Meta-Scheduling for MultiPath TCP with Neuroevolution of Augmenting Topologies

Meta-Scheduling für MultiPath TCP mittels Neuroevolution of Augmenting Topologies
Master-Thesis

Kay Luis Wallaschek
KOM-M-0717

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Elektrotechnik
und Informationstechnik
Fachbereich Informatik (Zweitmitglied)

Fachgebiet Multimedia Kommunikation
Prof. Dr.-Ing. Ralf Steinmetz

Meta-Scheduling for MultiPath TCP with Neuroevolution of Augmenting Topologies
Meta-Scheduling für MultiPath TCP mittels Neuroevolution of Augmenting Topologies

Master-Thesis
Studiengang: Informatik
KOM-M-0717

Eingereicht von Kay Luis Wallaschek
Tag der Einreichung: 14. August 2020

Gutachter: Prof. Dr.-Ing. Ralf Steinmetz
Betreuer: Jun.-Prof. Dr.-Ing. Amr Rizk, Prof. Dr. Boris Koldehofe

Technische Universität Darmstadt
Fachbereich Elektrotechnik und Informationstechnik
Fachbereich Informatik (Zweitmitglied)

Fachgebiet Multimedia Kommunikation (KOM)
Prof. Dr.-Ing. Ralf Steinmetz

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 und § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Kay Luis Wallaschek, die vorliegende Master-Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, den 14. August 2020

Kay Luis Wallaschek

Thesis Statement pursuant to § 22 paragraph 7 and § 23 paragraph 7 of APB TU Darmstadt

English translation for information purposes only:
I herewith formally declare that I, Kay Luis Wallaschek, have written the submitted thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are pursuant to § 23 paragraph 7 of APB identical in content.

For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Darmstadt, 14. August 2020

Kay Luis Wallaschek

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Problem Statement and Contribution . . . 3
- 1.3 Outline . . . 4

2 Background **5**
- 2.1 MultiPath TCP . . . 5
  - 2.1.1 Connection Management . . . 6
  - 2.1.2 Path-Manager . . . 6
  - 2.1.3 Congestion Control . . . 6
  - 2.1.4 Scheduler . . . 7
- 2.2 Neuroevolution of Augmenting Topologies - NEAT . . . 8
- 2.3 ProgMP . . . 12

3 Related Work **13**
- 3.1 Scheduling . . . 13
- 3.2 NEAT . . . 14
- 3.3 Summary . . . 14

4 Design **15**
- 4.1 Learning a Meta Scheduler . . . 16
  - 4.1.1 Learning . . . 16
  - 4.1.2 Creation . . . 18
- 4.2 Learning a Static Scheduler . . . 19
  - 4.2.1 Changes to the Learning Setup . . . 19
  - 4.2.2 Changes to the Neural Network . . . 20
- 4.3 Fusion of Learned Static Schedulers and Meta Scheduler . . . 20

5 Implementation **23**
- 5.1 MultiPath TCP and Scheduling in ns-3 . . . 23
- 5.2 Implementation of NEAT . . . 24
- 5.3 Implementation of Neural Network Based Schedulers and Connection of NEAT and ns-3 . . . 26
- 5.4 Translator . . . 28
  - 5.4.1 How does the translator translate? . . . 28
  - 5.4.2 Limitations of using the translator . . . 29
  - 5.4.3 Example Translation . . . 29
- 5.5 Summary . . . 29

6 Evaluation **31**
- 6.1 Evaluation Setup . . . 31
- 6.2 Evaluation Results and Analysis . . . 32
  - 6.2.1 Simple Two Flow Topology - ns-3 . . . 33
  - 6.2.2 Double Dumbbell 1 - ns-3 . . . 35
  - 6.2.3 Double Dumbbell 2 - ns-3 . . . 37
  - 6.2.4 Double Dumbbell 3 - ns-3 . . . 37
  - 6.2.5 Conclusions on ns-3 Evaluations . . . 39
  - 6.2.6 Mininet Evaluations . . . 39

7 Conclusions **41**
- 7.1 Future Work . . . 41

8 Appendix **43**

Bibliography **50**

List of Figures

2.1 Multipath TCP in the TCP/IP Layer Model . . . 5
2.2 Difference between Uncoupled Congestion Control and Coupled Congestion Control . . . 7
2.3 Multipath TCP Scheduling . . . 7
2.4 McCulloch-Pitts Neuron Model . . . 8
2.5 Sample Fully-Connected Neural Network . . . 8
2.6 Overview of NEAT's Learning Process . . . 9
2.7 Mutations in NEAT . . . 10
2.8 Crossover done by NEAT . . . 11

4.1 Overview of the Process from Learning to Deployment. . . . 15
4.2 Extention to NEAT's Evaluation Step . . . 17
4.3 Translator . . . 18
4.4 Meta Scheduler Structure . . . 19
4.5 Static Neural Network Based Scheduler Structure . . . 20
4.6 Supported Measures in ns-3 and ProgMP . . . 21

5.1 Inheritance Diagram of MultiPath TCP in ns-3 . . . 23
5.2 Structure of NEAT's Implementation . . . 25
5.3 Extension of ns-3 with NEAT . . . 26
5.4 NEAT's Learning Process in ns-3 . . . 27

6.1 Simple Two Flow Topology . . . 31
6.2 Double Dumbbell Topology . . . 32
6.3 Simple Two Flow Topology in ns-3 . . . 34
6.4 Double Dumbbell Topology 1 in ns-3 . . . 35
6.5 Double Dumbbell Topology 2 in ns-3 . . . 36
6.6 Double Dumbbell Topology 3 in ns-3 . . . 38
6.7 Double Dumbbell Topology 3 in Mininet . . . 40

8.1 Sample Translation Neural Network Structure . . . 47

List of Tables

6.1 The Different Configurations of the Double Dumbbell Topology on a per Link Basis 32
6.2 Evolved Schedulers 33
6.3 Averages of Simple Two Flow Topology with MinRtt30 as Baseline 34
6.4 Averages of Double Dumbbell 1 Topology with MinRtt30 as Baseline 36
6.5 Averages of Double Dumbbell 2 Topology with MinRtt30 as Baseline 37
6.6 Averages of Double Dumbbell 3 Topology with MinRtt30 as Baseline 38

Abstract

Using MultiPath TCP provides many benefits, but not every component of it is yet well understood. Besides congestion control and path-manager, scheduling is one of the main parts of MultiPath TCP. MultiPath TCP allows for connections with multiple TCP subflows, which are managed by the scheduler. There are many approaches to implement schedulers, from simple rules up to complicated deep neural networks. However, most of them come with problems. The schedulers that use simple rules, like the MinRTT scheduler, lack in performance on a wider range of different topologies. Whereas deep neural network based schedulers need lots of resources for learning and in operation.

We present a new way of creating schedulers. By using the genetic algorithm NEAT, neural networks can be evolved to tackle the problem of scheduling. These neural networks are of minimal size and thus don't need many resources in operation, unlike deep neural networks. We present two kinds of neural network based schedulers: static schedulers, which directly choose the path for each packet for a certain topology, and meta schedulers, which don't conduct scheduling directly. Rather, these meta schedulers have a pool of schedulers out of which they choose one for each invocation and switch the packet in regards to the chosen scheduler. We argue that small neural network should switch between well known and easy scheduling concepts to cover a wide variety of network topologies, while still having low overhead.

Our evaluation shows that it is possible to create static and meta schedulers with our approach. The evolved static scheduler reduces the mean end-to-end delay by about 17%, increases fairness in regards to Jain's Fairness Index by 15%, while having about the same total throughput, comparing to the well known Round Robin scheduler on a homogeneous topology with one two-subflow MultiPath TCP connection and two traditional TCP crossflows. The evolved meta scheduler, which can choose between MinRTT and RoundRobin, outperforms both of them on nearly every tested topology. This indicates that it is of benefit to be able to switch the scheduler within a data transmission.

1 Introduction

1.1 Motivation

Using TCP on mobile devices usually coincide problems. TCP connections are identified in part by the IP addresses of the sender and the receiver. Mobile devices usually have multiple interfaces and thus can be connected to the Internet in multiple ways at the same time. Each of these interfaces has a different IP address. However, TCP does not utilize multiple interfaces. For example: A mobile device initiates a TCP connection over Wi-Fi. Then the device moves out of the range of the Wi-Fi network and switches over to LTE. In this scenario, as the IP address changes, the TCP connection cannot be maintained and needs to be initiated again. To solve this problem, MultiPath TCP has been created [BPB11].
With MultiPath TCP it is possible to maintain TCP connections over multiple interfaces. This not only solves the prior mentioned scenario, but can also provide a performance boost, as multiple interfaces can be used simultaneously for data transmission. MultiPath TCP is not only useful in a mobile environment. In data-centers, for example, it can be used for load-balancing, where multiple servers can serve a single client [RPB+10].
MultiPath TCP consists of three components: path-manager, congestion control and the scheduler. The path-manager defines how to create and handle paths. Congestion control has the same purpose as in traditional TCP, but it is done differently. Whereas in traditional TCP the single connection has one congestion window, in MultiPath TCP there are multiple ways how to handle congestion: Treat every subflow as a traditional TCP flow or have one congestion window for the whole MultiPath TCP connection. Also, with wVegas [YMX12], some traditional congestion control algorithms have been adapted to MultiPath TCP. The scheduler answers the question: How should the packets be distributed over multiple paths? There exist a wide variety of schedulers with different targets. For example: A full redundant scheduler might send all packets redundantly over all paths simultaneously. Whereas another scheduler might saturate all paths with unique packets to improve throughput. However, the area of MultiPath TCP scheduling is not yet well understood.
With the recent rise of machine learning, especially deep learning, efforts have been made to create schedulers by using deep neural networks [ZLG+19][LSL19][XTY+19]. These consist of large deep neural networks, that usually get evaluated for every packet. However, this results in a big overhead compared to a simple rule based scheduler. Schedulers that need big amounts of resources can not be used by every device. Most mobile devices don't have or don't want to invest these resources for energy preserving reasons. For mobile devices, a low overhead, energy efficient scheduler is preferably used.
Also, scheduling is done on the transport layer, usually directly in the kernel, where performance is important as well. In the worst case, using a big deep neural network for scheduling could use up all the available resources, making the device unusable. This is unreasonable, so low-overhead, fast and simple schedulers should be used and developed. Especially having mobile devices in mind, where energy consumption is of big concern.
In environments where energy consumption is of no concern, for example in data-centers or server farms, online learning based approaches might be of use. But once the topology changes the scheduler needs to be retrained. As especially connections to mobile devices are unstable and frequently changing, the scheduler would need to be frequently retrained, which may result in performance issues. There exists a vast amount of different schedulers all outperforming each other on different network configurations. Hence, each possible network configuration has an 'optimal' scheduler. Thus, we argue that there can not be a scheduler that is optimal for all configurations, while also being low-overhead, fast and simple. This argument has been brought up and validated for queuing and scheduling in network switches by Sivaraman et al. [SWSB13].

To tackle the problem of scheduling we combine the idea of learning with conventional scheduling. In conventional scheduling, simple rules like "always use the path with the lowest round-trip time" are used. We argue that instead of a big deep neural network, a small network that switches between conventional schedulers should be used for scheduling to save resources, while also performing well on multiple network configurations.
We create schedulers for MultiPath TCP schedulers by using NEAT, a genetic learning algorithm. NEAT, or NeuralEvolution of Augmenting Topologies, creates minimal neural networks while optimizing to a goal, which fulfills our target of having the least amount of overhead.

1.2 Problem Statement and Contribution

We argue that a scheduler that performs optimally in every possible network scenario does not exist. Recently, there have been efforts to learn MultiPath TCP scheduling with deep neural nets, which may perform well, but are usually reliant on online learning. This poses a problem if the network topology frequently changes, as then the scheduler needs to relearn frequently. Relearning a deep neural network takes lots of resources, so in this scenario, online learning based schedulers have a big overhead. As scheduling is a low level matter, usually running directly on the kernel, the overhead should be as little as possible. We argue, that instead of building big neural networks, a small neural network should switch between

well known and simple scheduling concepts. Also, deployment of learning based schedulers poses a problem. With ProgMP [FRE+17] it is possible to easily incorporate schedulers into the linux kernel. As ProgMP is quite restrictive, most learning based schedulers cannot be deployed with ProgMP.
We create a meta scheduler that can be easily deployed with ProgMP and has some of the flexibility of online learning based schedulers, without relearning. We use NEAT for the creation of a neural network that switches between well known easy schedulers according to the current network state. We use NEAT, because it creates the neural networks with minimal topology, so the overhead we introduce is minimal.
The contributions of this thesis are as follows:

- Design, implementation and evaluation of a technique to generate simple- and meta-schedulers for MultiPath TCP.
- Generation of a meta scheduler that optimizes throughput, fairness and delay.
- Generation of a static scheduler that optimizes throughput, fairness and delay.
- Creation of a translator, that translates NEAT neural network specifications into ProgMP scheduler descriptions.
- Evaluation of generated schedulers in mininet with ProgMP.

1.3 Outline

This thesis is structured as follows: In chapter 2 the necessary background to understand this work is described. Then, chapter 3 consists of related work and chapter 4 gives a detailed overview of our approach and design decisions. Afterwards, in chapter 5, how we implemented our approach is described. Finally, chapter 6 consists of the evaluation, followed by conclusions and future work in chapter 7.

2 Background

2.1 MultiPath TCP

MultiPath TCP is an extension to TCP that allows for multiple paths over multiple interfaces per connection between hosts. In this section, we give an overview about how MultiPath TCP works, as well as in-depth explanations about the components it implements.

MultiPath TCP was first created to solve TCP problems on mobile devices. Namely, with traditional TCP there is no handover of connections to other interfaces. This is problematic, since mobile devices are prone to move out of range of wireless networks while still being able to connect to the internet via a cellular network. Because in traditional TCP the connections are in part identified by the IP addresses, which would change during such a handover, they would break down. MultiPath TCP solves this problem by allowing devices to maintain the connection via multiple interfaces. For example, if a mobile device is in range of a Wi-Fi and LTE network, it could maintain a connection to a data center via both networks simultaneously. So, if it would move out of the range of one of the networks the connection would not break down, rather it continues via the network it still is in range of.
However, MultiPath TCP provides more than reliability, it also allows to improve performance. MultiPath TCP utilizes multiple interfaces by maintaining multiple traditional TCP connections, called subflows. Each of them can use a different path to the destination, which means that a MultiPath TCP connection can use more resources. This is also an improvement for data-centers, where load-balancing is of great importance [RBP+11]. Being able to use multiple subflows over multiple paths, MultiPath TCP is able to improve the reliability of connections, performance as well as load-balancing [BHR12].
One of the design goals for MultiPath TCP was for applications to be able to seamlessly use it without changes. Rather, the application would just use the regular API for connection build up and if both the sender and the receiver support MultiPath TCP, more subflows would automatically be created [RPB+12]. In order to achieve this goal, MultiPath TCP is implemented on the transport layer like traditional TCP, visualized in Figure 2.1. The function of MultiPath TCP works on top of subflows. The subflows are managed by multiple components, namely the path-manager, congestion control and the scheduler. These components are further explained in the following subsections.

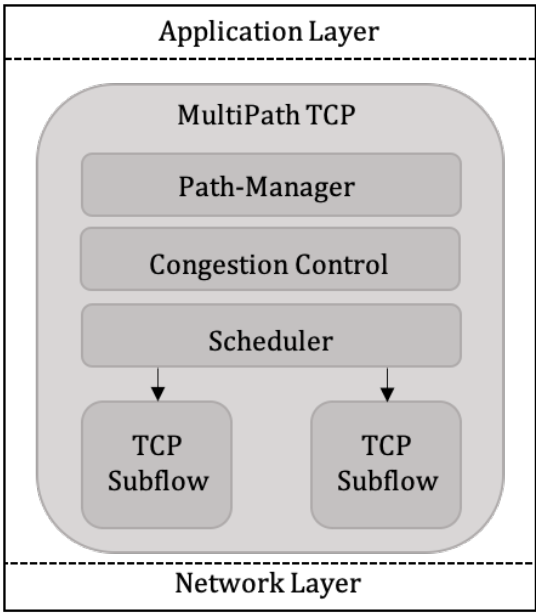


Figure 2.1: Multipath TCP in the TCP/IP Layer Model

2.1.1 Connection Management

MultiPath TCP uses the TCP three-way handshake to initialize connections. In this handshake the option *MP_CAPABLE* is set to signal capability to use MultiPath TCP. If it is not echoed by the other party, traditional TCP is used. However, there is a possibility that middle boxes drop these options. This results in a scenario where, even tough both parties are capable and wish to use MultiPath TCP, there is a possibility that it won't conclude and thus falls back to traditional TCP.
The characteristic of MultiPath TCP is to have multiple subflows. After the initialization, there is only a single subflow between the server and the client. There are several challenges in adding new subflows. MultiPath TCP makes a distinction between the client and the server. Only the client can add more subflows to the connection. The new subflow can not be easily started over a different path, since NATs and Firewalls usually drop data packets if they are not following a SYN [RPB+12]. Rather they need to be initialized again. For authenticity reasons a key, that is agreed upon between the two parties, is set in the *MP_CAPABLE* option. Also, another option, *MP_JOIN*, is set in the three-way handshake. This option contains a hash of the key to be able to identify and match the subflow to the existing connection. If the client is multi-homed, it can just add subflows this way for every interface it has. But the server can also be multi-homed. Since a server can not initiate new subflows, it advertises additional IP addresses it supports by setting the *MP_ADD_ADDR* option with these IP addresses. The client can then initiate a new subflow to these interfaces.
Traditional TCP uses sequence numbers to provide reliable in-order delivery. In MultiPath TCP two kinds of sequence numbers are used, the "normal" TCP sequence numbers for individual subflows as well as a data sequence number which is for the whole connection. Also, there is a mapping between the data sequence number and the individual sequence numbers to correctly identify losses. Accordingly, there are two kinds of acknowledgements, the "normal" *ACK* for the individual subflows as well as the *DATA_ACK* for the connection. If there is a lost segment, there are multiple ways to retransmit. Either it can be retransmitted by the same subflow as in traditional TCP or, newly possible with MultiPath TCP, over another subflow. In the Linux implementation the fast retransmit is done via the same subflow, but upon a timeout expiration the segment is free to be able to be sent via another subflow. If a subflow breaks down and still has unacknowledged segments, they all need to be sent via another subflow, since one can not be sure if they arrived.
There is a distinction between a connection shutdown and a subflow shutdown. Like with the acknowledgements, there are two kinds of flags for a connection tear down. The "normal" *FIN* for the individual subflows, as well as *DATA_FIN* for the connection. If a subflow sends a *FIN*, only this subflow would be closed with the connection still intact. If *DATA_FIN* is sent, the whole connection would be closed. Further, MultiPath TCP provides the message *REMOVE_ADDR*, which is the counterpart of the prior mentioned *MP_ADD_ADDR*. It would remove an IP address from the pool of available IP addresses on which subflows can be initiated to indicate that no new subflows can be initiated with that IP address. This is important for mobility. For example, if a mobile device moved out of the range of a Wi-Fi network, it is not reachable over that interface with the prior IP address anymore and it would send the *REMOVE_ADDR* message.

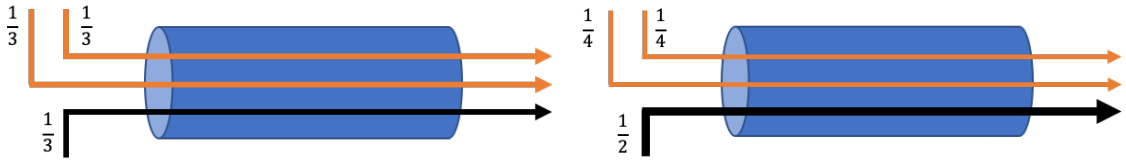
2.1.2 Path-Manager

The path-manager answers the question: Between which interfaces and how many subflows should be established? There are multiple ways to define a path-manager, but currently the Linux implementation supports the following four:

- "*default*": The default path-manager only accepts passive creation of new subflows. It will not establish new subflows on its own.
- "*fullmesh*": The fullmesh path-manager creates, as the name implies, a full mesh between all available interfaces.
- "*ndiffports*": The ndiffports path-manager creates a specified number of flows for one interface pair. Instead of alternating the IP-addresses, this path-manager only alters the ports.
- "*binder*": The binder path-manager [BFM13] uses Loose Source Routing [AC04]. It is a specific use case, where applications would get a benefit from gateway aggregation.

2.1.3 Congestion Control

Congestion control in MultiPath TCP has the same target as in traditional TCP, to avoid congestion. The simplest way to incorporate congestion control is to handle each subflow as an isolated TCP flow with its own congestion control. This is called "uncoupled congestion control" and can be seen as an unfair way, especially at shared bottlenecks. However, congestion control can be applied on a connection level, i.e. congestion control of the subflows is coupled together. This is called "coupled congestion control" and comes with three objectives that should be satisfied [RHW11]:



(a) Uncoupled Congestion Control **(b)** Coupled Congestion Control

Figure 2.2: Difference between Uncoupled Congestion Control and Coupled Congestion Control

1. Throughput should be improved. The MultiPath TCP connection should perform at least as good as a traditional TCP connection over the best path.
2. A MultiPath TCP connection should only use as much capacity as a traditional TCP connection over the same path.
3. The congestion should be balanced by MultiPath TCP taking a path that is congested less.

In Figure 2.2 it is visualized what the different kinds of congestion control approaches result in, when a MultiPath TCP connection with two subflows and a traditional TCP connection share a bottleneck. In the uncoupled case, every flow gets about the same bandwidth, which means that the MultiPath TCP connection gets about 2/3 of the bandwidth. In the coupled case, the MultiPath TCP connection shares the bandwidth fairly half-half with the other TCP connection, i.e. every subflow of the MultiPath TCP connection gets 1/4 of the bandwidth.

2.1.4 Scheduler

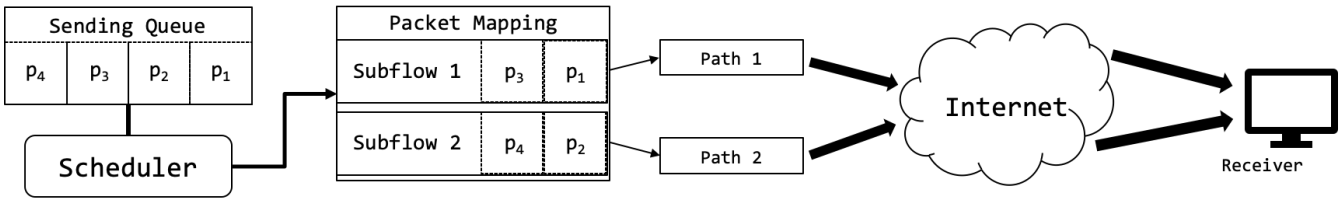


Figure 2.3: Multipath TCP Scheduling

The scheduler coordinates which packet is sent over which subflow. A naive way to implement a scheduler is to just spread the packets over all subflows. This is the Round-Robin scheduler and it iterates through a list of the available subflows and assigns a packet to each of them. It works well, as long as the subflows are homogeneous. Once there are heterogeneous subflows, which is the common case in real networks, more sophisticated schedulers are needed. The "default" scheduler of the Linux implementation is "MinRtt". It saturates the subflow with the lowest RTT. Once this subflow is saturated, the subflow with the next-lowest RTT is used. Further, there is a "redundant" scheduler. Rather than "MinRTT" it focuses on improving the reliability of the connection. It sends every packet over all available subflows, thus trading bandwidth for low latency and reliability.
The mentioned schedulers are the available schedulers in the Linux implementation of MultiPath TCP.

2.2 Neuroevolution of Augmenting Topologies - NEAT

Neuroevolution of Augmenting Topologies [SM02] or NEAT in short, is a genetic algorithm that mimics nature. With the premise that evolution works towards a certain goal, the survival of a species, NEAT evolves neural networks towards a given goal. Artificial neural networks are another concept that is based on nature, namely the neural circuit which is when several neurons are connected to each other. In nature, there are three different kinds of neurons: sensory neurons can be found especially in sensory organs and, as their name implies, sense or react to a stimulus and pass this reaction further on to the inter neurons. Inter neurons do not react directly to the "outside world" as they are only connectors, or relays, for sensory neurons and motor neurons. Finally, motor neurons are the opposite of sensory neurons. They are reacting to a signal from other neurons and induce motory actions of, for example, muscles.
Artificial neural networks, we refer to them as neural networks in this work, build on this concept. Neurons are represented as artificial neurons according to the McCulloch-Pitts neuron model visualized in Figure 2.4.
The model accounts for a weighted input, that is summed up and put into an activation function. The activation function

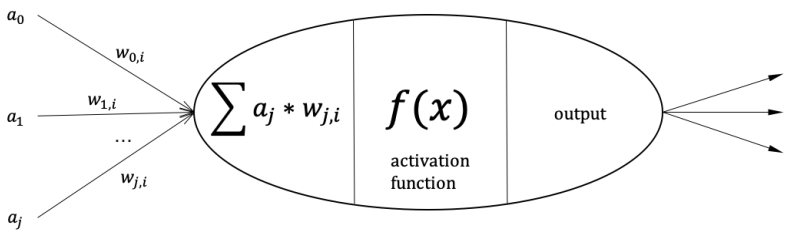


Figure 2.4: McCulloch-Pitts Neuron Model [MP90]

mimics the behavior of a neuron, where neurons would 'fire' (pass on an electrical impulse) depending on the impulses it is fed. The output of the activation function is passed on. There are again three different kinds of artificial neurons, which we call nodes. An input node serves the purpose of a sensory neuron, where it would get values as an input, for the neural network. Output nodes mimic the motor neurons in that they output a value. In between input and output nodes there are the hidden nodes, which get a weighted input from other nodes within the neural network and pass on their output to other nodes. A sample neural network can be seen in Figure 2.5. The different kinds of neurons can be seen at their position in the network. At the beginning, there are the input nodes, which are fed various values. These are passed on to the hidden layer, where the hidden nodes are located. Finally, the output nodes give the output values of the whole neural network. The purpose of neural networks is, that they can represent every possible mathematical function, given enough hidden nodes [HSW89].

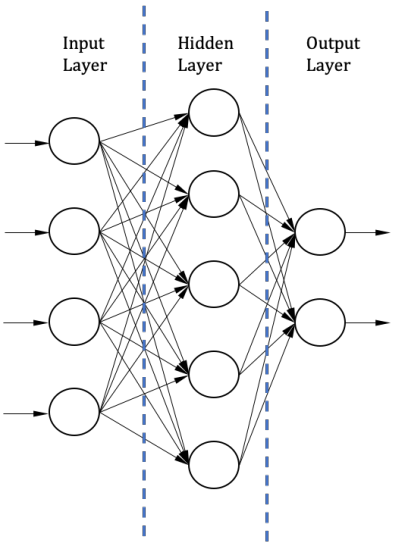


Figure 2.5: Sample Fully-Connected Neural Network

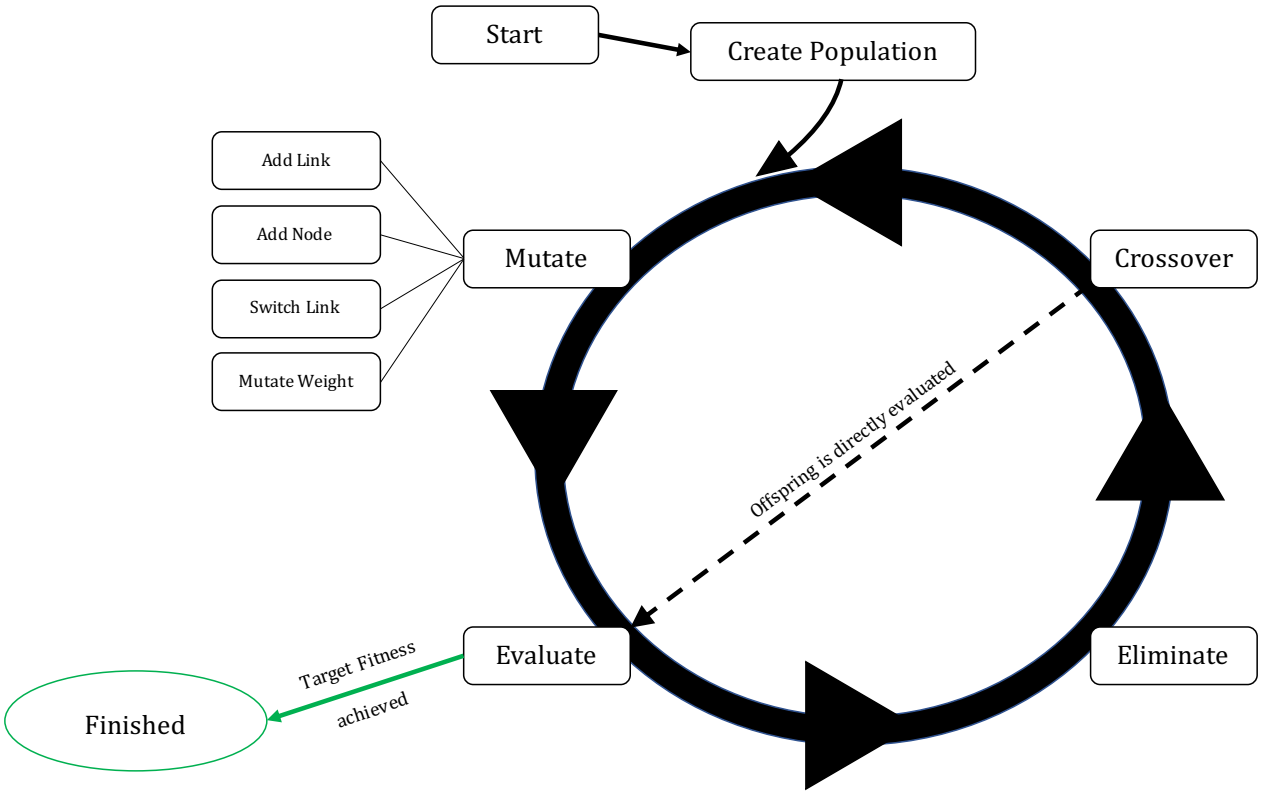
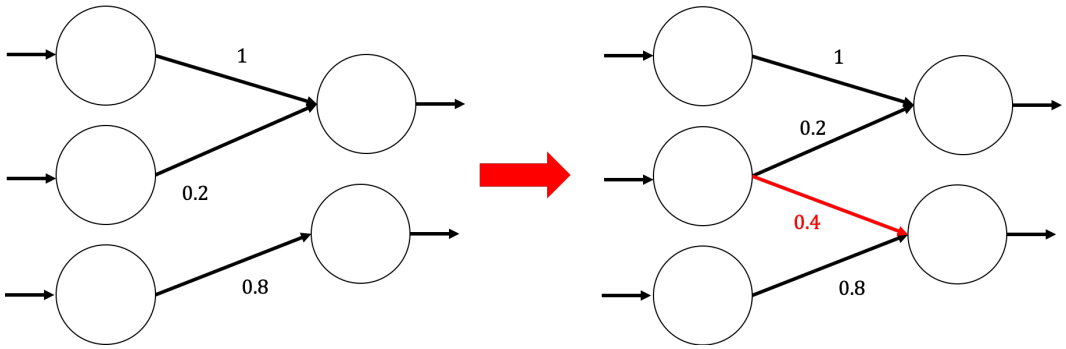


Figure 2.6: Overview of NEAT's Learning Process

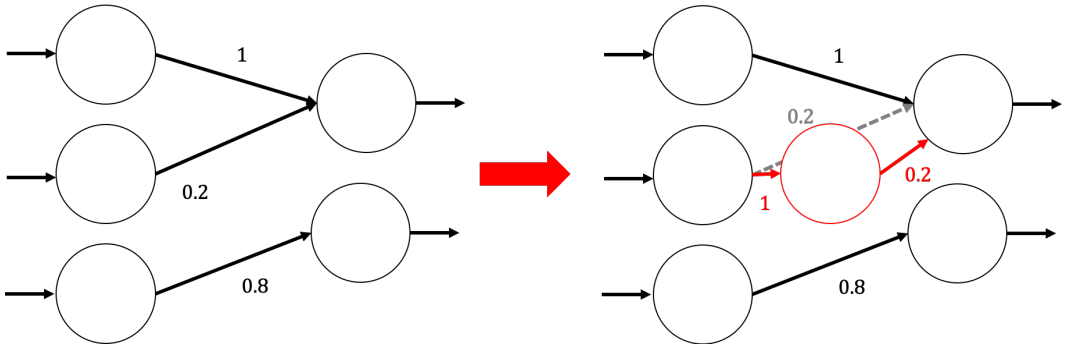
NEAT evolves neural networks by mimicking evolution in an iterative way by implementing the concepts of selection, mutation and reproduction found in nature. A visualization of NEAT's learning process is shown in Figure 2.6. NEAT needs a start topology, usually an empty neural network where only the input and output nodes are defined. The internal structure, i.e. the hidden layer, is created, or evolved, by NEAT's learning process.
NEAT starts by creating a population out of the start topology by mutating it randomly. There are four different kinds of mutations, visualized in Figure 2.7:

- *Add Link*: In this mutation a new connection between two previously unconnected nodes is created.
- *Add Node*: This mutation adds structure to the hidden layer. A new node is placed in between two connected nodes. Two new connections are created, so that the new node is connected to the nodes it is placed in between. Also, the link that connected these two nodes is disabled.
- *Mutate Weight*: In this mutation the weight of a link is changed randomly.
- *Switch Link*: This mutation can resolve into two results depending on the status of the link it mutates. If the link is active, the link is disabled. If the link is disabled, it is activated. A disabled link would be ignored by the neural network.

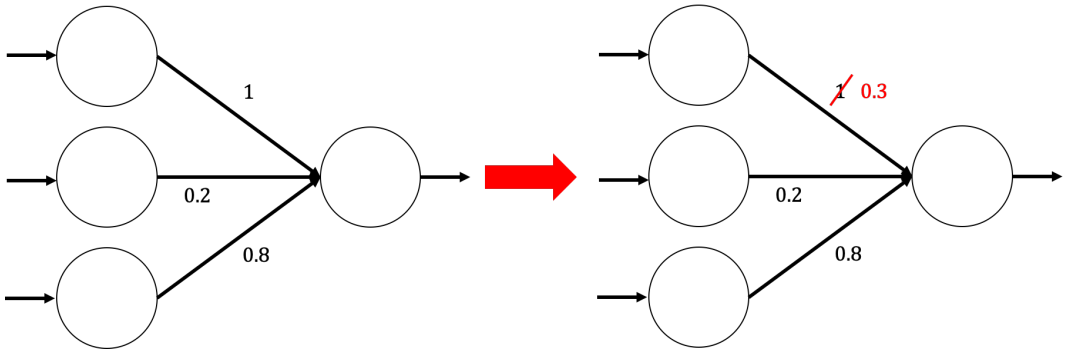
After mutating the population, it gets evaluated according to the fitness function. The fitness function represents feedback of the problem that needs to be solved. It shows how well the neural network solves the problem. Every neural network in the population gets evaluated and the achieved fitness is attached to it. Then, NEAT checks if the target fitness has been achieved by a neural network, i.e. a neural network can solve the problem satisfactorily. If there is such a neural network, NEAT's process stops. If not, the worst performing members of each species are eliminated.
Species are unique to NEAT and usually not existent in genetic algorithms and are implemented to further mimic nature, where different species, tribes, niches exist. In the context of machine learning this means that different approaches, that are not performing well yet, are not eliminated prematurely. This prevents NEAT to evolve in the direction of local maxima and rather gives the possibility to learn in different directions at the same time. To implement species NEAT keeps track of every mutation that happened, saved in genes, and if neural networks underwent similar mutations, they fall into the same species. Further, NEAT makes a distinction of the neural network itself and its genes. The neural network is called the phenotype. It is the neural network that is usable and no more information is saved there. Then, there is the genotype. There, the neural network is saved as a representation of genes and innovation. This representation is not directly usable and only important for the evolution part of NEAT. Out of the genotype, the phenotype can be created, but not vice versa. In terms of a learning algorithm, a species can be seen as an approach or model to solve the problem.



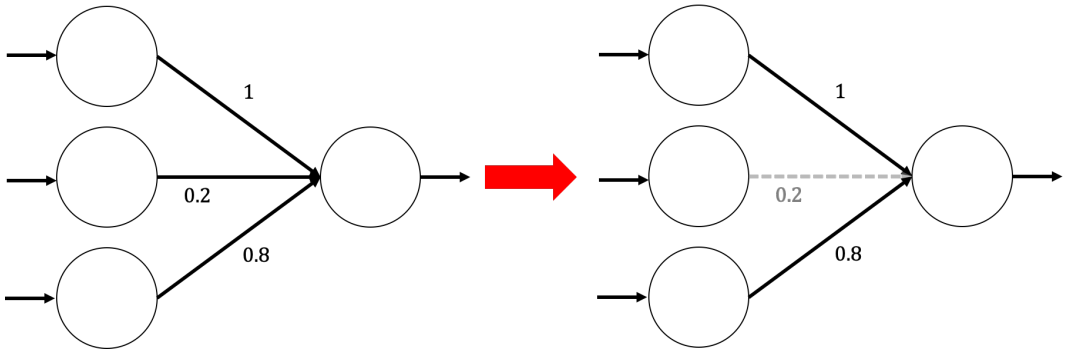
(a) Add Link Mutation



(b) Add Node Mutation



(c) Mutate Weight Mutation



(d) Switch Link Mutation

Figure 2.7: Mutations in NEAT

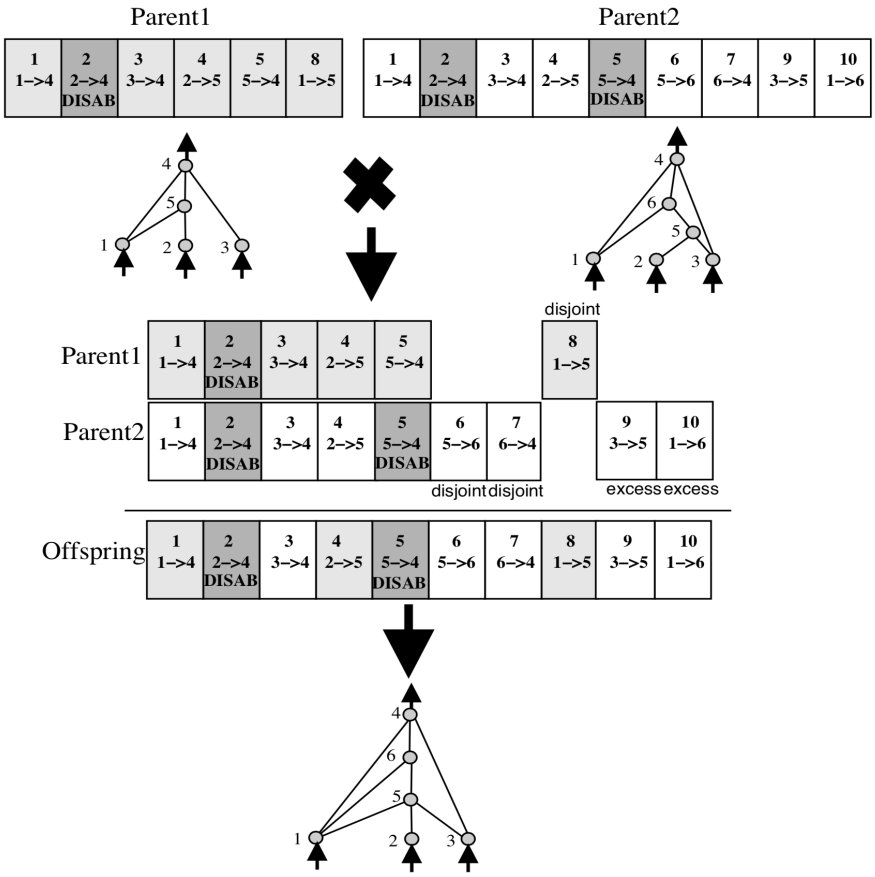


Figure 2.8: Crossover done by NEAT. All excess and disjoint genes are incorporated by the offspring, because in this example both parents have the same fitness.[SM02]

After the worst performing neural networks have been eliminated from the population there is room for new neural networks. Note that the number of neural networks before and after an iteration is constant. Here, the last concept of evolution is implemented, reproduction or crossover, to fill the space the eliminated neural networks left. To create a new neural network out of two neural networks both are aligned and similarities and differences are made out. As seen in Figure 2.8, both parents are aligned according to their innovation number. The innovation number indicates the time point where the mutation happened and shows the history of mutations. There are three different ways how genes can be labeled. There are the same genes, i.e. same innovation number and existent in both parents. Disjoint genes, where the gene with this innovation number is existent in only one parent, but the maximum innovation number of both parents is higher. And finally, there are excess genes that have a higher innovation number than the maximum innovation number of the other parent. Depending on this labeling the offspring is created. The same genes are taken at random. As for the disjoint and excess genes, only the ones from the fitter parent are adopted by the offspring. The new offspring is not directly mutated again by the next iteration of NEAT, rather it is directly evaluated.
After creating offspring until the population size is complete again, one iteration of NEAT's learning algorithm is over.

2.3 ProgMP

ProgMp [FRE+17] provides a programming model to specify and deploy schedulers directly to the linux kernel.
It closes the gap of the user space and the socket implementation of MultiPath TCP with an API. A user can specify a scheduler in the provided ProgMP language and directly deploy and use it on the socket. Further, ProgMP provides a runtime environment with an interpreter for the deployed schedulers. ProgMP is implemented by altering the MultiPath TCP implementation on kernel version 4.1.20, which is provided on their website.
In regards to the programming model, ProgMP provides a high-level language for specifying schedulers. Multiple functions for subflow filtering, like `MIN`, `MAX` or `FILTER`, are also provided. Further, there are built-in variables for sending queues, subflows and current time. ProgMP also allows to declare variables. However, these are single assignable, so once they are declared, they can not be changed. These get assigned for each scheduler invocation, which happens every time there is a new packet in the sending queue, an acknowledgement is received or there is a timeout. However, the number of variables is also restricted. There can only be 24 variables in a scheduler.
A sample scheduler implementation of the MinRTT scheduler is shown in the appendix, Listing 8.3. In this implementation the subflow candidates are found with the *FILTER* function. The next block stops the scheduler execution if there are no subflows to assign the packets to. After that, there is the retransmission block, which handles the retransmissions. Then, the logic of the MinRTT scheduler is described. The list of subflows candidates are filtered, so that only the subflows are checked for minimum RTT that have space for the packet.
Schedulers are saved in *.progmp* files which can be deployed via multiple means, either via the API in Python or C language or directly via the terminal.

There are 6 registers for each scheduler instance, which are constant between the scheduler executions and are set with extra functions. Also, it is possible to use standard arithmetic operators and comparisons, but one can only use unsigned integers for values. The programming language provided by ProgMP is rather restricted. Even though with this language it is easy to handle subflows and queues of a flow by providing a lot of helpful functions like the built-in filter, control flows are rather hard to implement. For this scenario one can use the prior mentioned registers. Registers are freely set and readable within an execution and are persistent in between executions. So for a simple if/else flow, where a variable is to be set depending on a result, one needs to detour with registers, like shown in Listing 2.1. In this listing it is checked whether subflow 0 or 1 has the lowest RTT. Depending on the result the variable *lowerRtt* is set.

```
VAR sub0Rtt = SUBFLOWS.GET(0).RTT;
VAR sub1Rtt = SUBFLOWS.GET(1).RTT;
IF (sub0Rtt > sub1Rtt) {
                SET(R3,1);
} ELSE {
                SET(R3, 0);
}
VAR lowerRtt = R3;
```

Listing 2.1: ProgMp If/Else Controlflow

3 Related Work

This chapter gives an overview about the state-of-the-art in MultiPath TCP scheduling, as well as an overview on NEAT.

3.1 Scheduling

There is a vast amount of different scheduling algorithms. Scheduling is significant for the performance of MultiPath TCP, especially if there are heterogeneous paths. We classify schedulers in two categories: static and dynamic schedulers. In static scheduling the scheduling decisions depend on human-understandable rules, unlike dynamic scheduling where the decision making changes within operation. As for static scheduling, there is a sheer endless amount of schedulers. Here we look at some static schedulers that differ in their targets.

As MultiPath TCP is also deployed in mobile devices, where energy consumption is a problem, there is a proposal of a scheduler that aims to reduce the energy consumption [MM17]. For example, if there are two interfaces, Wi-Fi and LTE, this scheduler takes into account how much energy the transmission would consume and makes the decision based on that. It builds on top of the MinRTT scheduler and optimizes the trade-off between performance and energy consumption.
Other schedulers, like proposed by [GNM+17], reduce the data chunk download time. This is achieved by minimizing subflow completion time, which is especially useful for websites that use multiple flows for parts of the website.
There is also a scheduler for video streaming. [CAPK+16] proposed a scheduler that aims to improve the QoE of video streaming over MultiPath TCP. It is a content-, application- and network-aware scheduling concept. This scheduler has knowledge about the data to be sent and the video stream that is arriving at the destination. It estimates for each video unit if it arrives at the destination in time, i.e. before it would be played. If it would not arrive in time, it is dropped and the next video unit is scheduled, resulting in a lower quality of the video stream, but consistent playback.
OTIAS [YWA14] is a scheduling concept where decisions are made according to the shortest time to arrive. Hereby, OTIAS lowers the possibility of out-of-order delivery of packets, improving the reliability of the connection. Because more in-order deliveries conclude, it also improves needed receive buffer space.
Other schedulers are closer to the "default" MinRTT scheduler, in that they take the concept, but improve on it. For example, ECF [LNTG17] does this. Instead of just taking the second lowest RTT path, like MinRTT, it calculates the arrival time. Only if the arrival time of the second lowest RTT path is lower than two times the lowest RTT, the second path is chosen. ECF improves upon the "default" scheduler as it prevents "bad" decisions of taking a path that will harm performance by introducing the "wait" mechanism. Instead of taking the "bad" path, like MinRTT is forced to, ECF waits for the "good" path to be available again.
BLEST [FAMB16] aims to reduce the buffer blocking, which happens because of heterogenous paths. It estimates the resulting buffer blocking that could occur with each transmission. Like the "default" scheduler, at first the lowest RTT path is fully saturated. Once this path blocks, it is calculated if a packet should be sent via the next lowest RTT path.
A scheduler that is similar to OTIAS and BLEST is STTF [HGB+19]. It also allows for packets to be assigned to paths whose congestion window is full, whereas before the scheduler would have "blocked" until a good decision is taken. It calculates the transfer time for all packets and also takes more subflow characteristics into account.
Most of these schedulers assume that all subflows are in congestion avoidance, but STTF also takes the TCP state into account in its calculation of the transmission time. Both, BLEST and STTF focus on improving the page load time of websites, whereas ECF and OTIAS improve general MultiPath TCP operation. There are also schedulers that are focusing on a special problem like video streaming.

Another way to implement schedulers is by using machine learning. With the recent rise of deep learning, there are approaches to improve scheduling by learning. These approaches are using online learning, so they learn within operation. We now look at some deep learning based schedulers. A Deep Q Network is usually used in combination with reinforcement learning [ZLG+19][XTY+19][LSL19]. With this approach, decisions are saved in a memory, so the deep neural network does not need to be evaluated again for the same inputs. While these Deep Q Network based approaches are similar to each other, there are some significant differences.
An approach proposed by [LSL19] uses online learning, meaning it is learning while in operation for every packet sent. As for inputs, this model only takes the sending windows of the paths into account. The authors did not specify the size of their deep neural network, but it is to be expected that this poses a huge overhead.
ReLes [ZLG+19] on the other hand uses asynchronous learning. Instead of learning directly on the network conditions, the learning is offloaded. Also, it uses more inputs, namely throughput, average congestion window size, mean RTT, number of unacked packets and retransmitted packets. The learning itself is done offline with the network traces and the scheduler is updated regularly. However, this approach, too, uses a lot of resources.

But, there are not only deep neural network based approaches. [JHKC17] for example, uses random decision forests. The target of this approach is the optimization of scheduling while using Wi-Fi and LTE, thus using specific inputs like RSSI for its decisions.

3.2 NEAT

A promising approach to learn neural networks with minimal structure is NEAT [SM02]. NEAT is explained in detail in section 2.2. NEAT has been proven to work in various fields, from balancing a pole on a two-dimensional wagon up to playing video games like super mario [Set].
There exists some work on NEAT in communication networks. NEAT-TCP [WKAH20] uses NEAT to evolve neural networks that conduct congestion control. It proofs to work in a frontier where traditional TCP congestion control fails to perform, in wireless multi-hop networks. In a very contested topology, NEAT-TCP accomplished to optimize fairness between all flows in the network while also improving delay at the cost of some throughput. This work proves that multi-criteria optimization for networks is possible by using NEAT. Each node in the wireless multi-hop network uses an instance of the same neural network, so this approach evolves neural networks that distributively solve the problem of congestion. The evolved neural networks use locally available network measures, that get mapped to a decision on the congestion window, whether to increase or to decrease it. In our case, we want to create a scheduler that takes decisions onto which path to send a packet. Thus in this work, as the use-case of NEAT-TCP is similar to our problem, we use an approach that is inspired by NEAT-TCP. We take the approach of NEAT-TCP and build our design on top of it.
However, there are many kinds of NEAT. NEAT-TCP uses the standard model of NEAT that is proposed in its first form in [SM02]. There are a various extensions to NEAT that let it take on more kinds of problems. To name a few: From the same authors of NEAT, there is rtNEAT [SBM05], which changes NEAT in a way that, instead of using the iterative manner of NEAT, the evolution is done in real time. Each of the organisms have a timer and are evaluated once the timer expires. Another extension is HyperNEAT [HLMS14], which allows for multi agent learning and much larger neural networks to be evolved. There is also DeepNEAT [RM16], which allows for the use of deep neural networks.

3.3 Summary

There are a lot of different approaches to implement scheduling in MultiPath TCP. Static schedulers are not performing optimally in every network topology. Rather, they are created with a particular aim to solve the scheduling problem for particular use cases. For a more general approach, scheduling with machine learning is useful. It optimizes on the current network topology and use case, but the ones that are currently available need to relearn once the network topology changes. Also, they usually coincide with a lot of overhead, because they are based on deep neural networks. While there are also machine learning based approaches that don't use deep neural networks, these also learn on the current network topology. However, since they learn on the current topology, once the network changes, it needs to be relearned. In this work, we try to close the gap between static scheduling and machine learning based scheduling. We combine the ideas of both, but rather learn to use the optimal scheduler for the current network topology. As NEAT has been proven to be capable of learning algorithms and behavior, we use NEAT as our approach to offline learn a discriminator that is able to choose the optimal scheduler, while still being minimal in size and overhead. Because there is already an implementation where NEAT learns congestion control algorithms, we build on top of this implementation.

4 Design

As stated in section 1.2, we want to create a meta scheduler that builds on top of static schedulers. Depending on the current network topology it chooses the static scheduler that performs best. According to the chosen static scheduler the current packet on top of the queue is assigned to a subflow. The meta scheduler makes the decision on which scheduler to use depending on locally available network measures. The decision making itself is done with a neural network. Neural networks have a high tolerance to noisy data and are capable to correctly handle unseen events. As communication networks have a big event space, that is, there are a lot of different network states that could occur, it makes sense to use neural networks. It is of minimal size and introduces low overhead. The neural network maps the network measures to the static schedulers and thus decides which scheduler to use. It is learned by simulating the environment the meta scheduler will be deployed in. Rather than learning online in the environment, it learns offline in simulations to reduce the overhead when in operation. With the learned mapping, there is a neural network that is represented as small amount of simple mathematical equations that can be used in ProgMP. Thus, the meta scheduler that is learned can be translated to the ProgMP language and deployed to the Linux kernel.

The overview of the process from learning to deployment of a meta scheduler is depicted in Figure 4.1. Before a meta scheduler can be generated, the environment it will be deployed in has to be analyzed. This is because it is generated in simulations, therefore the environment needs to be rebuilt for the simulator. Because an environment is rarely constant, there are multiple forms of how the environment can be modelled. We call these forms scenarios. For example, if an environment had two possibilities for delay of a link, say 20 ms and 10 ms, there would be two scenarios that can occur. Of course, in a real environment there are endless scenarios that only differ slightly. Because it would be exhausting and unnecessary to model all of them one by one, we only model representatives of a set of scenarios we call core scenarios. For example, if the delay of a link of an environment can be between 10 ms and 30 ms, and would be on 20 ms in the average case, then we would have three core scenarios. All possible core scenarios need to be built in ns-3 and the best performing scheduler for each of them needs to be found. With this set of schedulers and core scenarios, a meta scheduler is learned with NEAT. The resulting neural network is in NEAT's neural network specification language. It is then translated into the meta scheduler in ProgMP language. The result is a meta scheduler that can easily be deployed to the Linux kernel with ProgMP. In the following each step is explained in detail.

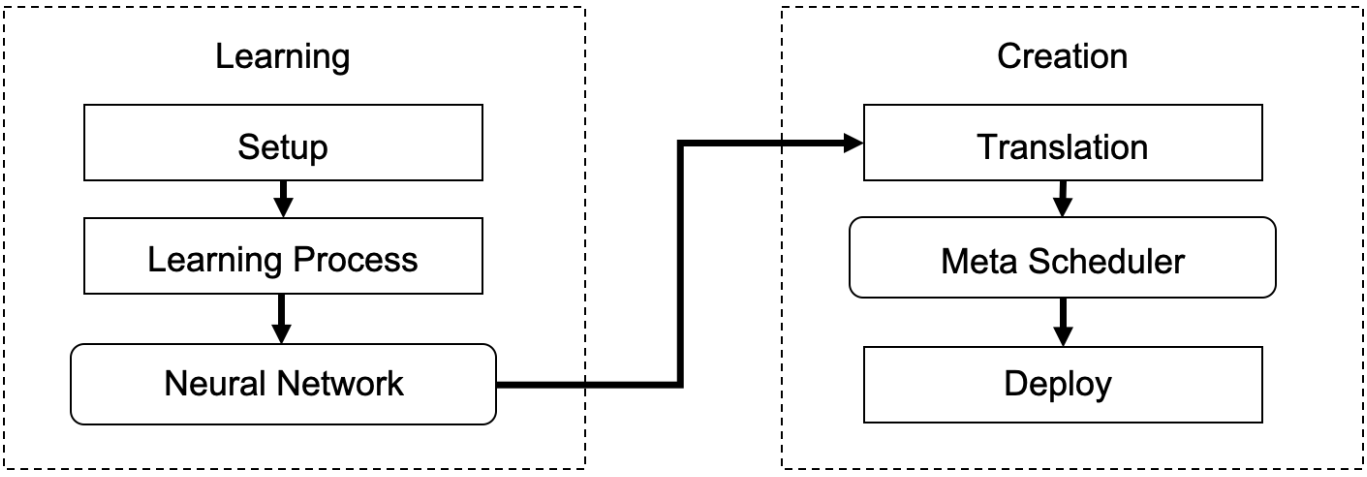


Figure 4.1: Overview of the Process from Learning to Deployment.

4.1 Learning a Meta Scheduler

4.1.1 Learning

In this section the first three steps of Figure 4.1 are explained. The preparations to be able to start learning a meta scheduler are explained in detail.

Setup

Before the learning can start the environment where the meta scheduler will be deployed must be thoroughly analyzed. In an analysis the following example questions should be answered:

- What are my network interfaces and how do they behave?
- How stable are the connections?
- What is the loss rate?
- What is the traffic shape?
- What are the usual network measures I can expect in the environment?

This list should give an insight into what has to be done in an analysis. The target of it is, to be able to create core scenarios, that realistically portray the environment. The closer these scenarios are to the real world, the better will be the performance of the meta scheduler. Also, the aim of the meta scheduler needs to be defined. Which measure should be optimized? Need my applications high throughput or as low of a delay as possible? Depending on the optimization target the core scenarios need to evaluated. Which static schedulers are usable and would perform best according to the optimization target needs to be found. The meta scheduler should perform like the best performing static scheduler for each scenario. As it learns to schedule schedulers and not scheduling itself, it should not improve performance for a single scenario. Rather, it improves the performance for a group of scenarios by mapping the single scenarios to the best performing scheduler.
These scenarios need to be built in ns-3, as the learning process is based on simulations, not the real world. This is because the learning process itself is time consuming and the more time can be saved, the better. That is why we use ns-3, as it is an event based simulator and its simulations don't run in real time. This shortens the time needed for the learning process. In order to evaluate the generated meta schedulers each of them needs to be put in every scenario. Hence, every core scenario needs to be built in ns-3.
The meta scheduler uses a neural network of which the output layer signals which static scheduler to use. The output layer is not predefined as the number of static schedulers which will be deployed is not fixed. Rather, the number of nodes in the output layer needs to be specified according to the number of static schedulers that will be used by the meta scheduler.
Depending on the measure the user wants to optimize for the fitness function needs to be specified. NEAT needs to be able to attach a single value to a neural network in order to know its performance. In this case, how the meta scheduler is able to choose the best scheduler in each scenario. This is done by the fitness function. To be able to calculate the fitness function, there needs to be another function that reflects the performance that is achieved in a scenario. The used static schedulers also need to be evaluated with this function for every core scenario. With the knowledge about the performance of the best static scheduler, the performance of the meta scheduler can be put into perspective. It should be able to reach the same performance as the best static scheduler, given it chose correctly. For the fitness function it should be 1 (or 100%) if the meta scheduler chose correctly for every scenario.
In summary, the following steps are necessary before the learning itself can start:

1. Choose the optimization target. Create a function that reflects performance according to the optimization target.
2. Build possible scenarios in ns-3. The number of scenarios should be as small as possible while representing the core scenarios satisfactorily to save time.
3. Find the best performing static scheduler for each core scenario according to the optimization target set in step 1 and find out the performance of that scheduler.
4. Build the output layer of the neural network according to the schedulers found in step 3.
5. Create the fitness function. It should be 1 (or 100%) if the meta scheduler performs like the best static scheduler on all network topologies. A sample fitness function is given in Equation 4.1, where p_n is the fraction of performance achieved by the meta scheduler and the performance of the best static scheduler. p_n reflects how good the meta scheduler performs in relation to the best performing static scheduler that is in use. Hence, it is 1 if the meta scheduler is as good as the best static scheduler. Hence, the fitness function (ζ) is 1 if the meta scheduler performs as good as the best static scheduler on every scenario.

$$\zeta = \frac{\sum^{n} p_n}{n} \tag{4.1}$$

Learning Process

After the learning setup, as depicted in subsubsection 4.1.1, the learning itself can start. For the learning we use NEAT, further explained in section 2.2. We extend NEAT in the evaluation step, where the fitness value is attached to the neural networks of the population. In this step, each neural network is evaluated according to the fitness function in all scenarios found in subsubsection 4.1.1. This step takes the longest time in the learning process. The network simulations should be as short and minimal as possible to not make the learning process unnecessarily long. As every neural network needs to be evaluated in network simulations and the number of neural networks in the population can be very high, the time needed can become very long. Also, if the results of a network simulation vary depending on a random variable, there should be multiple simulations to cover multiple scenarios.
For example: We have a population of 200 neural networks and 3 scenarios. All of them have a certain loss function that is depending on a random distribution. For each of these scenarios we need to have multiple simulations, as we don't want a neural network to get "lucky" or "unlucky" and perform better or worse only because of the randomness of the loss function. If this was happening, we would get false results, because we may have found a neural network that would satisfy our target, but because of the loss function it performed worse than it would in the average case. Vice versa, we may happen to get a neural network that we labeled as optimal, but actually it is performing rather bad and just got "lucky" with the loss function. To counter this we use multiple simulations to get an average of the performance to not fall into the trap of outliers. In this example we use the average performance of 5 simulations of each scenario for the evaluation. The average time needed for the network simulations is about 5 seconds. So a single iteration, or one epoch, of the learning process needs about 200*5*5 = 5000 seconds, or about 1 hours and 23 minutes. If we can reduce the average simulation time by 1 second we can get a save up of about 16 minutes in each iteration.
The learning process itself is depicted in Figure 2.6. In this thesis, we follow the standard NEAT learning process and extend the evaluation part as depicted in Figure 4.2.

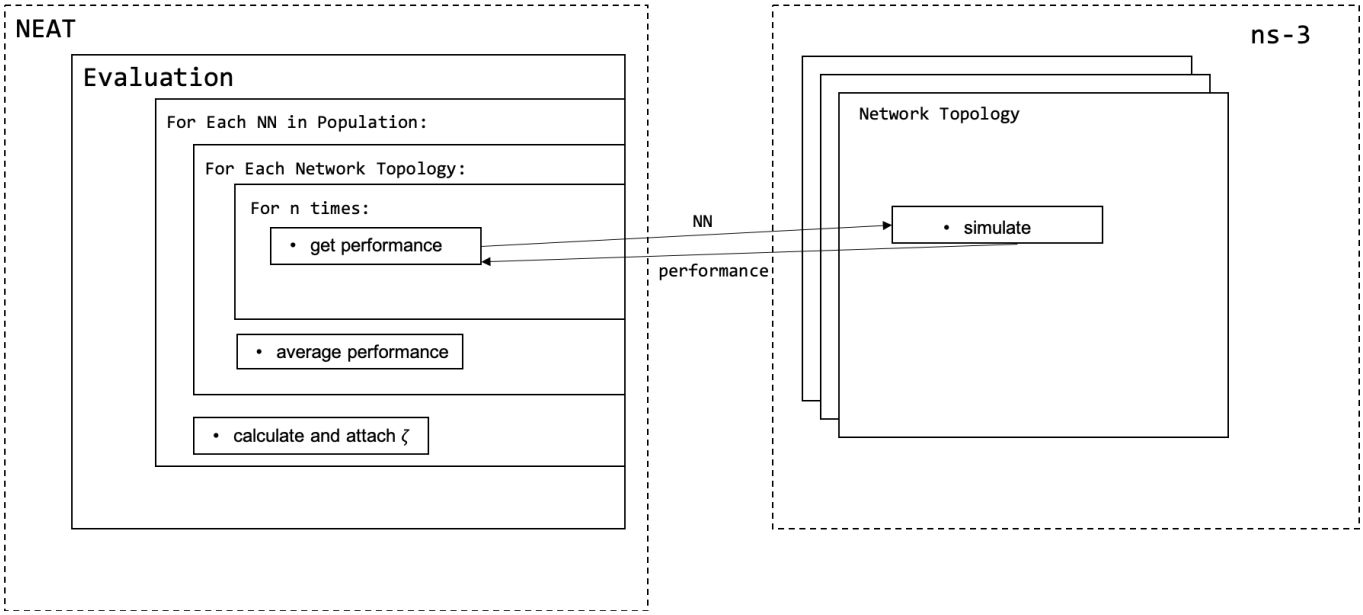


Figure 4.2: Extention to NEAT's Evaluation Step

The evaluation step is invoked once every epoch in NEAT. We need to attach a fitness value according to the fitness function to each neural network in the population. The fitness of each neural network is calculated according to the fitness function created in subsubsection 4.1.1. Since there are multiple simulations per scenario, we take the average of them for the calculation of the performance. These average values are then used in the calculation of the fitness. Once the target fitness is achieved, or in other words: the meta scheduler can correctly choose schedulers, the learning process stops and the satisfying neural networks are marked as "winner". In the next section these neural networks are further described.

Neural Network

The biggest part of the meta scheduler is the neural network. For the input values of the neural network we use the following measures:

- Round-Trip Time (RTT): It reflects the delay of a connection. RTT is calculated by tracking the time it takes to get an acknowledgement for a sent packet. This includes the time the packet and the acknowledgement needs to propagate.
- RTT Variance: Variance is a measure of the span values of a set can be spread from their average. It reflects how stable the RTT is.
- Congestion Window (CWND): The CWND reflects how saturated a link is, or how much can be sent on a link.
- The number of subflows

These measures give a good representation of the network. Keep in mind that we don't do scheduling itself, but rather want to choose a scheduler that performs optimal. So we need measures that broadly describe the network. Therefore we use relations of maximum and minimum of the available subflows, as it reflects the heterogeneity of the network. Those values are supported in ns-3 as well as in ProgMP, so this neural network can be easily ported between them. The supported measures of ns-3 and ProgMP are listed in Figure 4.6. For the activation function of the nodes we use ReLU, Equation 4.2.

$$ReLU(x) = max(0, x) \tag{4.2}$$

We can not use a more sophisticated activation function, like sigmoid, as these are not implementable in ProgMP. The output layer of the neural network has a node for every supported scheduler. The scheduler whose associated output node has the highest activation of the output nodes is chosen. This is why a simple step function would not be sophisticated enough, since we need one maximum activation in the output layer to know which scheduler to use. When using ReLU, the probability of having multiple outputs with exactly the same activation is rather low.
Only the input layer and the output layer are specified, as the internal structure is learned by NEAT. We use this neural network as the start neural network for the NEAT learning process. There are multiple ways to define the start neural network, but we found no significant differences in the learning process between them. However, NEAT needs at least one link between one input node and one output node defined. Also, a fully connected neural network can be used, i.e. all inputs are connected to all outputs. If the links of the start neural network all had a weight of 0, we found no significant differences in learning speed or ability to reach the learning goal.

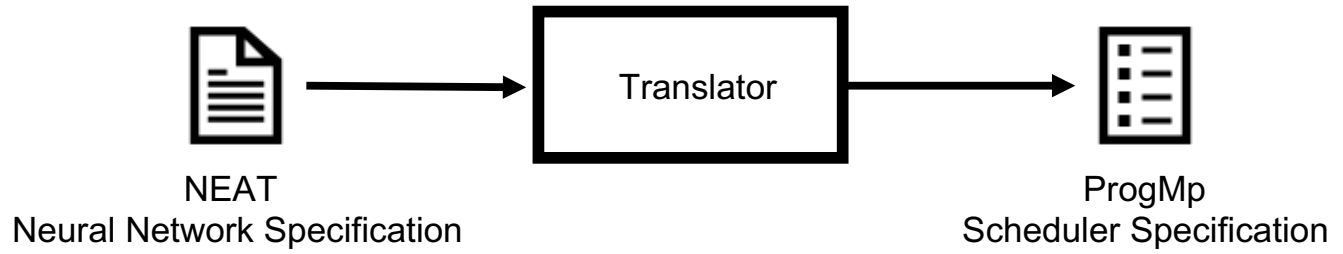


Figure 4.3: Translator

4.1.2 Creation

This section explains how to deploy the neural network that results from the learning process as a meta scheduler to the Linux kernel. We use ProgMP to deploy the meta scheduler, thus the neural network needs to be translated into a scheduler written in ProgMP language.

Translation

After the learning process generated a neural network that performs satisfactory, we have a specification of the neural network for the meta scheduler in NEAT's language. For deployment to the Linux kernel we use ProgMP, so we need to translate the neural network into a scheduler in ProgMP language. The translation is straight forward with the only problem being that the neural network can not be deployed as it is. Rather, it has to be written as a series of equations. This translation can be automized by the so called translator. The translator takes the specification of the neural network that NEAT generates and creates the scheduler in ProgMP language. In order to do so, the translator needs to know the input

values as well as the used static schedulers. These have to be specified within the translator, because the neural network itself has no information on them. The result of the translation process is a scheduler written in ProgMP language. This scheduler can be deployed with ProgMP directly to the kernel, either through console commands or within programs. The translator is further explained in section 5.4.

Meta Scheduler

The architecture of the meta scheduler is depicted in Figure 4.4. The meta scheduler takes the current network state and calculates the output in every invocation. Depending on the output layer the scheduler for that invocation is chosen. Since the activation function is ReLU (Equation 4.2), we can find the maximum of all nodes in the output layer. Because each node in the output layer is associated with one scheduler, the scheduler with the highest associated node activation is chosen and the packet for the current invocation is scheduled according to the chosen scheduler. The internal structure of the neural network is depending on the learning process and the environment the scheduler will be deployed in. However, because it is generated by NEAT, the structure is of minimal size and thus introduces minimal overhead. Also, the schedulers that can be chosen by the neural network are depending on the environment, like explained in subsubsection 4.1.1.

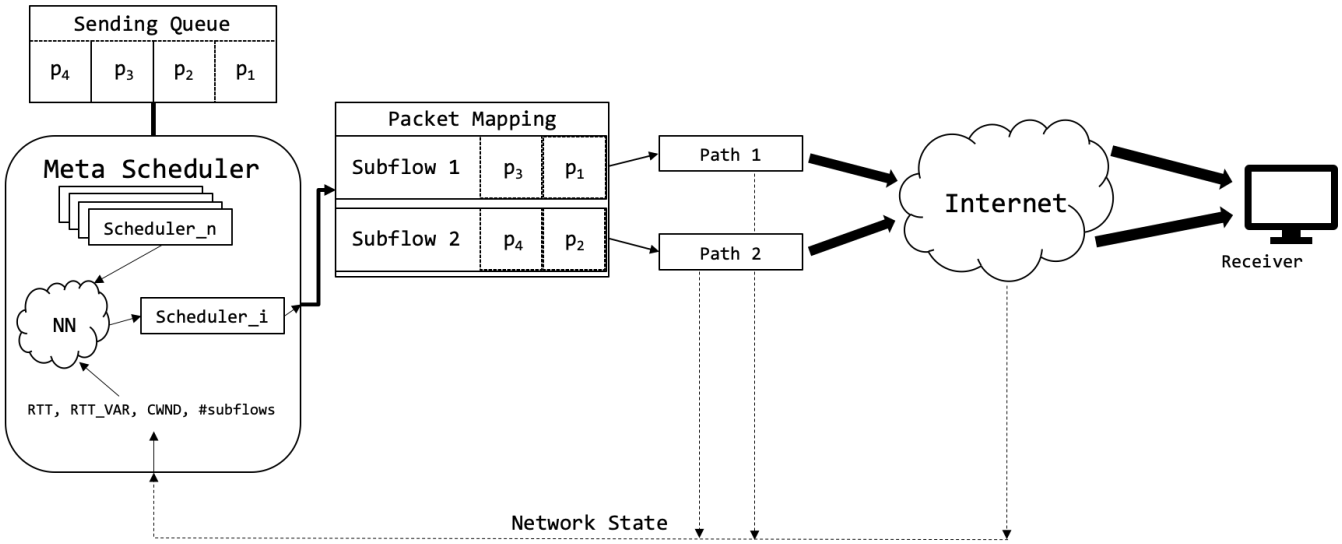


Figure 4.4: Meta Scheduler Structure

4.2 Learning a Static Scheduler

With the explained approach, static schedulers can be learned as well. The aim here is to create a scheduler that outperforms other schedulers on specific network topologies or scenarios. In order to generate static schedulers for specific network topologies, the priorly explained process only needs to be changed slightly. In the following, the needed changes are explained in detail.

4.2.1 Changes to the Learning Setup

In order to learn a static scheduler, only a single network scenario is needed. The network scenario is built in ns-3 and as close to the real world as possible. With static schedulers as well, the learning process is time consuming, but the time needed per iteration, or epoch, is shorter, because here only one network scenario is simulated instead of a set of core scenarios. Further, one does not have to research on other static schedulers, but in order to have knowledge about what to expect and if it can actually be improved, we advise to do so. There is no reason to learn a new scheduler if there is already one that is performing near optimum.
Again, we need to set the optimization target and come up with a fitness function. Here we might not know what the reachable optimum is, since we want to improve the current maximum performance of other schedulers. Hence, setting up the target fitness is more complicated. There are multiple ways to cope with this problem, but the best way is to set the theoretically possible performance as a fitness target. Most likely this target will not be achieved, especially if loss is involved. Rather, we would let NEAT run for a set number of iterations and take the best neural network at the end. We have

access to every neural network that is generated by NEAT, so intermediate results can be directly translated and deployed. Once NEAT generated a neural network that outperforms all other schedulers, we can directly use it, while continuing to try to improve upon it.

4.2.2 Changes to the Neural Network

The neural network itself needs to be changed. For inputs we now need to use measures of the subflows to be able to distinguish between them. This approach has more input nodes than the meta scheduler. We use four measures as input and one output node per subflow. The internal structure is again not specified and built by NEAT. Instead of choosing a scheduler to use in each invocation this neural network chooses the subflow to use. It takes the number of lost packets, RTT, RTT variance and the congestion window of each subflow and maps to subflows. We once again use ReLU, Equation 4.2, for the same reasons explained in subsubsection 4.1.1.

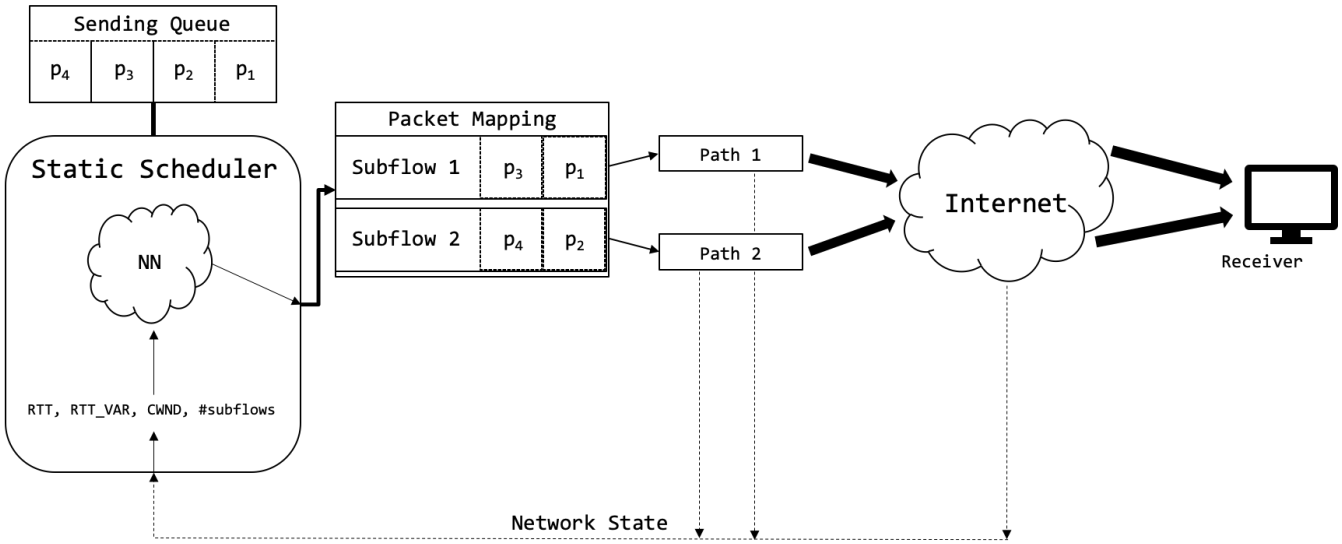


Figure 4.5: Static Neural Network Based Scheduler Structure

4.3 Fusion of Learned Static Schedulers and Meta Scheduler

We can now improve the performance of the meta scheduler. As the meta scheduler can only be as performant as the schedulers it can choose, in order to improve the performance of the meta scheduler we need to improve the performance of the schedulers it can choose. The performance can be improved while still introducing low overhead, because the most the meta scheduler has to do now is to solve two minimal structured neural networks. In this approach, we use the problem of overfitting as an advantage. Overfitting is a problem in machine learning approaches, where the model adapts too much to the training data and cannot work with unseen data anymore. Given that the network simulations are comparable with the real world scenario, this poses no problem in our approach. Since the meta scheduler chooses the scheduler that performs best for the given scenario, we actually want to overfit to a single scenario.

Parameter	ProgMP	ns-3
CWND	✓	✓
Lost Packets	✓	✓
RTT	✓	✓
RTT_VAR	✓	✓
Packets in Flight	✓	✗
Packets Queued	✓	✗
Remaining Connection Retries	✗	✓
Max Connection Retries	✗	✓
Connection Retry Timeout	✗	✓
Slow Start Threshold	✗	✓
Access to Sequence Numbers	✗	✓
Bandwidth	✗	✓
Congestion Control Variables	✗	✓
Duplicate ACK Counter	✗	✓
Number of Sent Packets	✗	✓

Figure 4.6: Supported Measures in ns-3 and ProgMP

5 Implementation

5.1 MultiPath TCP and Scheduling in ns-3

MultiPath TCP is officially not supported by ns-3. We use an implementation by Kheirkhah et al. [KWP15] which builds MultiPath TCP according to RFC 6824 [FRHB13] in ns-3.19. The structure of MultiPath TCP in this implementation is visualized in Figure 5.1. The classes in black colored boxes are from the official ns-3 implementation and the classes in red colored boxes are added by the MultiPath TCP implementation.
There are multiple applications provided in ns-3. However, the only two applications that support MultiPath TCP are the *BulkSendApplication* and *PacketSink*. They have an extra implementation in `mp-tcp-bulk-send-application` and `mp-tcp-packet-sink` in the *applications* folder. New implementations for applications are needed, because the applications would create a TCP or UDP socket and MultiPath TCP uses another socket that needs to be initialized differently. The new applications are basically copies of the old ones with different socket handling.

MultiPath TCP itself is implemented in `mptcp-socket-base` and extends `tcp-socket-base`, the implementation of basic TCP, with the interfaces for the upper layers. The `TcpL4Protocol` class is the interface for the lower network layer and only accounts for single-path connections, i.e. one connection as one IP end point. This class is extended to not only be able to identify connections by the TCP header's four tuple but also by the MPTCP unique identifiers.
Since `tcp-socket-base` implements `socket`, a dynamic cast can be used to create a MultiPath TCP socket:

```
m_socket = DynamicCast<MpTcpSocketBase>(Socket::CreateSocket (GetNode (), m_tid));
```

Listing 5.1: MultiPath TCP Socket Creation

In order to connect to the destination, an altered `connect` function is used. It creates a `mptcp-subflow` object, which is a striped down TCP connection, and connects it to the destination. The subflows do not implement any functions regarding TCP itself, rather, the whole logic is implemented in `mptcp-socket-base`. Congestion control, scheduling, receive/transmit logic, buffer handling, path-manager and window management are all done there and then mapped to the corresponding `mptcp-subflow` object. Scheduling is implemented in the `getSubflowToUse()` function of the `MpTcpSocketBase` class. It returns the ID of the subflow which should be used for the current packet. This function is called by the `SendPendingData()` function, which calls `SendDataPacket()`. `SendPendingData()` is called by the `SendBufferedData()` function, which is called by the applications once they filled the buffer. `SendDataPacket()` actually sends the packet over a subflow. Before sending a packet `SendPendingData()` calls the `getSubflowToUse()` function to receive the ID of the subflow that will be used. The implementation only comes with the *Round Robin* scheduler defined in a switch case, indicating that if new schedulers are added, they should be added here.

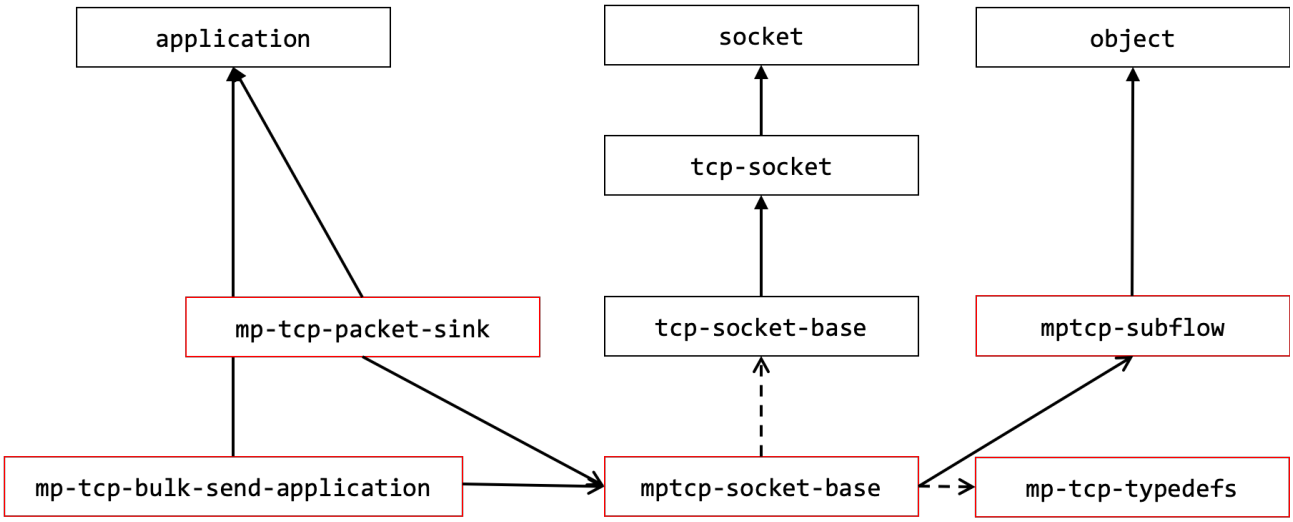


Figure 5.1: Inheritance Diagram of MultiPath TCP in ns-3

```
uint8_t nextSubFlow = 0;
  switch (distribAlgo)
    {
  case Round_Robin:
    nextSubFlow = (lastUsedsFlowIdx + 1) % subflows.size();
    break;
  default:
    break;
    }
  return nextSubFlow;
```

Listing 5.2: Scheduler Definition in ns-3

The switch switches over *distribAlgo*, which is an *enum* out of *DataDistribAlgo_t* defined in the *mp-tcp-typedefs.h* file. In the implementation where only *Round Robin* is defined, there also only exists *Round_Robin* as an *enum* of *DataDistribAlgo_t*. Here as well, if a new scheduler is defined, another *enum* needs to be added, so the `getSubflowToUse()` can have a case for it. The *enum*, or the scheduler, can be set by a ns-3 attribute.

```
.AddAttribute("SchedulingAlgorithm",
                    "Algorithm␣for␣data␣distribution␣between␣sub-flows",
          EnumValue(Round_Robin),
          MakeEnumAccessor(&MpTcpSocketBase::SetDataDistribAlgo),
          MakeEnumChecker(Round_Robin, "Round_Robin"))
```

Listing 5.3: Scheduler Attribute Definition in ns-3

This attribute allows the user to set the wanted scheduler when building a simulation without changing the simulator itself. When adding schedulers, they have to be defined in the *MakeEnumChecker* as well as in the *mp-tcp-typedefs.h* file. Note that *EnumValue* denotes the default case. In the shown listing, if nothing is set, *Round Robin* will be used. To change the scheduler, one can change it like any other attribute in ns-3. The easiest way is to use *Config::SetDefault*:

```
Config::SetDefault("ns3::MpTcpSocketBase::SchedulingAlgorithm",
                                                    StringValue("Round_Robin"));
```

Listing 5.4: Change Scheduler Attribute in ns-3

This is done in the *main* function of the simulation. The *StringValue* denotes the scheduler that the user wants to use and has to match with what is defined in the *MakeEnumChecker*. The *MakeEnumChecker* matches the *StringValue* that is set by the user with an *EnumValue* that is defined in the scheduler, so that it is capsuled away from each other. We created the *MinRTT* scheduler this way. First, we added an *enum* called `Min_Rtt` to *DataDistribAlgo_t* in *mp-tcp-typedefs.h*. Then we changed the *MakeEnumChecker* like in Listing 5.5. Afterwards, we extended the switch in `getSubflowToUse()` and implemented the logic of the scheduler as a case. This then returns the ID of the subflow to use according to the *MinRTT* scheduler.

```
MakeEnumChecker(Round_Robin, "Round_Robin",
                        Min_Rtt, "Min_Rtt"))
```

Listing 5.5: Addition of MinRTT

With that it is possible to choose the *MinRTT* scheduler in simulations by using the command shown in Listing 5.4 with *StringValue("Min_Rtt")*. This is the standard way of setting and changing attributes in ns-3 as the simulation does not need to import or define anything by itself, rather it just sets values that get matched to already implemented features.

5.2 Implementation of NEAT

For NEAT we use a C++ implementation available on GitHub by FernandoTorres [Fer], which closely follows the process of NEAT, as explained in section 2.2. We use this implementation, because it is tidier and more organized than the original NEAT implementation and since it is written in C++, we can easily incorporate it into ns-3. It uses two extra files as an input, the *startgenes* file and the *parameter* file. In the *startgenes* file the initial neural network is saved. It holds the *traits*, *nodes* and *genes*. A sample *startgenes* file can be found in the appendix in Listing 8.1. With this, the population is generated. The other file holds parameters for the learning process itself. A total of 33 parameters can be set. Most of the parameters that can be set are self-explanatory, which is why we refer to the appendix, Listing 8.2, where a sample parameter file can be found. Mostly, these are probabilities for which a mutation is performed and to what extend. For example, it can be set that a link is mutated by a probability of 20% and to a power of 2.0. This means that, when NEAT is in the mutation phase, each link of every neural network has a chance of 20% to be mutated by a value of up to 2.0. Also, the population size can

be set here as well as how often there should be a log file of the current population. In the log file every neural network in the population is printed in the format of the *startgenes*, at the end of every iteration with their achieved fitness.
NEAT's structure is visualized in Figure 5.2. The population consists of the number of organisms that is specified by the population size that is set in the parameters file. An organism consists of the network, which contains the neural network (phenotype), as well as the genome (genotype). Further, the innovation history and species are saved separately in the population.

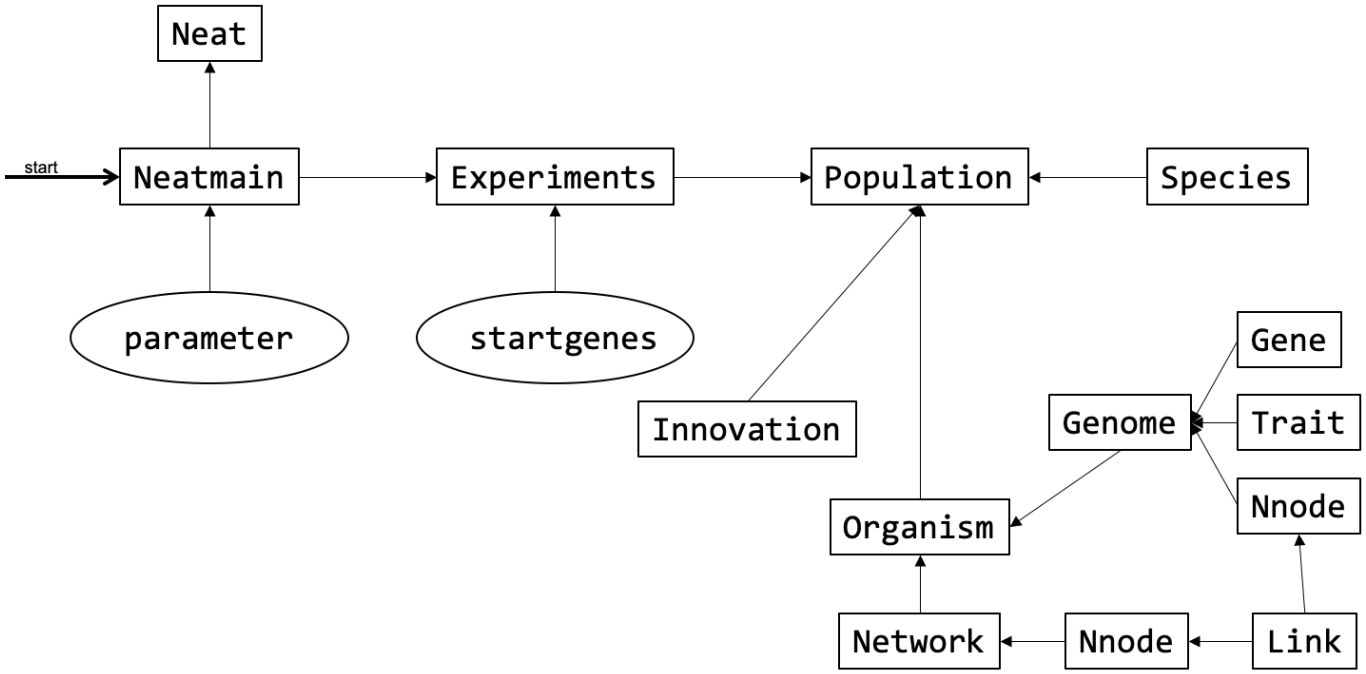


Figure 5.2: Structure of NEAT's Implementation

The NEAT learning process starts by reading in the parameters file. Then it asks which experiment should be conducted. Experiments are saved in a separate file and handle the learning process like visualized in Figure 2.6. The call to the experiment also specifies the maximum number of iterations to be done. In order to add new experiments, like our approach of scheduling, it needs to be added there with another number to call in the switch case. Then the experiment starts by reading the *startgenes* and creating a population out of it. Afterwards, the NEAT learning process starts by evaluating the population according to the problem of the experiment. The lowest performing organisms are destroyed and the space they leave is filled by the offspring of the mating process. These steps are all implemented in *population.cc*.
NEAT is capable of using any activation function. However, within one evolution, only one can be used at the same time. Generally NEAT uses a form of sigmoid function, thus the name of the activation function: `NEAT::fsigmoid`, which is saved in `neat.cc`. We change this function to the ReLU activation function to fulfill our design concept.
Further, we add a file we call *networkmodel.cc* with its header. Here we create the network scenarios that we want to evolve for. The network scenarios are callable by a function, which takes a `NEAT::Network` object and returns the achieved network measures. This function is called by the added experiment for every neural network in the population and with its returned values the fitness is calculated according to the fitness function. The achieved fitness is then attached to the organism the neural network is from.

5.3 Implementation of Neural Network Based Schedulers and Connection of NEAT and ns-3

To use NEAT with ns-3, we include the NEAT implementation in the ns-3 scratch folder. The scratch folder is provided by ns-3 to include custom simulations. We let ns-3 handle NEAT as a simulation that can be called as any other simulation. The point of the scratch folder is that the files in there are compiled separately. This makes frequent testing and changing of network scenarios, which should be put here, faster, since only the scratch folder needs to be compiled again, with the simulator itself staying untouched.
In order to build the neural network based schedulers we need to be able to use NEAT's neural network objects in the `MpTcpSocketBase` class. The problem here is that ns-3 is compiled and built with the python based *waf* compiler. The *waf* compiler uses a file called *wscript* for every subfolder of the *src* folder, where all the source files that need to be compiled are listed with their header files. If we would just include the `network.h` file of NEAT to use the neural networks, we would end up with errors, because the compiler does not compile and link it with the class that imports it. We need to add the paths to all the NEAT *.cc* and *.h* files to the *wscript*, so the compiler correctly compiles and links them with `mptcp-socket-base`. Then we can simply import `network.h` to `mptcp-socket-base` and use the neural networks, that are provided by NEAT.
Now that we can use the neural networks in the MultiPath TCP socket, we implement a static scheduler like explained in section 5.1. First, we add the *enum* for it to *DataDistribAlgo_t* in the *mp-tcp-typedefs.h* file. Then, we add the mapping of a string name to this enum in the *MakeEnumChecker* like in Listing 5.3. With that, the switch in `getSubflowToUse()` can switch to the new scheduler. A case is created with the defined *enum*. Here, we use the NEAT neural network object. For the static scheduler we gather the values of the subflows and save them in order, according to the model, into an *array* of *doubles*. NEAT networks provide a function called `load_sensors()`, which takes the array and loads the values into the input nodes. Afterwards, the function `activate()` is used to calculate the output of the neural network. To obtain the outputs from the network an iterator over NEAT nodes is defined. Hence, the file *nnode.h* must be included as well. Then, the iterator iterates over the output nodes of the neural network object and the respective values are saved again in an *array* in order. As for the static scheduler, the index of the maximum value of the output array is returned by `getSubflowToUse()`, as it indicates the chosen subflow. A sample implementation of the static scheduler can be found in the appendix, Listing 8.4.

We implement the meta scheduler with the same process as the static scheduler. In that, we define an *enum*, add that *enum* to the *MakeEnumChecker* and create a case the scheduler can switch to. However, implementing the meta scheduler itself is a bit more complicated. We create a new function called `metaScheduler()` to not over-bloat the `getSubflowToUse()` function. It returns the ID of the subflow to use to `getSubflowToUse()`, which relays the ID to `SendPendingData()`. The inputs for the neural network are again collected in an *array*, which is then put into the `load_sensors()` function. Also, the collection of outputs is the same as with the static scheduler. However, now we can not just return the index of the maximum of the array, since now it indicates the scheduler to use instead of the ID of a subflow. We take the index of the maximum and create a switch over it. In the switch there are as many cases as there are outputs of the neural network. Every case implements a static scheduler, which returns the ID of the subflow to use. A sample implementation of a meta scheduler with two static schedulers (MinRtt and RoundRobin) can be found in the appendix, Listing 8.5. The new dependencies are visualized in Figure 5.3.

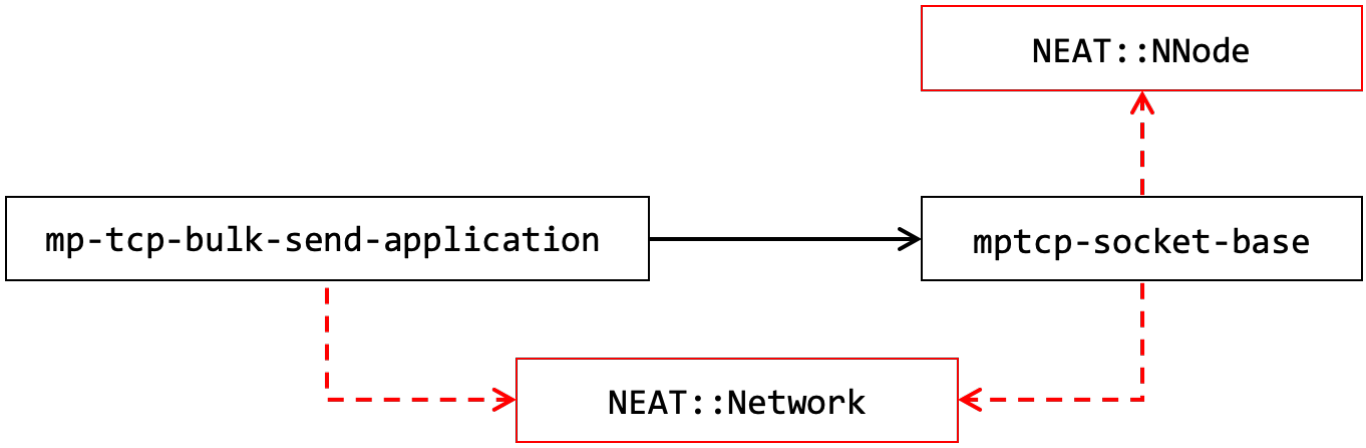


Figure 5.3: Extension of ns-3 with NEAT

We can now use NEAT's neural networks for scheduling. However, they would need to be defined inside the socket. What is left is the connection of NEAT and ns-3, so the generated neural networks can be used inside the socket. Like explained in section 5.2, we use the file *networkmodel* for the ns-3 simulations of our experiments. In this file, we use the *BulkSendApplication* for our MultiPath TCP connections. Applications in ns-3 are easy to use, as they encapsulate all the logic from the user. The user just has to set the parameters, like start and stop time, and does not have to define any logic. However, the user has no direct access to the socket the application defines and uses. This is a problem, since we want to set the neural networks that are generated by our learning algorithm in the socket, so they can perform the task of scheduling. To tackle this problem, we create a setter in `mptcp-socket-base`, which is receiving a `NEAT::Network` object and sets it in the socket. Now the application can set the neural network when initializing the socket. We just need to add another setter to the *BulkSendApplication*, so that we can set the `NEAT::Network` object directly in the application.

In Figure 5.4, it is visualized how the previously explained extensions work together. NEAT is invoked with a parameter file within ns-3, like shown in Listing 5.6

```
$ ./waf ––run "Neat␣p2nv.ne"
```

Listing 5.6: Change Scheduler Attribute in ns-3

Then the experiment is chosen with the corresponding number. NEAT creates a population out of the start topology that is defined in the experiment. In every epoch, or iteration of the learning process, every organism needs to be evaluated. This is done by calling the *networkmodel* with one neural network at a time. In the *networkmodel*, the network scenario is defined with using the *BulkSendApplication* and setting the *enum* to let the socket know which scheduler to use. Then, the setter of the *BulkSendApplication* for NEAT's neural networks is used to pass the neural network the *networkmodel* received. Once the simulation starts, the *BulkSendApplication* passes the neural network it received further to the socket, after initializing it. Once the simulation is finished, the wanted network measures are returned to the evaluation step of NEAT and the simulation is destroyed. There, the fitness is attached to the organism according to the fitness function and the next is evaluated. After the whole population is evaluated, it is checked if the target fitness was achieved. If not, the population is altered, like explained in section 2.2, and the next iteration is started.

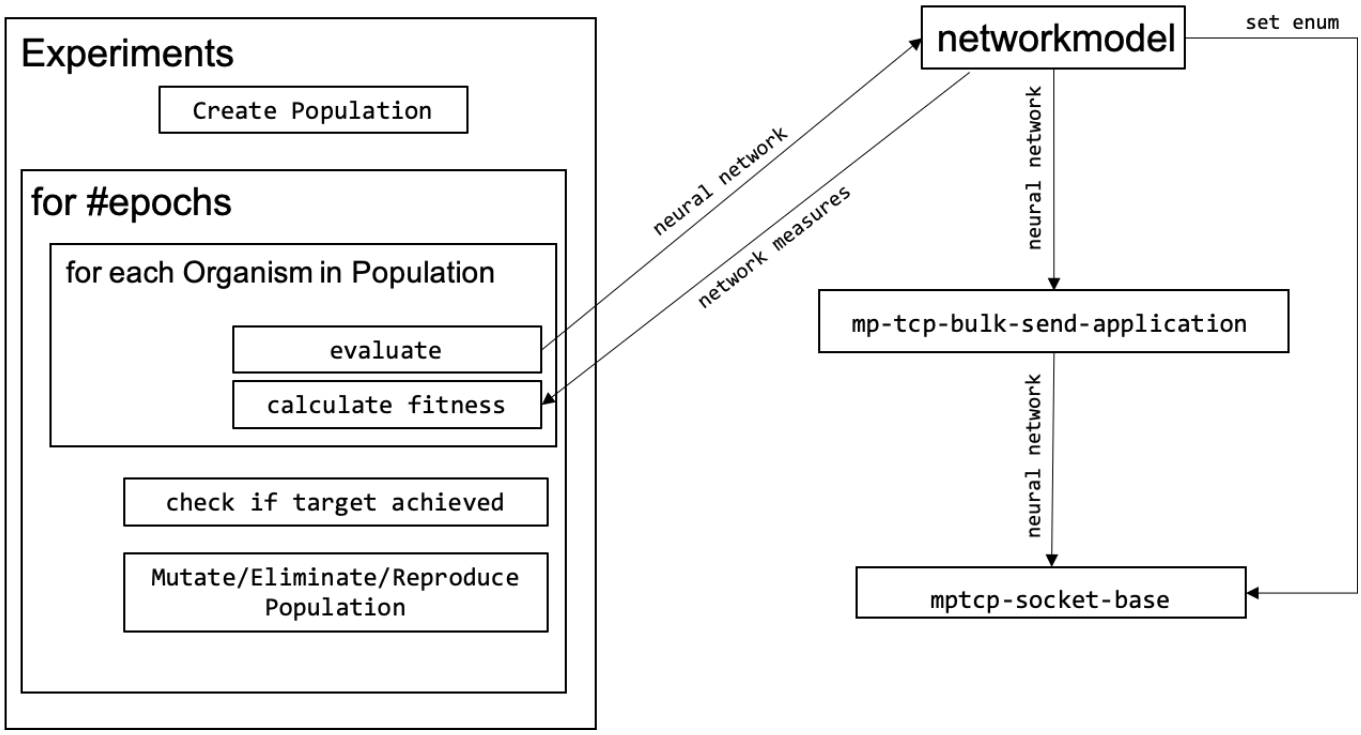


Figure 5.4: NEAT's Learning Process in ns-3

5.4 Translator

For a more convenient work flow we build an automatic translator. This translator takes a NEAT Neural Network specification as an input and writes the corresponding scheduling file in the .progmp format.
The translator is called as follows:

```
$ python translator.py --nn-f [NEAT NN spec] --out-f [output name] --model [NN model]
```

Listing 5.7: Translator program call

In the following, how the translator works and which limitations come with it are explained.

5.4.1 How does the translator translate?

There are several steps on how the translator conducts its translation.
Preliminary. We need to represent the neural network as a graph, as there are dependencies between nodes. If, for example, node6 is calculated with the output of node5 as an input, then node6 can only be calculated if the output of node5 is already known. To be sure of these dependencies the NN specification file is read and the nodes and links are saved as objects. The node objects contain just the node number and a list of *incoming nodes*. An *incoming node* is used for the calculation of the activation sum of a node object. It can be seen as a dependency of the node: If not all *incoming nodes* are calculated, then this node can't be calculated. The link objects contain the nodes which they connect, the weight and a flag if the link is recursive or not. A recursive link is a link which takes the output from the node of the current execution and feeds it into the node in the next execution. This special link can be seen as a short term memory, as it remembers the value from the node's last activation. If a recursive link is read, it is directly mapped to a free ProgMp register and saved in a dictionary. This is relevant for the printing of the net.
Building of the NN. After reading the file, we just have a list of objects. These need to be connected by filling the list of incoming nodes of each node in the network. Likewise, there is a list of dependencies for each node saved in each node itself.
Printing of the NN. Like priorly mentioned, nodes have dependencies on other nodes. There are three different ways a node is written, depending on the formula it inhibits. Since ReLU is used as the activation function, if there are only negative weights, the output of a node can only be 0. In that case the variable for this node is simply assigned 0 without further calculations. If there are only positive weights, then the sum of all incoming nodes multiplied by their corresponding weights is assigned to the variable of the node. The last possibility is the most complex one. Since ProgMp can only handle positive integers there is no way to check if the activation sum of a node is positive or negative. To solve that problem the negative weighted nodes are separated from the positive weighted ones. The negative weights are made positive and two separate sums are calculated. Then, if the sum of nodes with negative weights is bigger than the positive counterpart, the variable for the node is, again, simply assigned 0. Otherwise the difference of the positive and negative sum is assigned, since it is surely bigger than 0. Also, since variables in ProgMp are single assignable, a detour with registers needs to be taken. The template for this way of writing is shown in the following Listing 5.8.

```
VAR pos[node#] = n1 * w1 + ... + nN * wN;
VAR neg[node#] = n1 * w1 + ... + nM * wM;
IF (neg[node#] > pos[node#]) {
        SET(R3, 0);
ELSE {
        SET(R3, pos[node#] - neg[node#]);
}
VAR n[node#] = R3;
```

Listing 5.8: ProgMp node calculation template

Recursive Links. Since there is a possibility of recursive links, the check for dependencies is more complex. The node with a recursive link would be a dependency to itself. To combat this problem the check for dependencies disregards it, if it is the same node. Also, writing of the node is more complex. While writing sums for the node it needs to be checked if it is a recursive link, then the priorly assigned register is read from the dictionary and printed like that. Further, after the calculation of the current activation, the assigned register needs to be set to this activation for the next execution.

```
VAR n[node#] = n1 * w1 + n2 * w2 + R1 * w3;
SET(R1,n[node#]);
```

Listing 5.9: ProgMp node with recursive link calculation template

5.4.2 Limitations of using the translator

In the current state of the translator, it is a near optimal way to translate, while still preserving some readability. A prior state of the translator wrote all sensor nodes (inputs) as their own variables. Thereby, it uses a lot of precious variables and would not be able to write models for 3 flows with 4 inputs. It would use 12 variables for the sensor nodes and 9 variables for the output nodes. With the retransmission block a minimal 3 flow model, without a hidden layer, would already use 23 variables. Hence, only a minimal 3 flow model, where the inputs are connected to the outputs without hidden nodes is translatable. In the current state the sensor nodes are directly written inside the calculation of nodes without the detour by defining them as variables. This makes the prior example only use 11 variables, since the variables for the input nodes are economized.
All these limitations are there primarily, because ProgMp is really restrictive. However, certainly there are ways of making bigger neural nets usable in ProgMp, when analyzing and minimizing them by hand. However, for a general way of a fast and simple translation of NEAT neural networks into ProgMp schedulers it is a handy tool that saves a lot of time.

5.4.3 Example Translation

For this translation a model for 2 flows is used.
The topology of this net is visualized in Figure 8.1. Note that in the graphic the weights are neglected for better visibility. Also note that the traits in Listing 8.6 are irrelevant for the translation, since these are special parameters for the evolution inside NEAT and have no meaning outside of an evolution. The net itself is sufficiently specified by the nodes and links. As for the nodes, only the first and last number are relevant. The first number is the ID of the node and the last number is the position or role of the node (1 = sensor/input, 2 = output, 3 = hidden). As for links, the second, third, fourth and last numbers are relevant. The second and third number are the IDs of the nodes that will be connected by the link, the fourth number is the weight and the last number indicates whether the link is active or disabled.

5.5 Summary

In summary, we implemented our approach of creating schedulers by using a C++ implementation of NEAT and an implementation of MultiPath TCP in ns-3. We extended NEAT's experiments with the network simulations of ns-3. We implemented and extended ns-3 with the capability to use neural network based schedulers. Further, we created a translator that creates a scheduler out of a NEAT neural network description for ProgMP.

6 Evaluation

In this chapter we show that it is possible to generate schedulers with our approach explained in chapter 4. We create one static and two meta schedulers. With the implementation chapter 5 we evolve static and meta schedulers. Then, we evaluate the generated schedulers in ns-3 according to various network measures. Afterwards, we translate them with our created translation for ProgMP and evaluate the best meta scheduler with mininet and ProgMP.

6.1 Evaluation Setup

With our approach we generate a static scheduler we call *static_neat* for the topology seen in Figure 6.1. There are two hosts that are connected via two switches, so there are two paths between them. All links have the same capacity and different delay. As for the throughput, each link has 10 Mbit/s. The links in the path from h1 to h2 via s1 each have a delay of 5 ms. Via the other path, the links have a delay of 1 ms. Further, there are two TCP crossflows, one over each switch going from h1 to h2. The MultiPath TCP connection, for which we generate a scheduler, also goes from h1 to h2, but can use both switches. We use *uncoupled* congestion control and the *full-mesh* path-manager. Each node has a queue size of 50 packets and a packet loss of 1%. As for the applications of the flows, we use the *BulkSendApplication* for all flow, which start at the same time point within the simulations. We call this topology *simple_2_flow*.
To learn schedulers we create a fitness function as a combination of throughput, delay and fairness as shown in Equation 6.1, where T_{total} is the total or sum of the throughput of all flows and d_{mean} is the average mean end-to-end delay of all flows. The Jain's Fairness Index $\mathcal{J}$ is calculated over all flows, so each of the MultiPath TCP flows is treated as a single flow. Here we are calculating fairness with four flows, the two MultiPath TCP flows and the two crossflows.

$$\frac{T_{total}}{d_{mean}} * \mathcal{J} \tag{6.1}$$

$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2} \tag{6.2}$$

We need to give NEAT a single fitness value for its evaluation step. With this fitness function we set throughput, delay and fairness in relation to each other. Whereas the value is reduced if the delay is higher, increased if the throughput gets higher and vice versa. Finally, this value is multiplied with the achieved fairness. Thereby, the value can not get high if the result is unfair.
We don't set a fitness target and take the best performing organism after 100 epochs instead.
For the learning algorithm we use the parameter file p2nv.ne, Listing 8.2. We use this file, because it has been proven to work well for robot arm control [SM02] as well as for generation of congestion control algorithms [WKAH20]. Further, we use a neural network with 6 input nodes and 2 output nodes. The output nodes are representative of the flows. As for the inputs we use three values for each flow, namely Lost Packets, Round Trip Time and Congestion Window.

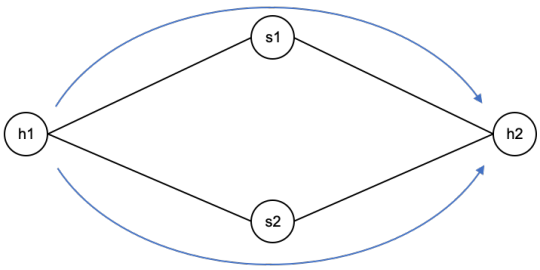


Figure 6.1: Simple Two Flow Topology

For the next experiments we use a double dumbbell topology, Figure 6.2. The flows from h1 to h4 via s1 and s3 as well as from h3 to h6 via s2 and s4 are TCP connections, whereas the connection from h2 to h5 is one MultiPath TCP connection for which we generate meta schedulers. For all configurations we use *uncoupled* congestion control and the *full-mesh* path-manager. All of these flows are saturated by the *BulkSendApplication*. Again, the nodes have a queue size of 50 packets and a packet loss of 1%. We use different configurations of link capacity and delay. Further, we divide the topology horizontally

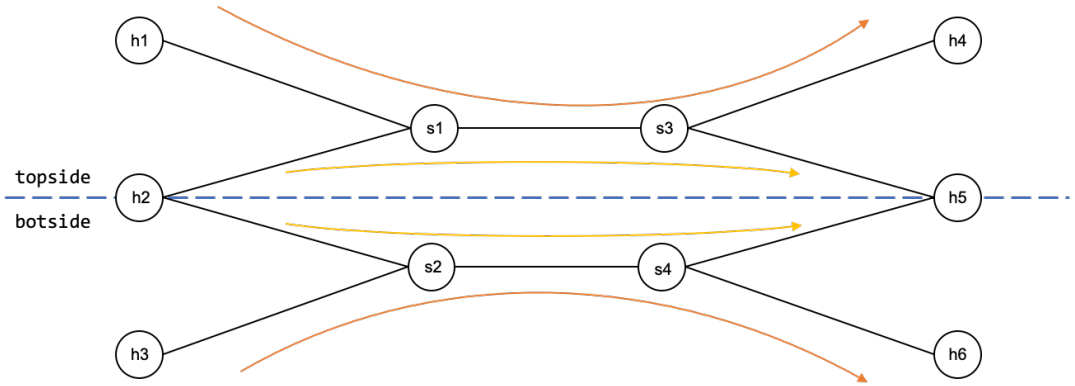


Figure 6.2: Double Dumbbell Topology : Divided into topside and botside along the blue line. There are two TCP flows from h1 via s1 and s3 to h4 and h3 via s2 and s4 to h6, marked in orange, as well as a MultiPath TCP connection from h2 to h5, marked in yellow.

name	topside		botside	
	throughput	delay	throughput	delay
ddumbbell_1	10 Mbit/s	2 ms	1 Mbit/s	20 ms
ddumbbell_2	10 Mbit/s	5 ms	10 Mbit/s	1 ms
ddumbbell_3	10 Mbit/s	5 ms	10 Mbit/s → 3 Mbit/s	1 ms → 25 ms

Table 6.1: The Different Configurations of the Double Dumbbell Topology on a per Link Basis

in the middle and make changes uniformly to each half. In the first configuration with the double dumbbell topology, we call *ddumbbell_1*, each link in the top side has a capacity of 10 Mbit/s with a delay of 2 ms. As for the bot side, each link has a capacity of 1 Mbit/s with a delay of 20 ms. We use this configuration as it represents heterogeneity.
The next configuration, we call *ddumbbell_2*, on the other hand is very homogeneous. In the topside all links have the same capacity of 10 Mbit/s and a delay of 5 ms. The botside links also have a capacity of 10 Mbit/s and a delay of 1 ms.
The last configuration, we call *ddumbbell_3*, changes parameters within the simulation. In the first half of the simulation we use the parameters like in *ddumbbell_2*. For the second half, we change the botside parameters to 3 Mbit/s capacity and a delay of 25 ms.

With these configurations we learn two kinds of meta schedulers. We generate meta schedulers with two static schedulers and with three static schedulers. For the meta schedulers with two static schedulers we use *MinRTT* and *RoundRobin*. We use these two, because they are performing optimally on different kinds of topologies. *MinRTT* performs better than *RoundRobin* in heterogenous topologies, whereas *RoundRobin* performs better than *MinRTT* in homogeneous topologies. For the learning algorithm we use the configurations *ddumbbell_1* and *ddumbbell_2*, as they represent both, a heterogeneous and a homogeneous topology. Again, we use the parameter file p2nv.ne, Listing 8.2.
Further, we conduct an evolution to generate a meta scheduler with three static schedulers. As for the static schedulers, we use *MinRTT* and *RoundRobin* because of the same reasons as before, as well as the static scheduler *static_neat* we evolved before. We use an implementation of *MinRTT* keeps a counter and makes sure to send a packet on each subflow every n packets. For our meta schedulers we use *MinRtt50* which makes sure that a subflow can only be chosen a maximum of 50 times in a row. In the evaluations we also show the results for *MinRtt30* which respectively has a maximum of 30. Now we use three topologies for the learning algorithm: *simple_two_flow*, *ddumbbell_1* and *ddumbbell_2*. Each of the used schedulers outperforms the other schedulers on one of these topologies. We call the resulting scheduler *meta_triple*. In Table 6.1 the different configurations used for the double dumbbell topology are listed. In Table 6.2 all the conducted evolutions are listed. For all evolutions we use the fitness function depicted in Equation 6.1 and the the parameter file p2nv.ne, Listing 8.2.

6.2 Evaluation Results and Analysis

We evaluate the following schedulers in regards to total throughput, mean end-to-end delay and fairness on all topologies described in section 6.1.

Scheduler	Used Static Schedulers	Used Network Topologies
static_neat	-	simple_2_flow
meta2	*Min_RTT50*, *RoundRobin*	ddumbbell_1, ddumbbell_2
meta_triple	*Min_RTT50*, *RoundRobin*, *static_neat*	simple_2_flow, ddumbbell_1, ddumbbell_2

Table 6.2: Evolved Schedulers in regards to which network topologies they are evolved on and to what static schedulers they have access to.

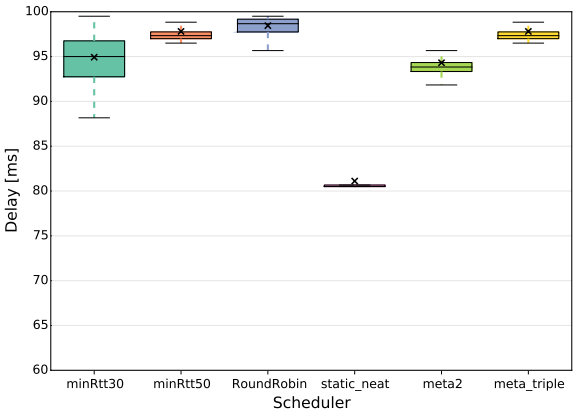
- **MinRtt30:** Our implementation of MinRTT where paths are not taken for a maximum of 30 packets
- **MinRtt50:** Our implementation of MinRTT where paths are not taken for a maximum of 50 packets
- **RoundRobin:** The RoundRobin scheduler
- **static_neat:** Our evolved static scheduler, optimized for the Simple Two Flow Topology, Figure 6.1
- **meta2:** Our evolved meta scheduler with access to *MinRtt50* and *RoundRobin*
- **meta_triple:** Our evolved meta scheduler with access to *MinRtt50*, *RoundRobin* and *static_neat*

All results are out of 100 simulations. This section is divided into two parts. The first part is about ns-3 simulations, the environment the schedulers are evolved on. In the second part the schedulers are translated into ProgMP and evaluated in mininet.

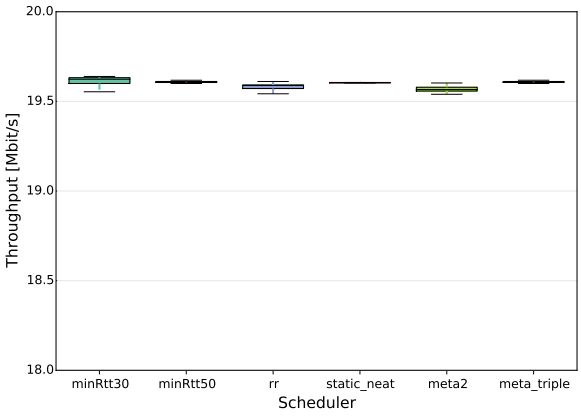
6.2.1 Simple Two Flow Topology - ns-3

For this topology we generated the static scheduler *static_neat* and the meta scheduler *meta_triple*. In Figure 6.3 it can be seen that *static_neat* achieves the lowest mean end-to-end delay out of all schedulers. As for the meta schedulers, an improvement can be seen only for *meta2*. The *meta_triple* scheduler performs exactly like *MinRtt50*, indicating that this meta scheduler did not learn the decision for this topology and just uses this scheduler, instead of switching to *static_neat* like intended.
What is interesting to see is, that *meta2*, which only uses *RoundRobin* and *MinRtt50*, could improve the delay and perform better than both.
Regarding throughput, all schedulers perform quite the same. It can be seen, that *meta_triple*, again, performs exactly the same as *MinRtt50*, which strengthens the claim that it just uses *MinRtt50*. Further, our generated static scheduler *static_neat* is in the midfield, but has nearly no jitter. Whereas *meta2* has less throughput than its parts. This means, here happened a tradeoff of delay for throughput.
Regarding fairness, we calculate it as priorly explained, with Jain's Fairness Index over all TCP flows and MultiPath TCP subflows. Again, *meta_triple* performs exactly the same as *MinRtt50*. But, *meta2* is in between its parts and leaning to the better performing one. This is a good sign, as it manages to nearly perform like our definition of optimal for meta schedulers. Here, *static_neat*, performs optimal in regards to fairness, with again nearly no variance in the data.
Finally, if we look at the function, that we optimized, Equation 6.1. It is clearly seen that *static_neat* outperforms all other schedulers, in Figure 6.3. That was to be expected, given the learning algorithm works. What was not to be expected, is that *meta2* also outperformed the other schedulers beside *static_neat*. We priorly assumed that a meta scheduler can only perform as good as its parts, but here we observe better performance. Also *meta_triple* did not perform well. It clearly did not perform as good as it is theoretically able to, since it has access to *static_neat*.

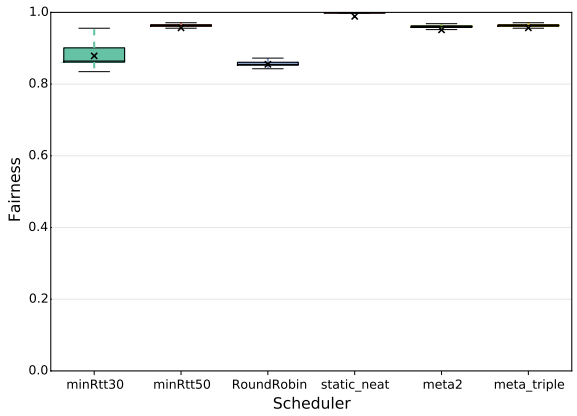
In conclusion: **static_neat:** We managed to generate a static scheduler, *static_neat* that performs near optimum. There is nearly no variance, in the data for *static_neat*. We have evaluated it multiple times and we used the exact same simulation for all schedulers, only changing the schedulers and the output file path. So, even though there is a random loss function involved, *static_neat* manages to perform nearly the same, throughout 100 simulations. For the learning we used 5 simulations and took the average out of those for the fitness function. This could indicate, that NEAT was not only capable to learn to improve the performance of the network through the scheduler, but also how to mitigate the random loss function.
meta2: This scheduler was not generated on this topology. So this is an unseen endeavour for it, but we can see, that it still makes good decisions.
meta_triple: We observe that this scheduler behaves exactly the same as *MinRtt50*, even tough it has access to the best performing static scheduler *static_neat*. This indicates that, even tough it was trained for this topology, it did not perform like we would expect.
In regards to the average case, in Table 6.3, all schedulers are listed with *MinRtt30* as a baseline. As we don't use *MinRtt30* in our meta-schedulers, it is easy to compare the other schedulers to each other, when we use *MinRtt30* as the baseline. It can be clearly seen, that *static_neat* improves the delay the most out of all schedulers, with 14.55% less end-to-end delay.



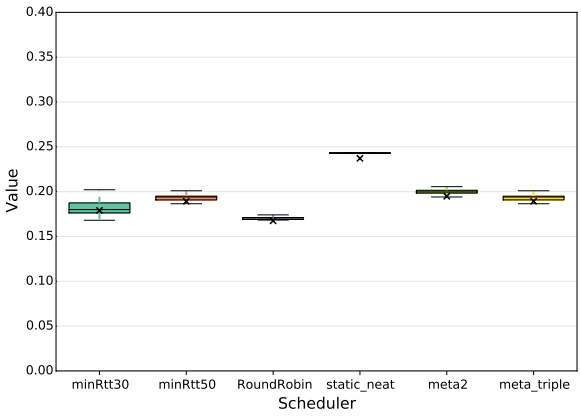
(a) Simple Two Flow Topology - Mean End-to-End Delay



(b) Simple Two Flow Topology - Total Throughput



(c) Simple Two Flow Topology - Fairness



(d) Simple Two Flow Topology - Value (Fitness Function)

Figure 6.3: Simple Two Flow Topology in ns-3

For fairness as well, *static_neat* improves fairness the most, by 12.53%, reaching near optimal fairness. Regarding the value of the fitness function we see an improvement of 32.43%, which, again, is the highest out of all schedulers. Our generated static scheduler achieves all of these improvements by only decreasing the total throughput by 0.06%. Further, it can be seen, that *meta_triple* indeed only chooses to use *MinRtt50*, as it has the exact same measures. Whereas *meta2* improves the delay slightly, but also slightly decreases throughput, which results in an increase of the value of the fitness function of 8.80%, which is higher than *MinRtt50* (5.70%) and *RoundRobin* (-6.48%). Even tough it consists out of these two, *meta2* manages to increase the performance. This indicates that it is possible to for the meta scheduler to perform better than its parts. So it may be of benefit to be able to switch the scheduler within a data transmission. Ultimately, our assumption that the meta scheduler can only be as performant as its single parts, is wrong. Clearly, it is of advantage to be able to change the scheduler within one data transmission.

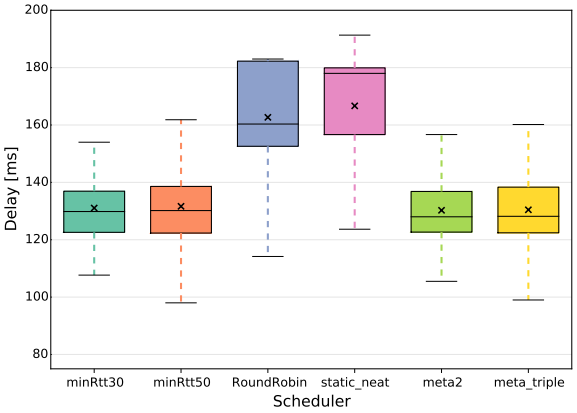
Table 6.3: Averages of Simple Two Flow Topology with MinRtt30 as Baseline.
Where Throughput is the total throughput of all flows (higher is better), Delay is the mean end-to-end delay (lower is better), Fairness is Jain's Fairness Index over all flows (higher is better) and Value is the value of the fitness function (higher is better)

	MinRtt30	MinRtt50	RoundRobin	static_neat	meta2	meta_triple
Throughput	0.00%	-0.03%	-0.17%	-0.06%	-0.25%	-0.03%
Delay	0.00%	3.03%	3.71%	-14.55%	-0.65%	3.03%
Fairness	0.00%	8.88%	-2.68%	12.53%	8.25%	8.88%
Value	0.00%	5.70%	-6.48%	32.43%	8.80%	5.70%

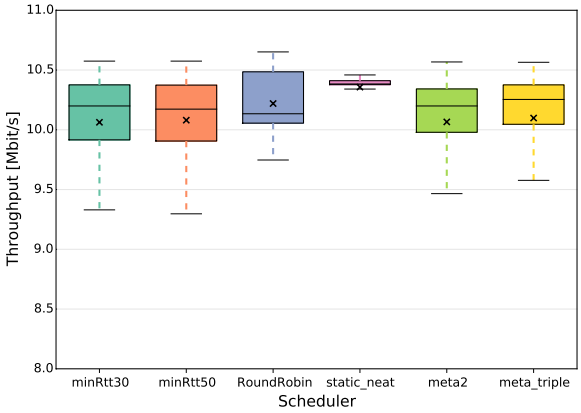
6.2.2 Double Dumbbell 1 - ns-3

This topology was part of the learning algorithm of *meta2* and *meta_triple*, so it is to be expected that they perform well. Because *static_neat* is evolved for a homogeneous topology, we would expect that it will not perform as well as it did in the prior experiment.

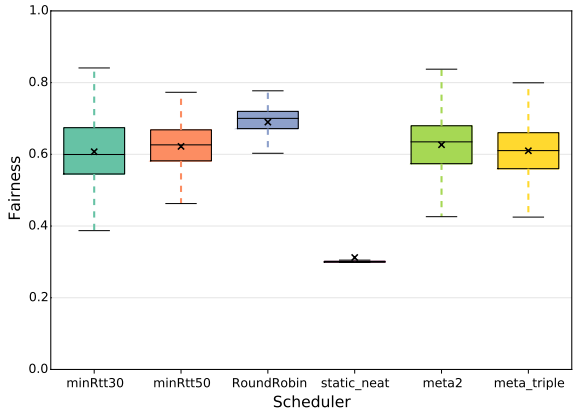
The delay of *meta2* and *meta_triple* is indeed as good as the best static scheduler in *MinRtt50* or *MinRtt30*. As expected,



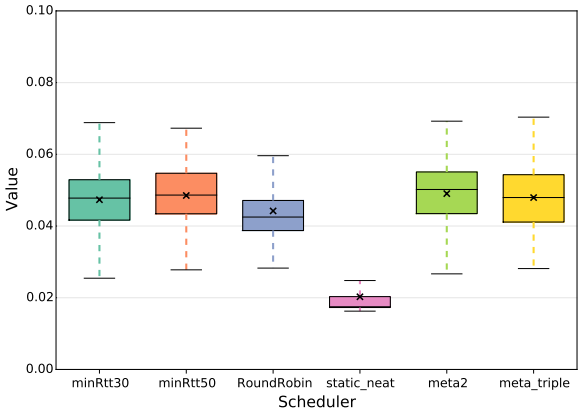
(a) Double Dumbbell Topology 1 - Mean End-to-End Delay



(b) Double Dumbbell Topology 1 - Total Throughput



(c) Double Dumbbell Topology 1 - Fairness



(d) Double Dumbbell Topology 1 - Value (Fitness Function)

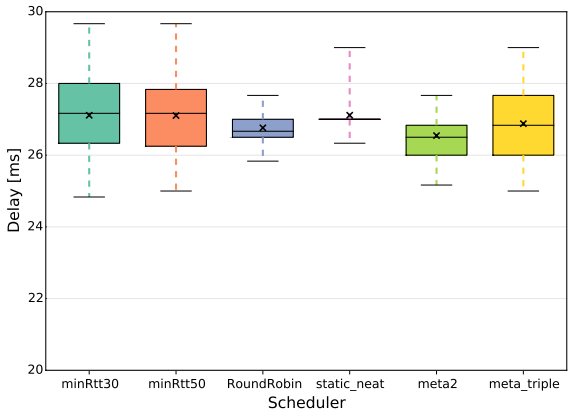
Figure 6.4: Double Dumbbell Topology 1 in ns-3

RoundRobin and *static_neat* have the highest delay out of all. *RoundRobin* as well as *static_neat* are expected to perform worse on heterogeneous topologies.

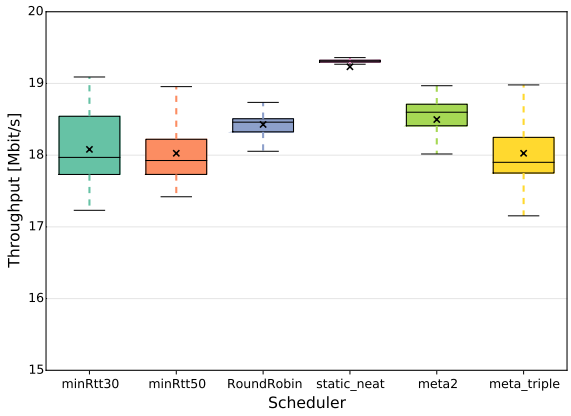
As for the throughput, *static_neat* actually manages to achieve the best throughput out of all. However this is only because it does not saturate the path on the topside and lets the crossflows use most of the capacity. And since there is less competition with *static_neat*, the highest total throughput is achieved. However, with *static_neat* the MultiPath TCP connection has the lowest throughput out of all the schedulers. Further, we can see that the meta schedulers we generated are performing slightly better than their *MinRtt50* part, but worse than *RoundRobin*, while having the lowest delay of both. Hence, there is actually a slight improvement, as it is again better than its single parts.

The priorly explained low-coming of *static_neat* can be seen in its fairness, Figure 6.6. However, a perfect fairness cannot be achieved here, as the crossflows have to work with capacities that are different by a factor of 10. Again, we see the same behavior here as with the throughput, the two meta schedulers are performing about the same as *MinRtt50*.

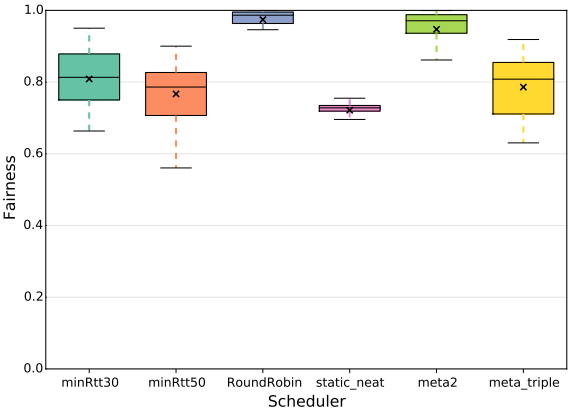
With the fitness function we can see all three measures in relation to each other. The same pattern can be observed, as the meta schedulers are about the same as *MinRtt50*. Further, with this measure, the flaw of *static_neat* can be seen, in that it may have improved the throughput, but sacrificed delay and fairness for it, which makes it the least performing scheduler out of all. The same with *RoundRobin*, in that it is the second least performing scheduler. Because both are for rather homogeneous topologies, instead of heterogeneous ones as used here, this was to be expected. Finally, we observe that our generated meta schedulers learned to choose the "correct" *MinRtt50* scheduler most of the time.



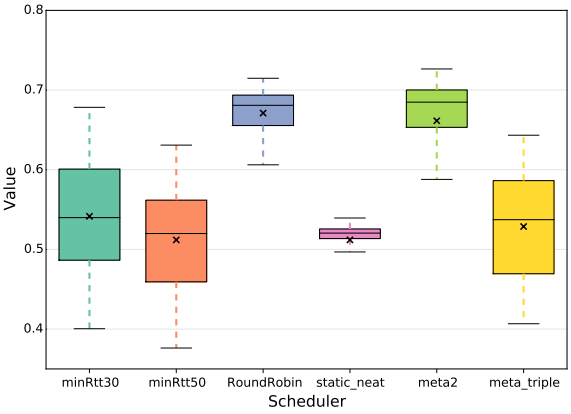
(a) Double Dumbbell Topology 2 - Mean End-to-End Delay



(b) Double Dumbbell Topology 2 - Total Throughput



(c) Double Dumbbell Topology 2 - Fairness



(d) Double Dumbbell Topology 2 - Value (Fitness Function)

Figure 6.5: Double Dumbbell Topology 2 in ns-3

Regarding the average case, Table 6.4, the flaws of *static_neat* can be clearly seen. It may increase the total throughput by 2.92%, but regarding the other measures it fails to perform. Increasing the delay by 27.17% and slicing the fairness by 48.58%, it decreases the value of the fitness function by 57.18%. We see that *meta_triple* (1.27%) is performing in between the two best schedulers it has access to, *RoundRobin* (-6.60%) and *MinRtt50* (2.49%), regarding the value of the fitness function. *Meta2*, again, manages to outperform its single parts by 1.11%, further supporting the thought that it may be beneficial to be capable to switch schedulers within a data transmission.

Table 6.4: Averages of Double Dumbbell 1 Topology with MinRtt30 as Baseline.
Where Throughput is the total throughput of all flows (higher is better), Delay is the mean end-to-end delay (lower is better), Fairness is Jain's Fairness Index over all flows (higher is better) and Value is the value of the fitness function (higher is better)

	MinRtt30	MinRtt50	RoundRobin	static_neat	meta2	meta_triple
Throughput	0.00%	0.18%	1.56%	2.92%	0.03%	0.35%
Delay	0.00%	0.45%	24.12%	27.17%	-0.61%	-0.47%
Fairness	0.00%	2.43%	13.58%	-48.58%	3.19%	0.43%
Value	0.00%	2.49%	-6.60%	-57.18%	3.60%	1.27%

6.2.3 Double Dumbbell 2 - ns-3

This topology is rather homogeneous, so it is to be expected that *RoundRobin* would be the best performing static scheduler. We expect the meta schedulers to correctly choose *RoundRobin* most of the time in most of the simulations.
As for the mean end-to-end delay, we observe that indeed *RoundRobin* is the best static scheduler. The meta scheduler *meta2*, which only has access to *MinRtt50* and *RoundRobin*, actually has better delay than either of them. On the other hand, *meta_triple* performs closely to *MinRtt50*, which is unexpected, since it should have chosen *RoundRobin* most of the time. This indicates that *meta_triple* can not correctly discriminate between the topologies.
As for total throughput, again, *static_neat* outperforms all other schedulers. However, this is the same phenomenon as in the previous experiment, as again the MultiPath TCP connection has the lowest throughput out of all flows. Therefore, we consider *meta2* as the best performing scheduler in this category. Once more, we observe that it performs better than either of its single parts. Further, *meta_triple* is again performing like *MinRtt50*, which supports the claim that it does not choose the "correct" static scheduler.
In regards to fairness, again, we see again the flaw of *static_neat*, in that it has the lowest fairness, because it suppresses the MultiPath TCP connection and reaches better throughput out of less contention. *RoundRobin* achieves the highest fairness, which is to be expected in a homogeneous topology. Again, we see that *meta_triple* performs about the same as *MinRtt50*. The best performing scheduler in terms of throughput, *meta2*, has a lower fairness than *RoundRobin*, indicating that it sacrificed fairness for throughput.
When looking at the value of the fitness function, for which the generated schedulers were optimized, we see that *meta2* is the best performing scheduler out of all. Also performing better than its parts again, by having better performance than *RoundRobin*. On the other hand, *meta_triple* is performing just slightly better than *MinRtt50*, indicating that it sometimes makes decisions not to take *MinRtt50*.

Regarding the average case, shown in Table 6.5, it can be seen that *meta2* performs about as good as *RoundRobin*. It achieves more throughput and a lower delay for the cost of fairness. Also, *meta_triple* not only chooses *MinRtt50*, as it performs better than it in the average case, but it still does not perform like expected, as it is theoretically able to achieve a performance increase of 22% of the baseline, but rather performs 2% worse than it. What is interesting to see is, that *static_neat* achieves about the same value as *MinRtt50* and does not perform as bad as in the previous experiment. This is because the prior topology was a heterogenous one, where as this one is rather homogeneous, and as *static_neat* was optimized for a homogeneous topology we don't see as much of a performance drop here.

Table 6.5: Averages of Double Dumbbell 2 Topology with MinRtt30 as Baseline.
Where Throughput is the total throughput of all flows (higher is better), Delay is the mean end-to-end delay (lower is better), Fairness is Jain's Fairness Index over all flows (higher is better) and Value is the value of the fitness function (higher is better)

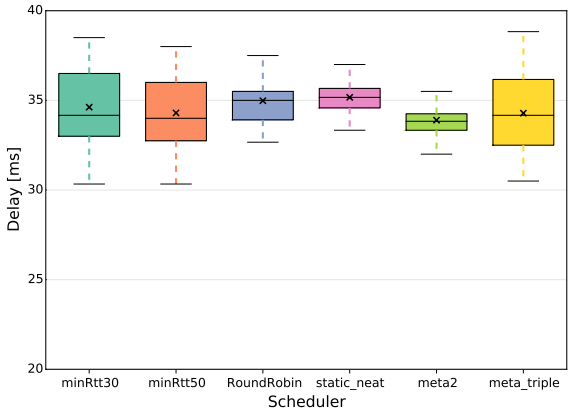
	MinRtt30	MinRtt50	RoundRobin	static_neat	meta2	meta_triple
Throughput	0.00%	-0.30%	1.92%	6.37%	2.30%	-0.30%
Delay	0.00%	-0.03%	-1.29%	0.01%	-2.09%	-0.86%
Fairness	0.00%	-5.17%	19.21%	-10.79%	17.11%	-2.81%
Value	0.00%	-5.46%	22.08%	-5.45%	22.14%	-2.36%

6.2.4 Double Dumbbell 3 - ns-3

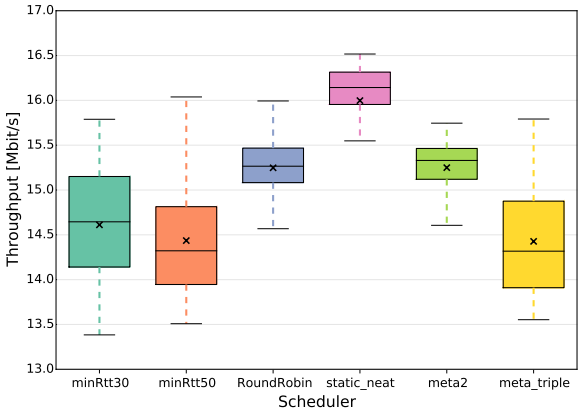
This topology changes from being homogeneous to being heterogeneous halfway within the simulation. Here we expect the meta schedulers to correctly switch between the best performing static schedulers they have access to. Because *meta2* and *meta_triple* have access to *MinRtt50* and *RoundRobin*, we expect them to first choose *RoundRobin*, and then, after the topology changes, to choose *MinRtt50*.

As for mean end-to-end delay, we observe that *meta2* improves performance, by switching correctly. But we observe again that *meta_triple* is about the same as *MinRtt50* if not worse, since there is a higher variance with wider fliers.
With total throughput we see again, that *meta_triple* is a bit worse than *MinRtt50*. Like in the previous experiments, *static_neat* has the highest total throughput out of all schedulers, but again this is only because it suppresses the MultiPath TCP connection. Further we see that, *meta2* is on par with *RoundRobin*.
As for fairness, *static_neat* is the worst performing out of all, because of the same reasons as in the previous experiments. The meta scheduler *meta_triple* is performing slightly worse than *MinRtt50*. This would be an good indicator that it would choose *MinRtt50* over *static_neat*, but in the first experiment, subsection 6.2.1, we see that *meta_triple* does not choose *static_neat*.

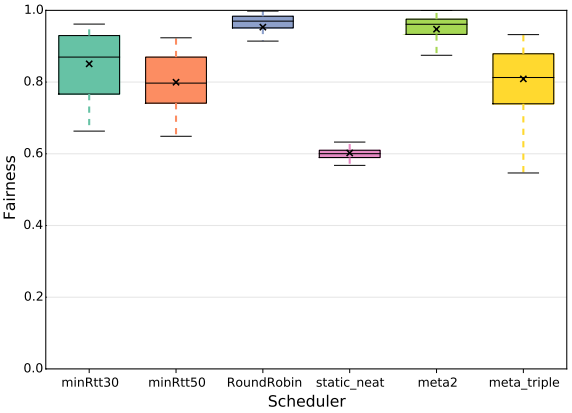
Indicating that it never chooses *static_neat* to begin with and rather chooses *MinRtt50* most of the times, regardless of the topology. When looking at the value of the fitness function, we see that *meta2* indeed improves the performance by being able to switch between *MinRtt50* and *RoundRobin*. Again, *meta_triple* only chooses *MinRtt50* showing that it is not able to switch at all. The generated static scheduler, *static_neat*, is the worst out of all, which is to be expected, since it did not perform well for the previous two experiments and is optimized for other kinds of topologies.



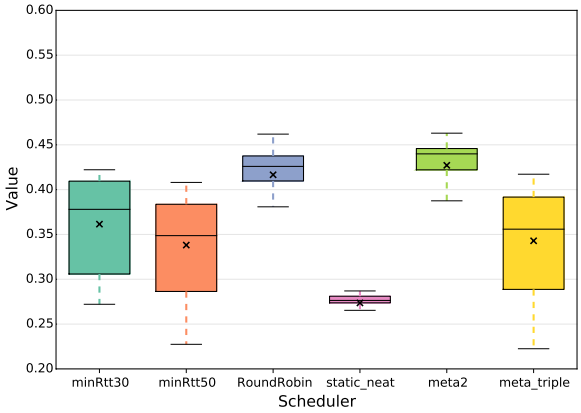
(a) Double Dumbbell Topology 3 - Mean End-to-End Delay



(b) Double Dumbbell Topology 3 - Total Throughput



(c) Double Dumbbell Topology 3 - Fairness



(d) Double Dumbbell Topology 3 - Value (Fitness Function)

Figure 6.6: Double Dumbbell Topology 3 in ns-3

Regarding the average case, Table 6.6, we see that *meta2* outperforms its single parts again. It achieves about the same total throughput as *RoundRobin*, but with 3% less delay at the cost of 0.67% fairness. This totals in a 3% better performance according to the value of the fitness function. Again, we see that *meta_triple* does not perform like expected.

Table 6.6: Averages of Double Dumbbell 3 Topology with MinRtt30 as Baseline.
Where Throughput is the total throughput of all flows (higher is better), Delay is the mean end-to-end delay (lower is better), Fairness is Jain's Fairness Index over all flows (higher is better) and Value is the value of the fitness function (higher is better)

	MinRtt30	MinRtt50	RoundRobin	static_neat	meta2	meta_triple
Throughput	0.00%	-1.20%	4.36%	9.49%	4.37%	-1.26%
Delay	0.00%	-0.93%	1.04%	1.58%	-2.10%	-0.97%
Fairness	0.00%	-6.03%	12.03%	-29.22%	11.36%	-4.93%
Value	0.00%	-6.46%	15.23%	-24.31%	18.15%	-5.14%

6.2.5 Conclusions on ns-3 Evaluations

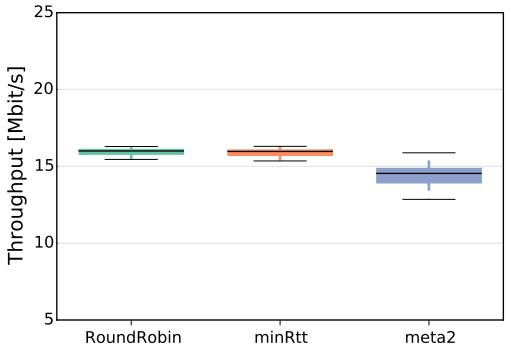
In conclusion, we see that it is possible to generate static and meta schedulers with our approach. We now take a look at all generated schedulers individually.
static_neat: This static scheduler is only evolved on a single topology, the simple two flow topology. On this topology it outperforms every other tested scheduler in terms of fairness and especially delay, while being on par in regards of throughput. This also shows in the value of the fitness function, where it outperforms all other tested schedulers by far. However, when we evaluate this scheduler in other topologies it was not trained on, it can not perform. In the other three experiments we have done, it reached peak total throughput every time, but only because it suppresses its own connection, so that the other TCP flows within the topology have most of the bandwidth. The throughput is higher, because there is less contention and the crossflows have "less to fight about" and can freely send their data. This can also be observed when looking at fairness. On the topology it was trained on, it achieves nearly perfect fairness. However, on the other topologies it has by far the worst fairness out of all. Even tough it was trained on a homogeneous topology, it can not perform on the homogeneous double dumbbell topology *(ddumbbell_2)*, indicating that it is overfitted.
In conclusion, we can say that it is possible to generate static schedulers with our approach, which optimize for a given target.

meta_triple: This meta scheduler was evolved for all topologies besides *ddumbbell_3* and has access to *MinRtt50*, *RoundRobin* and finally *static_neat*. We have expected it to be one of the best performing schedulers in all of our experiments, since it has access to all of the best performing static schedulers. However, it shows to just choose *MinRtt50* nearly all the time giving no real improvement. The reason for this is that the learning algorithm did not run long enough. It takes a lot of time to evolve a meta scheduler, because there have to be a lot of network simulations. We took the best performing scheduler out of our learning algorithm after having it evolve for more than 5 days, which resulted in having only about 30 epochs of evolution. This was clearly not enough.

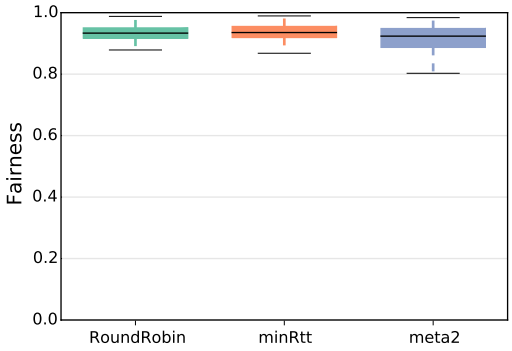
meta2: This meta scheduler has access to only two simple static schedulers. It is evolved on two network topologies. Against our expectations, it performed better than its static schedulers, *MinRtt50* and *RoundRobin*, in all our experiments. We had the assumption that this meta scheduler can only perform as good as the best static scheduler it has access to, but this clearly is not the case. Actually, it can perform better than its single parts. Which could mean that being able to switch the scheduler within a data transmission is beneficial. We expected this scheduler to only outperform its single parts in the case that the network topology changes *(ddumbbell_3)*. Here it performed like expected, but the other results hint at that in a static case, switching the scheduler is beneficial as well.

6.2.6 Mininet Evaluations

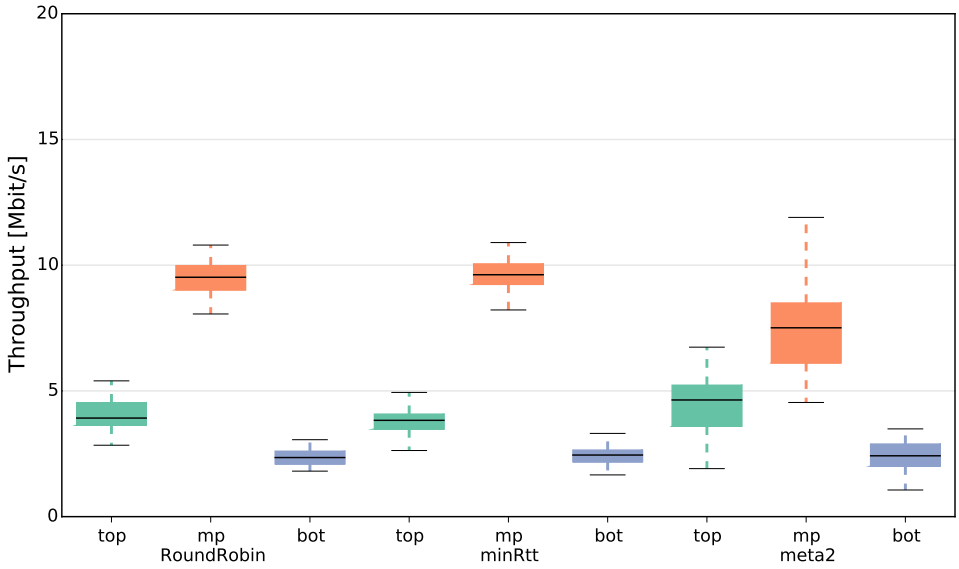
We evaluate the best performing meta scheduler, *meta2*, in mininet on the changing *ddumbbell_3* topology. Because it is not possible to change the topology within one simulation in mininet, we use the minievents [Gir]. With minievents it is possible to predefine events in a .json file, that are then executed within the simulation, like a change of bandwidth or delay. We translate *meta2* with our automatic translator and deploy it with ProgMP. In Figure 6.7 it can be seen that the meta scheduler does not perform as well as in the prior ns-3 simulations. The total throughput is lower, the fairness is lower and regarding the throughput of each flow individually, it can be seen that there is more variance in the data as well as less throughput of the MultiPath TCP connection.
This result can have various reasons. We built the topology in mininet the same way as in ns-3, but the applications are different. In ns-3 we use the *BulkSendApplication* and in mininet we use the program *iperf3*. Both saturate the flow as much as possible, but they are implemented differently. Further, the MultiPath TCP implementation in ns-3 could handle schedulers differently than the Linux implementation, even though the developers of the implementation claimed to be close to it. We don't think that our translation is the problem, as we thoroughly checked that it is correct. Rather, the problem should be caused by either the implementation in ns-3 or the Linux implementation handling schedulers or data transmission differently or the applications pushing their data differently. We think it is unlikely that the applications are the problem, as both just saturate the flows and want to send as much data as possible. The simulation with mininet itself should not be the problem, since the other two schedulers have comparable results to ns-3.



(a) Total Throughput



(b) Fairness



(c) Throughput of Each Flow when Using Different Schedulers

Figure 6.7: Double Dumbbell Topology 3 in Mininet

7 Conclusions

In this work we showed a mean of creating schedulers for MultiPath TCP by using Neuroevolution of Augmenting Topologies or NEAT. We presented two kinds of schedulers that can be created. First, a static scheduler, which directly conducts scheduling by assigning packets to paths. Second, we presented meta scheduling. In meta scheduling, a neural network switches between schedulers instead of assigning packets. The results we gathered with the network simulator ns-3 indicate that our approach of creating schedulers works. The static scheduler we created outperforms the *MinRtt* and *RoundRobin* scheduler in delay and fairness, while having comparable throughput. Our generated meta scheduler, that can switch between *MinRtt* and *RoundRobin*, consistently achieves better measures than either of them on various topologies. Thus, we assume that being able to switch the scheduler within a data transmission can have beneficial effects. Our preliminary assumption about meta schedulers was that the meta schedulers can only be as performant as the schedulers they are able to use. Clearly, this assumption was not entirely correct, in that they can actually increase performance. We observed these results with our best performing meta scheduler. Moreover, a meta scheduler that can switch between *MinRtt*, *RoundRobin* and our static scheduler was evolved. However, we could not get good performance after days of evolution and preliminarily stopped the evolution. This can be clearly seen in the evaluation, as it has not learned to switch the scheduler.
Then we deployed our meta scheduler to the linux kernel with ProgMP and evaluated it with mininet. To our surprise, we could not replicate the results that were gathered in ns-3. The reason for this could be that the assumption that we could re-model the topology was wrong. As mininet could use another loss function and we used different applications for our tests, the topologies could have been too different. Further, we believe that another reason could be that the ns-3 implementation of MultiPath TCP is handling scheduling differently than ProgMP, resulting in non-replicability.
In conclusion, one can create static and meta schedulers with our presented method, but they are not yet deployable to the kernel.

7.1 Future Work

As the time for this work was limited, we give a list of ideas on how to continue. There are still some parts that do not work to our satisfaction and we gathered some ideas which we were not able to implement.

- **Accelerate Evolutions:** Currently there can only be a single evaluation of each organism in the evaluation step of NEAT's learning algorithm. This is the bottleneck of the whole evolution and should be accelerated. Instead of evaluating one-by-one, there should be a parallel evaluation. As the simulations are independent on each other, there should be a way to parallelize the evaluation step. However, currently ns-3 does not directly support parallelization and it is not possible to use libraries like OpenMP [WKAH20].
- **Improvements on meta_triple:** As priorly mentioned, we could not create a meta scheduler that can choose between three schedulers and perform satisfactorily. One of the reasons was that we clearly did not give it enough time to evolve. However, there is also another idea to create these kinds of schedulers. We are able to create schedulers that choose between two schedulers in a low amount of time, compared to using three schedulers. This indicates that NEAT can divide the event space into two more efficiently than into three. So instead of creating a meta scheduler that is able to directly choose three schedulers, we propose to again create a meta scheduler that chooses between the meta scheduler that can choose two (*meta2*) and the new scheduler (*static_neat*). In that we would build a binary decision tree and play in to the strength of NEAT in dividing the event space.
- **Different Schedulers:** We created our meta schedulers only with *MinRtt*, *RoundRobin* and our created *static_neat*. However, there are a lot more schedulers that perform better in different topologies. The next step in meta scheduling is to use more schedulers to cover a wider variety of topologies.
- **Extension to MultiPath TCP in ns-3:** The used implementation of MultiPath TCP for ns-3 is not officially supported. To this date there is no official support for MultiPath TCP in ns-3. Even though there are several proposals,, none of them has been adopted yet. The implementation we used is one of those proposals to be included in ns-3. However, this one is lacking some functionality. As mentioned in chapter 5 only a single scheduler is included and we think that the usage of the schedulers is different than on the linux kernel. Also, the possibility to send data redundantly over multiple paths is not implemented. It would be a new frontier for our static schedulers to also be capable to send data redundantly, asvfor now they are forced to send on one path.

8 Appendix

```
genomestart 1
trait 1 0.1 0 0 0 0 0 0 0
trait 2 0.2 0 0 0 0 0 0 0
trait 3 0.3 0 0 0 0 0 0 0
trait 4 0.4 0 0 0 0 0 0 0
node 1 0 1 1
node 2 0 1 1
node 3 0 1 1
node 4 0 1 1
node 5 0 0 2
gene 1 1 5 0.0 0 1 0 1
gene 2 2 5 0.0 0 2 0 1
gene 3 3 5 0.0 0 3 0 1
gene 4 4 5 0.0 0 4 0 1
genomeend 1
```

Listing 8.1: Sample Startgenes File. Four Input Nodes Fully-Connected With One Outputnode.

```
trait_param_mut_prob 0.5
trait_mutation_power 1.0
linktrait_mut_sig 1.0
nodetrait_mut_sig 0.5
weight_mut_power 1.8
recur_prob 0.05
disjoint_coeff 1.0
excess_coeff 1.0
mutdiff_coeff 3.0
compat_thresh 4.0
age_significance 1.0
survival_thresh 0.4
mutate_only_prob 0.25
mutate_random_trait_prob 0.1
mutate_link_trait_prob 0.1
mutate_node_trait_prob 0.1
mutate_link_weights_prob 0.8
mutate_toggle_enable_prob 0.1
mutate_gene_reenable_prob 0.05
mutate_add_node_prob 0.01
mutate_add_link_prob 0.3
interspecies_mate_rate 0.001
mate_multipoint_prob 0.6
mate_multipoint_avg_prob 0.4
mate_singlepoint_prob 0.0
mate_only_prob 0.2
recur_only_prob 0.2
pop_size 200
dropoff_age 15
newlink_tries 20
print_every 1
babies_stolen 0
num_runs 1
```

Listing 8.2: NEAT Parameterfile, p2nv.ne

```
/*
 * Scheduler sending packets on the subflow with the lowest RTT which has cwnd.
 */

SCHEDULER illustratingMinRTT;

VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.SKBS_IN_FLIGHT
                                        + sbf.QUEUED AND !sbf.THROTTLED AND !sbf.LOSSY);

IF(sbfCandidates.EMPTY) {
  RETURN;
}

IF (!RQ.EMPTY) {
  VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(RQ.TOP)
                                          AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
  IF (sbfCandidate != NULL) {
    sbfCandidate.PUSH(RQ.POP());
    RETURN;
  }
}

IF (!Q.EMPTY) {
 sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(Q.TOP)).MIN(sbf => sbf.RTT).PUSH(Q.POP());
}
```

Listing 8.3: MinRTT Implementation in ProgMP [FRE+17]

```
uint8_t
MpTcpSocketBase::getSubflowToUse()
{
  NS_LOG_FUNCTION(this);

  uint8_t nextSubFlow = 0;

  switch (distribAlgo)
    {
        case Neat_meta:
        {
                return metaScheduler();
        }
        case Round_Robin:
        {
                nextSubFlow = (lastUsedsFlowIdx + 1) % subflows.size ();
                break;
        }
        case static_2_flow:
        {
                double out[2] = {0,0};
                double input[8] = {0,0,0,0,0,0,0,0};
                for (int i = 0; i < subflows.size (); ++i)
                {
                Ptr<MpTcpSubFlow> sFlow = subflows[i];
                input[i*4] = sFlow -> num_lost_pkts;
                input[i*4+1] = sFlow -> rtt -> GetCurrentEstimate ().GetMilliSeconds ();
                input[i*4+2] = sFlow -> rtt -> GetCurrentVariance ().GetMilliSeconds ();
                input[i*4+3] = sFlow -> cwnd;
                }
                ntw -> load_sensors (input);
```

```
                bool success = ntw -> activate ();
                if(success) {
                                std::vector<NEAT::NNode*>::iterator out_iter;
                                out_iter=ntw->outputs.begin();
                                out[0]=(*out_iter)->activation;
                                ++out_iter;
                                out[1]=(*out_iter)->activation;
                                if (out[0] > out[1]) {
                                nextSubFlow = 0;
                                } else {
                                if (subflows.size()>1)
                                        nextSubFlow = 1;
                        }
      }
    break;
  }
  default:
    break;
   }
 }
  return nextSubFlow;
}
```

Listing 8.4: Neural Network Based Static Scheduler for Two Flows

```
uint8_t
MpTcpSocketBase::metaScheduler()
{
 // Get Inputs
  double minRtt = std::numeric_limits<double>::max();
  double maxRtt = std::numeric_limits<double>::min();
  double minRttVar = std::numeric_limits<double>::max();
  double maxRttVar = std::numeric_limits<double>::min();
  double minCwnd = std::numeric_limits<double>::max();
  double maxCwnd = std::numeric_limits<double>::min();
  for (int i = 0; i < subflows.size (); ++i)
  {
    Ptr<MpTcpSubFlow> sFlow = subflows[i];
    double currRtt = sFlow -> rtt -> GetCurrentEstimate ().GetMilliSeconds ();
    double currRttVar = sFlow -> rtt -> GetCurrentVariance ().GetMilliSeconds ();
    double currCwnd = sFlow -> cwnd.Get ();
    minRtt = std::min(currRtt, minRtt);
    maxRtt = std::max(currRtt, maxRtt);
    minRttVar = std::min(currRttVar, minRttVar);
    maxRttVar = std::max(currRttVar, maxRttVar);
    minCwnd = std::min(currCwnd, minCwnd);
    maxCwnd = std::max(currCwnd, maxCwnd);
  }
  double input[4];
  input[0] = minRtt/maxRtt;
  input[1] = minRttVar/maxRttVar;
  input[2] = minCwnd/maxCwnd;
  input[3] = subflows.size();

  std::vector<double> values = {0,0,0};
  meta_ntw -> load_sensors (input);
  bool success = meta_ntw -> activate ();
  if(success) {
    std::vector<NEAT::NNode*>::iterator out_iter;
        out_iter=meta_ntw->outputs.begin();
        values[0]=(*out_iter)->activation;
        ++out_iter;
        values[1]=(*out_iter)->activation;
        }
        schedulerToUse = std::max_element(values.begin(), values.end())-values.begin();

  switch(schedulerToUse)
  {
    case 0: // MinRtt
    {
      for (int i = 0; i < subflows.size (); ++i)
      {
      Ptr<MpTcpSubFlow> sFlow = subflows[i];
      int64_t currRtt = sFlow -> rtt -> GetCurrentEstimate ().GetMilliSeconds();
      // If RTT is 1500 or 1000 -> The flow has just been created and needs to be probed
      if (currRtt == 1500|| currRtt == 1000){ return i;}
      }
      int currentLowest = 1500;
      int sFlowIdx = 0;


      for(auto it = rttMap.begin(); it != rttMap.end(); it++){
        if(it->second == maxPktsUnprobed){
```

```
        rttMap[it->first] = 0;
        return it->first;
      }
    }
    for (int i = 0; i < subflows.size (); ++i)
    {
    Ptr<MpTcpSubFlow> sFlow = subflows[i];
    rttMap[i]++;

    int64_t currRtt = sFlow -> rtt -> GetCurrentEstimate ().GetMilliSeconds();
    uint32_t currCwnd = sFlow -> cwnd.Get();
    //std::cout << "Flow" << i << ": " << currRtt << "ms" << std::endl;
    if(currRtt<currentLowest && currCwnd > sFlow->MSS){
      currentLowest = sFlow -> rtt -> GetCurrentEstimate ().GetMilliSeconds();
      sFlowIdx = i;
    }
    }
    rttMap[sFlowIdx] = 0;
    return sFlowIdx;
  }
  case 1: // RoundRobin
  {
    return (lastUsedsFlowIdx + 1) % subflows.size ();
  }
  default:
    return 0;
 }

}
```

Listing 8.5: Neural Network Based Meta Scheduler for Two Static Schedulers

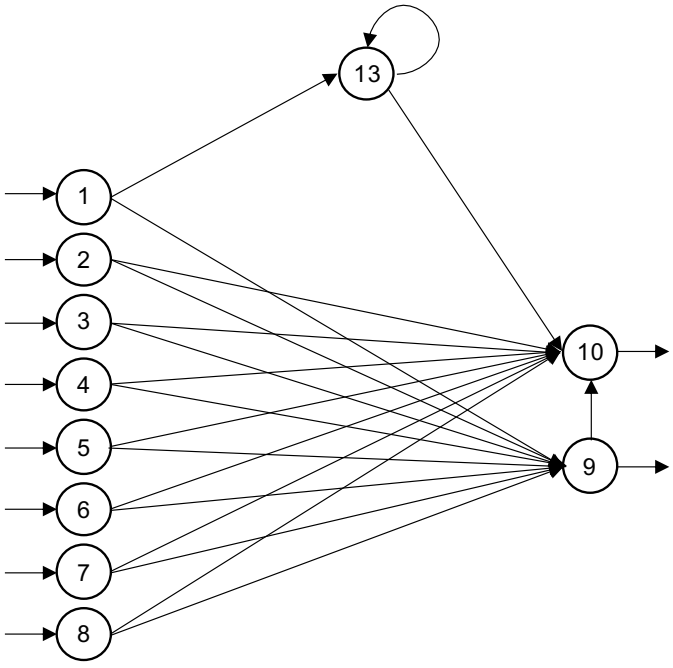


Figure 8.1: Sample Translation Neural Network Structure

```
genomestart 52
trait 1 0.1 0 0 0 0 0 0 0
trait 2 0.396372 0.0415226 0 0 0.159105 0 0 0
trait 3 0.3 0 0 0 0 0 0 0
trait 4 0.1 0 0 0 0 0 0 0
trait 5 0.2 0 0 0 0 0 0 0
trait 6 0.3 0 0 0 0 0 0 0
trait 7 0.1 0 0 0 0 0 0 0
trait 8 0.2 0 0 0 0 0 0 0
trait 9 0.3 0 0 0 0 0 0 0
trait 10 0.1 0 0 0 0 0 0 0
trait 11 0.2 0 0 0 0 0 0 0
trait 12 0.3 0 0 0 0 0 0 0
trait 13 0.1 0 0 0 0 0 0 0
trait 14 0.2 0 0 0 0 0 0 0
trait 15 0.3 0 0 0 0 0 0 0
trait 16 0.1 0 0 0 0 0 0 0
node 1 7 1 1
node 2 8 1 1
node 3 7 1 1
node 4 7 1 1
node 5 7 1 1
node 6 10 1 1
node 7 5 1 1
node 8 2 1 1
node 9 12 0 2
node 10 9 0 2
node 13 1 0 0
gene 3 1 9 1.405 0 1 1.405 1
gene 7 2 9 −0.568846 0 2 −0.568846 1
gene 16 3 9 −2.75604 0 3 −2.75604 1
gene 12 4 9 −0.947498 0 4 −0.947498 1
gene 13 5 9 1.30167 0 5 1.30167 1
gene 15 6 9 2.48005 0 6 2.48005 1
gene 6 7 9 2.7454 0 7 2.7454 1
gene 15 8 9 1.35972 0 8 1.35972 1
gene 6 1 10 1.78969 0 1 1.78969 0
gene 5 13 13 1.11000 0 13 1.11000 1
gene 12 2 10 −1.62793 0 2 −1.62793 1
gene 8 3 10 −2.66324 0 3 −2.66324 1
gene 11 4 10 0.714075 0 4 0.714075 1
gene 14 5 10 −0.46466 0 5 −0.46466 1
gene 3 6 10 −0.267898 0 6 −0.267898 1
gene 7 7 10 0.480736 0 7 0.480736 1
gene 1 8 10 1.93832 0 8 1.93832 1
gene 6 1 13 2.85359 0 15 2.85359 1
gene 6 13 10 2.82563 0 16 2.82563 1
gene 2 9 10 0.496565 0 23 0.496565 1
genomeend 52
```

Listing 8.6: ProgMp Node With Recursive Link Calculation Template

```
SCHEDULER output;

IF (R5 == 4){
        SET(R5 == 0);}
IF (R6 == 5){
        SET(R6 == 0);}
SET(R2,0);
SET(R3,0);
SET(R4,0);
VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.SKBS_IN_FLIGHT
                                + sbf.QUEUED AND !sbf.THROTTLED AND !sbf.LOSSY);
IF (!RQ.EMPTY) {
                IF (!SUBFLOWS.FILTER(sbf => !RQ.TOP.SENT_ON(sbf)).EMPTY) {
                        DROP( RQ.POP());
                        RETURN;
                } ELSE {
                VAR retransmissionCandidates = sbfCandidates.FILTER(sbf =>
                sbf.HAS_WINDOW_FOR(RQ.TOP) AND !RQ.TOP.SENT_ON(sbf));
                IF(!retransmissionCandidates.EMPTY) {
                        FOREACH(VAR sbf IN retransmissionCandidates) {
                        sbf.PUSH(RQ.TOP);
                }
                RETURN;
                }}
}
IF (!Q.EMPTY) {
        VAR n13 = R1 * 111 + SUBFLOWS.GET(0).LOST_SKBS * 285;
        SET(R1,n13);

        VAR pos9 = SUBFLOWS.GET(0).LOST_SKBS * 140 + SUBFLOWS.GET(1).LOST_SKBS * 130
                + SUBFLOWS.GET(1).RTT/8000 * 248 + SUBFLOWS.GET(1).RTT_VAR/8000 * 274
                + SUBFLOWS.GET(1).CWND * 135;
        VAR neg9 = SUBFLOWS.GET(0).RTT/8000 * 56 + SUBFLOWS.GET(0).RTT_VAR/8000 * 275
                + SUBFLOWS.GET(0).CWND * 94;
        IF (neg9 > pos9) {
                SET(R3,0);
        } ELSE {
                SET(R3, pos9 - neg9);
        }
        VAR n9 = R3;

        VAR pos10 = SUBFLOWS.GET(0).CWND * 71 + SUBFLOWS.GET(1).RTT_VAR/8000 * 48
                 + SUBFLOWS.GET(1).CWND * 193 + n13 * 282 + n9 * 49;
        VAR neg10 = SUBFLOWS.GET(0).RTT/8000 * 162 + SUBFLOWS.GET(0).RTT_VAR/8000 * 266
                + SUBFLOWS.GET(1).LOST_SKBS * 46 + SUBFLOWS.GET(1).RTT/8000 * 26;
        IF (neg10 > pos10) {
                SET(R3,0);
        } ELSE {
                SET(R3, pos10 - neg10);}
        VAR n10 = R3;

        IF (n9>n10){
                SUBFLOWS.GET(0).PUSH(Q.POP());
        } ELSE {
                SUBFLOWS.GET(1).PUSH(Q.POP());
        }
}
```

Listing 8.7: ProgMp Node With Recursive Link Calculation Template

Bibliography

[AC04] Katerina Argyraki and David R. Cheriton. Loose source routing as a mechanism for traffic policies. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, FDNA '04, pages 57–64, New York, NY, USA, 2004. Association for Computing Machinery.

[BFM13] Luca Boccassi, Marwan Fayed, and Mahesh Marina. Binder: A system to aggregate multiple internet gateways in community networks. pages 3–8, 09 2013.

[BHR12] Olivier Bonaventure, Mark Handley, and Costin Raiciu. An overview of multipath tcp. *login Usenix Mag.*, 37, 2012.

[BPB11] Sébastien Barré, Christoph Paasch, and Olivier Bonaventure. Multipath tcp: From theory to practice. In Jordi Domingo-Pascual, Pietro Manzoni, Sergio Palazzo, Ana Pont, and Caterina Scoglio, editors, *NETWORKING 2011*, pages 444–457, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[CAPK+16] Xavier Corbillon, Ramon Aparicio-Pardo, Nicolas Kuhn, Géraldine Texier, and Gwendal Simon. Cross-layer scheduler for video streaming over mptcp. In *Proceedings of the 7th International Conference on Multimedia Systems*, MMSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[FAMB16] S. Ferlin, Ö. Alay, O. Mehani, and R. Boreli. Blest: Blocking estimation-based mptcp scheduler for heterogeneous networks. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 431–439, May 2016.

[Fer] FernandoTorres. Neat implementation on github. *https://github.com/FernandoTorres/NEAT [20.07.2020]*.

[FRE+17] Alexander Froemmgen, Amr Rizk, Tobias Erbshaeusser, Max Weller, Boris Koldehofe, Alejandro Buchmann, and Ralf Steinmetz. A Programming Model for Application-defined Multipath TCP Scheduling. In *ACM/IFIP/USNIX Middleware*, 2017.

[FRHB13] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, IETF, January 2013.

[Gir] Carlos Giraldo. Minievents. *https://github.com/mininet/mininet/wiki/Minievents:-A-mininet-Framework-to-define-events-in-mininet-networks [3.08.2020]*.

[GNM+17] Yihua Ethan Guo, Ashkan Nikravesh, Z. Morley Mao, Feng Qian, and Subhabrata Sen. Accelerating multipath transport through balanced subflow completion. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, pages 141–153, New York, NY, USA, 2017. Association for Computing Machinery.

[HGB+19] P. Hurtig, K. Grinnemo, A. Brunstrom, S. Ferlin, Ö. Alay, and N. Kuhn. Low-latency scheduling in mptcp. *IEEE/ACM Transactions on Networking*, 27(1):302–315, Feb 2019.

[HLMS14] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, Dec 2014.

[HSW89] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989.

[JHKC17] Jonghwan Chung, D. Han, J. Kim, and Chong-kwon Kim. Machine learning based path management for mobile devices over mptcp. In *2017 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 206–209, Feb 2017.

[KWP15] Morteza Kheirkhah, Ian Wakeman, and George Parisis. Multipath-tcp in ns-3. *CoRR*, 2015.

[LNTG17] Yeon-sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. Ecf: An mptcp path scheduler to manage heterogeneous paths. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 147–159, New York, NY, USA, 2017. Association for Computing Machinery.

[LSL19] J. Luo, X. Su, and B. Liu. A reinforcement learning approach for multipath tcp data scheduling. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0276–0280, Jan 2019.

[MM17] Ignaciuk P. Morawski M. On implementation of energy-aware mptcp scheduler. *Borzemski L., Świątek J., Wilimowska Z. (eds) Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems Architecture and Technology*, vol 655.:241–251, ISAT 2017.

[MP90] Warren Sturgis McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 52:99–115, 1990.

[RBP+11] C. Raiciu, S. BarrÈ, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *SIGCOMM 2011, Toronto, Canada*, August 2011. See http://www.multipath-tcp.org for related work on MPTCP and the Linux kernel implementation used in this paper.

[RHW11] Costin Raiciu, Mark Handley, and Damon Wischik. Coupled congestion control for multipath transport protocols. *RFC*, 6356:1–12, 2011.

[RM16] Aditya Rawal and Risto Miikkulainen. Evolving deep lstm-based memory networks using an information maximization objective. *Genetic and Evolutionary Computation Conference (GECCO 2016)*, 2016.

[RPB+10] Costin Raiciu, Christopher Pluntke, Sébastien Barré, Adam Greenhalgh, Damon Wischik, and Mark Handley. Data center networking with multipath tcp. page 10, 01 2010.

[RPB+12] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 399–412, San Jose, CA, April 2012. USENIX Association.

[SBM05] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, pages 653–668, 2005.

[Set] SethBling. Mari/o. *https://www.youtube.com/watch?v=qv6UVOQ0F44 [14.08.2020]*.

[SM02] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, June 2002.

[SWSB13] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. No silver bullet: Extending sdn to the data plane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery.

[WKAH20] K. L. Wallaschek, R. Klose, L. Almon, and M. Hollick. Neat-tcp: Generation of tcp congestion control through neuroevolution of augmenting topologies. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6, 2020.

[XTY+19] Z. Xu, J. Tang, C. Yin, Y. Wang, and G. Xue. Experience-driven congestion control: When multi-path tcp meets deep reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1325–1336, June 2019.

[YMX12] Yu Cao, Mingwei Xu, and Xiaoming Fu. Delay-based congestion control for multipath tcp. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, Oct 2012.

[YWA14] Fan Yang, Qi Wang, and Paul D. Amer. Out-of-order transmission for in-order arrival scheduling for multipath tcp. In *Proceedings of the 2014 28th International Conference on Advanced Information Networking and Applications Workshops*, WAINA '14, pages 749–752, USA, 2014. IEEE Computer Society.

[ZLG+19] H. Zhang, W. Li, S. Gao, X. Wang, and B. Ye. Reles: A neural adaptive multipath scheduler based on deep reinforcement learning. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1648–1656, April 2019.