TECHNISCHE
UNIVERSITÄT
DARMSTADT

PRACTICAL EVALUATION OF NEAT-TCP IN A WIRELESS TESTBED

KAY LUIS WALLASCHEK

Secure Mobile Networking Lab and Project

August 10, 2018

Secure Mobile Networking Lab
Department of Computer Science

Practical Evaluation of NEAT-TCP in a Wireless Testbed
Secure Mobile Networking Lab and Project
Lab Report

Submitted by Kay Luis Wallaschek
Date of submission: August 10, 2018

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisor: Robin Klose, M.Sc., and Lars Almon, M.Sc.

Technische Universität Darmstadt
Department of Computer Science
Secure Mobile Networking Lab

ABSTRACT

The problem of congestion control in computer networks has been addressed by a broad range of algorithms and implementations, most notably a large body of well-known TCP congestion control algorithms, explicit congestion notification, and active queue management. However, many of these methods are not suitable for wireless multi-hop networks as channel disturbances and interferences leading to frame loss may mistakenly trigger congestion mechanisms and hence reduce the network's performance.
One promising novel approach to solving the problem of congestion control in wireless multi-hop networks is NEAT-TCP. NEAT-TCP generates congestion control algorithms in form of neural networks through evolution. However, this approach has only been evaluated in ns3-simulations and has not been tested in real world scenarios. We port a generated NEAT-TCP congestion control algorithm to Linux as a pluggable kernel module and evaluate it on a real testbed consisting of Raspberry Pis. The gathered results indicate that NEAT-TCP can perform as well on the testbed as in ns-3 simulations.

CONTENTS

I INTRODUCTION 1

1 INTRODUCTION 3
1.1 Contributions . . . 3
1.2 Outline . . . 4

2 NEAT-TCP 5
2.1 NEAT-TCP Algorithm . . . 5
2.2 NEAT-TCP Linux Implementation . . . 6
2.2.1 Transform the NEAT Neural Network into Logic Formulars . . . 7
2.2.2 Compile a Congestion Control Kernel Module . 8

II EXPERIMENTS AND RESULTS 11

3 TESTBED 13
3.1 Raspberry Pi 3 Setup . . . 13
3.2 Testbed Locations . . . 14

4 EXPERIMENTS 15

5 RESULTS 17
5.1 Results on the 5x5 Testbed . . . 17
5.2 Results on the 3x3 Testbed . . . 17

III DISCUSSION AND CONCLUSION 21

6 DISCUSSION 23
6.1 Future Work . . . 23

7 CONCLUSIONS 25

IV APPENDIX 27

A APPENDIX 29

BIBLIOGRAPHY 33

LIST OF FIGURES

Figure 1 NEAT-TCP network topology 5
Figure 2 NEAT-TCP neural network 7
Figure 3 Boxplot of fairness on the 3x3 topology in a small auditorium 18
Figure 4 Boxplot of throughput on the 3x3 topology in a small auditorium 18
Figure 5 Boxplot of loss on the 3x3 topology in a small auditorium 19
Figure 6 Small auditorium 29
Figure 7 Big auditorium 30
Figure 8 Free grassland 30
Figure 9 Testbed material 31
Figure 10 Raspberry Pis with aluminium foil to reduce transmit range 32
Figure 11 Compiling kernel on a Raspberry Pi with external cooling 32

LIST OF TABLES

Table 1 Truth table for the output node of the NEAT-TCP neural network 8

Part I

INTRODUCTION

1 INTRODUCTION

TCP in wireless multi-hop networks (WMNs) generally comes with bad performance, especially if there are multiple crossflows in the topology [1]. This problem occurs because of incorrect assumptions of congestion from the congestion control. Whereas congestion control in general would prevent congestion in the network in wired topologies satisfactorily, it does not in WMNs, in particular because the assumption that congestion is indicated when a packet is lost is not sufficient in WMNs, because frame losses on the wireless channel can mistakenly trigger conventional congestion control mechanisms.
Conventional TCP is also not considering fairness. With multiple flows in the network, congestion control can also take care that no flow is overpowering others, thus congesting the network. Conventional TCP is prone to be unfair, as it does not explicitly take fairness into account. Isolated flows are prone to perform better than flows that are interfering with each other because these have to be coordinated or some are overpowering the rest to an extend that the overpowered flows can not send any data.
A novel approach on solving congestion control in WMNs is NEAT-TCP [4]. NEAT-TCP generates congestion control algorithms in form of neural networks through evolution by using the NEAT algorithm [2]. It has been shown that NEAT-TCP outperforms other congestion control algorithms in ns-3 simulations. However NEAT-TCP has not been evaluated in a practical scenario. Thus, in this work, we evaluate NEAT-TCP on a real testbed with Raspberry Pis.

1.1 CONTRIBUTIONS

There are three main contributions of this work:

- Port NEAT-TCP to Linux: Implementation of a NEAT-TCP generated congestion control algorithm on the Linux kernel as a pluggable kernel module.
- Planning and creation of a 5x5 and a 3x3 testbed consisting of Raspberry Pis on multiple locations.
- Evaluating the implemented NEAT-TCP algorithm in respect to other congestion control algorithms and comparison to the results of the ns-3 simulation.

1.2 OUTLINE

This report is structured as follows: Chapter 2 describes how NEAT-TCP works and how it is implemented in Linux. Chapter 3 is about the creation of the nodes for the testbed and gives some insight on the circumstances of the experiments, which are described in Chapter 4. The evaluation of the experiments are presented in Chapter 5. Finally this work is discussed in Chapter 6. Then this work is concluded in Chapter 7.

2 NEAT-TCP

2.1 NEAT-TCP ALGORITHM

NEAT-TCP uses the NEAT algorithm to generate congestion control algorithms. The NEAT algorithm uses evolution to evolve neural networks that can solve priorly defined problems. It uses mutation, speciation, mating and elimination to evolve in the direction of solving a problem. By giving each neural network or organism a fitness measure on how good the problem is solved, it can be evolved in the specific optimisation goal.
In the case of one of the experiments in the NEAT-TCP thesis, the fitness measure is a linear combination of throughput and fairness achieved in a network simulation, with in this case a network topology like in Figure 1, which we rebuilt with Raspberry Pis. NEAT-TCP then produces neural networks like the one in Figure 2. For a deeper explanation on how and why NEAT-TCP works, the thesis about it should be consulted.

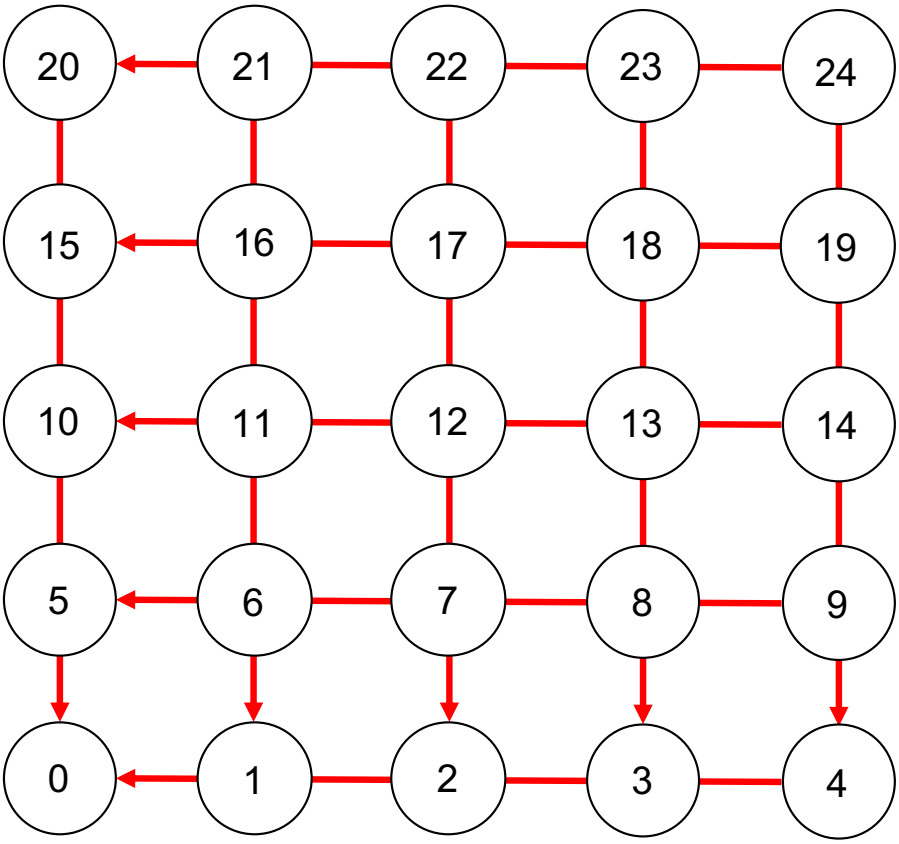

Figure 1: NEAT-TCP network topology

2.2 NEAT-TCP LINUX IMPLEMENTATION

The linux kernel has the option of pluggable congestion control modules. This means that a congestion control algorithm can be compiled for itself and then plugged into the running kernel, without recompiling the whole kernel or rebooting the system. So the aim of this section is to create such a pluggable module for NEAT-TCP.
The structure of such a module is strictly predefined. Every congestion control algorithm needs to have the following functions:

- `.ssthresh`
- `.cong_avoid`

As these names suggest, both are used in the traditional congestion control with slow start and congestion avoidance. The underlying structure of TCP is using this procedure and without completely rewriting TCP in the socket, it has to be coped with. To disable `.ssthresh`, this function just returns the current congestion window, so it has no effect. The `.cong_avoid` just evaluates the NEAT neural network and sets the congestion window accordingly.
For NEAT-TCP to work there needs to be more functions than these two, namely `tcp_neat_eval`, `.init`, `.pkts_acked`, `.cwnd_event`, `.undo_cwnd`, `.in_ack_event` and a struct `neat`. Note that `tcp_neat_eval` has no dot at the beginning, as this is a function defined only for the kernel module, where as the others are implementing existing kernel functions. In the following all of these functions are explained on when they are called and what they do:
Structure `neat`: This struct holds the variables for the NEAT-TCP algorithm. There the *last activation*, *duplicate acknowledgement indicator*, *timeout indicator* and the *last packet acknowledged* is saved.
`.init`: Simple initialisation function to set the struct `neat` to 0 and the congestion window to 1.
`.pkts_acked`: This function is called every time a packet is acknowledged. It provides information on the packets that are acknowledged so it can be seen if it is a duplicate acknowledgement. For that it is tested against the *last packet acknowledged* in the `neat` structure. If it is the same, then the *duplicate acknowledgement indicator* is set to 1. If not the *duplicate acknowledgement indicator* is set to 0 and the *timeout indicator* is also set to 0 as the connection seems stable.
`.cwnd_event`: This function is called every time a congestion event is happening. In this case, only the event `CA_EVENT_LOSS` is interesting, as it indicates that a packet is lost. If this event happens the *timeout indicator* is set, as timeout basically indicates a loss and there is no way to easily get a timeout in the kernel.
`.undo_cwnd`: For this function it is not clear, when it is called. It should be called if the connection is set as broken and the connection needs to be setup again. Because the ns-3 simulator has no similar function,

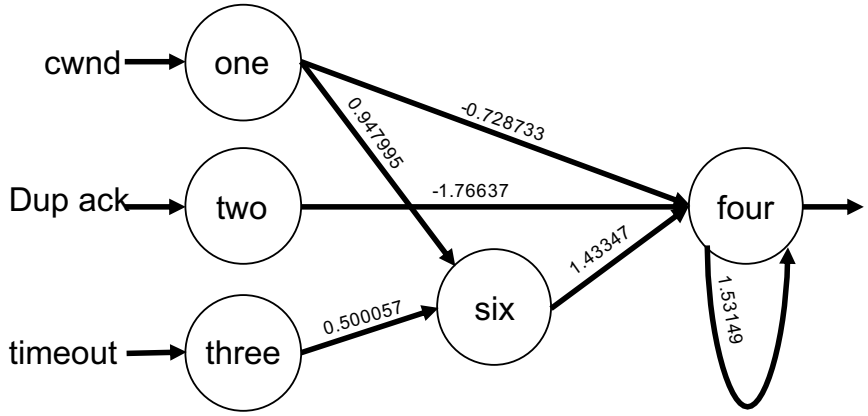

Figure 2: NEAT-TCP neural network

this function is set to just return the current congestion window, to negate its effect it would have.

`.in_ack_event`: This function is called whenever an acknowledgement arrives at the sender. This is similar to the `.pkts_acked` function and evaluates the NEAT neural network and sets the congestion window accordingly.

`tcp_neat_eval`: This function incorporates the NEAT neural network. One problem with the kernel modules is, that they have no rights to access the FPU of the CPU, so there is no way to calculate with floats. But the weights of the neural network are floats. Luckily the activation function is binary (either 1 if the input is > 0 or 0 otherwise) a truth table can be setup.

2.2.1 *Transform the NEAT Neural Network into Logic Formulars*

As no congestion control module has access to the floating-point units (FPUs) the neural network could be translated into logical formulas. This is possible as the activation function is binary (either 1 if the input is > 0 or 0 otherwise). Only node six and node four need to be translated into logical formulas, as for the others it is trivial as they only have one input. Node six, as shown in Figure 2, can be defined as the function:

$$\text{node}_6 = 0.500057 * \text{node}_3 + 0.947995 * \text{node}_1$$

Or as logical:

$$\text{node}_6 = \text{node}_1 \vee \text{node}_3$$

For node four it is not as easy. The mathematical function is:

$$\text{node}_4 = -0.728733 * \text{node}_1 - 1.76637 * \text{node}_2 + 1.43347 * \text{node}_6 + 1.53149 * \text{node}_{4,\text{recurr}}$$

To translate this mathematical function a truth table is constructed as shown in Table 1.

Table 1: Truth table for the output node of the NEAT-TCP neural network

one	two	six	last act.	result
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

This translates to:

$$\text{node}_4 = (\neg\text{node}_2 \wedge \neg\text{node}_6 \wedge \text{node}_{4,\text{recurr}}) \vee (\neg\text{node}_2 \wedge \text{node}_6 \wedge \text{node}_{4,\text{recurr}})$$

$$\vee (\neg\text{node}_2 \wedge \text{node}_6 \wedge \neg\text{node}_{4,\text{recurr}}) \vee (\text{node}_2 \wedge \text{node}_6 \wedge \text{node}_{4,\text{recurr}})$$

And further condensed to:

$$\text{node}_4 = (\neg\text{node}_2 \wedge (\text{node}_6 \vee \text{node}_{4,\text{recurr}})) \vee (\text{node}_2 \wedge \text{node}_6 \wedge \text{node}_{4,\text{recurr}})$$

With these logic formulas, NEAT-TCP can be implemented as a pluggable congestion control kernel module.

2.2.2 *Compile a Congestion Control Kernel Module*

Kernel modules are written in C and are compiled with the whole kernel. So to compile kernel modules the currently running kernel needs to be downloaded and compiled. After that `lib/modules/` should be available for the current kernel. Because the kernel is already compiled, it does not take a long time to compile the kernel modules. The command for compiling kernel modules is:

```
$ make -C /lib/modules/$(shell uname -r)/build M=$(pwd) modules
```

This creates the needed `.ko` file for the kernel and this can be plugged in with this command:

```
$ sudo insmod tcp_neat.ko
```

And a congestion control can then be enabled with setting its name in sysctl:

```
$ sudo sysctl -w net.ipv4.tcp_congestion_control=neat
```

Part II

EXPERIMENTS AND RESULTS

3 TESTBED

3.1 RASPBERRY PI 3 SETUP

General Setup: At first each Raspberry Pi 3 needs an operating system. For that Raspbian [3] is the obvious choice, as it is optimised for Raspberry Pis. After installing Raspbian on all the Pis `ssh` should be enabled via the raspbian config:

```
$ sudo raspi-config
```

All Pis that are sending, need to be able to use the NEAT-TCP kernel module and for that the kernel needs to be upgraded:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

At this point `iperf3` should be installed, which is used for the experiments.

```
$ sudo apt-get install iperf3
```

Network Setup: To setup an ad-hoc network the file `/etc/network/interfaces` needs to be altered:

```
$ sudo nano /etc/network/interfaces
```

To:

```
# interfaces(5) file used by ifup(8) and ifdown(8)

# Please note that this file is written to be used with dhcpcd
# For static IP, consult /etc/dhcpcd.conf and 'man dhcpcd.conf'

# Include files from /etc/network/interfaces.d:
source-directory /etc/network/interfaces.d

auto lo
iface lo inet loopback

iface eth0 inet dhcp

auto wlan0
iface wlan0 inet static
  address 10.2.1.n
  netmask 255.255.255.0
  wireless-channel 1
  wireless-essid PiAdHocNetwork
  wireless-mode ad-hoc
  wireless-txpower p
up route add -host 10.2.1.k gw 10.2.1.l dev wlan0
```

Where n is the number of the node + 1, as 0 is not possible for obvious reasons. `Wireless-txpower` denotes the power of the transmission as *dbm*. This has to be found out by testing several p to satisfy the constraints needed. The `up route` row enables static routing, here k stands for the target node + 1, and l stands for the next hop that should be taken. For example, consider Figure 1 and node 20. There these two lines should be added:

```
up route add -host 10.2.1.25 gw 10.2.1.22 dev wlan0
up route add -host 10.2.1.1 gw 10.2.1.16 dev wlan0
```

This has to be done for all nodes in the testbed accordingly. Note that the routes need to be setup in both directions, as to work with TCP a bidirectional channel is needed. Also note that these are only the routes that are interesting for the experiments to be conducted in this work. For other routes not mentioned, there needs to be an entry, or in general a routing algorithm could be used like Optimized Link State Routing (OLSR). However, static routing is satisfying for this work.
The nodes can not simply forward these packages unless it is enabled. To enable ip-forwarding this line needs to be added to the file `/etc/sysctl.conf`:

```
net.ipv4.ip_forward = 1
```

At this point the testbed itself is finished. All nodes are in the same network, can route to their destinations and forward the packages.

3.2 TESTBED LOCATIONS

To be as close to the simulation as possible, a location that is isolated from other wireless signals should be chosen. The simulation scenario is further explained in Chapter 4. Also the field should be even, without too much of height differences. It is not feasible to build the simulation network in real life, as a 2km^2 free field would be needed and there is currently no possibility to use one. Also other nodes as Raspberry Pis would be needed, as Raspberry Pis do not have a 500 m transmission range. However, the topology can be built in smaller scale, as it is only important that the nodes are not capable of communication with their diagonal partners.
For this work three locations have been chosen. (I) a free grassland, (II) a big auditorium, (III) at home and (IV) a smaller auditorium. For (I), (II) and (III) there are some pictures in the appendix.
The Raspberry Pis for all locations have been covered in aluminium foil, to reduce their transmission range. This has been done by carefully wrapping each Pi so that at every point there is only one layer, as two layers shut down wireless communication completely. A picture of the wrapped Pis can be found in the Appendix.

4 EXPERIMENTS

In this work, several TCP congestion control algorithms were evaluated on a 5x5 and 3x3 testbed. The tested congestion control algorithms are: Bic, Vegas, Cubic, Hybla and NEAT-TCP [4]. On top of that these are also evaluated in the ns-3 network simulator. For the evaluation of one congestion control algorithm consider Figure 1, at a certain point in time all flows start and are sending for 300 seconds. For the 3x3 topology the flows are constructed according to the 5x5 topology Figure 1, so 3 from top to bottom and 3 from the right to the left.
To coordinate the flows `crontab` is used. With `crontab` it is possible to execute commands or scripts at a certain time. For that to work, all Raspberry Pis need to be set on the same time. As these have no real-time clock or batteries they can not maintain the current time between boot ups. To solve this a script can be used that sets the time to a point after the experiments should have happened. For example, if the experiments are conducted between 12:00 and 14:00, the script could set the time to 15:00, so there is no possibility for the experiments to start too early.
With this command the time can be set:

```
$ sudo date -s 15:00
```

To set the time at start up with crontab this command needs to be saved in a `.sh` file with a leading:

```
#!/bin/bash
```

And made executable with the command:

```
$ sudo chmod +x time.sh
```

To execute this at start up the script needs to be added to the `crontab` file with the `@reboot` flag.

```
$ sudo crontab -e

@reboot /home/pi/time.sh
```

Also the congestion control modules should be loaded at startup with this script:

```
#!/bin/bash
$ sudo insmod /home/pi/Desktop/congestion_control_modules/
    tcp_bic.ko
$ sudo insmod /home/pi/Desktop/congestion_control_modules/
    tcp_hybla.ko
$ sudo insmod /home/pi/Desktop/congestion_control_modules/
    tcp_vegas.ko
```

```
$ sudo insmod /home/pi/Desktop/congestion_control_modules/
    tcp_neat.ko
```

Note that the path needs to correspond with the correct path on your drive.
To start a flow, this script is used:

```
#!/bin/bash
$ sudo sysctl -w net.ipv4.tcp_congestion_control=hybla
$ iperf3 -c 10.2.1.4 -V -i 1 -t 300 --logfile /home/pi/Desktop/
    output/hybla_23-to-3.log
```

This command sets the correct congestion control algorithm and runs `iperf3` in verbose mode for 300 seconds and logs the output in 1 second intervals to a file.
The finished `crontab` file for node 23 could look like this

```
@reboot /home/pi/Desktop/insert_modules.sh
@reboot /home/pi/Desktop/23/time.sh
15 12 * * * /home/pi/Desktop/23/neat_23to3.sh
21 12 * * * /home/pi/Desktop/23/neat_23to3.sh
27 12 * * * /home/pi/Desktop/23/neat_23to3.sh
33 12 * * * /home/pi/Desktop/23/bic_23to3.sh
39 12 * * * /home/pi/Desktop/23/bic_23to3.sh
45 12 * * * /home/pi/Desktop/23/bic_23to3.sh
51 12 * * * /home/pi/Desktop/23/cubic_23to3.sh
57 12 * * * /home/pi/Desktop/23/cubic_23to3.sh
3 13 * * * /home/pi/Desktop/23/cubic_23to3.sh
9 13 * * * /home/pi/Desktop/23/hybla_23to3.sh
15 13 * * * /home/pi/Desktop/23/hybla_23to3.sh
21 13 * * * /home/pi/Desktop/23/hybla_23to3.sh
27 13 * * * /home/pi/Desktop/23/reno_23to3.sh
33 13 * * * /home/pi/Desktop/23/reno_23to3.sh
39 13 * * * /home/pi/Desktop/23/reno_23to3.sh
45 13 * * * /home/pi/Desktop/23/vegas_23to3.sh
51 13 * * * /home/pi/Desktop/23/vegas_23to3.sh
57 13 * * * /home/pi/Desktop/23/vegas_23to3.sh
```

Note that `15 12 * * *` runs the command at 12:15. This is a setup to test all congestion control algorithms for 300 seconds 3 times with a 1 minute pause in between.

5 RESULTS

The results chapter is divided into two parts. In the first part the results on the 5x5 testbed are described and in the second part the results on the 3x3 testbed are described.

5.1 RESULTS ON THE 5X5 TESTBED

Sadly the experiments on the 5x5 testbed did not work out due to unexpected technical problems.
Free grassland: We only had access to the free grassland for one day and at that day it was a really hot day. While we setup the testbed, with tests if the routes work, it has been noticed, that routes that worked at the start of the setup, did not work a little while after. This is because, the hotter the Raspberry Pis got, the lower is their transmission range. After about 2 hours in the hot summer sun the Pis in aluminium foil were nearly untouchable, because they heated up too much. Then this experiment has been aborted and the results gathered are not valuable.
Big Auditorium: At first this location was more promising. Especially because there are seating rows which makes measuring easier. But the height difference is too high. Between one line of nodes there is about 1 m of height difference. The first results gathered were nothing like in the simulator. So it has been tried again, but testing if the topology of the simulator has been satisfied. There it has been found out that the topology was quite different. For example, consider Figure 1, node 8 had a better connection to node 2 than to node 7. There was no possibility to recreate the simulator topology, because for that the diagonal nodes have not to be reached. As there is a better connection to the diagonal nodes than to the direct partners this location has been abandoned. Two of possible reasons for this behaviour could be, that (I) something in the seating row has some electrical radiation which pollutes the direct link, and (II) the transmission is reflected at the walls of the auditorium.

5.2 RESULTS ON THE 3X3 TESTBED

After the bad results of the 5x5 testbed the 3x3 testbed has been build. As in the thesis about NEAT-TCP, there are also results for this generated congestion control algorithm in a 3x3 simulation. At first this 3x3 has been tried at home with only 1 m distance between nodes. The results looked promising, that if the distance would be higher

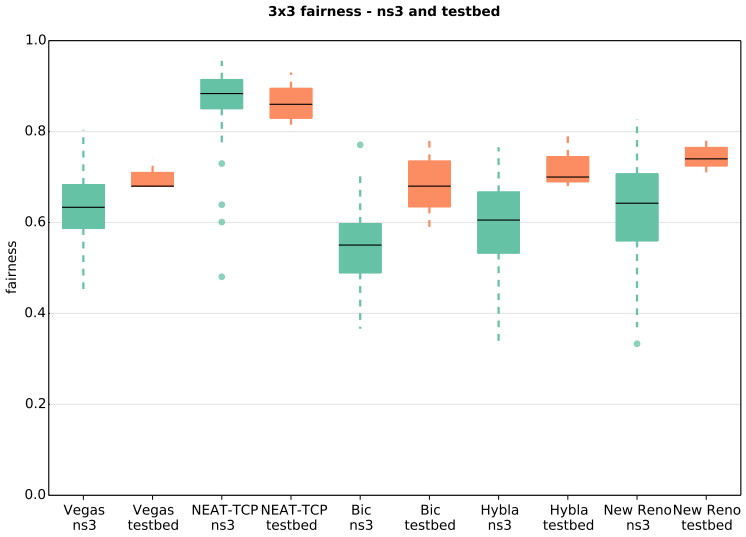

Figure 3: Boxplot of fairness on the 3x3 topology in a small auditorium

the results could be close to the results of the simulator. Then in a smaller auditorium with less height difference and no electronics in the seating rows the 3x3 topology has been built with 3 m distance between nodes. The results can be seen in Figure 3. Only fairness is shown, as throughput is different to the simulations, as the bitrate can not be restricted in `iwconfig`. Also delay could not be tested with `iperf3`. So fairness is the only measure that is meaningful. In Fig-

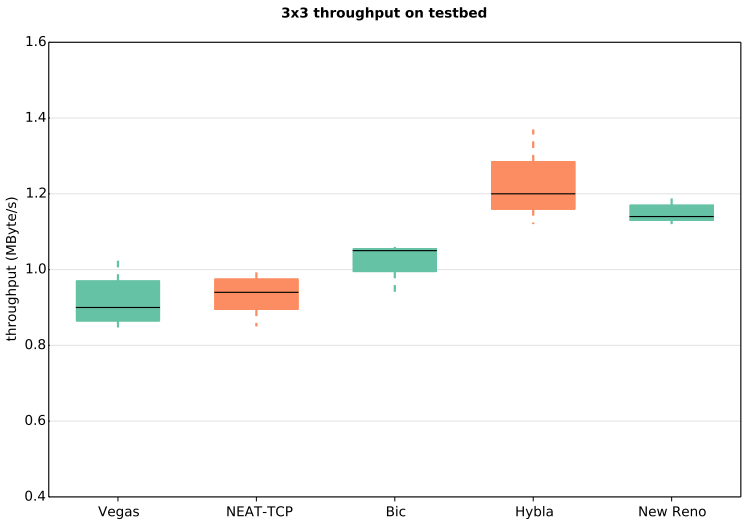

Figure 4: Boxplot of throughput on the 3x3 topology in a small auditorium

ure 3 it can be seen, that all congestion control algorithms, besides NEAT-TCP, performed slightly better on the testbed than in the simulator. But all of them are in the range of the results of the simulator.

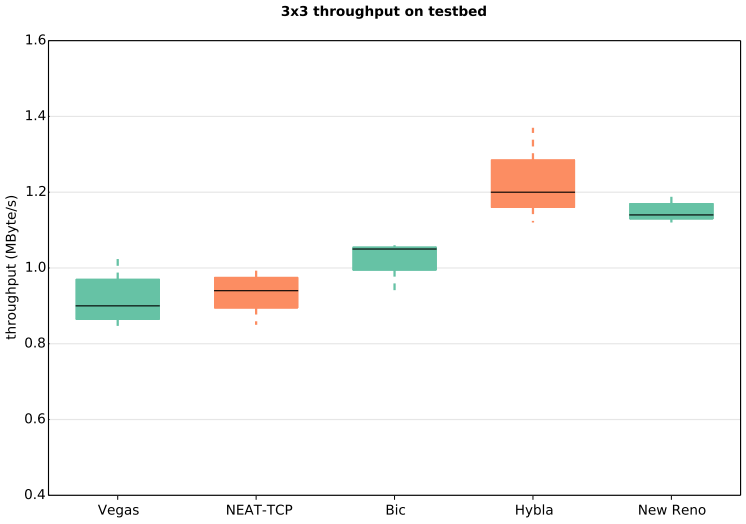

Figure 5: Boxplot of loss on the 3x3 topology in a small auditorium

Note that there are only 3 testbed values and 100 simulator values. NEAT-TCP performed like it was expected, as it is in the range of the box.
As the transmission range of the Raspberry Pis can not be restricted the throughput can not be compared to the ns3 results. The loss can be seen in Figure 5. Note that NEAT-TCP with the highest loss is on par with Vegas with the lowest loss in terms of throughput. It indicates that NEAT-TCP is pushing the limits of what the network can handle, while providing good fairness. Compared to Hybla, which has the highest throughput and low loss, NEAT-TCP provides much more fairness. These results are further discussed in Chapter 6.

Part III

DISCUSSION AND CONCLUSION

6 DISCUSSION

As mentioned in Chapter 5 all results are comparable to the ones from the simulator. This means, that the port of NEAT-TCP is close enough to the one of the simulator. Also, this shows that NEAT-TCP does not only work in the simulator but also on a real testbed. In return this means that the ns-3 simulator is also close to reality, as all tested congestion control algorithms are close to each other. The small difference of the boxes could be because these are only 3 values. To be completely sure that these are the same, one should do 100 runs on the testbed for all algorithms. But this would take a lot of time, about 60 hours.

Why did the 5x5 testbed not work? As mentioned in Chapter 5 the 5x5 testbed did not work. The main reason why it did not work on the free grassland was heat. It turned out that, as Raspberry Pis overheat, the transmission power suffers. The free grassland is still the most promising location, but it should be done in the winter, or at a day where it is not hot and the sun does not shine, so that the Raspberry Pis do not overheat. The problem of the big auditorium was most likely the height difference and the walls. Also there is another WLAN which could interfere, but there were no connections happening at the time of testing, so the influence should be minimal. The reflections of the walls in combination with the height difference is a worse influence than the other WLAN, as this prevents the creation of the constraint that the diagonal nodes can not be reached.

6.1 FUTURE WORK

The results are quite promising, that the implementation of NEAT-TCP is sufficient. So it would be the best to test with a different testbed configuration. Meaning, test again on the free grassland but with better weather conditions. This location provides no walls where the waves could be reflected, a nearly even area, so no height difference and no other WLAN interferences. It could not be done within this work, as there was only access for the one day with the too sunny weather. What could also be done, is the generation of another NEAT-TCP algorithm for topology of the small auditorium. This is not quite easy, as the height difference and walls need to be implemented.

7 CONCLUSIONS

Evaluating the novel approach of evolving neural networks to perform congestion control of NEAT-TCP has been hereby done for the first time. The gathered results show that NEAT-TCP can perform as well on the 3x3 testbed as in the simulations, which gives the implication that ns-3 simulations are enough to show how algorithms perform, without building a real testbed. In this work, we build a wireless multi-hop testbed with Raspberry Pis to compare the performances of NEAT-TCP in a real world scenario. Also the generated NEAT-TCP congestion control algorithm is ported to the Linux kernel as a pluggable kernel module. The thesis around NEAT-TCP only considers simulation results and left out the evaluation on a real world scenario. This work describes how this testbed is constructed and compares the result with the ones from the ns-3 network simulator. Multiple experiments are done on multiple locations with differing characteristics.

Part IV

APPENDIX

A

APPENDIX

In this chapter some pictures that were made in the process of the experiments are shown.

Figure 6: Small auditorium

Figure 7: Big auditorium

Figure 8: Free grassland

Figure 9: Testbed material

Figure 10: Raspberry Pis with aluminium foil to reduce transmit range

Figure 11: Compiling kernel on a Raspberry Pi with external cooling

BIBLIOGRAPHY

[1] Sumit Rangwala, Apoorva Jindal, Ki-Young Jang, Konstantinos Psounis, and Ramesh Govindan. "Understanding congestion control in multi-hop wireless mesh networks." In: *MobiCom 08, Proceedings of the 14th ACM international conference on Mobile computing and networking* (2008), pp. 291–302.

[2] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks Through Augmenting Topologies." In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. URL: http://nn.cs.utexas.edu/?stanley:ec02.

[3] Mike Thompson and Peter Green. *Rasbian*. URL: https://www.raspbian.org (visited on 06/28/2018).

[4] Kay Luis Wallaschek. "NEAT-TCP: Generation of TCP Congestion Control through Neuroevolution of Augmenting Topologies for Wireless Multi-Hop Networks." In: (2018), Bachelor Thesis, SEEMOO, Technische Universität Darmstadt.