UNIVERCITY OF MACEDONIA
BACHELOR STUDIES
DEPARTMENT OF APPLIED INFORMATICS

EXPERIMENTAL STUDY OF DATA CENTER NETWORK LOAD BALANCING MECHANISMS

Thesis

of

Konstantinidis Menelaos

Thessaloniki, July 2021

Περίληψη

Εξισορρόπηση φορτίου στα δίκτυα υπολογιστών είναι η τεχνική διαμοιρασμού του φορτίου ανάμεσα σε διαφορετικούς συνδέσμους του δικτύου ή ανάμεσα σε διαφορετικούς διακομιστές του. Στα κέντρα δεδομένων η εξισορρόπηση φορτίου της δικτυακής κίνησης είναι πολύ σημαντική για την ομαλή και σωστή λειτουργία τους.

Η παρούσα πτυχιακή εργασία στοχεύει στη σύγκριση ενός δυναμικού και ενός στατικού αλγορίθμου εξισορρόπησης φορτίου, προκειμένου να φανεί ποιος από τους δύο αποδίδει καλύτερα. Η βιβλιογραφική ανασκόπηση της εργασίας περιλαμβάνει τη θεωρητική μελέτη των κέντρων δεδομένων. Αυτή η μελέτη αναφέρεται στο δίκτυο, τις τοπολογίες και τις αρχιτεκτονικές των κέντρων δεδομένων. Περιλαμβάνεται επίσης, μελέτη για την εξισορρόπηση φορτίου, όπου αναλύονται οι διάφορες κατηγορίες των αλγορίθμων εξισορρόπησης φορτίου. Επίσης, περιγράφονται αρκετοί αλγόριθμοι δίνοντας βαρύτητα στη λογική και στην λειτουργία τους.

Στο πείραμα προσομοιώθηκε μια τοπολογία 3 επιπέδων fat tree χρησιμοποιώντας τον προσομοιωτή Mininet. Αυτό το δίκτυο συνδέθηκε με τον ελεγκτή Floodlight, στον οποίο υλοποιήθηκαν οι δύο αλγόριθμοι εξισορρόπησης φορτίου που χρησιμοποιήθηκαν στο πείραμα μαζί με την λειτουργία καταγραφής των αποτελεσμάτων. Το πείραμα περιλαμβάνει σενάρια στα οποία το σύστημα τέθηκε υπό φόρτο με τη βοήθεια της γεννήτριας πακέτων D-ITG, προκειμένου να παραχθούν αποτελέσματα από τις μετρικές που χρησιμοποιούνται. Με βάση αυτά τα αποτελέσματα αναλύθηκαν και αξιολογήθηκαν οι επιδόσεις των δύο αλγορίθμων.

Λέξεις κλειδιά: Εξισορρόπηση φορτίου, Κέντρα Δεδομένων, Δίκτυα Οριζόμενα μέσω Λογισμικού

Abstract

Load balancing on computer networks is a technique used to spread data traffic or task load across multiple network links or servers. In data centers the load balancing of the traffic is crucial for their efficient and smooth operation.

This thesis aims to compare two load balancing algorithms, a static and a dynamic in order to find out which performs better. The background of the thesis includes the theoretical study of data centers. Their network, topologies and most important architectures are explored. Also, load balancing is included, with a focus on its application on data centers and the classification of load balancing algorithms. Moreover, some load balancing algorithms are presented with a focus on their procedure and logic.

In the experiment a 3-level fat tree topology was simulated using the Mininet simulator. This network was connected to the Floodlight controller, in which the two load balancing algorithms used in the experiment were implemented along with the monitoring. The experiment included scenarios that stressed the system with the help of the D-ITG traffic generator in order to generate results from the monitored metrics. Based on these results, the efficiency of the two algorithms is evaluated and analyzed.

Keywords: Load Balancing, Data Centers, Software-Defined Networks

Table of Contents

Table of Charts

Table of Figures

Table of Tables

1 Introduction

In recent years the increase in volume of data traversing the internet is exponentially larger than the increase in computing power of modern systems [42]. Due to that, more and more applications are hosted on cloud infrastructure that also accommodates cloud services that are based on the client-server model. Data centers offer their hosted applications and services high availability, fault tolerance and plenty of computing and storage capacity.

Data centers also offer, high bandwidth to their hosted applications and services. Due to their size and bandwidth needs they face problems, like bottlenecks when traditional routing and forwarding protocols are applied. This is why, data centers are built using architectures, like SEATTLE [27], TRILL [28] and Portland [2] that try to compensate for the problems of traditional layer 2 or layer 3 fabrics [38]. The general construction and orientation of the hardware of a data center is referred to as its topology. There are a lot of types of topologies, with each one offering different advantages. The fat-tree topology is one of the most well-known topologies.

Having control over the data traffic in a data center network is imperative for the efficient and smooth operation of the data center [40]. Expensive systems and network equipment are not enough to ensure that the data traffic will always be controlled and balanced, due to its unstable and unpredictable nature. When the network is unable to handle the traffic, links and servers become congested. The result of this congestion is for hosted services and applications to become unavailable and unresponsive. Therefore, the load needs to be balanced across multiple links on the network or the applications or the services needs to be hosted in many servers.

Load balancing in computer networks is an important technique that tries to use existing resources to better distribute data packets, while avoiding equipment overload. Load balancing can be applied at the application layer, where its main duty is to select the appropriate server to respond to a particular request [20]. In order to do that, there are a lot of servers hosting the same application scattered across the network. Each request is sent to the server that can respond the fastest.

Load balancing can also be applied in the network as link load balancing. The strategy of link load balancing, is to spread the traffic across different links that lead to the same destination, in order to avoid network congestion. In practice, this method detects congested paths and tries to relocate flows to other less congested paths, in order to relief or even remove the bottleneck.

These load balancing techniques are enforced by load balancing algorithms that are installed in the network. Load balancing algorithms are divided into many categories. The most known categorization is static and dynamic algorithms.

Static algorithms are the more common and less complex of the two. These algorithms are unaware of the network's status, like congestion and are not very efficient when dealing with a data center's traffic due to its nature.

Software-Defined Networks (SDNs) [29] [30], give the ability to control the data traffic within the network, without needing the use of expensive equipment [21]. It enables the network to be configured dynamically and programmatically efficient. Dynamic load balancing algorithms use SDNs to monitor real time information about the network, like congestion. Using this information these algorithms create tailored routing strategies and apply them to the network.

1.1 Objectives and purpose

The purpose of this thesis is to understand how data center networks work and how load balancing algorithms can help balance the data traffic across their network. The objective of these load balancing algorithms is the prevention of network congestion and the more efficient usage of resources.

The analysis starts with a research on data centers concerning their network, topologies and architectures, like Portland. Load balancing is also presented, focusing on its applications on data centers. Then, classifications of load balancing algorithms are presented, like static and dynamic or centralized and distributed. Finally, the definitions of some load balancing algorithms are presented.

Lastly, a static and a dynamic load balancing algorithm are compared in a Software-Defined Network. During this procedure some metrics are monitored in order to assess each algorithm's performance. These resulting metrics are used for the comparison and evaluation of each algorithm's performance.

1.2 Contribution

This thesis has the following as contribution:

1) The theoretical framework of data center networks, topologies and architectures is analyzed. Categorization of the aforementioned is defined and some of the most prevalent architectures are described.

2) Load balancing algorithms are described along with the categories that they are divided into, depending on their characteristics. Also, many load balancing algorithms are presented, describing their definitions and advantages they offer.
3) The load balancing efficiency of two algorithms is assessed in the experiment procedure of this thesis.

1.3 Structure of the Thesis

The following chapter provides an overview of data centers. The network of data centers is described. Topologies and their categories, are also presented. Lastly some of the most important data center network architectures are presented, with an emphasis on their basic characteristics and advantages over the traditional forwarding architectures.

In chapter three load balancing in data centers is explained. Categorizations of load balancing mechanisms are featured and followed by a presentation of some load balancing algorithms.

Chapter four depicts the experiment's implementation. Firstly, the tools that were used in the experiment and their contribution in it are mentioned. Following is a description of the experiment's test environment setup, the implementation of the load balancing algorithms and of the monitoring procedure.

In chapter five the experiment's execution is described and results are presented and explained.

Finally, chapter six, presents conclusions, along with some considerations about future work and this thesis' limits and restrictions.

2 Data Centers

2.1 Definition

Data Centers are comprised by clusters of servers, storage and network devices such as switches, routers and modems, power distribution systems and cooling systems. These servers are interconnected to each other using switches in many variations called topologies. Data Centers are used for hosting web sites, for storage, to run large scale and data intensive tasks, for example indexing Web pages, for data analysis and for networking. They constitute an economically viable infrastructure that satisfies the above uses and they are run by large companies like Amazon, Google and Microsoft. Their spread is due to the radical trend towards cloud computing.

2.2 Data Center Network

A Data Center Network is the communication infrastructure used in the Data Center, and is described by the network topology, routing/switching equipment, and the used protocols (for example Ethernet and IP). The Data Center Network's performance can be evaluated by well-known metrics like bandwidth, latency and cost. Independent of the Data Center's topology, it should be decided if the network will be managed as a Layer 3 or Layer 2 addressing domain [1]. Each option has several advantages and trade-offs regarding to the Data Center's scalability, switch state, manageability and support for end host virtualization.

A Layer 3 fabric would need every switch in the Data Center to be configured with a subnet mask. Also, DHCP servers would need to be configured so that IP addresses that contain the correct subnet mask of the connected switch are given to each host that requests one. This creates a problem, since every host will be powering several Virtual Machines, all with assigned IP addresses. Since IP addresses have topological meaning, Virtual Machine migration to different host outside its current host cannot be done. If a Virtual Machine migrates, a different IP address will be assigned to it, resulting in existing TCP connections to be invalidated.

A Layer 2 fabric on the other hand can support Virtual Machine migration, since it identifies each recipient machine with its unique MAC address. A layer 2 fabric is almost like plug-and-play since switches use protocols like Rapid Spanning Tree, to learn a machine's position in the local area network. Communication between hosts is done by forwarding packets using MAC addresses. However, that requires switches to have stored MAC forwarding tables with size proportional to the size of the whole Data Center network.

The biggest problem though is that both layer 2 and 3 solutions require broadcasts and that can introduce a significant overhead to the network, because with every update a broadcast is sent to all the switches in the network. Solutions like SEATTLE, TRILL and Portland architectures try to medicate this problem with the most prominent being the Portland architecture.

2.3 Topologies

The Data Center architecture is optimized depending on the function of the Data Center. If the Data Center for example, has compute-intensive workload then it is equipped with powerful nodes. There is a plethora of Data Center architectures that can be divided into three categories depending on where the packet forwarding is carried out [3]. In the switch-only category, packet forwarding is enacted by switches. On the other hand, server-only has the servers responsible for running applications and forwarding packets between themselves. Lastly the hybrid category uses both servers and switches for packet forwarding.

- **Server-only topology**

    1. **CamCube**: In this type of topology [15], servers are end hosts and also are responsible for packet forwarding and routing. CamCube's main goal is to make the development of services in data centers easier. Servers need to have multi-core processors and high-performance network interface cards (NIC) with multiple ports. Every server has its ports connected directly to other servers. CamCube uses a direct-connect 3D torus topology, where each server connects to a small set of other servers without any switch or routers. An address is given to each server that represents the server's relative position against an arbitrary origin server in the 3D torus. This address cannot be changed as long as the server functions inside the network. Inside the CamCube network, one-hop neighbors communicate using Ethernet packets. Traffic from outside the network is delivered over an IP-based switched network. CamCube also provides a simple and very limited API that is used by applications to implement personalized routing protocols to improve the application desired characteristics such as trading higher latency for better path convergence. Illustrated below in Figure 1, is a 3D torus topology with 64 servers, where servers are represented by blue squares.

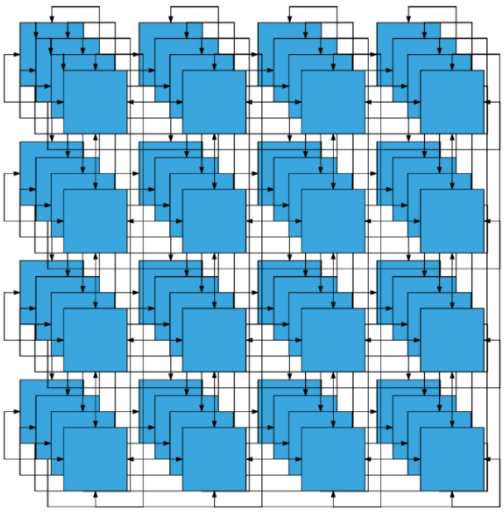

Figure 1. 3D torus with 64 servers [16]

- **Hybrid topology**

2. **BCube**: BCube [18] provides high performance and robustness and is aimed for the data intensive environment of a modern data center. BCube also, aims to be cost efficient by using low-cost switches. BCube uses both servers and switches for packet forwarding. Each server has a small number of ports, no more than four. Multiple layers of low-end switches are used to interconnect these servers, in a way that multiple parallel paths exist between any pair of servers. This improves fault tolerance and load balancing while providing higher bandwidth. BCube runs a source routing protocol called BSR (BCube Source Routing), that places intelligence solely on servers. Switches are only connected to servers and never directly to other switches, and can be treated as dummy crossbars that connect several neighboring servers. BSR takes advantage on the multi-path property of the topology and is constantly probing the network to dynamically balance the network and handle failures. It should be noted that BSR's performance is decreased gracefully as the switch or server failures increase. A $BCube_0$ topology is comprised by an n-port switch and n servers all connected to a switch port but not to each other. A $BCube_k$ topology is recursively defined from $BCube_0$. It is constructed by using n $BCube_{k-1}$topologies and n^k n-port switches. It has $k+1$ levels, $N = n^{k+1}$ servers with $k+1$ ports each, that are connected to a switch at each level. Below, Figure 2 is an example of a $BCube_1$ topology.

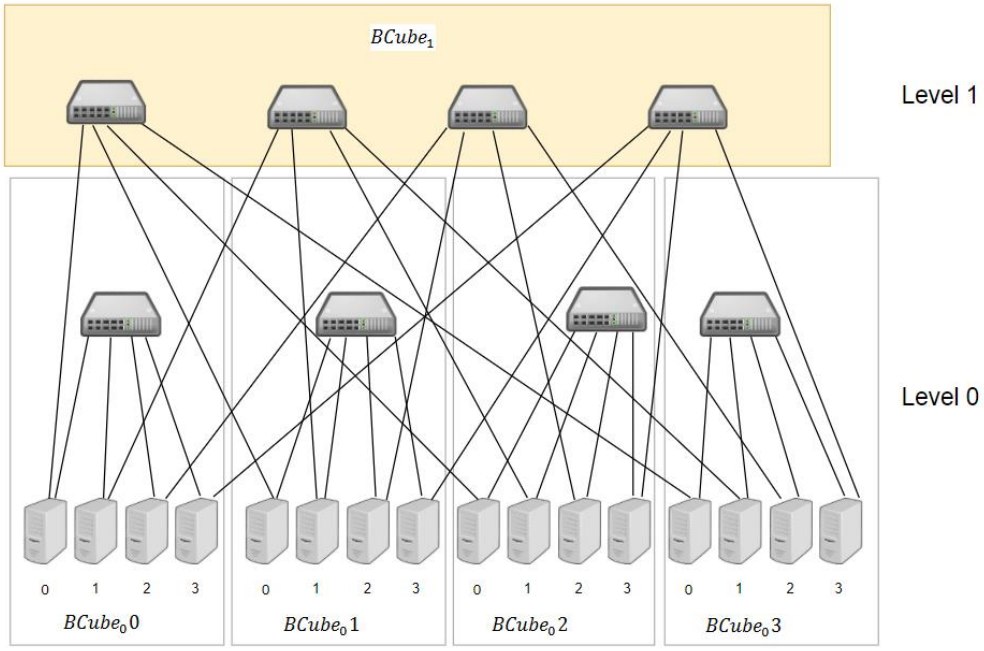

Figure 2. $BCube_1$

- **Switch-only topologies**

3. **Flattened Butterfly Network**: The flattened butterfly topology [19] is comprised by low-diameter networks that are symmetrically positioned. It is a cost-efficient topology for high radix networks that takes advantage of high radix routers and global adaptive routing. The flattened butterfly is more cost efficient than a Clos (for example Fat Tree) network that offers comparable performance on load balanced traffic. It achieves that by eliminating redundant hops, where they are not needed for load balance. In flattened butterfly servers are not connected directly to each other, but are connected through high radix routers. When a packet is transmitted by its source server, it requires a hop to travel to the local router. From there, zero or more inter-router hops and a final hop, from the final router to the destination, are required. Flattened butterfly is multidimensional and is described by two numbers. A k-ary n-flat flattened butterfly is a topology, where the n is the dimension of the topology and the k is the number of switches in each dimension. A k-ary n-flat flattened butterfly can be constructed from a k-ary (n-1) flattened butterfly and a k-ary 2-flat flattened butterfly. In Figure 3 an 8-ary 2-flat is displayed, comprised by 8 interconnected switches, where each switch is connected to 8 hosts-servers.

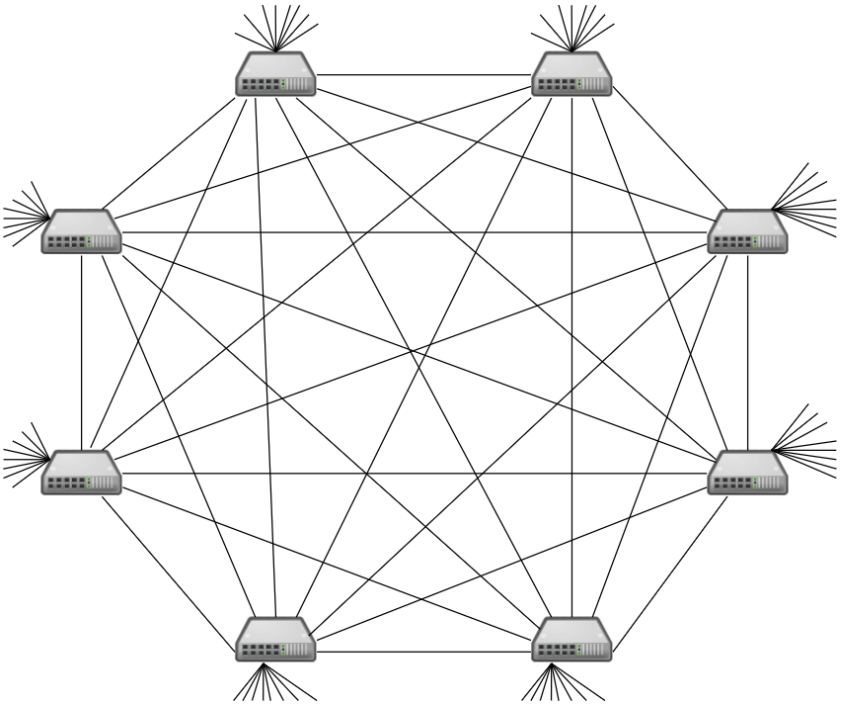

Figure 3. Flattened butterfly

4. **Multi-tiered Network**: Multi-tiered topologies are the most common in the industry. A three-tiered topology has three levels of switches, the core, the aggregation and the edge level. The edge level switches are connected to the hosts. A basic parameter of multi-tiered networks is the over-subscription ratio, which is the ration of bandwidth for downlinks to the bandwidth for uplinks.

5. **Fat Tree Network**: The Fat Tree topology [14] is an instance of the traditional multi-tiered topology that factors in the large price difference between high end switches, that have more ports and higher bandwidth capacity, and lower end switches. Fat Tree topology is a three-tiered topology that has its aggregation and edge level high end switches, replaced by low end ones that are interconnected. Each set of interconnected low-end switches is called a pod. The number of uplinks and downlinks are equal across all pods. The Fat Tree topology is highly scalable and economical. A k-ary Fat Tree topology has k number of pods. Each pod has $k/2$ aggregation and $k/2$ edge switches. Each switch has k number of ports. Edge layer switches use their $k/2$ ports to connect to hosts and the remaining $k/2$ ports to connect to aggregation layer switches of the same pod. In the core layer there are $k^2/4$ core switches and each of them use their k ports to connect to each pod.

In total there are $k^2/4$ core switches, k pods, $k^2/2$ aggregation switches, $k^2/2$ edge switches and $k^3/4$ hosts. Figure 4 displays a Fat Tree topology where $k = 4$.

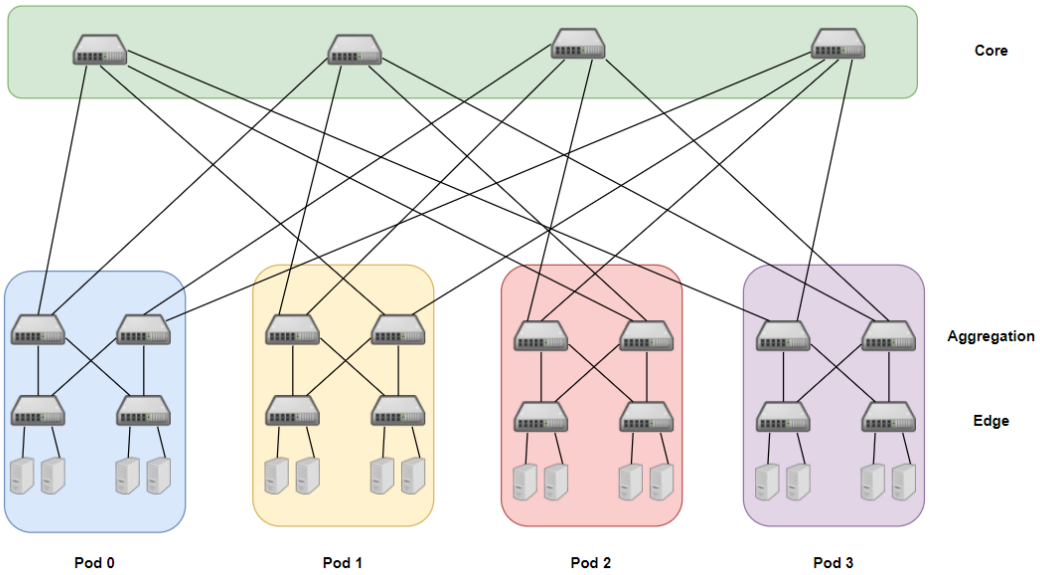

Figure 4. Example of a Fat Tree Topology where k=4

2.4 Data Center Network Architectures

2.4.1 Portland

Portland [2] is a set of layer 2 compatible routing, forwarding and address resolution protocols that tackles the problems, that using a layer 2 or layer 3 fabric would introduce, like lack of scalability, difficult management, inflexible communication and limited support for virtual machines. It is designed to be scalable, fault tolerant and “plug-and-play”.

It is mainly built to fulfill the below requirements.

- Any Virtual Machine can be migrated to any physical machine, while keeping its original IP address. If a virtual machine cannot keep its original IP address, existing TCP connections will break.
- A switch should not need to be configured by an administrator before it’s deployed.
- A server should be able to efficiently communicate with any other server in the data center, using any of the available paths.

- There should be no forwarding loops.
- Failure detections should be rapid and efficient. If a failure occurs, the unicast and multicast sessions should be unaffected.

Portland is applied on hierarchical network architectures like the Fat-Tree topology. It uses a logically centralized fabric manager that can be a redundantly connected host to the network or run on a separate control network. The fabric manager maintains soft state about the network configuration. It also, assists with ARP resolution, fault tolerance and multicast. Portland achieves its efficiency in forwarding, routing and virtual machine migration by introducing Pseudo MAC addresses (PMAC). A PMAC addresses is given to each end host in the network and indicates the position of the host in the topology. A PMAC address has 48 bits and is comprised by: 16 bits that indicate the pod that the host is connected to, 8 bits that reveal the position in the pod of the switch that the host is connected to, 8 bits that are translated as the port number of the switch and 16 bits for the virtual machine ID. Each host has an IP, MAC and PMAC address. Hosts are ignorant of PMAC addresses and use their original MAC addresses for requests. Switches have flow tables that match MAC and IP address to PMAC addresses. When a local switch sees a request from a new MAC address, it creates an entry for the host's IP and MAC address to its local table and then creates a suitable PMAC for the host. After that it sends the IP and PMAC address to the fabric manager so that he can update his PMAC mapping table. Lastly the switch updates the entry in its local table, that now contains the IP, MAC and PMAC addresses of the host. ARP requests are handled with the help of the fabric manager and are illustrated in Figure 5 and explained below.

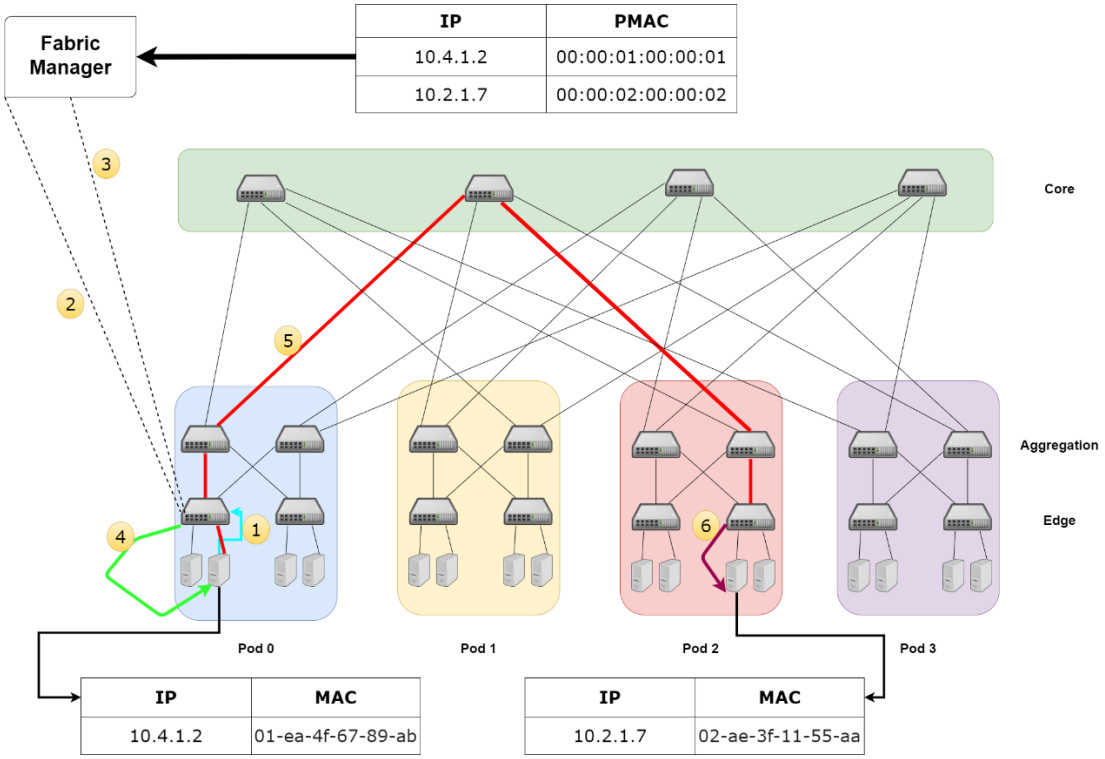

Figure 5. ARP request example in Portland

In this example a server with IP 10.4.1.2 wants to send a packet to the server with IP 10.2.1.7. In step 1, the edge switch intercepts an ARP request for an IP to MAC address mapping and forwards the request to the fabric manager in step 2. The fabric manager consults its PMAC table to see if an entry is available for the target IP address. If it finds an entry, it returns the PMAC in step 3 to the edge switch, which in turn in step 4 returns it to the original host. The host in step 5 forwards the packet, with headers the destination IP and PMAC address. Lastly in step 6 the edge switch using its local table replaces the destination's PMAC with its MAC address and forwards the packet to the destination server. It should be mentioned that if the fabric manager doesn't have the requested IP to PMAC mapping due to a failure, it will broadcast to all end hosts in order to retrieve the mapping and will proceed to send the PMAC to the edge switch. In Portland switches use their position in the global topology to perform more efficient forwarding and routing. Switch position could be set manually by an administrator due to the slow changes in switch positions in a topology but in order to keep the promise of "plug-and-play", Portland uses a Location Discovery Protocol (LDM) to automate this process. A LDM message contains the following information:

- Switch ID: a globally unique identifier for each switch
- Pod number: This number is set for switches that belong in a pod. Each pod has a unique number. Switches that belong in a pod have the same number.
- Position: this number is assigned to edge switches and is unique within each pod.

- Tree level: 0, 1 or 2 depending on if the switch is an edge, aggregation or core switch.
- Up/Down: this bit indicates if the switch port is facing upwards or downwards in the multi-rooted tree.

In Portland, switches periodically send LDM messages out from all their ports to other switches in order to learn their positions in the network and to monitor the state of the topology. During the LDM messaging phase packet forwarding is postponed. It should be noted that in order to ensure that switches inside a pod use unique numbers, the fabric manager is used to assign these numbers to switches inside each pod.

2.4.2 SEATTLE

Large scale IP networks are too difficult to manage and Ethernet, while simpler doesn't scale beyond small local area networks. SEATTLE [27] tries to combine the scalability of IP with the simplicity of Ethernet. It provides plug-and-play functionality, while being scalable and efficient through hash-based resolution of host information and shortest path routing. SEATTLE provides the below features:

- A one-hop, network-layer DHT: In SEATTLE end hosts use MAC addresses to forward packets. However, switches aren't required to maintain the state of each host or to determine host locations, using network-wide floods. Distributed hash tables (DHT) are used by a link-state routing protocol to store the location of each host in the system. This network layer DHT is used for address resolution (mapping IP to MAC addresses) and for more flexible service discovery.
- Traffic-driven location resolution caching: In enterprise networks, hosts will communicate with a small number of other hosts, which makes caching highly effective. In SEATTLE, switches cache responses to queries to avoid excessive load on the directory service and to forward packets using the shortest paths. Also, SEATTLE provides a way to include location information to ARP replies, making location resolution not needed when forwarding packets.
- A scalable, prompt cache-update protocol: SEATTLE supports host mobility. When a host is not reachable or other network-layer changes, switches check and update their caches while deleting invalid entries. This cache update protocol is based on unicast and ensures that packets are delivered on time and that overhead in the network is kept low.

SEATTLE switches learn host location information via a hashing mechanism. Figure 6 illustrates how a new end-host is handled, and how a packet is forwarded to this host.

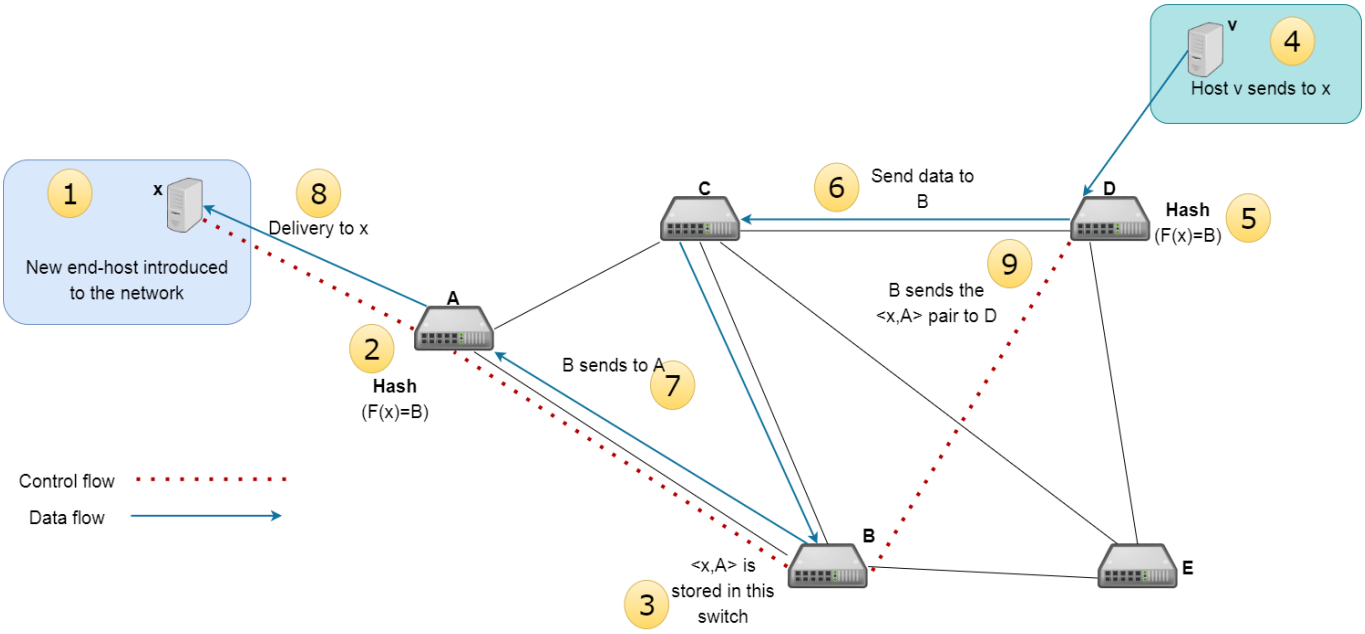

Figure 6. SEATTLE functionality

In step 1 the new host is introduced to the network. SEATTLE uses a key pair to store host location information in the form of (key, value). Here the pair is (x, A). The switch A, in step 2, uses a hash function on the key to map it to a switch identifier, in this example B, $F(x) = B$. Then in step 3 the pair is stored in switch B. This works because every switch uses the same hash function and also, knows every other switches' identifiers. In step 4 another host wants to send a packet to the newly introduced host. Step 5 has the switch D use the hash function on x to find the resulting identifier of switch B. In step 6 the data is sent to the switch B. In step 7, the packet has reached B and since B has the pair <x, A>, it knows that it needs to send the packet to switch A in order to reach the host x. In step 8 the packet is delivered. Finally, the switch B sends its stored pair to switch D, so that the next time host v wants to send a packet to host x, switch D will be able to directly send, using the stored pair, the packet to x rather than use the hash function. It should be noted that, SEATTLE can react to network failures due to Consistent Hashing. In order for SEATTLE to be compatible with existing applications and not require modified end hosts, it allows end host to generate ARP and DHCP requests and converts them internally to unicast queries against a directory service. Traditional broadcast-based ARP requests are not used in SEATTLE. As in the example above pairs of keys and values are used, but here the IP addresses are used as keys. In similar fashion a hash function is used on an IP address to result in a switch identifier, that points to the switch where the IP and MAC addresses of the host are stored plus the identifier of the switch that the host is connected to. This way when an ARP broadcast request is intercepted by a switch, the hash function is used on the destination IP address. Then through unicast the request is delivered to the switch identifier, which in turn through unicast send the requested IP, MAC addresses plus the identifier of the switch that the host is connected to. DHCP

request are handled in a similar way. The key differences are that as keys the MAC addresses are used and that that the switch that the hash function points to, has stored the MAC address of the DHCP server and the switch identifier of the switch that the server is connected to. In Figure 7, an example of a DHCP request is illustrated.

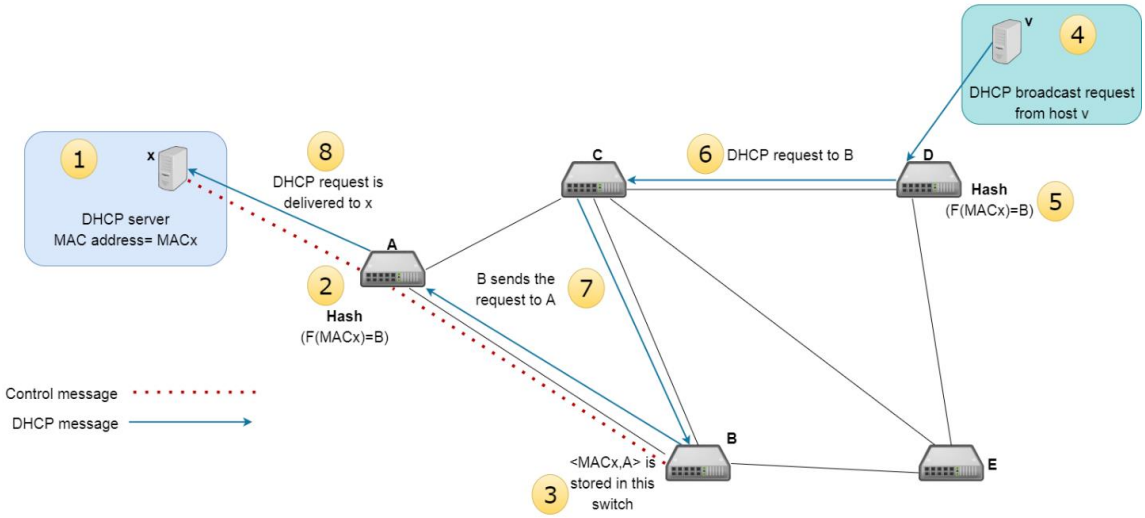

Figure 7. DHCP request example

2.4.3 VL2

VL2 [35] [36] is a practical network architecture that provides scalability, high-capacity paths between servers, performance isolation between services and uses Ethernet layer-2 semantics. VL2's vision is to overcome limitations and problems introduced by other topologies like oversubscription, traffic flood of a service affecting others that use the same links and IP fragmentation, resulted from dividing servers among VLANs. This vision results in three objectives that VL2 wants to meet:

- Uniform high capacity: The maximum rate of a server's output flow should be limited only by the available capacity of its Network Interface Card. Also, assigning servers to a service should independent of network topology.
- Performance isolation: Traffic of a service should not affect traffic from other services.
- Layer-2 semantics: Any server should be easily assigned to any service and also given whatever IP the service expects. Virtual Machines should be able to migrate and keep their original IP addresses. Lastly, features like link-local broadcast should be work, due to application compatibility.

VL2 consists of a network arranged on a Clos topology and is divided into edge, aggregation and core layer from bottom to top. VL2 uses 1GbE links to connect end hosts with the edge switches and 10GbE links for the interconnection of the upper layer switches. The edge switches have n_0 10GbE ports and $(10 * n_0)$ 1GbE ports, aggregation switches have n_1 10GbE ports and core switches have n_2 10GbE ports. Each edge switch uses its 1GbE port to connect to $(10 * n_0)$ hosts and its 10GbE port to connect to n_0 aggregation switches. Each aggregation switch uses half of its ports to connect to edge switches and the other half to connect to core switches. In total in a VL2 topology there are $\frac{n_1}{2}$ core switches, n_2 aggregation and edge switches and $\frac{n_1 n_2}{2n_2}$ hosts. VL2 uses Valiant Load Balancing (VLB) to cope with the high divergence and unpredictability of data center traffic. VLB ensures that inside the network packets will not interfere with other packets and spreads traffic across all available paths without any centralized coordination or traffic engineering. Also, since VL2 is based on IP routing and forwarding technologies that are already available in switches, ECMP is used too. VLB distributes traffic across a set of intermediate nodes and ECMP distributes across equal cost paths. It should be noted that, VL2 uses flows instead of packets as the basic unit of traffic spreading to avoid out of order delivery. VL2 uses two different IP-address families. The Location Address (LA), is a location-specific IP address. Every switch and interface in the network is assigned an LA that stays for life. These LA's are advertised to other switches by an IP-based link state routing protocol. This allows switches to obtain the complete switch-level topology and to forward packets encapsulated with LAs. Each application is given a block of Application-specific IP Addresses (AA) that remain unaltered even when the servers that host the application migrate. If a new server is assigned to host an application, an unused AA will be mapped to the server's LA. AA-to-LA mapping is handled by a fast and reliable directory that is not visible to applications. This mapping is created when servers are assigned to applications. For packet forwarding to be possible in a network that knows routes for LAs, while applications use AAs, VL2 agents are needed at each server to trap packets. These agents encapsulate these packets with the LA address of the Top of the Rack switch (ToR) of the destination AA. When the packet arrives to the LA, the switch decapsulates the packet and delivers it to the AA. Servers that host a service believe that they belong to the same IP subnet. That's why, when an application sends a packet for the first time, the ARP request for the destination AA, generated from the host, is intercepted by an VL2 agent and sent through unicast to the directory system. The directory system answers the ARP request with the LA of the ToR that corresponds to the destination AA. Also, VL2 agents cache the mappings from AA to LA similar to a host's ARP cache. It should be noted that, an application will not be able to send packets to another if the directory system refuses to provide the AA-to-LA mapping. This means that the directory system could offer access-control polices. Lastly DHCP requests are intercepted by DHCP agents placed in the ToR switch and are send through unicast to DHCP servers in order to eliminate broadcasts.

3 Load Balancing

3.1 Load Balancing in Data Centers

Load Balancing is the process of distributing tasks across multiple resources, in order to improve efficiency, reduce each task's response time and maximize the utilization of the resources. Load Balancing is a well-known topic that has been thoroughly researched and is applied on systems worldwide. The effectiveness of a load balancing method is highly dependent on the nature of the tasks, therefore, information about the nature of the tasks, can be used at the decision-making time, making optimization possible.

Today's data centers are usually built using easily scalable architectures like the Fat Tree. In topologies like this, there is a large number of interconnected servers with multiple paths connecting one server to another. Due to that, balancing the traffic evenly across multiple paths is important in order to avoid bottlenecks. These topologies and traffic characteristics make traditional load balancing algorithms not capable for data center environment. For example, traditional networks are using load balancing methods that cannot work well on data centers' bursty traffic loads. Also, conventional static load balancing algorithms, do not use information like, congestion detection or bottleneck bandwidth of a path, leading them to produce congestion instead of preventing it.

In recent years, researchers and engineers have introduced a large number of load balancing mechanisms for data centers. Each mechanism focuses on a different objective [17]. In short, the objectives of a load balancing mechanism will mainly be focused on throughput, latency, robustness, scalability and energy efficiency. In more detail:

- Throughput and latency concentrate on improving network utilization and reducing flow completion time.
- Robustness describes the reaction of the mechanisms to topology changes like link failures and switch crashes.
- Scalability entails that the mechanisms could be installed on large scale data center networks with reasonable cost.
- Energy efficiency mechanisms try to finish tasks, while using the least number of devices necessary, in order to reduce energy consumption.

In this thesis, we focus on mechanisms that want to improve throughput and reduce latency.

3.1.1 Link Load Balancing vs Server Load Balancing

Load balancing mechanisms can be categorized into link load balancing and server load balancing [17].

Server load balancing mechanisms balance traffic between servers. They further are classified into layer 4 and layer 7 server load balancing mechanisms. Layer 4 server load balancing mechanisms are unaware of application information. They use IP addresses and port information to determine where traffic is directed to. Layer 4 server load balancing has an important impact in Internet services like, web searches, and in e-commerce. These services usually run across multiple servers in data centers. Each server has a distinct IP address named direct IP (DIP). A service provides one or more virtual IP addresses to users. The load balancer's job is to redirect traffic from the VIP addresses across a corresponding DIP. Layer 7 server load balancing mechanisms are aware of application information. They direct requests depending on their contents.

Link load balancing mechanisms mainly balance traffic across different links to avoid congestion or under-utilization of links. In this thesis, we are focusing on link load balancing mechanisms. Link load balancing mechanisms are comprised by two main processes, collecting congestion information and selecting paths. Mechanisms need to determine if a flow path is congested. There are various methods used for collecting congestion information like, 1) TCP-based, where a TCP packet loss is treated as a sign of congestion, 2) sending rate, where the average sending rate of flows are calculated and determine congestion and 3) switch queue length where, the length of the switch queue represents path utilization and congestion. Each load balancing mechanism uses the information gained by the collection congestion information method to select paths for flows across different paths. The spread of the flows across the different paths depends on the goals of the algorithm used by the load balancing mechanism.

3.2 Static and Dynamic Load Balancing Algorithms

3.2.1 Static Load Balancing Algorithms

The load balancing mechanisms are divided [39] into two types, static and dynamic. Static algorithms use priori information about the system. The performance of each node is determined at the commencement of execution. The load is distributed without considering the current load of the system. Also, once a flow has its path determined, it cannot change.

A good example of a static load balancing algorithm is the **Round Robin Algorithm** [41]. This algorithm distributes flows across all possible paths in circular order, without considering priority or congestion information. Although this algorithm is simple and starvation free, it is easy to create flow collisions in heavy-load scenarios. Due to that, there is an altered version of this algorithm called **Weighted Round Robin**, that benefits systems that use different types of servers

and switches. The algorithm remains fairly the same, with the difference that each node has a weight, depending on its performance, and that weight gives the node priority over another.

3.2.2 Dynamic Load Balancing Algorithms

Dynamic algorithms monitor the system in real time. The information they collect concerns active connections, path congestion, utilization of resources and more. Using this information, they redistribute the work load accordingly to avoid collisions and congestion. Dynamic algorithms try to calculate each flow's path using the real time information that they gather. Examples of dynamic algorithms will be described below.

Least Congestion Algorithm collects congestion information for every possible path and chooses the path with the least congestion. The algorithm's effectiveness and performance depend heavily on the accuracy of the path congestion information.

Dynamic Weighted Round Robin is the same algorithm as the Weighted Round Robin, with the difference being that the weights are calculated in real time.

3.3 Centralized and Distributed Mechanisms

Mechanisms can be categorized into centralized and distributed depending on their methods of scheduling flows [17].

3.3.1 Centralized Mechanisms

Centralized mechanisms run software on a central controller to gather information like, congestion information, real-time traffic patterns and link utilization of the network. This information is used by the load balancing mechanisms in order to assign flows to paths in a way where congestion is avoided. The central controller is able to communicate with switches and gather this information though protocols like OpenFlow[31] and various measurement applications.

In short OpenFlow is a protocol that allows remote controllers to determine the path of packets through switches. These switches have their control level separated from the routing level and transferred to the remote controller. This allows switches from different vendors to and with different interfaces to be able to work together and also allows a more sophisticated traffic management to be possible. It uses Transport Layer Security (TLS) to ensure security. It also uses flow tables in switches that are similar to traditional Ethernet forwarding tables.

It should be stated that, this additional information that is needed from the controller introduces a communication overhead in the network.

3.3.2 Distributed Mechanisms

In contrast to centralized mechanisms distributed mechanisms don't need a central controller or protocols like OpenFlow. They process flows distributely and locally in end-hosts or switches. This way they can handle bursts of traffic in data centers. Although, the bottlenecks and overhead that a central controller introduces are avoided, distributed mechanisms are designed for symmetric topologies and cannot react to topology changes like switch crashes or link failures.

3.4 Per Flow and Per Packet Load Balancing Mechanisms

Load balancing mechanisms are also categorized into per flow and per packet mechanisms [17].

Per packet mechanisms focus on balancing the load across network devices (routers and switches) and not hosts. Per packet load balancing means that the network device sends packets, of a request destined for the same destination, alternatively over the links that lead to the destination. This guarantees equal load across all links. However, it is possible for the packets to arrive out of order at the destination, due to the fact that different delays occur across the different links. Also, since a lot of links are used in per packet load balancing, it is possible for links with poor transmission quality to be used which may lead to delay, packet loss and error packets. If these poor-quality links cause a big delay or packet loss or error packets, the affected packets will need to be retransmitted which will hurt the network's performance.

Per flow mechanisms classify packets into different flows based on a specified rule, for example based on source and destination address. Packets of the same flow use the same link. For example, in a network all packets sent to destination1 comprise a flow and they use a specific path, while packets sent to destination2, that comprise another flow will use another path and so on. Due to that, packets preserve their order. Yet this method can lead to unequally used links. If one host receives a lot of traffic, then the path used for that host will be congested, while other links are unused.

3.5 Load Balancing Algorithms

In this section some load balancing algorithms will be described briefly.

The Equal Cost Multiple Paths (ECMP) algorithm [1] [45] evenly load-balances traffic over multiple equal-cost paths to a destination, increasing the bandwidth by fully utilizing otherwise unused links. Equal-cost paths have the same cost to the destination. When forwarding a packet, the network device must decide which next-hop path to use. The device takes into account the packet header fields that identify a flow, like source and destination address. All possible next-hop paths of equal cost are referred to as an ECMP set, and are entries in the routing table. These entries are multiple next hop addresses for the same destination with equal cost. A packet with an active ECMP set for it, is forwarded by using hash functions on its header fields to determine the next-hop address that it will be sent to.

Hedera [5] is centralized and dynamic algorithm that aims to balance large flows in data centers and is designed to support any multi-rooted tree topology, like fat-tree. Hedera can be described, in short, as a control loop of three basic steps. First, Hedera finds large flows at the edge switches, then it estimates the natural demand of large flows and uses placement algorithms to calculate good paths for them. Finally, these paths are installed on the switches. Hedera wants to fully utilize the path diversity that multi-rooted tree topologies offer. Therefore, a packet's path upwards in the tree is non-deterministic and is calculated along the way to the core. However, the downwards path is deterministic from the core to the destination edge switch. To be able to predefine this downwards path, core switches are given the prefixes for the IP address ranges of the destination pods. In similar manner the aggregation switches are equipped with the prefixes of the downwards ports of the edge switches in their pod. These ports lead directly to end hosts. When a flow starts, it is forwarded based on a hash, like ECMP. However, the flow is monitored and when it grows beyond a threshold, Hedera dynamically calculates a path for it. This threshold is defined as the 10% of the capacity of the link. If the flow grows beyond that threshold, it is assumed to be a large or elephant flow. Hedera employs a central scheduler that assigns non conflicting paths to flows. Specifically, the scheduler tries to not place multiple flows on the same link, when the bandwidth demands of these flows combined will surpass the link's capacity. A demand estimator is responsible for calculating a flow's capacity demands. Its process is to perform repeated iterations of increasing the flow capacities from the sources while decreasing the exceeded capacity at the receivers until the flow capacities converge. In every iteration one or more flows will converge; however, the process ends when all flows have converged. This process has an estimated time complexity of $O(|F|)$.

FDALB [6] divides flows into two categories, short and long. Short flows are transmitted by switches using static mechanisms like ECMP. Long flows on the other hand are balanced dynamically by a central controller. A flow is marked as long by the end host, who marks the flow with tags. Then the controller uses global congestion information to appropriately schedule and decide a path for the large flow. FDALB's process is better summarized below:

1. Adaptive threshold adjustments: FDALB uses sent bytes to detect long flows. If the sent bytes of a flow exceed the threshold, then the flow is marked as a long one. The threshold is dynamically adjusted by the box plot method and is also updated after a long flow have finished transmitting.
2. Load balancing at the controller: The controller uses a greedy Round Robin algorithm to schedule flows. Also, it uses the Additive Increase Multiplicative Decrease (AIMD) mechanism, that exists in TCP, to assume that long flows will share equally the available bandwidth of a link. A long flow is scheduled to a path, where it obtains the maximum bandwidth according to the following formula, $\frac{remaining\ bandwidth\ of\ link}{number\ of\ long\ flows\ in\ link}$.

Freeway [7], like Hedera, uses the terms elephant and mice flows for large and small flows respectively. It accepts that elephant flows need large and stable bandwidth to achieve high throughput, while being under no deadline. Mice flows on the other hands are considered to be usually under deadlines and to be sensitive to latency. Freeway is a centralized and dynamic algorithm that tries to fulfill the above requirements of each flow. The algorithm scans the network and marks the low latency paths as Low-latency Oriented Links (LOL) and the high bandwidth paths as High-throughput Oriented Links (HOL). The controller is responsible for monitoring the links' utilization. If the link utilization of half of LOLs passes a threshold, a HOL is transformed to a LOL and vice versa. The controller maintains counters for each link, that indicate if the link is used in a LOL or a HOL. If the counter of a link is set to 0 then the link is used in a HOL path, but if the counter is bigger than 0, then that link is used in LOL path or paths depending on the current number of the counter. Elephant flows are handled by the controller and are transmitted through HOL paths, while mice flows are transmitted directly to LOL paths using static mechanisms like ECMP.

The Most-Efficient-Server-First or MESF [8] algorithm is focused on being energy efficient while meeting the deadline constrains of its tasks. It is a centralized and dynamic algorithm that uses a central scheduler to process where to distribute tasks, according to real time scores of servers. A server has its score calculated by the computing capabilities of the server and the type of the data center it is housed in. There are two types of data centers regarding this algorithm.

- Data centers with heterogeneous servers. In order to ensure energy efficiency, the central scheduler prefers to assign tasks to the more powerful servers. The computing power of a server is calculated by the number of the maximum tasks, that the server can process in parallel. The concept here is that a more powerful server will use less energy to process a task than a less powerful one. The central scheduler assigns tasks by first assigning tasks to the most energy-efficient server, then the second and so on, until it runs out of servers or tasks.
- Data centers with identical servers. Here MESF introduces a saturation point for servers, which if exceeded the servers' performance is degraded significantly. The central scheduler assigns tasks depending on the current saturation points of the servers. The scheduler keeps a sorted list with current efficiency of each server, with the most efficient being on top, and assigns tasks to servers from top to bottom. When a server hits its saturation point, its entry in the sorted list is removed.

FlowBender [9] is a distributed and dynamic algorithm that is also congestion aware. Static schemes like ECMP do not reassign paths to flows, and that can lead to performance decrease when a path, used by a flow, becomes congested. FlowBender tries to dynamically reroute flows when their paths are congested. The algorithm has two main processes: congestion detection and rerouting congested flows. In order to avoid requiring modified switches, FlowBender's functions are carried out by the servers. A fraction of marked packets is monitored by FlowBender. Based on these packets a flow is deemed congested or not. If a flow is congested then the flow's packet headers are modified in order to trigger the switches to rehash them, in essence rerouting the flow's path according to the new hash value.

CLOVE [10] is a distributed dynamic algorithm that uses the virtual switch in the hypervisor of each host to control packet routing. This way there is no need for hardware and end host stack modifications. CLOVE uses standard ECMP in the physical network and influences the packet routing by changing the packet headers at the software switches. CLOVES process can be divided into three components.

- Path discovery. Software switches send probs with different port addresses to give information to the hypervisor. This way the hypervisor learns to choose the correct port in order to send a packet to a destination.
- Flowlets. Flowlets are used in CLOVE to avoid packet reordering. Flowlets are bursts of the same flow separated by a given timeout value. The flowlets are detected by the virtual switches
- Congestion-aware routing. CLOVE handles flowlets in a weighted round robin fashion. CLOVE uses the Explicit Congestion Notification (ECN) to detect congestion and calculate the weight of a path.

RepFlow [11] is a simple mechanism aimed for helping short flows to be faster while not requiring any modifications to switches or hosts. The main concept of RepFlow is that short flows in data centers can suffer from long flows congesting their paths, while there are other links that are unused in the network. RepFlow's solution to that is to duplicate short flows and transmit the duplicates to different paths of equal cost that lead to the same destination. Flows are handled by ECMP, and so duplicated flows need to have different hash values than the originals. RepFlow's main process can be summarized into three procedures.

- Define short flow. RepFlow considers a flow's size to be known before its transmission. A flow is marked as short when its size is smaller than 100KB.
- Decide when to replicate a short flow. RepFlow replicates a short flow when it starts transmitting, instead of when it is blocked by a long flow. This is done because the time cost of discovering that a short flow is blocked is greater than the overhead that the duplicate short flow will introduce to the network, which is considered small and acceptable.
- Decide the number of replications. Since each duplicate will use a different path, a suggestion would be to have multiple duplicates of each short flow. The number of duplicates though needs to be small due to the overhead and complexity that they

will introduce in the network. RepFlow for simplicity replicates its short flows only once.

DRILL [12] is a distributed algorithm that uses local switch information. DRILL doesn't collect global congestion information. It implements a random packet allocation scheme using local switch information. To forward a packet, the switch randomly picks two available ports and compare their queue length against a recorded port. The recorded port is the port that transmitted the last packet. The current packet is sent to the port with the smallest queue between the recorded port and the two random ports. DRILL requires no hardware modification while introducing no overhead to the network.

TinyFlow [13] is a distributed algorithm that tries to address long flow collision and short flow latency. TinyFlow renames large and small flows to elephant and mice flows respectively. TinyFlow splits elephant flows into 10KB mice flows and then uses ECMP to forward them. Mice flows are kept intact. TinyFlow's process and be summarized into two main procedures.

- Elephant flow detection. TinyFlow uses packet sampling to detect elephant flows. Edge switches sample the ports that are connected to hosts for packets that are of the same flow. If two packets of the same flow are found within 500μs then the flow is marked as an elephant flow.
- Dynamic random routing. This process is based on OpenFlow and transforms an elephant flow into mice ones while working well with ECMP. When an elephant flow is detected, the edge switch counts its bytes. When the bytes exceed a threshold, the switch will forward the next bytes of the flow through a different port that results in an equal cost path. Also, the byte counter is reset.

4 Evaluation Setup and Methodology

4.1 Tools Overview

4.1.1 Mininet

Mininet [22] is a network emulator, or more precisely a network emulator orchestration system. It runs a collection of end-hosts, switches, routers and links on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code. A Mininet host behaves just like a real machine. The host can be accessed via ssh and it can run arbitrary programs (including anything that is installed on the underlying Linux system.) The programs that run on the Mininet host can send packets through what seems like a real Ethernet interface, with a given link speed and delay. Packets get processed by a simulated Ethernet switch, router, or middlebox, with a given amount of queueing. When two programs, like an iperf client and server, communicate through Mininet, the measured performance should match that of two (slower) native machines. In short, Mininet's virtual hosts, switches, links, and controllers are the real thing, they are just created using software rather than hardware, and for the most part their behavior is similar to discrete hardware elements. It is possible to create a Mininet network that resembles a hardware network, or a hardware network that resembles a Mininet network, and to run the same binary code and applications on either platform.

Mininet [34] has a lot of advantages, some of which will be presented below. The first and most important advantage of Mininet is that it can be easily setup to a Linux system and run. Mininet is not very demanding on system requirements, so that it can be run on laptops, servers, on virtual machines etc. Moreover, Mininet lets the user create and run custom topologies and supports OpenFlow, so that its switches can be programmed through OpenFlow, in order to customize packet forwarding in the network. Mininet’s hosts can run any program that the Linux host has, which makes easier the procedure of creating a network and running for example traffic generation programs in some or all Mininet hosts. Lastly, Mininet is an open-source project, which means that its code is accessible and editable, so that the user can customize it to his needs. Also, there is documentation and a big community to help answer some questions if needed.

As mentioned before, Mininet allows for custom topologies to be created and offers a variety of parameters to be configured. This is done with the help of its Python API. Custom topologies can be written is short Python scripts that import Mininet’s libraries and then these

files can be run to initiate the Mininet network. The most important classes of Mininet's Python API will be mentioned below:

- Topo: the base class for Mininet topologies
- build(): the method that creates the topology
- addSwitch(): adds a switch to the topology and returns the switch name
- addHost(): adds a host to the topology and returns the host name
- addLink(): adds a bidirectional link to the topology
- Mininet: main class to create and manage the network
- start(): starts the network
- pingAll(): tests connectivity between all the nodes by making them ping each other
- stop(): stops the network

4.1.2 Floodlight

Floodlight [23] is a community-developed, open-source Java-based OpenFlow controller, supporting OpenFlow protocols 1.0 through 1.5. It is intended to run with standard JDK tools and ant and can be optionally run in Eclipse. Floodlight is multithreaded, which means that it has very good performance. Also, it can be easily customized through its representational state transfer application program interfaces (REST API) or by editing its open-source code. The Floodlight Controller can be advantageous for developers because it offers them the ability to easily adapt software and develop applications. Also, the Floodlight website offers coding examples and documentation that aid developers in building the product.

Floodlight is not just an OpenFlow controller [44]. Floodlight is an OpenFlow controller (the "Floodlight Controller") and a collection of applications built on top of the Floodlight Controller. The Floodlight Controller realizes a set of common functionalities to control and inquire an OpenFlow network, while applications on top of it realize different features to solve different user needs over the network. The Floodlight Controller is a Java project with the applications built as Java modules that are compiled with Floodlight, and exposed from the Floodlight REST API. When run, the controller and the set of Java module applications start running. The REST APIs exposed by all running modules are available via the specified REST port (8080 by default). Any REST applications, written in any language, can now retrieve information and invoke services by sending http REST commands to the controller REST port. Also, Floodlight features a Web Graphical User Interface that displays information about which applications are loaded, general information about the system, OpenFlow [32] information like switches' flow tables, a topology illustrator and more.

Tested with both physical and virtual OpenFlow-compatible switches, the Floodlight Controller can work in a variety of environments and can coincide with what businesses already have at their disposal. It can also support networks where groups of OpenFlow-compatible switches are connected through traditional, non-OpenFlow switches.

The Floodlight Controller is compatible with OpenStack, a set of software tools that help build and manage cloud computing platforms for both public and private clouds. Floodlight can be run as the network backend for OpenStack using a Neutron plugin that exposes a networking-as-a-service model with a REST API that Floodlight offers.

4.1.3 D-ITG

Distributed Internet Traffic Generator (D-ITG) [24] is a platform capable to produce traffic that accurately adheres to patterns defined by the inter departure time between packets and the packet size stochastic processes. It offers a rich variety of probability distributions like constant, uniform, exponential and Poisson. Also, it supports protocols like, TCP, UDP, ICMP, Telnet and VoIP. D-ITG offers a lot of parameters to be configured from the user by simple parameters on the command line. Its installation and use are easy and fast. D-ITG [25] allows to store information both on the receiver side and the sender side. It is thus possible to retrieve information on the traffic pattern generated. Additionally, D-ITG enables the sender and the receiver to send the logging operation to a remote log server. This option is useful when the sender or the receiver have limited storage capacity. Another innovative feature is that the sender can be remotely controlled by using ITGApi. This means that the D-ITG sender can be launched in daemon mode and wait for commands that instruct it to generate traffic flows.

D-ITG's most important components are featured below:

- ITGSend: is the sender component of the D-ITG traffic generation platform. It operates in three different modes. The first mode is called single flow and it is for generating only one flow using only one thread. The second mode is called multiple flows and it generates a set of flows using multiple threads. The third and final mode is called daemon mode and it used through the ITGApi.
- ITGRecv: always work as a concurrent daemon listening on the default or set by parameter port for incoming connections from ITGSend. When a connection request arrives, a new thread is generated, which is responsible for the management of the communication with the sender.
- ITGLog: is a log server, running on a different host than ITGSend and ITGRecv, which receives and stores log information from multiple senders and receivers. The log

information can be sent using either a reliable channel like TCP or an unreliable channel like UDP.

4.2 Evaluation Setup

All the aforementioned tools were installed on a laptop running Ubuntu 21.04 according to the instructions found in their respective official manuals. The tools' versions and the specifications of the computer system that the experiment was run on are presented on the tables below (Table 4.1, 4.2).

Operating System	Ubuntu 20.10	64-bit
CPU	Intel i5-5200U (2 cores)	2.20GHz
RAM	8 GB	1600MHz
Storage	Samsung 840 Evo	240GB

Table 4.1 - Computer Specifications

Name	**Version**
Mininet	2.3.0
Floodlight	1.2 (latest)
D-ITG	2.8.1-r1023-4
OpenFlow	1.3
Python	2.7

Table 4.2 - Tools Versions

4.3 Evaluation methodology

4.3.1 Overview

The experiment consists of a simulated SDN network, by Mininet with a fat-tree topology. This network is connected to a remote controller, Floodlight. In Floodlight the load balancing algorithms are installed along with the monitoring thread that evaluates their performance. The goal of this experiment is to evaluate the performance of a static and a dynamic link load balancing algorithm on the same network, same volume of traffic and the same communication patterns.

A custom fat tree topology is generated by a custom Python script. This script generates a fat tree topology with OpenVswitches [33] that run OpenFlow 1.3. From the script, the D-ITGRecv component is initialized to run on the background of every host, so that all hosts are ready to receive flows. This network is connected to the Floodlight controller, which sends LLDP packets that are responsible for topology discovery, when it is connected with a network. From Floodlight an alternation of ECMP and Hedera are installing flow entries so that each host can communicate with all the other hosts. When the controller discovers the complete topology through the required rounds of sending LLDP packets, all the paths that connect the hosts, are installed through flow entries in the switches. The experiment begins by sending ITGSend commands to all hosts. Throughout the duration of the experiment, the monitoring thread that runs in Floodlight, writes in a CSV file the results of the metrics that will be used to evaluate ECMP's and Hedera's performance.

The experiment's procedure will be described in more detail in the following sub-chapters in a per-component basis.

4.3.2 Topology creation with Mininet

Firstly, the procedure of creating the topology and network through Mininet will be explained. For this experiment the fat-tree topology will be used, as it is one of the most known and used topologies. It will be a 3-level fat-tree topology with $k = 4$, meaning that switches will have 4 ports each. This topology is comprised by 4 core switches and 4 pods that contain 2 aggregate and 2 edge switches each. Each edge switch is connected to 10 hosts, so that in total there are 80 end-hosts in the network. Each link has a capacity of 10Mbits per second, with the exception of links that connect the end-hosts with their respective edge switch. These links have a capacity of 1Mbit per second each. In the figure below the topology is illustrated by Floodlight by the "topology" section of its web GUI. The figure displays the core switches, that are clearly highlighted in green in the center. There 4 pods, each containing 2 aggregate switches (highlighted in red) that are connected with the core switches and 2 edge switches that connect to the aggregate switches. Each edge switch is connected with 10 hosts. Due to the nature of the script, that generates the topology the numbering of the switches will always be the same, for example switch 1 (s1), switch 2 (s2), switch 3 (s3) and switch 4 (s4) will always be the core switches.

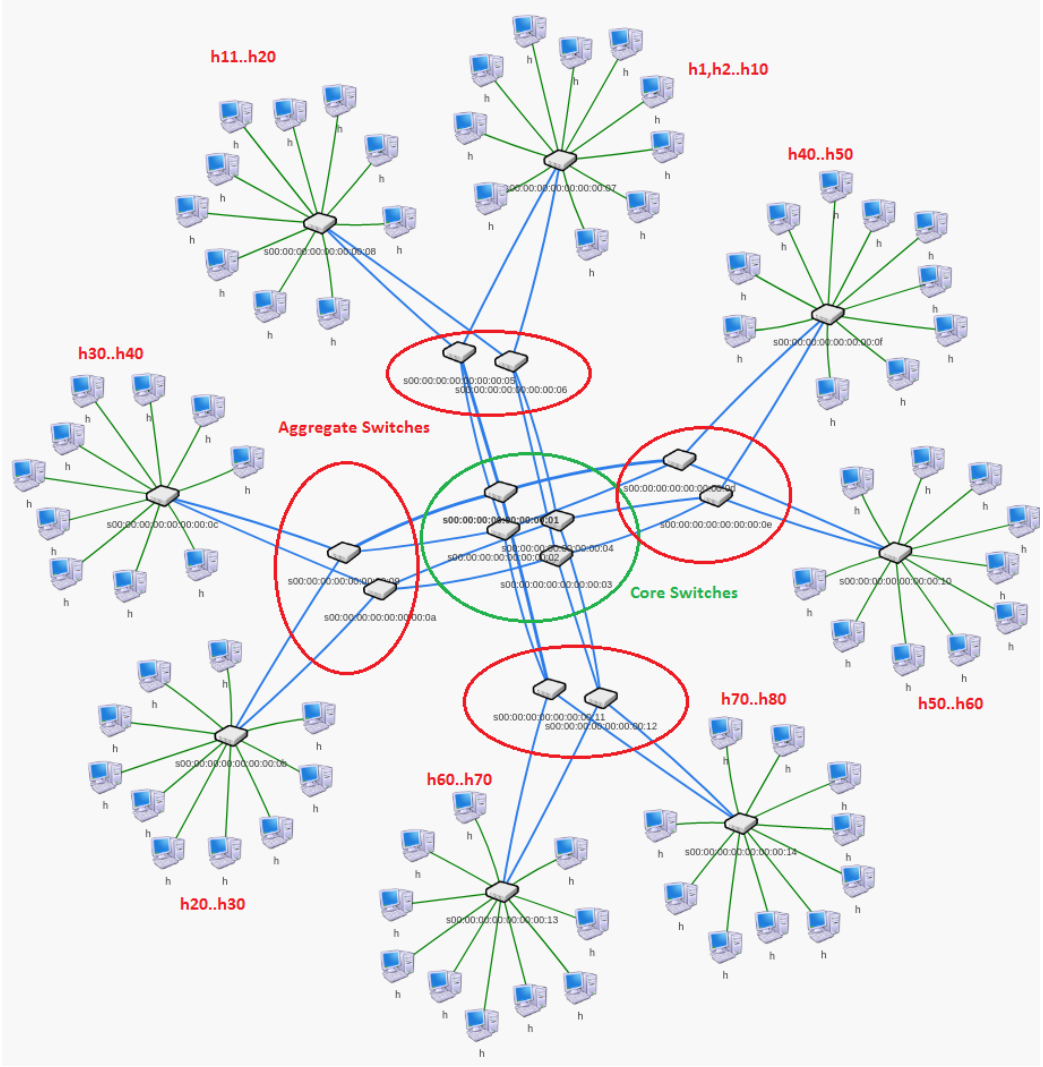

Figure 8. Mininet Topology

The Python script used to generate this topology has the following structure. Firstly, the necessary Mininet libraries are imported. Following, is a class named “MyTopo”, where the topology is defined. In this class, switches, end-hosts and links are created. It should be noted that the version of OpenFlow [32] that the switches use and the capacity of the links are specified in the commands that create them as parameters. A method follows, where a command that runs ITGRecv in the background is called for every host in the Mininet network. Lastly, the “main” of this script is defined. There, the Mininet network is initialized with its parameters. These parameters follow below:

- topo: This parameter accepts the desired topology for the network that will be created. Here the class “MyTopo”, which creates the fat-tree is inserted
- switch: This parameter accepts the type of switches the topology will use. Here OVSKernelSwitch (switches with OpenFlow) is used.

- controller: This parameter determines the controller of the topology, here the controller is remote and points to the Floodlight controller running in the system.
- autoSetMacs: this is set to “true”, because when set to true Mininet set easily readable Mac addresses to its nodes, that makes them easier to manage.
- autoStaticArp: this is set to “true”, so that each host will have installed in its ARP table, entries for all the hosts in the network. This is needed so ARP request handling will not be necessary in the experiment.

After the initialization of the Mininet network, the method that runs the D-ITGRecv command to all hosts using the aforementioned function and the CLI (command line interface) of Mininet are started. A screenshot of the Python script follows, with comments that make the understanding of the script easier.

```
#!/usr/bin/env python
#import of Mininet's libraries
from mininet.cli import CLI
from mininet.log import lg, info
from mininet.net import Mininet
from mininet.node import RemoteController, OVSKernelSwitch
from mininet.topo import Topo
from mininet.link import TCLink

#Topology class
class MyTopo( Topo ):
    def build( self ):
        #initialize variables for k, hnum= number of host, snum = number of switch,
        #core is an array where core switches will be stored
        hnum = 0
        snum = 0
        core = []
        k = 4

        #create core switches
        for i in range(k):
            snum+=1
            core.append( self.addSwitch('s'+str(snum),protocols=["OpenFlow13"]))

        #create pods k pods
        for i in range(k):
            aggr = []
            edge = []
            # create aggregate switches of pod
            for i in range(k/2):
                snum+=1
                aggr.append( self.addSwitch('s'+str(snum),protocols=["OpenFlow13"]))

            #create edge switches of pod
            for i in range(k/2):
                snum+=1
                edge.append( self.addSwitch('s'+str(snum),protocols=["OpenFlow13"]))

            #create links for pod
            b=0
            for i in range(len(aggr)):
                self.addLink( aggr[i], core[i+b] ,cls=TCLink,bw=10)
                self.addLink( aggr[i], core[i+1+b],cls=TCLink,bw=10 )
                b=1

            for i in range(len(edge)):
                self.addLink( edge[i], aggr[0],cls=TCLink,bw=10 )
                self.addLink( edge[i], aggr[1],cls=TCLink,bw=10)

                #create and connect hosts
                for j in range(10):
                    hnum+=1
                    host = self.addHost('h'+str(hnum))
                    self.addLink( edge[i], host, cls=TCLink, bw=1 )

#method that runs ITGRecv in the background of every host in the network
def ditgrecv( net ):
    "Run ifconfig on all hosts in net."
    hosts = net.hosts
    for host in hosts:
        info( host.cmd( '/usr/bin/ITGRecv > /dev/null 2>&1 &' ) )

#main method where the topology is initialized, the method that runs ITGRecv in all hosts is run
#and the mininet command line interface is started
if __name__ == '__main__':
    lg.setLogLevel( 'info' )
    info( "*** Initializing Mininet and kernel modules\n" )
    info( "*** Creating network\n" )
    network = Mininet(topo=MyTopo(),switch=OVSKernelSwitch,controller=RemoteController('cfloodlight','127.0.0.1',6653),autoSetMacs=True,autoStaticArp=True)
    info("*** starting ITGRecv on all hosts\n")
    ditgrecv(network)
    info( "*** Starting network\n" )
    network.start()
    info( "*** Starting CLI\n" )
    CLI( network )
    info( "*** Stopping network\n" )
    network.stop()
```

Figure 9. Python script that creates the Mininet Topology

4.3.3 Floodlight

Floodlight is the controller used in the experiment. In Floodlight alternations of ECMP and Hedera algorithms are created and run. Floodlight is a controller, where a lot of applications are included and run alongside the controller. A crucial application is called forwarding. OpenFlow 1.3 [32] has a flow entry in each switch that sends packets that cannot be forwarded by existing flow entries in the switch (table-miss) to the controller. When a table miss occurs the packets are processed by the forwarding application which processes the packets and sends tailored flow entries to the needed switches so that the packets can travel to their destination. For this experiment to work correctly the forwarding application needs to be disabled so that packets are only forwarded by ECMP and Hedera. This is done by removing forwarding's entries from the Floodlight's default properties file and from the file that loads all the applications that is located in the "METANF" folder of Floodlight.

Floodlight, also has a statistics class. This class is chosen to accommodate ECMP and Hedera along with the monitoring thread. The statistics class is comprised by threads that collect OpenFlow statistics. Two threads are crucial here. The first one is a thread called "portStats", that as its name suggests is responsible for collecting the port statistics that OpenFlow supports. Due to the fact that OpenFlow doesn't support bandwidth statistics, some modifications are needed to be done to this thread. The bandwidth is calculated by using the difference of the bytes counted between the current and previous run of the thread and divided by the time elapsed to produce the bandwidth. After that modification, the "portStats" thread is capable of delivering Port Bandwidth transmitting (TX) and receiving (RX). The other thread is called "FLowStatCollector" and its job is to collect the flow entries from all the switches along with some metrics, like byte count, packet count, hard time out, idle time out. In this thread a similar modification is done where the byte count of the previous and the current result of the thread run are used combined with the elapsed time in order to calculate the RX and TX bandwidth of the flow. These two threads store their results in Java's HashMaps. In order to ensure thread safety Reentrant locks are applied between the "portStats", "FlowStatCollector" and Hedera threads where they are needed.

In the following sub-chapters, the implementation of ECMP, Hedera and monitoring is explained.

4.3.4 ECMP

ECMP's procedure is that it searches for paths between two hosts. It selects the path with the least number of hops between them and then installs that path as flow entries in the

switches. For the implementation of the ECMP, the routing service that Floodlight provides is used. Floodlight's routing service is responsible for the following. Firstly, it generates a set number of paths between all switches of the topology that it detects. This number is set in the Floodlight's default properties file. Under the hood these paths are calculated with the help of Dijkstra's algorithm [26] and saved in a list sorted in order from the fastest to the slowest. Also, Floodlight's device Manager Service is used. This service, as its name suggests is responsible for collecting and storing topology device information. ECMP wants to loop through all end-hosts and install a path between all the possible pairs of hosts. The problem faced here is that the device manager does not discover the full topology immediately. Floodlight uses Link Discovery packets called LLDP, to discover switches and end-hosts of a topology. Fully discovering a topology may need many rounds of LLDPs. That's why the implementation of ECMP here has a check in its beginning where it compares the list containing all the devices, that it got from the device manager, from the current and its previous run. ECMP only continues running if these lists are not equal. After this check ECMP, loops through all hosts and using the fastest path found by the routing service it installs the necessary flow entries using Floodlight's Static Entry Pusher. Static Entry Pusher is the class that is responsible for installing flow entries from the controller. It requires an OpenFlow match [43] (information that is used to match an incoming flow with a flow entry in a switch), the switch that the flow will be sent to and the action that will be taken if the flow matches to an incoming packet. In this experiment the match properties that will be used to identify a flow will be the following:

- Ethernet type: here it will be IPv4
- IP protocol: in this experiment UDP packets will be sent but D-ITG requires to send some TCP packets for flow initialization.
- IPv4_SRC: IP address of the source host
- IPv4_DST: IP address of the destination host
- IN_PORT: the port that the packet entered the switch
- UDP_DST: the destination port of the UDP packet (for UDP packets)

ECMP installs flow entries for ICMP, TCP and UDP packets. ICMP are used to ensure that full host connectivity is achieved and therefore the experiment can begin. TCP are used from D-ITG traffic generator to initialize a connection between sender and receiver and start the packet sending procedure. UDP is used for the actual packets that consist the traffic of the experiment. When a packet matches a flow entry, the switch acts depending on the action of the flow entry. ECMP, instructs the switch through the flow entry action to output the packets from the correct port, so that they can follow their path to their destination. Lastly a flow entry is comprised by some other parameters some of which are used by ECMP. These are:

- Priority: priority determines the order in which the switch selects which of the matching flow entries of a packet to select (the flow entry with the highest priority is selected).
- Idle Timeout: idle timeout is the time after which an inactive (inactivity in OpenFlow is the time passed without any packets matching the flow entry) flow entry is deleted.
- Hard Timeout: hard timeout is the number of seconds after which the flow entry is deleted. This time is counted after the flow entry's creation.

ECMP sets these parameters in its flow entries as follows. Priority is set equal to 5. The number here doesn't matter as long as it is higher than 0, which value of priority is given to the flow entry that sends packets to the controller for processing when a table miss occurs. Although, it should be noted, that if two or more flow entries with the same priority match a packet, then the flow entry that is higher on the flow table would be selected. The flow entry that sends packets to the controller is always on the bottom of the flow table so if they are matching flow entries for a packet it would not be chosen, but for good practice reasons the priority for ECMP flow entries needs to be higher than zero. Idle and hard timeout are set to infinity because ECMP's flow entries do not need to be removed, they need to be static to ensure host connectivity.

4.3.5 Hedera

Hedera like ECMP, is a thread that runs at intervals. It runs on top of a network that is routed by ECMP. Hedera applies its routing strategies only when some criteria are met. The most important of which is when a flow consumes more that 10% of the links capacity in the link connecting the end-host with the edge switch. This implementation of Hedera first checks only for flows that are routed by ECMP. After a flow is found that it meets the criteria a routing strategy is applied. This implementation of Hedera uses the Worst-Fit allocation strategy, where each flow is routed to the path with the most available bandwidth. Hedera uses Floodlight's routing service to get the paths between the two hosts and applies the Worst-Fit algorithm. The path with the most available bandwidth is selected and like ECMP, Floodlight's Static Entry Pusher is used to install the needed flow entries to the switches.

Hedera uses the same matching parameters as ECMP. Although, some of the OpenFlow parameters are different. The priority of a Hedera's flow entry is higher than ECMP's. Also, Hedera's flow entries have idle timeout of 10 seconds, so that if they are unused, they need to be deleted, because Hedera is only used on flows that consume more than 10% of the links capacity. Also, it should be noted that like ECMP, Hedera install flow entries for ICMP, TCP and UDP packets.

Lastly, Hedera needs information about the flow's Bandwidth to calculate if the flow consumes more than 10% of the link's capacity. Also, it needs to access port Statistics to calculate each path's available bandwidth so that it can choose a path based on the Worst-Fit strategy. These procedures access HashMaps that are continually overwritten by the FlowStatCollector and portStats thread respectively. To ensure that the data that Hedera accesses are correct and that Thread Safety is enforced, Hedera and the aforementioned threads are secured with Reentrant Locks. More specifically, the ReentrantReadWriteLock is applied where Hedera is the reader and the other two threads are the writers of their respective locks. This kind of lock works by not allowing a reader to read the protected variable while a writer edits it.

4.3.6 Monitoring

In order for this experiment to have a conclusion, metrics need to be monitored and saved. A thread responsible for doing that is created and run between intervals of 10 seconds. This thread is responsible for calculating the two following metrics.

Core Link Load Balancing Level is calculated by dividing the max link load of links that connect core switches with aggregate switches, by the average link load of these links. With this metric one can understand how efficiently the load balancing algorithm handles the load of the network. When this metric is equal to 1 the load balancing of the network is optimal. Core Link Load Balancing Level is expressed by a pure number. This means that the numbers that represent this metric have no measurement units attached to them.

Bisection Bandwidth is the average utilization of the core links. The utilization of a link is calculated by dividing the link's consumed bandwidth by the link's capacity. Bisection Bandwidth translates into the spread of the load across the core links. A higher Bisection Bandwidth means that the load is spread across more core links than lower Bisection Bandwidths. Due to that, for better load balancing in the network, higher Bisection Bandwidth values are desired. This alternation of Bisection Bandwidth has no measurement unit attached to it, is a pure number as well, due to its calculation method.

The monitoring thread outputs these metrics in Comma-separated Values (CSV) format so that they can be easily read and transferred to MS Excel.

5 Evaluation Results

5.1 Overview

The experiment's goal is to compare ECMP against Hedera in the same test environment, while using the same communication patterns to generate the resulting metrics. These metrics will help evaluate each algorithm's performance.

The experiment begins by initiating the controller and the topology. After that, Floodlight needs time to fully discover the topology, which usually requires three rounds of sending LLDP packets. The full discovery of the topology is confirmed by a ping all command. This is the reason that ECMP installs flow entries for ICMP packets. The experiment begins with 20 seconds of idle time. After this idle time a script is run that initiates D-ITG's send command to all the hosts. The metrics include 20 seconds after D-ITG has finished sending packets from all hosts.

Two communication patterns are used in this experiment:

- Stride(i): A host with index: x, sends to the host with index: $(x + i) mod (number\_of\_hosts)$. Here $i = 30$, is chosen so that every host sends to a host, who is outside of the sender's pod. This way all traffic runs through the core links, which are monitored from the monitoring thread.
- Random: A host sends to any other host in the network. The mappings here are bijective, meaning that each host receives flows from one distinct sender.

As mentioned in the paragraph above, the concurrent initiation of D-ITG's send in all the hosts, is done by a command, which Mininet exposes in its command line interface *source*. This command instructs Mininet to read commands from an input file. Each row of the input file contains a command, destined for a host in the network. Each command uses Mininet's *x* command, which creates an X11 tunnel to the given host and runs the following command. In simpler terms with the help of the *x* command, D-ITG's send command can be sent to all hosts so that the traffic generation can begin. An example of a command in the input file follows:

$$x\ h1\ /user/bin/ITGSend\ -a\ destination\_IP\ -rp\ destination's\ port$$
$$-U\ \text{min}\ \_\text{packet\_rate}\ \text{max}\ \_\text{packer\_rate}\ -\text{c packet\_size}$$
$$-\text{T Protocol} - \text{t duration\_in\_miliseconds}$$

As mentioned above *x* sends the following command to be run in the following node in this example host one. The command that is run in the node is ITGSend and explanation for its parameters will follow.

1. -a: is for entering the IP address of the desired destination
2. -rp: this sets the port number that will be used to send the packets
3. -U: this parameter uses the Uniform distribution to send packets at rates between the minimum and maximum set values.
4. -c: this parameter sets the constant size of the packets sent
5. -T: this parameter sets the Protocol of the packets
6. -t: is the duration in milliseconds that the ITGSend command will send packets

The experiment consists of four scenarios. The two first scenarios are run using ECMP with Stride(30) and random communication patterns. Likewise, the next two scenarios are run having Hedera enabled on top of ECMP with the same communication patterns. All the scenarios last 180 seconds, 40 of which are idle time (20 seconds before ITGSend and 20 seconds after) and 140 seconds that D-ITG generates traffic. The protocol used in the experiment is UDP, and the size of the packets is constant at 1500 bytes. The packet rate is changing according to the Uniform distribution [37] between 1000 packets per second and 2000 packets per second. UDP port 7000 is chosen. It should be noted that flow entries concerning UDP packets use the destination port number as a matching parameter.

The resulting metrics, Core Link Load Balancing Level and Bisection Bandwidth, due to the way they are calculated have no measurement units attached to their values. These values are translated as described below:

Core Link Load Balancing Level starts from the value 1 which represents the optimal load balancing level of a network. The bigger the value is the worse is the load balancing in the network is.

Bisection Bandwidth translates to how spread is the traffic across all the core links. The bigger its value the more spread is the traffic, which means better load balancing in the network. Bigger spread of the network's traffic equals to better load balancing, because this way link congestion is avoided and link utilization is increased.

In the following sub-chapters, each scenario will be presented along with its results. Each scenario is run 10 times in order to have a sufficient sample to produce averages. These averages are used to create plots that will be presented in the sub-chapters below. The results from the 10 runs of each scenario are included in the Appendix section of this thesis.

5.2 Scenario 1: ECMP with Random communication pattern

In this scenario Hedera is disabled, only ECMP installs flow entries. The experiment's procedure was followed as explained above. This scenario was executed 10 times using the same parameters in order to produce a substantial sample so that accurate averages can be produced. Here, the traffic that was generated was with a random communication pattern. While the full results of all 10 runs are included in the Appendix section, below a table is shown with the average values across these 10 runs. This table uses the acronym CLLBL, which stands for Core Link Load Balancing Level, BB, which stands for Bisection Bandwidth, RX which stands for receiving traffic and TX for transmitting traffic. It should be noted that these acronyms are used in the tables of the following sub-chapters as well.

Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.67492	1.64636	0.0000092	0.0000094
20	1.46728	1.34817	0.0000235	0.0000248
30	5.27475	5.45535	0.0828052	0.0784918
40	4.63177	4.83548	0.2566411	0.2424418
50	4.10872	4.16565	0.2525916	0.2515538
60	4.00046	4.00049	0.2473116	0.2473249
70	4.00035	4.00045	0.2500303	0.2500253
80	4.00043	4.00052	0.2462330	0.2462374
90	4.00045	4.00050	0.2517574	0.2517502
100	4.00052	4.00023	0.2506221	0.2506344
110	4.00052	4.00061	0.2496569	0.2496550
120	4.00058	4.00049	0.2544772	0.2544923
130	4.00060	4.00033	0.2500995	0.2500991
140	4.00042	4.00037	0.2473664	0.2473659
150	4.27890	4.27885	0.2351991	0.2394821
160	5.45692	5.44046	0.1250814	0.1333910
170	1.48048	1.54335	0.0000147	0.0000144
180	1.46861	1.37474	0.0000060	0.0000062

Table 5.1 - Averages of Scenario with ECMP and Random communication pattern

In the first 20 seconds, since the system is idle, both Bisection Bandwidth and Core Link Load Balancing Level values are low. After the initiation of the traffic generation the Core Link Load Balancing Level rises and then settles to around 4. Bisection Bandwidth value rises linearly and

settles around 0.25 for the whole 140 seconds that D-ITG sends flows. The last 20 seconds see the decline of the metrics' values, since the system returns into an idle state again.

5.3 Scenario 2: ECMP with Stride(30) communication pattern

In this scenario Hedera is again disabled, only ECMP installs flow entries. The experiment's procedure was followed as explained above. This scenario was executed 10 times using the same parameters in order to produce a substantial sample so that accurate averages can be produced. Here, traffic was generated according to the stride(30) communication pattern. While the full results of all 10 runs are included in the Appendix section, below a table is shown with the average values across these 10 runs.

Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.61072	1.49655	0.0000111	0.0000112
20	1.53841	1.56074	0.0000274	0.0000274
30	5.80504	6.20632	0.0481615	0.0421557
40	4.71032	5.08959	0.2022496	0.1930783
50	4.16978	4.16372	0.2451072	0.2416352
60	4.03528	4.10535	0.2516636	0.2493381
70	4.00791	4.01783	0.2477546	0.2471717
80	4.00045	4.00036	0.2499452	0.2499421
90	4.00025	4.00020	0.2488182	0.2488119
100	4.00064	4.00055	0.2517892	0.2517958
110	4.00046	4.00047	0.2510299	0.2510268
120	4.00029	4.00041	0.2499511	0.2499458
130	4.00041	4.00048	0.2501702	0.2501656
140	4.00042	4.00048	0.2512909	0.2512950
150	4.27782	4.22014	0.2298144	0.2317529
160	5.28188	5.15272	0.0766640	0.0733037
170	2.27254	2.21838	0.0000516	0.0000548
180	1.60758	1.65068	0.0000149	0.0000146

Table 5.2 - Scenario with ECMP and Stride(30) communication pattern

In the first 20 seconds the network is idle which explains the low Bisection Bandwidth values and the almost optimal Core Link Load Balancing Level. After the generations of traffic have started, an increase in the values of both metrics is observed. Core Link Load Balancing Level, again settles

around 4, while Bisection Bandwidth settles at a value around 0.25. The last 20 seconds have these values to be again decreased to portray the idle state of the system.

5.4 Scenario 3: Hedera with Random communication pattern

This scenario has Hedera enabled along with ECMP. Here when the topology is initialized and connected to Floodlight, ECMP will install flow entries for communication between all hosts. Hedera will run between its intervals and check all existing flows for flows that consume bandwidth bigger than the 10% of the link capacity of the link that connects the host to the edge switch. When such a flow is found Hedera applies its routing strategy, in this experiment Worst-fit, and installs flow entries that have higher priority than ECMP's. From that point and until the flow ends, at which point the flow entries would be deleted due to their idle timeout parameter, Hedera is responsible for the routing of the flow. Again, this scenario was run 10 times, using the same parameters, in order to produce a substantial sample so that averages can be produced. The full results of all 10 run are included in the Appendix section. Following is a table with the average values across these 10 runs.

Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.76319	1.69839	0.0000103	0.0000108
20	1.28626	1.32218	0.0000243	0.0000236
30	5.68760	5.72756	0.0702962	0.0628335
40	4.38500	4.64964	0.2025906	0.1984964
50	2.77405	2.74331	0.3327972	0.3397117
60	2.39159	2.32005	0.3810299	0.3840739
70	2.38328	2.35463	0.4096218	0.4095991
80	2.38682	2.37098	0.4005013	0.4007132
90	2.38554	2.39350	0.4001667	0.4002022
100	2.40055	2.38533	0.4022134	0.4021941
110	2.42390	2.40248	0.4009736	0.4009388
120	2.42642	2.40513	0.4128057	0.4128035
130	2.42551	2.40504	0.4106867	0.4107020
140	2.42624	2.40484	0.4006728	0.4006639
150	2.55745	2.48414	0.3722648	0.3725752
160	3.98297	3.81505	0.0642307	0.0616984
170	1.73467	1.64784	0.0000421	0.0000487
180	1.13477	1.39050	0.0000072	0.0000077

Table 5.3 - Averages of Scenario with Hedera enabled and Random communication pattern

The first 20 seconds have the idle network to produce low values for both Bisection Bandwidth and Core Link Load Balancing Level, as expected. When the traffic generation begins these values rise to levels like the previous scenarios. This is normal since here ECMP is actually routing the traffic. However, at the 50 second mark and on, the metrics' values don't display similar behavior compared to the previous scenarios. With Hedera routing the traffic, Core Link Load Balancing Level value settles around 2.4. This is a significant improvement compared to ECMP's results that were shown in the corresponding scenario. Also, Bisection Bandwidth's value settles around 0.4 which is higher than the value of the corresponding value of ECMP. Both results show that the network has better load balancing when Hedera is applied. The last 20 seconds, as expected show the typical decrease in the metrics' values that show that they system is idle.

5.5 Scenario 4: Hedera with Stride(30) communication pattern

In this scenario Hedera is enabled, as well. When the topology is initialized and connected to Floodlight, ECMP installs flow entries for communication between all hosts. Hedera will run between its intervals and check all existing flows for flows that consume bandwidth bigger than the 10% of the link capacity of the link that connects the host to the edge switch. When such a flow is found Hedera applies its routing strategy, in this experiment Worst-fit and installs flow entries that have higher priority than ECMP's. From that point and until the flow ends, at which point the flow entries would be deleted due to their idle timeout parameter, Hedera is responsible for the routing of the flow. The traffic pattern generated was according to the stride(30) communication pattern. This scenario was run 10 times, using the same parameters, in order to produce a substantial sample so that averages can be produced. While the full results of all 10 runs are included in the Appendix section, below a table is shown with the average values across these 10 runs.

Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.49120	1.48649	0.0000125	0.0000117
20	1.50472	1.50363	0.0000219	0.0000218
30	5.65605	6.27246	0.0774385	0.0660819
40	3.95534	4.17549	0.2670288	0.2572713
50	3.14810	3.17321	0.3232535	0.3206467
60	3.06267	3.06285	0.3250515	0.3250298
70	2.85449	2.85431	0.3513133	0.3513378
80	2.76502	2.76502	0.3631875	0.3631992
90	2.52486	2.52347	0.3996837	0.4000444
100	2.35890	2.36082	0.4262155	0.4258547
110	2.11435	2.12113	0.4582252	0.4590212
120	1.95699	1.98382	0.4900499	0.4908497

130	1.85899	1.89407	0.4830529	0.4844803
140	1.69678	1.73938	0.4900793	0.4916954
150	3.24296	3.13516	0.2524954	0.2524932
160	3.91541	3.86599	0.0699909	0.0700859
170	1.53193	1.55811	0.0000489	0.0000493
180	1.32340	1.27144	0.0000145	0.0000176

Table 5.4 - Averages of Scenario with Hedera enabled and Stride(30) communication pattern

The first 20 seconds the network is idle and produces low values for both Bisection Bandwidth and Core Link Load Balancing Level, as expected. When the traffic generation begins these values rise to levels like the previous scenarios. This is normal since here ECMP is routing the traffic. However, at the 50 second mark and beyond, the metrics' values don't display similar behavior compared to the previous ECMP scenario. With Hedera routing the traffic, Core Link Load Balancing Level value settles around 2.5. This is a significant improvement compared to ECMP's results that were shown in the corresponding scenario. Also, Bisection Bandwidth's value settles around 0.45 which is higher than the value of the corresponding value of ECMP. Both results show that the network has better load balancing when Hedera is applied. The last 20 seconds, as expected show the typical decrease in the metrics' values that show that they system is idle.

5.6 Comparison

The algorithms used in the experiment were: ECMP and Hedera. They were used in similar scenarios, in order to collect results and make a comparison between the two. ECMP is a static load balancing algorithm, while Hedera is a dynamic. In the first two scenarios ECMP was used alone with two different communication patters, stride(30) and random. The following two scenarios used the same communication patterns but had Hedera enabled to work on top of ECMP. The resulting metrics show that when Hedera was applied the load balancing in the network was better. This is done because Hedera tries to spread the load across different paths in the network, to avoid congestion. Four charts were created to better represent the results and to visualize the comparison between the two load balancing methods.

In the following sub-chapters these charts will be displayed, and observations will be made for the resulting values of the metrics.

5.6.1 Comparison for Random communication pattern

The chart below contains the average values of ECMP's and Hedera's Core Link Load Balancing Levels.

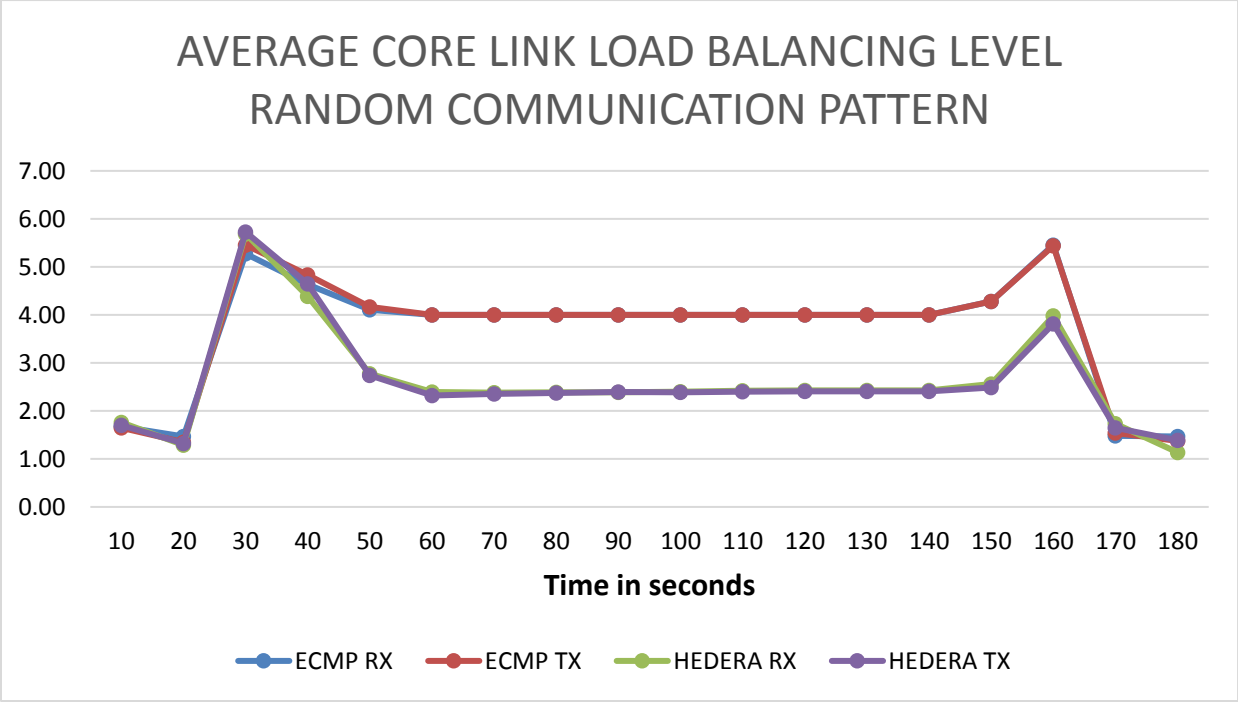

Chart 1. Average Core Link Load Balancing Level for Random communication pattern

It should be reminded that Core Link Load Balancing Level has no measurement unit attached to it, it is a pure number. When its values are close to 1 or even equal to 1, then the network is load balanced optimally. In the above chart the following are observed:

1. Receiving (RX) and Transmitting (TX) metrics average at the same values. This is expected, since in the network no other flows or packets are sent during the experiment's duration.
2. Until the 20 second mark the values of both ECMP and Hedera are similar.
3. When the traffic generator starts to send flows, both ECMP's and Hedera's scenario results seem to be the same. This is, again expected because Hedera has ECMP route the flows, until a flow consumes more than 10% of the links capacity.
4. After Hedera starts to route flows, Core Link Load Balancing Level values decrease almost by half than ECMP's. This shows, the difference between the states of load balancing in the networks of the two scenarios.
5. The rest of the experiment has the two metrics to show similar behavior, with Hedera's keeping lower values across the experiment. This proves that Hedera indeed balanced the load better than ECMP.

In this chart, averages of Bisection Bandwidth of ECMP's and Hedera's scenario are displayed.

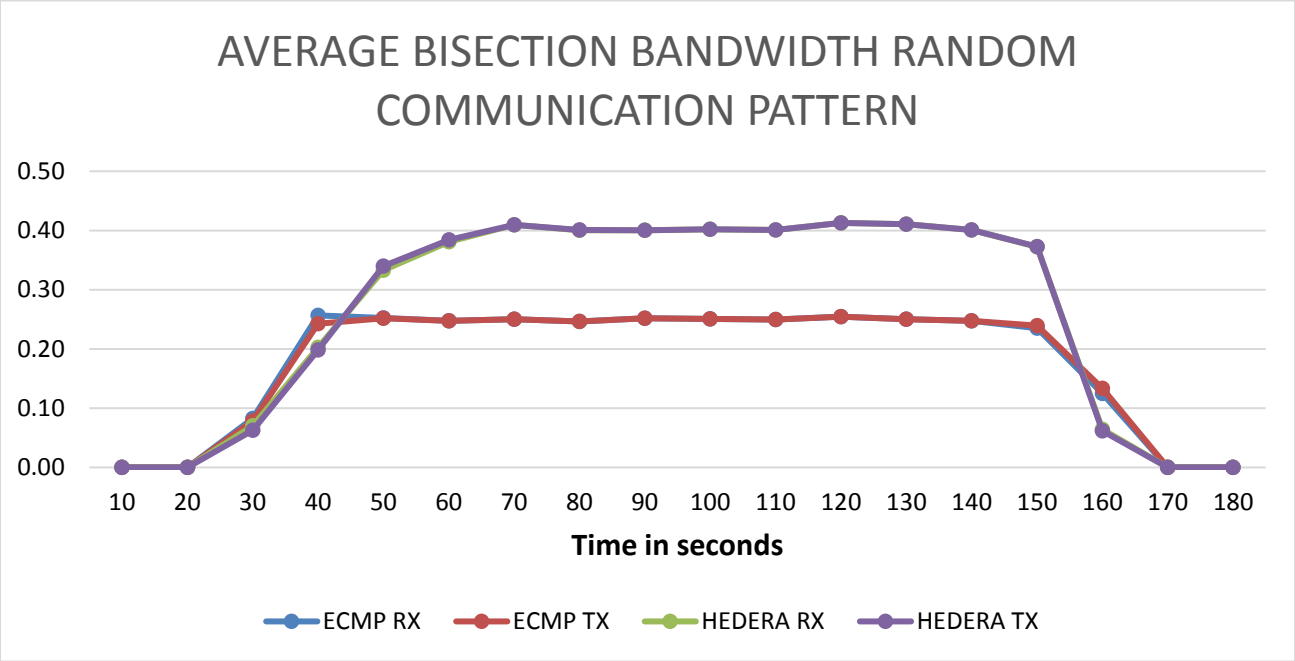

Chart 2. Average Bisection Bandwidth for Random communication pattern

This implementation of Bisection Bandwidth is a pure number as well. Its values are interpreted as the spread of the load across the core links. The bigger the value the more spread is the load, which translates into better load balancing across the network. In this chart the below are observed:

1. The RX and TX values of ECMP and Hedera are the same here as well.
2. Like Core Link Load Balancing Level, the values of Bisection Bandwidth for ECMP and Hedera are almost the same. However, as soon as Hedera starts routing flows, its values of Bisection Bandwidth rise, while ECMP's stay at around the same. This shows that, when Hedera is enabled, the traffic is better spread across the core links and in turn better load balancing.
3. Hedera's Bisection Bandwidth values are higher than ECMP's until the traffic generation has stopped, in which point both ECMP's and Hedera's values decrease to almost 0.
4. This graph shows that Hedera almost doubles ECMP's Bisection Bandwidth values in the time window that traffic generation is active. This translates in significant improvement of the balancing in the network in Hedera's scenario.

These results are expected. ECMP chooses the paths with the smallest number of hops. This results in many paths sharing the same links. When under load, these links are overflown by flows, while other links in the network remain unused. Hedera on the other side, with its Worst-fit implementation tries to spread the load so that links in the network are not left unused. This difference is the reason that the resulting metrics favor Hedera.

It should be also noted that Hedera is never applied immediately. ECMP routes the flows and Hedera checks them between the intervals that its thread is run. This means that depending on the interval that Hedera is run, it may yield better or worse results depending on if Hedera was applied earlier or later on flows that consume a lot of bandwidth. In this experiment since the interval was the same across the runs Hedera yielded similar results because it was run between the same intervals.

5.6.2 Comparison for Stride(30) communication pattern

The following charts contain the results of ECMP's and Hedera's of the two metrics of the scenarios that used the stride(30) communication pattern.

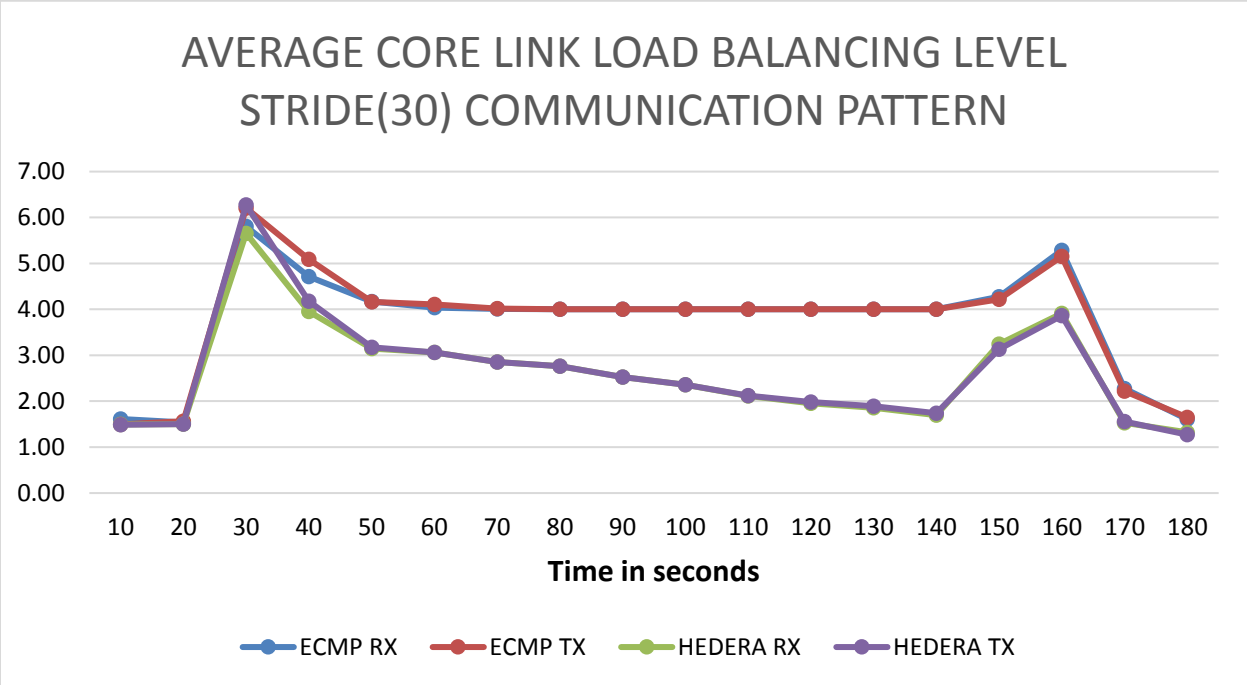

Chart 3. Average Core Link Load Balancing Level for Stride(30) communication pattern

From the above chart the following are observed:

1. Like in previous scenarios, the RX and TX average values of ECMP and Hedera respectively are the almost equal.

2. Here, Core Link Load Balancing Level values for both the algorithms are almost the same up until the moment, that Hedera starts routing flows.
3. After Hedera starts routing flows, its Core Link Load Balancing Level, like the scenario with random communication pattern is significantly lower than ECMP's.
4. It should be noted that, while ECMP's Core Link Load Balancing Level values are almost the same, Hedera's after the 80 second mark decrease and come below 2. This is unlike the other scenario, in which Hedera's values were stable at around 2.5. Here, this decrease is due to the fact that with stride(30) communication pattern all hosts send to hosts outside of their pod. This means that all the flows go through the core links, which in turn translates in opportunity for Hedera to balance more flows.

Below follows the chart for the Bisection Bandwidth of ECMP and Hedera for the scenarios that used stride(30) communication pattern.

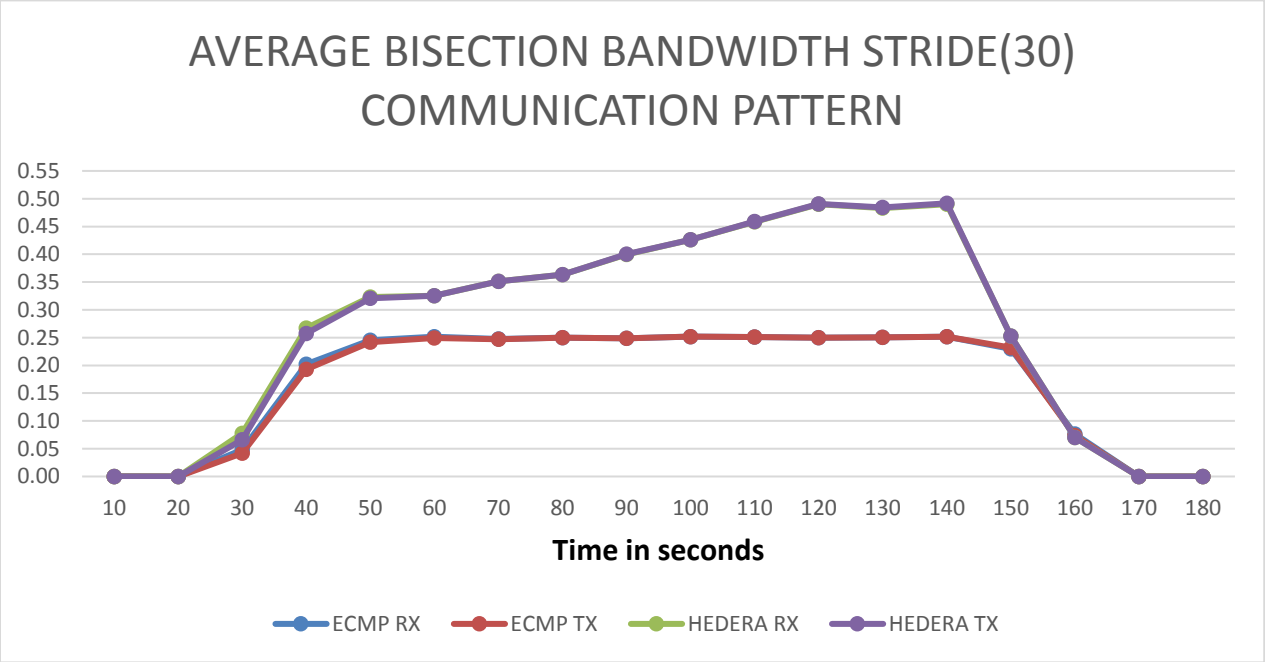

Chart 4. Average Bisection Bandwidth for Stride(30) communication pattern

From the above plot, the following are observed:

1. In here, as well RX and TX average values of ECMP and Hedera respectively are almost equal.
2. Bisection Bandwidth values of ECMP and Hedera are similar until traffic generation begins.

3. After traffic generation has begun, the Bisection Bandwidth values of Hedera increase more than ECMP's. This increase, proves again, like Core Link Load Balancing Level displayed in the previous chart, that Hedera balances significantly better the network.
4. In the previous chart, it was observed that Core Link Load Balancing for Hedera had a declining tendency for the duration that traffic generation was enabled. This is also, observed here, where Bisection Bandwidth for Hedera has an increasing tendency up until the moment that the traffic generations stops. This shows that in stride(30) communication pattern Hedera is able to better balance the network than ECMP and itself in the random communication pattern scenario.

Here the results are similar with the results from the scenarios that used random communication patterns. The main difference here is that with stride(30), all hosts send to hosts that are outside of their pod. This means that more traffic travels through the core links that are measured to produce the resulting metrics. This increase in traffic does not change ECMP's results.

On the other hand, Hedera in both metrics displays a tendency to improve its result, with increasing Bisection Bandwidth and decreasing Core Link Load Balancing Level. This is done because there are more flows that are spread across the core links by Hedera. Worst-Fit ensures that each flow will be relocated to the path that has the most available bandwidth, which in turn results in better load balancing across the network.

5.7 Evaluation

Based on the above plots and results, it is shown that Hedera is the better load balancing algorithm. It consistently outputs values that show that the network is better balanced than ECMP. Also, it is shown that when using the stride(30) communication pattern, where every host sends packets to another host, who is outside its pod, Hedera performed better than with the random communication pattern, in which some hosts sent packets to hosts inside their pod.

Overall Hedera performed better than ECMP. It had with both communication patterns results that portray better load balancing across the network. In order to have a visualization of this conclusion, charts displaying the averages of the average results of both metrics are included below.

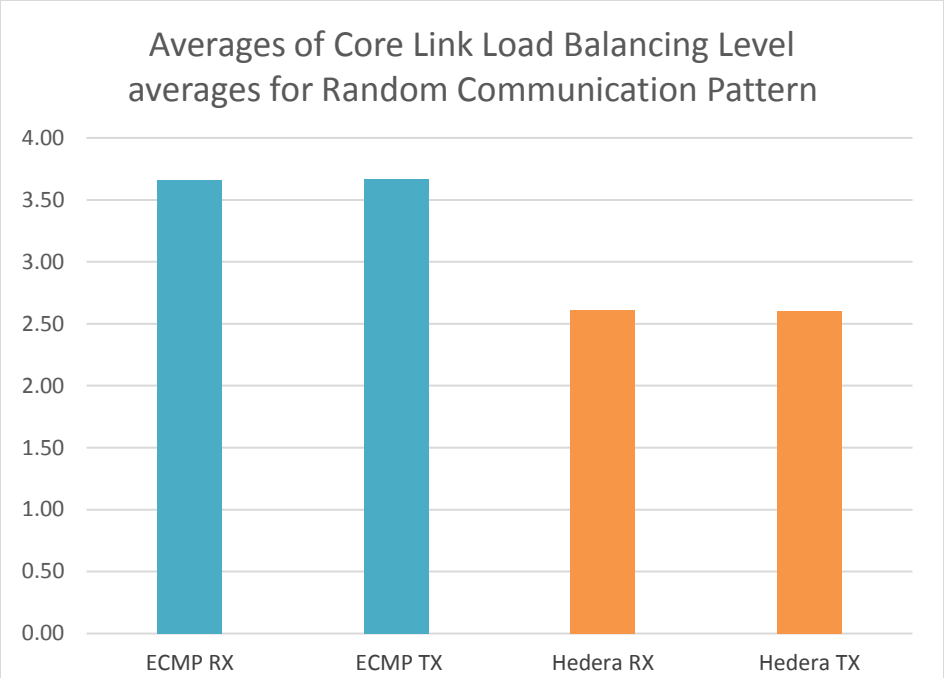

Chart 5. Average Core Link Load Balancing Level for Random communication pattern

The above chart depicts the average value of Core Link Load Balancing Level throughout the experiment. This chart further proves that Hedera balanced the network better than ECMP throughout the experiment. The average Core Link Load Balancing Level value of Hedera is lower that ECMP's. This is translated into better load balancing from Hedera.

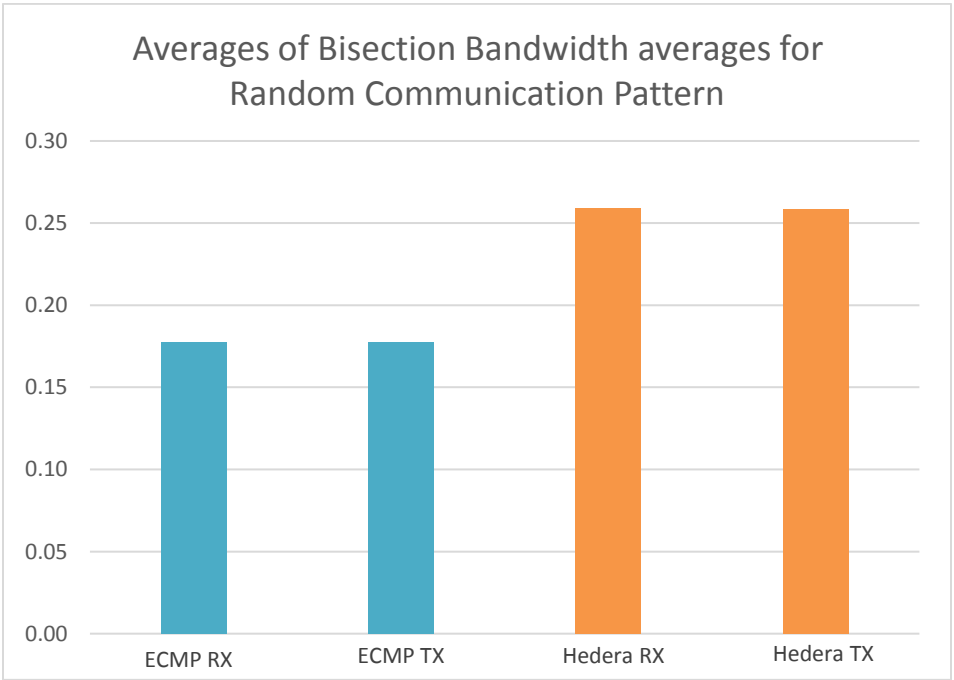

Chart 6. Average Bisection Bandwidth for Random communication pattern

In the above chart the average Bisection Bandwidth values are displayed. This chart shows, that throughout the experiment when the random communication pattern was applied, Hedera outputted higher Bisection Bandwidth than ECMP. This in turn backs up the findings from Core Link Load Balancing Level and further proves that Hedera balanced the network better.

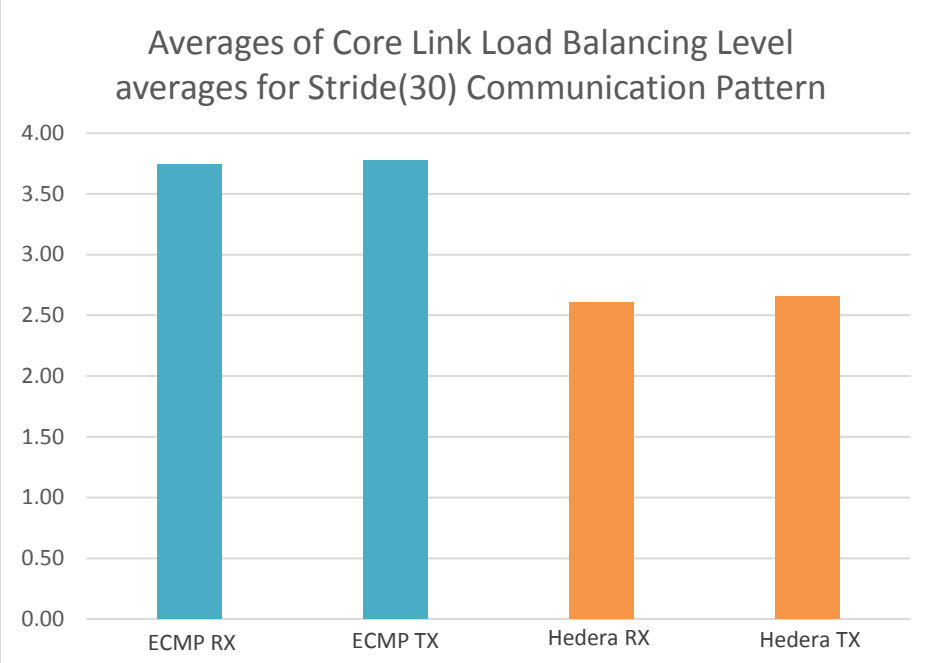

Chart 7. Average Core Link Load Balancing Level for Stride(30) communication pattern

The above chart presents the Core Link Load Balancing average values when using the stride(30) communication pattern. Similarly with the results from the random pattern, hedera outputs lower values than ECMP. This means that Hedera throughout the runs that stride(30) was applied, was balancing better the load of the network.

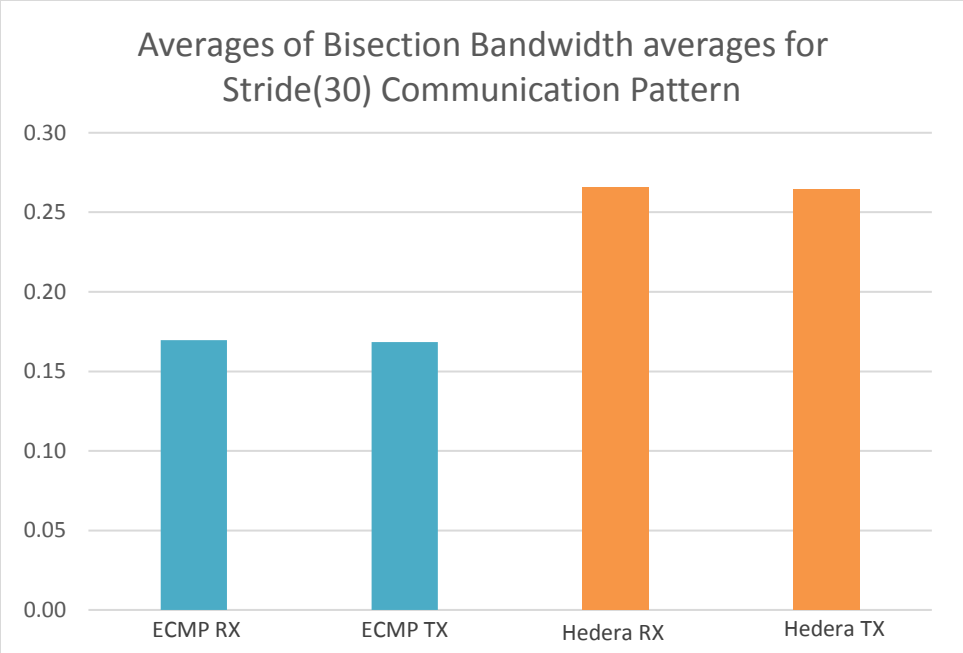

Chart 8. Average Bisection Bandwidth for Stride(30) communication pattern

In this chart, the average Bisection Bandwidth for the stride(30) is offered. As expected, the average Bisection Bandwidth that Hedera outputs is higher than ECMP's. This further proves, Core Link Load Balancing Level's results, that conclude that Hedera balances better the load in the network.

These averages express the overall performance of each algorithm across all the runs and scenarios. ECMP selects proactively paths for all the available pairs of hosts in the network. These paths have the least number of hops between the source and destination host. This tactic yields the same results (performance) in this experiment due to the fact that traffic parameters and communication patterns of each scenario are kept the same. ECMP's performance degrades even more, when more traffic is introduced in the network. Hosts from the same pods are inclined to use the same switches as in-between hops in their paths and that results in the overuse of the links that comprise these paths.

In fat tree topologies there are alternative paths between any pair of hosts equal to *k* which is four for this experiment. These alternative paths are used by Hedera in order to spread the traffic between them. It should be noted here, that Hedera can use different placement algorithms. Worst-fit is not the most efficient but it was chosen for its simple logic and less complex implementation. More complex placement algorithms like Simulated Annealing [46], which performs a probabilistic search to efficiently compute paths for flows, would yield even better results because the traffic would be better balanced across the network.

Concluding, the results are explained through the algorithms' categories. ECMP, a static algorithm, installs flows proactively and selects the "shorter" paths. Hedera, a dynamic algorithm, on the other hand responds on the needs of the network, preventing link congestion and using links on the network that were otherwise unused.

6 Conclusions and future work

In this thesis, data centers and their networks, topologies and architectures were presented. Also, load balancing was explored along with the categories it is divided into. Moreover, some load balancing algorithms were presented. Following that, the experiment for comparing ECMP against Hedera was introduced and analyzed. Lastly, the experiment's results were presented and evaluated.

6.1 Conclusions

In the experiment, a 3-level fat tree topology with 80 end hosts was simulated. This topology was connected to a controller, Floodlight. From Floodlight the load balancing algorithms ECMP and Hedera routed the network, by installing flow entries in the switches through OpenFlow. These algorithms were compared by having them try to load balance the network while traffic was generated. The comparison was made by two metrics that were produced throughout the process of the experiment.

By analyzing the results for Core Link Load Balancing Level and Bisection Bandwidth of the network it was determined that Hedera, the dynamic algorithm was able to balance the network better than ECMP with both random and stride(30) communication patterns. It should be noted that the difference in the results of these metric was substantial, with Hedera generating significantly better results than ECMP.

The conclusion of this thesis is that a dynamic algorithm is better fit to balance a network. This was concluded by the experiment, which was run a lot of time in order for the results to be accurate. The dynamic algorithm, Hedera showed better performance under load by outputting lower Core Link Load Balancing Levels and higher Bisection Bandwidth values throughout the experiment consistently.

6.2 Limitations of this work

The most important limitation of this thesis was the lack of real hardware. To compensate for that, Mininet was used. Consequently, the size of the topology, the number of end hosts and the amount of data traffic that was generated was limited due to hardware restrictions.

Another limiting factor was the lack of more algorithms. This was because either algorithms were too complex to implement or that an implementation for that complex algorithm couldn't be found. Also due to the nature of the test environment distributed algorithms couldn't be implemented and therefore tested.

6.3 Future Work

It would be interesting to implement the experiment in real hardware in order to compare the results against the results from the simulated experiment. Also, because using real hardware would mean having a lot more computing power, the experiment's scale could be bigger. That means bigger topology, with more switches and hosts and the ability to generate a lot more flows in the time frame of the experiment.

Another expansion could be the implementation of more algorithms dynamic or static that are mentioned in the related chapter. An interesting suggestion is to include distributed algorithms that install their logic in the switches of the network. Interesting distributed algorithms to be included in a future expansion would be ones similar to DRILL, that instead of collecting global congestion information, rely on local switch information. It should be interesting to see if this logic has actual benefits. By not having to collect global congestion information the network overhead is eliminated but the actual routing is done by random chance and simple logic.

References

[1] A. Vahdat et al., Scale-Out Networking in the Data Center, IEEE Micro 2010

[2] R. Mysore et al., PortLand: A Scalable Fault-Tolerant Layer Data Center Network Fabric, ACM SIGCOMM 2009

[3] F. Yao, J. Wu, G. Venkataramani and S. Subramaniam, "A Comparative Analysis of Data Center Network Architectures", Proceedings of International Conference on Communications IEEE, 2014.

[4] T. Benson, et al., Network Traffic Characteristics of Data Centers in the Wild, ACM IMC 2010

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks,” in Proc. USENIX NSDI, San Jose, CA, USA, 2010

[6] Shuo Wang, Jiao Zhang, Tao Huang, Tian Pan, Jiang Liu and Yunjie Liu, "FDALB: Flow distribution aware load balancing for datacenter networks," 2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS), Beijing, 2016

[7] W. Wang et al., “Freeway: Adaptively isolating the elephant and mice flows on different transmission paths,” in Proc. IEEE ICNP, Raleigh, NC, USA, 2014

[8] N. Liu, Z. Dong, and R. Rojas-Cessa, “Task scheduling and server provisioning for energy-efficient cloud-computing data centers,” in Proc. IEEE ICDCS, Philadelphia, PA, USA, 2013

[9] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, “FlowBender: Flow-level adaptive routing for improved latency and throughput in datacenter networks,” in Proc. ACM CoNEXT, Sydney, NSW, Australia, 2014

[10] N. Katta et al., “CLOVE: How I learned to stop worrying about the core and love the edge,” in Proc. ACM HotNets, Atlanta, GA, USA, 2016

[11]] H. Xu and B. Li, “RepFlow: Minimizing flow completion times with replicated flows in data centers,” in Proc. IEEE INFOCOM, Toronto, ON, Canada, 2014

[12] S. Ghorbani, B. Godfrey, Y. Ganjali, and A. Firoozshahian, “Micro load balancing in data centers with DRILL,” in Proc. ACM HotNets, Philadelphia, PA, USA, 2015

[13] H. Xu and B. Li, “TinyFlow: Breaking elephants down into mice in data center network,” in Proc. IEEE LANMAN, Reno, NV, USA, 2014

[14] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in Proc. ACM SIGCOMM, Seattle, WA, USA, 2008

[15] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O’Shea, and A. Donnelly, “Symbiotic routing in future data centers,” in SIGCOMM, 2010

[16] By Yixuan Li - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=53610656

[17] J. Zhang, F. R. Yu, S. Wang, T. Huang, Z. Liu, and Y. Liu, “Load Balancing in Data Center Networks: A Survey,” IEEE Communications Surveys & Tutorials, 2018
[18] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, “BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers,” in SIGCOMM, 2009
[19] J. Kim, W. J. Dally, and D. Abts, “Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks,” in ISCA, 2007
[20] R. Godha and S. Prateek, “Load Balancing in a Network,” International Journal of Scientific and Research Publications, 2014
[21] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China, Q. Du, and H. Zhuang, “OpenFlow-Based Dynamic Server Cluster Load Balancing with Measurement Support,” Journal of Communications, 2015
[22] “Documentation mininet/mininet Wiki GitHub.” [Online] Available: https://github.com/mininet/mininet/wiki/Documentation
[23] “Floodlight OpenFlow Controller” Project Floodlight. [Online] Available: http://www.projectfloodlight.org/floodlight/
[24] “D-ITG (Distributed Internet Traffic Generator)” DITG. [Online]. Available: https://github.com/jbucar/ditg
[25] S. Avallone, S. Guadagno, D. Emma, A. Pescape and G. Ventre, "D-ITG distributed Internet traffic generator," First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings., 2004
[26] Javaid, Adeel. (2013). Understanding Dijkstra Algorithm. SSRN Electronic Journal
[27] C. Kim et al., Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises, ACM SIGCOMM 2008
[28] R. Perlman, "Challenges and Opportunities in the Design of TRILL: A Routed Layer 2 Technology," 2009 IEEE Globecom Workshops, 2009
[29] “Software-Defined Networking (SDN) Definition”. [Online]. Available: https://www.opennetworking.org/sdn-definition/
[30] “SDN Overview.” [Online]. Available: https://www.datacomm.co.id/en/telco/sdn/
[31] N. McKeown et al., “OpenFlow: Enabling Innovation in Campus Networks”, ACM SIGCOMM Comput. Commun Rev.
[32] O.N Foundation, “OpenFlow Switch Specification 1.5.1”,vol. 0, pp. 1-36, 2015
[33] C. Metz, “VMware Pays $1.26B for the Future of Networking”, 2012. [Online]. Available: https://www.wired.com/2012/07/vmware-buys-nicira/
[34] B. Lantz, B. Heller, and N. McKeown. 2010. “A network in a laptop”, 9th ACM SIGCOMM Workshop on Hot Topics in Networks – Hotnets, 2010

[35] Greenberg, Albert, et al. "VL2: A scalable and flexible data center network." Proceedings of the ACM SIGCOMM 2009 conference on Data communication. 2009.
[36] “A quick summary of the VL2 data-center network scheme”. [Online]. Available: https://blog.moertel.com/posts/2011-03-17-a-quick-summary-of-the-vl2-data-center-network-scheme.html
[37] Kuipers, Lauwerens, and Harald Niederreiter. Uniform distribution of sequences. Courier Corporation, 2012
[38] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/univercd/cc/td/doc/ solution/dcidg21.pdf
[39] Rajan, Rajesh George, and V. Jeyakrishnan. "A survey on load balancing in cloud computing environments." International Journal of Advanced Research in Computer and Communication Engineering, 2013
[40] Benson, Theophilus, et al. "Understanding data center traffic characteristics." ACM SIGCOMM Computer Communication Review 40.1 2010
[41] Pradhan, Pandaba, Prafulla Ku Behera, and B. N. B. Ray. "Modified round robin algorithm for resource allocation in cloud computing." Procedia Computer Science 85 2016
[42] A. Khiyaita, H. E. Bakkali, M. Zbakh and D. E. Kettani, "Load balancing cloud computing: State of art," 2012 National Days of Network Security and Systems, 2012
[43] “OpenFlow Match Structure”. [Online]. Available: http://flowgrammable.org/sdn/openflow/classifiers/#tab_ofp_1_3_0
[44] L. V. Morales, A. F. Murillo and S. J. Rueda, "Extending the Floodlight Controller," 2015 IEEE 14th International Symposium on Network Computing and Applications, 2015
[45] HOPPS, C. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, IETF, 2000
[46] Van Laarhoven, Peter JM, and Emile HL Aarts. "Simulated annealing." Simulated annealing: Theory and applications. Springer, Dordrecht, 1987

Appendix

Results from runs with ECMP and random communication pattern

Run 1				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.73825	1.65118	0.0000098	0.0000104
20	1.36245	1.39286	0.0000229	0.0000224
30	5.13080	5.62731	0.0762785	0.0668655
40	4.05810	4.12585	0.2468275	0.2427293
50	4.00040	4.00034	0.2485336	0.2485401
60	4.00026	4.00036	0.2467587	0.2467965
70	4.00026	4.00036	0.2467587	0.2467965
80	4.00017	4.00032	0.2466760	0.2467038
90	4.00084	4.00038	0.2527041	0.2526857
100	4.00021	4.00014	0.2502887	0.2502918
110	4.00050	4.00034	0.2491635	0.2491629
120	4.00060	4.00054	0.2971033	0.2971527
130	4.00044	4.00028	0.2502122	0.2502181
140	4.00058	4.00080	0.2499643	0.2499750
150	4.23896	4.12268	0.2379886	0.2451370
160	5.38355	4.73948	0.1101995	0.1172574
170	1.64539	1.82677	0.0000070	0.0000063
180	1.39022	1.82822	0.0000077	0.0000081

Run 2				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.49550	1.56604	0.0000111	0.0000106
20	1.08711	1.08711	0.0000287	0.0000287
30	5.21014	5.78532	0.0405474	0.0805464
40	4.60210	4.91503	0.1809773	0.1664280
50	4.00066	4.00044	0.2493711	0.2493848
60	3.99997	4.00030	0.2304821	0.2304577
70	4.00033	4.00050	0.2724308	0.2724143

80	4.00055	4.00065	0.2466639	0.2466695
90	4.00008	4.00039	0.2469349	0.2469204
100	4.00062	4.00044	0.2532840	0.2533288
110	4.00024	4.00052	0.2498193	0.2498197
120	4.00037	4.00030	0.2502016	0.2502312
130	4.00106	4.00038	0.2496530	0.2496405
140	4.00045	4.00006	0.2499798	0.2499910
150	4.00021	4.00055	0.2500384	0.2500291
160	5.24345	5.70466	0.1643419	0.1763485
170	1.86545	1.75492	0.0000086	0.0000086
180	1.77164	1.46218	0.0000055	0.0000054

Run 3				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.47982	1.83333	0.0000102	0.0000108
20	1.35948	1.38976	0.0000123	0.0000224
30	4.80775	5.25307	0.0578448	0.0403948
40	4.00169	4.07084	0.2529305	0.2485317
50	4.00051	4.00071	0.2493828	0.2493720
60	4.00173	4.00176	0.2504988	0.2505457
70	4.00042	4.00073	0.2485373	0.2485220
80	4.00052	4.00053	0.2497692	0.2497722
90	4.00036	4.00007	0.2475312	0.2475472
100	4.00107	4.00042	0.2502767	0.2503068
110	4.00023	4.00040	0.2491054	0.2490951
120	4.00033	4.00047	0.2495797	0.2495748
130	4.00014	4.00026	0.2481460	0.2481509
140	4.00036	4.00026	0.2521200	0.2521215
150	4.81471	4.73795	0.2053702	0.2154625
160	5.38355	5.73948	0.1101995	0.1172574
170	1.84568	1.74196	0.0000098	0.0000044
180	1.54726	1.84143	0.0000086	0.0000098

Run 4				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.72917	1.72917	0.0000096	0.0000096
20	1.30544	1.30544	0.0000239	0.0000239
30	5.65989	5.98735	0.0897436	0.0954297
40	4.87909	4.82880	0.2428948	0.2048763
50	5.08163	5.65318	0.2773727	0.2669059
60	4.00040	4.00037	0.2500704	0.2500775
70	4.00060	4.00047	0.2448764	0.2448706
80	4.00060	4.00047	0.2448764	0.2448706
90	4.00024	4.00045	0.2499977	0.2499937
100	4.00079	4.00005	0.2502126	0.2502210
110	4.00017	4.00080	0.2510317	0.2510270
120	4.00120	4.00055	0.2500959	0.2501129
130	4.00018	4.00049	0.2500290	0.2500118
140	3.99995	4.00017	0.2497970	0.2497851
150	4.00063	4.00035	0.2497832	0.2498189
160	5.00080	5.00031	0.1498896	0.1499189
170	1.16917	1.14760	0.0000266	0.0000271
180	1.15445	1.22247	0.0000086	0.0000086

Run 5				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.81673	1.81673	0.0000094	0.0000094
20	1.25301	1.12230	0.0000249	0.0000278
30	5.53738	5.48184	0.0904201	0.0904232
40	5.10971	5.61365	0.2781475	0.2684193
50	4.00059	3.99998	0.2505966	0.2505933
60	4.00003	4.00035	0.2475661	0.2475702
70	4.00003	4.00035	0.2475661	0.2475702
80	4.00019	4.00026	0.2487503	0.2487529
90	4.00042	4.00072	0.2511260	0.2511112
100	4.00032	4.00028	0.2511018	0.2511099
110	4.00046	4.00058	0.2492638	0.2492801
120	4.00033	4.00055	0.2476454	0.2476801
130	4.00072	4.00067	0.2525183	0.2525071
140	4.00052	4.00031	0.2463976	0.2463965
150	4.00041	4.00063	0.2540057	0.2540176
160	5.90675	5.75033	0.1390021	0.1452136
170	1.82418	1.45877	0.0000091	0.0000086

180	1.00025	1.00055	0.0000015	0.0000015

Run 6				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.82418	1.72917	0.0000091	0.0000096
20	1.52195	1.30544	0.0000205	0.0000239
30	5.69700	5.69643	0.1006140	0.0906130
40	5.10933	5.61642	0.2805510	0.2704006
50	4.00043	3.99995	0.2498187	0.2498576
60	4.00029	4.00029	0.2455097	0.2455066
70	4.00029	4.00029	0.2455097	0.2455066
80	4.00033	4.00060	0.2441832	0.2441755
90	4.00060	4.00079	0.2586028	0.2585593
100	4.00005	4.00002	0.2505339	0.2505410
110	4.00025	4.00074	0.2490666	0.2490309
120	4.00024	4.00028	0.2503956	0.2504000
130	4.00077	4.00033	0.2496947	0.2497149
140	4.00015	4.00093	0.2499526	0.2499458
150	4.00108	4.00025	0.2500083	0.2500347
160	5.26785	5.38734	0.1364697	0.1420802
170	1.77778	1.82822	0.0000043	0.0000081
180	1.65400	1.00248	0.0000005	0.0000017

Run 7				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.72917	1.82418	0.0000096	0.0000091
20	2.29965	1.83916	0.0000144	0.0000143
30	5.61513	5.58317	0.1004616	0.0904646
40	5.09907	5.62024	0.2807700	0.2704892
50	4.00008	4.00015	0.2499504	0.2499553
60	4.00036	4.00013	0.2469717	0.2469586
70	4.00036	4.00013	0.2469717	0.2469586
80	4.00059	4.00034	0.2418561	0.2418968
90	4.00042	4.00045	0.2585747	0.2585450
100	4.00081	4.00011	0.2500531	0.2500505
110	4.00156	4.00070	0.2501077	0.2501216
120	4.00049	4.00043	0.2497162	0.2497491
130	4.00088	4.00020	0.2503801	0.2503814

140	4.00061	4.00025	0.2499087	0.2499152
150	4.00120	4.00062	0.2505523	0.2505884
160	5.93729	5.44794	0.0963290	0.0926096
170	1.22578	1.21451	0.0000086	0.0000086
180	1.54448	1.22589	0.0000096	0.0000096

Run 8				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.63614	1.55981	0.0000100	0.0000104
20	1.36243	1.39282	0.0000235	0.0000230
30	5.55969	4.86982	0.0850719	0.0699721
40	5.45668	5.51211	0.2888440	0.2412603
50	4.00069	4.00042	0.2503421	0.2503577
60	4.00041	4.00024	0.2534933	0.2535237
70	4.00056	4.00088	0.2502314	0.2501969
80	4.00069	4.00022	0.2426718	0.2427086
90	4.00025	4.00033	0.2500799	0.2501095
100	4.00034	4.00021	0.2499735	0.2499906
110	4.00017	4.00053	0.2501800	0.2501697
120	4.00063	4.00026	0.2489531	0.2489833
130	4.00035	4.00037	0.2498267	0.2498205
140	4.00062	4.00017	0.2489241	0.2489481
150	4.00142	4.00139	0.2509737	0.2509643
160	5.28588	5.73735	0.1287796	0.1422986
170	1.15689	1.26680	0.0000489	0.0000489
180	1.82445	1.64375	0.0000076	0.0000085

Run 9				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.64356	1.25400	0.0000101	0.0000086
20	1.36235	1.27886	0.0000230	0.0000245
30	4.58438	4.92087	0.0740651	0.0591742
40	4.00021	4.00065	0.2524829	0.2524827

50	4.00200	4.00109	0.2526386	0.2526591
60	4.00087	4.00062	0.2496921	0.2497121
70	4.00039	4.00050	0.2476925	0.2476963
80	4.00021	4.00093	0.2471467	0.2471107
90	4.00056	4.00056	0.2516958	0.2516954
100	4.00039	4.00009	0.2493737	0.2493958
110	4.00106	4.00077	0.2502074	0.2502327
120	4.00081	4.00068	0.2498474	0.2498338
130	4.00064	4.00027	0.2502608	0.2502325
140	4.00053	4.00026	0.2263656	0.2263480
150	5.09421	5.42001	0.1909998	0.2062231
160	5.90597	5.89919	0.1005977	0.1005972
170	1.14855	1.54472	0.0000155	0.0000122
180	1.02155	1.26580	0.0000057	0.0000015

Run 10				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.65667	1.50000	0.0000030	0.0000050
20	1.75891	1.36792	0.0000412	0.0000366
30	4.94535	5.34833	0.1130048	0.1010347
40	4.00173	4.05119	0.2619850	0.2588010
50	4.00021	4.00023	0.2479096	0.2479124
60	4.00031	4.00050	0.2520727	0.2521002
70	4.00024	4.00025	0.2497287	0.2497206
80	4.00044	4.00083	0.2497364	0.2497135
90	4.00068	4.00085	0.2503270	0.2503348
100	4.00057	4.00054	0.2511225	0.2511075
110	4.00052	4.00071	0.2486237	0.2486098
120	4.00075	4.00088	0.2512341	0.2512054
130	4.00079	4.00003	0.2502741	0.2503131
140	4.00043	4.00052	0.2502539	0.2502326
150	4.63616	4.50409	0.2122708	0.2225458
160	5.25411	4.99852	0.1150051	0.1503286
170	1.14588	1.64920	0.0000086	0.0000110
180	1.77778	1.25466	0.0000043	0.0000076

Results from runs with ECMP and Stride(30) communication pattern

Run 1				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.77778	1.79522	0.0000042	0.0000075
20	1.60322	1.58831	0.0000187	0.0000188
30	5.56756	5.56756	0.0023196	0.0023196
40	4.92674	5.51235	0.2023630	0.1808015
50	4.00152	4.02927	0.2498727	0.2481550
60	4.00091	4.00029	0.2481316	0.2481215
70	4.00090	4.00045	0.2489063	0.2489153
80	4.00078	4.00025	0.2488746	0.2488814
90	4.00025	4.00031	0.2358567	0.2358227
100	4.00040	4.00043	0.2695702	0.2695998
110	4.00011	4.00066	0.2500594	0.2500480
120	4.00007	4.00033	0.2509610	0.2509462
130	4.00061	4.00036	0.2501868	0.2501984
140	4.00066	4.00064	0.2499204	0.2499519
150	4.42632	4.30158	0.2259001	0.2324694
160	5.14694	5.02839	0.0889701	0.0906174
170	2.30518	2.98255	0.0000088	0.0000068
180	1.86958	1.90377	0.0000077	0.0000048

Run 2				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.75006	1.47809	0.0000261	0.0000251
20	1.74138	1.43009	0.0000116	0.0000141
30	5.88505	5.98180	0.0457478	0.0396493
40	5.34030	5.96608	0.1903290	0.1704032
50	4.00051	4.03975	0.2494784	0.2470495
60	4.00032	4.00031	0.2416239	0.2416198
70	4.00032	4.00031	0.2416239	0.2416198
80	4.00026	4.00019	0.2503147	0.2502933
90	4.00027	4.00013	0.2481306	0.2481473

100	4.00046	4.00034	0.2501943	0.2501939
110	4.00062	4.00041	0.2495162	0.2495386
120	4.00041	4.00048	0.2507930	0.2507770
130	4.00070	4.00008	0.2501086	0.2501290
140	4.00016	4.00028	0.2500631	0.2500483
150	4.00036	4.00065	0.2513446	0.2513365
160	5.60999	5.59750	0.0861425	0.0890628
170	1.79352	1.72555	0.0000281	0.0000276
180	1.72740	1.22100	0.0000093	0.0000083

Run 3				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.15000	1.20000	0.0000040	0.0000040
20	1.46261	1.28134	0.0000417	0.0000476
30	5.82963	6.45241	0.0684213	0.0588748
40	4.55941	4.59372	0.2197398	0.2180343
50	4.00414	4.08808	0.2496990	0.2445423
60	4.00117	4.00068	0.2478431	0.2478290
70	4.00039	4.00079	0.2483969	0.2483727
80	4.00036	4.00024	0.2530777	0.2530629
90	4.00027	3.99991	0.2496477	0.2496356
100	4.00048	4.00078	0.2501391	0.2500964
110	4.00041	4.00050	0.2502924	0.2503004
120	4.00039	4.00032	0.2498215	0.2498330
130	4.00044	4.00039	0.2502754	0.2502757
140	4.00067	4.00067	0.2503542	0.2503483
150	4.05451	4.00446	0.2504678	0.2503003
160	5.65556	5.96648	0.1034311	0.1001893
170	2.82469	2.80485	0.0000433	0.0000433
180	1.00000	1.00000	0.0000080	0.0000080

Run 4				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.18738	1.21978	0.0000063	0.0000091
20	1.97354	1.98009	0.0000974	0.0000957

30	6.05973	6.54241	0.0334048	0.0285537
40	4.22111	4.53488	0.2397875	0.2232340
50	4.00109	4.00075	0.2450997	0.2450707
60	4.00067	4.00018	0.2490216	0.2490529
70	4.00034	4.00013	0.2498803	0.2498695
80	4.00017	4.00022	0.2492698	0.2492716
90	4.00025	3.99996	0.2507474	0.2507662
100	4.00145	4.00143	0.2489681	0.2489998
110	4.00078	4.00073	0.2516837	0.2516900
120	4.00023	4.00013	0.2494359	0.2494473
130	4.00044	4.00050	0.2507790	0.2507710
140	4.00024	4.00039	0.2584070	0.2584099
150	4.78866	4.05212	0.2133143	0.2236762
160	5.19780	5.04192	0.0896508	0.0909233
170	2.27046	2.55835	0.0000512	0.0000512
180	1.55421	1.84371	0.0000083	0.0000094

Run 5				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.88235	1.60936	0.0000091	0.0000107
20	1.27223	1.65444	0.0000160	0.0000127
30	5.67204	6.39265	0.0704009	0.0597081
40	4.15580	4.22726	0.2407346	0.2367183
50	4.00014	4.00000	0.2471578	0.2471597
60	4.00021	4.00202	0.2471578	0.2471597
70	4.00038	4.00023	0.2499924	0.2500231
80	4.00059	4.00012	0.2501895	0.2501789
90	4.00022	4.00017	0.2515993	0.2515854
100	4.00061	4.00026	0.2485370	0.2485161
110	4.00037	4.00022	0.2537840	0.2537954
120	4.00028	4.00001	0.2497701	0.2497639
130	4.00022	4.00034	0.2503372	0.2503255
140	4.00033	4.00054	0.2513083	0.2513176
150	4.72470	4.88154	0.2135616	0.2175013
160	5.85799	5.40070	0.0630632	0.0630631
170	2.73041	2.02734	0.0000447	0.0000451
180	1.79242	1.81716	0.0000099	0.0000096

Run 6				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.77707	1.70074	0.0000030	0.0000050
20	1.11665	1.21081	0.0000251	0.0000231
30	5.85144	5.83325	0.0104574	0.0104585
40	4.44271	4.89474	0.2000054	0.2201460
50	4.76937	4.12118	0.2595910	0.2515131
60	4.34714	4.99605	0.2857251	0.2656457
70	4.07535	4.17458	0.2453315	0.2394997
80	4.00052	4.00042	0.2496763	0.2496835
90	4.00013	4.00018	0.2517186	0.2517239
100	4.00087	4.00060	0.2482977	0.2483127
110	4.00058	4.00107	0.2499729	0.2499181
120	4.00024	4.00027	0.2500795	0.2501094
130	4.00014	4.00037	0.2499340	0.2499331
140	4.00042	4.00044	0.2517088	0.2516910
150	4.00043	4.00010	0.2499187	0.2499416
160	5.00044	5.00060	0.0481983	0.0481891
170	2.12950	2.01151	0.0000888	0.0000561
180	1.48664	1.27046	0.0000022	0.0000044

Run 7				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.53881	1.00000	0.0000111	0.0000085
20	1.25116	1.25116	0.0000162	0.0000162
30	5.77219	6.26946	0.0506971	0.0421280
40	5.23103	5.80462	0.1950910	0.1758184
50	4.00032	4.02912	0.2496527	0.2479077
60	4.00118	4.00078	0.2501423	0.2501146
70	4.00017	4.00056	0.2436928	0.2436912
80	4.00045	4.00078	0.2497436	0.2497257
90	4.00016	4.00021	0.2500977	0.2500716
100	4.00078	4.00045	0.2507583	0.2508012
110	4.00034	4.00042	0.2501978	0.2501644
120	4.00064	4.00091	0.2507150	0.2507207

130	4.00021	4.00030	0.2502835	0.2502873
140	4.00026	4.00036	0.2500780	0.2500889
150	4.01171	4.00052	0.2510855	0.2517948
160	4.40888	4.23784	0.0392165	0.0392165
170	2.90708	2.54646	0.0000851	0.0000859
180	1.90244	1.84111	0.0000099	0.0000084

Run 8				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.82418	1.82418	0.0000091	0.0000091
20	1.29073	1.21321	0.0000157	0.0000166
30	6.00007	6.00007	0.0707303	0.0707303
40	5.51193	6.46596	0.1246101	0.1039126
50	4.91995	5.32844	0.2034402	0.1878463
60	4.00044	4.05255	0.2499098	0.2467299
70	4.00037	4.00027	0.2501716	0.2501592
80	4.00048	4.00062	0.2471286	0.2471732
90	3.99996	4.00014	0.2486437	0.2486375
100	4.00040	4.00015	0.2514952	0.2514939
110	4.00014	4.00011	0.2510066	0.2509817
120	4.00030	4.00062	0.2481499	0.2481193
130	4.00028	4.00148	0.2498734	0.2497934
140	4.00035	4.00050	0.2502212	0.2502296
150	4.00084	4.00072	0.2499290	0.2499603
160	5.22593	5.00620	0.0920584	0.0505051
170	1.39911	1.85100	0.0000374	0.0000699
180	1.09044	1.96545	0.0000069	0.0000069

Run 9				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.39467	1.31330	0.0000284	0.0000233
20	1.65179	1.75465	0.0000143	0.0000135
30	5.73399	6.51434	0.0636975	0.0538177
40	4.34333	4.40964	0.2309923	0.2274970
50	4.00056	4.00047	0.2490723	0.2490855
60	4.00056	4.00047	0.2490723	0.2490855

70	4.00047	4.00036	0.2502389	0.2502555
80	4.00021	4.00026	0.2499847	0.2499553
90	4.00023	4.00029	0.2509032	0.2508907
100	4.00020	4.00062	0.2498411	0.2498382
110	4.00042	4.00027	0.2532282	0.2532455
120	4.00011	4.00051	0.2498729	0.2498533
130	4.00053	4.00072	0.2500622	0.2500393
140	4.00026	4.00036	0.2503260	0.2503001
150	4.53607	4.76812	0.1949078	0.1924269
160	5.17552	5.07113	0.0820684	0.0861663
170	2.84929	2.20801	0.0000482	0.0000819
180	1.82987	1.83261	0.0000777	0.0000775

Run 10				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.82485	1.82485	0.0000092	0.0000092
20	2.02080	2.24330	0.0000168	0.0000152
30	5.67874	6.50927	0.0657381	0.0553171
40	4.37080	4.48660	0.1788431	0.1742177
50	4.00019	4.00017	0.2480082	0.2480219
60	4.00019	4.00017	0.2480082	0.2480219
70	4.00043	4.00060	0.2493113	0.2493109
80	4.00065	4.00054	0.2511924	0.2511951
90	4.00074	4.00074	0.2508370	0.2508382
100	4.00077	4.00042	0.2500905	0.2501055
110	4.00086	4.00034	0.2505574	0.2505856
120	4.00022	4.00055	0.2499120	0.2498877
130	4.00049	4.00028	0.2498620	0.2499036
140	4.00081	4.00059	0.2505218	0.2505645
150	4.23457	4.19159	0.1977150	0.1981216
160	5.53970	5.17642	0.0738402	0.0751043
170	1.51617	1.46814	0.0000803	0.0000803
180	1.82278	1.81150	0.0000089	0.0000084

Results from runs with Hedera and Random communication pattern

Run 1				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.82383	1.56584	0.0000121	0.0000141
20	1.24619	1.39853	0.0000229	0.0000205
30	5.78505	5.48180	0.0447478	0.0396000
40	4.69382	5.05633	0.1470476	0.1333528
50	4.00048	4.00802	0.2523957	0.2518930
60	2.58393	2.51632	0.3774694	0.3871352
70	2.66613	2.66715	0.3816009	0.3814684
80	2.66598	2.66727	0.3797211	0.3796746
90	2.66643	2.66649	0.3749481	0.3749365
100	2.66641	2.66684	0.3750386	0.3750489
110	2.66682	2.66639	0.3749616	0.3749576
120	2.67010	2.66985	0.3718704	0.3718353
130	2.66774	2.66753	0.4037132	0.4036773
140	2.66604	2.66712	0.3749101	0.3749319
150	2.66698	2.66664	0.3727294	0.3727062
160	3.05347	2.90780	0.0566951	0.0591652
170	1.84460	1.82418	0.0000086	0.0000091
180	1.11000	1.21500	0.0000022	0.0000044

Run 2				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.82614	1.82614	0.0000094	0.0000094
20	1.16418	1.12230	0.0000268	0.0000278
30	5.99148	5.00011	0.0904635	0.0804630
40	4.55837	4.84101	0.1809036	0.1672123
50	4.00012	4.00037	0.2506137	0.2506333
60	2.23899	1.76282	0.3675593	0.3795435
70	2.02956	1.80262	0.4427653	0.4427648
80	2.02956	1.80262	0.4427653	0.4427648
90	2.02866	2.02900	0.4439472	0.4439576
100	2.02778	2.02777	0.4414782	0.4414787
110	2.28983	2.22605	0.4385010	0.4384901
120	2.28789	2.21898	0.4365171	0.4365274

130	2.28795	2.22765	0.5011506	0.5011765
140	2.28929	2.22608	0.4372003	0.4372120
150	2.29291	2.22705	0.4282989	0.4283096
160	2.54201	2.76350	0.0456287	0.0456292
170	1.54623	1.12400	0.0000478	0.0000774
180	1.00000	1.00000	0.0000086	0.0000086

Run 3				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.82418	1.82418	0.0000091	0.0000091
20	1.27852	1.27852	0.0000243	0.0000243
30	6.14723	6.14723	0.0400302	0.0400302
40	4.85568	5.27613	0.1240250	0.1106683
50	4.00222	4.05289	0.2489802	0.2459009
60	2.66865	2.55860	0.2962137	0.3050050
70	2.71168	2.71218	0.5078374	0.5077431
80	2.72899	2.72926	0.3644971	0.3644561
90	2.71095	2.71173	0.3643817	0.3643469
100	2.71266	2.71250	0.3794325	0.3794587
110	2.71248	2.71208	0.3689016	0.3689598
120	2.71545	2.71607	0.3693536	0.3693461
130	2.71161	2.71181	0.3684421	0.3684183
140	2.71153	2.71162	0.3692516	0.3692952
150	2.71273	2.71288	0.3684461	0.3684593
160	3.27498	2.99058	0.0778060	0.0802646
170	1.24890	1.24890	0.0000283	0.0000283
180	1.00001	1.96873	0.0000086	0.0000092

Run 4				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.83391	1.73732	0.0000091	0.0000096
20	1.27918	1.39421	0.0000245	0.0000225
30	4.99193	5.49604	0.0925460	0.0812461
40	4.03235	4.09850	0.2098178	0.2064363
50	2.55017	2.48002	0.3043260	0.3143494
60	2.66711	2.66745	0.3689105	0.3688648
70	2.66711	2.66745	0.3689105	0.3688648

80	2.66575	2.66645	0.3752578	0.3752948
90	2.66511	2.66638	0.3745262	0.3745463
100	2.66678	2.66690	0.3747175	0.3747022
110	2.66696	2.66721	0.3637588	0.3636597
120	2.66584	2.66592	0.3745373	0.3745251
130	2.66687	2.66669	0.3730811	0.3730598
140	2.66713	2.66696	0.3774075	0.3774190
150	3.08117	2.90002	0.3181576	0.3208077
160	3.00150	2.98540	0.0940299	0.0754299
170	1.55140	1.84143	0.0000086	0.0000098
180	1.15530	1.65140	0.0000005	0.0000005

Run 5				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.82418	1.72917	0.0000091	0.0000096
20	1.27869	1.33333	0.0000244	0.0000234
30	6.38230	6.67041	0.0883697	0.0780103
40	4.28551	4.54767	0.2340753	0.2206433
50	1.71288	1.72149	0.4173234	0.4290754
60	1.85522	1.85559	0.4272210	0.4272316
70	1.85522	1.85559	0.4272210	0.4272316
80	1.85468	1.85500	0.4339887	0.4340120
90	1.85549	1.85512	0.4304843	0.4304755
100	1.85560	1.85554	0.4332420	0.4332514
110	1.85521	1.85486	0.4278062	0.4277955
120	1.85763	1.85763	0.4334592	0.4334598
130	1.85456	1.85449	0.4255842	0.4256056
140	1.85521	1.85520	0.4264222	0.4264216
150	1.85500	1.85500	0.4355796	0.4355796
160	2.98755	3.03810	0.0490171	0.0640171
170	1.26490	1.24890	0.0000254	0.0000276
180	1.00010	1.00200	0.0000078	0.0000078

Run 6				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.73578	1.64911	0.0000130	0.0000136
20	1.16418	1.33333	0.0000268	0.0000234

30	5.25723	5.68857	0.0688341	0.0608420
40	4.15769	4.74510	0.2104140	0.2465410
50	2.27629	2.23420	0.3718139	0.3836324
60	2.39948	2.39953	0.3746954	0.3746827
70	2.40060	2.40113	0.3696531	0.3696425
80	2.39892	2.39880	0.3748026	0.3748247
90	2.40034	2.39991	0.3745388	0.3745300
100	2.24952	2.24959	0.4006320	0.4006211
110	2.24472	2.24427	0.3999472	0.3999454
120	2.25027	2.25024	0.3989654	0.3989669
130	2.25023	2.24998	0.4004123	0.4003801
140	2.24996	2.24951	0.3973870	0.3973978
150	2.25046	2.25145	0.4036301	0.4036083
160	5.51037	4.87124	0.0462088	0.0462214
170	1.84650	1.49730	0.0000444	0.0000783
180	1.00010	1.00070	0.0000079	0.0000079

Run 7				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.69667	1.64356	0.0000101	0.0000101
20	1.39286	1.33333	0.0000224	0.0000234
30	5.55984	5.54932	0.0484068	0.0484063
40	4.05673	4.15183	0.2439279	0.2382805
50	2.46519	2.59169	0.3863908	0.3902411
60	2.21810	2.30689	0.4312501	0.4313388
70	2.21810	2.30689	0.4312501	0.4313388
80	2.14033	2.22300	0.4501228	0.4499869
90	2.13929	2.22240	0.4480600	0.4481952
100	2.20419	2.28692	0.4277891	0.4276136
110	2.19803	2.28225	0.4496368	0.4494486
120	2.19556	2.28595	0.4335935	0.4334709
130	2.19488	2.28585	0.4443004	0.4444710
140	2.20348	2.28620	0.4371480	0.4370935
150	2.20487	2.28661	0.4410027	0.4411506
160	4.69755	4.59307	0.0461024	0.0461024
170	1.94323	1.94323	0.0000943	0.0000943
180	1.02150	1.82418	0.0000086	0.0000091

Run 8				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.83391	1.83391	0.0000091	0.0000091
20	1.18406	1.25301	0.0000263	0.0000249
30	5.22604	5.73664	0.0787525	0.0689240
40	4.06437	4.13771	0.2462633	0.2419114
50	2.05622	1.80686	0.3292190	0.3411348
60	2.25206	2.00422	0.3922612	0.3922824
70	2.25206	2.00422	0.3922612	0.3922824
80	2.11701	2.11649	0.4246414	0.4246640
90	2.11995	2.11693	0.4252666	0.4252661
100	2.35351	2.11788	0.4263619	0.4263622
110	2.34909	2.11459	0.4227649	0.4227659
120	2.35240	2.11841	0.5483887	0.5483736
130	2.35314	2.11776	0.4262854	0.4263075
140	2.35192	2.11695	0.4249616	0.4249504
150	2.34398	2.31011	0.3727515	0.3727515
160	4.21579	4.79531	0.0795200	0.0545470
170	1.91500	1.95918	0.0000486	0.0000492
180	1.51032	1.21650	0.0000092	0.0000092

Run 9				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.83333	1.78334	0.0000090	0.0000085
20	1.27886	1.25325	0.0000245	0.0000250
30	5.99988	5.99988	0.0904293	0.0904293
40	4.00023	4.00064	0.2526395	0.2526597
50	2.32557	2.19463	0.3943020	0.4082995
60	2.45954	2.46242	0.4038227	0.4037900
70	2.45954	2.46242	0.4038227	0.4037900
80	2.46100	2.46205	0.4063208	0.4063860
90	2.46100	2.46110	0.4045587	0.4045484
100	2.46123	2.46144	0.4071376	0.4071048
110	2.44888	2.44930	0.4074882	0.4074887
120	2.46165	2.46198	0.4056902	0.4057128
130	2.46093	2.46167	0.4070353	0.4070353
140	2.46117	2.46120	0.4057679	0.4057575

150	3.35885	2.82403	0.2247845	0.2251214
160	4.76950	3.91460	0.0991440	0.0945400
170	2.64110	2.24647	0.0000570	0.0000550
180	1.54400	1.59710	0.0000099	0.0000076

Run 10				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.40000	1.39130	0.0000129	0.0000148
20	1.59591	1.52195	0.0000196	0.0000205
30	5.53503	5.50558	0.0603823	0.0403838
40	5.14520	5.64149	0.1767922	0.1672581
50	2.35135	2.34291	0.3726070	0.3819568
60	2.57280	2.66663	0.3708961	0.3708645
70	2.57280	2.66663	0.3708961	0.3708645
80	2.80597	2.78883	0.3528951	0.3550677
90	2.80819	2.80595	0.3609553	0.3612191
100	2.80778	2.80795	0.3563048	0.3562990
110	2.80702	2.80776	0.3559701	0.3558766
120	2.80737	2.80626	0.3556814	0.3558166
130	2.80716	2.80698	0.3568621	0.3568885
140	2.80671	2.80759	0.3562717	0.3561601
150	2.80756	2.80763	0.3572671	0.3572581
160	5.77697	5.29090	0.0481553	0.0510676
170	1.54481	1.54481	0.0000579	0.0000579
180	1.00640	1.42942	0.0000086	0.0000127

Results from runs with Hedera and Stride(30) communication pattern

Run 1				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.00000	1.74492	0.0000084	0.0000101
20	1.87004	1.88782	0.0000210	0.0000201
30	5.76165	6.51585	0.0753322	0.0636385
40	3.18234	3.24836	0.2720597	0.2665568
50	3.07825	3.07803	0.3197550	0.3197852
60	3.07825	3.07803	0.3197550	0.3197852
70	2.91102	2.91085	0.3436836	0.3437334
80	2.91091	2.91069	0.3418241	0.3418680
90	2.38817	2.38798	0.4077877	0.4078297
100	2.38881	2.38864	0.4305479	0.4305687
110	2.01311	2.10518	0.5545500	0.5546562
120	2.00883	2.10571	0.4786598	0.4784216
130	1.43673	1.64153	0.4777715	0.4777711
140	1.43536	1.64079	0.4972978	0.4972982
150	3.34382	3.52333	0.1682770	0.1682770
160	2.92511	3.85067	0.0829945	0.0829945
170	1.35568	1.35360	0.0000169	0.0000252
180	1.48466	1.08239	0.0000327	0.0000323

Run 2				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.76129	1.00000	0.0000116	0.0000087
20	1.35948	1.51825	0.0000230	0.0000205
30	5.60139	5.45312	0.0101008	0.0101021
40	4.65561	5.20672	0.2705535	0.2594151
50	3.42939	3.57714	0.3468817	0.3325326
60	3.26356	3.26364	0.3023421	0.3023385
70	3.26356	3.26364	0.3023421	0.3023385
80	2.75992	2.76006	0.3620573	0.3620900
90	2.76048	2.76053	0.3638218	0.3638628
100	2.37145	2.37129	0.4219339	0.4218923
110	2.36332	2.36274	0.4145095	0.4145905
120	2.07765	2.07770	0.5287690	0.5289061
130	2.07800	2.07793	0.4824926	0.4824483

140	1.82431	1.99307	0.4874265	0.4962788
150	1.84574	1.84583	0.4924922	0.4924585
160	4.12814	3.90844	0.0564110	0.0564219
170	1.26353	1.05512	0.0000714	0.0000544
180	1.29573	1.01648	0.0000081	0.0000076

Run 3				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.53815	1.64126	0.0000123	0.0000112
20	1.39266	1.33333	0.0000225	0.0000235
30	5.91738	6.62054	0.0369628	0.0309226
40	3.33273	3.38047	0.3122574	0.3078067
50	3.13851	3.13812	0.3184160	0.3184430
60	3.13535	3.13539	0.3138632	0.3138633
70	2.50219	2.50227	0.3993700	0.3993906
80	2.50015	2.49995	0.3986286	0.3986125
90	2.18572	2.18574	0.4578229	0.4579409
100	2.18786	2.18855	0.4531763	0.4530425
110	1.64172	1.64632	0.4871018	0.4871018
120	1.64156	1.64209	0.4869745	0.4869966
130	1.64281	1.64270	0.4858392	0.4857615
140	1.64070	1.64059	0.4923803	0.4924019
150	3.57078	3.47054	0.1679728	0.1679781
160	4.59508	3.46893	0.0203646	0.0303532
170	1.52931	1.23089	0.0000325	0.0000434
180	1.63324	1.67232	0.0000055	0.0000540

Run 4				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.82485	1.00000	0.0000092	0.0000087
20	1.52289	1.59695	0.0000206	0.0000197
30	5.63812	6.44601	0.0699935	0.0586755
40	3.38080	3.43100	0.3024622	0.2979829
50	3.13539	3.13528	0.3157378	0.3157462
60	3.13539	3.13528	0.3157378	0.3157462
70	2.96540	2.96514	0.3371907	0.3373259
80	2.96412	2.96401	0.3373075	0.3373608

90	2.46406	2.46450	0.4068490	0.4068514
100	2.46289	2.46332	0.4035415	0.4035763
110	2.16234	2.14054	0.4647235	0.4694816
120	2.16151	2.16002	0.4608139	0.4611205
130	2.03660	2.05363	0.4832980	0.4888941
140	1.62122	1.62085	0.4956169	0.4956165
150	4.08207	3.49874	0.1714001	0.1714024
160	3.97614	3.21879	0.0270702	0.0270703
170	1.55129	1.54936	0.0000805	0.0000801
180	1.00981	1.00223	0.0000027	0.0000032

Run 5				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.64356	1.72917	0.0000101	0.0000096
20	1.28016	1.23016	0.0000243	0.0000253
30	5.72731	6.35592	0.0560513	0.0472233
40	5.17845	5.82855	0.1944112	0.1727157
50	2.92988	3.02152	0.3468070	0.3363871
60	2.85824	2.86007	0.3503583	0.3500677
70	2.85810	2.85786	0.3493160	0.3493336
80	2.85921	2.85886	0.3505580	0.3505582
90	2.85745	2.85755	0.3453890	0.3454068
100	2.46180	2.46172	0.4069448	0.4070004
110	2.45806	2.45787	0.4047649	0.4047860
120	2.13369	2.10609	0.4690762	0.4751476
130	2.13321	2.12792	0.4684363	0.4696078
140	2.05754	1.92749	0.4794018	0.4831783
150	1.85652	1.84829	0.4404277	0.4404387
160	3.96927	4.23325	0.1044105	0.1044105
170	1.68331	1.97817	0.0000638	0.0000782
180	1.52914	1.52914	0.0000580	0.0000580

Run 6				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.14341	1.26429	0.0000256	0.0000232
20	1.87952	1.87952	0.0000166	0.0000166
30	5.62636	6.45777	0.0762295	0.0638502

40	3.55289	3.82601	0.2857802	0.2653307
50	3.13797	3.13778	0.3176037	0.3175665
60	2.71930	2.71955	0.3694396	0.3694160
70	2.46352	2.46351	0.4053724	0.4053846
80	2.46155	2.46160	0.4039347	0.4039270
90	2.13282	2.11684	0.4685959	0.4721390
100	2.13269	2.13399	0.4678601	0.4676364
110	1.64136	1.64144	0.4874985	0.4874769
120	1.64219	1.64202	0.4870841	0.4870733
130	1.64081	1.64124	0.4879282	0.4879056
140	1.64008	1.64014	0.4876638	0.4876634
150	3.83907	3.54977	0.1687893	0.1687905
160	3.14961	4.57033	0.0298731	0.0298731
170	1.79332	1.80422	0.0000148	0.0000261
180	1.63597	1.62900	0.0000014	0.0000033

Run 7				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.56604	1.82418	0.0000106	0.0000091
20	1.20930	1.20696	0.0000258	0.0000258
30	5.51795	6.34074	0.1419557	0.1193817
40	4.87526	5.21767	0.2088853	0.1951848
50	3.17366	3.17308	0.3149107	0.3149260
60	3.07809	3.07805	0.3240536	0.3240713
70	3.07809	3.07805	0.3240536	0.3240713
80	2.85837	2.85851	0.3494017	0.3493898
90	2.85721	2.85782	0.3497366	0.3497753
100	2.42468	2.42448	0.4185502	0.4185540
110	2.42460	2.42444	0.4090277	0.4090430
120	1.84952	2.04723	0.4943129	0.4959792
130	1.84778	2.05169	0.4880987	0.4880914
140	1.62216	1.80332	0.4874843	0.4909975
150	2.65741	2.65816	0.3029745	0.3029636
160	3.98997	4.75011	0.0843773	0.0843881
170	1.23234	1.47255	0.0000124	0.0000204
180	1.12001	1.12001	0.0000012	0.0000012

Run 8				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.64424	1.72997	0.0000120	0.0000114
20	1.33226	1.27786	0.0000233	0.0000243
30	5.62208	6.29073	0.0732795	0.0625720
40	3.47033	3.54917	0.3010653	0.2943073
50	3.13785	3.14564	0.3189632	0.3181334
60	3.13745	3.13750	0.3164476	0.3165005
70	2.85789	2.85701	0.3428590	0.3428645
80	2.85971	2.86033	0.3578272	0.3578262
90	2.50210	2.50207	0.3994720	0.3994979
100	2.49654	2.49649	0.3997349	0.3997542
110	2.13404	2.12670	0.4687227	0.4704261
120	2.13287	2.13348	0.4693364	0.4692887
130	1.98068	1.90781	0.4810954	0.4893872
140	1.64091	1.64133	0.4982765	0.4982866
150	4.54694	4.31543	0.1674136	0.1674249
160	4.45481	3.88520	0.1230321	0.1230322
170	1.48266	1.48253	0.0000462	0.0000467
180	1.41875	1.51539	0.0000047	0.0000064

Run 9				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.06127	1.28487	0.0000159	0.0000165
20	1.83846	1.74303	0.0000185	0.0000195
30	5.66748	6.29705	0.0824595	0.0699876
40	3.26576	3.32771	0.3079898	0.3022811
50	3.20011	3.20030	0.3103341	0.3103393
60	3.20011	3.20030	0.3103341	0.3103393
70	2.62416	2.62405	0.3807620	0.3807653
80	2.61730	2.61745	0.3795603	0.3795972
90	2.24339	2.24464	0.4473419	0.4470781
100	2.23517	2.25240	0.4455708	0.4422168
110	1.88223	1.88346	0.4875522	0.4888549
120	1.84386	1.84595	0.4883676	0.4883596
130	1.71492	1.71805	0.4870234	0.4864268

140	1.84404	1.84484	0.4923337	0.4923114
150	4.26544	3.90142	0.1538566	0.1538590
160	4.03246	3.74510	0.1144805	0.1054200
170	1.51480	1.74160	0.0000754	0.0000441
180	1.10663	1.14741	0.0000260	0.0000051

Run 10				
Time(seconds)	CLLBL RX	CLLBL TX	BB RX	BB TX
10	1.72917	1.64620	0.0000096	0.0000086
20	1.36245	1.36245	0.0000229	0.0000229
30	5.48082	5.94682	0.1520206	0.1344658
40	4.65924	4.73925	0.2148238	0.2111315
50	3.11996	3.12524	0.3231258	0.3226072
60	3.02096	3.02067	0.3281834	0.3281703
70	3.02096	3.02067	0.3281834	0.3281703
80	2.85895	2.85877	0.3507760	0.3507624
90	2.85716	2.85707	0.3500206	0.3500620
100	2.42710	2.42735	0.4142948	0.4143054
110	2.42271	2.42261	0.4038014	0.4037953
120	2.07824	2.07786	0.5371047	0.5372036
130	2.07836	2.07817	0.4885456	0.4885091
140	1.64143	1.64141	0.4829109	0.4829211
150	2.42181	2.74004	0.2913500	0.2913395
160	3.93347	3.02910	0.0568950	0.0568955
170	1.91310	1.91310	0.0000748	0.0000748
180	1.00006	1.00006	0.0000046	0.0000046