Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

SwitchML++: Extending SwitchML's Architecture for Dynamic Horizontal Scaling

Author: Gianni van der Galien (2693280)

1st supervisor: Lin Wang
2nd reader: Francesc Verdugo

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 18, 2023

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Machine learning (ML) has become a pivotal point for decision-making in complex problem domains. However, ML model training can be a time-consuming process. A way to expedite this process is to train the models in a distributed manner. Often a parameter server synchronises all models in the cluster after training to combine the efforts. Recent developments in programmable switches have allowed us to offload computations into the network. SwitchML is a solution that effectively offloads the computations for synchronising ML models into the network. However, SwitchML only supports a single top-of-rack switch, limiting itself to the nodes in the rack. Supporting more aggregators could speed up the training process as more training nodes become available.

This paper aims to investigate the possibilities and effects of adding support for multiple aggregation switches and the runtime adaptability of the aggregation tree.

For this paper, we employed systematic searches in different databanks to find relevant literature. We designed the system's architecture with the Attribute Driven Design method 3.0. Lastly, we utilised different statistical tests to interpret our gathered experiment data.

Our results revealed that the overhead created to support horizontal scalability creates a trade-off between scalability and speed. Furthermore, our results showed that increasing switch levels in the aggregation tree further slows the overall execution, highlighting the importance of keeping the workers close together to minimise the switch levels needed.

Our proposed design and evaluation provide a stepping stone towards achieving runtime scalability for in-network aggregation by utilising programmable switches.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

In the age of data-driven innovations, machine learning (ML) has become a pivotal approach for decision-making, particularly in complex problem domains where mere algorithms fall short. However, ML model training can be a time-consuming process. To expedite this procedure, designers employ various techniques, such as vertical scaling, utilising hardware acceleration through GPUs or Application Specific Intergraded Circuits (ASICs). Another approach is horizontal scaling, which distributes the workload across multiple machines to achieve faster training times [1].

Many existing distributed machine learning (ML) systems adopt a data-parallel computation approach, employing a distributed mini-batch Stochastic Gradient Descent (SGD) algorithm. In this schema, the system trains models on various data segments iteratively and synchronises the model parameters at the end of each iteration. Often a parameter server (PS) is employed for this synchronisation process. In this setup, each node sends its ML model gradients to the PS, aggregating all the values received from the nodes. Subsequently, the PS sends the aggregated values back to the nodes, allowing them to update their respective models with the synchronised parameters [2]. Although the PS server approach offers a solution for training ML models in a distributed manner, it has limitations. The transmission of model gradients from nodes to the PS can potentially congest the network, as all these data packets converge in a single location. Furthermore, a notable limitation arises from the fact that a node must act as the PS, causing it to be unavailable for actively participating in the model training process.

Advancements in network technologies, particularly in the realm of programmable switches, have ushered in a new era of possibilities, notably through the concept of in-network aggregation. This innovation allows computational tasks to be effectively delegated to the

network, marking a departure from the constraints posed by a PS. With in-network aggregation, the requirement for a dedicated PS is avoided, simultaneously introducing the capability to process partial model gradients gradually. This approach naturally shrinks the volume of packets as they traverse further into the network via an aggregation tree, leading to a streamlined and more efficient data transmission process [3].

A solution taking advantage of this progress is SwitchML, which enables distributed training of ML models. Leveraging programmable switches, SwitchML achieves efficient in-network aggregation of gradient values from numerous models. However, SwitchML's current limitation lies in its support for only a single top-of-rack (ToR) switch as an aggregator, limiting itself to the number of nodes in the rack [4]. Prominent language models, such as OpenAI's ChatGPT, have witnessed a notable rise in adoption [5]; however, their effectiveness hinges on the volume of training data, varying according to specific application scenarios. To illustrate, the training dataset for GPT-2 consisted of 8 million documents and was approximately 40 GBs in size [6]. In this context, supporting multiple levels of aggregator could speed up the training process by a significant amount as we can use more nodes for training.

For this reason, the paper aims to investigate the possibilities and effects of adding support for multiple aggregation switches and the runtime adaptability of the aggregation tree. With this aim, we formulated the following two research questions:

RQ1: *How can the architecture of SwitchML be extended to support both multiple aggregation switches and runtime adaptability of the aggregation tree?*

RQ2: *How does support for multiple aggregation switches and runtime adaptability of the aggregation tree affect the performance?*

We collected the literature for this paper through systematic searches on Google Scholar, the ACM digital library, and O'Reilly Media. We specifically targeted papers related to in-network computing and aggregation to ensure relevance to our study. These papers were evaluated based on their significance, publishers, authors, and the number of citations received. We designed the system employing the Attribute Driven Design 3.0 (ADD) method described by Cervantes and Kazman [7]. This approach goes beyond traditional requirements-based design and considers various aspects influencing software architecture. ADD emphasises providing a clear rationale for each design decision and justifying the rejection of alternative options, making it a well-suited approach for this paper's design process.

In this paper, we present SwitchML++, a modified version of SwitchML that we designed to support multiple aggregation switches and runtime adaptability of the aggregation tree. The evaluation of SwitchML++ shows that the design serves its intended purpose with a trade-off between scalability and speed.

We summarise our main contributions as follows:

1. We propose a design for supporting multiple aggregation switches and runtime adaptability of the aggregation tree.
2. We implemented our design into a proof of concept (PoC).
3. We present the experimental results of our PoC.

This paper follows the following outline: Chapter 2 presents relevant background information. Chapter 3 contains details on the completed design. Chapter 4 covers the implementation. Chapter 5 assesses the developed design and implementation. Chapter 6 presents the discussion. Chapter 7 describes works related to this paper. Finally, Chapter 8 presents the conclusion.

2

Background

This chapter serves as a foundation for understanding the upcoming chapters, offering essential background information. The chapter starts with section 2.1, exploring software-defined networking. Section 2.2 delves into programmable data planes. Section 2.3 discusses in-network aggregation, and section 2.4 describes SwitchML.

2.1 Software-Defined Networking

In traditional IP networks, network operators must configure each device separately using low-level and often vendor-specific commands. On top of this, network environments have to endure faults and adapt to changes in load. Automatic reconfiguration and response mechanisms are practically nonexistent. Enforcing the required policies in such an environment is complex and challenging. Besides this, traditional IP networks are also vertically integrated. The control plane, which manages the forwarding rules and the data plane, which forwards the traffic accordingly, are tightly coupled in networking devices, reducing flexibility [8].

Software-defined networking (SDN) is a shift in the networking paradigm in which we move away from traditional IP networks and their problems by segregating the control plane from the data plane. With this segregation, network switches become simple forwarding devices while we implement the control logic on an SDN controller. We can realise the segregation between the two planes through a well-defined API between the switches and the SDN controller. The SDN controller exercises direct control over the state in the data plane elements (mainly switches) via this API, also known as the southbound API [8]. One of the most well-known southbound APIs is OpenFlow [9].

The Open Networking Foundation (ONF) [10] proposed an open architecture to address the problems in traditional networks with the potential to enable the automation of network configurations and fully program the network. The SDN architecture comprises three layers: Infrastructure, Control, and Application [11]. Figure 2.1 shows a simplified overview of the SDN architecture.

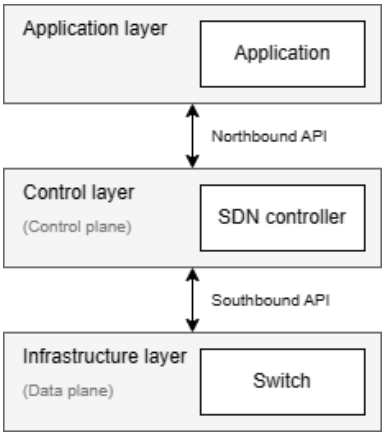

Figure 2.1: SDN architecture overview

The infrastructure layer consists of forwarding elements like switches responsible for forwarding packets. To achieve the separation between the control and data plane forwarding element needs to be remotely accessible. A southbound API does this. The control layer, also known as the network operating system (NOS), provides the application layer with an abstract network view, giving just enough information to specify policies while hiding all the implementation details. The control layer comprises a centralised controller responsible for communicating between the application and infrastructure layers. It translates requirements from SDN applications to the infrastructure layer and returns relevant information to the applications. The application layer contains SDN applications, which implement the network control logic and strategies. It interacts with the control layer via a northbound API. The SDN application communicates network requirements to the controller, translating them into southbound specific commands and forwarding rules. Examples of such applications are firewalls and load-balancing [11].

2.2 Programmable Data Plane

Abstraction is a fundamental and essential concept that empowers computer systems to handle change effectively and simplifies complex implementations. SDN abstracts network

functions by segregating the control and data plane. The control plane is decoupled and moved to external software, granting it programmability. This programmable control enables network operators to introduce new functionalities to their networks while preserving the behaviour of existing protocols. The interface between the control and data plane uses a Match-Action approach, wherein a data plane element matches a subset of packet bytes against a table. The matched entry specifies the action(s) the element will apply to the packet. Traditional data plane elements support a limited amount of action primitives to specify the processing of packet headers in hardware. With these elements, supporting new features requires replacing the hardware. Reconfigurable Match Tables (RMT) is an architecture that solves this by allowing for enough reconfiguration in the field to support new types of packet processing at runtime. With this architecture's programmability comes a natural trade-off between programmability and speed [12]. Figure 2.2 shows an overview of the RMT architecture.

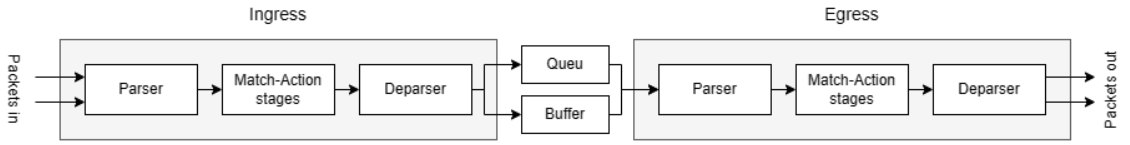

Figure 2.2: RMT architecture overview

RMT architecture is implemented mainly on custom ASICs, each with a low-level interface. Programming these chips can be a difficult task. Programming Protocol-Independent Packet Processors (P4) is a higher-level domain-specific language for programming these chips. With P4, a developer can tell a switch how to process a packet without knowing all the low-level details. P4 has three main goals. The first one is reconfigurability. The control plane should be able to redefine the packet parsing and processing while deployed. The second goal is to be protocol independence. The data plane must not be limited to specific packet formats. Instead, the control plane should be able to define a packet parser for extracting header fields and a set of match-action tables. The last goal is to be target independent. The programmer should not have to know the underlying data plane element details. Instead, a compiler should handle this while compiling P4 into a target-dependent program [13].

2.3 In-Network Aggregation

In traditional distributed machine learning, model training involves the iterative update of model parameters using gradient information computed locally on each compute node. These gradients are then communicated among nodes to update the model collaboratively. However, the communication overhead incurred during gradient aggregation can become a bottleneck, especially as the number of nodes and the complexity of the model increase [14].

In-network aggregation is a networking approach in which network devices, such as switches, actively contribute to accelerate aggregation tasks. When specifically looking at ML, instead of aggregating the model gradients at a destination server like a PS, the network devices will aggregate the values as they advance along the network path. Typically in-network aggregation uses a so-called aggregation tree built from the underlying network topology. The tree consists of aggregators and workers. Workers send their gradients into the aggregation tree, where the aggregators will aggregate a subset of the gradients and send them further into the tree. The tree's root will aggregate the subsets of its children into a complete set of aggregated gradients and send them back to the workers [14]. Figure 2.3 shows an overview of such an aggregation tree where the aggregators are switches.

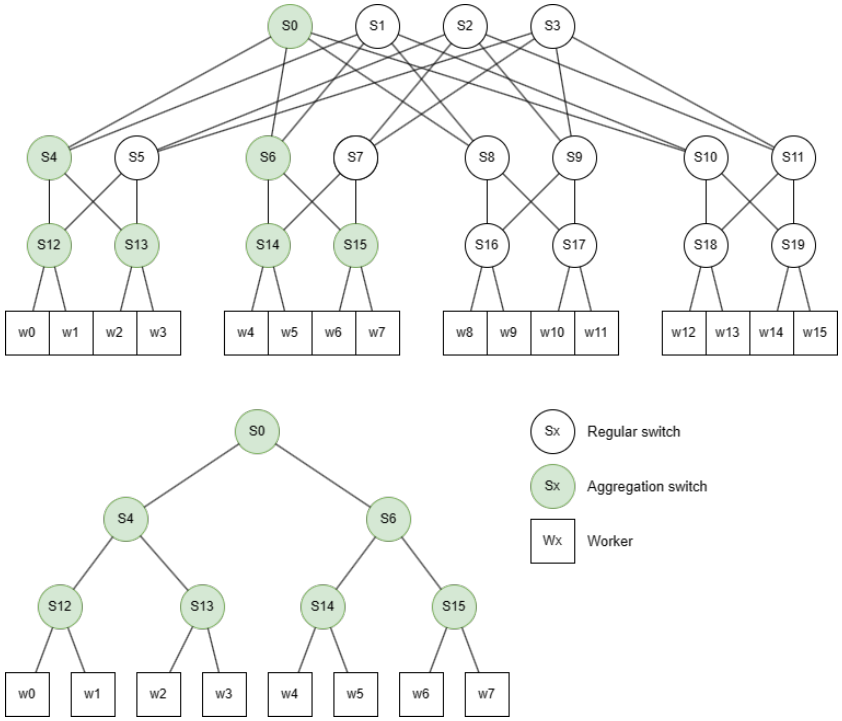

Figure 2.3: Example aggregation tree out of a Fat tree with k=4 ports

2.4 SwitchML

SwitchML [4] represents a solution for advancing distributed ML training by integrating in-network aggregation and programmable switches. SwitchML effectively accelerates ML training while mitigating the limitations of programmable switches. One of the main points of focus in SwitchML revolves around effectively managing the limitation of only being able to handle simple integer arithmetic operations. ML model gradients often take the form of sizeable floating point vectors. SwitchML bypasses this limitation without compromising the training accuracy by quantising the floating points to 32-bit integers by adopting a block floating-point-like approach on the end host.

Programmable switches have limited storage and computational power, therefor the in-network aggregation of SwitchML works by dividing the gradient vector into sizable chunks. Workers each send a chunk to the switch. Upon receiving a chunk, the switch aggregates the values and saves the partial result into memory. When the switch receives chunk N from all workers, it sends the aggregated chunk back to the workers. This process repeats until the whole vector is complete. SwitchML developed a lightweight recovery scheme using shadow copies and time-outs to handle packet loss. The shadow copy is an alternating storage space on the switch to keep the results of the previous aggregated chunk. The time-outs are triggers implemented on the worker to resent a chunk in case a chunk is lost. When the switch has received chunk N from all workers, it knows that the chunk was lost and resents the values. Figure 2.4 shows a simplified overview of SwitchML's in-network aggregation process.

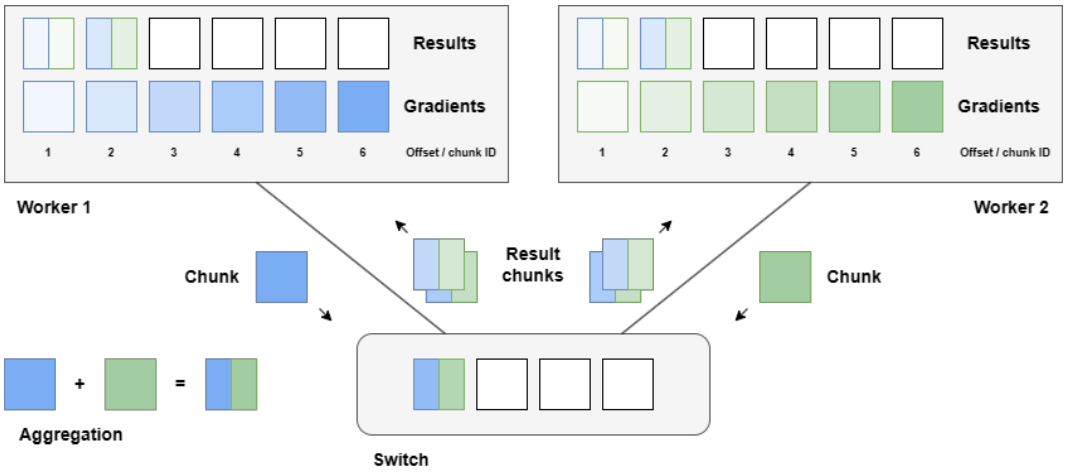

Figure 2.4: SwitchMLs in-network aggregation process overview

Despite its promising potential, the current implementation of SwitchML limits itself to a single programmable ToR switch. This constraint challenges large-scale ML clusters, which often comprise numerous nodes. As the demand for more extensive and complex ML models grows, a need arises to extend SwitchML's capabilities to support multiple levels of aggregators to accommodate larger clusters.

3

Design

This chapter presents the design of our scalable in-network aggregation solution. Section 3.1 delves into the essential architectural elements that drive the design. Moving forward, Section 3.2 provides a comprehensive global overview of the system and explains the inter-relationships between its various components. To better understand the design, we zoom in on specific system components in Sections 3.3, 3.4, and 3.5, offering detailed descriptions and rationales behind the design decisions made for each component. Finally, in Section 3.6, we address reliability.

3.1 Architectural Drivers

We acknowledge that there are more drivers than covered in this section; however, this paper limits itself to the ones mentioned in this section.

3.1.1 Design Purpose

The current version of SwitchML is limited to a single programmable ToR switch, which limits the number of workers that can simultaneously train a specific machine learning model to the number of workers in the rack. The design seeks to introduce horizontal scaling of workers by supporting multiple levels of switches in the network to facilitate more workers than a single switch can handle. The primary objectives of this design are (I) how to support multiple switches for aggregation in SwitchML and (II) how to adapt the aggregation tree at runtime, empowering data centres to efficiently handle a variable number of workers and effectively address the research questions.

3.1.2 Use Cases

This section presents the most relevant use cases for the design in Table 3.1

Use case	Description
UC-1 : Scale-in	A data centre scheduler or user requests an extra worker to join a training group.
UC-2 : Scale-out	After completing all training iterations, a worker will leave the active aggregation network.
UC-3 : SML	The workers and Switches in a training group perform SwitchML.

Table 3.1: Use Cases

3.1.3 Quality Attribute Scenarios

This section presents the most relevant quality attribute scenarios with their associated use case(s) in Table 3.2.

ID	Quality Attribute	Scenario	Associated Use Case
QA-1	Reliability	When a SwitchML packet is lost, the system should recover 100% of the packets without affecting the efficiency (not including the effect of the retransmission mechanism)	UC-3
QA-2	Scalability	When workers join/leave an active aggregation network, resources should scale accordingly only to use the necessary resources.	UC-1, UC-2
QA-3	Usability	The solution should not require any changes in the data centre's network to be able to use it.	UC-3
QA-4	Modifiability	Changes in the network should require, at max, a low effort to adjust the implementation accordingly.	UC-3

Table 3.2: Quality Attribute Scenarios

3.2 Overview

Like its predecessor SwitchML, our design adopts a switch-host architecture to preserve the core functionalities while striving to achieve the design objective. The focal point of our design is the dynamic adjustment of the number of workers at runtime, necessitating real-time adjustments to the data plane. Our design incorporates an SDN controller to facilitate these on-the-fly changes. The SDN controller plays a crucial role in computing the required modifications, updating the data plane accordingly, and concurrently maintaining the state of the active aggregation network. Figure 3.1 shows an overview of all components in our design within a simple network.

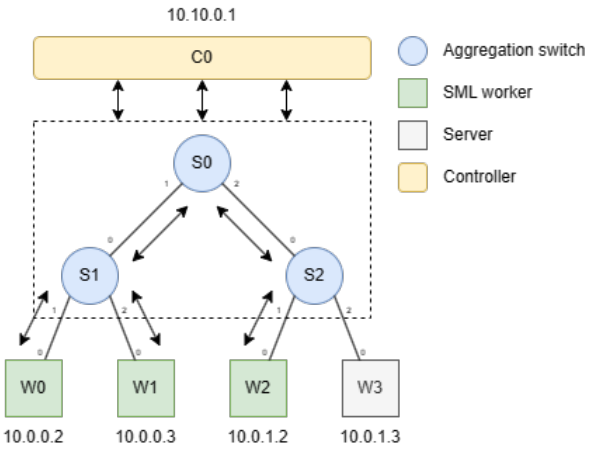

Figure 3.1: Design overview

In our design, each active aggregation network is uniquely identified by an aggregation group ID, which plays an essential role in the worker subscription and synchronisation process. When a worker starts, an allocator or user sets the aggregation group ID for the worker. Subsequently, the worker starts the subscription process, sending the controller a subscription packet with the given ID as its initial step.

During aggregation, the workers send their ML model's gradients to the switch in chunks. The flexible design allows workers to join or leave the active network anytime. To ensure seamless integration, newly joined workers initiate a synchronisation sequence after completing the subscription process. During synchronisation, the worker requests the current chunk from the root of the aggregation tree, effectively aligning itself with the ongoing aggregation process. Upon finishing, the worker again starts the subscription process to let the controller know it completed their task.

A crucial aspect of our design is using on-path aggregation to handle multiple levels of switches effectively. As aggregation packets pass through each switch, they undergo aggregation, with each switch aggregating a subset of the values. This strategic approach minimises network traffic, optimising the overall efficiency of the aggregation process.

3.3 Controller

In our design, the controller plays a central role in handling incoming requests from workers. Upon receiving a request, the controller initiates a series of steps based on the received (un)subscribe primitive. The controller follows a well-defined process with respective actions to the received primitive:

1. **Calculating the optimal switch:** The first step the controller takes is determining the new root of the aggregation tree with or without the worker in the tree (depending on the request) and selecting this switch.

2. **Creating the path:** The second step involves creating a path from the worker to the selected switch. This path ensures we can traverse all the necessary switches in the next step.

3. **Updating Forward Tables and Aggregation Tree:** The third step entails traversing the created path and updating the data plane of all switches accordingly. Additionally, the controller updates its in-memory reference model of the aggregation tree, effectively incorporating or removing the worker from the aggregation tree.

The controller is an essential part that does most of the heavy lifting required for enabling runtime scaling. The controller is responsible for managing the active aggregation nodes in the network necessitates carefully selecting the appropriate controller type. Generally speaking, there are three types of controllers: centralised, distributed and hybrid.

We opted for a centralised controller for our design due to its inherent advantages. The primary benefit of a centralised controller lies in its ability to maintain a global view of the network, enhancing the design's resilience to network changes and favouring QA-4. The global view allows for effective management and coordination of workers, optimising the system's adaptability and performance. While a distributed controller eliminates the single point of failure in a centralised setup, its partial view of the network makes it less suitable for supporting QA-4. Additionally, deploying multiple controllers in a distributed setup

may enhance scaling capabilities but could conflict with QA-2, leading to resource inefficiencies. Hybrid controllers are mainly for transitioning legacy networks from distributed to centralised controllers. However, this aspect lies outside the scope of our design, making the benefits of a hybrid approach irrelevant to our specific objectives [15].

Effectively managing the active aggregation nodes in the network requires a robust data structure capable of accommodating multiple connections to and from nodes. Two potential candidates for this purpose are a graph and a tree. A graph offers excellent flexibility in handling multiple connections between nodes; however, it lacks a clear distinction between parent and child nodes. On the other hand, a tree provides a clear hierarchical relationship between parent and child nodes while offering enough flexibility to handle multiple connections. After careful consideration, we have chosen to utilise a tree data structure. The clarity a tree's parent-child relationship provides outweighs the additional flexibility that a graph could offer. At the same time, a node in a tree can only have one parent. The tree structure aligns seamlessly with the hierarchical nature of the aggregation process, simplifying node management and ensuring efficient coordination within the aggregation network [16].

To determine the root switch of our aggregation group, the controller employs the Lowest Common Ancestor (LCA) approach, which allows us to identify the path with the smallest number of hops between the root and a worker [17]. An alternative approach involves selecting the switch that handles the least amount of network traffic. This approach aims to prevent congestion on busy switches caused by the additional traffic generated by aggregating in-network, a clear benefit for QA-2 concerns. However, we discard this alternative approach for a crucial reason. While it may alleviate congestion in some cases, it can lead to packets travelling more hops than needed, potentially clashing with QA-2 requirements. This increased hop count could lead to suboptimal performance and latency issues, undermining the overall effectiveness of the aggregation process.

Topologies often have redundant links, creating possible loops in the network. Not modifying the network affects the algorithms' correctness. We adopt a Steiner tree derived from the network topology to enhance the design's resilience against changes and remove the potential loops (QA-3). By constructing a Steiner tree, the algorithms can leverage this tree to perform their computations, providing adaptability to network topology changes (QA-4). Steiner trees allow us to specify the nodes we want to include, offering more flexibility and robustness [18].

Furthermore, the Steiner tree's capability to select nodes facilitates quicker identification of the tree's root, streamlining the aggregation process. Conversely, an alternative approach

using a Minimal Spanning Tree (MST) would yield similar benefits of removing potential loops and adding an abstraction layer. However, we decided against the MST because it does not provide the needed information for transforming the active aggregation nodes into a tree structure, as we require knowledge of each node's parent. Figure 3.2 illustrates an example of a Steiner tree based on a Fat tree, encompassing all workers within the design [18].

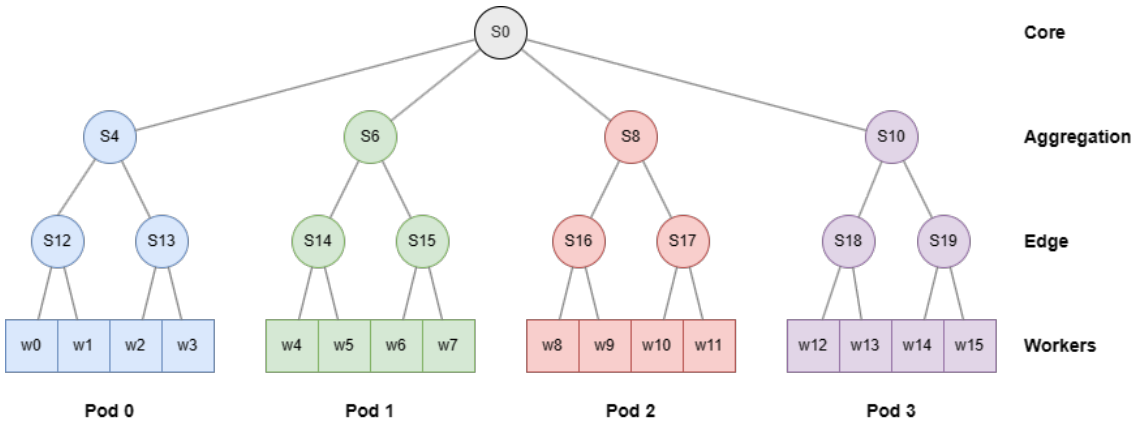

Figure 3.2: Overview of a Steiner tree based on a Fat tree with $k = 4$ where all workers are included

3.4 Aggregator

Like SwitchML, aggregators in our solution aggregate incoming data chunks workers send. However, a significant distinction lies in our approach, which leverages on-path aggregation to support multiple levels of aggregators in the network, reducing the traffic as the aggregation packets go further into the network. To facilitate on-path aggregation, the aggregator first checks if it has a parent and sends the partial results to that parent.

The root of the aggregation tree plays a crucial role as it aggregates data from its children and maintains the current chunk offset. This offset information is essential for synchronisation, allowing workers to effectively align themselves with the ongoing aggregation process. To synchronise and obtain the current chunk offset, the worker sends a request to one of the leaf aggregators. The leaf aggregator handles the request and forwards it to its parent aggregator, continuing this process until it reaches the root aggregator. Once the root aggregator receives the request, it sends back the current offset to the worker, completing the synchronisation step.

3.5 Worker

When a worker finishes its first training iteration, it communicates its intent to join a user-specified aggregation group by notifying the controller. After this initial step, the worker proceeds with its synchronisation process, which requests the current offset of the aggregation group from the switch. Once the worker receives this offset information, it can seamlessly synchronise its operations with the ongoing aggregation process. After synchronisation, the worker starts streaming the computed gradients to the corresponding switch. Once finished, the worker starts with the next iteration. Upon completing all training iterations, the worker finalises participation in the aggregation group by notifying the controller of its departure. This notification indicates that the worker is no longer actively engaged in the aggregation process for that specific group.

In our design, we encounter the challenge of synchronising workers who can join the aggregation process at any moment. We have considered several strategies to address this issue, each with advantages and disadvantages.

The first strategy involves letting the worker wait and join the aggregation process when the group aggregates for a new iteration. While this approach is relatively straightforward to implement, it has a significant downside. The downside is that the new worker has to wait, leading to resource wastage in the data centre (QA-2).

The second strategy involves requesting the offset and all aggregated values from one of the workers in the group. This approach allows a new worker to join the aggregation group mid-process. However, this strategy is more complex due to packet loss handling, and it requires all other workers to wait until the new worker receives all previous chunks, leading to resource wastage (QA-2).

In the third and chosen strategy, the worker requests the offset from the switch and begins sending from chunk n (the offset). The worker fills in the missing values with its local gradients. This strategy's main benefit is minimising resource wastage (QA-2). However, there is a trade-off, as this approach may lead to slightly less accurate training due to how we handle the missing values.

By selecting this synchronisation strategy, we balance resource efficiency and training accuracy, aiming to optimise the overall performance of our design. The chosen approach allows workers to join and contribute to the aggregation process instantly, ensuring smooth scalability and adaptability in data centre environments.

3.6 Reliability

Ensuring packet loss recovery is paramount for the practical implications of this design in real-world scenarios. Even though SwitchML provides a solution for packet loss recovery, we need more than simply extending this solution for our design's multi-layered aggregation tree. In the SwitchML, a worker sends a chunk to the switch, and if the switch does not respond within a specified time, the worker initiates a time-out and resends the chunk. This process continues until the worker receives the aggregated chunk [4]. However, when packet loss occurs in a multi-layered aggregation tree, all workers will eventually time out and resend their packets to the root of the aggregation tree. This redundancy can lead to network congestion and hinder overall performance.

The packet loss handling will be split into two parts to address potential congestion. The first part pertains to packet loss occurring between workers and switches. We can leverage SwitchML's end-to-end approach with a minor adjustment for this part. Specifically, in scenarios involving multiple levels of switches, the switches will drop seen packets originating from a worker until it has received the final result from the above switch. When the final results traverse a switch, the switch will cache these results. This approach will ensure that the retransmission because of time-outs will stay within the first switch and not go through the whole network.

The second part involves addressing packet loss handling between switches, which, due to testability constraints, lies outside the scope of this paper. Nonetheless, we have conceived a potential solution that requires further research and exploration. A strategy similar to the previous approach can be employed to ensure reliability between switches. Switches can leverage hardware interrupts as timers to trigger packet retransmissions. Another more hardware-specific way to achieve periodic tasks would be to use event-driven packet processing as it requires the ASICs to support events [19].

4

Implementation

This chapter provides comprehensive details about the implementation of various components within Mininet [20]. Our implementation includes the controller and worker, coded in Python3 [21] utilising the Mininet library. We implemented the switch with P4 version 16 [13], and the packets' implementation combines Python and P4. We offer a brief implementation description for each component, accompanied by the design decisions made during the implementation process. The chapter is structured as follows: Section 4.1 delves into the implementation details of the controller. Following is section 4.2, which provides the details of the switch. Then section 4.3 presents the worker, and lastly, we dedicate section 4.4 to the implementation details of the different packets.

4.1 Controller

As previously mentioned, we implemented the switches using the P4 programming language. However, Mininet does not inherently support P4 programmable switches. To integrate our P4 program into Mininet, we utilise the P4App [22] extension, which offers the P4Runtime [23] library. This library enables us to update the forwarding tables of switches, a crucial function not natively supported by Mininet. Due to the absence of direct P4 support in Mininet, the P4Runtime library becomes essential for updating the forwarding table. As a result, we can not use the conventional off-the-shelf controllers provided by Mininet. We opted to extend a Mininet host to implement our controller to overcome this limitation. By extending a Mininet host, we can update the forwarding tables of switches while simultaneously receiving packets from workers, making it a practical and effective approach for our implementation.

The controller listens on a dedicated TCP socket to receive incoming worker requests. When a subscription packet is received, the controller initiates a series of steps to manage the active aggregation tree in memory. If no tree exists, and the worker wishes to subscribe, the controller creates a new tree and adds the worker and the switch above him to this tree. When a tree is present, the controller searches for the LCA between the tree's root and the new worker. The controller creates two paths from the LCA, one from the aggregation tree's root to the LCA and another from the worker to the LCA.

The purpose of the first path is to determine whether the aggregation tee's root needs to change. The second path serves to connect the worker to the aggregation tree. In our implementation, the tree data structure handles the forwarding updates. Whenever nodes are added or removed from the tree, this triggers a change process. In this process, the tree checks which action (create, update, or delete) to take and promptly performs the required actions. Figure 4.1 presents a class diagram showing our controller and tree implementation.

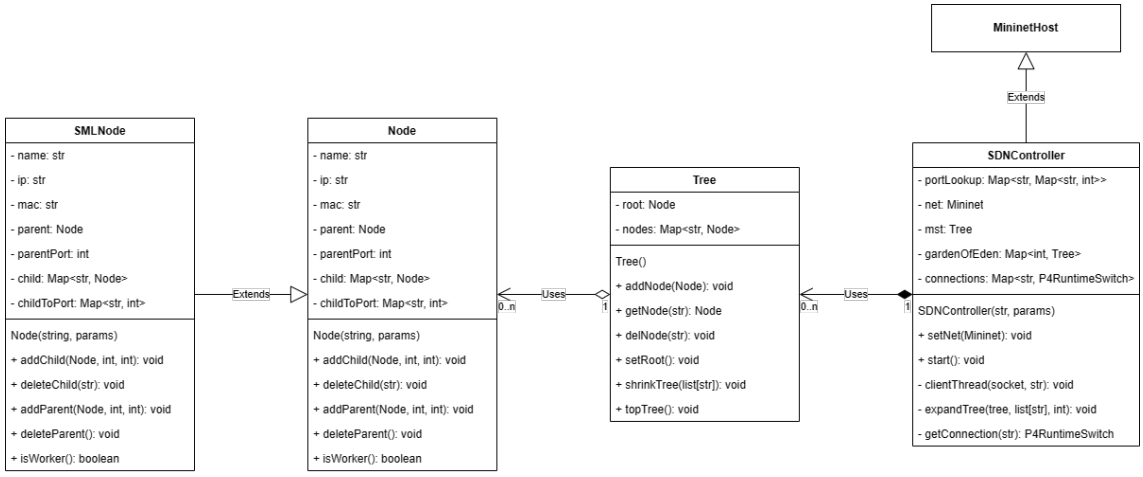

Figure 4.1: Class diagram of the controller

4.2 Aggregator

We based our aggregator implementation on our pre-existing code of SwitchML, implemented on a switch and introduced modifications to enable horizontal scaling. Here are the essential modifications that we made to scale the number of workers:

1. **Dynamic worker adjustment:** To facilitate dynamically changing the number of workers at runtime, we replaced the constant variable representing the number of workers with a table entry in the switch.

2. **Switch ranks:** We introduced the concept of ranks for switches through a table entry. Each switch sets its rank in the aggregation packet whenever it handles this packet type. This addition enables switches to identify previously encountered partial aggregated values in the same manner as it identifies previously encountered packets from workers.

3. **Instruction table:** Another significant change involved adding a table entry to instruct the switch on the next step with each aggregation packet. This table entry specifies whether the switch should forward the packet to the next switch and, if so, which port to use for the transfer.

By making these essential modifications accompanied with P4 code to handle these changes, our implementation can support horizontal scaling, allowing the system to adjust the number of workers at runtime dynamically and support multiple levels of aggregators.

We implemented the synchronisation of workers on the switch by caching the highest seen offset in the register of the aggregation tree's root switch. By utilising the previously mentioned instruction table, a switch knows if it is the root when it does not have the instruction to send aggregation packets further up the tree. In addition, switches also use this table to direct an incoming synchronisation packet to the root. Once the synchronisation packet arrives at the root, the root sets the offset, changes the type flag, and sends it back. By changing the type, other switches know it is a response and will handle it as a regular IPv4 packet and forward it back to the worker.

When the root of the aggregation tree changes, it is possible that a worker requests the offset, but the offset has yet to reach the root. To account for this scenario, we implemented a series of checks to determine the appropriate response. First, the root checks if its saved offset is non-zero. If it is, we know that the root already possesses the current offset and can send it back to the requesting worker; however, when the offset is zero, the root proceeds with further checks. It evaluates whether the number of workers in the network equals one. If this condition holds, it signifies that no other workers are in the network; therefore, the offset must be zero. If the number of workers exceeds one, there are two potential scenarios to consider. The first scenario is when the root has yet to encounter the current offset, and the second scenario is when all workers are new, resulting in a zero offset. We implemented a counter that counts the number of seen workers to ensure the correct offset is delivered. Upon receiving a request in this scenario, we mark the worker as seen and send back the request with an error flag. When the counter equals the number of workers,

we know all workers are requesting, making the correct offset zero. The switch can now respond to the subsequent request with a zero offset.

4.3 Worker

We extended the pre-existing SwitchML worker code to incorporate the desired functionalities for our worker implementation. The first step involved enabling the worker to join/leave an aggregation group. To achieve this, the worker establishes a TCP connection with the controller and sends a subscription packet with the appropriate type flag to indicate its desire to join/leave the group. Once a worker has joined a group, it immediately initiates the synchronisation process with the group. We implemented this by sending a packet over UDP into the network. We implemented a timer to account for potential packet loss, which resends the request up on timing out. The timer stops once the worker receives a response with no error code. If the response contains an error code, the worker deduces that the response does not contain the correct offset and drops the packet, waiting for the timer to expire before reissuing the request.

4.4 Packets

Our design incorporates three distinct types of packets, the first being the subscription packet. This packet incorporates three essential fields in our implementation: rank, MGID, and type. The rank field serves as an identifier for the controller, indicating which worker has sent the request. The multicast group ID (MGID) field lets the controller know for which aggregation group the request is. The type field specifies the type of request (subscribe or unsubscribe). Figure 4.2 shows an overview of the subscription packet.

Offset TCP payload	0								1								2								3							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	Rank																															
4	MGID																Type															

Figure 4.2: Subscription packet overview

The second type of packet used in our design is the synchronisation packet, which comprises four fields: MGID, type, offset, and rank. The MGID field serves as an identifier, indicating the specific aggregation group from which the worker seeks to obtain the offset. The type field is essential for switches to determine the nature of the packet, distinguishing between a request, response, or error. Based on this classification, switches forward the

packet up the tree or back to the worker as necessary. The offset field is for the switch to set the offset and for the worker to get the offset value. Lastly, the rank field plays a vital role in switch operations, enabling them to identify whether they have already received a request from a particular worker. Figure 4.3 shows an overview of the synchronisation packet.

Offset UDP payload	0								1								2								3							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	MGID																Type															
4	Offset																															
8	Rank																															

Figure 4.3: Synchronisation (Sync) packet overview

The third and last type of packet in our design is the aggregation packet. In contrast to SwitchML, our implementation of the aggregation packet has two additional fields: the type and MGID field. The type field specifies to the switch whether he should forward the packet up the aggregation tree to the next switch or multicast it back to the participating workers. The MGID field serves as an identifier, specifying the multicast group to which the switch must transmit the results when multicasting the packet back to the participating workers. Figure 4.4 shows an overview of the aggregation packet.

Offset UDP payload	0								1								2								3							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	WID																Ver								Type							
4	IDX																															
8	Offset																															
12	MGID																Size															
16	Payload																															

Figure 4.4: Aggregation packet overview

In our implementation, we standardised the bit length for all packet fields except for the MGID field. The aim was to maintain uniformity and ensure that the packet length remained multiple of 32 bits. For the MGID field specifically, we allocated a fixed bit length of 16 bits. This decision was because of the requirement imposed by the P4 programming language. In P4, multicast IDs must be represented as 16-bit variables, necessitating this specific bit length for the MGID field.

5

Evaluation

This chapter presents the evaluation of our design. We start with section 5.1, in which we will evaluate the design based on formulated quality attributes of section 3.1.3. Section 5.2 gives a detailed description of the setup used for the experiments. Finally, section 5.3 presents the experiment on execution time.

5.1 Design Evaluation

To determine the effectiveness of our design in achieving its intended purpose, we will assess the design based on how well it supports the formulated quality attribute scenarios. Table 5.1 presents a quick overview of this assessment.

QA ID	**Addressed**
QA-1	Partially
QA-2	Completely
QA-3	Completely
QA-4	Completely

Table 5.1: Overview of the quality attribute senario state

QA-1 Reliability: Currently, our design lacks a specific mechanism to handle packet loss between switches, as explained in Section 3.6. However, we have made noteworthy progress by developing a potential solution for this challenge. While the solution still needs to be fully integrated into the design, its development represents a step towards improving

the reliability aspect of our system. Further testing and refinement are required to fully address the packet loss issue and ensure robust reliability in our design.

QA-2 Scalability: Our design effectively addresses the scalability quality attribute scenario. The critical feature enabling scalability is integrating a tree data structure designed to track active SwitchML nodes efficiently. Our implementation can dynamically manage resource allocation and purging by leveraging this tree data structure. This delegation of responsibility to the data structure significantly streamlines the process, facilitating smooth and flexible scalability as the system evolves. This tree data structure's careful consideration and implementation demonstrate our commitment to providing a scalable solution that can adapt to varying workloads and efficiently manage resources for optimal performance.

QA-3 Usability: The design addresses this quality attribute scenario in its entirety. Considering the data centre's existing setup, assuming programmable switches and an SDN controller are already in place, we have ensured seamless integration of our solution without additional hardware.

QA-4 Modifiability: Our design extensively addresses the modifiability quality attribute scenario by incorporating an abstraction layer. By creating a Steiner tree as a foundational structure, we have effectively added a versatile abstraction layer on top of the network topology. This strategic approach ensures that our design remains independent of any specific topology, providing adaptability and ease of maintenance. In the event of network modifications, developers only need to verify if the algorithm generates a Steiner tree that accurately reflects the changes. If necessary, developers can make minor adjustments to align the tree with the updated network configuration.

5.2 Experimental Setup

All experiments use the same proving ground, ensuring standardised conditions for the evaluation process. The proving ground consists of a virtual machine hosted on VirtualBox version 6.1 [24], running the Ubuntu 20.04.5 LTS [25] operating system. We equipped the virtual machine with 8 GB of DDR5 RAM and 6 AMD Ryzen 9 5900HS processor cores. We chose this configuration to give enough computational power for a stable and reliable experimental environment. We mitigate any potential impacts of power fluctuations by plugging the host machine into a socket during all experimentation, reducing the likelihood of performance fluctuations that could influence the results.

On our virtual machine, we conducted the experiments using Mininet version 2.3.1b1 [20]. We employ a Fat tree topology with k set to 4 (representing the number of switch ports). We configured all network links to have a bandwidth of $1Gbit/s$. Based on the rationale outlined in Section 3.6, we assumed no packet loss between switches. We simulate packet loss between workers and switches by dropping packets at the worker with a given probability. We set the probability of each scenario to 0.3, giving us a combined probability of $1 - (1 - 0.3)^2 = 0.51$ for packet loss to occur.

Due to the limited computational power of a single computer and to create a consistent and fair proving ground, we do not train an actual machine-learning model in our experiments. Instead, we use a custom module that generates pseudo-random values to simulate the gradients. In the experiments, we use a fixed seed to consistently obtain the same gradients across multiple runs, eliminating potential variability and enabling an equitable assessment of the various scenarios.

5.3 Experiment

The primary objective of this experiment is to investigate whether the level of switches in the proposed design significantly impacts the execution time, both when compared to each other and to the original SwitchML design. As the effect of the proposed design on execution time remains uncertain, we employ a two-sided statistical test with a significance level of $\alpha = 0.05$ to test the following hypothesis:

$$H_0 : \mu_i = \mu_j, \qquad \forall i, j \in \{\text{SML}, 1\,\text{lvl}, 2\,\text{lvls}, 3\,\text{lvls}\}, i \neq j$$

$$H_a : \mu_i \neq \mu_j, \qquad \forall i, j \in \{\text{SML}, 1\,\text{lvl}, 2\,\text{lvls}, 3\,\text{lvls}\}, i \neq j$$

5.3.1 Overview

To achieve the objectives of this experiment, we conducted multiple executions for each variation, comprising regular SwitchML, our design with one switch level, our design with two switch levels, and our design with three switch levels. In all variations, we employ two workers that send 42 data chunks. Choosing two workers allowed us to maintain consistency across all configurations, enabling a fair comparison of execution times. For each variation, we gathered a sample of 50 executions to ensure a robust statistical analysis. Figure 5.1 shows an overview of the aggregation trees used in the test setup.

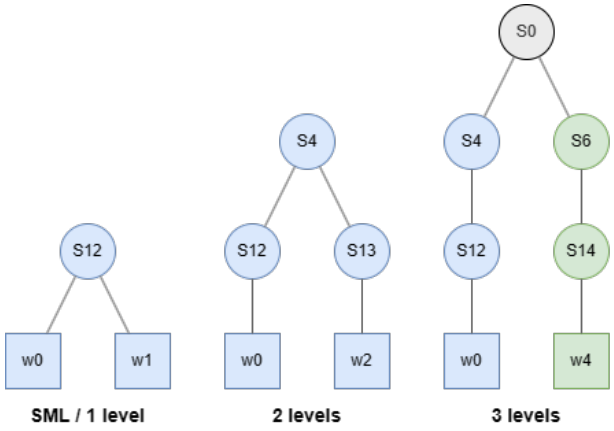

Figure 5.1: Aggregation trees used in the test setup

To record the execution time accurately, we developed a Python script specifically for this experiment. The script seamlessly manages the Mininet environment, ensuring a clean and reliable test setup for each execution. After execution, the script automatically restarts Mininet to prepare for the subsequent test, ensuring consistent conditions throughout the experiment. The test setup captures the duration from the initiation of the first worker until the completion of the last one.

As an initial exploratory step, we created a violin plot to gain a comprehensive overview of the gathered data. Figure 5.2 provides insights into our design's distributions and central tendencies compared to SwitchML. From this figure, it is evident that the average execution time of our design lies higher than that of SwitchML. The violin plot also reveals that the standard deviation is relatively close to the mean, indicating a tendency towards normality in the data distribution. To provide a more detailed understanding of the visualised data, we present Table 5.2, which offers comprehensive descriptive statistics for each configuration.

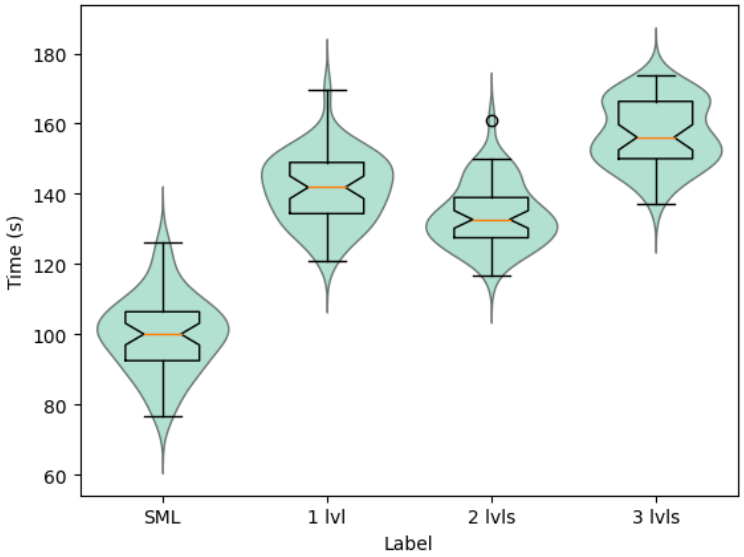

Figure 5.2: Violin plot of the execution time data per label

	SML	1 lvl	2 lvls	3 lvls
Mean	99.941932	141.565460	133.984410	157.235023
Min	76.527806	120.849598	116.776475	137.107641
Q1	92.468872	134.358586	127.488130	150.006657
Q2	100.035194	141.852354	132.694196	156.086791
Q3	106.407380	148.929488	138.946162	166.324582
Max	126.115696	169.645404	161.111706	173.627201

Table 5.2: Overview of the execution time data per label

5.3.2 Normality

Before proceeding with hypothesis testing, assessing whether the data follows a normal distribution is crucial. To accomplish this, we employed two essential methods. Firstly, we utilised QQ plots [26], as depicted in Figure 5.3, to visually inspect the distribution of all labels. The QQ plots indicate that the measurements closely follow the red reference line, indicating a normal distribution. While a few outliers are present, they do not significantly deviate from the overall pattern.

Secondly, we conducted the Shapiro-Wilk test [27] to validate the normality assumption

further, setting the significance level to $\alpha = 0.05$. Table 5.3 presents the results of this test. With all p-values above the alpha threshold, the Shapiro-Wilk test confirms that the data adheres to a normal distribution. Combining the insights from both steps of the normality testing process, we find enough statistical evidence to support the conclusion that the data follows a normal distribution.

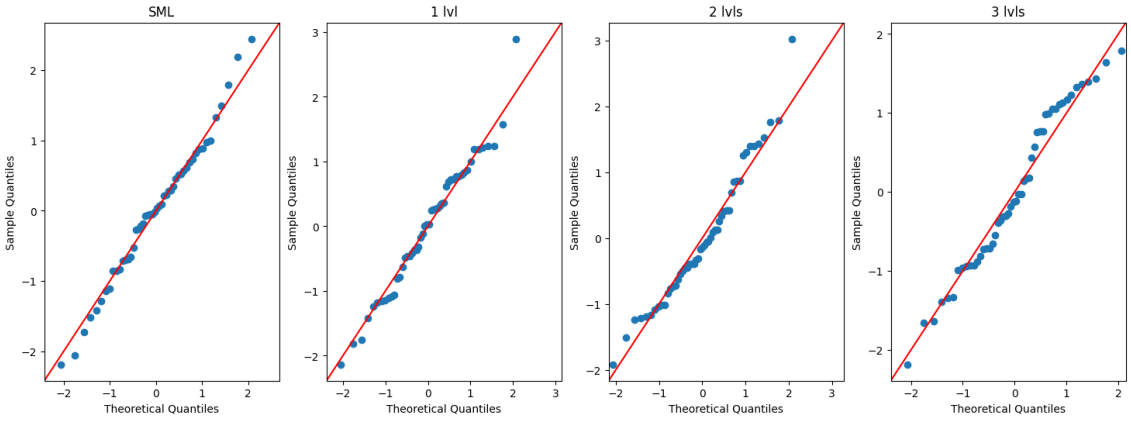

Figure 5.3: QQ-Plots of the execution time data per label

Data	p-value
SML	0.9318583607673645
1 lvl	0.5345391035079956
2 lvls	0.11688331514596939
3 lvls	0.10745035111904144

Table 5.3: P-values of the Shapiro-Wilk test on the execution time data

5.3.3 Hypothesis Testing

Based on the normality test results in the previous subsection, the t-test [26] was chosen as the appropriate statistical method to evaluate our hypothesis. Table 5.4 presents the obtained p-values and t-scores from the performed t-test. Upon examination of this table, it is evident that all calculated p-values are below the chosen alpha, leading us to reject the null hypothesis and accept the alternative hypothesis.

Additionally, the t-scores captured in the table offer practical insights into the direction of the observed differences. Almost every test exhibits a negative t-score, suggesting that the mean of group A1 lies below the mean of group A2 for the respective comparisons.

This finding indicates a consistent trend across the majority of the tested configurations. However, one noteworthy exception stands out when comparing one level's mean to two switch levels. In this instance, we observe a positive t-score, indicating that the mean of the two levels configuration is lower than that of the one-level configuration of switches.

$\mathbf{A_1}$	$\mathbf{A_2}$	**p-value**	**t-score**
SML	1 lvl	1.2042599007550021e-36	-20.161985137440475
SML	2 lvls	2.004151297271285e-31	-17.1592847531964
SML	3 lvls	4.0014805407095435e-49	-28.42394039224852
1 lvl	2 lvls	0.00010621611953559026	4.038569100894637
1 lvl	3 lvls	9.006809538258652e-13	-8.210366110314682
2 lvls	3 lvls	1.3504205142577843e-22	-12.743933455727342

Table 5.4: P-values and t-scores for all hypotheses

6

Discussion

In this chapter, we present the discussion of this paper. We interpret the results from the design and experiment in section 6.1. Then in section 6.2, we identify and discuss the limitations of our work, acknowledging potential constraints and areas for improvement. Finally, in section 6.3, we offer valuable suggestions for future research directions, exploring possible avenues to expand upon and enhance the scope of this study.

6.1 Findings

RQ1 In Chapter 3, we thoroughly discussed our design and the various design choices. Based on this design, an appropriate SDN controller must be selected to enable runtime scalability. Additionally, we need to implement a mechanism for worker synchronisation, as workers can join the aggregation process at any time. To support dynamic changes in the number of workers, the switches in our network require a flexible entry in the data plane rather than a fixed constant value. This entry will allow on-the-fly adjustments and enable our design to handle varying numbers of workers effectively.

RQ2 In the experiment, we compared the means of execution time to assess the performance difference between our implementation and SwitchML. The statistical analysis produced evidence to support the existence of significant variations among the means. Notably, our findings reveal a substantial increase in execution time for our implementation with a single switch level compared to SwitchML. This difference in means indicates that the modifications to facilitate runtime scalability have adversely affected the overall performance. These results highlight the trade-off between runtime scalability and performance.

When comparing the means of different switch levels, precisely one level to three levels, it becomes apparent that the increase in the mean execution time is less pronounced. This difference suggests that the overhead incurred between different switch levels is relatively minor but still present, highlighting the critical role of locality in achieving optimal performance.

When examining the execution time of two levels of switches, a notable observation emerges: the mean execution time for this configuration is lower compared to both one and three levels of switches. However, it is essential to consider certain factors that may have influenced these results. During the recording of execution times for one and three levels of switches, unexpected freezing of the Mininet occurred multiple times, necessitating the restart of the test script. Consequently, the results for these configurations consist of several partial runs. We recorded the test runs for one, two and three switch levels on separate days when the temperature was relatively high; the day on which we measured the execution time for two switch levels exhibited significantly lower temperatures compared to the other days. This variance in temperature may have introduced an additional factor affecting the execution time results. Considering these factors, it is reasonable to suggest that the lower mean execution time observed in the two levels of switches configuration can likely be due to a combination of lower temperatures and more efficient caching utilisation.

6.2 Limitations

In this paper, we explored ways to extend SwitchML with runtime scalability and the effect this has on performance. Recognising that the research has limitations we should consider when interpreting the results is essential. These limitations encompass various aspects of the research design, data collection, and generalisability of the findings.

A limitation of this paper lies in utilising Mininet as the network virtualisation program. While Mininet provides a flexible and efficient platform for emulating network environments, it is essential to acknowledge that network virtualisation may only partially replicate physical networks' complexities, performance characteristics and behaviour. As a result, the findings from our experiments using Mininet may not precisely mirror the outcomes of a physical network.

Another limitation encountered during this paper's design and implementation phase was the inherent constraints posed by Mininet, which prevented us from designing and implementing a packet loss recovery mechanism between switches without causing network congestion. As a result, we had to assume that no packet loss would occur between switches.

Although necessary under the circumstances, this assumption may lead to underestimating the actual difference in execution time between switch levels. Therefore, we should interpret the execution time means cautiously, as they may be smaller than in a real-world setting with packet loss.

An essential limitation of this paper is related to the execution of the experiment under ideal conditions. While we tried to control potential confounding variables and create a controlled environment, it is crucial to acknowledge that real-world scenarios may differ from the controlled setting. We did not account for factors such as additional network traffic and congestion in this paper. Therefore, the findings should be interpreted within the context of the controlled conditions and may only partially capture the complexity and variability in realistic settings.

Lastly, a notable limitation of this paper is the sample size of the collected data. Due to time constraints in the research timeline, gathering a larger sample size was not feasible. The findings may be influenced by the relatively small sample, potentially limiting the generalisability of the results. Even though we tried to ensure the sample's representativeness, a more significant sample would have enhanced the statistical power and strengthened the external validity of the paper.

6.3 Future Work

This section highlights potential future research and development directions based on our findings. While we contributed to the topic with our research, some avenues still need to be explored. We describe areas for further exploration in this part, addressing both the limits of the current study and prospective extensions.

As mentioned previously, a notable limitation of this paper is the inability to ensure reliable packet transmission between switches. Addressing this limitation and improving the overall reliability of the network infrastructure represents a critical area for future research. Further research can focus on developing a robust design that guarantees the reliability of packets during their transfer between switches.

In the current design, a worker handles packet loss by retransmitting the packet upon time-out, using a fixed interval. However, a common scenario is that a worker successfully sends a packet to the switch but experiences a time-out because it has not yet received a response. This situation occurs when the switch waits for responses from other workers, leading the worker to send unnecessary packets. Future work could investigate the effects

of employing a dynamic interval that gradually increases upon time-out, particularly when utilising different levels of switches.

Based on the findings derived from our experiment, the inclusion of runtime scalability support introduces an overhead that adversely impacts performance. Future research could focus on identifying weaknesses within our design. Once further research identifies the weak points, it can focus on devising strategies to minimise the adverse impact on overall performance.

7

Related Work

In this section, we will explore various papers related to our subject. To organise these papers, we have divided them into two subsections. The first subsection covers papers with a broader focus on in-network computations, while the second subsection focuses specifically on papers related to in-network aggregation. Each paper in both subsections will be summarised and discussed in terms of how it differs from the current paper.

7.1 In-Network Computation

A study by Sapio and his team [28] aimed to address the question of what types of computations should be performed by networks as they become capable of computing, particularly in the context of data centre networks. The study aimed to identify computations that could be done within the network while significantly reducing network traffic, requiring minimal changes to the application, and ensuring the correctness of the computation. The authors suggested that applications using a partition/aggregate workload pattern could meet these criteria, including frameworks such as MapReduce, big data analytics, machine learning, graph processing, and stream processing. The researchers concluded that it is time to offload some computational tasks to the network, but programmers need to construct tasks in compliance with the limitations of programmable devices. The experiments showed that in-network computation presents opportunities for aggregation functions. This paper closely relates to ours, looking at the best ways to offload computations into the network. However, this paper differs in its goal. The described paper looks at whether in-network computation is possible, while this paper looks specifically at how to scale in-network aggregation.

IncBricks is a system created by Liu et al. [29] that tries to achieve the idea of in-network computation: offloading a set of compute operations from end servers to programmable network devices so that it reduces network traffic, serves remote operations on the fly with low latency, and servers can be in low power mode. While also overcoming the three challenges of offloading computation to the network (I) limited hardware resources, (II) keeping states coherent across networking elements is complex due to network component failures and many different paths between end-points, and (III) To serve a wide range of data centre applications, in-network computation requires a simple and general computing abstraction that can connect with application logic. IncBricks is a co-design hardware-software in-network caching fabric with basic computing primitives for data centres. IncBricks consists of two components. The first component is IncBox, a programmable middlebox that combines a reconfigurable switch and a network accelerator. The second component is IncCache, a distributed key-value store built on IncBoxes. Prototypes have shown a 30% decrease in request latency and double the throughput. This paper differs from ours as we try to offload the computations onto the switch instead of using special middleboxes that require changes in the network.

7.2 In-Network Aggregation

Sapio *et al.* [4] conducted a study to mitigate the communication bottleneck of distributed machine-learning applications that use in-network aggregation. This paper shows that in-network aggregation implemented on programmable switches can accelerate the workloads of distributed machine-learning models. The authors created SwitchML, a co-design of in-switch processing with an end-host transport layer and machine-learning frameworks. SwitchML handles the acceleration part and addresses challenges that come with using programmable switches (I) limited processing capabilities and on-chip memory, (II) machine-learning frameworks use floating points while the computing units of programable switches can only handle integer values, and (III) in-network aggregation deviates from standard communication patterns like multi- and unicast, thus requiring mechanisms for synchronising workers and detecting packet loss. The proposed solutions the authors show that programmable switches can perform in-network aggregation at line rate. While the authors of SwitchML focused on creating a solution to reduce the communication bottleneck and handling the additional challenges that come with using programmable switches, this paper focuses on expanding the findings and idea of SwitchML to support dynamically

adding/removing workers at runtime in a multi-layered tree topology. Therefore this paper can be considered complementary to SwitchML.

Mai *et al.* [30] want to solve the network performance bottleneck of the partition/aggregation pattern. This pattern distributes tasks across servers that process data locally, and then the partial results are sent and aggregated at edge servers. The aggregation step causes the network to be the bottleneck, as the network struggles to support many-to-few, high-bandwidth traffic between servers. In this paper, the authors show with the creation of NetAgg that aggregating along network paths is a more efficient way of aggregating than aggregating on edge servers. NetAgg software platform uses on-path aggregation to aggregate values for network-bound partition/aggregation applications. NetAgg uses middleboxes with high bandwidth links to switches for aggregating values, mitigating hotspots in the network by only forwarding a tiny fraction of the initial traffic to the next hop. This paper relates closely to our paper in that it achieves aggregation using multiple levels of switches. However, it differs because we try to achieve this by utilising the switch as the aggregator instead of using middleboxes.

De Sensi et al. [31] propose Flare, a flexible in-network Allreduce accelerator, which addresses the limitations of existing in-network Allreduce solutions. The authors identified three limitations that they want to solve with Flare. The first one is custom operators and data types. Existing solutions lack the support for complete customizability of operators and data types due to limitations like not having floating point units on programmable switches. Flare solves this by utilising a PsPIN switch. The second limitation is sparse data. Many solutions must handle sparse data and may only send the non-zero values to optimise performance. Flare solves the handling of sparse data with a specially designed algorithm for sparse Allreduce. The third and last limitation is reproducibility. Scientific applications may require the computed results to be reproducible across different runs. Flare guarantees reproducibility. Like our design, Flare uses a tree to reduce the data at each hop. In contrast to our design, Flare requires changes in the network to work, which differs from our design. Besides this, the focus of Flare lies on solving different problems to ours.

8

Conclusion

This paper presented a novel design for achieving runtime scalability in SwitchML. Our approach involved redefining the execution flow and incorporating an SDN controller to efficiently manage runtime changes in the data plane. By offloading most of the computations required for scalability to the SDN controller, we aimed to enhance the system's adaptability and flexibility.

Critical aspects of our design involved modifying the worker's execution flow to enable seamless registration onto the network and synchronisation before starting the aggregation process. Additionally, we extended the logic of the switches to facilitate worker synchronisation, a critical element for efficient runtime scalability.

The successful implementation of our design allowed us to evaluate its performance based on execution time. However, our findings revealed an essential trade-off that we must consider. While achieving runtime scalability was successful, the changes introduced to support it resulted in an overhead that negatively impacted overall system performance. This trade-off highlights the significance of balancing runtime scalability and system efficiency when designing distributed machine learning systems.

Furthermore, our evaluation sheds light on the impact of increasing switch levels on the in-network aggregation process. As the number of switch levels rises, the packets must traverse longer paths within the network, leading to slower in-network aggregation. This insight emphasises the importance of carefully considering the number of switch levels and their implications on system performance.

In conclusion, our proposed design and evaluation provide a stepping stone towards achieving runtime scalability in SwitchML. We have identified areas for refinement and optimisation, offering valuable guidance for future research and advancements in distributed machine learning systems.

References

[1] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. **A Survey on Distributed Machine Learning**. *ACM Comput. Surv.*, **53**(2), mar 2020. 1

[2] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. **Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce**. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2262–2270, New York, NY, USA, 2021. Association for Computing Machinery. 1

[3] Elena Fasolo, Michele Rossi, Jorg Widmer, and Michele Zorzi. **In-network aggregation techniques for wireless sensor networks: a survey**. *IEEE Wireless Communications*, **14**(2):70–87, 2007. 2

[4] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. **Scaling Distributed Machine Learning with In-Network Aggregation**. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021. 2, 9, 18, 38

[5] OpenAI. **GPT-4 Technical Report**. 2023. 2

[6] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. **Language models are unsupervised multitask learners**. *OpenAI blog*, **1**(8):9, 2019. 2

[7] Humberto Cervantes and Rick Kazman. *Designing Software Architectures: A Practical Approach*. Addison-Wesley Professional, May 2016. 2

REFERENCES

[8] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. **Software-defined networking: A comprehensive survey**. *Proceedings of the IEEE*, **103**(1):14–76, 2014. 5

[9] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. **OpenFlow: enabling innovation in campus networks**. *ACM SIGCOMM computer communication review*, **38**(2):69–74, 2008. 5

[10] **Open Networking Foundation**, 7 2022. 6

[11] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. *Software-Defined Networking 2*. John Wiley & Sons, 1 2023. 6

[12] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. **Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN**. *ACM SIGCOMM Computer Communication Review*, **43**(4):99–110, 2013. 7

[13] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. **P4: Programming Protocol-Independent Packet Processors**. *SIGCOMM Comput. Commun. Rev.*, **44**(3):87–95, jul 2014. 7, 19

[14] Aoxiang Feng, Dezun Dong, Fei Lei, Junchao Ma, Enda Yu, and Ruiqi Wang. **In-network aggregation for data center networks: A survey**. *Computer Communications*, **198**:63–76, 2023. 8

[15] Suhail Ahmad and Ajaz Hussain Mir. **Scalability, Consistency, Reliability and Security in SDN Controllers: A Survey of Diverse SDN Controllers**. *J. Netw. Syst. Manage.*, **29**(1), jan 2021. 15

[16] Y. Daniel Liang. *Introduction to Java Programming*. Prentice Hall, 1 2015. 15

[17] Alfred V. Aho, Yehoshua Sagiv, Thomas G. Szymanski, and Jeffrey D. Ullman. **Inferring a Tree from Lowest Common Ancestors with an Application to the Optimization of Relational Expressions**. *SIAM Journal on Computing*, **10**(3):405–421, 8 1981. 15

[18] Deep Medhi and Karthik Ramasamy. *Network Routing*. Morgan Kaufmann, 9 2017. 15, 16

[19] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. **Event-Driven Packet Processing**. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 133–140, New York, NY, USA, 2019. Association for Computing Machinery. 18

[20] Mininet Project Contributors. **MiniNet**. 19, 27

[21] **Welcome to Python.org**, 7 2023. 19

[22] P4lang. **GitHub - p4lang/p4app at rc-2.0.0**. 19

[23] The P4.org API Working Group. **P4Runtime specification**. 19

[24] Oracle. **VirtualBox**. 26

[25] Canonical and community. **Ubuntu**. 26

[26] Mario F. Triola. *Elementary Statistics*. Pearson, 12 2012. 29, 30

[27] Stephen M. Shapiro and M. B. Wilk. **An analysis of variance test for normality (Complete samples)**. *Biometrika*, **52**(3/4):591, 12 1965. 29

[28] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. **In-Network Computation is a Dumb Idea Whose Time Has Come**. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 150–156, New York, NY, USA, 2017. Association for Computing Machinery. 37

[29] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. **IncBricks: Toward In-Network Computation with an In-Network Cache**. *SIGARCH Comput. Archit. News*, **45**(1):795–809, apr 2017. 38

[30] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. **NetAgg: Using Middleboxes for Application-Specific On-Path Aggregation in Data Centres**. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 249–262, New York, NY, USA, 2014. Association for Computing Machinery. 39

[31] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. **Flare: Flexible in-Network Allreduce**. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. 39

Appendix A

Raw Data

	SML	**1 lvl**	**2 lvls**	**3 lvls**
0	92.45228242874146	148.60922813415527	161.1117057800293	154.46799325942993
1	100.7163519859314	148.19886469841003	141.6824598312378	167.99492716789246
2	105.42670488357544	141.89597821235657	146.84343028068542	169.7701506614685
3	90.79892683029176	138.43346405029297	124.68658304214478	166.91838669776917
4	109.41022300720216	127.75774121284483	145.67909288406372	151.19594812393188
5	110.6351923942566	149.0410089492798	132.6941957473755	166.33815264701843
6	97.17959260940552	133.97735238075256	127.09349131584167	167.65024495124817
7	99.20412921905518	145.15592885017395	150.0050971508026	166.2838716506958
8	94.3239929676056	151.2487690448761	129.10901641845703	158.85383296012878
9	87.7402012348175	130.81603908538818	126.46026825904846	150.64372634887695
10	100.99350762367249	147.50949454307556	133.38699507713318	154.40457224845886

11	102.88533735275269	144.11457991600037	129.5205192565918	156.04174137115548
12	115.98627281188963	139.8983871936798	149.83756613731384	144.9837999343872
13	106.53723573684692	138.04191637039185	129.98619103431702	148.69612073898315
14	99.44532513618468	137.13335132598877	137.017724275589	169.43370914459229
15	77.93767166137695	149.27611184120178	131.00508522987366	153.90054416656494
16	90.83755898475648	149.94204592704773	130.44445037841797	164.32729506492615
17	83.67376232147217	135.50228667259216	130.43247437477112	172.32822728157043
18	123.35367441177368	156.8398814201355	133.0167429447174	166.91563534736633
19	99.34703922271729	144.0993525981903	130.4410102367401	149.81485104560852
20	92.51863980293274	120.84959769248962	124.89566159248352	148.72879004478455
21	107.74119925498962	124.55466032028198	122.86847257614136	167.43682956695557
22	92.30511951446532	169.64540433883667	129.88990545272827	149.17715048789978
23	102.22520756721497	136.8832266330719	123.30561900138856	158.7977933883667
24	102.30469155311584	130.98941373825073	137.74849486351013	157.01824712753296
25	126.11569619178772	145.01718163490295	128.3625464439392	148.62978506088257
26	107.33098220825195	153.10429906845093	147.66912341111786	156.99932289123535
27	92.86878514289856	137.9754557609558	141.74831414222717	168.49245285987854
28	104.78274767918396	130.08810973167 42	134.07487964630127	144.8906753063202
29	99.255309343338	144.31301641464233	131.25480365753174	155.5131733417511
30	88.10439920425415	148.6175775527954	123.46489691734314	142.05215716362
31	114.17967629432678	144.68294477462769	137.7590148448944	153.67768788337708
32	109.26583576202393	153.60177326202393	134.80705308914185	161.17702078819275

33	99.76135754585266	123.96649551391602	127.52999830245972	150.5820734500885
34	84.73523116111755	153.3454787731171	141.72534584999084	150.70430445671082
35	86.13534283638	141.58774161338806	140.1333086490631	162.4527189731598
36	103.0518124103546	133.73086309432983	132.49054408073425	154.75888347625732
37	103.6951322555542	149.03345775604248	124.87853860855104	144.42397332191467
38	108.69511795043944	153.10762882232666	146.49826002120972	148.1735327243805
39	106.01781225204468	140.4278542995453	127.44626188278198	137.1076409816742
40	91.02038764953612	143.88921999931335	135.14362382888794	142.18043613433838
41	119.05895757675172	137.13694214820862	136.31812691688538	173.6272006034851
42	97.67893409729004	129.580246925354	135.14061999320984	152.1759753227234
43	105.5861837863922	149.6072015762329	120.4218237400055	156.13184094429016
44	100.30903029441832	153.635648727417	116.77647519111632	170.05793380737305
45	76.52780604362488	141.8087294101715	123.08813643455504	164.21841311454773
46	110.4333918094635	130.40798926353455	146.54433917999268	170.40301203727722
47	97.0921552181244	131.32908725738525	124.2818796634674	158.5081775188446
48	97.96013236045836	130.33943796157837	137.67934441566467	164.29591464996338
49	81.45452809333801	137.52451419830322	133.52207779884338	148.3942837715149