding or removing backend machines in the cluster. Unlike Layer 4 (L4) load balancers, such as Google Maglev [2], it allows active TCP connections to migrate to a different backend without the client to notice, enabling better cluster resource utilization and service paritioning or sharding. Unlike Layer 7 (L7) load balancers or proxies, such as HAProxy [3], Prism allows the backends to communicate with the client without their data transfer being mediated by the frontend. Therefore, Prism reduces the CPU and network load of proxies, which is caused by data copy, encryption and host link bandwidth usage.

Unfortunately, further research with Prism is difficult. Production systems are complicated and diverse in hardware specification and configuration, and application behavior. It is an open question how Prism could perform in a range of deployments, because Prism was only evaluated in a small cluster that consists of three physical servers that partition their resources into six dual-core logical servers, also using just two applications. Prism has fundamental performance trade-off based on the connection handoff overheads, and, by design, it would even decrease the end-to-end performance of a traditional proxy-based system depending on workloads. Connection handoff overheads are hard to characterize, because it involves many operations that incur processing or network latency, or both, making validation of generality and practicality of Prism hard. Even if hardware resources, such as small testbed, public cloud and CloudLab, are available, it is hard to start a new research project with Prism, because available configuration and hardware are still limited or expensive. Further, although the source code of Prism is publicly available, it is a research prototype that is difficult to run in a different platform.

To enable further research with Prism without using a large amount of hardware resources or building a specific system from scratch, we implement Prism in NS3 [4], a widely used network simulator. In addition to enabling large scale experiment with implementing an applications in the simulator, which is easier than doing so in the real operating system, simulation allows us to analyze the impact of connection handoff overheads, which is unique to Prism, on end-to-end performance, because we can easily increase or decrease the delay of any operation. Since none of existing simulators, including NS3, support the TCP features required by Prism and available in Linux, such as `TCP_REPAIR` and `TCP_INFO`, we extend the NS3 network simulator to implement those features, which is one of the main contributions of this thesis. We use our custom NS3 simulator to analyze the impact of connection handoff overheads on end-to-end performance at different cluster scales.

We find that the proxy implementation in NS3 was limited in throughput, while the Prism implementation scaled better with the addition of more backends. The presence of connection handoff overheads was confirmed to be a crucial factor affecting performance in the Prism simulation.

Chapter 2

Background

2.1 TCP

Transmission Control Protocol (TCP) is an integral pillar of modern networking. Delving into its core features and understanding its challenges offer insights into its ubiquitous application in today's digital world.

One of the most foundational attributes of TCP is its connection-oriented mechanism [5]. Unlike its counterpart, the User Datagram Protocol (UDP), TCP establishes a connection before any data transmission commences. Functions like `connect()` are essential in this handshake process, ensuring both sender and receiver are synchronized for impending communication. This setup ensures a stable and dedicated communication channel, vital for applications requiring reliability.

In the vast and unpredictable realm of networks, data packets can be lost, duplicated, or delivered out of order. TCP's retransmission strategy ensures that lost packets are resent, bolstering the protocol's reliability [6]. This feature is underpinned by using acknowledgement for received data. The absence of an acknowledgement within a specified timeframe triggers retransmission, ensuring data integrity.

TCP does not perceive data as discrete packets but as a continuous bytestream [6]. This abstraction simplifies the process for higher-level applications. Irrespective of how data is segmented during transmission, the receiving end reconstructs the original bytestream, rendering the underlying complexities invisible to the end-users and applications.

Networks can become congested, much like road traffic during peak hours. TCP's congestion control mechanisms adjust the rate of data transmission based on network conditions [7, 8]. Through algorithms such as `Reno` and `CUBIC`, TCP dynamically tweaks the data flow, ensuring the network isn't overwhlemed, leading to a more efficient and fair utilization of resources [9, 10].

TCP's robust suite of features has cemented its position as a pivotal protocol in the networking world [6]. Whether it's browsing a website, sending an email, ot transferring a file, TCP is invariably involved. Methods like `getsockopt()` and `setsockopt()` exemplify its versatility, providing applications a range of options to customize their networking behaviour [11]. Its reliability make TCP the protocol of choice for a vast majority of Internet applications.

However, a limitation lies in TCP's inability to adapt to changing addresses or ports in the connection. The rigidity can become a hindrance in environments like mobile networks where IP addresses can frequently change. Once a TCP connection is established using the initial IP address and port, altering them disrupts the connection. This behavior is intrinsic to its design; the tuple of source IP, source port, destination IP, and destination port defines the connection [6]. While this ensures security and stability, it lacks the adaptability demanded by certain modern use-cases.

TCP stands as a testament to the intricate balance between reliability and adaptability in the digital world. It's been our steadfast companion in ensuring our digital exchanges are dependable. Like any tried-and-true method, it has areas to improve, but its enduring presence highlights its significance in our connected age.

2.2 Layer 7 Load Balancer

In the landscape of networking, the Layer 7 load balancer (L7LB), commonly known as an application load balancer or simply a proxy, plays a vital role in managing internet traffic. It stands at the application layer of the OSI model, ensuring that incoming requests from clients are seamlessly distributed across various backend servers [12, 13].

As its essence, the L7LB functions by delving deep into the content of data packets [12]. deciphering the user-friendly URLs, or even perusing the intricate details embedded in HTTP headers. It does not simply redirect traffic; it reads, understands, and then makes informed decisions about where that traffic should go. For instance, it can route a client's request based on the content type, be it video, images, or textual data.

In terms of its position within network topology, the L7LB typically situates itself between client devices (like computers or mobile devices) and backend servers [12]. When a client sends a request, it first reaches the L7LB. The balancer, after processing the request's content, forwards it to the most appropriate backend server. This ensures not only efficient load distribution but also optimizes the client's experience by minimizing response times and avoiding server overloads.

Layer 7 load balancers and Layer 4 load balancers (L4LB) both aim to distribute traffic efficiently, but they operate at different layers and have distinct mechanisms [12, 14, 15]. While L7LBs function at the application layer, focusing on content like HTTP headers, L4 load balancers operate at the transport layer. L4LBs are more concerned with information like IP addresses and port numbers. Their decisions are based on simpler, more foundational data about the traffic, rather than its content. In essence, L4LBs see the source and destination of the data while L7LBs comprehend the actual content of the data. As a result, L7LBs can provide more sophisticated and content-based load balancing decisions, such as directing all video requests to a specific set of servers optimized for video streaming.

Despite its advanced capabilities, the L7LB is not devoid of challenges. A notable problem is the inherent latency in its operation. Given that the L7LB acts as an intermediary, it doesn't just passively forward traffic [12]. Instead, it actively relays requests to the backends and then waits to relay the response back to the clients. Its strength, which lies in deeply inspecting packets, can also be its Achilles' heel. This deep inspection often requires more computational resources, potentially introducing latency. Furthermore, as it deals with application-level data, L7LBs can become targets for application-level attacks, demanding robust security mechanisms. There's also matter of complexity: the sophisticated decisions L7LBs make require configurations. An incorrect setting can lead to misrouted traffic or even render an application inaccessible.

2.3 Prism

Prism is an innovative approach tailored to amplify the performance of object storage systems. At its essence, Prism strives to harmoniously integrate the adaptability and safety of age-old frontend proxy designs with the robustness and agility of today's key-value stores [16, 1]. These key-value repositories are fine-tuned for minimal I/O operations, leveraging specialized UDP-based protocols within data center environments.

A distinguishing facet of Prism is its pioneering connection migration mechanism. The iterative TCP connection transition stands as a central principle, allowing a TCP connection to seamlessly shift across various machines during its active phase [1]. This capability enables the frontend to scrutinize requests without the necessity to forward extensive data or associated payloads, effectively countering a significant limitation of conventional proxy designs.

By using this hand-off, Prism can distribute the ”I/O, compute, and network bandwidth usage across the backends” [1]. This distribution helps in bypassing data path restrictions and ensures smoother and more efficient data transfer.

The practicality of Prism’s hand-off technique is anchored in two cutting-edge tech evolutions: TCP state serialization for restoring the TCP socket and programmable switches for redirecting packet to the backend [1]. The hand-off method of Prism resonates with the TCP state serialization capability present in the latest Linux iterations, a feature also on the horizon for platforms like FreeBSD. The emergence of granular state management methods for adaptable switches equips Prism with the prowess to adeptly oversee a multitude of simultaneous flows.

`TCP REPAIR`, introduced into the Linux kernel around version 3.5 as a socket option, is a feature for handling TCP connections [11]. Sometimes, due to various reasons, a connection might need to be moved from one place to another. Instead of tearing down the connection and starting a new connection from scratch, `TCP REPAIR` lets us pause the current connection, shift it to where it needs to go, and then continue right from where it left off. It enables applications to seize control over the TCP connection, allowing for relocation of network connections. By saving and restoring all the TCP connection state, `TCP REPAIR` enables TCP connection migration to another host.

In Linux, putting a TCP socket into the repair mode is done using `setsockopt()` with the `TCP REPAIR` argument [11]. The kernel then freezes the socket by stopping and allows the user to manipulate its internals. The user can retrieve and save the current state of the TCP connection, including the negotiated TCP options (e.g., maximum segment size, window scale, timestamp), sequence numbers to send or receive data, and data in send or receive queues [11]. When the socket is in the repair mode, the Linux kernel closes the socket without sending `FIN` or `RST` packets, simply calling `tcp_disconnect()`. After a new socket is created on the target backend and put into the repair mode, the socket can be restored for the continuity of the TCP connection by calling `setsockopt()` with the saved information and corresponding `TCP REPAIR` options.

To migrate the TCP connection, a new TCP socket can be created on the target backend. The user then can put the new socket into the repair mode and use the saved information to restore the socket’s state to a similar condition as the socket on the original backend [11]. Once the restoration process is complete and state is fully recovered, the user can switch off the repair mode, and the kernel of the target backend will resume normal operation of the TCP connection as if it had never been interrupted.

TCP REPAIR stands as a cornerstone in fortifying the reliability and continuity of TCP connections. By preserving the socket state, it facilitates the safeguarding and subsequent recovery of the connection's status. This capability not only negates the need for initiating new TCP connections but also enhances the resilience of internet communications.

For Prism, a programmable switch contributes to enhancing the efficiency and flexibility of data flow. The programmable switch in Prism is tasked with managing and directing data traffic [1]. It works in tandem with Prism's unique connection hand-off protocol, allowing TCP connections to be smoothly migrated between different machines. This ensures that data can be processed more efficiently, without unnecessary relays or interruptions.

Furthermore, with the advent of advanced state management techniques, these programmable switches can handle a vast number of concurrent data flows. To ensure data is delivered promptly without any hindrances, the switch is tailored using custom UDP packets [1]. These packets provide directives on how the switch should adjust the IP addresses and port numbers. For incoming packets from the client, the switch is instructed to alter the packets' destination IP addresses, ports, and MAC addresses to match those of the backends. Conversely, for outgoing packets directed to the client, the switch modifies the packets' source IP addresses and ports to align with the frontends.

The hand-off protocol introduced by Prism not only guarantees superior data rates and response times compared to its traditional counterparts but also sets the stage for a new era of streamlined and fortified object storage infrastructures in the times ahead.

2.4 NS3

NS3 is a discrete-event network simulator that operates primarily through scripting, typically using C++ or Python [17, 18, 19]. It doesn't come with a built-in graphical user interface (GUI) for designing or running simulations. Instead, users write scripts to define the network topology, set parameters, and run simulations.

NS3 allows users to design network topology with ease. In NS3, creating a network topology is akin to crafting a story using node [4]. Using the `NodeContainer` class, users can instantiate as many nodes as they need. Next, define the interconnections between nodes. Depending on the desired link type—be it point-to-point, Wi-Fi, or others—utilize the appropriate helper classes, such as `PointToPointHelper` or

`WifiHelper`, to facilitate the configuration of the physical and MAC layers. Essential attributes, including data rates and delays, can be fine-tuned to optimize the topology. Upon establishing nodes and their links, the `InternetStackHelper` class is used to equip each node with an Internet stack. IP addresses can then be assigned via the `Ipv4AddressHelper` class. With the configurations in place, the NS3 script now presents a well-defined network topology, prepared for subsequent simulations and analyses.

In NS3, establishing a TCP connection between two nodes involves creating socket instances for both nodes using the `TcpSocketFactory` and the `Socket::CreateSocket` method [4]. The server node binds it socket to a specific address and port using the `Bind()` method and then invokes the `Listen()` method to await incoming connection requests. The client node uses the `Connect()` method to initiate a connection to the server's address and port.

To manage different stages of the TCP connection and ensure a structured and efficient communication process in NS3, callbacks are extensively used to handle various events that can occur during the lifecycle of a TCP connection [4]. For instance, the `SetConnectCallback` is employed to handle both successfully connection establishments and failed attempts. For server-side sockets, `SetAcceptCallback` is used for handling incoming connection requests. It has two associated functions: the first is triggered when a new connection request is received, allowing the server to decide whether to accept or reject the connection; the second is invoked once the connection is successfully established with the client. After the TCP connection is established, the client can send data using `Send()` method. When data is ready to be read from a socket, the `SetRecvCallback` is invoked, while the `SetDataSentCallback` is triggered when a packet has been sent from transport protocol. Through these callbacks, NS3 adeptly handles asynchronous network events, negating the need for constant socket or connection status checks and ensuring an efficient, event-driven simulation.

Chapter 3

Problem Statement

In the realm of modern networking, the advent of Prism has introduced a novel approach to enabling scale-out systems. By dynamically adjusting service capacity through the addition or removal of backend machines in a cluster, Prism promises enhanced resource utilization and efficient service partitioning. However, as with any innovative technology, Prism is not without its challenges and uncertainties.

Prism's unique capability allows active TCP connections to migrate to a different backend without the client noticing, thereby optimizing cluster resource utilization. However, this introduces connection handoff overheads, which have a performance trade-off. These overheads are challenging to characterize as they encompass numerous operations that can introduce processing or network latency, or even both. The intricate nature of these operations makes it difficult to ascertain their overall impact on end-to-end performance. For instance, while Prism was evaluated in a small cluster comprising three physical servers partitioned into six dual-core logical servers, its performance in diverse and larger deployments remains an open question. The potential decrease in end-to-end performance, depending on the workloads due to these overheads, further clouds Prism's generality and practicality.

The scalability of Prism, especially in larger deployments, is yet to be thoroughly understood. While simulations can offer insights into its performance in large-scale experiments, the real-world implications of connection handoff overheads, especially in extensive clusters, remain ambiguous.

In conclusion, while Prism presents a promising solution to some of the challenges faced by modern networking systems, its full potential can only be realized once these uncertainties are addressed. A comprehensive understanding of the impact of connection handoff overheads on end-to-end throughput and latency, as well as the performance characteristics in large-scale systems, is crucial for its widespread adoption and optimization.

Chapter 4

Design

To address the problems described in the previous Chapter, we evaluate Prism using a custom simulator. We extend NS3 by implementing a set of TCP features that are absent in the NS3 TCP implementation although required for Prism and available in Linux. Some features, such as extracting connection status, are trivial to implement, but those modifying TCP behavior, such as `TCP REPAIR`, are not, as described in the rest of this chapter.

4.1 TCP REPAIR

We implement `TCP REPAIR` in NS3. To facilitate the use of `TCP REPAIR` in NS3, we have introduced several modifications and new methods. Unlike Linux, which toggles `TCP REPAIR` using the `setsockopt` interface, NS3 lacks such a native interface. To address this gap, we implement a new method called `SetTcpRepair` within the `TcpSocketBase` class, which instantiates a TCP connection. The socket instance can be cast to this specialized class for the activation or deactivation of repair mode.

We add a flag named `m_repair` to the `TcpSocketBase` class, which serves as an indicator for the repair mode status of the connection. We choose to extend `TcpSocketBase` rather than `TcpSocket` because the former provides the concrete implementation of TCP functionalities, while the latter merely defines a set of common interfaces that can be employed across various TCP implementations supported by NS3 (NS3 accommodates multiple TCP implementations, unlike Linux). We use that flag to enable operations specifically allowed in the repair mode. Since TCP connections do not send and receive data, nor do they process data in the repair mode in Linux, we modify the NS3 TCP implementation to suppress regular packet transmission and reception based on this flag. Likewise, since no `FIN` or `RST` are sent when the socket is closed in the repair mode in Linux, we modified the `close` implementation in NS3 to simply change its TCP state to `CLOSED` after calling `DeallocateEndpoint()` if the socket is in the repair mode.

We then implement the `GetRepairWindow` method that retrieves window-related information from the repair-mode socket. It provides the same functionality with `getsockopt` with `TCP_REPAIR_WINDOW` in Linux. Since Linux returns the window parameters in `struct tcp_repair_window`, we define an equivalent structure, `TcpRepairWindow`. The Linux implementation also provides information regarding the received urgent pointer through `TCP REPAIR_WINDOW`. However, we have deferred its implementation in NS3 to future work, considering that the use of urgent pointers is increasingly rare and not pertinent to Prism.

Finally, we implement a method named `SetTcpRepairQueue`. It specifies which queue, which is either send or receive queue, will be targetted by subsequent queue operations. The selected queue is internally recorded in the `m_repairQueue` variable and can be queried using the `GetTcpRepairQueue` method. `GetTcpQueueSeq` returns the sequence number being used for the new data transmission (`m_tcb->m_nextTxSequence()`) when the queue has been set to `TCP_SEND_QUEUE`; it returns the sequence number expected in the next in-order segment (`m_tcb->m_rxBuffer->NextRxSequence()`) when it has been set to `TCP_RECV_QUEUE`.

4.2 Other Socket Operations

TCP connection migration needs a number of operations in addition to those specific to repair-mode sockets. Fortunately, we found that most of them are already available or trivial to implement in the `TcpSocketBase` class, unlike the repair-mode operations described in the previous section.

We need local and remote port numbers when restoring the connection. We can use built-in methods, `GetSockName` and `GetPeerName` instead of `getsockname` and `getpeername` in Linux. The restored connection needs the maximum segment size. It can be retrieved using `GetAttribute()` with the `SegmentSize` attribute. To retrieve the latest `TSval` found in the Timestamp option [20] received from the peer, which the restored connection should echo, we implement the `GetTimestampToEcho` method that simply reads the `m_timestampToEcho` in `TcpSocketBase`.

When restoring the connection, we also need the scaling factor of the advertised window that has been negotiated over the Window Scale option during the connection setup. In Linux, it can be retrieved as a part of `TCP INFO` in `getsockopt`. Although we would simply read the scaling factor value (`m_rcvScaleFactor`), we opted for implementing entire `TCP INFO` in NS3, because of future extensibility.

4.2.1 TCP Connection Serialization

TCP connection serialization plays a vital role in the migration of TCP connections. It is the process of transforming the state of a TCP connection into a format suitable for storage and future use. The state of a TCP connection encompasses crucial information, as we described before, including sequence numbers for the next byte to be sent or received, TCP options, congestion window status, and more. Through serialization, the intricate state is converted into a compact, storable format that captures all the necessary details. The serialized state can then be saved, allowing for the recreation and continuity of the original TCP connection at a later time.

Within NS3, serialization also holds significant importance in evaluating Prism because it enables operations to retrieve and replicate the TCP connection state of a socket and helps with the restoration of a new socket on the target backend to the required state.

Built-in methods such as `GetSockName()` and `GetPeerName()` offer basic socket information, delivering the port numbers and addresses of the socket and its remote peer, respectively. To enhance this, we add five specialized methods in NS3 that emulate the capabilities of the `getsockopt()` system call for a comprehensive snapshot of the connection state:

- `GetTcpInfo()` extracts a broad set of TCP metrics and returns them within a `TcpInfo` struct.
- `GetRepairWindow()` fetches the TCP repair window state and encapsulates it in a `TcpRepairWindow` struct.
- `GetTimestampToEcho()` retrieves the timestamp currently set to be echoed by the socket.
- `GetTcpRepairQueue()` identifies the active queue (either send or receive) when the socket is in TCP repair mode.

- `GetTcpQueueSeq()` discloses the next sequence number for either sending or receiving data, depending on the active queue.

In NS3, when we establish a TCP connection between two nodes, the accepted socket can be dynamically cast to `TcpSocketBase`. By setting `m_repair` to 1 through `SetTcpRepair()` and invoking aforementioned methods, we can extract the serialized data of the socket. Furthermore, built-in methods `GetTxBuffer()` and `GetRxBuffer()` provide details about the send queue and the receive queue. Once the retrieved data is validated, we can save it in the format returned by these methods for precise deserialization at a later time. The formats used to store the socket state, like `TcpInfo` and `TcpRepairWindow` structures, contain sufficient detail to facilitate a comprehensive restoration of the TCP connection state.

TCP connection serialization serves as the foundation of our `TCP REPAIR` implementation in NS3. By capturing and storing the TCP connection state as serialized data, we gain the ability to pause and resume the TCP connection without noticing the client.

4.2.2 TCP Connection Restoration

TCP connection restoration is an essential feature for facilitating the live migration of TCP connections between hosts. Simply closing and reopening a TCP socket would inherently disrupt ongoing network communications. This could be especially problematic for real-time or mission-critical applications. Restoring the TCP connection ensures that the communication remains uninterrupted after migration.

The restoration process commences by capturing the existing state of a TCP socket through serialization. This state is subsequently replicated at the destination host through deserialization. A new socket is instantiated on the destination host and immediately placed into repair mode. This state allows for manual adjustments to the socket's internal parameters, leveraging the serialized data to reconstruct the original socket's state.

In the NS3 environment, a newly created socket on a different node can be dynamically cast to the `TcpSocketBase` class and set into repair mode. Following this, we employ a series of specialized methods to gain granular control over the connection state. These methods emulate the functionalities available through Linux's `setsockopt()` but are tailored for NS3:

- `SetTimestampToEcho()` specifies the timestamp to be echoed in subsequent TCP acknowledgments.

- `SetTcpInfo()` adjusts the socket's internal state based on the supplied `TcpInfo` structure.
- `SetHighTxMark()` establishes a new high-water mark for the transmission sequence number.
- `SetTcpRepairWindow()` modifies the `TcpRepairWindow` parameters, applicable only when the socket is in repair mode.
- `SetTcpQueueSeq()` sets the sequence number for either the send or receive queue, depending on the repair queue's current state.
- `SetTcpRepairQueue()` selects the active repair queue (either send or receive) for the socket in repair mode.
- `SetTxBuffer()` and `SetRxBuffer()` adjust the properties of the transmission and reception buffers, respectively.
- `SetTcpRepairOptions()` modifies various TCP attributes when the socket is in repair mode, taking input as a `TcpOptions` structure.

Upon successful restoration of the socket, it can be bound to the node's local address and connected to the client without initiating any packet transmissions. Finally, the socket exits repair mode, enabling it to resume standard communications.

4.3 Connection Handoff Protocol

Serialization and restoration are not enough to implement Prism. To enable the TCP connection handoff protocol, we implement a programmable switch for Prism and host-side control plane, including UDP-based RPCs, to configure the switch based on connection handoff requests.

We implement a Prism switch class called **PrismNetDevice** by extending the `BridgeNetDevice` source code. The `BridgeNetDevice` in NS3 serves as a foundational module that simulates a simple bridge or switch, making it an ideal starting point for our Prism switch implementation. It inherent design allows for the forwarding of packets between bridged devices, closely mirroring the basic functionality required by the Prism switch. By leveraging the existing mechanisms within the `BridgeNetDevice`, we could focus on layering the unique features of the Prism on top, rather than building from scratch.

The Prism switch requires the ability to modify packet addresses for specific TCP connections based on the custom UDP packets it receives. To facilitate this, we define a struct named `PrismAction` for adding and changing switch rules. When a device needs to configure a switch rule, it serializes an instance of `PrismAction` into a UDP packet and sends it to the switch. Upon receiving these UDP packets, `ReceiveFromDevice()` passes them to `ConfigReq()`. Depending on the specified action (e.g., `ADD`, `DELETE`, `CHOWN`, `LOCK`, `UNLOCK`), `ConfigReq()` invokes the appropriate function to execute the action and returns a status indicating success or an error. After the status returns, the switch invokes `ConfigPrepareResponse()` to write the status into the packet and swap the source and destination IPs and ports, then sends the response back to the original sender.

To simulate prism hash table in the original source code, we defined two structs: `Key` and `Prism`. The `Key` struct represents a combination of an IPv4 address and a port, while the `Prism` struct encapsulates information about a virtual address, its corresponding port, the owner's address, port, and MAC address, as well as a locking status. To efficiently store and retrieve Prism objects based on their associated Key, a custom hash function, `KeyHash`, is provided. This function computes a unique hash value for each Key by combining the hash of its IPv4 address and port. Finally, we defined an unordered map `m_prism` that uses `Key` as its key and `Prism` as its value, enabling efficient storage and retrieval of `Prism` objects.

In addition to the table `m_prism`, we defined a series of functions to manage the entried of `m_prism`: `AddEntry()` checks if the key already exists based on the provided `PrismAction` and if not, adds a new entry to `m_prism` map; `DeleteEntry()` removes an entry if the key provided is found; `ChangeOwner()` updates the ownder details of an existing `m_prism` map entry and modifies the owner's IP, port and MAC address according to the `PrismAction`; `LockEntry()` and `UnlockEntry()` toggle the locked state of an entry by searching for the key in `m_prism` and update the locked status accordingly.

Before forwarding a TCP packet, the switch consults the `m_prism` map to see if the packet's source or destination IP address and port match any key. If a match is found, the switch modifies the packet's details accordingly. For incoming packets with a source address and port that match a key, the switch updates the packet's destination IP, port, and MAC address to reflect the associated owner's details. On the other hand, for outgoing packets destined for an address and port present in the key, the switch adjusts the packet's source IP and port to their virtual counterparts.

Our adaptation of the `BridgeNetDevice` module in NS3 to create the Prism switch demonstrates a tailored approach to packet processing. By introducing specialized structures and enhancing packet handling logic, we've seamlessly integrated the unique functionalities of the Prism switch into the NS3 environment, ensuring efficient and dynamic packet address modifications.

Chapter 5

Evaluation

5.1 Methodology

We compare NS3 implementation of Prism against the traditional proxy, which we also implement in NS3. While the proxy represents the conventional approach that has been widely adopted, Prism emerges as a promising contender that aims to tackle the inherent challenges of the traditional proxy, which include link utilization and CPU utilization [1].

In the proxy, the frontend terminates a TCP connection with the client, and relays the data between the client TCP connection and another established with a chosen backend. Our proxy maintains a pool of persistent TCP connections with each backend so that the frontend can retrieve data from the backend without newly establishing a TCP connection. The frontend reserves an available connection in the pool by calling `GetAvailableFrontendConnection()`. If no connections are available, the request is queued for that selected backend. Once another request completes and releases a connection, our proxy processes the next request in the queue. Unless otherwise stated, we use 5 TCP connections in the pool for each backend in all the experiments.

Although we evaluate Prism mainly with migrating the connection for every request, as in the Prism paper [1], we also evaluate Prism without connection migration except for the first request. This experiment demonstrates the upper bound of Prism performance without the vast majority of the overheads.

We evaluate Prism's scalability, which was not done in the original study due to their limited hardware-testbed experiment, by varying the number of backends. We also vary the object sizes because the connection handoff overheads of Prism occur per object and thus are amortized when the object size is large. This experiment helps us validate our implementation of Prism in NS3.

5.2 Basic Tests

We use a star-topology to connect the client, frontend, and backends. All the links are 1 Gbps, except that the client connects to the switch over a 5 Gbps link, mimicking a high-bandwidth top-of-rack switch uplink in a datacenter. We vary the number of backends from 4 to 32; although the original Prism paper used six backend, we experiment with more backends, taking the advantage of simulation. We vary the object size between 4KB and 256KB. The client uses 10, 100 or 200 persistent TCP connections; inside each connection, the client sends the next request once the full response is received. We use a single connection between the frontend and each of the backends to perform connection handoff.

We expect Prism exhibits higher throughput than the traditional proxy, especially when object sizes are large, where the overheads of Prism can be amortized, and when the number of backends are large, where more backend links can be used without being constrained by the frontend link. We also expect that Prism without multiple connection handoffs exhibits higher throughput than than Prism that migrates the connection for every request due to the reduced connection handoff overheads.

Figure 5.1 plots the results. As expected, the proxy throughput remain at 1Gbps in all the configurations, constrained by the frontend link bandwidth. Prism's throughput varies, but not constrained by the frontend link bandwidth, achieving at least 3.23Gbps. When 4 backends are used, 4Gbps is theoretical maximum, because each backend can offer at most 1Gbps. Prism (one handoff) can achieve nearly 4Gbps due to minimum connection handoff overheads, whereas regular Prism configuration takes at least 196KB of the object size or 200 concurrent connections to achieve the same rate to amortise the connection handoff overheads. High connection count helps Prism achieve the higher throughput, because more connections can be active while some connections are not in the connection handoff process, minimizing the link underutilization.

Similar discussion applies to the cases with larger numbers of backends. When the number of backends is 16 or 32, the theoretical maximum throughput with Prism is 5Gbps, which is the link bandwidth of the client. When object sizes are small, frequent connection handoff overheads prevent Prism from achieving high throughput. That can be confirmed by high throughput with small objects in Prism without multiple connection handoffs.

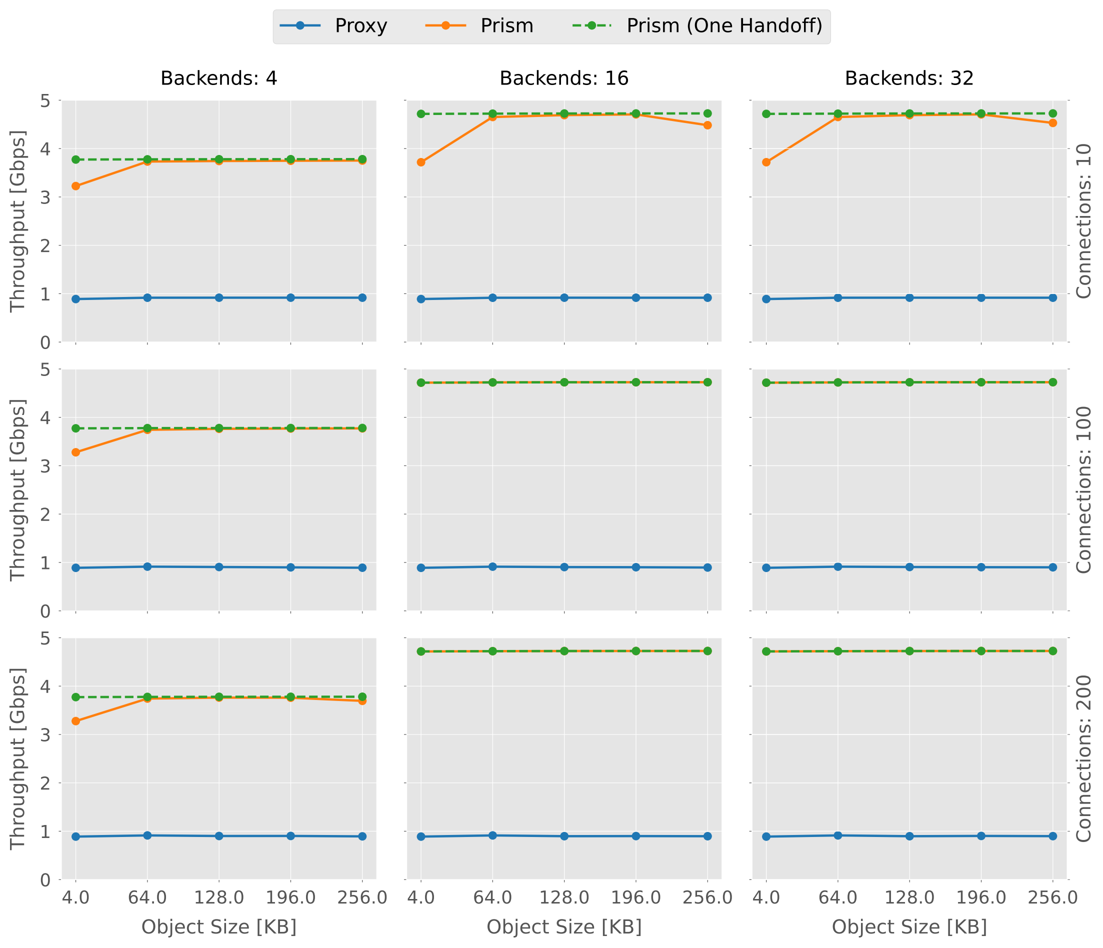

Figure 5.1: **Basic test results.**

5.3 Advanced Tests

We now evaluate whether our Prism simulator with NS3 can simulate the performance of Prism in a real hardware testbed. We therefore configure the simulation topology based on the parameters used in the Prism paper. Object sizes are 1KB–4MB, and 6 servers are used, one acting as the frontend and the others acting as the backends. Each server is connected to the switch via a 10 Gbps link, while a 40 Gbps link is used for the client. The client uses 500 persistent TCP connections, and we use 6 TCP connections between the frontend and each of the backends for connection handoff.

To see how the connection handoff overheads impact the throughput, we also vary the connection handoff latency by adding up to 300μs of latency on every connection handoff event. We expect that larger objects are needed to amortise the connection handoff costs when the handoff latency is higher.

Figure 5.2 plots the results. Although throughput of regular Proxy stays at most

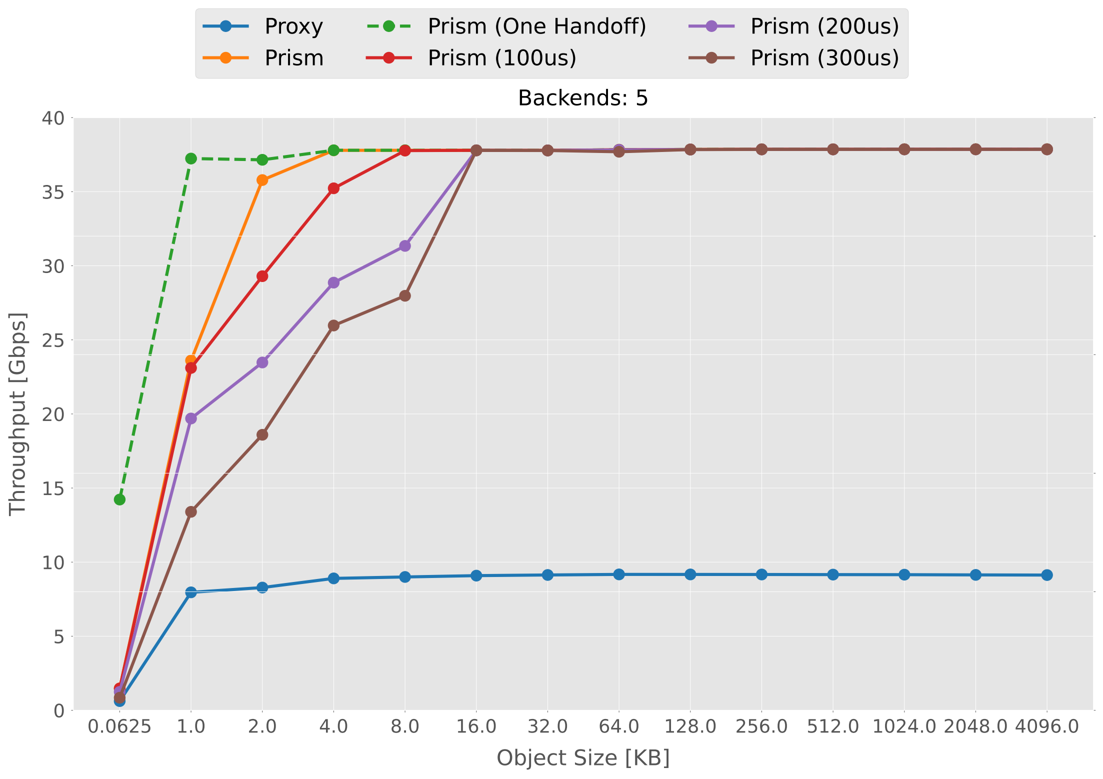

Figure 5.2: **Advanced test results.**

approximately 10 Gbps, again constrained by the frontend link bandwidth, Prism can achieve nearly the uplink bandwidth (i.e., 40Gbps), utilizing the aggregated bandwidth of the backend links when the object size is 64KB or larger. For smaller object sizes, we observe that connection handoff overheads are gradually amortized with increasing object sizes. Since our results are similar to those in the original Prism paper, whose throughput is maximized with 128KB or larger objects, we conclude that our simulator enables a reasonable approximation of Prism. It is clear that we could observe even closer throughput to the original paper when we increase the connection handoff time, which we leave as future work.

When extra latency is added to connection handoff, as expected, we observe lower throughput when object sizes are small, confirming that connection handoff overheads impact Prism's throughput.

5.4 Simulation Time

Finally, we report the wall-clock time taken to execute our simulation experiments. We employ a single instance of `CDH_32C96G` in Tencent Cloud, equipped with a 32-core

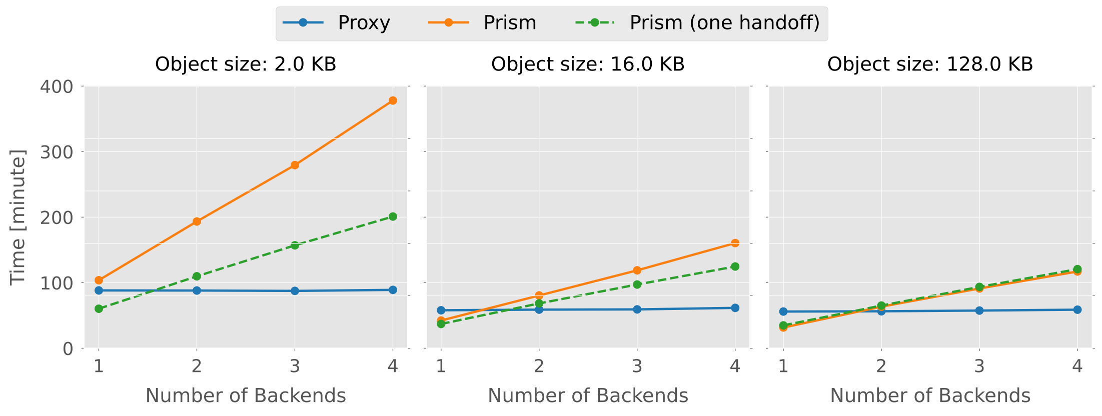

Figure 5.3: **Simulation time.**

CPU, 96GB of RAM, and 50GB of a disk. Since our experiment is simulation, we expect that, the more packets are generated (by small objects), the longer the simulation takes. We thus also expect that Prism exhibits longer simulation time than proxy due to higher throughput.

Figure 5.3 plots the results. When the number of backends is one, Prism (one handoff) exhibits the shortest simulation time. This is because only two nodes are involved during most of the simulation. For Proxy, the simulation time is almost the same regardless of the number of backends, because the throughput is the same (10Gbps) in the baseline. Prism exhibits the longest simulation time due to a handoff request packets that are sent at every request. Therefore, when more data packets are sent for each objects or the object size is larger, the relative (simulation) cost of handoff request packets become lower, resulting in the similar simulation time to Prism (one handoff) case when the object size is 128KB.

5.5 Discussion

In the basic tests, throughput for the proxy was limited to approximately 1 Gbps regardless of the object size or the number of backends, despite the 5Gbps client link bandwidth, because the frontend must relay the data between the client and backend. We observed the same in the advanced tests where the client bandwidth is 40Gbps. This indicates the validity of our proxy implementation in NS3, which maximizes the performance using connection pools that allow the frontend to retrieve data from a backend without performing three-way handshake.

For Prism, with just four backends, the peak throughput was close to 4Gbps. As the number of backends increases, that approached to 5 Gbps, which is the client link bandwidth. Those observations indicate validity of our Prism implementation in NS3. Simultaneous requests from the client lead to concurrent responses from all the backends (recall that requests are redirected to different backends in a round-robin fashion), enabling the client to utilize the combined backend bandwidth. When the number of backends is large, the client link bandwidth becomes the bottleneck. We confirmed that throughput stays at most at 5Gbps in Chapter 5.2 and 40Gbps in Chapter 5.3 regardless of the number of backends.

The presence of the connection handoff overheads in our NS3 Prism implementation was crucial, and we confirmed that by experimenting with a variant of the Prism application that migrates connections only at their first requests. This variant consistently achieved the maximum throughput even with small object sizes, because of the negligible (i.e., only once through the simulation trial) connection handoff overheads. The throughput difference from regular Prism demonstrates that the connection handoff overheads are successfully included in our NS3 simulation.

Chapter 6

Conclusions

This thesis implemented and evaluated Prism in the NS3 simulator. The evaluation of the traditional proxy, which we also implemented in NS3, and Prism has highlighted distinct performance characteristics of those systems. Since the traditional proxy relays requests and responses at the frontend server, its throughput is constrained by the link bandwidth of the frontend. Prism outperforms the proxy, leveraging combined backend bandwidths as long as the client bandwidth is available, especially with large object sizes and increased numbers of TCP connections. Although Prism's repeated TCP connection migrations introduce overheads of serializing and restoring the connection and configuring the switch, those overheads can be amortized when the object sizes are large. Our experiment in comparison to the one-time connection-handoff variant of Prism confirmed those overheads, demonstrating the validity of our implementation of Prism in the NS3 simulator. We believe our work made significant contribution to enabling further research with Prism without physical server resources.

Bibliography

[1] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. Prism: Proxies without the pain. In *NSDI*, pages 535–549, 2021.

[2] Danielle E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[3] HAProxy. The reliable, high performance tcp/http load balancer.

[4] George F. Riley and Thomas R. Henderson. The ns-3 Network Simulator. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[5] Lydia Parziale and International Business Machines Corporation, editors. *TCP/IP tutorial and technical overview*. IBM redbooks. IBM International Technical Support Organization, United States, 8th ed edition, 2006.

[6] Behrouz A. Forouzan. *TCP/IP protocol suite*. McGraw-Hill Forouzan networking series. McGraw-Hill, Boston, 2nd ed edition, 2003.

[7] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. Technical Report RFC5681, RFC Editor, September 2009.

[8] Habibullah Jamal and Kiran Sultan. Performance Analysis of TCP Congestion Control Algorithms. 2(1), 2008.

[9] J. Padhye, V. Firoiu, D.F. Towsley, and J.F. Kurose. Modeling TCP Reno performance: a simple model and its empirical validation. *IEEE/ACM Transactions on Networking*, 8(2):133–145, April 2000.

[10] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, July 2008.

[11] Jonathan Corbet. TCP connection repair [LWN.net].

[12] Benefits of Layer 7 Load Balancing | NGINX Load Balancer.

[13] Layer 7 Load Balancing | Load Balancer Algorithms for Traffic Management - Kemp.

[14] What Is Layer 4 Load Balancing? | NGINX Load Balancer.

[15] Layer 4 Load Balancing | Layer 4 Load Balancer Switch - Kemp.

[16] Yutaro Hayakawa, Lars Eggert, Michio Honda, and Douglas Santry. Prism: a proxy architecture for datacenter networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 181–188, Santa Clara California, September 2017. ACM.

[17] Thomas R Henderson, Mathieu Lacage, and George F Riley. Network Simulations with the ns-3 Simulator.

[18] nsnam. ns-3.

[19] Lelio Campanile, Marco Gribaudo, Mauro Iacono, Fiammetta Marulli, and Michele Mastroianni. Computer Network Simulation with ns-3: A Systematic Literature Review. *Electronics*, 9(2):272, February 2020.

[20] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. TCP Extensions for High Performance. RFC 7323, September 2014.