

Advanced Topics in Communication Networks (Fall 2019)

Group Project Report

NetChain: Scale-Free Sub-RTT Coordination

Authors:

Haoyu Zhu

Francisco Dumont

Advisor: Thomas Holterbach

Supervisor: Prof. Dr. Laurent Vanbever

Submitted: Dec 16, 2019

15 Pages

Abstract

This report focus on the p4 implementation of the paper "NetChain: Scale-Free Sub-RTT Coordination"[4] published in the 15th USENIX Symposium. For the implementation of this paper a virtual machine with linux was used, which had all necessary python, p4, c and mininet tools. NetChain is an in-network solution for coordination services. By coordinating within the network, the Sub-RTT system can reduce query latency, maintain higher throughput, can be scaled easily and provide consistency in the presence of failures. The implementation will be explained in three different levels: Host, Controller and Switch Data Plane. The host was implement in c, most of the controller in python and the data plane in p4. As the implementation is done in a virtual machine, the evaluation aspects are limited. Despite the limitations, the evaluation focus on the correct functionality of the different operations, protocols and the time interval that the throughput is affected by a failure and its recovery.

Contents

1	Introduction	1
1.1	Goal	1
1.2	Paper summary	1
1.3	Paper review	2
1.4	Mayor Challenge	2
1.5	Applied consistent hashing concept in the protect	2
1.6	Applied recovery concept in the protect	3
2	Implementation	3
2.1	Basics and definitions	3
2.2	Host Implementation	4
2.2.1	Chain generating and initialization	4
2.2.2	Packet generating and capturing	4
2.2.3	Updating and scaling	4
2.2.4	Evaluating	4
2.3	Controller Implementation	5
2.3.1	Initialization	5
2.3.2	Insert and delete operation handler	5
2.3.3	Fast fail-over handler	6
2.3.4	Fast recovery handler	6
2.4	Switch Data Plane Implementation	7
2.4.1	Parsers and Headers	7
2.4.2	General view of the Ingress Processing	8
2.4.3	Forwarding processor	8
2.4.4	NetChain packet processor	9
2.4.5	Fast fail-over processing	9
2.4.6	Fast Recovery processing	9
3	Evaluation	10
3.1	Functionality	10
3.2	Performance	11
3.3	Robust	13
4	Conclusion	14
4.1	Work summary	14
4.2	Future works and expectations	14
	References	16
A	Group organization	I

1 Introduction

Networks are about to experience mayor changes with the introduction of programmable switches. This new tool open the possibility to solve current challenges in a more efficient, fault-tolerant, consistent and innovative way. One of those challenges is the how to increase the performance of coordination systems, which up to this day use a server-based solution. Programmable Switches open the possibility to use an alternative approaches by using the switches in the network as data storage. This take advantage of how fast switches can process packets in comparison with a traditional solution.

1.1 Goal

The goal of this implementation is to write the required code to have a working NetChain system in a virtual machine. The implementation tries to stay in line with the paper as much as possible, but some implementation decisions had to be made in order fulfill some aspects that weren't explicitly stated in the paper.

1.2 Paper summary

To start, two mayor ideas of protocols have to be recognised: Steady state and reconfiguration protocol. The first one deals with read and write operations of data, which is mainly performed in the data plane. The second protocol will use the network control plane for reconfiguration operations like joining (insert a new key), leaving (delete a key), failure behaviour behaviour and scalability.

The First building block of the NetChain system is the concept of 'Chain Replication', which can be described as a virtual nodes organized in a chain structure with a head and a tail. This chain of virtual nodes is placed along different real switches and the key-value information is replicated in each one of them. The next building block in the system are the virtual nodes themselves. These are an abstraction used to evenly spread the data storage between the real switches. The virtual nodes with the help of consistent hashing are mapped to a hash ring, meaning that each node is responsible for a certain amount of hashing values. A mapping of virtual nodes and real switches has to be made.

NetChain uses an additional packet structure implemented in the UDP payload, which is shown in figure1. The packet is organized in the following order: First, a count of how many switches the packet still has to go through, followed by the IP addresses of those switches. Then the operation to be performed. Third, a sequence number for the key. And Finally the key and its particular value. The IP addresses chain only shows the ordered path of switches the packet has to visit, but the routing between two of them is left to the IP protocol. This means, in the IPv4 protocol the destination address is updated depending on the next switch in the chain. Thus, once the destination is reached, the counter is reduced by one and the next IP address in the chain is updated to the IPv4 protocol. The operations described in the paper are insert, delete, write and read. Inserting a key requires the controller to assign a register index to that key in each switch and keep track of it in all the virtual nodes it is assigned to. The delete action also needs the controller to act so that it can free the memory register in the switches and erase the key from the virtual nodes. On the other hand , the write and read operation doesn't need any controller involvement. While write operations will travel trough each one of the switches in the chain starting from the head, the reading will only go to the tail to, get the value and sent it back to the host.

The next component in the packet structure is the sequence number. This helps to keep the saved values of each key updated in the correct order. Once a packet arrives at the designated switch, the sequence number is compared. If it is lower than the one saved in the switch the

Count	S0	...	Sn	OP	Seq	Key	Value
2Byte	4*Count Byte			2Byte	4Byte	2Byte	8Byte

Figure 1: NetChain packet structure used in our project

packet will be dropped, otherwise the value is updated and the packet forwarded to the next destination.

In the case a certain switch fails, two processes can be described. This first is fast-failover, here the controller quickly reconfigures the neighbor switches of the failed one in a way to bypass it and continue to the next IP address in the chain or send it back to the host if the failed switch was in the tail. The second procedure is failure-recovery. The goal of this procedure is to replace the failed switch with another. In this regard the controller has to perform two mayor things. To start it has to reconfigure the mapping of virtual nodes and real switches. Second, insert the needed keys into the new mapped switches and once this is done the fast failover state can be lifted, so that the packets can travel through the whole chain.

1.3 Paper review

The Paper was not easy to read. In some cases, we had some issues figuring out what the the authors really wanted to be implemented. One of the factors that contributed to this 'issue' was certainly some lack of knowledge about certain topics. The authors assume those topics to be known by the readers. Despite that, the paper is well written and with patience and some help we were able to understand it fully.

The authors present the results of their implementation in real switches. That is very useful to see the real scope of what an impact the system can have. In our implementation, of course, we won't be able to compare many of those same factors.

1.4 Mayor Challenge

One of the aspects we struggle a lot with, was the relationship between the consistent hashing, virtual nodes, the mapping with the real switches and the assignment of the of switches on the chain. The way to implement it, is not explicitly mentioned in the paper, there are only two references linked to the virtual nodes and consistent hashing: [1] and [3]. However, the idea was inspected several times, with the help of Thomas Holterbach, till we arrived at a implementation concept.

1.5 Applied consistent hashing concept in the protect

Each key is hashed by the method Md5 and the values are modulated to generate a hashing ring. In this hashing ring all the virtual nodes, 1024 in our case, will be equally distributed. The modulated hash values between two nodes will be assigned to the virtual node that is on the clock wise limit side of the hash ring arc. This assigned virtual node would be the reference starting the chain. Then also following a clock wise direction from the reference node, virtual nodes will be added to the chain till the total length of the chain is reached and no switch is repeated.

As the virtual nodes will be mapped to a real switches randomly, each key will have a different chain of real switches. For this reason our implementation will need to save this chain assigned to each key in a dictionary.

The consistent hashing is implemented in the controller and in the host, but the mapping between switches and nodes is given by the controller to the host by a cvs file.

1.6 Applied recovery concept in the protect

The recovery in our implementation looks what active switches are available to be used as replacement for the switch that has failed. Previously it was mention that each key has a different chain due to the distribution of nodes in the hashing ring and the mapping they have with real switches. Thus when one switch fails, the mapping of nodes and switches has to be recomputed. The new mapping will also have a random factor, because we don't want to assign the same switch to all those nodes, otherwise the distribution of load would not be equal. With this concept, adding a new switch to the NetChain system won't be a problem, only a reassignment of switches to nodes is needed.

2 Implementation

2.1 Basics and definitions

During the implementation section this `writing style` with specific colors will be used in order to point out some specific attributes. The attributes are the following: `cyan` for anything that has a relationship with the host implementation; `red` for anything that mainly belong to the p4 implementation code; `violet` for anything that mainly is implemented for or in the controller; and `brown` for general concept, names, commands or descriptions that are important to note, but don't belong exclusively on one of the three part of the implementation.

The topology used in our implementation has 5 switches and two hosts. The name of each one of them is described respecting the following abbreviation ' `sX` with X in 1,2,3,4,5 and `hY` with Y in 1,2 respectively. The hosts will be connected to the first switch `s1` and the rest of them will be mapped with the virtual nodes. All the switches are connected in a full mesh manner, thus regardless of what switch happens to fail, all the others will be reachable.

The strategy used to create the network is `13`, but despite that a auxiliary IP will be used to refer a specific switch. This Auxiliary IP will be used as the real IP of the switch for any purpose, thus the assigned IP from the `13` won't be really used. In our case that auxiliary IP is `20.0.X.0` with X in 1,2,3,4,5.

With this in mind, our implementation can be separated in three topics: Host, Controller and Switch Data Initialization. The Host is the one who will send all the keys with their value towards the network. The host has two threads, one of them sending packet and one for sniffing. The implementation of the host is done in c because python sniffing and sending threads weren't working properly when short intervals of time between them were needed.

The controller is the one that will be responsible for several tasks. Among them is the filling of all tables, the mapping of virtual nodes with real switches and the behavior of the system during the processes of fast-failover and recovery. The controller is mostly implemented in python also with different threads. One thread is always sniffing packets coming from the interfaces between the controller and the switches. The second thread is waiting for a switch input, which will indicate that this switch has failed. From there the fast-failover process can start. At this point the System will be functional, but the controller thread will be waiting again for the input of the failed switch, but this time the recovery process will be executed. The host will receive from the controller a csv file in which the mapping of the virtual nodes and the real switches is defined. Of course this does not reflect a real case scenario, but this is enough for the simulation done in the virtual machine.

And lastly the data switch plane is implemented in p4 language. In this plane no involvement of the controller is needed when it comes to the read and write operations. Though all the table matches are set by the controller of course.

2.2 Host Implementation

The host code is written in C++, including several files implementing the functionality of NetChain initialization, Packet generating and receiving, and updating from controller. The main entrance of host file is `main()` function located at [send.cpp](#)

2.2.1 Chain generating and initialization

As is talked before, the host and the controller uses a consistent hashing method to keep chain synchronized. The hashing algorithm used here is MD5(Message Digest 5)[2]. MD5 have a good distribution performance when doing consistent hashing. When host starts, It will load the virtual node-physics switch assignment from a file named "`nodes.csv`". All the communication between controller and host is only this file. When the node assignment is loaded, the host will calculate the switch IPs using what we assigned to switch before. Now the host system is established and waiting for commands.

2.2.2 Packet generating and capturing

When a command is inputed: `Insert`, `Delete`, `Write` or `Read`, host will do the hashing of the key to obtain the switch that belongs to the chain. This is done by obtain the hashing of the key¹, and mod it by 2^{32} , then divide by 2^{22} to find the virtual node that it belongs to. After that, the host will find the chain along the hashing ring, to find a chain contains all different switches.

For simplicity, we in this part will save the chain in program, which should not be a common way in practice. The packet will then generated with a NetChain Structure and send by using socket. When packet is successfully sent, it will switch to sniffer mode and receive the packets from outbound port. If the return packet contains operation "Finish", the queue will be considered successful.

2.2.3 Updating and scaling

At the command input stage, `update` command can be used to check system status and update the node assignment for scaling. Here we divide scaling into two parts: Adding fault tolerance or Adding storage size.

To add fault tolerance ability to netchain, it equals to add one node to any chain. i.e. increase `f` by one. At host it will be simply modify the number of `f` and everytime the packet will be generated contains a new chain.

To add the storage ability to netchain, it equals to add one unused switch to the topology and simply assigns some virtual nodes to it, after that, the process is more likely to do a recovery: copy the value and entry from a previous node or following nodes. In host, it is better to update the node assignment for this scenario. But if it don't, the recovery strategy will make sure it reaches new switch correctly.

2.2.4 Evaluating

The host also contains a evaluation program, when enter `evaluation`, the evaluation program will be activated. There are four possible testing commands in evaluation: `insert`, `write`, `read` and `thrp`. Basically the insert command will insert a number of keys to netchain. Write and read will continuously do a write/read operation for a number of seconds. And `thrp` will try to start several threads and send packets randomly to evaluate the throughput.

¹<https://github.com/JieweiWei/md5>

2.3 Controller Implementation

The controller is connected to every switch and is responsible for the initialization of the system, the mapping of virtual nodes with real switches, writing/erasing of matches in the tables, the activation of temporary behaviour during a failure and its recovery process. The controller is composed mainly of four scripts: `NetChain-controller.py`, `routingcontroller.py`, `hashing.py` and `queue.py`. The last script doesn't need much explanation. It is just important to note that in order to receive, send and parse the packets correctly between the mininet and the controller, the `scapy` library of python is used. This library also has available tools to create our own layer, in our case a NetChain packet. And this is implemented in the script `queue.py`.

2.3.1 Initialization

To start the controller, the script `NetChain-controller.py` has to be started in a terminal window by using the command `sudo python NetChain-controller.py`. The script will start by initializing several variables, a class object `consistent_hashing` named `self.hashing`, saving the mapping between virtual nodes in real switches into the file `nodes.csv` (only done once for the host) and executing the function `init()`.

Among the variables initialized are: `self.sw_memory_key_index` a dictionary of lists that will keep track of the keys saved in a particular switch. The positions number of the key represents also the index position in the switch registers; `self.node_keys` a list of list that keeps track of what keys are assigned to which virtual node; `self.ip_addr` a dictionary that saves the IP addresses used for each one of the switches with its respective abbreviation; and `self.keys` dictionary which stores the what chain of switches is assigned to a particular key.

It is also worth mention how `self.hashing` is initialized. In the script `hashing.py` the number of virtual nodes, number of real switches and the `f` parameter are saved. Consequently the function `assign_virtual_node_to_sw()` of this object is applied. This function receives the list of active switches from the `NetChain-controller.py` script and what it does is assign randomly virtual nodes to switches and save it in two different ways. The first one is a list `self.list_node_sw` with one switch abbreviation on each element, this represents the switch that is assigned to the virtual node (The index in the list represents the node). The other format is a dictionary `self.list_sw_node`, which all the nodes that are assigned to a switch.

Continuing with the initialization in `NetChain-controller.py`, the function `init()` will use the class object `RoutingController` from the script `routingcontroller.py` to set every mirror port of the switch with the controller and the to fill all the default matches of the following tables: `ipv4_lpm`, `op`, `seq` and `pkt_for_me`.

Once this has been done the cpu is set in a loop by the function `run_cpu_port_loop()`. The function will start a thread for the failure and begin the sniffing on the interfaces of the controller. The failure thread `wait_fail_over()` waits for the user to type the abbreviation of the switch that has failed. On the other hand, the sniffing will execute the function `recv_msg_cpu()`, which will check that the packets have the correct UDP destination port and will execute different functions according to the operations in the NetChain protocol.

2.3.2 Insert and delete operation handler

The inserting and deleting of the keys is among the topics with needed most review as stated before, due to the difficulty to definitively state what was needed to combine the consistent hashing, the mapping of virtual nodes with real switches and the chain of IPs a key had to be assigned.

Everything starts with receiving a `insert` operation packet from the host through the `s1` interface. From the sniffing loop, once the function `recv_msg_cpu()` recognizes the `insert`, it checks if the key is not in the `self.key` dictionary. Then it proceeds to include the key

to the system by executing the function `assign_key_to_nodes()`. This function begins with assigning a hashing value to the key through `self.hashing.calculate_hashing()`, which uses the hashing `md5` method, modulated by 2^{32} and then divided with 2^{22} so get the assigned virtual node. After this, `self.hashing.find_chain_node()` will get the list of the nodes to be assigned in the chain. The process can be summarized as a loop that will search through the list `self.list_node_sw` of the object class `consistent_hashing` and if the switch is not previously in the chain list, it will be added, otherwise the next node is checked. Once the length of the chain is reached, the list with the switch abbreviations is returned. With the previous assignments it's now possible to save the key in the nodes lists `self.node_keys`, in the switches dictionary `self.sw_memory_key_index`, in the dictionary `self.keys` and update the table `find_key_index` of all the switches involved in the chain to match correctly the key, operation and index in the register. The last step is sending a `write` operation back to the `mininet` with `send_back_to_host()`, that way the value of the key key will be written in all the needed switches.

When a `delete` operation is received by the controller, the process is similar, but in this case all the previous variables instead of adding something they remove or pop it in the function `eliminate_key_from_nodes()`. Of course the table `find_key_index` has to delete the matches with the key as well. Once this is done, the controller sends a packet back to the host through the `mininet`. The operation of this packet is `fin`, despite not being depicted in the paper, it is just a minor thing, just to indicate to the host that the key has been erased.

It is important to remember that in `self.sw_memory_key_index` the keys are saved in an orderly manner. That order also represents the index of the register in the switch, where the value for that key is saved. Thus if a key is eliminated, that position has to be replaced by a `None` string in `eliminate_key_from_group()`. In a similar way `insert_index_to_switch_table()` will add first a key to a position with a `None` string before appending it with higher index.

2.3.3 Fast fail-over handler

As stated before, in the function `run_cpu_port_loop()` a new thread `wait_fail_over` is started, which will be waiting for the abbreviation of the failed switch. But before that input is typed in, the links of that specific switch have to be put down by the command `link sX sY down` in the `mininet`.

As soon as the switch is typed in the thread, the tables `fast_failover` and `fast_failover_my` of each neighbor of the failed switch will be filled. The table `fast_failover` is meant to be applied if the packet that arrives at the switch, is not meant to be part of the chain, like `s1`. On the contrary, the table `fast_failover_my` is applied when the switch is part of the chain. In both cases similar actions will be applied, but with minor differences. With this configuration the fast-failover is activated and the thread will wait for the user to type again the switch that has failed in order to execute the recovery process.

2.3.4 Fast recovery handler

The Fast-Recovery process is waiting for an input switch in order to execute. When the switch is given, the function `recovery()` will be executed. The first step in the recovery is getting the possible permutations of IP chains that contain the failed switch using `get_chain_permutation()`. This is a recursive function that will go through all the possible chains, but will only save the list that has the failed switch inside in `self.possible_chain_permu`.

With the list of chains, a for loop is entered that will go through all these chains, select randomly one switch among the active ones that aren't in that chain, get the keys that have this particular chain assigned, execute `do_recovery()` accordingly to fill the respective tables and replace the failed switch in the mapping of nodes with the randomly selected switch from before by applying `self.hashing.replace_chain_node()`

The `do_recovery()` function receives a reference switch, the new randomly selected switch, the failed switch and the chain been used in `recovery()` for loop. The reference switch will be the next switch after the failed switch or the one before it if the failed switch is in the tail. First, in the new switch the keys will be added to dictionary `self.sw_memory_key_index()`, the chain list in `self.keys` will be updated with the new switch and the value of the key in the register `key_value_reg` from the reference switch will be written to the new switch registers in the correct index. At this point a new thread will be started `sync_values` joining the scheduler and allowing immediate updates of the registers in the new switch if the value of the keys has been overwritten by a new value in the reference one. Continuing, the `write` operation of the switches in the chain of the key have to be blocked momentarily. Thus the match of the key and the `write` operation in the table `find_key_index` is deleted. Then to the tables `recovery` and `recovery_my` of all the neighbors of the failed switch will be added the match of the failed switch IP. The last step is reactivating the `write` operation in those switches that it was previously canceled.

2.4 Switch Data Plane Implementation

In the controller subsections some tables and objectives of each one of them were mentioned, but in the following subdivisions those same functionalities can be explored in more detail. The code for the Data Switch plane is written in several files in order to make it more understandable. There is a headers file `headers.p4`, a parsers file `parsers.p4` and the ingress processing is separated in 4 scripts. These are `check_nc_pkt.p4`, `fast_fo.p4`, `l3_forward.p4` and `netchain.p4`. The last one is the main file, which compiles all of the others together.

2.4.1 Parsers and Headers

In the files `headers.p4` and `parsers.p4` the packet foundations are written so that the rest of the p4 implementation can use them. The `headers.p4` file describes the `ethernet_t`, `ipv4_t` and `udp_t` headers the same way previous examples of the class would do it. The header that has a different logic is the NetChain packet. As the number of IPs in the chain is variable, the parser needs to be flexible in this aspect. For this purpose, the header is separated in three different parts: `netchainlen_t`, `netchainip_t` and `netchain_t`. The `netchainlen_t` only has the amount of IP addresses in the chain, consequently `netchainip_t` saves only one IP of the chain and finally `netchain_t` has all the other parameters of the NetChain packet. As in other examples of the course, a struct is created which includes all the previous headers, the `netchainlen_t` as `netchainlen`, an array of 32 `netchainip_t` named `netchainip[32]` and the `netchain_t` as `netchain`. With this structure at our disposal, the parser can be flexible enough to manage a dynamic length if IP chains.

The parser extracts in a expected way the ethernet, ipv4 and UDP segments, but once the UDP is done, the parser will be directed to the state `parse_netchain.len`, which will extract the `netchainlen` component. If the value is zero the next state will be `parse_netchain` and the header `netchain` will be extracted. Were this not the case, the parser will go through states named `parse_netchain_ipX` where X is the ordered number of the IP address in the chain. In each one of these states a `netchainip` header will be extracted and orderly saved in one of the 32 blocks of the array. Once the `netchainlen` length matches in one of the IP states, the parser will be directed to `parse_netchain` and also extract the the `netchain` components for the structure. In the Deparser process everything will be set to be emitted, but only those blocks of the `netchainip` array with a valid IP will actually be sent.

In the `headers.p4` file there is also a struct `metadata` with 2 elements, `seq` and `index`. They will help in the processing of the sequence number and the index of the register assigned to each key in the switch.

2.4.2 General view of the Ingress Processing

For the processing logic many tables and actions are required, but a general view of it will make it more easy to understand. The processing pipeline is described in the file `netchain.p4`. The first thing it does is check if the header `netchainip` of the struct is valid, if this is not the case it will simply forward the packet with the table `ipv4_lpm`. Otherwise the table `pkt_for_me` will be applied and the following processing will depend on a hit or not. The table `pkt_for_me` only checks that the destination IP is the same as the IP of the current switch and that the UDP destination port is the same as the port assigned to the NetChain protocol, no actions are executed.

If the packet was not destined to this switch, thus no match in the table `pkt_for_me`, it will apply the tables `recovery`, `fast_failover` and `ipv4_lpm`. The recovery and fast-failover processing will be explained later in more detail.

On the contrary, if the packet is destined to this switch, the `seq_reg` register will be read and its value saved in the `seq` variable of the metadata `meta`. Following this, the sequence number of `netchain` will be compared with the register value and matched in the table `seq`. If the sequence number of the parsed NetChain packet is smaller than the value saved in the register or it doesn't match with the value zero in the table `seq`, the packet will be dropped. The table `seq` has only one match, the value zero. Matching the value zero means that the packet needs a sequence number to be assigned to it by the switch.

If the packet is not dropped, the sequence number of the packet will be saved in the register `seq_reg` and consecutively a first review of the NetChain packet operation will be done by the table `op`. This table has two elements, a match for the `insert` and another for the `delete` operation. In both cases the packet has to be sent to the controller. The other operations will be processed in the last part of the logic. The table `find_key_index` will execute the actions for the operations `write` and `read` respectively. At this point, the only step left is the application of the tables for the failure, recovery and the forwarding processes.

2.4.3 Forwarding processor

The basic forwarding tables and actions are found in the file `l3_forward.p4`. Two tables and two actions are defined, the names of each one of them are `ipv4_lpm`, `drop_table`, `set_nhops` and `drop`. The logic behind them is the same as the implementations revised in class. The tables are going to match the destination address of the switches, which are `20.0.X.1` with X in {1,2,3,4,5}. The actions either drop the packet or set the correct port to which the packet has to be sent.

In the forwarding process additional actions are used at some point: `Forward_next_node`, `Forward_next_2node` and `Forward_back`, which are defined in the file `check_nc_pkt.p4`. They are used in several tables, which are going to be discussed in more detail later. Essentially, what the first and second actions do is reducing by one the length component of `netchainlen`, set the block of the `netchainip` array containing the failed switch IP address to invalid, adjust the necessary components of the previous protocols and check if the packet has to be sent to the next destination according to the IP chain or back to the host. For forwarding it to the next destination, only the destination address in the IPv4 protocol has to be changed to the subsequent IP address in the `netchainip` array. Otherwise the action `Forward_back` is executed, which basically will arrange the IPv4, UDP and NetChain segments so that the packet is transmitted back and accepted by the host. At the end of this action the `netchain` Operation is set to 5, the host recognises it as an `fin` operation. The `fin` operations is not mentioned in the paper, but it is just included so that the host could recognize the successful ending of other operations.

2.4.4 NetChain packet processor

When it comes to the NetChain packet process, several tables and actions are involved. Given that a general understanding of the processing logic has been provided, here the tables and actions involved can be described in more detail. All of this tables and actions are written in the file `check_nc_pkt.p4`.

First we have the table `seq`. This table matches the `netchain` sequence number, but it only has the value zero. When this happens means the received packet needs a sequence number to be assigned to it. The action executed for this match is `assignseq`. What this action does is increase by one the value of `meta.seq`, write it down to the sequence number of the `netchain` header and the register `seq_reg` of the switch.

The next table is `op`. This table only looks for `insert` and `delete` operations and if it matches one of them, the actions `NetChain_insert` and `NetChain_delete` will be called respectively. Both of these actions do the same, they use the function `clone` to push the received packet as it is toward the controller.

Finally, the last table to be reviewed in this section is `find_key_index`. This table matches the key and the operation components of the `netchain` header. Depending on the key, the index will be given as an input for the following actions `NetChain_write` and `NetChain_read`. `NetChain_write` will overwrite the register `key_value_reg` with the new value. After this, the action `Forward_next_node` is executed so that the packet can be prepared correctly for the forward processing. For the read action the register `key_value_reg` is read according to the given index by the table and saved in the value component of `netchain`. As the host sends the `read` operations packets with a reversed IP chain, the first block in the `netchainip` array is already the tail. Thus the next step is setting the `netchainlen` length component to one, adjust all other protocols, set the other IPs in the chain as invalid and call the action `Forward_next_node`.

2.4.5 Fast fail-over processing

Tables and actions for this process are written in the file `fast_fo.p4`. When the processing pipeline is inspected two similar tables are applied, but they have slightly differences. The table `fast_failover_my` is applied when the packet is intended to arrive at this switch for a NetChain processing. Here the table will match the destination address in the IPv4 header, but as this IP has already been updated by the table `find_key_index`, what the table is really matching is the next destination in the IP chain. In case of a match the action `Failover_my` is called. The action will set the second block in the `netchainip` array as invalid, because this represents the destination address in the IPv4 header, and execute the action `Forward_next_2node`.

The other similar table is `fast_failover`. This one is applied when the packet is not intended to have a NetChain packet processing, like the switch `s1`. The table will compare the IPv4 destination address, which is really the first block in the `netchainip` array. For a match in the table, the action `Forward_next_node` will be executed.

2.4.6 Fast Recovery processing

The recovery uses two different tables that are quite similar, but due to the same reason as in fast-failover they will execute slightly different actions. The tables are `recovery` and `recovery_my` and both of them match the IP destination address of the IPv4 header with the Key attached to the packet. In case of a match the functions `modify_dst` and `modify_dst_my` will be called respectively and a new IP will be given as input from the table. What changes in essence is the next IP the packet is destined to. The difference in the actions comes from the fact that, if the packet is destined to this switch and had an NetChain process, the address towards the next IP is updated to `netchainip[1]` and the previous block `netchainip[0]` has already been set invalid by the table `find_key_index`. Instead, when the packet is not origi-

nally destined to this switch, the destined IP in the chain continues to be in the first block of `netchainip`.

Minor note to remember: this process changes the IP in the chain, because from the controller a new switch has been assigned due to a failure. In our implementation the `recovery` and `recovery_my` tables will not lose their matches at any point, because the host is not updated on the new mapping between virtual nodes and real switches. This means in case of a switch failure, the host will continue sending packets with the IP of the failed switch inside the chain. Thus the matches in these tables must be maintained.

3 Evaluation

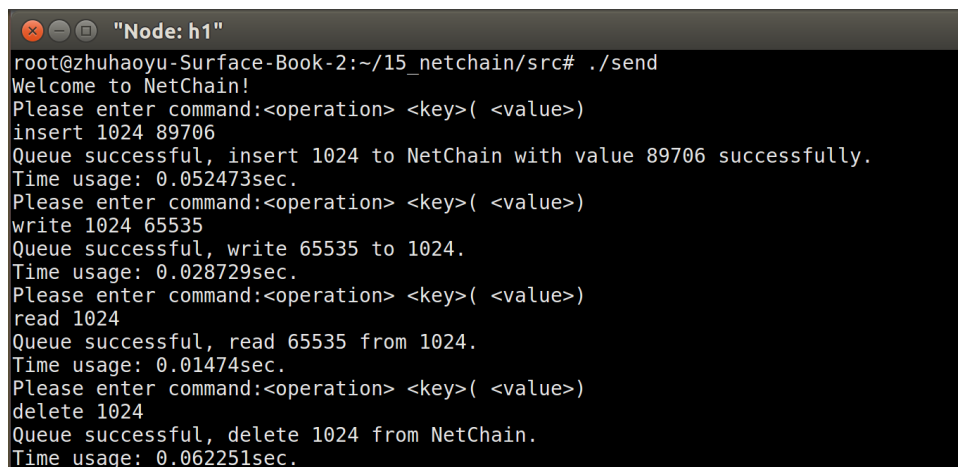
In the host program there is a part of evaluation, which is used to test the performance and functionality. All the test log can be found at `evaluation/`.

3.1 Functionality

In this project, we have implemented a series of functions: NetChain routing, NetChain packet parsing, chain establishment, NetChain write, read, insert and delete operations, Fast Fail-over, Fail Recovery. What we will test in this section is their functionalities.

Step1: the functionality of NetChain routing, packet parsing, chain establishment and operations.

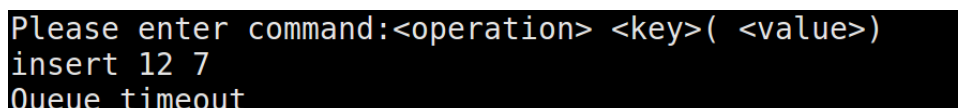
Here we will send four packets, using insert, write, read and delete operations to show the system works well.



```
"Node: h1"
root@zhuhaoyu-Surface-Book-2:~/15_netchain/src# ./send
Welcome to NetChain!
Please enter command:<operation> <key>( <value>)
insert 1024 89706
Queue successful, insert 1024 to NetChain with value 89706 successfully.
Time usage: 0.052473sec.
Please enter command:<operation> <key>( <value>)
write 1024 65535
Queue successful, write 65535 to 1024.
Time usage: 0.028729sec.
Please enter command:<operation> <key>( <value>)
read 1024
Queue successful, read 65535 from 1024.
Time usage: 0.01474sec.
Please enter command:<operation> <key>( <value>)
delete 1024
Queue successful, delete 1024 from NetChain.
Time usage: 0.062251sec.
```

Figure 2: Functionality evaluation: routing, packet parsing, chain generating and queuing

Then we manually down all links between s3 and its neighbors to simulate a switch failure. Now most of the packets will be dropped, only a few packets(doesn't go through s3 or read from other nodes) will have a reply.



```
Please enter command:<operation> <key>( <value>)
insert 12 7
Queue timeout
```

Figure 3: Functionality evaluation: Queue error when links down

At controller, we entered 's3' as the parameter of fail-over, after several milliseconds, the entries will be added to its neighbors' tables. When it finished, the queue processing will have a reply again.

```

Please enter command:<operation> <key>( <value>)
write 12 1025
Queue successful, write 1025 to 12.
Time usage: 0.01515sec.

```

Figure 4: Functionality evaluation: Successful queue when fail-over is presented

When fail-over is finished, input anything can start the recovery process. The failed nodes will be randomly assigned to other switches to archive the f node fault-tolerance.

3.2 Performance

Here we used 4 different scenarios in evaluation program to test the performance: single host insert performance, single host write performance, single host read performance and single node multi-thread throughput.

1. single host insert performance

We inserted 4096 keys to NetChain to test its insert performance. The result is in `performance/Insert_4096 keys.log`:

```

Inserting 4096 keys to Netchain finished. Statistics:
Succeed insertings : 4096
Failed insertings  : 0
Success rate       : 1
Time used          : 244.953      sec
Time used per key  : 0.0598031   sec

```

2. single host write performance

We write values to random keys to NetChain to test its writing performance. The test lasts 30 seconds. The result is in `performance/Write_30 sec.log`:

```

writing 3547 keys to Netchain finished. Statistics:
Succeed writings   : 3547
Failed writings    : 0
Success rate       : 1
Time used          : 30.0033sec
Time used per key  : 0.00845877sec

```

3. single host read performance

We write values to random keys to NetChain to test its writing performance. The test lasts 30 seconds. The result is in `performance/Read_30 sec.log`:

```

Reading 6442 keys to Netchain finished. Statistics:
Succeed readings   : 6442
Failed readings    : 0
Success rate       : 1
Time used          : 30.0043sec
Time used per key  : 0.00465761sec

```

Here from the previous 3 scenarios, we can figure out that the Insert operation is way slower than write and read. And the write operation needs to travel 3 switches (in project) to finish the operation while read operation needs to go through only one.

4. single node multi-thread throughput

In this scenario we tested the performance using 1,2,4,8,16 threads, each test lasts 30 seconds. In each test, every thread will randomly generate read or write operations and send it to switch. To better test the real throughput, the timeout of a queue is set to 100ms. If a packet is timeout, it will retry for five times. Which means, a failed queuing means failed for 5 retries. The result is in `performance/Throughput_*.log`:

```
1 Thread:
Queueing 4810 keys to Netchain finished. Statistics:
Succeed Queues      : 4810
Failed Queues       : 0
Success rate        : 1
Time used           : 30.0086 sec
Time used per key   : 0.0062388 sec
Throughput          : 160.287Queue/sec

2 Thread:
Queueing 6126 keys to Netchain finished. Statistics:
Succeed Queues      : 6126
Failed Queues       : 0
Success rate        : 1
Time used           : 30.0099 sec
Time used per key   : 0.00489878 sec
Throughput          : 204.132Queue/sec

4 Thread:
Queueing 9521 keys to Netchain finished. Statistics:
Succeed Queues      : 9521
Failed Queues       : 0
Success rate        : 1
Time used           : 30.0167 sec
Time used per key   : 0.00315268 sec
Throughput          : 317.19Queue/sec

8 Thread:
Queueing 12616 keys to Netchain finished. Statistics:
Succeed Queues      : 12617
Failed Queues       : 0
Success rate        : 1.00008
Time used           : 30.0433 sec
Time used per key   : 0.00238137 sec
Throughput          : 419.927Queue/sec

16 Thread:
Queueing 14332 keys to Netchain finished. Statistics:
Succeed Queues      : 14334
Failed Queues       : 0
Success rate        : 1.00014
Time used           : 30.0475 sec
Time used per key   : 0.00209653 sec
Throughput          : 476.979Queue/sec

32 Thread:
Queueing 11744 keys to Netchain finished. Statistics:
Succeed Queues      : 11747
Failed Queues       : 0
Success rate        : 1.00026
Time used           : 30.0882 sec
Time used per key   : 0.00256201 sec
Throughput          : 390.319Queue/sec
```

Here we can found that if we increase the threads, the queue throughput increases dramatically. But due to the cost of switching threads, The highest throughput occurs when using around , which is around 400 Queue per second.

3.3 Robust

Here we tested the system stability when failure occurs. We uses 2 scenarios to show the system stability when fast fail-over and recovery is performed.

1. fast fail-over performance

In this scenario, we start 2 xterm terminals. Host h1 will do write operation for 30 seconds and h2 will do read operations. When they are functioning, the link between s3 and its neighbor will be manually down, and after approximately 5 seconds, the fail-over will perform. We will calculate the time used between fast fail-over is applied and network is back to work again. The result is in `Failover_read.log` and `Failover_write.log`.

```
Reading 4367 keys to Netchain finished. Statistics:
Succeed readings   : 4349
Failed readings    : 18
Success rate       : 0.995878
Time used          : 30.0037sec
Time used per key  : 0.00687055sec

writing 2574 keys to Netchain finished. Statistics:
Succeed writings   : 2556
Failed writings    : 18
Success rate       : 0.993007
Time used          : 30.0032sec
Time used per key  : 0.0116562sec
```

In fact, when links are manually down, the system drops most packets since they need to go through this switch. But when fast-failover is performed, it back to work in less than 1 queue timeout. I.e. when we start to do fast-failover,

2. recovery time performance

In this scenario, we start 2 xterm terminals after fail-over is done. Host h1 will do write operation for 30 seconds and h2 will do read operations. When they are functioning, the fail recovery script is activated, and the write queue will be put on hold several times. We will calculate the queue successful rate when the recovery is in progress. The result is in `Recovery_read.log` and `Recovery_write.log`.

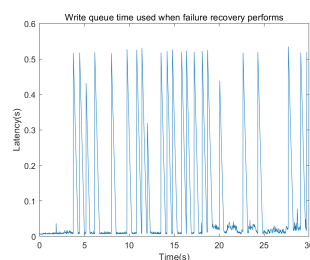
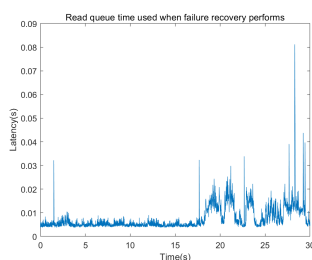


Figure 5: Read latency when recovery performs Figure 6: Write latency when recovery performs

```
Reading 4902 keys to Netchain finished. Statistics:
Succeed readings   : 4902
Failed readings    : 0
Success rate       : 1
Time used          : 30.0028sec
Time used per key  : 0.00612051sec

writing 1689 keys to Netchain finished. Statistics:
Succeed writings   : 1669
```

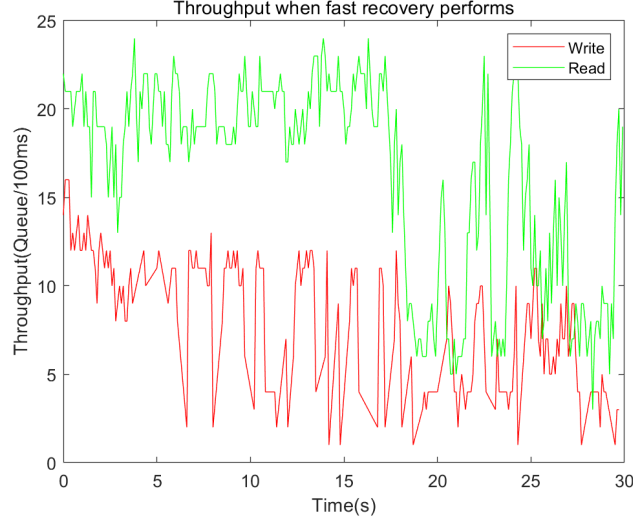



Figure 7: Throughput when recovery performs

```

Failed writings      : 20
Success rate        : 0.988159
Time used           : 30.2643sec
Time used per key   : 0.0179185sec

```

In this evaluation we can see that when doing recovery, only few writing packets are dropped even 4096 keys is stored in NetChain, and reading packets is never influenced.

4 Conclusion

4.1 Work summary

In our project, we reproduced the paper "NetChain: Scale-Free Sub-RTT Coordination" in a p4 way. The project successfully implemented and tested most functionalities concerned in the paper. In a coordination between p4 switch and a python-based controller script, our project can perform basic functionalities: node assignment, key-value storage and chain routing. Also the reproduced project can handle a fail-over and recovery when switch failure occurs. The project is based on Software Defined Networks(SDN) and over normal routing algorithms. It shows the probability and feasibility of offloading the key-value storage to switch data-plane to boost the IO speed when doing key-value storage. The project also shows that this well-designed chain replication have a strong fault-tolerance which can handle f error nodes when $f+1$ nodes are contained in a chain.

4.2 Future works and expectations

In this project, most the functionality is implemented and works perfectly, but what is expected but missing is the scalability of the chain. In our project, we have a way and framework to scale the chain, but do not have enough time to finish it. Moreover, in the paper, the authors talked about the value size scalability. when a value is too large for one packet to carry or for switch on-chip register to store, there is one possible way is to divide it into several parts and store it in several key-value pairs.

In the implementation of the project we also founds some disadvantages in the paper that may need to be solved. First is about the matching table size usage. Apparently every key needs more than two entries. One is for reading and another one is for writing. When fail recovery performs, the recovery table of the neighbor switch of the failed one will be filled with all keys

and destinations to the newly assigned switch. Another is the fail recovery sync problems. In the paper it said that they will monitor the registers between source switch and destination switch. In our project we designed a new packet operation called "transfer", which is not implemented due to the time. It reads value from the source switch and write it to destination switch, which is performed with sequence number to keep consistency.

References

- [1] D. KARGER, E. LEHMAN, T. L. R. P. M. L., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving ht spots on the world wide web. In *STOC '97 Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (1997), ACM, pp. 654–663.
- [2] DEN BOER, B., AND BOSSELAERS, A. Collisions for the compression function of md5. In *Workshop on the Theory and Application of Cryptographic Techniques* (1993), Springer, pp. 293–304.
- [3] F. DABEK, M. F. KAASHOEK, D. K. R. M., AND STOICA, I. Wide-area cooperative storage with cfs. In *SOSP '01 Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), ACM, pp. 202–215.
- [4] JIN, X., LI, X., ZHANG, H., FOSTER, N., LEE, J., SOULÉ, R., KIM, C., AND STOICA, I. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, Apr. 2018), USENIX Association, pp. 35–49.

A Group organization

Haoyu Zhu

1. Netchain Header defined, Parser of NetChain finished, Write and Read Queue.
2. Forwarding and routing.
3. Implement sequence on switch but not host to enable parallal operations.
4. Fast-recovery dealing.
5. Topology init and routing table filling.
6. Generate and send Netchain queues.
7. Scapy Netchain header defined
8. NetChain node routing.
9. Evaluation script and program.
10. Report of implementation of hosts.

Francisco Dumont

1. Forwarding and routing.
2. NetChain runtime table filling.
3. Filling index table and implement Insert and Delete function. Francisco.
4. Fast-failover table filling. Francisco.
5. Fast recovery when failure occurs.
6. Generate and send Netchain queues.
7. Most part of report, including Introduction, Abstract, Paper review, Implementation of controller and switch.

Francisco have a lot of contributions to the implementation part, Althouth some code may not be used at last, He meets every week with Haoyu at saturday, and works when he is free. Haoyu have developed Host program, more than half of the switch code and at last week, he changed some codes in controller because of our misunderstanding of paper.