POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Performance Analysis of Switch Workload Management Protocols for Burst Tolerant Networks

Supervisors

Prof. Guido MARCHETTO

Prof. Alessio SACCO

Prof. Flavio ESPOSITO

Candidate

Lorenzo PANTANO

July 2024

Abstract

One of the main causes of packet loss and performance degradation in today's datacenter operations, is represented by microsecond-scaled congestion events, known as microbursts. Microbursts are characterized by sudden spikes in network traffic, and are likely to lead to congestion, impacting the overall efficiency of the datacenter. Existing solutions, such as packet deflection techniques, have shown promise in mitigating microburst effects. However, further research is needed to explore alternative approaches that can enhance network performance and stability. In this study, we propose Robinhood, a solution aimed at mitigating microbursts in datacenter networks as well as improving the overall performance. We exploit the insights gained from existing solutions, taking inspiration from the job-scheduling domain, and implement novel algorithms based on work-stealing scheduling techniques to address the challenge of microburst mitigation. We implemented a simulation framework from scratch, featuring BRITE for network topology generation, to test different networks under various degrees of load. Through extensive simulations and performance evaluations, we demonstrate the effectiveness of our proposed work-stealing scheduling algorithms in mitigating microbursts: for example, in a leaf-spine architecture network under 80% of load, the flow completion time improves by 22%, 6% and 7% when applying work-stealing techniques to switch buffers, compared to ECMP, DIBS and Vertigo respectively. Additionally, the flexibility of these algorithms, makes them suitable for applicability to other network environments beyond datacenters, such as 5G and other burst tolerant networks. For future research purposes, we released the source code of the simulation framework, in order to be tested with different network topologies and configurations.

Acknowledgements

Table of Contents

List of Tables

List of Figures

Acronyms

DCN
Data Center Network

TCP
Transport Control Protocol

UDP
User Datagram Protocol

IP
Internet Protocol

ToS
Type Of Service

DCTCP
Data Center TCP

ECN
Explicit Congestion Notification

CE
Congestion Experienced

ToR
Top Of Rack Switches

RFS
Remaining Flow Size

SR

Steal Request

SC

Steal Cancel

DIBS

Detour-Induced Buffer Sharing

ECMP

Equal Cost Multi-Path Routing

Chapter 1

Introduction

Datacenters are, as of today, at the core of our modern technologies, forming a critical infrastructure underlying cloud computing, and enabling a huge amount of services and applications. Most of the data processing and network traffic happens and goes through data center servers and nodes, from simple website hosting to complex data processing or analytics and machine learning workloads.

A commonly known scheme used by services like web search, social network content composition and the selection of advertisements, requires requests at higher level to be partitioned and framed to low-level workers in data centers, then the responses are aggregated to produce a meaningful result. The efficiency and reliability of these networks are thus essential, as the disruption of these services can cause major inconveniences or even financial losses. With these stringent low-latency requirements, and thanks to the technologies that they implement, the disaggregated resources carried on in data centers such as GPUs, memory, disk, etc. are extremely fast.

So, as of today, the network is frequently the performance bottleneck for data center operations. Among the varying challenges faced by data center networks, congestion control is one of the most persistent and significant, given the vast amount of traffic and data to be processed. The most prominent cause of congestion is represented by microbursts, which are sudden, short-lived and intense spikes in the network traffic. As an immediate consequence, they lead to significant packet retransmission that causes loss in performance. Traditional routing and congestion control algorithms, such as ECMP (Equal Cost Multi-Path Routing) proved to be often inadequate in handling the intensity of microbursts. Additionally, their short-lived nature, lasting no more than a millisecond, require networks to be able to react to them in real-time, with a fast and efficient solution. As a result, there has been a growing interest for better mitigation strategies, that effectively address the problems of microbursts, with more advanced and complex congestion control techniques.

1.1 Objectives

The main goal of this study is, therefore, to introduce and evaluate a novel algorithm, named Robinhood, that aims at mitigating congestion in data center networks, and improving the overall performances, especially in the presence of microbursts. Robinhood has been designed taking into consideration the limitations of existing solutions, providing an alternative approach in the congestion control mechanisms. Current strategies predominantly revolve around the fundamental concept of redirecting incoming packets away from full buffers to more optimal and less congested nodes, a technique commonly referred to as packet deflection. Robinhood similarly operates on this principle. However, the intricate decisions involved in determining factors such as the choice of nodes for traffic redirection, initiation time for deflection, and the specific target node for rerouting make way for a multitude of algorithmic variations, which serve as the foundational basis for existing solutions. We also seek to compare Robinhood against three of the currently well-established approaches, such as Vertigo [1], DIBS [2] and ECMP showing a promising enhancement in terms of average flow completion times and throughput.

1.2 Contributions

Robinhood's inspiration comes from the job-scheduling domain [3]. Task and job scheduling efficiency problems share a common interest with packet deflection techniques; at their core, their main objective is to improve load balancing, achieving a better parallelization of tasks and packet processing, leading to an efficiency boost. Looking into the task scheduling domain, studies [3] on efficient scheduling policies have shown that the work-stealing technique is the most promising in balancing load and core efficiency for microsecond-scaled tasks. Transposing this solution, with all the required adjustments, into the network plane, is what Robinhood proposes to implement, given the short-lived nature of microbursts. Robinhood is then a packet deflection technique based on the work-stealing implementation in the network plane, aiming at observing the impact of these techniques on data center network performance. Robinhood's fundamental idea is to offload packets among neighboring switches to balance the load, whenever possible, preemptively choosing suitable candidates. Furthermore, naive deflection leads to excessive packet reordering, with a direct impact on network performance. To address this issue the deflection process is flow-based, where a flow is a sequence of packets uniquely identified by common attributes such as source, destination and other properties. The flow-based deflection results in less need for reordering packets once they reach their destination. To test and evaluate the performance of these

algorithms, we have developed a network simulation framework from scratch, rather than adopting existing ones, to allow for more flexibility.

The software comprises three main components: the event generator, which generates packets based on configurable distributions; the simulation core, which includes all necessary network components and the implemented algorithms (with a margin for adding more); and a metrics component that tracks and updates all performance metrics.

Most of the results are evaluated on a leaf-spine architecture, validating the findings against an equivalent three-tiered network, which are the same network architectures that Vertigo and DIBS use for their tests; using the BRITE [4] software to define nodes and edges, though, allows the generation of custom topologies, with different degrees of complexity and scale, making it possible to further extend the testing of the algorithms in other use-cases beyond data centers. Despite having undergone different iterations and versions, the final version of Robinhood has been able to achieve significant improvements in terms of average flow completion times, allowing faster data center operations. Robinhood has improved average flow completion time by 22% when compared to ECMP, 7% and 6% when compared to Vertigo and DIBS, respectively.

Chapter 2

Existing Solutions and Challenges

Given the need for high throughput and low-latency efficiency, and the importance of data center networks, a variety of solutions has been proposed to mitigate the effects of congestion. In this chapter we start taking a look at modern data center network (DCN) architectures and the problems they face, with a brief explanation and introduction to the main solutions that have been proposed, and the limitations they carry.

2.1 Data Center Network Architectures

The modern architectures proposed for data centers are mainly two: Three-Tiered and Fat-Tree [5] topologies. Three-Tiered architectures are more of a traditional setup, consisting, as the name suggests, in three levels of switches, each which different characteristics and performance capabilities. The core switches are the ones responsible for connecting the data center with the Internet, bringing great performance capabilities; then there are aggregation (or distribution) layer switches, often equipped with load balancing services and firewalls as a level of protection and management for the rest of the network. Finally there are access layer switches, commonly referred as Top-Of-Rack (ToR), which are switches responsible for connecting to the servers. ToR switches are equipped with a relatively large amount of ports, since a single ToR switch is often connected to several (24-48) servers. The three-tiered architecture gives its best performance when used against north to south traffic, meaning traffic entering the data center from outside, or exiting the data center network. Nowadays, a vast amount of traffic moves instead from east to west, going from server to server among the same data centers.

The Fat-Tree (or Two-Tiered) architecture is the preferred choice for this kind of

traffic, considerably reducing the number of hops for intra-DCN traffic. Essentially, in Fat-Tree architectures, the core and aggregation layers are condensed into one layer of switches, called Spine, while ToR nodes are now called Leaf switches, even though they preserve the same task of being connected to end servers. The number of hops from intra-DCN traffic is reduced, favoring east-west traffic, but at the same time a larger amount of switches per-layer is required. Fat-Tree topologies can also be three-tiered: a generalization of this architecture, the k-ary Fat-Tree consists in the same core, aggregation and edge switches, in a higher number of nodes. $(k/2)^2$ core switches are connected to k pods. Each pod consists of edge (ToR) and aggregation switches. In each pod, edge switches are connected up to $k/2$ servers and up to $k/2$ aggregation switches, while the latter are each one connected to $k/2$ core switches and $k/2$ edge switches.

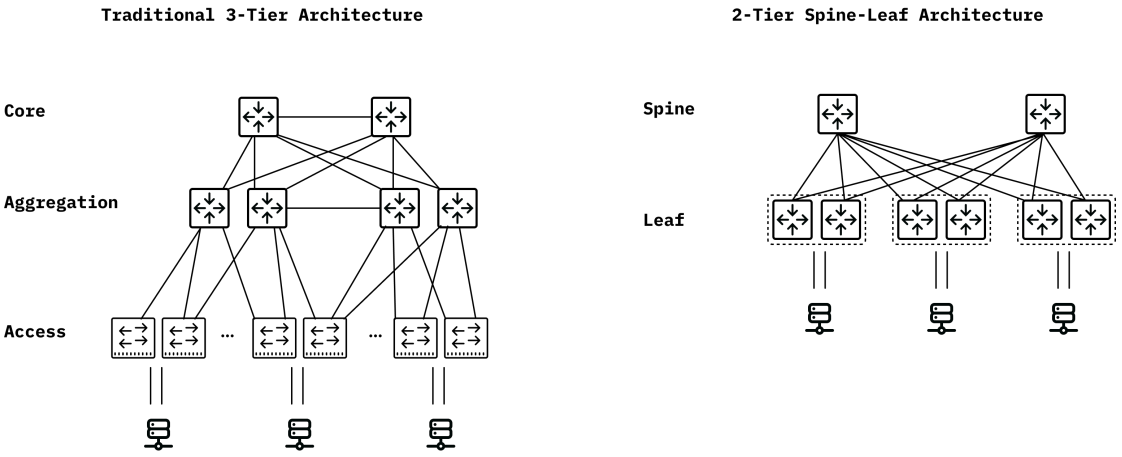

Figure 2.1: Data Center Network Architectures

2.2 Congestion Control Mechanisms

As previously mentioned, modern strategies for microbursts mitigation rely on packet deflection, which involves rerouting incoming packets to neighboring switches, independently of the architecture. While straightforward and efficient [6], most of the deflection approaches suffer from two main problems: activating only during the congestion, not trying to avoid it, and excessive packet reordering. One of the main solutions, which sets the basis for more advanced solutions, is data center TCP (DCTCP) [7], an enhanced version of the TCP protocol specifically crafted to minimize congestion in data center network environments. DCTCP [7] exploits the TCP explicit congestion notification (ECN) to give continuous feedback about congestion status. Marking packets, with ECN specific fields (e.g. CE - Congestion

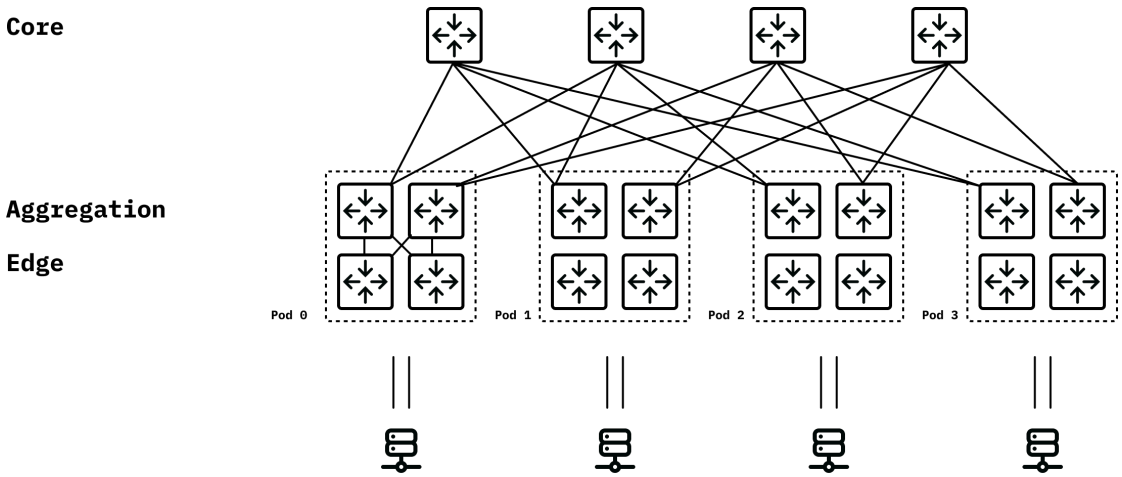

Figure 2.2: Data Center Network Architectures - K-ary Fat-Tree Three-Tiered

Experienced) instead of instantly dropping them, provides a more fine-grained control over the congestion window size of the sender, which once adjusted, provides lower queuing delays and higher throughput.

Although, DCTCP has been a breakthrough in data center network environments, its reliance on continuous feedback makes it not particularly efficient when handling sudden increases in the network traffic, such as the ones brought by microbursts. Most of the solutions presented in this study for comparison, and Robinhood itself, are built on the foundations of DCTCP, and work alongside it without trying to replace it. The main solutions referenced in this study, that exploit the concepts of packet deflection, and that rely on a DCTCP foundation, are Vertigo [1] and DIBS [2] which both suffer from the previously mentioned issues. We also consider Equal-Cost Multi-Path routing (ECMP) [8] as an alternative approach that does not rely on packet deflection, but provides more static flow distribution routing paths.

2.2.1 Deflection challenges

In this section, before diving into the details of the aforementioned protocols, we explore the fundamental ideas behind deflection, and the challenges that its naive realization creates. As already stated, packet deflection means re-routing incoming packets to congested hot-spots instead of dropping them. Despite being effective at managing bursts, a naively deployed deflection still brings high drop rates, excessive rerouting and path stretch, causing packets to follow longer paths and increasing latency, along with an inevitable out-of-order delivery. Regarding the packet ordering issue, both Vertigo and DIBS, try to solve it with different solutions,

as will be further detailed. DIBS for instance, addresses reordering by disabling the fast re-transmission option in congestion control protocols such as DCTCP, after receiving out-of-order packets; Vertigo instead operates on end hosts to mark packets with packet-specific value information and retrieve it at destination hosts, necessitating additional components. Vertigo's reordering solution brings another issue to the table of deflection challenges: implementing deflection techniques is not always feasible [6], and most of the time, not having access to end hosts, operating directly at the network core is the best option. This is also what Robinhood proposes to do. It is a solution easily implementable in modern programmable switches and that operates at the network core, without the need to impact end hosts.

2.2.2 Vertigo

Diving deeply in the proposed approaches for packet deflection, Vertigo adopts the mechanism of so called Selective Deflection [6], meaning that it carefully selects which packets are to be deflected, based on their impact on the current congestion [1], prioritizing relatively short flows instead of longer ones. Vertigo consists of three components (figure 2.3), implied both on the end hosts and at the core of the network. An end host component marks a packet with remaining flow size (RFS) information in an additional header (flow-info header) provided by the protocol. This information proves to be very useful when estimating how many packets are expected to arrive at a given node in the network, since it contains the value of how many bytes are left to transmit in the flow the packet belongs to; and will also be exploited in the early versions of Robinhood for a flow-based selective deflection. When a packet encounters a switch with a full output buffer, Vertigo's core component selects the packet with the highest RFS for deflection and dequeues it from the buffer. It then randomly selects two queues in the switch, deflecting the selected packet to the least loaded one. If both queues are full, the packet with the highest RFS value selected between the deflected packet and others in the queues, is dropped and retransmitted, with the RFS value halved to prevent starvation, and an additional value is set in the flow-info header to indicate the halving, since this process of RFS being halved can happen multiple times for the same packet.

Vertigo's main strength is the ability to react to microbursts in real-time and preventing an overloaded buffer to become even more congested by selectively deflecting packets that are more likely to contribute to congestion, leaving short flows to be transmitted faster, thus reducing the overall load on buffers.

However, the random selection of a deflection port and the deflection choices made on each packet arrival, lead to excessive out-of-order delivery. Vertigo third component on end-hosts takes care of packet reordering based on the RFS value of the packets; if a packet RFS value has been halved due to the deflection algorithm it

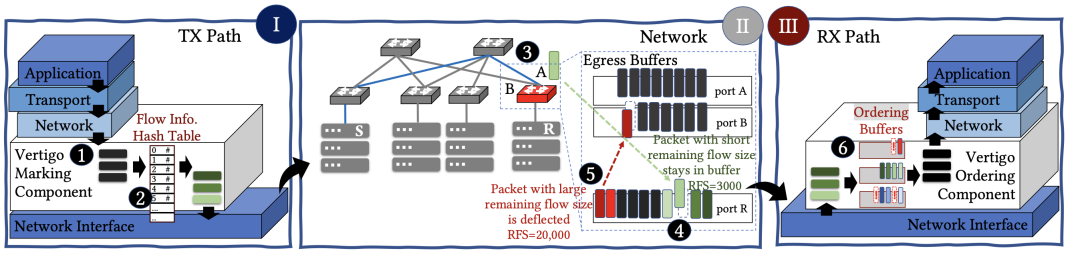

Figure 2.3: Vertigo Design Overview. From Vertigo design article [1]

is multiplied by the value set in the header at the moment of drop or re-transmission. The RFS value marking in end hosts, even if introducing a minimal additional header for the packets and thus processing overhead, proves to be very useful for both the packet selective deflection and reordering. Along with the additional overhead, the deployment of different components on different parts of the network, over which our control, especially for the end-hosts, is not granted, is another of the issues carried on by Vertigo. In Robinhood we therefore looked into a solution that is employable directly at the network core, without the need to operate on end-hosts, and that is also capable of bringing the same deflection efficiency Vertigo presents.

2.2.3 DIBS

The Detour-Induced Buffer Sharing (DIBS) technique [2] instead, focuses on a more brute-force approach, deflecting packets to randomly selected ports each time it is needed. This method is also referred to as Simple Deflection, opposed to the Selective Deflection of Vertigo. When a packet arrives at a switch input port, the switch checks to see if the buffer for the destination port is full. If so, instead of dropping the packet, the switch selects one of its other ports to forward the packet to, avoiding ports whose buffers are full and ports connected to end-hosts. Other switches will buffer and forward the packet following the same protocol, and the packet will eventually make its way to its destination, possibly coming back through the switch that originally detoured it.

Although great performances are achieved in terms of drop rates and low-latency, DIBS struggles to avoid possible loops, often causing packets to be sent back and forth among different switches, before actually reaching its destination; DIBS studies [2] (figure 2.4) show how packets get detoured up to 15 times across a set of a small number of switches, causing a major slow-down in packet delivery times, impacting throughput and causing a non-negotiable reordering problem. To face the problem of reordering, a direct consequence of the random detour, DIBS disables

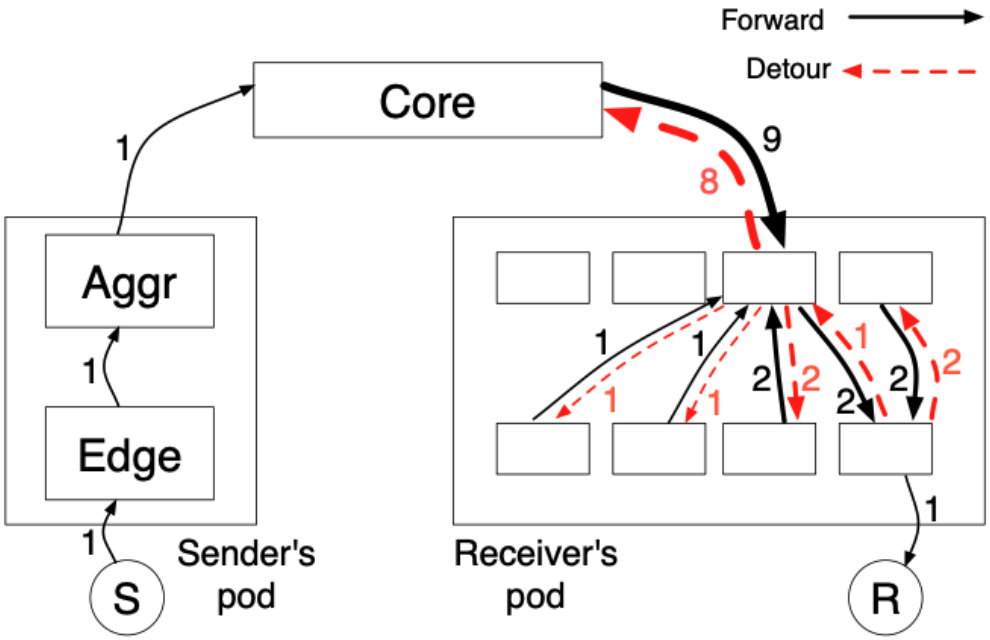

Figure 2.4: DIBS example of 15 detours, in a K=8 Fat-Tree topology. From DIBS article [2]

the fast re-transmit option in congestion control protocols such as DCTCP or standard TCP which reduces the sending rate after receiving consecutive reordered packets. However, this solution unfortunately does not fix the problem completely. Robinhood's proposal is also to limit the deflection targets to avoid unnecessary detours and, operating at flow level, we eliminate the problem of reordering packets, as we will discuss in detail.

2.3 Robinhood Vision

2.3.1 Strengths and Limitations of Modern Strategies

Despite bringing a significant contribution to network efficiency and performance, the referred solutions focus on solving the congestion problem when it has already been consolidated. Microbursts are certainly the main motivation behind this behaviour, and advanced packet deflection techniques such as Vertigo and DIBS, definitely pose a graceful limitation to the problem. The specific problems, carried on by Vertigo and DIBS, though, are the foundation for our newly developed Robinhood algorithm; addressing reordering and unnecessary detours with a network-core

strategy. As mentioned, both Vertigo and DIBS address the reordering overhead in different approaches. Vertigo, with its selective deflection [6], in contrast to DIBS which follows a more simple deflection approach, additionally operates on end hosts both in the transmission and reception, having them perform marking steps to overcome the out-of-order delivery. DIBS instead opts to operate on the fast retransmit options of congestion control protocol, only after receiving non-ordered packets. The cause that brings to reordering is mainly a direct consequence of deflection itself: detouring packets will necessarily bring them to be delivered out-of-order, contrary to following the same established shortest path; and these strategies are trying more to repair the damage made by deflection than solving the problem at the core. Robinhood vision proposes instead to operate at flow-level in case of deflection. Detouring packets belonging to the same flow, to the same targets, will surely make them follow the same, even if not the shortest, path, resulting in an ordered delivery. Operating at the network core, also solves the issue of feasible implementation. The management control over end host is, in most of the cases, not granted; making the marking step required by Vertigo hard to realize.

2.3.2 Job Scheduling Domain

Robinhood's inspiration comes from job scheduling policies, which provide numerous strategies for efficient load balancing and core efficiency [3], which are particularly valuable in managing traffic congestion, as rerouting packets is a direct implication of load balancing. Robinhood's primary proposed approach, work-stealing (and a work-donating variant), derives from its respective scheduling policy. In the task-scheduling domain, work stealing means redistributing the load by letting under-loaded nodes execute part of the jobs scheduled on another node, and it was found to be [3] the most promising solution when it comes to microsecond-scaled tasks, which become particularly meaningful when adopting this approach against microbursts. In the network plane, this approach simply translates into the offloading of packets by an overloaded switch to an under-loaded one. In our proposed method, switches are constantly aware of the state of their neighbors by exchanging state messages that we will discuss later. This information allows switches that are experiencing huge amounts of traffic, to offload part (or most) of their load to neighbors who are not overwhelmed by packets. It is important to note that a direct translation of work-stealing paradigms into the network plane is not an immediate solution. As previously stated, to be aware of the under-loaded neighbors, nodes need to exchange information in some way or another, which brings additional overhead, that can be minimized making smart protocol decisions.

Chapter 3

Robinhood

3.1 Design Principles

Given the need for an alternative and efficient approach to deal with microbursts, and to minimize packet reordering, we present Robinhood: a work-stealing based workload management protocol for burst tolerant networks. Robinhood's principal idea is simple: to share buffer capacity among neighboring switches, so that overloaded switches can offload packets to under-loaded ones. To realize this idea, switches need to be aware of the state of their neighbors; to do so we introduce also a basic communication protocol to allow switches exchanging their load status information. The guiding principle of Robinhood is also to avoid or minimize reordering: in order to do so, we have opted for a flow-based deflection where a flow is uniquely identified by the values of source IP, destination IP, source port, destination port, protocol TCP/UDP of a single packet. We aimed at deflecting packets belonging to the same flow, to the same neighboring switches, so that, even if deflected, packets of the same flow are likely to follow the same path, resulting in an ordered delivery. Robinhood's results are promising; the design choices focus on solving existing deflection problems, while maintaining high throughput and low-latency. Its design is also a suitable implementation for modern switches, operating directly at the network core without having to alter end-hosts operations.

3.2 Protocol Overview

Every certain X period of time, switches will broadcast their status to neighbors. The period of time X is set, in our simulations, to every simulation time unit, in order to react to congestion in real-time. This value can, however, vary depending on the situations. Increasing the value will reduce the number of status updates provided by switches, thus impacting on the overhead of packet processing, but

at the same time, a much quicker response time to congestion may be needed. The sooner switches find out about congestion in progress, the sooner they will be able to react. For an optimal solution that does not introduce overhead and maximises the number of updates, state messages are embedded in the Type of Service (ToS) field of IP, present in both IPv4 and IPv6 as Figure 3.1 shows. Each switch will keep in memory a list of the available neighbors, that are ready to accept offloaded packets in case of congestion; the list is updated by the status messages they exchange. Once a switch as been acknowledged as overloaded, it will start deflecting packets towards one or more neighboring switches in its list, performing deflection at flow-level, meaning that every packet of the same flow is deflected towards the same switch. **The flow-level deflection is critical to avoid reordering**. Neighbor candidates are picked randomly, when choosing a deflection target, in the final version of the protocol. To help with deflection decisions, a data structure is needed: the **Deflection Table** is what holds the associations between deflected flow - target neighbor, and will be populated on-the-go, as packets come to the full buffer and its flow is not listed in the table. Any packet incoming to the overloaded switch will then see its flow id being checked against this table for a matching entry. If not found, a random neighbor is chosen and an entry created for that flow.

As previously mentioned, and introduced by Vertigo, the remaining flow size info provides a good estimate of how much is left to transmit for a given flow. Picking the one which has the most packets still to transmit will provide a smarter avoidance of the congestion. In the early versions of Robinhood we exploit this value, giving priority to the flows that are more likely to lead to congestion, and changing the mode of neighbor selection, preferring the least loaded. This simpler approach though, will not prevent the buffer to become full, since packets coming from other flows may contribute to an already congested buffer. In the final version of Robinhood, we instead deflect, as already stated, not only the flow with more packets to transmit, but also any other incoming flow, until the buffer is no longer congested.

3.3 Buffer Threshold

To better define the status of a switch we set a threshold for the buffers. The main strength of Vertigo and DIBS is their ability to react to congestion when packets arrive at full buffers. In Robinhood we challenge this methodology and propose that switches will react to incoming congestion before buffers are actually full, setting a threshold on the buffer size. If the threshold is overcome by the actual buffer size, switches will start to refer to themselves as overloaded. After several experiments, the best value for the threshold is found to be around 15%

less of the original buffer size; if the size of a buffer is 300KB, then the threshold is set to 250KB. **The value of the threshold is critical**: decreasing this value will activate Robinhood and hence, packet deflection, more frequently, often when switches are not actually congested; a relatively high value is needed to ensure that deflection is performed only when switches are effectively experiencing congestion. The premature deflection of packets will, in Robinhood, lead to longer completion times for the flows, since deflected packets are least likely to follow the shortest path. Further research on the Robinhood solution, may introduce an adaptive buffer threshold, that adjusts to traffic in real-time, possibly introducing predictive machine learning models based on the collected data. A low-value of the threshold activates Robinhood more frequently, but it has also the advantage of starting deflection processes sooner, eventually absorbing the congestion faster. In this study, though, we kept a fixed value for the threshold, leaving the adaptive value implementation for future development.

3.4 Exchange Messages

The background assumption of the Robinhood approach requires switches to exchange messages about their status to neighbors. These messages, are called **Steal Requests (SR)**, and as the name suggests indicate the willingness for a neighbor to accept the offloading of packets. Every X time period, if switches are under-loaded, they will broadcast Steal Request messages to their neighbors. For a switch to be under-loaded it is required that the size of their buffers be currently below the set threshold. Upon reception of an SR message, neighbors will append the SR sender to the list of possible helping switches, in case of needed deflection. The candidates for deflection are listed together with an expiry timeout. If a switch does not receive a Steal Request after the timeout has expired, it means that the neighbor is experiencing congestion and is no longer a suitable deflection target. It will be therefore removed from the list.

There is another possible solution for a switch to remove a candidate, which is the reception of a **Steal Cancel (SC)** message. This message is sent, broadcast, by a switch when it becomes overloaded, to ensure that it will not receive any deflected packets. Upon receiving a SC message, switches will immediately remove the sender from the candidate list, even if the timeout has not expired. Steal Cancel messages are optional, but recommended, since the expiring timeout will take care of removing overloaded switches from the list, but at the same time, to reduce overhead, timeout is set high enough so as not to introduce more overhead than what is actually needed. So, an explicit notification would prove useful to remove the candidate faster, avoiding the chance to possibly chose the sender as a deflection target. Figure 3.1 shows the explicit set of bits in the Type Of Service

(ToS) field in the IPv4 header. Robinhood sets the last bit of the field to 0 to indicate a Steal Request (switch is under-loaded), and in case of explicit notification of overloading (Steal Cancel), the bit is set to 1.

3.4.1 Work Donating

A variant of the Robinhood protocol, opposed to work-stealing principles, suggests that instead of preemptively sending steal requests, overloaded switches should be the ones in charge of sending requests for assistance (Donation Request), and neighbors should respond with their availability. Since this process introduces an increased number of message exchanges and does not provide any significant improvement (actually it introduces a little more overhead) we have decided to not to go on with this approach, but have listed it here as part of the development process.

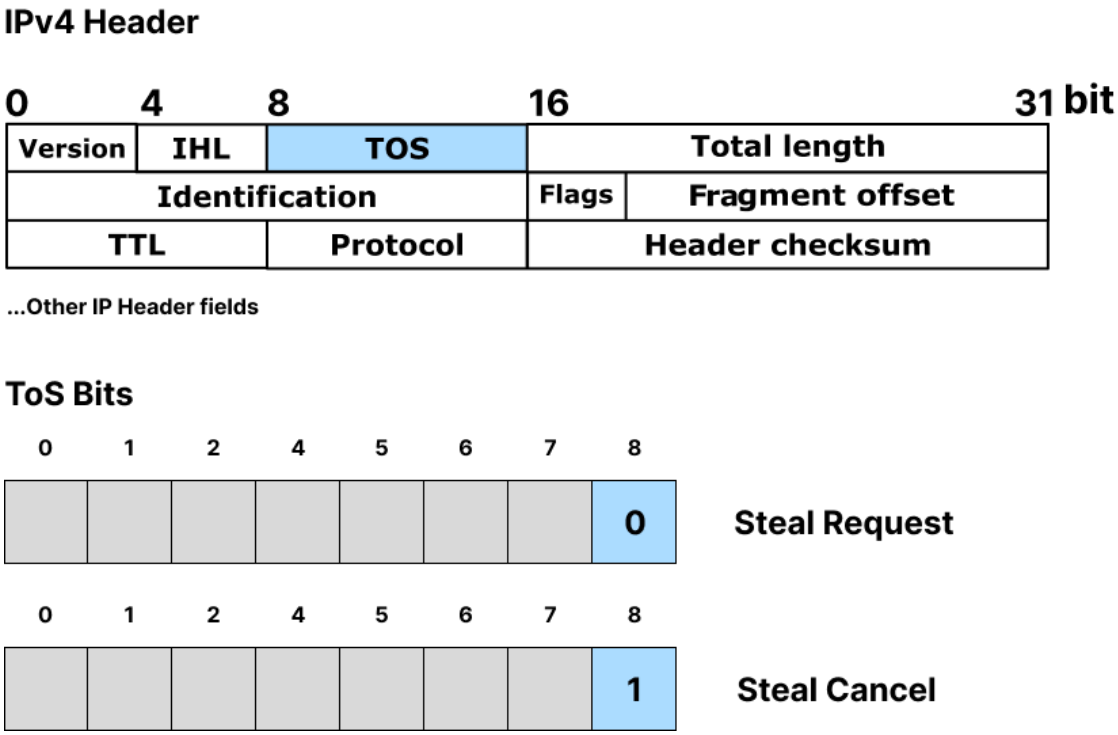

Figure 3.1: RobinHood - Steal Request/Steal Cancel messages as IPv4 ToS bits. Last bit is set to 0 to indicate Steal Requests. Last bit is set to 1 to indicate Steal Cancels.

3.5 Robinhood's Deflection Development

Once the stealing neighbors have been defined though the exchange of Steal Request messages, a switch becoming overloaded has now the ability to start deflecting packets towards neighbors. We present here the development of the protocol and the conclusions that led us to the final decisions; from a first iteration, where we deflect one single flow to a single neighbor, to a final version where we deflect multiple flows to multiple neighbors. It is important to note that even deflecting multiple flows to multiple neighbors, we still avoid deflecting the same flow to different neighbors, to avoid reordering. All the iterations are based on an already available deflection candidate list. What changes in the different versions are the selection of flows being deflected, how many flows are deflected, and the way we choose the deflection targets.

3.5.1 First Iteration

In the first iteration of the protocol, we exploit the remaining flow size information as intended and introduced by Vertigo. Note that the exploitation of this value introduces more overhead since an additional header for the packets is needed, and a marking step on end hosts is required. A newly overloaded switch will start by picking the packet with the highest RFS value in the overflowing buffer and the flow it belongs to. It will then pick a neighbor from the list of available candidates, and will start deflecting every incoming packet of that flow to the selected neighbor.

The neighbor selection process can be random, but for the first version of the protocol, since we are only deflecting one flow, the best choice would be the least overloaded one, although this will require additional overhead for the load information of every switch. **The selection of the least loaded neighbor will bring different results when deflecting multiple flows**. From now on, every incoming packet belonging to the chosen flow, will be deflected to the least loaded neighbor. The procedure in Algorithm 1 describes how a newly overloaded switch behaves when its buffer threshold has been surpassed. With this primitive and early approach Robinhood is able to overcome ECMP static paths average flow completion times, but with higher drop rates. As mentioned earlier, the deflection of a single flow, even with a set threshold, does not prevent buffers to become fully loaded, causing packets not belonging to the deflected flow to be dropped.

3.5.2 Second Iteration

In the attempt to improve drop rates, we proceed to the second iteration of Robinhood: we start deflecting multiple flows, still preferring the ones with highest number of bytes left to transmit and still exploiting the RFS information in the

Algorithm 1 Robinhood - First Iteration

```
1: procedure RobinhoodFirst(BS, BT)
2:     ▷ BS is the Buffer Size of the switch
3:     ▷ BT is the Buffer Threshold set by the algorithm
4:     ▷ [S_1, S_2, ..., S_n] are the candidate neighbors for offloading
5:     ▷ MaxRFSPacket is the packet with highest RFS value in the overloaded
   buffer
6:     Flow = MaxRFSPacket.FlowId
7:     N = LeastLoadedNeighbor in [S_1, S_2, ..., S_n]
8:     while BS > BT do
9:         ▷ P_i is incoming packet
10:        if P_i.FlowId = Flow then
11:            Deflect P_i towards N
12:        else
13:            Ignore P_i
14:            ▷ Ignore is intended as "ignore for deflection"
15:            ▷ P_i is put the corresponding output buffer
16:            ▷ or retransmitted or dropped
17:        end if
18:    end while
19: end procedure
```

header. The selection of the flows to deflect, follows the same approach of the first iteration: we pick the packet with highest RFS in the overflowing buffer, but this time we then proceed to pick also the second, the third, and so on.

Algorithm 2 Robinhood - Second Iteration

```
1: procedure RobinhoodSecond(BS, BT)
2:     ▷ BS is the Buffer Size of the switch
3:     ▷ BT is the Buffer Threshold set by the algorithm
4:     ▷ [S_1, S_2, ..., S_n] are the candidate neighbors for offloading
5:     ▷ Procedure is triggered as soon as BS > BT
6:     ▷ N_f is the number of flows in the overloaded buffer
7:     ▷ X is the percentage of flows to deflect
8:     NumberFlowsToDeflect = Math.floor(X*N_f)
9:     FlowsToDeflect = getMaxRFSPackets(NumberFlowsToDeflect)
10:    ▷ The flows of the top X packets with highest RFS values in the buffer
11:    StealingNeighbors = [ ]
12:    for flow in FlowsToDeflect do
13:        StealingNeighbors.append(random in [S_1, S_2, ..., S_n])
14:    end for
15:    while BS > BT do
16:        ▷ P_i is incoming packet
17:        if P_i.FlowId in FlowsToDeflect then
18:            Deflect P_i towards corresponding StealingNeighbor
19:        else
20:            Ignore P_i
21:            ▷ Ignore is intended as "ignore for deflection"
22:            ▷ P_i is put the corresponding output buffer
23:            ▷ or retransmitted or dropped
24:        end if
25:    end while
26: end procedure
```

The number of selected flow highly depends on the current number of flows in the buffer. We chose to deflect a percentage X of the current number of flows, to optimize performance. An optimal value for X was found to be around 50% of the flows: deflecting too many flows will result in the deflection target buffers themselves becoming overloaded. At the same time, deflecting fewer flows, will bring no noticeable difference from the first iteration. There are some cases, although rare, in which for instance the number of flows in the buffer is very limited, and the congestion is caused by mostly one single flow. In this case, picking half of the flows in the buffer, falling into a generalized first iteration, is the best choice;

deflecting flows that are contributing to congestion, while leaving other flows to be forwarded through their shortest path, in a Vertigo-style prioritization. This phase of the development led to interesting results: deflecting multiple flows has its repercussions on the selected stealing neighbors; if we stick to selecting the least loaded neighbor from the list, we end up overloading that same neighbor. The speed at which the deflection target fills up also depends on the X percentage value of the deflected flows. The more we deflect, the higher the probability and speed at which the neighbor becomes congested. **This observation led us to the conclusion that when deflecting multiple flows, randomly choosing multiple stealing neighbors is the best choice**, as Algorithm 2 shows. The neighbor selection is then performed together with the flows selection; distributing the flows randomly among the candidate neighbors. Any subsequent packet, belonging to one of the selected flows, will be deflected according to its relative entry in the Deflection Table.

At this point of the development, Robinhood is able to overcome ECMP and Vertigo in terms of average flow completion times. DIBS is still able to achieve better results, due to its behaviour in deflecting every single packet that comes to a congested switch in every available port buffer. DIBS does not take into consideration already congested switches, but aggressively forwards packets. Vertigo and Robinhood follow a more conservative approach, taking the congested neighbors into consideration even with different solutions, with the advantage that Robinhood deflects the same flow to the same switch, eliminating the need for a reordering component, thus achieving better flow completion times. This version of Robinhood still has the downside of not deflecting every single incoming packet, leaving packets that do not belong to deflected flows to be dropped, even if multiple flows are chosen.

3.5.3 Final Version

We then come to the final and last version of the protocol, in which, every incoming packet is deflected, respecting the *same flow - same switch* relationship, leading to outstanding results for flow completion times, which will be better analyzed in the relative section. As listed in Algorithm 3, as soon as a packet arrives, an overloaded switch will look into its Deflection Table to find an entry for the corresponding flow. If found, the packet is then deflected to the entry deflection target node. If, on the contrary, no entry is found, it will be created for subsequent packets of that flow, randomly selecting a neighbor from the candidate list. As mentioned, the random selection of the stealing neighbor at this point has its advantages. By preferring the least loaded neighbor, a larger quantity of flows, and subsequently packets, will be poured on it. From this point on, we will refer to Robinhood as in its final version.

Algorithm 3 Robinhood - Final Iteration

```
1: procedure ROBINHOOD(BS, BT)
2:     ▷ BS is the Buffer Size of the switch
3:     ▷ BT is the Buffer Threshold set by the algorithm
4:     ▷ [S_1, S_2, ..., S_n] are the candidate neighbors for offloading
5:     ▷ DefT is the deflection table for the current switch
6:     ▷ Procedure is triggered as soon as BS > BT
7:     while BS > BT do
8:         ▷ P_i is incoming packet
9:         if P_i is Steal Cancel then
10:            ▷ S_x is the sender of the Steal Cancel message
11:            removeEntry(S_x) in DefT
12:        end if
13:        if P_i.FlowId has entry in DefT then
14:            Deflect P_i towards corresponding stealing neighbor in DefT
15:        else
16:            Neigbor = random in [S_1, S_2, ..., S_n]
17:            DefT addEntry(P_i.FlowId, Neighbor)
18:            Deflect P_i towards Neighbor
19:        end if
20:    end while
21: end procedure
```

3.6 Example Scenario

In this section we present an example scenario in which Robinhood takes action, supposing we have a network defined as figure 3.2 shows.

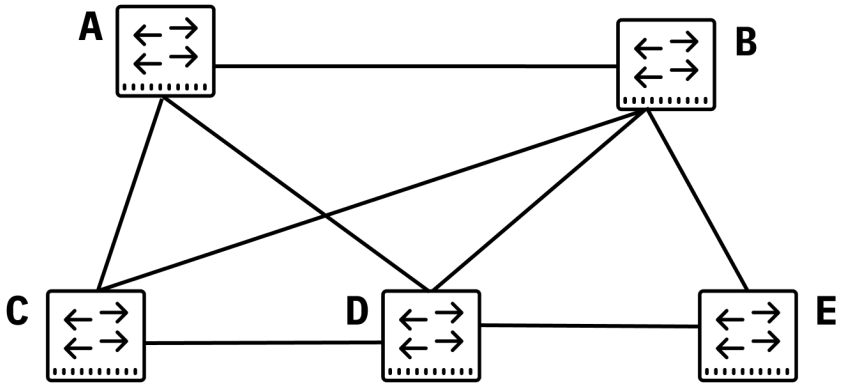

Figure 3.2: Network scheme for protocol workflow illustration

Supposing at the start no switch is overloaded, after exchanging Steal Request messages, each switch has its neighbors as a potential candidate for deflection. Table 3.1 reflects this situation.

Switch	Stealing Neighbors Candidates
Switch A	[B, C, D]
Switch B	[A, C, D, E]
Switch C	[A, B, D]
Switch D	[A, B, C, E]
Switch E	[B, D]

Table 3.1: Stealing Neighbors candidates list at first step.

Now suppose that switch B becomes overloaded. The first step of the protocol requires switch B to send an explicit Steal Cancel message to its neighbors, so that the offloading of packets into switch B is avoided. The sending of explicit steal cancel is not mandatory, since switch B will not send any further Steal Request messages and the expiry timeout on the neighboring switches will take care of removing it from the candidates list. In each case, table 3.2 reflects the situation.

Following the steal cancel message, switch B is now able to pick random neighbors for deflection. Sticking with the final version of Robinhood, we are deflecting every packet that comes to switch B, and with them, every different flow; leaving the

Switch	Stealing Neighbors Candidates
Switch A	[C, D]
Switch B	[A, C, D, E]
Switch C	[A, D]
Switch D	[A, C, E]
Switch E	[D]

Table 3.2: Stealing Neighbors candidates list after switch B is overloaded.

stealing neighbor selection random. An alternative could be to select the neighbors in a round-robin fashion, but as shown by simulations, since the pool of targets is relatively small, the mode of selection does not provide a significant difference, and at the end of the process, since we are deflecting multiple flow, multiple neighbors will be chosen either way. The important detail that switch B has to consider is the relationship between flows and switches to avoid and prevent reordering: **multiple flows can be deflected to the same switch, but never deflect the same flow to multiple switches.** Switch B will remember the associations made for this purpose in the already introduced Deflection Table, made of an entry for each different flow and the port where incoming packets should be deflected.

Flow	Stealing Neighbor
121	Switch C
99	Switch A
192	Switch C

Table 3.3: Example of Deflection Table for Switch B

Table 3.3 shows an example of what the Deflection Table of switch B could look like when deflecting multiple flows. However, the table is dynamic and subject to change, since neighboring switches statuses may change even during deflection. As line 9 of Algorithm 3 shows, if the switch receives a Steal Cancel message, (or timeout expiry) it will proceed to automatically find a replacement for the flow that steal cancel sender was deflecting. In this example every packet coming from flows 121, 99 and 192 will be deflected to switches C, A and C respectively. Supposing a steal cancel message is received, coming from switch A, switch B will promptly remove A from stealing neighbor candidates, and its corresponding associations in the Deflection Table; then since the association is removed, an incoming packet belonging to flow 99 will find no corresponding entry in the table, and then a newly random chosen switch will be selected, this time with switch A removed from the

candidate list and thus not available for possible selection. Every other packet, not belonging to flows 121, 99, and 192, will follow the same procedure: randomly choose an available neighbor and start deflecting the corresponding flow. With this more dynamic and reactive approach, Robinhood ensures that every packet gets deflected, but at the same time flow-based deflection avoids reordering steps. Robinhood is able to achieve improved performance compared to Vertigo and DIBS, but still with a higher drop rate, which suggests room for improvement, and will be analyzed in the performance evaluation section.

Chapter 4

Simulation Framework

4.1 Overview

The simulation framework, on which Robinhood and the other algorithms were tested, has been coded in the Python language, in an object-oriented fashion, making use of different libraries, especially *numpy* for helping with the collecting, tracking and calculation of performance metrics. The project is made up of three main components which are the event generation, the network core and the performance analytics which will be detailed in the following sections.

4.1.1 Event Generation

The event generation component takes care of generating traffic for the simulation. What is referred as traffic is made up of arrays of `Packet` objects (subclass of Event class), listed together in a bigger data structure called `universe`. Packets are generated with some common attributes such as `arrivaltime`.

```
1 class Packet(Event):
2     def __init__(self, arrival_time, service_time, src, dst, size,
    flow_id):
3         super().__init__(arrival_time, service_time)
4         self.id = id(self)
5         self.src = src  # Host
6         self.dst = dst  # Server
7         self.size = size  # Bytes
8         self.flow_id = flow_id # Flow
```

- **Arrival Time** and **Service Time** are the two parameters that are inherited from the `Event` parent class. Arrival time is the time at which the packet is

generated and the scheduled time at which its processing will start. Service time is the time required for a network node to process the packet, serving as a baseline for a comprehensive study on the queue being formed in network devices. The service time is what will actually be used by servers for processing time of a packet, and in the simulation is set as default to 0, indicating an immediate response of the server. What actually generates queues in the system are forwarding delays relative to switches and hosts, as we will see later.

- **Src** and **Dst**, as the name suggests are network devices source and destination of the packet. These two attributes will become relevant when generating flows and when defining the static paths of routing for the network. Devices are uniquely identified by their IP address which in this framework is a simple string.
- **Size** Packet are defined with a fixed size of 1512 byes.
- **Flow ID** Although a flow is uniquely defined by the tuple (srcIp, dstIp, srcPort, dstPort, protocol), having an id, generated in phase of flow generation, proves to be useful when it comes to deflecting decisions for switches.

The flow generation process, follows a straightforward approach. When generating traffic, the process is divided in two steps: generating a background load, and then generating load for the network load parameter, that goes from 0.3 to 0.8. To guarantee a varying load, with an increase in the parameters, follows an increase in packets generated per interval; at full load, a number of packets is generated every time unit so that buffers become full.

```
1     packets_to_fill_buffer = math.floor(math.floor(Config.buffer_size
      / Config.max_packet_size) * Config.network_load)
```

Flows are generated independently, since the number of sources and destinations is limited. Packets are then added to the flow, in a first step when generating background traffic, and then in a later step when generating varying load. One of the main parameters is also the arrival rate. Varying this value, will increase (or decrease) the "closeness" of events among each other. The event generator also includes a random seed parameter, coming from a more general simulation configuration, that randomizes the packets generation interval for varying load.

```
1 flow = Flow()
2 src = src_parameter else random.choice(NetworkUtils.hosts)
3 dst = dst_parameter else random.choice(NetworkUtils.servers)
```

```
flow.src = src
flow.dst = dst
for _i in range(flow_size)
    inter_arrival_time = random.expovariate(1/self.arrival_rate)
    arrival_time += inter_arrival_time
    packet = Packet(arrival_time, ..., flow.id)
    ...
    flow.packets.append(packet)
```

4.1.2 Network Core

The classes that compose the network core are mainly the ones regarding the network devices (switches, end hosts and servers), one class regarding static path generation (routing) and the controller classes. Controller classes are at the core of the simulation, since they provide the algorithms for handling full buffers and deflecting packets in each of the implemented algorithms (Vertigo, DIBS, ECMP and Robinhood), with the flexibility to add more whenever needed. We will talk in detail about controller classes implementation in their relative section, and about static routing in the network topology section, focusing in this paragraph on the device classes.

Hosts, switches and server classes, inherit from a base `ForwardingDevice` class, that implements some common methods and attributes. `ForwardingDevice` class itself inherits other device generic attributes, such as the IP address, from an abstract `NetworkDevice` class. General forwarding devices, include the implementation of `Port` objects for every reachable destination, populated in the static routing phase as we will later discuss, and a `ForwardingTable` implementation, listing entries for destination - corresponding port. Ports, are the objects that bring a queue mechanisms, that is intended as the buffer, one for each port of the switch.

```
class Port:
    def __init__(self, port_id, buffer_size=Config.buffer_size):
        self.id = port_id
        # ...
        # Buffer capacity
        self.buffer_capacity = buffer_size if buffer_size else Config
    .buffer_size
        self.buffer = []
        self.egr_buffer_size = 0 # Actual buffer size

    # ... Port methods ...

    def put_packet(self, packet):
        # Buffer size exceeds buffer capacity
```

```
14          if self.egr_buffer_size + packet.size > self.buffer_size:
15              raise FullBufferException()
16          else:
17              # ...
18              heapq.heappush(self.buffer, packet)
19              self.egr_buffer_size += packet.size
20
21      # ... Port methods ...
22
23  class NextHop:
24      def __init__(self, port, hop, distance, hops_to_destination=0):
25          self.port = port
26          self.hop = hop
27          self.distance = distance
28          self.hops_to_destination = hops_to_destination
29
30      # ...
31
32  class ForwardingTableEntry:
33      def __init__(self, dst, next_hop, port: Port, distance,
        hops_to_destination):
34          self.dst = dst
35          self.possible_next_hops: list[NextHop] = []
36
37      # ...
```

The way network devices they work, according to the simulation, is by constantly processing and/or handling an incoming packet, which means putting it in the relative port buffer according to the packet destination. In which port to put the packet, is a task delegated to the device's forwarding table, which is implemented in its own class. Once the packet has been put in the corresponding output buffer, two scenarios might present: the port is full, so we go on retransmitting or dropping the packet, or the port is not full, so we keep the packet in the buffer queue. Every simulation step, based on the device's state, so based on if there are packets to forward, or incoming packets to handle, the corresponding methods will be called.

```
1  # Forwarding Device class methods
2
3  def move_packet_to_output_buffer(self, packet, output_port=None):
4      # Output port parameter for manual forwarding
5      output_port = output_port or Controller.
       handle_moving_packet_to_output_buffer(self, packet)
6
7      try:
8          output_port.put_packet(packet)
9      except FullBufferException:  # Output port buffer is full
10         Controller.handle_full_buffer(self, packet, output_port)
```

```
11
12 # ...
13
14 def handle_moving_packet_to_output_buffer(device, packet):
15     possible_next_hops = device.forwarding_table.get_next_hops(packet
   .dst.ip_addr)
16     # ...
17     next_hop = possible_next_hops[0]  # Hops are ordered by distance
18     output_port = device.forwarding_table.get_port(next_hop.ip_addr)
19     # ... Eventual ECMP Handling
20     return output_port
```

Retransmission or drop of packets are handled by the inherited method of `ForwardingDevice` class. There are many options available when handling retransmission, depending on the parameters of the simulation; most of them are discussed later in the relative section; but for context, retransmission can be enabled indefinitely, enabled with a maximum attempt (preferred choice for the performance evaluation) or disabled completely. Note that retransmitting a packet will simply mean generating a new packet with the current simulation time + offset as arrival time, but will all the same characteristics (flowId, src, dst, service time).

```
1 def handle_retransmission(device, packet):
2     packet.departure_time = -1  # Set departure time to -1 to
   indicate that the packet was dropped
3     if Config.retransmission: # Retransmission enabled
4         # Schedule retransmission if Config.retransmission_attempts
   are not exceeded, ignore if Config.retransmission_attempts is -1
5         if packet.retransmissions < Config.retransmission_attempts or
    Config.retransmission_attempts == -1:
6             # print(f"Scheduling retransmission for packet {packet.id
   } from {device.ip_addr}")
7             device.schedule_retransmission(packet)
8         else:
9             print(f"DROPPING packet {packet.id}")
10            Statistics.update_dropped_packets()
11    else:
12        print(f"DROPPING packet {packet.id}")
13        Statistics.update_dropped_packets()
```

In the case of not full buffer, going on with the simulation, switches, which are the devices in charge of forwarding packets until they reach their destination server, will keep the packet in processing for a simulation time specified in the forwarding delay parameter (simulation parameters section). Once the time has passed, they will proceed with the forwarding. The packet has already been put in the corresponding output port for destination; and at that time an attribute on the packet class `nextHop` has been set. To simply forward the packet we

remove the packet from the output buffer and call the corresponding method for handling incoming packets on the next hop device. Note that at this stage, since we are handling packets one at a time, switches need to pick another packet to be processed from the queues they have. To focus on the algorithmic performances of Robinhood and other solutions we decided to schedule the next packet to be forwarded, from the port with most packets in queue. The scheduling mechanism is though implemented separately, allowing for easy editing whenever needed.

```
def schedule_next_packet(self):
    port = self.scheduler.schedule(self.ports)
    if port is None:
        # This means that all the ports have empty output buffers
        self.current_packet = None
        self.current_output_port = None
        self.current_next_hop = None
    else:
        self.current_packet = port.select_next_packet()
        self.current_output_port = port
        self.current_next_hop = self.forwarding_table.get_next_hops(
    self.current_packet.dst.ip_addr)[0]
```

4.1.3 Performance Metrics and Analytics

For collecting and tracking performance metrics we make use of a class composed mainly of static methods to make them accessible throughout the entire project. Most of the metrics are calculated at the end of the simulation, while data is collected while the simulation is running. Most meaningful metrics include:

- Number of dropped packets - with drop rate
- Number of retransmitted packets - with retransmission rate
- Generated packets
- Algorithms handles: increments each time a packet is handled by Vertigo, DIBS, ECMP or Robinhood, to have a better idea of how many times, a proposed solution comes into play.
- AVG Throughput: At the end of the simulation is a mean of throughput values. Each throughput value is updated every simulation step with the number of processed packets in that simulation step.
- AVG Flow Completion Times: At the end of the simulation is a mean of all the flow completion times collected (also at the end of the simulation). Dropped packets are not counted in this step.

- Buffer utilization data: are calculated considering the buffer utilization (in percentage) on each switch. The percentage of buffers being used is $B_i(t)$ where i is the switch and t the simulation time. Then aggregated metrics over time are the average buffer utilization for switch i: $AVG(B_i) = 1/T * \sum(B_i(t))$ where T is the final simulation time; and total average buffer utilization: $1/N * sum(AVG(B_i)$ where N is the total number of switches.

The `Statistics` class in which all these methods are grouped, also contains the code for saving the results in CSV format files and plot them accordingly.

4.2 Network Topology Generation

The topology generation is delegated to the BRITE [4] software. BRITE is a software developed by the Boston University that allows for a flexible and highly versatile topology generation. BRITE [4] works by generating a plane with a set of coordinates, placing nodes inside the plane, according to one of the chosen model and then connecting the nodes according to a variety of parameters, primarily preferential connectivity, degrees of a node, min/max bandwidth. Then it generates a list of nodes, and a list of edges that connect the nodes, complete with distance and other information. The models which are used for placing the nodes on the plane are the Waxman and the Barabasi-Albert model. One the key features of BRITE is its capability to generate independent but interconnected, two or higher level topologies. Parameters for the different network level can be set independently, having for instance a waxman based first level topology with 100 nodes and then a second level topology of 1000 nodes. The nodes of the same level topology are connected according to its relative parameters, while BRITE [4] allows for setting up ways of interconnecting the two levels with other values. Recursively generating, following this approach, will allow for higher than two-level topologies. We exploited BRITE [4] generation when generating the core of the network topology, which are switches. Using a two-level topology we are able to mark different level networks as two Autonomous Systems (AS); re-adapting this scenario in a data center network leaf-spine architecture this will translate in a first level of Spine switches, and a second level of Leaf switches, when using a Fat-Tree topology. Hosts and servers are kept separated from the BRITE-generated network, but are instead generated manually according to the simulation parameters, setting the exact value for how many hosts and how many servers are needed. Hosts and servers are then connected to switches accordingly, taking care of connecting hosts to a set of Spine switches and servers to a set of Leaf switches, in a round robin fashion. BRITE [4] runs as a standalone program, but once run, gives as an output a .brite file that will be parsed by a dedicated component in the simulation framework. A .brite file looks like this:

```
Topology: ( 12 Nodes, 17 Edges )
Model (5 - TopDown)
Model (3 - ASWaxman):   3 1000 100 1   2   0.15 0.2 1 1 -1.0 -1.0
Model (1 - RTWaxman):   4 1000 100 1   2   0.15 0.2 1 1 -1.0 -1.0

Nodes: ( 12 )
3    449 245 3    3    0    RT_BORDER
4    150 280 2    2    0    RT_NONE
5    598 115 4    4    0    RT_BORDER
6    302 70  3    3    0    RT_NONE
...

Edges: ( 17 )
2    5    3    197.73973    0.0 -1.0    0    0    E_RT_BACKBONE    U
3    5    4    477.4191     0.0 -1.0    0    0    E_RT_BACKBONE    U
4    6    4    259.23734    0.0 -1.0    0    0    E_RT_BACKBONE    U
5    6    5    299.40106    0.0 -1.0    0    0    E_RT_BACKBONE    U
6    3    6    228.54759    0.0 -1.0    0    0    E_RT_BACKBONE    U
...
```

Each generated node has an assigned ID, xposition and yposition in the plane, number of incoming connections (inDegree), number of outgoing connections (outDegress), AS ID, and a node type, in order from left to right. The node type, is the attribute that will be used to separate different types of switches. Edges instead are generated together with an assigned id, start and end node, edge length, bandwidth, transmission delay, AS-from and AS-to ids, if a two-level model is used, and an edge type. As we can see, BRITE [4] allows for highly customized networks, making it possible to test Robinhood and other algorithms in different environments.

```
# ... Parser object class

def parse_node(self, line):
    node_id, _xpos, _ypos, _indegree, _outdegree, as_id, node_type =
    line.split()
    self.nodes.append(Node(node_id, as_id, node_type))

def parse_edge(self, line):
    edge_id, start, end, length, delay, bandwidth, as_from, as_to,
    edge_type, _whatisthis = line.split()
    self.edges.append(Edge(edge_id, start, end, length, bandwidth,
    delay, as_from, as_to, edge_type))
```

Exploiting nodes and edges, generated by BRITE [4], once parsed, the simulation framework takes care of generating static paths for connecting hosts to servers.

The `Routing` class handles these operations firstly by creating an adjacency list of the graph and then running well-known shortest paths algorithms such as Dijkstra (or the Yen-K variant) to find the top-k shortest paths for each host-server pair. As a result of running these algorithms, the forwarding tables of the devices (including switches) gets to be populated with the found destinations along with a hop number and distance. These values, such as hop number and distance are hold in the `Port` class, which is the main component of an entry in a forwarding table, along with a queue acting as an output buffer.

```
def dijkstra_paths(self, src, dst, k=10):
    priority_queue = [(float(0), src.ip_addr, [], 0)]  # (distance, node, path, hops)
    top_k_paths = []

    while priority_queue and len(top_k_paths) < k:
        (current_distance, current_node, current_path, hops) = heapq.heappop(priority_queue)
        # If the current node is the destination, return the path
        if current_node == dst.ip_addr:
            top_k_paths.append((current_path + [current_node], current_distance, hops))

        # Check neighbors for the current node
        for neighbor, weight in self.adjacency_list[current_node]:
            if neighbor not in current_path:
                heapq.heappush(priority_queue, (current_distance + float(weight), neighbor, current_path + [current_node], hops + 1))

    return top_k_paths
```

4.3 Algorithms Simulation

What makes the simulation take key decisions when handling full buffers or deflection, are the implemented algorithms controllers. Each of them, depending on the algorithm, goes into action in different moments and parts of the simulation. Vertigo and DIBS for instance, replace the standard behaviour of retransmitting or dropping a packet when arriving at a full buffer, ECMP and Robinhood instead operate preemptively, making the necessary routing and forwarding decisions.

4.3.1 ECMP

ECMP's forwarding decisions take action whenever packets needs to be handled by a switch, before actually moving them into the corresponding output buffer for

their destinations. The switch, after finding out all the possible next hops for the packet destination, forms a list of the next hops with the same cost, defined in the simulation as the number of hops or distance, based on parameters. Then, several inputs, such as packet's source and destination address are given to an hashing function (CRC32), together with a random integer between 0 and 100. A modulo N operation is then performed on the result of the hashing function, where N is the number of equal cost paths found for the packet's destination, giving as a result the index of the chosen hop. In the real implementation of ECMP, other inputs are provided, such as source and destination port, but since in this simulation they are not considered, a random number is required, otherwise packets belonging to the same flow will always be redirected to the same next hop, making the ECMP process useless.

```
def ecmp_handle(device, packet):
    possible_next_hops = device.forwarding_table.
    get_equal_cost_next_hops(packet.dst.ip_addr)
    next_hop = ECMPController.ecmp_hash(packet, possible_next_hops)
    output_port = device.forwarding_table.get_port(next_hop.ip_addr)
    packet.next_hop = next_hop
    Statistics.ecmp_handles += 1
    return output_port
```

```
def ecmp_hash(packet, possible_next_hops):
    hash_input_1 = packet.src.ip_addr
    hash_input_2 = packet.dst.ip_addr
    hash_input_3 = packet.remaining_flow_size
    hash_input_4 = random.randint(0, 1000)  # Random number to
    differentiate the packets
    hash_input = [hash_input_1, hash_input_2, hash_input_3,
    hash_input_4]

    hash_result = zlib.crc32(str(hash_input).encode('utf-8'))  # Hash
    the packet
    next_hop_index = hash_result % len(possible_next_hops)
    return possible_next_hops[next_hop_index].hop
```

As mentioned, ECMP handles incoming packet to a switch, even if not congested, whenever needs to be moved to the corresponding output buffer, as you can see from the code snippet in the 4.1.2 paragraph. Robinhood also operates preemptively, but its deflection decisions are delegated to switches, actually moving the packet to an output port passed as a manual parameter. Vertigo adn DIBS instead operate as soon as buffers become full.

```
def handle_moving_packet_to_output_buffer(device, packet):
    possible_next_hops = device.forwarding_table.get_next_hops(packet
    .dst.ip_addr)
    # ...
    if Config.ecmp:
        output_port = ECMPController.ecmp_handle(device, packet)
    return output_port
```

4.3.2 Vertigo

Vertigo's approach, differing from ECMP's, operates directly when handling full buffer exceptions. The implementation of Vertigo is not carried on only in its relative controller, but also includes other attributes and mechanisms that has to be considered, such as the remaining flow size info in a packet and the halving of such value when a packet is dequeued. So, as for Vertigo, other packet attributes are needed. When a packet comes at a full buffer, the simulation standard behaviour is to raise a `FullBufferException`, and subsequently retransmit of drop the packet. Vertigo (and DIBS too) intervenes in this exact moment, instead of dropping the packet, the controller takes action and performs the task required by the algorithm. The packet with the highest RFS value is dequeued from the buffer making space for the newly arrived packet; then the least loaded port among two of the possible next hops is selected and the max RFS packet is enqueued, if possible, in this selected port. If still the least loaded port buffer is full, then another exception is triggered, but this time with a set parameter indicating to ignore Vertigo, in order to proceed with the retransmission or eventual drop.

```
def vertigo_handle(device, packet, output_port):
    # Dequeue max RFS packet
    max_rfs_packet = output_port.get_max_rfs_packet()
    output_port.remove_packet_from_buffer(max_rfs_packet)
    # Enqueue new packet
    output_port.put_packet(packet)
    # ...
    # Find other 2 possible next hops for the max RFS packet
    possible_next_hops = device.forwarding_table.get_all_next_hops()
    # ...
    # Get the ports for the other possible next hops (except the
    current next hop)
    random_ports = random.sample(possible_next_hop_ports, 2)

    # Choose the least loaded port from the two random ports
    least_loaded_port = min(random_ports, key=lambda port: port.
    egr_buffer_size)
```

```
16
17      try:
18          # Put the max RFS packet in the least loaded port
19          max_rfs_packet.remaining_flow_size = int(max_rfs_packet.
    remaining_flow_size / 2)
20          least_loaded_port.put_packet(max_rfs_packet)
21          Statistics.vertigo_handles += 1
22      except FullBufferException:
23          # If the least loaded port is full, drop the packet/schedule
    retransmission
24          Controller.handle_full_buffer(device, max_rfs_packet,
    output_port, vertigo_param=False)
```

4.3.3 DIBS

Same as Vertigo, Detour-Induced Buffer Sharing algorithm executes whenever a packet comes to a full buffer, raising a `FullBufferException`. The procedure is then, for a switch, to move the packet to another random port, making sure to exclude: ports that are directly connected to end hosts or servers, and ports whose buffer is already full. Instead of Vertigo, since we are excluding full buffers from the port selection, exceptions are not occurring, what might happen is instead that when the packet will be forwarded might encounter other full ports and so the procedure is repeated until packets reach their destination. Another possible outcome for DIBS procedure is that there are no available ports for deflection. In this case, the network controller will take care of handling the full buffer, exploiting the same strategy as for Vertigo, which is setting a parameter to indicate to ignore DIBS intervention.

```
1  def dibs_handle(device, packet, output_port):
2      available_ports = []
3      for port in device.ports:
4          if port.is_overloaded():
5              continue
6          if port.other_end_device().is_switch():
7              available_ports.append(port)
8
9      if len(available_ports) == 0:  # No ports are available to move
    the packet
10         Controller.handle_full_buffer(device, packet, output_port,
    dibs_param=False)
11
12     Statistics.dibs_handles += 1
13     port = random.choice(available_ports)
14     port.put_packet(packet)
```

4.3.4 Robinhood

Given the articulate and preliminary steps required to implement Robinhood protocol, the main deflecting action is performed directly at switch level, while all the preparatory steps, such as the updates on deflection candidates list are performed every simulation step in the Robinhood controller class. Every time the simulation is advanced, a loop goes through all the switches to find out under-loaded and overloaded ones, taking different actions based on the status. Overloaded switches will send Steal Requests (SR) in the form of an help offer to neighboring switches, which will take care of inserting the sender in the deflection candidates list. Overloaded switches will instead send a Steal Cancel message, requiring neighbors to remove them from the possible deflection targets. In the simulation, the use of a direct approach is preferred, instead of carrying on specific packet implementations, since as previously mentioned, these messages can be carried on by a single IP state message, introducing minimal overhead.

```
def activate_robinhood(device, packet, output_port):
    for switch in self.network.switches:
        if switch.is_overloaded() is None:
            self.send_steal_request(switch)
        else:
            self.send_steal_cancel(switch)
```

```
def send_steal_request(self, switch):
    switch_neighbors = self.network.get_neighboring_switches(switch)
    for neighbor in switch_neighbors:
        if switch not in neighbor.neighbors_that_can_help:
            neighbor.neighbors_that_can_help.append(switch)
```

```
def send_steal_cancel(self, switch):
    neighbors = self.network.get_neighboring_switches(switch)
    for neighbor in neighbors:
        if switch in neighbor.neighbors_that_can_help:
            neighbor.neighbors_that_can_help.remove(switch)
```

The `isOverloaded()` method, checks whether a switch has at least one buffer surpassing the size threshold set by Robinhood. The result of the check, not only broadcasts the adequate message to neighbors, but is also the trigger to initiate the deflection process. Every switch, together with the candidates list, keeps also track of the flows it is currently deflecting (Deflection Table), in a specific switch attribute, made up of entries on the `DeflectingTo` class.

```
class DeflectingTo:
    def __init__(self, flow_id=None, switch=None):
        self.flow_id = flow_id
        self.switch = switch
```

When the process starts, the candidate list is empty, but as soon as initiated, every incoming packet will create an entry in the Deflection Table if not already present, with the flow it belongs to and the randomly chosen deflection target. If an entry for the flow is already present in the table, the switch then proceeds to move the packet into the corresponding output buffer of the target neighbor.

```
def robinhood_deflect_multiple(self, packet):
    for deflecting_item in self.deflecting_to_multi:
        if packet.flow_id == deflecting_item.flow_id:
            self.deflect_packet(packet, deflecting_item.switch)
            return

    # If flow is not already in deflection
    stealing_neighbor = self.pick_stealing_neighbor()
    self.deflecting_to_multi.append(DeflectingTo(packet.flow_id,
    stealing_neighbor))
    output_port = self.forwarding_table.get_port(stealing_neighbor.
    ip_addr)
    # Move with forced output port
    self.move_packet_to_output_buffer(packet, output_port)
```

Early versions of Robinhood, follow a different approach, based on the iteration. Both the first and second iteration, though, make use of the RFS value, of Vertigo, selecting a single, or multiple flows based on this value.

```
# ... First Iteration
def robinhood_activate_first_iteration_deflection(self, packet):
    least_loaded_neighbor = sorted(switch.neighbors_that_can_help,
    key=lambda x: x.get_load())[0]
    switch.deflecting_to.switch = least_loaded_neighbor
    # Get the flow id of the packet with the highest remaining flow
    size
    port = switch.is_overloaded()
    flow_id = port.get_max_rfs_packet().flow_id
    switch.deflecting_to.flow_id = flow_id
```

```
# ... Second Iteration
```

```
def robinhood_activate_second_iteration_deflection(self, packet):
    max_rfs_packets = RobinHoodController.
    get_max_rfs_packets_of_flows(switch)
    for packet in max_rfs_packets:
        stealing_neighbor = random.choice(switch.
    neighbors_that_can_help)
        deflecting_to = DeflectingTo(packet.flow_id,
    stealing_neighbor)
        switch.deflecting_to_multi.append(deflecting_to)

def get_max_rfs_packets_of_flows(switch):
    port = switch.is_overloaded()
    # Count different flows in overloaded port
    different_flows = []
    for packet in port.buffer:
        if packet.flow_id not in different_flows:
            different_flows.append(packet.flow_id)
    total_flows = len(different_flows)
    flows_to_deflect = math.ceil(total_flows * 0.5)
    # Get top flows_to_deflect packets with the highest different
    remaining flow sizes
    max_rfs_packets = port.get_top_k_max_rfs_packets_different_flows(
    flows_to_deflect)
    # print(f"Num max_rfs_packets: {len(max_rfs_packets)}")
    return max_rfs_packets
```

4.4 Simulation Parameters

As suspected, the simulation revolves around a multitude of parameters to allow for an highly customized environment. They are stored in an .ini file which is loaded at the start of the simulation and populates a static `Config` class that enables access to the values, everywhere in the simulation. Following there is a list of the most significant and adopted ones.

- **Debug parameters**: multiple boolean values that enable/disable the printing of statements, drawing of network nodes and plots the generated packets
- **Algorithm parameters**: multiple boolean values that enable/disable a single algorithmic choice among Vertigo, ECMP, DIBS and Robinhood. Algorithms should be enabled one at a time.
- **Robinhood modes**: integer value for enabling/disabling the execution of a different version of the iterations of Robinhood protocol.
- **Arrival Rate**: defines the "closeness" of interarrival times when generating packets.

- **Service Time**: time (in simulation units) required by a server to process a packet.
- **Forwarding Delays**: one for hosts and one for switches, defines the time a host or a switch takes to process and forward a packet.
- **Retransmission parameters**: these include a retransmission timeout, which is an offset for when to reschedule retransmitted packets after the current simulation time; a retransmission attempts value for indicating how many times a packet should be retransmitted before it gets dropped
- **Number of Hosts and Servers**: integer numbers indicating the number of hosts and the number of servers which will be generated and attached to switches.
- **Network load**: float value ranging from 0 to 0.8
- **Packet Size**: set as default to 1512 bytes
- **Buffer Size**: capacity of a single buffer, set as default to 300KB
- **Spine Buffer Size**: capacity of a single buffer of a Spine or Core switch, set to default as Buffer Size but could be increased, since these switches are the ones directly attacched to hosts.
- **Buffer Threshold**: the value for Robinhood buffer threshold is set as default to 250KB, about 15% less of the Buffer Size
- **Background Traffic**: positive value to enable/disable background traffic and its intensity; set as default to 200 which means that each flow generated contains at least 200 packets.
- **Max and Min Flow Size**: used in case of fixed flow sizes. Flow size is inteded as the number of packets belonging to the flow.
- **Brite File**: input topology file for the parsed, from which nodes and edges are generated.
- **Results File**: Output file in which to collect performance metrics.

Chapter 5

Performance Evaluation

We evaluate Robinhood using a custom built network simulation framework, under the same conditions where Vertigo and other approaches have been tested, which are Fat-Tree and equivalent Three-Tiered architectures with a varying number of hosts and servers, a varying network load, supported by background traffic. Summarizing our findings:

- Robinhood achieves better throughput under various degrees of load, comparing to Vertigo, DIBS and ECMP.
- The flow completion times are improved by 22% compared to ECMP, 6% and 7% compared to Vertigo and DIBS, respectively.
- The improvement on FCT, comes at the cost of a reduced average buffer utilization after which follows a higher drop rate.

5.1 Simulation Setup

5.1.1 Network Topologies

We perform our simulations using the BRITE [4] software for generating the topologies. BRITE works by placing nodes inside of a plane, and then connecting the nodes according to a variety of parameters, primarily preferential connectivity, degrees of a node, min/max bandwidth. The main strength of BRITE software is that it allows for multi-level topologies: in our simulations we mainly used 2-level hierarchically generated networks, in order to map Spine switches, directly connected to hosts, and Leaf switches, connected to servers. The primary used topology for the simulation is a Fat-Tree topology with 4 1-level switches and 8 2-level switches. We also validate our findings against an equivalent three-tiered architecture with 4 core and 8 aggregate switches. We also tested fully and randomly

connected topologies, proving that Robinhood performs well under circumstances that may be different from the ones in data centers. Hosts, which are in charge of sending the packets into the network, and servers, which are in charge of the final processing, are detached from the topology generated by BRITE, to maintain a highly customized network.

5.1.2 Workloads

For testing different network loads, we regulate the incoming packets and flows based on the switch buffer capacity. At full load, the hosts periodically (at each simulation step) send packets to servers for a fixed time interval, with the number of packets per switch matched to fill its buffers. The time interval is randomly chosen, so the simulations are repeated for different random seeds, to validate the results. An alternative approach, would be to modify the arrival rate of the packets, operating on the inter-arrival times, to make the same number of packets to be generated with shorter time difference. Also, additionally to the varying load, in every simulation there is a background traffic set to 20% of the full load, in order to keep nodes busy and more easily manipulate the load. The background traffic is distributed across multiple, already defined flows. The final network load is the sum of the background load + the variable load, that for every test goes from 30% to 80%, making and aggregate load of 100% at full capacity.

5.2 Results

5.2.1 Early Versions

In the first iteration of the protocol, where we deflect one single flow for an overloaded switch, we can see a positive improvement compared to ECMP, in terms of average FCT, as shown in Figure 5.1. The average flow completion times improved by 10% with respect to ECMP results at 80% network load, but with a slightly higher drop rate, 5.8% drops compared to ECMP 5.4% as figure 5.3 shows. This is primarily because of the single flow deflection, leaving any packet that does not belong to the deflected flow to experience re-transmission or drops. ECMP lacks the ability to react to bursts in real-time, thus exacerbating congestion in some cases. Robinhood instead takes notice of non-congested switches and paths when making deflection decisions, improving the final flow completion time. This early version also takes the RFS into consideration, making a sizeable difference when deflecting a single flow, picking the one with most bytes left to transmit. As already noted, this approach leaves many packets to be dropped, but at the same time, non-deflected packets follow their shortest path towards their destination.

Vertigo and DIBS still perform better than Robinhood in every metric, given that they deflect every incoming packet by nature.

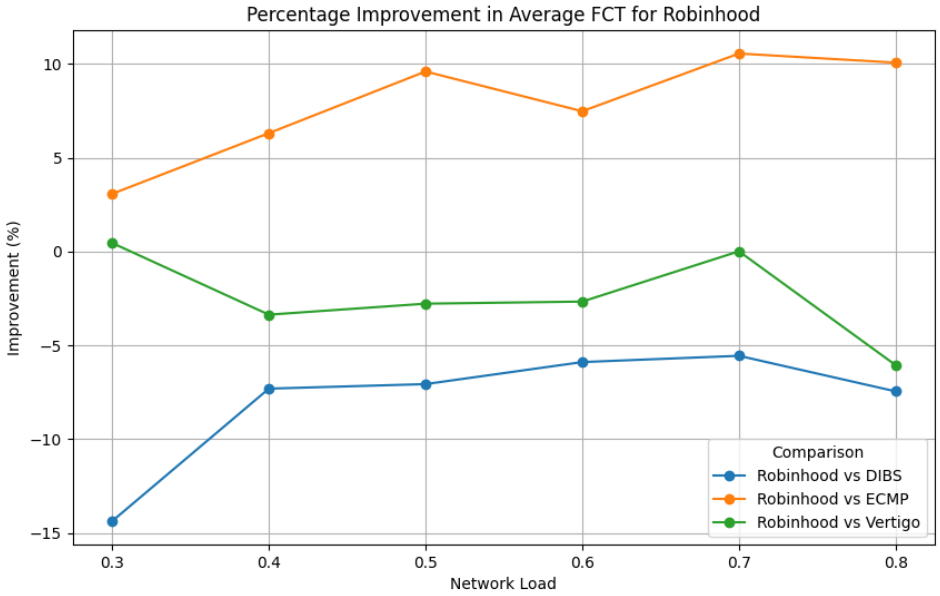

Figure 5.1: Robinhood (First Iteration) Improvement over other algorithms in Average FCT. Major improvements are only noticeable against ECMP, with a 22% improvement at full network load, while Vertigo and DIBS perform better at any load.

Then, starting to deflect multiple selected flows we can see an advantage in terms average fct even when compared to Vertigo. In this stage where we deflect a fixed percentage of flows, the buffer threshold mechanism of Robinhood prevents multiple buffers from becoming fully overloaded. Incoming packets are following the shortest path by nature, thus, preemptively deflecting flows that are more likely to contribute to congestion and leaving packets with small remaining flow size to follow their best path, can have a considerable impact on the overall delivery time. Another contribution to the performance is also given by the avoidance of packet reordering which provides a consistent overhead for Vertigo. As we can see from figure 5.4 the improvement gap provided by Robinhood is reduced and brought to zero when the network is fully loaded. In such conditions, buffers at many switches are likely to be frequently overloaded or close to their thresholds. Robinhood's buffer threshold mechanism becomes less effective as most buffers are likely to surpass these thresholds, leading to a situation where almost all switches are no longer candidates for deflection. Despite the improvement on flow completion times,

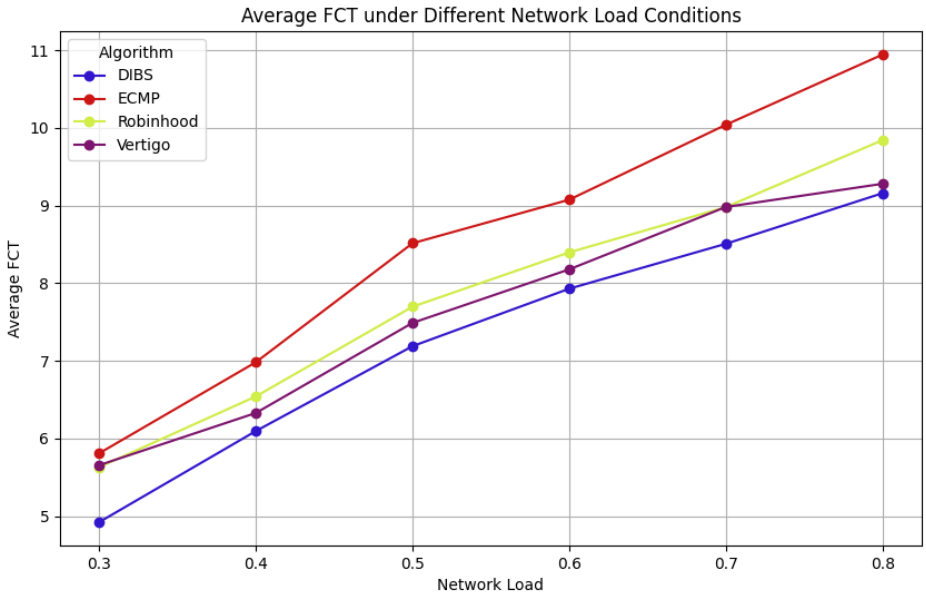

Figure 5.2: Average FCT - First Iteration. Robinhood place itself between Vertigo and ECMP. Performance on average flow completion time in the first iteration are almost equal among the three at smaller loads.

drop rates are still high. The limited number of flows still has its own repercussions on the remaining non-deflected flows; also the limited number of deflection targets sets the foundation for the biggest downside of Robinhood protocol.

5.2.2 Major Results

Putting aside the early versions of Robinhood, we now focus on the results of the final version, deflecting every incoming packet towards a flow-based neighbor. The hints of the previous iterations gives us a major improvement of the flow completion times. Analyzing the results coming from the simulation we can state that:

Robinhood deflection is resilient to the scale and flow size of network load.

As figure 5.7 shows, Robinhood achieves great performances at different network loads, proving its ability to effectively manage microbursts and prolonged periods of congestion, improving flow completion times. At lower loads, DIBS performs slightly better than Robinhood. The reason is that DIBS randomly chooses a port for deflection among all the switch ports, giving a more effective load balancing

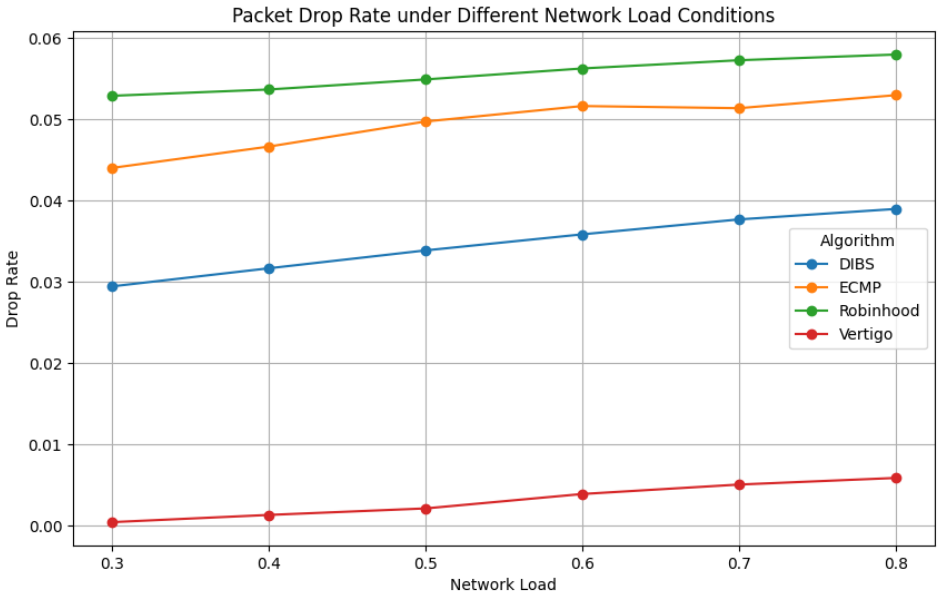

Figure 5.3: Drop Rate - First Iteration

and parallelization of packet forwarding. At higher loads, the detour produced by DIBS, leads to packets being deflected multiple times, even coming back to the first-deflection switch, before reaching their destination. As already pointed out, DIBS research [2] shows packets being deflected even 15 times, going back and forth among multiple switches, causing multiple loops that find an end only when congestion is over. Packet drops are prevented using DIBS approach, since packets are constantly detoured, but the delayed delivery is an unavoidable consequence.

Robinhood is consistent in average fct improvement

The cumulative distribution function plot (figure 5.9), shows Robinhood's curve residing generally left to the others, showing better performance. The curve is also generally steeper than the others, showing significant consistency in the majority of the flows. These results are achieved by the congestion avoidance mechanism that Robinhood carries on thanks to the buffer threshold. The main point of other algorithms is to react to microbursts and congestion in real-time, while Robinhood, also reacting in real-time, provides a smarter avoidance of congestion, in cases where it is limited to a smaller time frame. Also, without depending on the number of flow, or network load, or network architecture, Robinhood is consistent in improving average flow completion times thanks to its limited and parallelized

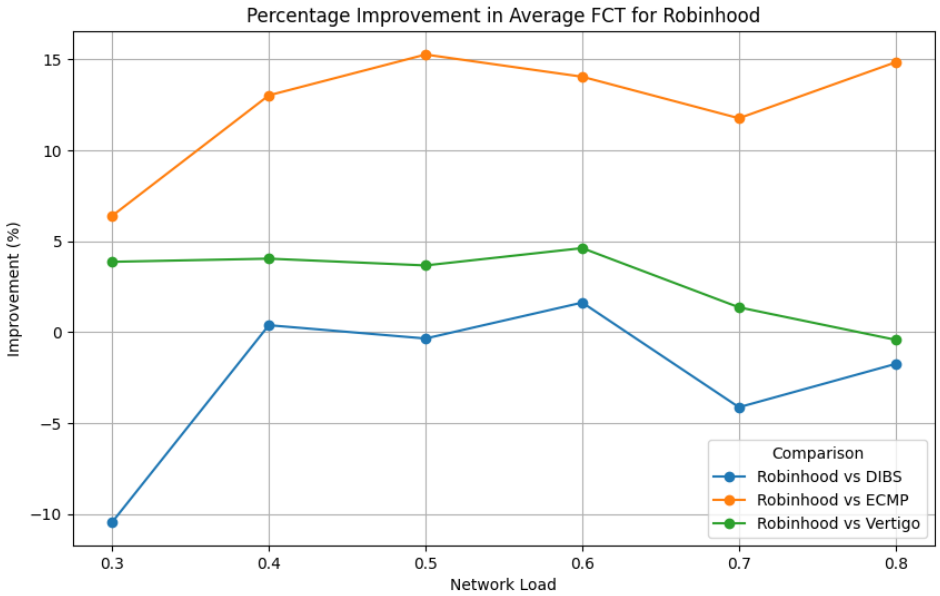

Figure 5.4: Robinhood (Second Iteration) Improvement over other algorithms in Average FCT.

capability of packet processing, allowing packets to reach their destination faster, without unnecessary detours like DIBS or established paths like ECMP.

Robinhood significantly improves throughput

While throughput for other algorithms like DIBS, ECMP, and Vertigo remains relatively stable or fluctuates slightly, Robinhood shows a clear advantage with higher throughput values, maintaining a noticeable lead over the others as seen in figure 5.10. This indicates that Robinhood not only handles microbursts effectively but also significantly enhances overall network speed and performance. This result is directly bound to the improvements on average flow completion times and vice-versa; the selective deflection on specific under-loaded neighbors, allows packets to be processed more frequently, even if not following the direct shortest path; proving that deflecting has a significant advantage in periods of congestion.

Drop rates are the major downside

Packet drop causes differ among the different iterations of Robinhood. In the early stages of development, the majority of drops were being caused by all the non-deflected flows coming to a full buffer. As for the final version, the primary

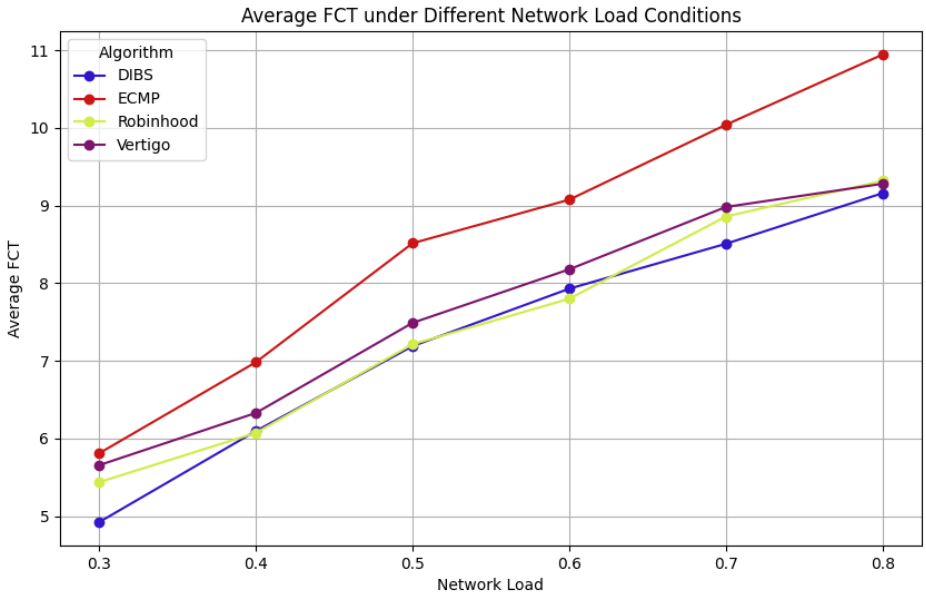

Figure 5.5: Average FCT - Second Iteration

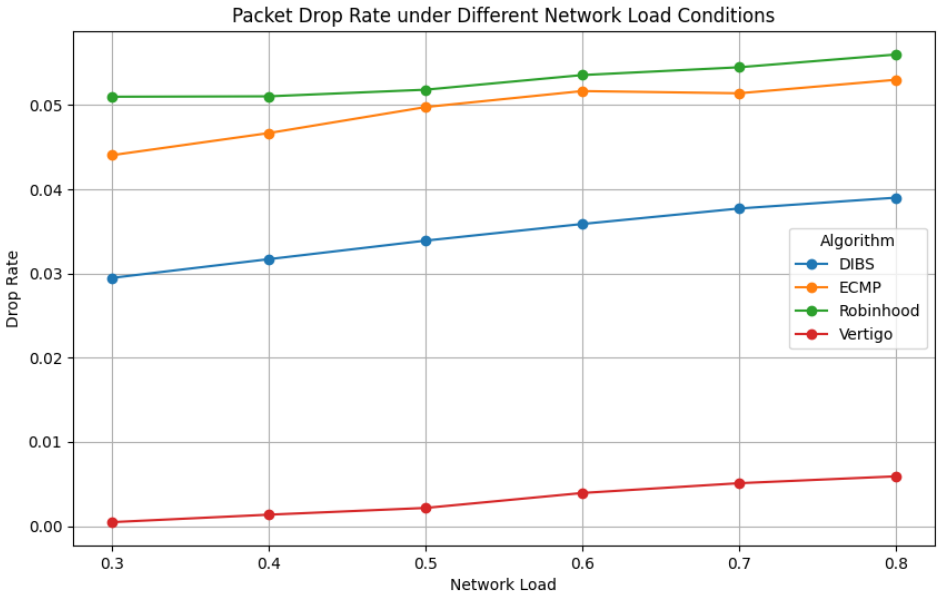

Figure 5.6: Drop rate - Second Iteration

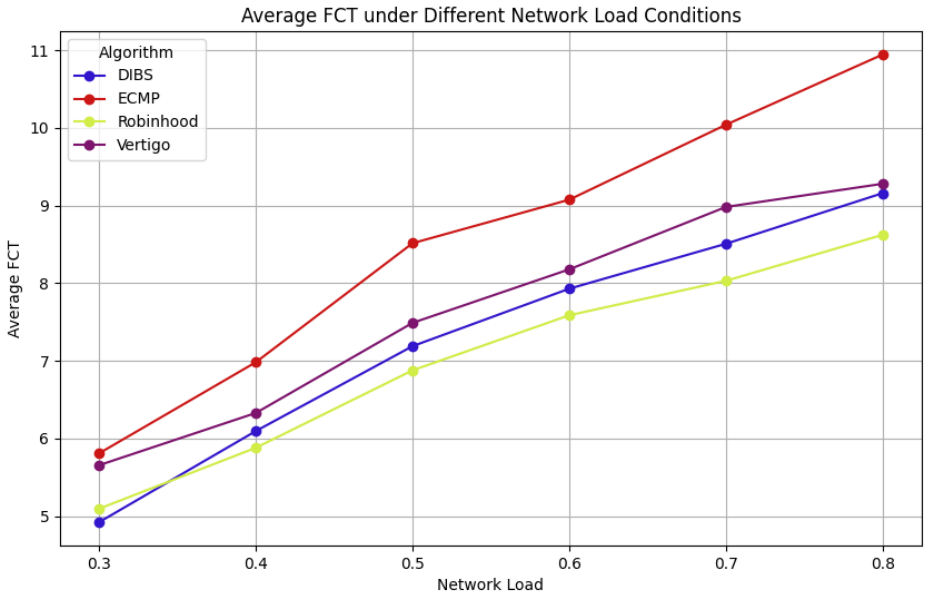

Figure 5.7: Average FCT under various degrees of load.

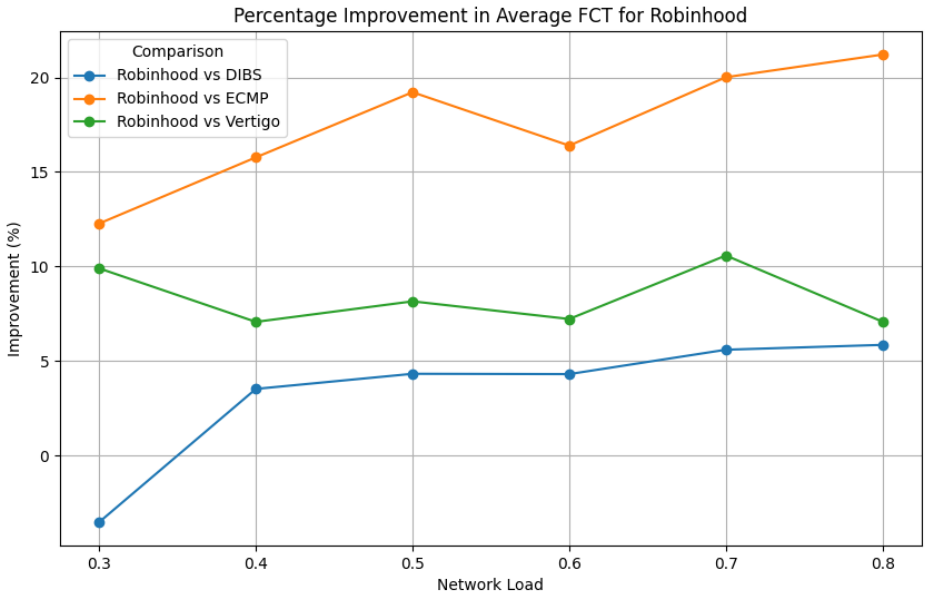

Figure 5.8: Average FCT under various degrees of load.

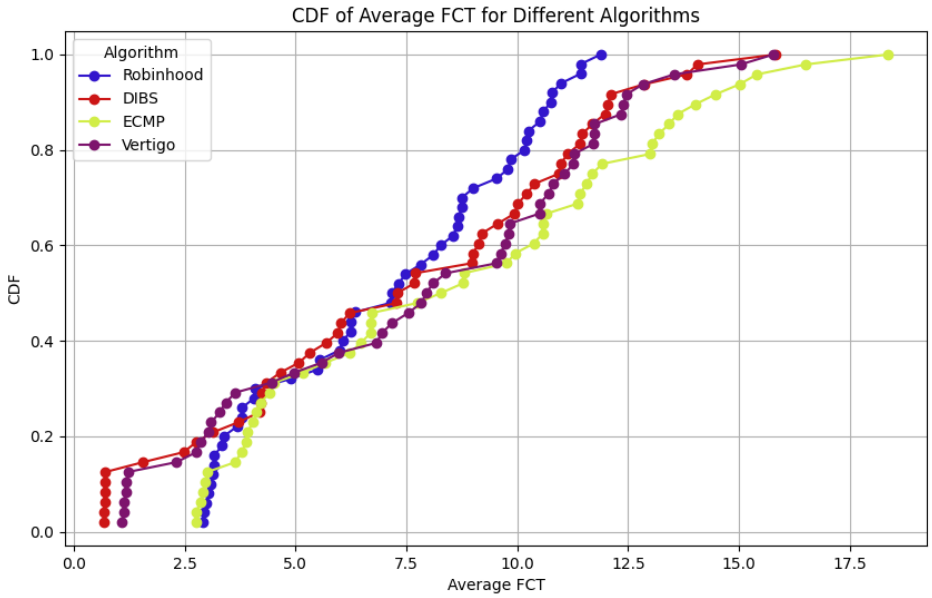

Figure 5.9: Robinhood and other algorithms, average FCT - CDF

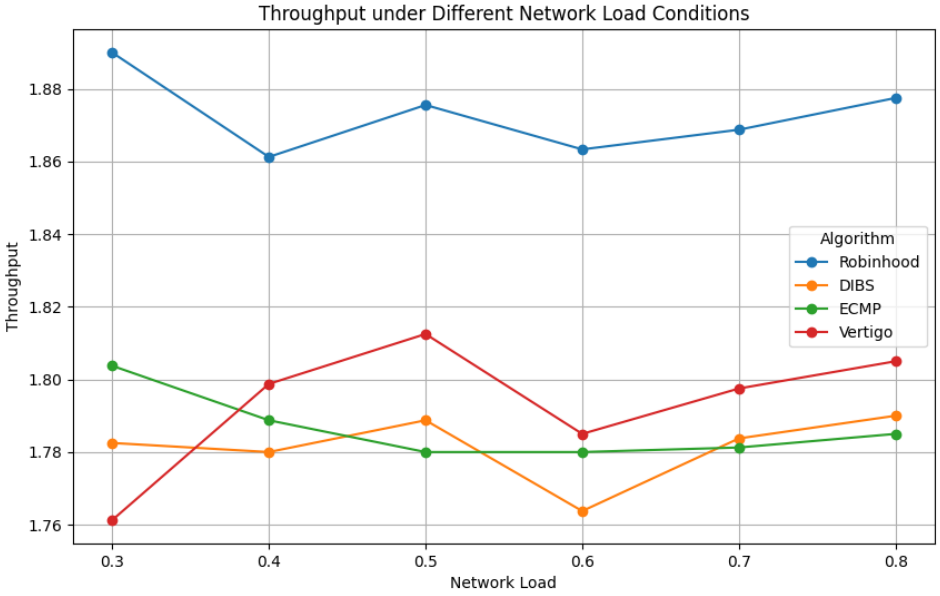

Figure 5.10: Throughput under various degrees of load.

reason behind this behaviour, deflecting every packet, is the neighbor-selection based deflection performed by Robinhood. Having a small, almost fixed, sample deflection candidates compared to Vertigo or DIBS, the packet drops are simply redistributed across multiple buffers, but not effectively managed. What happens most of the time, is that overloaded switches, deflecting every incoming packet, end up overloading other buffers when deflecting. The threshold has already been surpassed, so we end up emptying the buffer until it is below the threshold, then, becoming an available candidate, it will receive packets from neighboring switches, surpassing the threshold again. Other algorithms have every switch port as a candidate for deflection, so they pick their best target on a larger pool of options. Improvements on packet drop rates are present between the different stages of Robinhood, but still higher than ECMP even in the final version.

5.3 Future Development

Despite the improvements on throughput and average flow completion times, the performance boost comes at the cost of higher drop rates. This trade-off makes Robinhood more suitable in environments where more drops are tolerated, but more efficient delivery is needed. One of the main reasons behind this behaviour is the fixed value for the threshold; we believe that with an adaptive value, coming from a predicting machine learning model based on the collected data, drop rates are likely to improve. An adaptive buffer threshold allows the protocol to adjust to varying network conditions in real-time, so when network congestion increases, the threshold can be lowered to trigger deflections earlier, preventing buffers from becoming critically overloaded and thus reducing the risk of packet drops; when instead network congestion is at lower values, the threshold can be kept high enough to prevent Robinhood from activating, thus leaving packets to follow their shortest path without any major deflection. Future development of Robinhood also includes the implementation on modern programmable switches, using P4, which would consist in the deployment of exchange messages, setting the specific ToS field of IP; two data structures holding the deflection candidates list and the Deflection Table and the implementation of the deflection itself.

Chapter 6

Appendix

Simulation software:
`https://github.com/lorenzopantano/event-driven-simulator.git`
BRITE software:
`https://www.cs.bu.edu/brite/`

Bibliography

[1] Sepehr Abdous, Erfan Sharafzadeh, and Soudeh Ghorbani. «Burst-tolerant datacenter networks with Vertigo». In: *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 1–15. ISBN: 9781450390989. DOI: 10.1145/3485983.3494873. URL: https://doi.org/10.1145/3485983.3494873 (cit. on pp. 2, 6–8).

[2] Kyriakos Zarifis, Rui Miao, Matt Calder, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. «DIBS: just-in-time congestion mitigation for data centers». In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. ISBN: 9781450327046. DOI: 10.1145/2592798.2592806. URL: https://doi.org/10.1145/2592798.2592806 (cit. on pp. 2, 6, 8, 9, 43).

[3] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. «Efficient Scheduling Policies for Microsecond-Scale Tasks». In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 1–18. ISBN: 978-1-939133-27-4. URL: https://www.usenix.org/conference/nsdi22/presentation/mcclure (cit. on pp. 2, 10).

[4] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. *BRITE: Universal Topology Generation from a User"s Perspective*. Tech. rep. USA: Boston University, 2001 (cit. on pp. 3, 29, 30, 39).

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. «A scalable, commodity data center network architecture». In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM '08. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 63–74. ISBN: 9781605581750. DOI: 10.1145/1402958.1402967. URL: https://doi.org/10.1145/1402958.1402967 (cit. on p. 4).

[6] Sepehr Abdous, Erfan Sharafzadeh, and Soudeh Ghorbani. «Practical Packet Deflection in Datacenters». In: *Proc. ACM Netw.* 1.CoNEXT3 (Nov. 2023).

DOI: 10.1145/3629147. URL: https://doi.org/10.1145/3629147 (cit. on pp. 5, 7, 10).

[7] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. «Data center TCP (DCTCP)». In: *SIGCOMM Comput. Commun. Rev.* 40.4 (Aug. 2010), pp. 63–74. ISSN: 0146-4833. DOI: 10.1145/1851275.1851192. URL: https://doi.org/10.1145/1851275.1851192 (cit. on p. 5).

[8] Fiqih Rhamdani, Novian Anggis Suwastika, and Muhammad Arief Nugroho. «Equal-Cost Multipath Routing in Data Center Network Based on Software Defined Network». In: *2018 6th International Conference on Information and Communication Technology (ICoICT)*. 2018, pp. 222–226. DOI: 10.1109/ICoICT.2018.8528730 (cit. on p. 6).