IMPROVING DATA-SHUFFLE PERFORMANCE IN DATA-PARALLEL DISTRIBUTED SYSTEMS

BY

SHWEELAN SAMSON

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved ____________________________
Advisor

Chicago, Illinois
July 2018

ACKNOWLEDGMENT

I would like to show my appreciation to my advisor Professor Dr. Boris Glavic for his supportive pieces of advice and vast knowledge throughout my journey at Illinois Institute of Technology.

Also, I would like to thank Professor Dr. Ioan Raicu for serving on my thesis defense committee. Thanks for the enjoyable discussion and the fantastic suggestions.

Also, I would like to express my special appreciation and many thanks to the Fulbright Foreign Student Program for presenting this opportunity to achieve my Masters of Science in Computer Science at IIT.

In the end, I would like to share this success with my family and friends and thank them for showing me the most prominent support and encouragement to achieve my goal.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Scaling up to thousands of nodes in data-parallel systems like MapReduce is prevailing. Many logical network topologies like Torus, Binomial Graph, and Hypercubes have been proposed to solve network connections concurrency problem that comes along with this scale. These topologies enforce a limit on the number of the allowed concurrent network connections per node and structure an overlayed network topologies around it. Many implementations of these topologies set the connections' limit value depending on folklore guesses. One contribution of this work is to experimentally understand the effect of concurrency degree on the machine's resources and the whole network resources. We implemented a benchmark tool which was used to test different combinations of concurrency and buffer size parameters. We evaluated the results and showed how does the number of concurrent connections affects latencies, throughput, throughput stability, and compute resources consumption. Furthermore, we showed that buffering and messages chunk size are critical for performance and resources utilization. However, The scalability of these topologies comes at the cost of having potentially longer communication paths. With longer paths, more network bandwidth will be utilized in forwarding the messages through intermediate hops. We proposed a hybrid approach that enables data-parallel systems to deploy direct connections to the most congested paths selectively. We implemented a simulation for Binomial graph topology and deployed this hybrid approach. Experiments showed that with skewed datasets, the hybrid approach yielded a decent performance boost, and a good saving in the forwarded messages count, hence increasing bandwidth efficiency, and improving resources utilization.

CHAPTER 1
INTRODUCTION

The fast growth of the data generated by different sources, such as web services, mobile applications, sensors, the internet of things devices, etc, brought the necessity of the distributed systems clusters that are able to scale up to hundreds or thousands of nodes [1, 2] to be able to analyze and process that amount of data in an acceptable time frame. Thus, Data-parallel distributed systems frameworks such as MapReduce [3], Hadoop [4], Spark [5], and Dryad [6] have become widespread as they are designed to fulfill this expansion; they provide data distribution mechanisms, extreme parallelism, fault tolerance, and load balancing.

These systems partition their datasets across most or all of their worker nodes. In the real world, these data partitions are replicated over multiple nodes, so that these systems are more fault tolerant. Data processing tasks or queries are forced to be designed as small sub-tasks that can be processed in parallel and as independently as possible. Such systems are great for the end user since they abstract the parallelism complexity, and provide simpler interfaces for both the developers and the end-users. Moreover, these systems keep their nodes utilizations high. Using load balancing techniques, they are able to distribute the workload over the cluster as evenly as possible. Query execution in data-parallel systems is frequently interspersed with data transfer. Workers nodes transfer considerable amounts of data during their computations. These platforms handle data communications during query execution.

These features made it easy for data-parallel platforms to scale up to a large number of nodes and abstract the program parallelizing and dataset distribution from the end-user. The success of data-parallel systems is driven by their ability to scale to thousands of nodes and become very powerful at a lower cost, where hardware scalability is expensive.

1.1 Data-parallel Systems

With the fast pace of the data growth, investing in data parallelism became a necessity to achieve scalability. Data-parallel systems abstract the concurrency control dilemma away from the developers. These systems make this possible by partitioning the data across their nodes and provides simple interfaces for the developers to build their software with data parallelism in mind. They also manage the resource allocation, job scheduling, inter-nodes data flow control, and nodes failures.

1.1.1 Data Partitioning. In general, such systems aim to partition datasets into smaller pieces of data in a way that can process each chunk of these data parts in parallel on different machines that may belong to different networks, or even located at different geographical locations. Systems like MapReduce [3], Hadoop [4], and Spark [5] make use of distributed file systems such as Google file system (GFS) [7] and Hadoop distributed file system (HDFS) [8, 9] to abstract the data partition away from the user. These frameworks typically enforce a programming model where the developer implements the task as a data-flow model, where the program's dataset is pushed through a set of operators (i.e., functions or subroutines) allowing these frameworks to schedule each sub-task (i.e., data chunk) on a different node and restart the sub-task if errors were encountered. On the other hand, systems such as HRDBMS [10], use the local filesystem to store data directly to their cluster nodes. They implement their own storage to have the full control over the data locality which is critical for both performance and scalability.

1.1.2 Locality Awareness. Usually, the network resources on commodity machines are scarce, expensive, and perform poorly (i.e., high latencies, low throughput) if the physical network is not configured correctly. Data-parallel systems try to cope with such a paucity of network resources. These systems are designed to feature locality-awareness on their data fractions; they keep track of where a data fraction resides, and try to schedule the task on the node where the data reside to keep network utilization as low as possible.

However, these systems analyze and process variant datasets to answer users query, these datasets are usually interrelated (Database records for example) at some applications, or at least queries results are somehow related (word count example, where each node counts the words within its partition, and then the results are aggregated from all the nodes), thus, the whole processing of the data fractions may not be completed independently in parallel on the corresponding local machines that hold them, at some point, these systems' nodes are ordinarily compelled to collaborate with each other to exchange data between them in order to complete the process and produce meaningful results.

1.1.3 Data Transfer.

Despite the attempts to process queries locally and in parallel, frequently, queries processing requires data to be transferred between nodes in order to proceed with the computations. There are many patterns of data transfer between the nodes. The most popular patterns which occur in roughly all the data-parallel systems are broadcast and shuffle. Data broadcast pattern follows the one-to-many communication pattern, where a node broadcasts the same message to multiple or all the other nodes. The coordinator nodes in HRDBMS [10] are responsible for transactions commit operations. A coordinator keeps track of the nodes involved in a transaction and performs a Hierarchical Two-Phase Commit Protocol. This operation involves the broadcasting operation to all the nodes involved. Data shuffle pattern follows the many-to-many communication pattern, where multiple or all nodes send different pieces of data to other nodes; possibly to all of the nodes. In MapReduce

[3], the data shuffle is introduced between the map and reduce phases. Map phase distributes the map function over multiple nodes to be processed locally; the map function generates a key-value pair for every data point. Afterward, the mappers sort the data based on the keys and transfer the data to the reducers. Reduce phase performs the collection and combination of its input to provide the final results. In HRDBMS, executing a query with a self-join operator on a table that is rows-based partitioned, the execution engine transfers records between the nodes at the run-time to generate the results, these transfers follows many-to-many pattern.

1.2 Problem And Motivation

Network communications can be a troublesome bottleneck. Data transfer patterns that involve sending data to multiple - possibly to all - nodes are intricate to well design and implement. Data broadcast and Data shuffle, are good examples. The most apparent obstacle is maintaining an enormous number of concurrent connections at the same time. Many solutions were proposed to cope with high degree connections concurrency problem.

In broadcast operations, many systems [10, 11] implements tree based overlayed topologies to limit the number of concurrent connections and parallelly get the tree lower level nodes to reroute the message the level-below nodes. In HRDBMS [10], the inherently hierarchical operators are implemented to use a tree topology, an example of such operator is the distributed merge sort, where each level of the tree represents and processes a single merge phase; the intermediate merge phases are done in parallel which provides a more load balanced operation. The Hierarchical Tow Phase Commit Protocol is also implemented on top of the tree topology. Dremel [11] also implements tree topology to execute aggregation queries, where the queries are passed down the tree to the leaves, servers on the leaves generate intermediate results and pass them up the tree, the intermediate servers aggregate the partial results in parallel.

On the other hand, shuffle operations are more complicated, where different data chunks are transferred to various destinations. This may require each node to communicate with virtually all the nodes on the cluster. Many studies [12, 13, 14, 15] introduced logical network topologies that tend to force a limit on the number of the concurrent connections a single machine can maintain at any moment, and building routing plans between the nodes as intermediate hops to deliver the data from source to destination. More topologies and examples fo such solutions will be discussed in Chapter 2.

The number of connections maintained on a single node is called “degree”, and the maximum number of intermediate hops between any source and destination is called “diameter”. The degree and diameter of a network topology are inversely proportional, that is a network with small degree will have larger diameter, such as the ring topology, where the degree is 2, where any node is only connected to the next and previous siblings, this will make the maximum diameter to be half the number of nodes on the network. Networks with longer diameter tend to encounter high latency problem, and physical network excessive utilization, as a consequence, the overall system throughput is affected since the physical network is utilized to resend repeated messages over and over. On the contrary, a fully connected graph maintain a degree that is equal to the number of nodes on the network. Thus each node will maintain a direct connection with every other node. Hence this topology diameter is 1. Machines that belong on a network with high degree suffer to maintain a huge number of concurrent connections.

The different implementations of the solutions proposed to solve the connections concurrency problem tend to enforce a limit on the number of the concurrent open connections (degree). The imposed limit is usually set based on folklore guesses and estimations for a decent degree. However, the number of concurrent open connections is critical to the system communications framework and affects multiple aspects of the system. Enforcing a limit on the number of network connections and build an overlayed network topologies around it comes at the cost of potentially longer communication routes. With longer routes, the messages will be retransferred multiple times through many hops to arrive at their destinations. With more messages retransferred means less effective throughput, and more cost. In [16], they show that the data transfer accounts for more than 50% of the computation time in data-parallel platforms. An improvement to these systems communications performance will hold an impact on the overall system performance. Moreover, it will have an impact cost wise; data transfer in cloud platforms is expensive [17].

In Chapter 2 we will demonstrate the related work, and how different studies proposed topologies to cope with the networking concurrency problems and difficulties that are faced by the data-shuffle operations. In Chapter 3, we will try to understand the real effect of the connections concurrency on the machines regarding the resources utilization and state management, i.e. (how would the operating system handle the massive number of concurrent network connections?). Furthermore, we will try to evaluate the effect on the whole network concerning latency and throughput. Afterward, in Chapter 4, we will discuss a hybrid approach that mixes between the use of direct connections and topologies like Binomial Graph to boost performance and reduce network traffic during data-shuffle operations. We will experimentally evaluate this hybrid approach and show its impact on performance and saving network traffic. In Chapter 6 we suggested more ideas and plans to improve the hybrid approach proposed in this work.

CHAPTER 2
RELATED WORK

Fully connected graph is optimal concerning latency, fault tolerance, diameter(route length), and the network utilization efficiency; no messages need to be forwarded. Unfortunately, it is not scalable. Ring topology is more scalable than a fully connected graph, but it introduces a much higher diameter. The community has been researching and trying to propose topologies that can find a decent balance between node degree and route diameter. Another essential feature for the communications framework graphs that is advantageous to maintain is to be a regular graph, that is all the nodes on the system keep the same degree. Network topologies like Hypercubes [13], Torus Interconnect [14, 15], and Binomial Graph [12] tried to achieve the best topological properties and find a decent balance between the network degree and diameter.

2.1 Network Logial Topologies

Hypercubes network, also called N-cube, maps the nodes as a graph of the cube shape. With n-cube, where n is the cube dimensions, n nodes are labeled upon the 2^n binary labels. For every node on the system, it will only connect to the nodes that hold the labels that are different by exactly 1 bit. This topology has a good degree and diameter that is $DE = DI = Log2(N)$ where DE is the degree, DI is the diameter, and N is the number of nodes on the network. However, this topology is restricted to the number of concurrent connections. Going more on the number of nodes will introduce longer routes easily. With 10-cube that can label 1024 nodes, the longest route is 10 hops away. Hypercubes has other variants ShufflNet [18], Folded Petersen cube Networks [19], De Bruijn Graph [20], and Kautz Graph [21]. Some of those are not scalable. Others introduce longer routes easily.

On Torus Interconnect networks, the nodes are laid out on a multidimensional lattice. In general, ND Torus have N dimensions, and each node will connect to the closest $2N$ nodes. Torus networks are complicated to implement, and they have higher average diameter.

Other k-ary tree based topologies such as Hypertree [22], and Hierarchical Clique [23] came up with good topological properties. They are scalable, fault tolerant, and maintain a decent low average diameter. However, the resulted graph is not regular, where some nodes have a higher degree that makes them more congested.

Binomial Graph can be imagined as a ring network topology but with a difference that any node is connected to $Next^i$ and $Previous^i$ nodes for $i = 1..L$ where L is the maximum-limit number of connections a node can maintain. Implementing such topology will allow the network to have a logarithmic diameter with a lower degree, i.e. $(DI = Log_L(N))$ where DI is the diameter and N is the number of nodes on the network.

Dryad [6] allows the developers to take control over the overlayed communication graph. A Dryad developer can specify a directed acyclic graph for Dryad to express the data flow and the communication pattern between the nodes on the system. This flexibility in the runtime communication pattern is providing good performance while compromising on programmability; the developer needs to know about the nature of the data and its distribution.

Introducing these network topologies solves the problem of the too many concurrent network connections. However, despite providing decent diameters, they presented the challenge of sending repeated messages through the network. A significant fraction of the messages will be sent to the destination using 1 or more intermediaries. The middlemen will reroute the message to its direct connections, repeatedly until the message arrives at its destination. The number of intermediaries depends on the network diameter, which depends on the number of the maximum concurrent connections allowed on the system and the overlayed network topology. Thus, the ultimate goal is always to have a diameter that is as low as possible, a diameter of 1 preferably since having a direct connection is the optimal route any data transfer can be sent over, but as explained above it is impossible for multi-thousands machines clusters. Peer-To-Peer topologies, such as Content-Addressable Network [24], Chord [25], Pastry [26], Tapestry [27], and Zero-Hop Distributed Hashtable(ZHT) [28] are scalable and fault tolerant, they also provide perfect average diameter, ZHT for example provide a diameter of 1, where all the connections are direct to the data source and are dynamically switched using LRU connections cache. These topologies were designed for dynamic application based on distributed hash tables. However, the overhead of their dynamic communication framework will affect communication efficiency.

However, in parallel database platforms [29] such as HRDBMS [10], the overlayed communication graph is implicit. The end user need know nothing about the communication framework; in an optimal world, the users must feel no difference between the traditional databases and the distributed ones.

2.2 HRDBMS

Highly-scalable Relational DBMS (HRDBMS) [10, 30] is a distributed shared-nothing database system that is designed to have per-node performance that is comparable to traditional relational databases or even to Massively Parallel Processing (MPP) databases, and the ability to scale as linearly as the databases such as Spark SQL [31], Dremel [11], Tenzing [32], and Hive [33], that are build on top of Big Data platforms such as Hadoop [4], Spark [5], and other implementations of MapReduce [3].

With bottlenecks in mind, where MPP databases are limited in scaling up to huge clusters, and Big Data platforms have per-node performance that is inadequate, HRDBMS mixed the techniques and ideas from the 2 systems, with the proper communication, and the well-designed compute level parallelism and data level distribution techniques to achieve its goals. HRDBMS execution engine is distributed and asynchronous by design, which supports it to feature highly-parallelized operators, non-blocking, hierarchical data shuffling, and locality-aware query execution.

In HRDBMS, data is stored on the workers' local storage. Data is partitioned across nodes based on two partitioning strategies, row-based, and columnar-based partitioning. The user selects the preferred strategy at the table creation time. By handling the data storage directly, the execution engine is able to control data locality perfectly. Moreover, HRDBMS implemented an external table framework to access external data sources, such as HDFS [9].

HRDBMS enforces scalable communication patterns. It implements the tree topology for the operators that are inherently hierarchical, such as the Two-Phase Commit Protocol (2PC), and the distributed merge sort operator. In this topology, every node only communicates with a parent, and it's direct children. This enforces each node to have a limit on the number of concurrent connections. The second utilized topology is Binomial Graph Topology which is applied for data shuffle operations that follow many-to-many communication pattern. As we mentioned before, at the cost of possibly longer routes, this topology enforces a limit on the concurrent network connections a machine can maintain and uses some nodes as intermediate hops. However, with both strategies, the limit on the number of connections that any machine can maintain concurrently; either per query/machine or for the whole machine, is usually decided based on guesses.

CHAPTER 3

UNDERSTANDING THE EFFECT OF HIGH DEGREE CONCURRENT CONNECTIONS

One contribution of this dissertation is to study machines and network performance under pressure, and what are the configurations to be tweaked to maximize the system performance. We will illustrate the tool we implemented to benchmark data transfers over the network between the nodes, and show the effect of some factors on network performance, and on individual machines performance.

3.1 The Simulation Tool

We designed a tool (`github.com/shweelan/network-benchmark`) to measure network performance on a machine. The tool is built with 3 modules: a server module, a client module, and the main module. This tool is written in pure JAVA; no third party libraries where used. This tool can be tested over a set of different setups and parameters, see Table 3.1. Using this tool we were able to understand how different combinations of networking parameters can affect the machine or the network, see Table 3.2.

Table 3.1. The tool parameters description

Parameter	Description	Key	Short key	Default value	example
Hosts	A comma-separated array of remote servers addresses	-hosts	-h	localhost	-h 19.90.29.12, 19.90.3.12
Port	The port number for remote servers	-port	-p	2912	-p 3000
Clients count	The number of concurrent connections to be used	-clientscount	-cc	10	-cc 128
Chunk size	The size - in bytes - of each message communicated to the server	-chunksize	-cs	1024	-cs 8192
Duration	The test duration in seconds	-duration	-d	10	-d 30
Chunk delay	The delay to add between 2 consecutive messages sent to the server in milliseconds	-chunkdelay	-cd	0	-cd 100
Latency duration	The latency test duration in seconds	-latencyduration	-ld	10	-ld 5
Use downlink	A boolean parameter to force the server to reply the messages; make a ping pong	-usedownlink	-uld	false	-uld

Table 3.2. The tool results description

Result	Decription
Throughput	The test over all throughput in MegaBit per second
Min latency	The minimum latency in the test in milliseconds
Max latency	The maximum latency in the test in milliseconds
Median latency	The median latency in the test in milliseconds
Average latency	The average latency in the test in milliseconds
(1, 25, 75, 99) Percentiles	The latency percentiles in milliseconds
JVM median	The median percentage of the CPU used by Java Virtual Machine
Whole system median	The median percentage of the CPU usage for the whole machine

3.1.1 The Server Module. The server module is a simple server that listens to incoming TCP connections on a configurable port, and assign each incoming TCP connection a thread. Connections threads perform no complicated tasks. A Thread will be fetching messages from its TCP connection data-input-stream, and it will either reply with the same message that was received if the client requested the reply, or ignore the message. It would close the connection and exit if the client sent termination command.

3.1.2 The Client Module. The client module is more complicated than the server module, it takes a combination of parameters and sets the environment for it and start to run. The most important parameters for this experiment are, the number of concurrent connections to be maintained for the test, the size of each message in bytes, the duration of the test, and the duration for the ping-pong test (latency measure test). Several more configurations can be useful to use. A delay can be configured to be paused between any 2 consecutive messages sent to server from the same connection, and this can be useful to reduce the stress on the available resources. Multiple remote servers can be included in the configurations, and the module will try to divide the connections over the available servers evenly. A configuration available for the client to ask the server to answer its' messages; this configuration is forced when the module is running latency test.

The client module will open each TCP connection in a new thread and will assign a server host and port for that connection. Each thread will try to connect to the designated server and then immediately will run the latency test for some configurable amount of time. It will send messages to the server and wait for the server to reply while it keeps on sending. When the server responds, it will read the message id and update the record for time elapsed between send and receive. Afterward, it will overflow the connection by messages over and over until the test duration ticks off. The client will sleep each thread whenever it sends a 100MB of data, and this trick was introduced to add stability to the system and reduce the number of failures occurred; especially if the test was running on a limited resources machine. In this part of the test will read incoming messages from the server and clear the buffer to maintain space for more incoming messages. When the duration is elapsed the client will ask the server to terminate and connection, and it will exit the thread after saving all the results.

3.1.3 The Main Module. This module is used to automate the bulk testing. This module gets the path to a configuration file, and starts the remote servers and performs the test cases expressed in the configuration file.

3.2 Experiments

We conducted multiple experiments using the tool described above. In these experiments we tweaked 2 parameters, the number of concurrent connections and the data chunk size, we also tweaked the test duration accordingly adding more time to the cases with the larger number of connections or bigger data chunk size to maintain stability to the system and get reasonable readings. The number of connections parameter set contained $\{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048\}$ connections. The data chunk size parameter set contained $\{256B, 512B, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB\}$ data chunk size.

For this experiment the hardware setup we used was Amazon Web Services Elastic Compute Cloud (Amazon EC2) instances (`aws.amazon.com`), "c4.4xlarge" in particular. "c4.4xlarge" instances offer 16 VCPUs, 30 GBs of memory, and a shared network bandwidth which they mark as high bandwidth; our tests concluded that the network bandwidth was around 4580 MegaBit/Second. As the network bandwidth throughput can be efficiently maximized if we have enough number of concurrent connections sending messages that have an acceptable chunk size, on the other hand, the compute power had to be sufficient to handle a large number of connections sending big data chunks concurrently. We used 2 machines as remote servers and 1 client machine. The goal was to stress the client machine while keeping the server less-busy, that is if we have N connections on the client, then each server will have $\frac{N}{2}$ connections, thus making the bottleneck at the client machine; since the client is where the experiment is evaluated.

We executed the main module on a configuration file that covers all the combinations of "Clients count" and "Chunk size" parameters.

3.2.1 Experiment Results. The results were evaluated and analyzed to better understand the effect of each combination on the nodes or the network. The result can be clustered into 3 significant points of view.

3.2.1.1 Throughput. We noticed that the throughput could be efficiently maximized by opening enough number of connections with proper messages size. Using the machines described above, we were able to maximize the throughput when we had 4 concurrent connections and a message size 8192 Bytes, see Figure 3.1 and Figure 3.2.

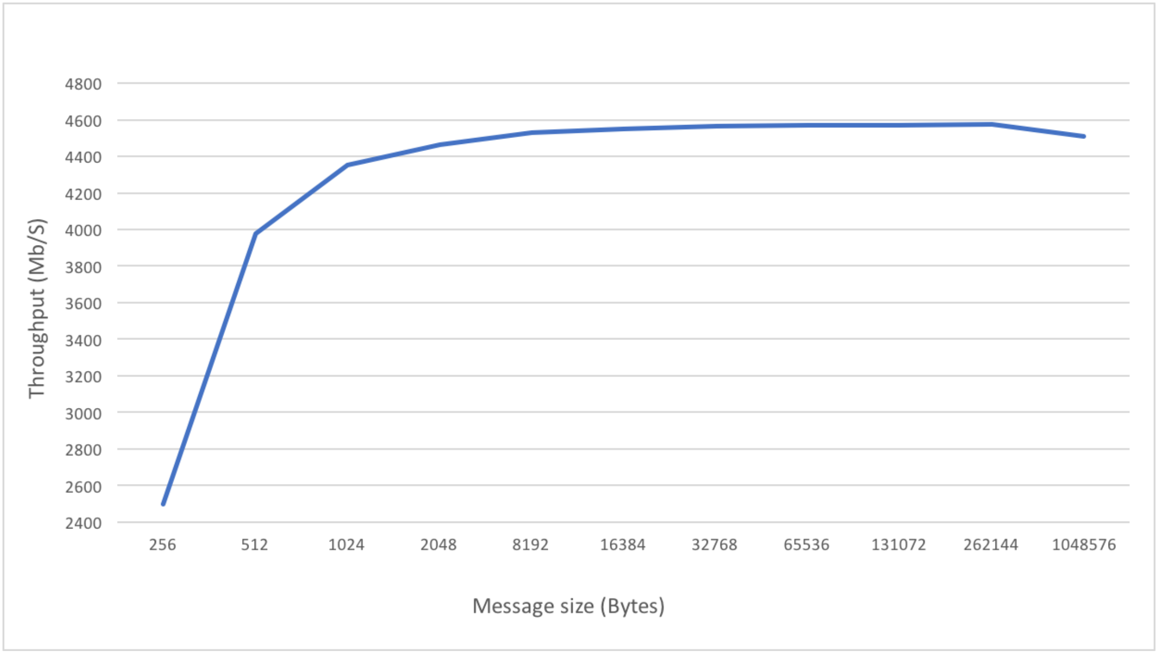

Figure 3.1. Throughput readings with 4 concurrent connections

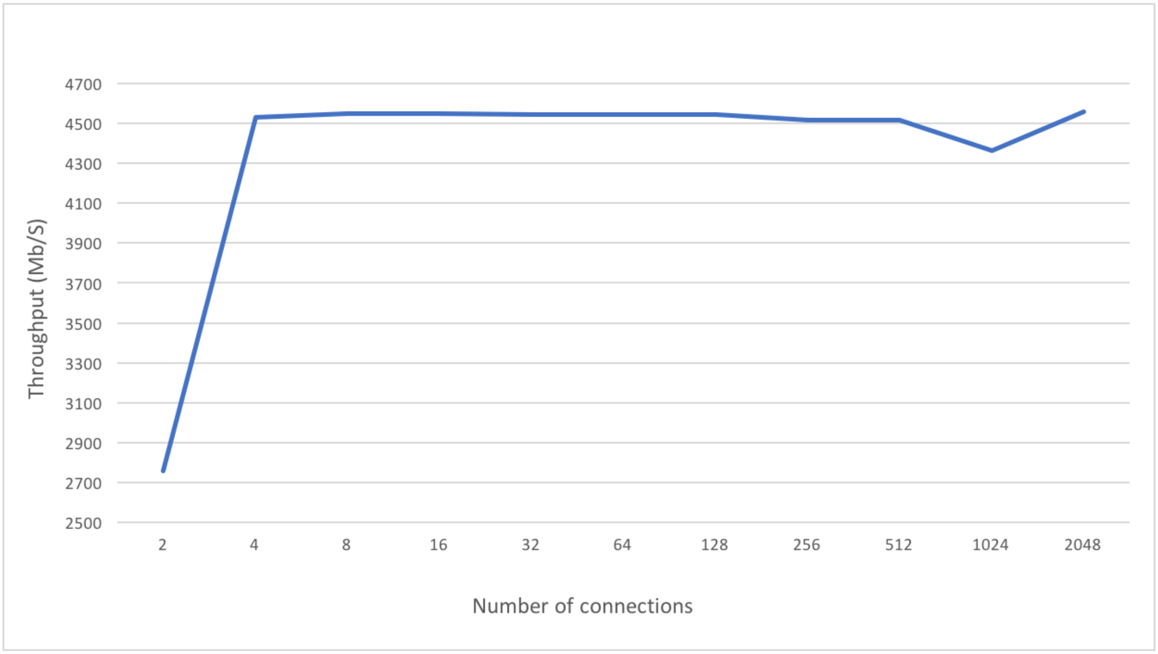

Figure 3.2. Throughput readings with 8192 bytes message size

However, small drops in the throughput were noticed in some cases, mainly when the number of concurrent connections and the message size was too significant.

For instance, it dropped to hit 4000 Mb/S when the test case was 1024 simultaneous connections and 1 MegaByte message size, see Figure 3.3. With larger concurrent connections count, the throughput readings where less stable, the drops were not significant, see Figure 3.4. With bigger message size - 1048576 bytes for instance - we encountered more failures. The client machine came to a halt, and the test failed. This was noticed in Figure 3.5, where the zero readings represent a failure.

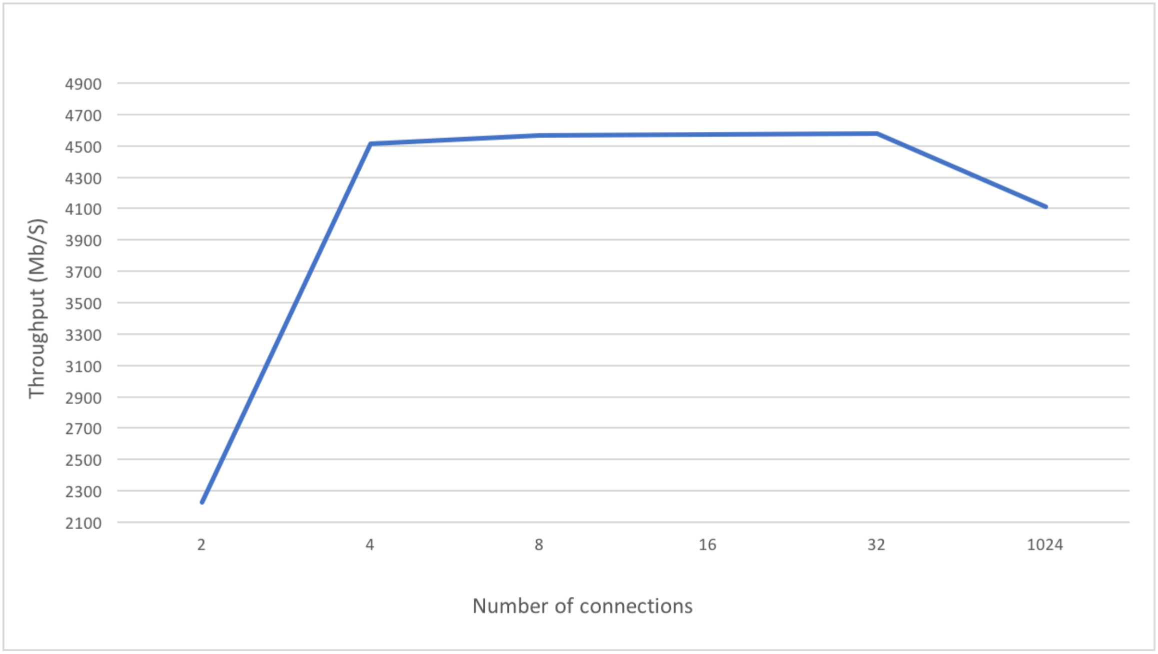

Figure 3.3. Throughput readings with 1048576 bytes message size

In some cases, we noticed abnormal spikes in the throughput readings. This occurred with 2048 concurrent connections and 16384 and 32768 bytes of data, see Figure 3.6 and Figure 3.7. Our best reasonable explanation is that our instances on Amazon AWS are virtual machines and the physical hardware is shared with other virtual machines. However, we did not further investigate this behavior. We rerun the cases causing these spikes multiple times, but the spikes were not regenerated.

3.2.1.2 Latency. We measured the ping-pong latency for the test cases mentioned above. In other words, the client sends a message to a server and measures the

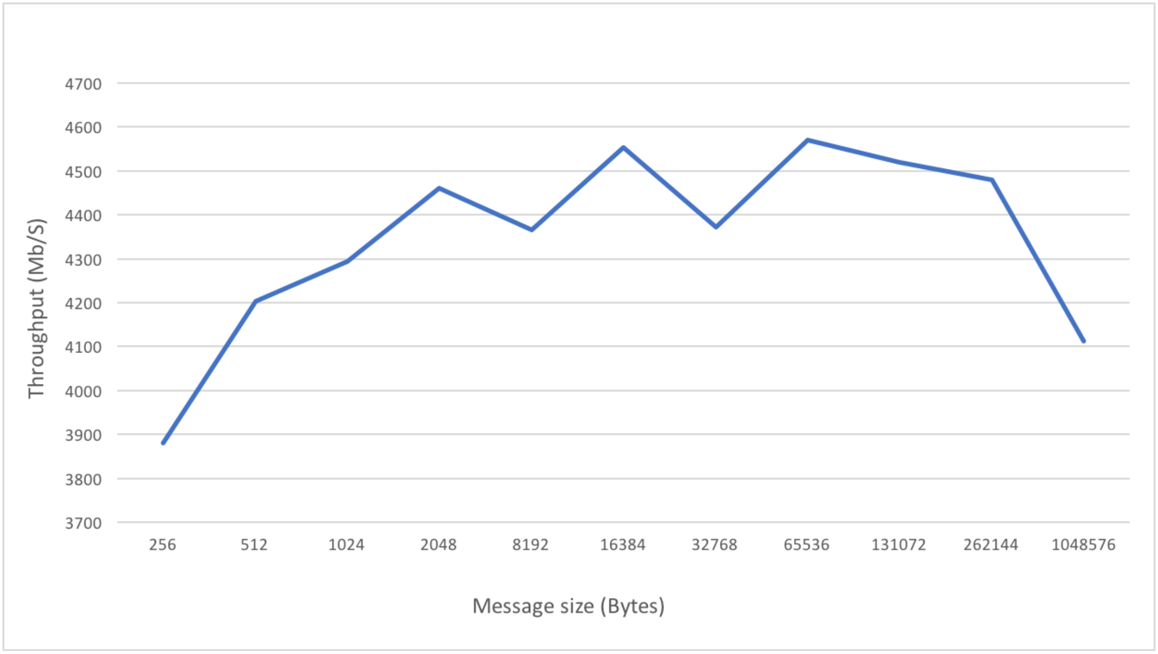

Figure 3.4. Throughput readings with 1024 concurrent connections

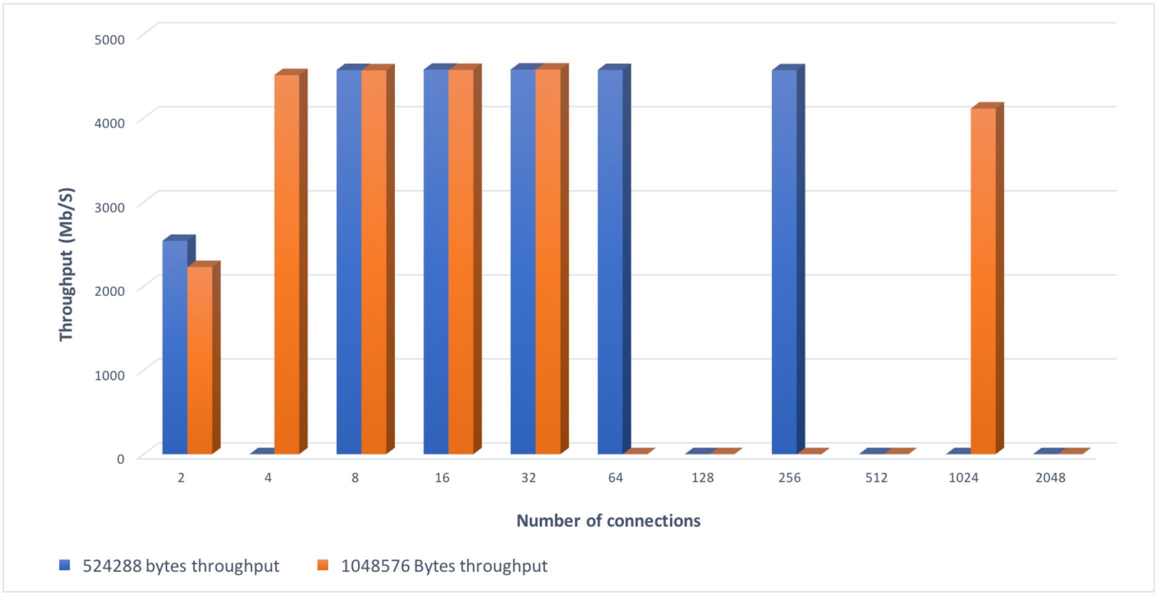

Figure 3.5. Throughput readings with (1048576, 524288) bytes message size showing system failures

response time from the server for that particular message when the server returns the same message.

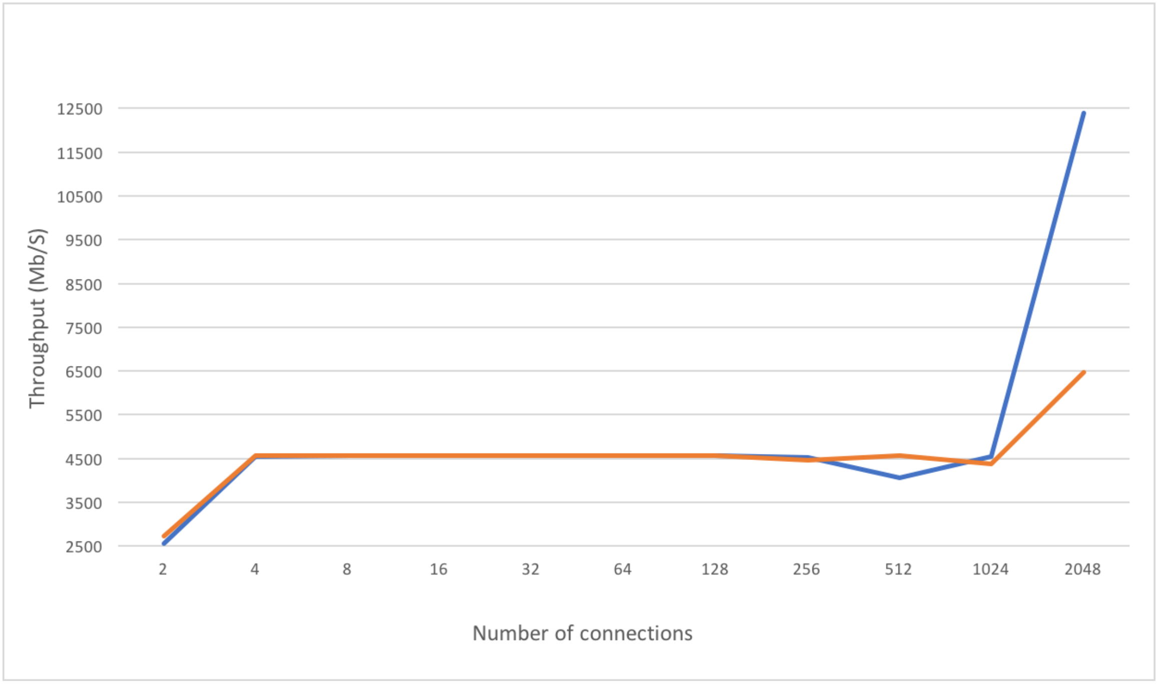

Figure 3.6. Throughput readings with (16384, 32768) bytes message size showing througput spikes

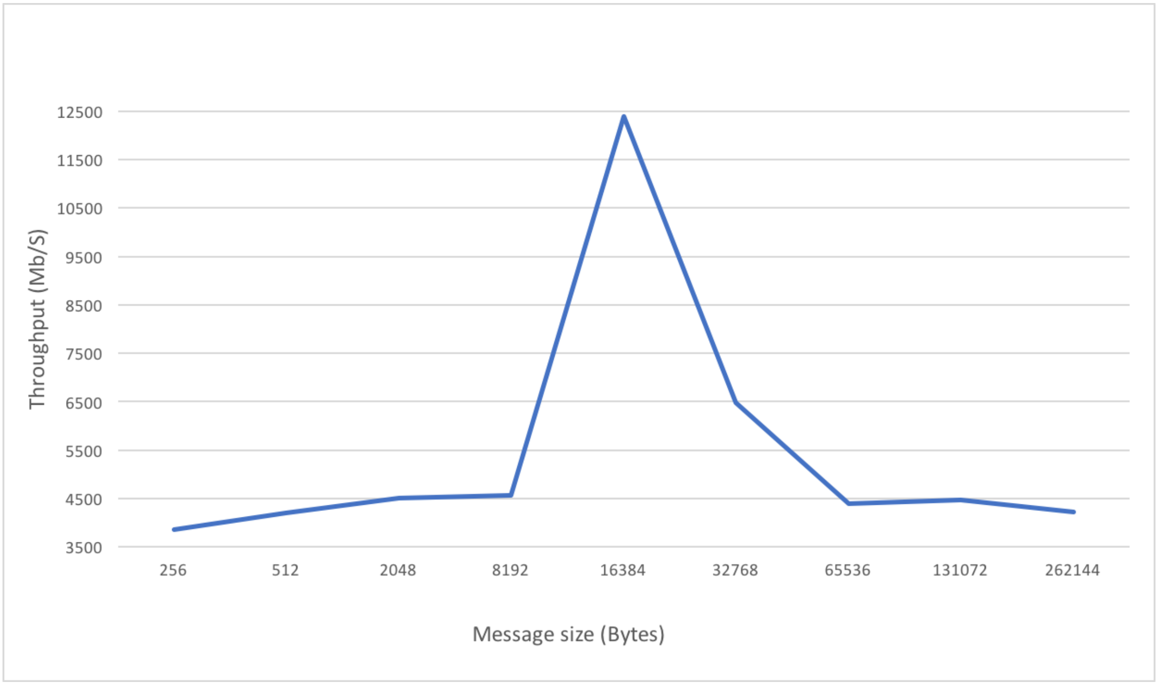

Figure 3.7. Throughput readings with 2048 concurrent connections showing througput spikes

In some test cases where the number of clients is little, and the size of messages is small, the latency of around 20 percent of the messages was higher than expected, see Figure 3.8. This is the effect of Nagle's Algorithm [34] that is implemented by the TCP. Where the TCP protocol keeps small messages buffered, and sends them in one packet once the buffer is full. As we increased the size of the messages, the latency started to be stabilized. When the message size was too big, the latency of some fraction - less than 10 percent - of the messages sent began to increase, see Figure 3.9.

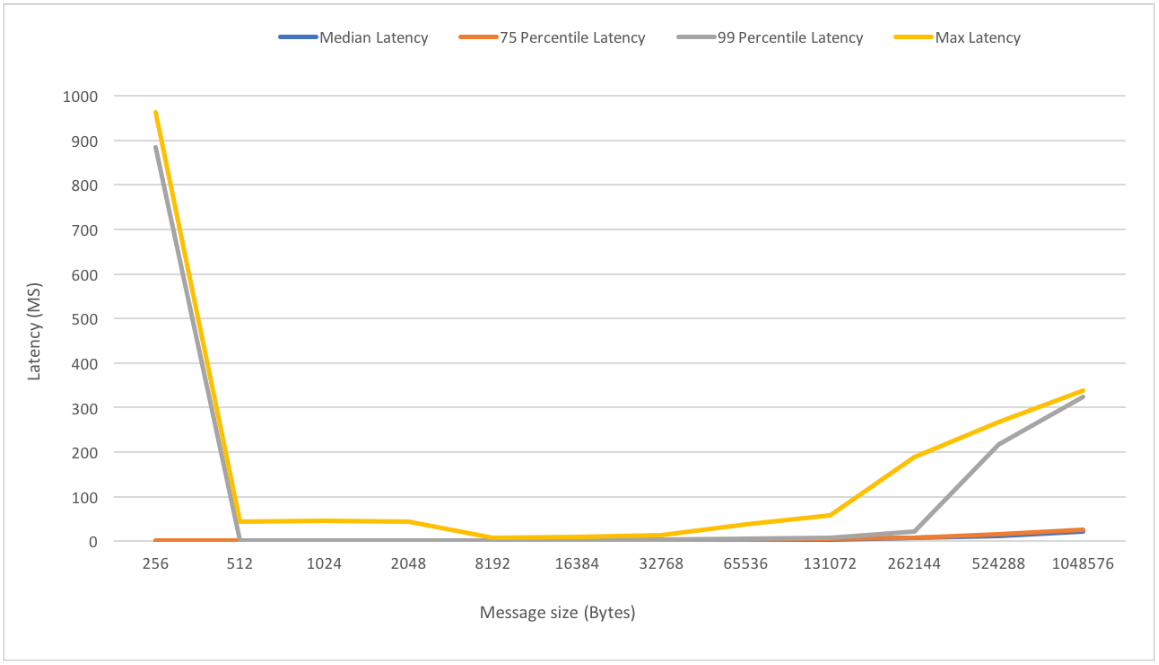

Figure 3.8. Latency readings with 2 concurrent connections

Going more abundant on the number of connections increases the latency, it also increases the portion of the messages that suffered higher latencies, see Figure 3.10 and Figure 3.11. Increasing the number of connections to be more than the number of CPUs cores, causes the latencies to increase significantly, see Figure 3.13. This is an apparent effect of the operating system context switching. Failures frequency increased as we increased the number of concurrent connections, see Figure 3.12.

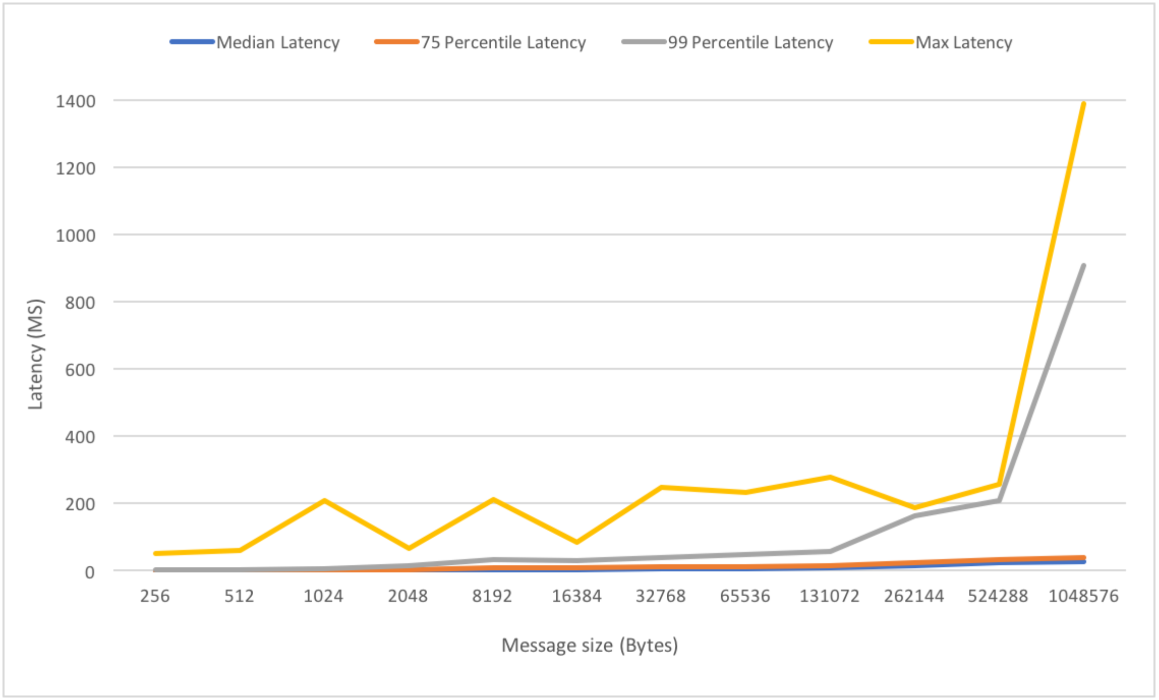

Figure 3.9. Latency readings with 16 concurrent connections

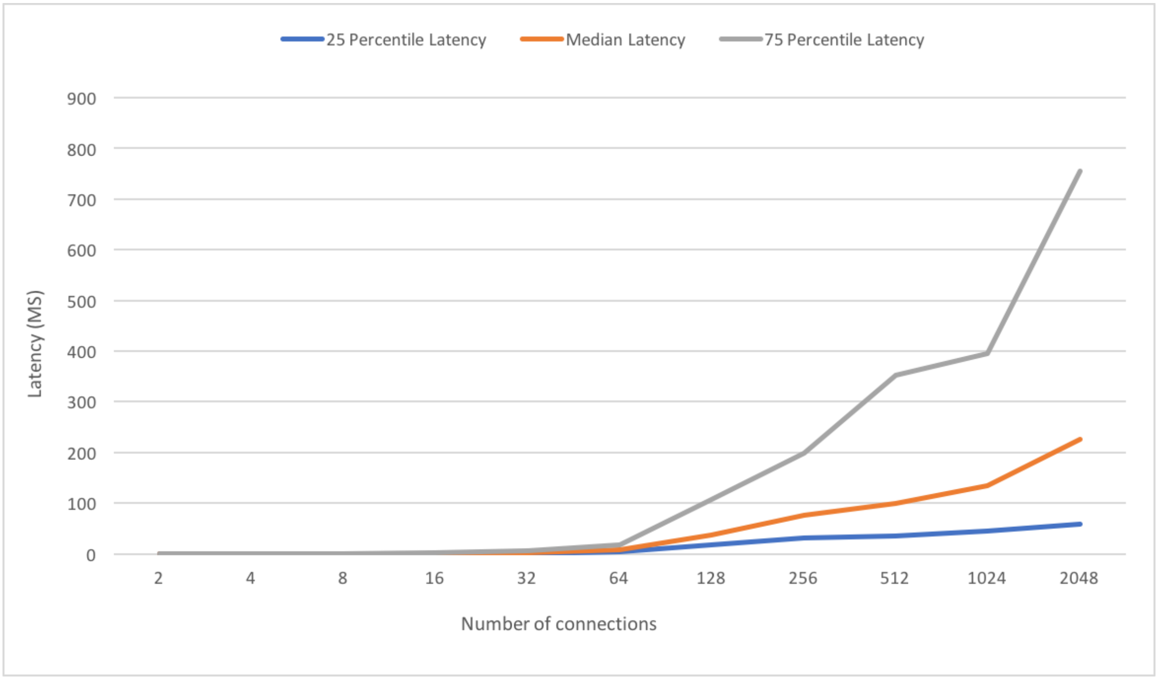

Figure 3.10. Latency readings with 2048 bytes message size

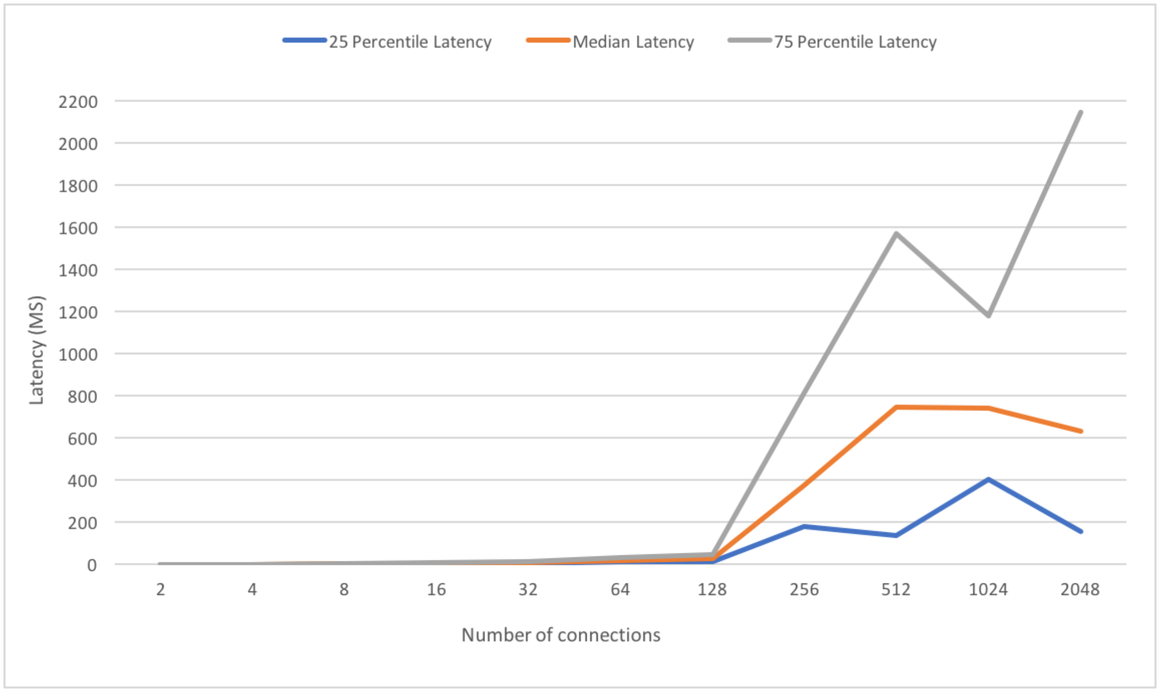

Figure 3.11. Latency readings with 16384 bytes message size

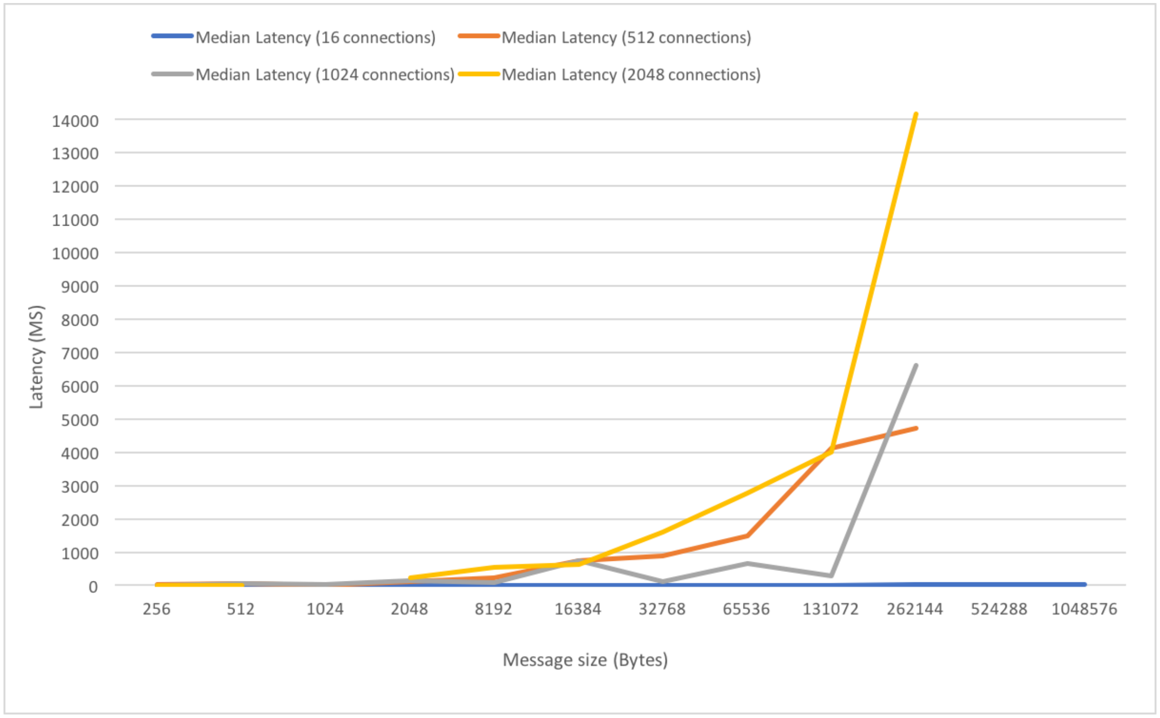

Figure 3.12. Latency readings with larger number of concurrent connections compared to 16 connections

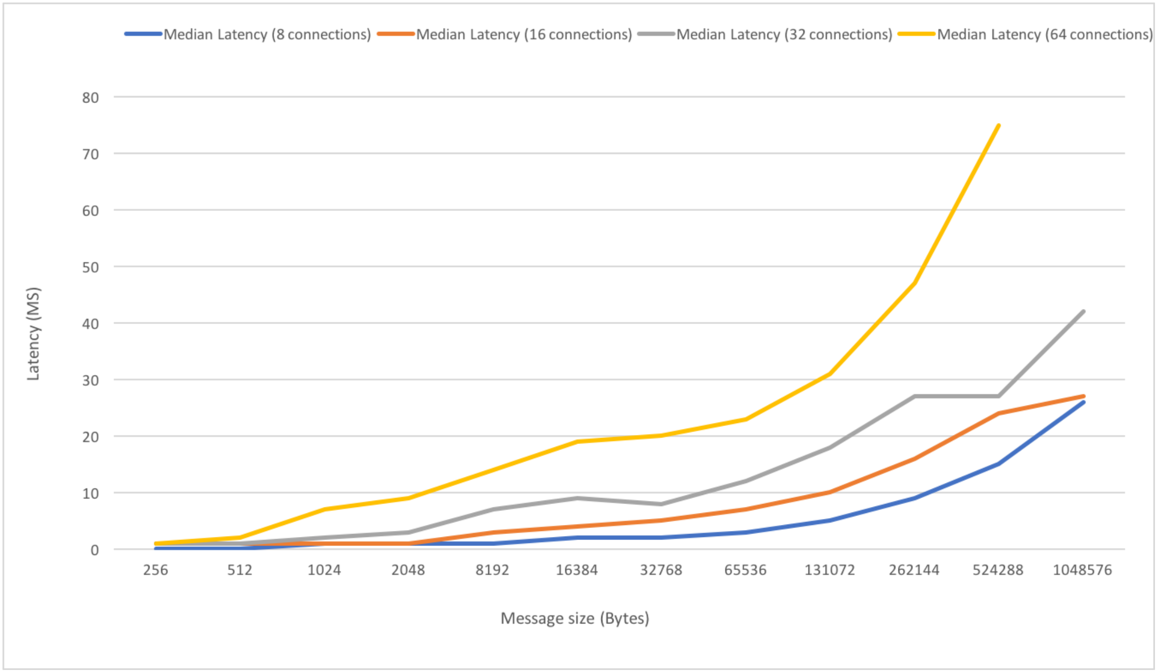

Figure 3.13. Latency readings where concurrent connections are more than the number of CPUs cores

3.2.1.3 CPU usage. In this method, we measure the median CPU usage for the JVM process and the CPU load for the whole system. The JVM process is measured by the percentage of the number of cores available on the system. Since we used 16 cores machines, so the readings of JVM process CPU usage are out of 1600%. For the whole system utilization, the readings are out of 100%. We noticed that the smaller messages kept the CPU more utilized; testing with (256, 512, 1024, 2048) bytes, the whole system utilization readings were 2-4X the utilization of the test with bigger messages. Extremely large messages also increased the CPU utilization, but with a smaller factor, talking about 1.5-2X the utilization of the test with smaller messages. However, using reasonable message size resulted into very stable utilization, see Figure 3.14 and Figure 3.15. The rise in the whole system CPU utilization was non-linear with the increase in the number of concurrent connections. In other words, doubling the number of connections does not necessarily double the CPU usage. The CPU utilization was very much stable while deploying more concurrent connections until we hit the extreme case, where we used 2048 concurrent connections, at that point the CPU utilization started to increase, see Figure 3.16 and Figure 3.17.

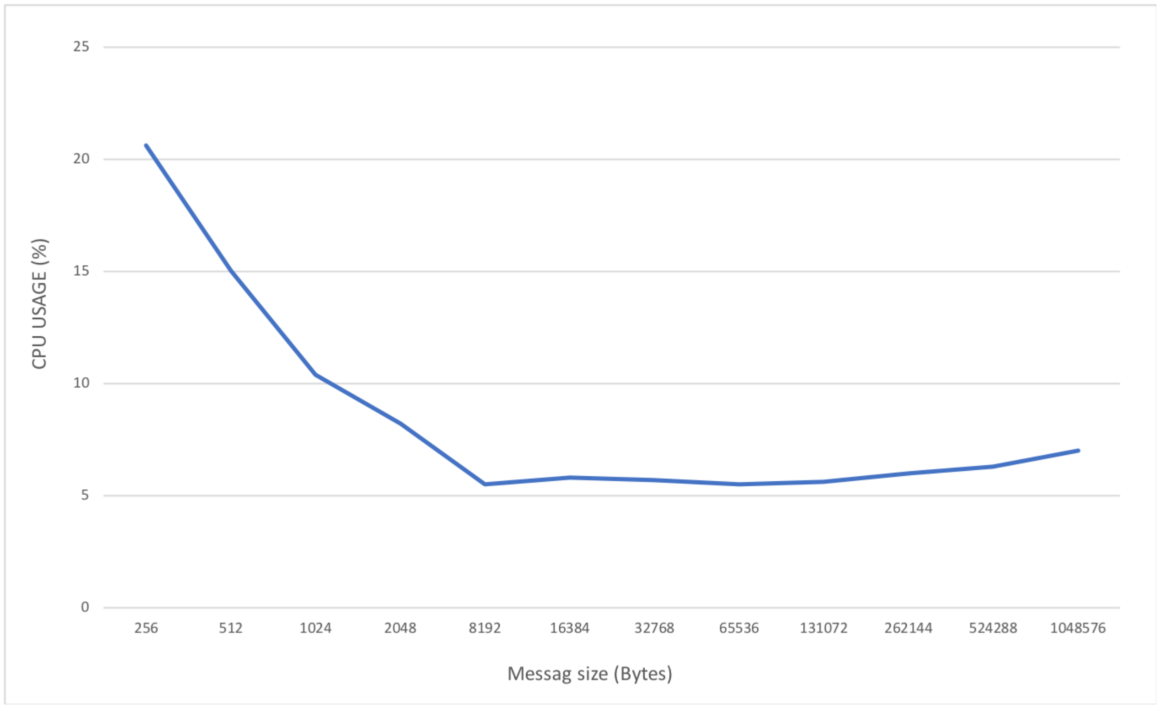

Figure 3.14. Median whole system CPU utilization readings with 16 concurrent connections

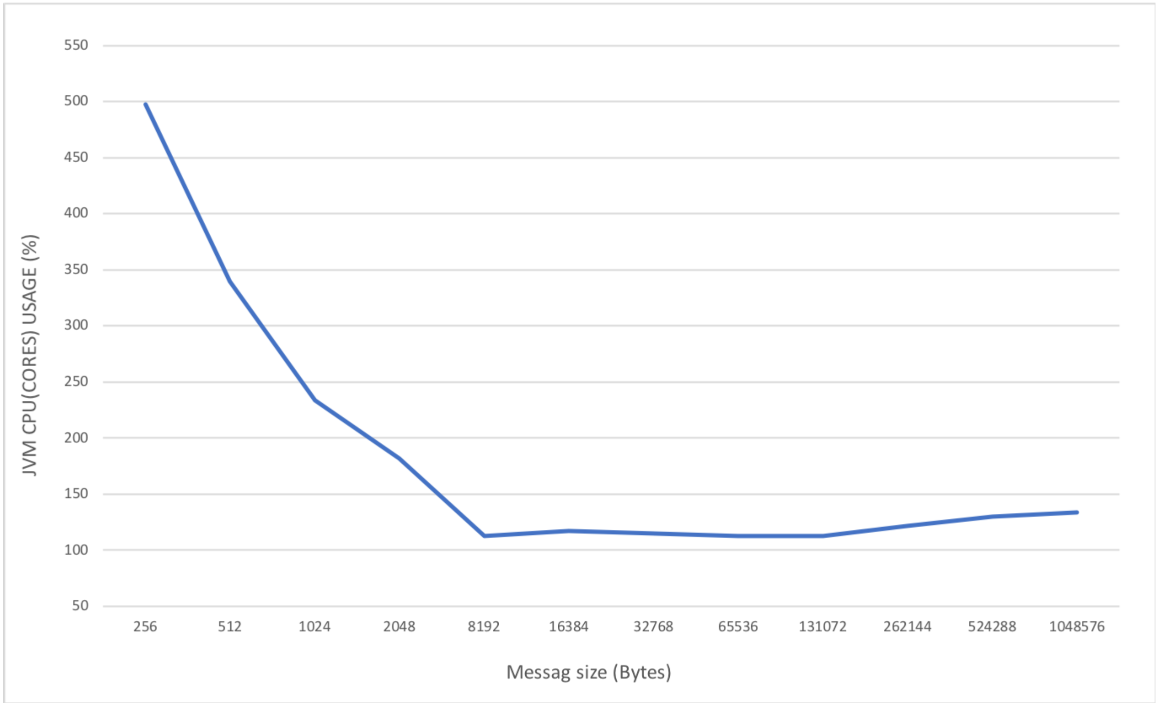

Figure 3.15. Median JVM CPU cores utilization readings (out of 1600%) with 16 concurrent connections

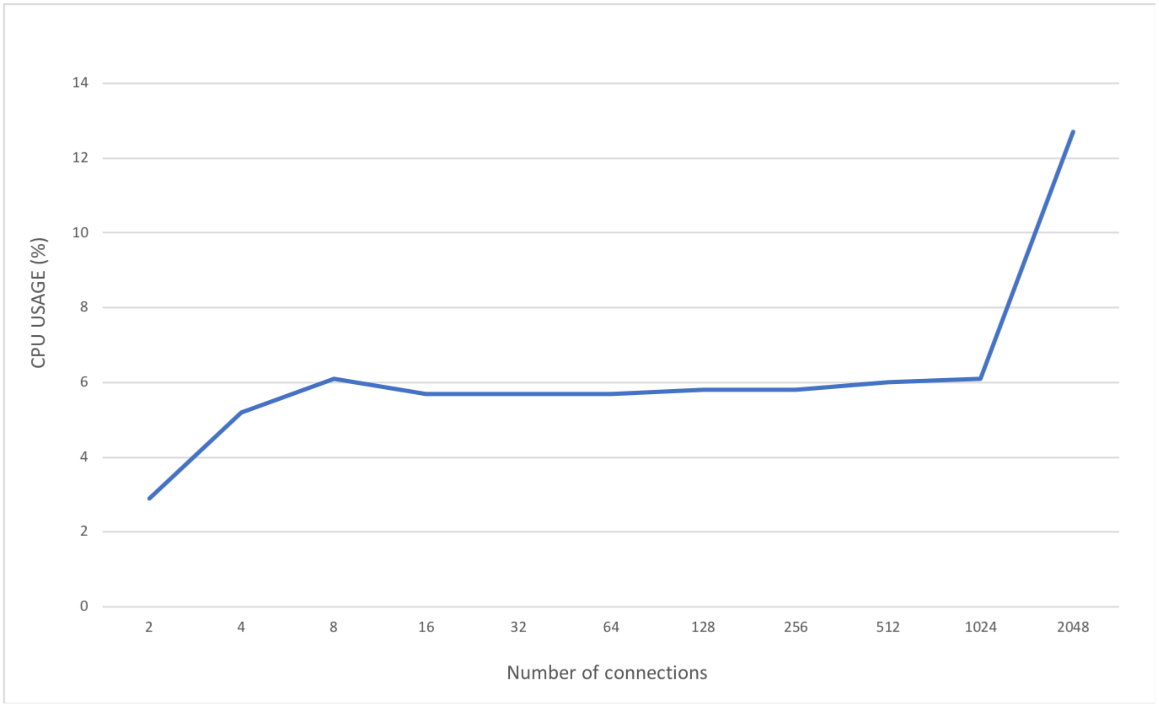

Figure 3.16. Median whole system CPU utilization readings with 32768 bytes message size

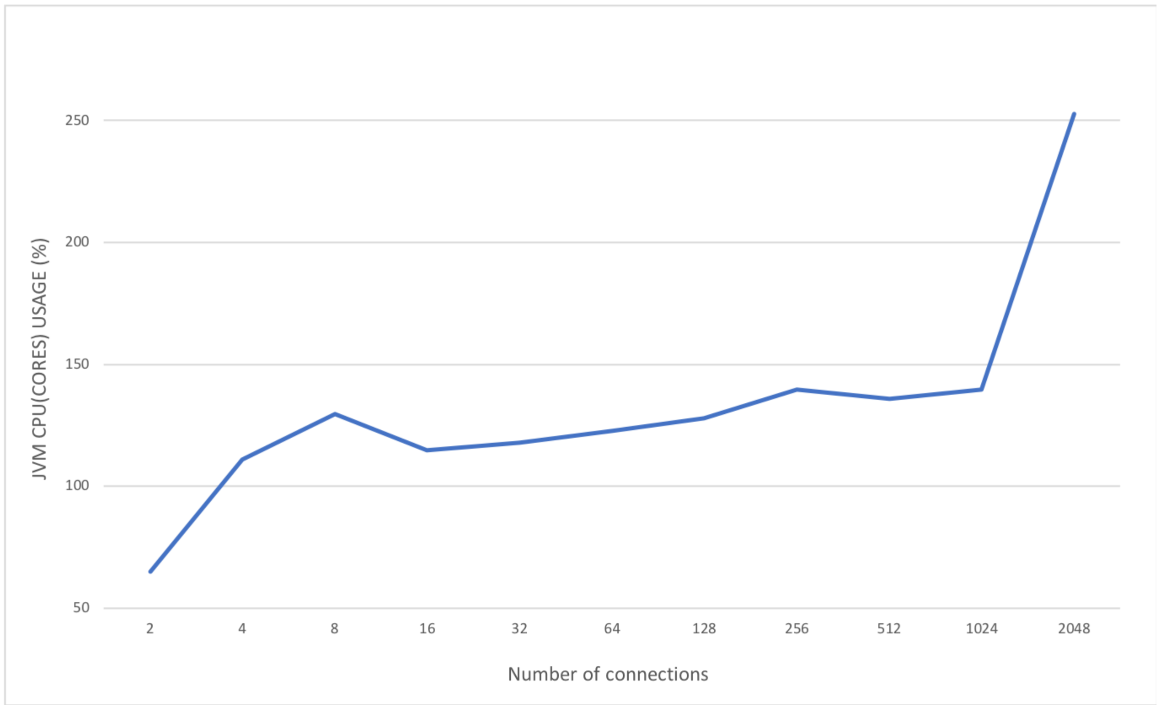

Figure 3.17. Median JVM CPU cores utilization readings (out of 1600%) with 32768 bytes message size

CHAPTER 4
USING DIRECT CONNECTIONS POOL TO IMPROVE DATA-SHUFFLE

Direct connections are always optimal. With direct connections, the diameter is equal to one. This results in having minimum latencies and no wasted messages; the messages rerouting is not needed with direct connections. Using direct connections minimizes the network usage compared to other topologies that involve rerouting. Unfortunately, deploying direct connections to connect to every other node on the system is not scalable. As we noticed in the previous experiments, maintaining too many concurrent connections is problematic.

Topologies such as Binomial Graph Network [12] are scalable. They maintain a low average diameter while deploying a limited number of concurrent connections. The average diameter of the topology is logarithmic to the number of the total nodes on the system. In Binomial Graph topology, where every node maintains a limited number of concurrent connections (L). The overlayed topology creates a mesh network between the nodes and uses routing algorithms to deliver messages to any other node. Every node is reachable $Log_L(N)$ hops at maximum. Beside being scalable, Binomial Graph Topology is fault tolerant, where every destination is reachable by multiple paths. Apparently, some destinations are reachable with higher diameters, i.e., a higher number of intermediate hops; some messages will suffer higher latencies, and will be more wasteful where they will be rerouted many times to be delivered to a destination. However, with uniformly distributed datasets ($AverageDiameter - 1$) more messages will be rerouted. When working on datasets that contain some values that appear more frequently than others, the rerouted messages rate is even worse.

In the rest of this chapter, we will propose a hybrid approach in an attempt to gain more performance and reduce the number of wasted (rerouted) messages sent through the network.

4.1 Solution Idea

In order to have the best of both worlds, the lower degree of limiting the number of concurrent connections per node, and the optimality of the direct connections, we proposed a hybrid approach that enables the execution engine - during runtime - to selectively employ direct as needed relieving the most congested routes. The basic idea is to maintain a pool of extra direct socket connections, and an algorithm to selectively assign these connections to the most congested routes.

During execution time, the algorithm runs every X seconds. It calculates the most congested routes and dynamically opens a socket connection with the most congested routes. In this work, the congested routes are assessed based on the routing heuristics collected during the runtime. Based on 2 factors, the route utilization, and the route length. The route utilization is measured by the number of bytes sent to a particular destination. The route length is the number of intermediate hops between source and destination. The route congestion score is measured as $(RouteLength - 1) * BytesSent$, and this score can be viewed as the number of bytes forwarded through the intermediate hops since the last rerouting cycle. Relieving the most congested routes based on these two factors is basically relieving the routes that cost the systems the most number of bytes wasted. The reduction in the number of the wasted messages leads to utilize the network bandwidth efficiently.

When the data are uniformly distributed over the nodes on the cluster, the longest routes will be relieved with the direct connections. On the other hand, when the data are not uniformly distributed, the most utilized routes will be discharged with direct connections.

4.2 The Simulation Tool

The initial attempt was to implement the direct connections pool on HRDBMS[30]. Unfortunately, for some unforeseen technical difficulties with HRDBMS, we were unable to achieve that. The Binomial Graph Topology was not appropriately implemented with HRDBMS source code available on their Github repository. We designed a tool (`github.com/shweelan/binomial-graph-simulation`) to simulate the data shuffle over a Binomial Graph overlay. The tool is built with 3 modules: the controller module, a startup script, and the main module. This tool is written in pure JAVA; no third party libraries where used. This tool can be tested over a set of different setups and parameters, see Table 4.1. Using this tool we were able to understand how different combinations of networking parameters can affect the machine or the network, see Table 4.2.

Table 4.1. The simulation tool parameters description

Parameter	Description	Key	Default Value	example
Instances count (startup script)	Number of instances to start on each physical machine	-	1	10
n Max	Binomial graph maximum connections	nmax	3	5
Extra Connections	Direct connections pool size	extracons	0	3
Rerouting frequency	Rerouting frequency in milliseconds	reroutefreq	1000	3000
Message size	Message size in Bytes	msg size	4096	65536
Messages count	Number of messages for each Instance to send	msg count	1000	100000
Data distribution (Optional)	A comma-separated array represents the percentage of extra data to load on each instance	datadist	null	3,5,16,15

Table 4.2. The simulation tool results description

Result (Per instance)	Description
Test duration	The total time elapsed to finish the process (query)
Simulation time	The time elapsed while sending the messages
Messages sent, received, forwarded during simulation	Messages transactions readings during the simulations
Messages sent, received, forwarded before simulation	Messages transactions readings before the simulations
Messages sent, received, forwarded after simulation	Messages transactions readings after the simulations
Latencies readings	Different latency readings (Min, Max, Median, Average . . .)

4.2.1 The Controller Module. A remote HTTP module to control the workflow of the simulation, it is used to keep the instances in sync with each other. We used Redis (`redis.io`) and Webdis (`webd.is`) to create a remote HTTP controller server. The data are stored on a Redis server, and we used Webdis to reach Redis via HTTP. It is used by the startup script to declare each physical machine's IP address. After all the machines become ready, it is used to set the bootstrap trigger. When the instances are running, this module is used to store the configurations for each test. It is also used to keep the nodes in sync and aware of each other. At the end of the simulation, it used to record the simulation results.

4.2.2 The Startup Script. A bash script that is started on every physical machine. It clones the project from Github, then announces the machine IP to the remote controller, it also announces the number of instances to be initiated. After that, it waits for the bootstrap key to be set on the controller. When the bootstrap key is set, it launches the instances and waits for them to finish executing. If no errors were returned from the instances, it loops over what it does. See Figure 4.1.

4.2.3 The Main Module. This module is the instance module, every process of this module is an instance of the simulation. It starts by reading the configurations

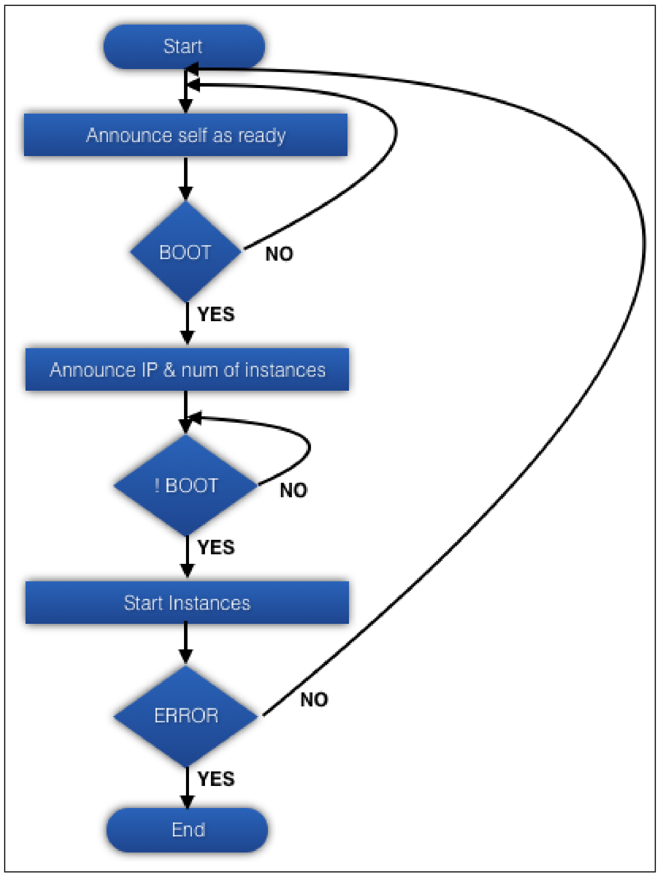

Figure 4.1. Startup Script Flowchart

from the remote controller, and starts its TCP connections listener, and announces its IP and port to the controller. Afterward, it waits for all the expected instances to start their listeners. Then, it calculates it calculates the Binomial Graph overlay based on the configurations provided. It also calculates the routes to every other instance and open TCP connections with the instances that it has a direct connection with. Using the remote controller, it waits again for all the other instances to be ready. Once all the instances are ready and connected, it starts to generate and send messages to random destinations and keeps the stats updated. During the simulation, this module frequently calculates the most congested routes and opens direct connections with the most congested routes. Finally, when all the nodes complete the simulation, it records the results it collected to the remote controller. See Figure 4.2.

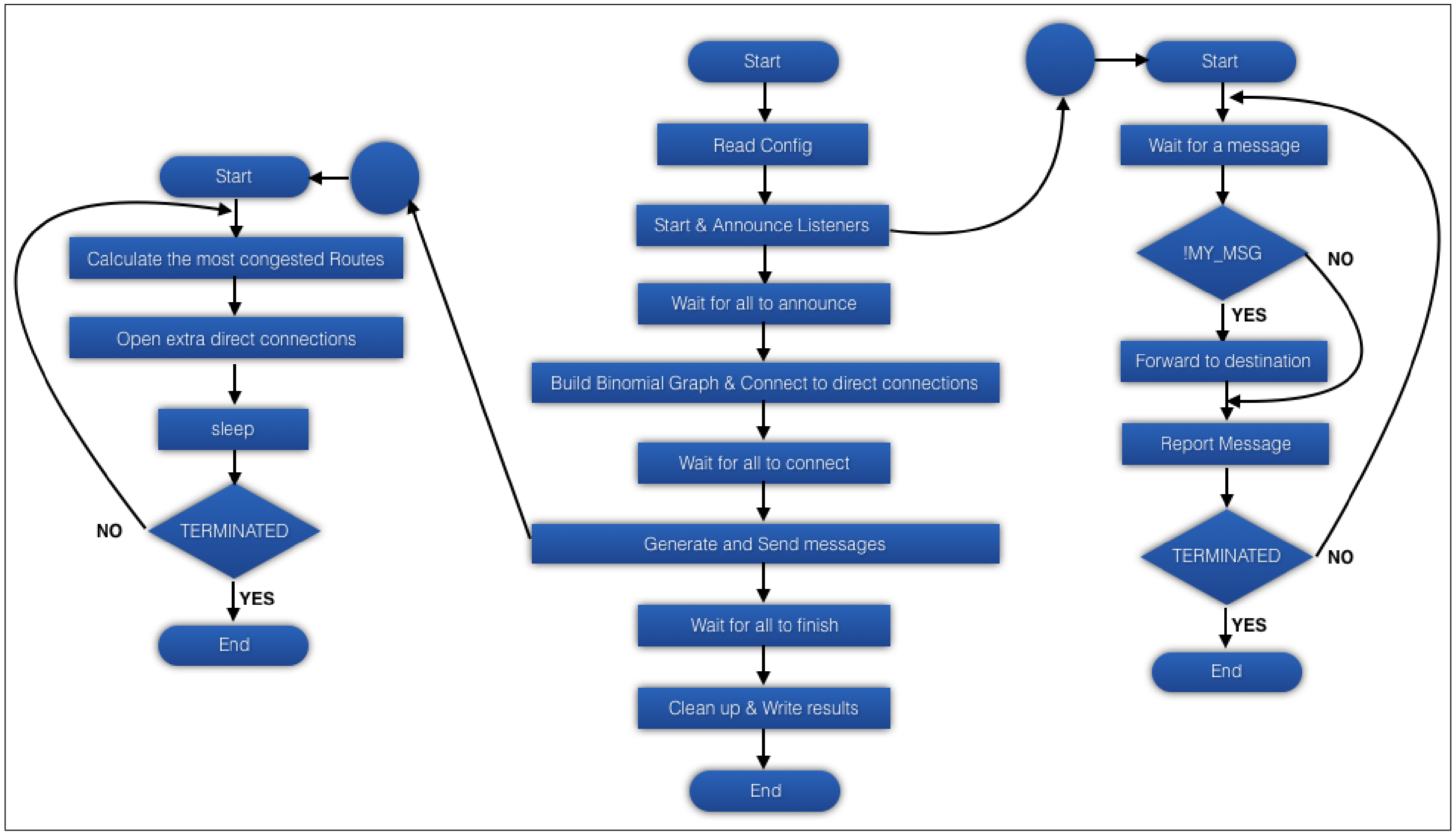

Figure 4.2. Main(instance) Module Flowchart

4.3 Experiments

We conducted multiple experiments using the simulation tool described above. In these experiments we tweaked some parameters, such as the number of direct connections, number of instances, data chunk size, and the percentages of data distribution over nodes; the data distribution was skewed in some cases, and it was random in others. In some test cases, some delays were frequently introduced to achieve more fairness among the processes. Most of the test cases were retried many times to make sure that the readings are accurate and consistent.

For this experiment, the hardware setup we used was Chameleon Chameleon Cloud Baremetal instances (chameleoncloud.org). Baremetal instances offer 48 Hyper-threaded cores processor, 128 GBs of memory, and a dedicated network 10 GigaBit/Second bandwidth. We used 15 physical machines. In some cases we launched 3 instances on each, in other cases we launched 4.

4.3.1 Experiment Results.

The results of this experiment were grouped into 2 major groups.

4.3.1.1 Forwarded Messages Count.

The messages forwarded are the messages sent through the intermediate hops to a destination. When the connection is direct to the destination, there are no forwarded messages.

Small rate - around 1% - of reduction in forwarded messages count was noticed with random data distribution, where the data are uniformly distributed between the cluster instances. This was expected since with the hybrid approach the direct connections pool size is reduced from the number of the original topology is enforcing. Hence it is reducing the average diameter for every other route that is not replaced with a direct connection.

On the other hand, when we skewed the data distribution, and the reduction in data forwarded messages count was ranging from 10% to 30% depending on the data distribution. See Figure 4.3, Figure 4.4, Figure 4.5, and Figure 4.6.

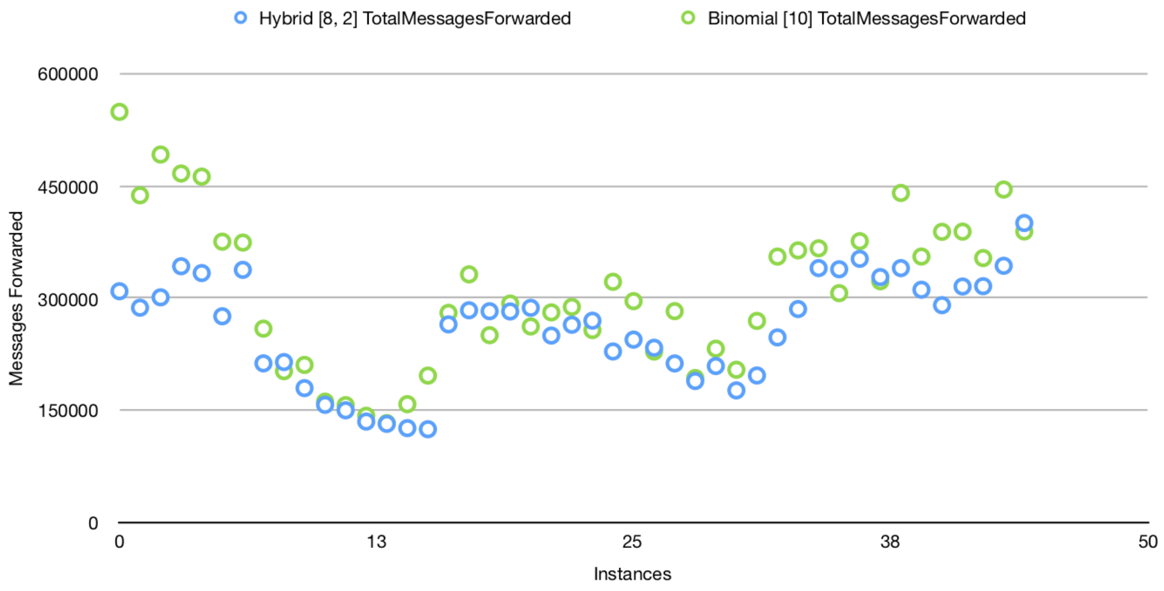

Figure 4.3. Messages Forwarded distribution, showing 16% saving in wasted messages, data points are per instance ID

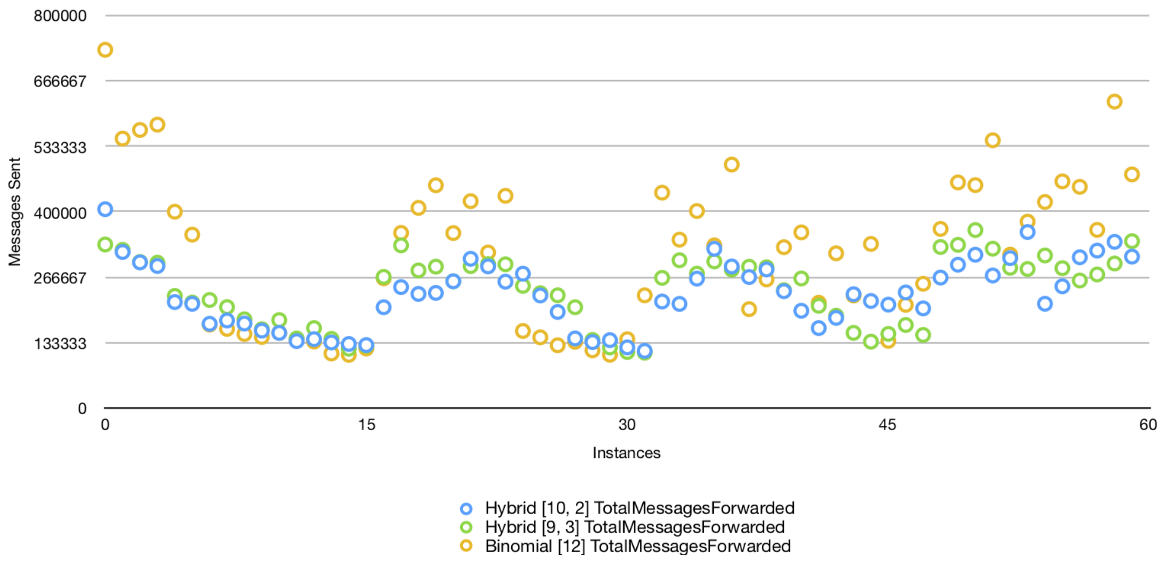

Figure 4.4. Messages Forwarded distribution, showing 24% - 27% saving in wasted messages, data points are per instance ID

As a side effect, we noticed that the running instances were more evenly uti-

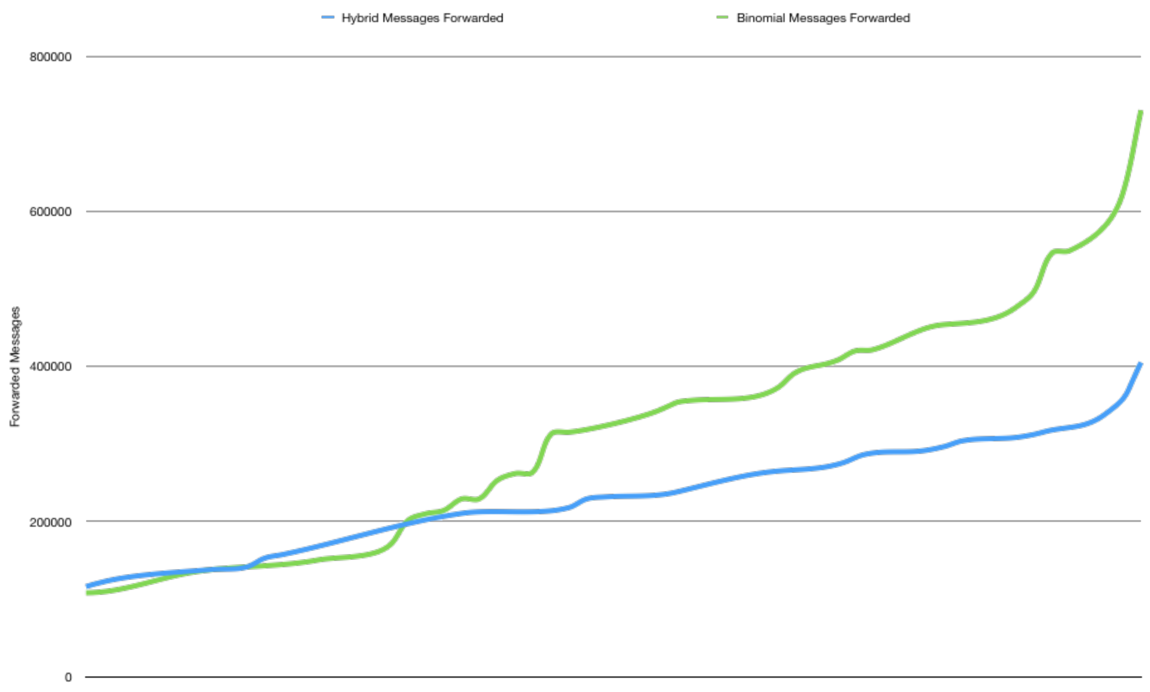

Figure 4.5. Variance in the number of messages forwarded between nodes for every approach, data points are sorted based on the number of forwarded messages

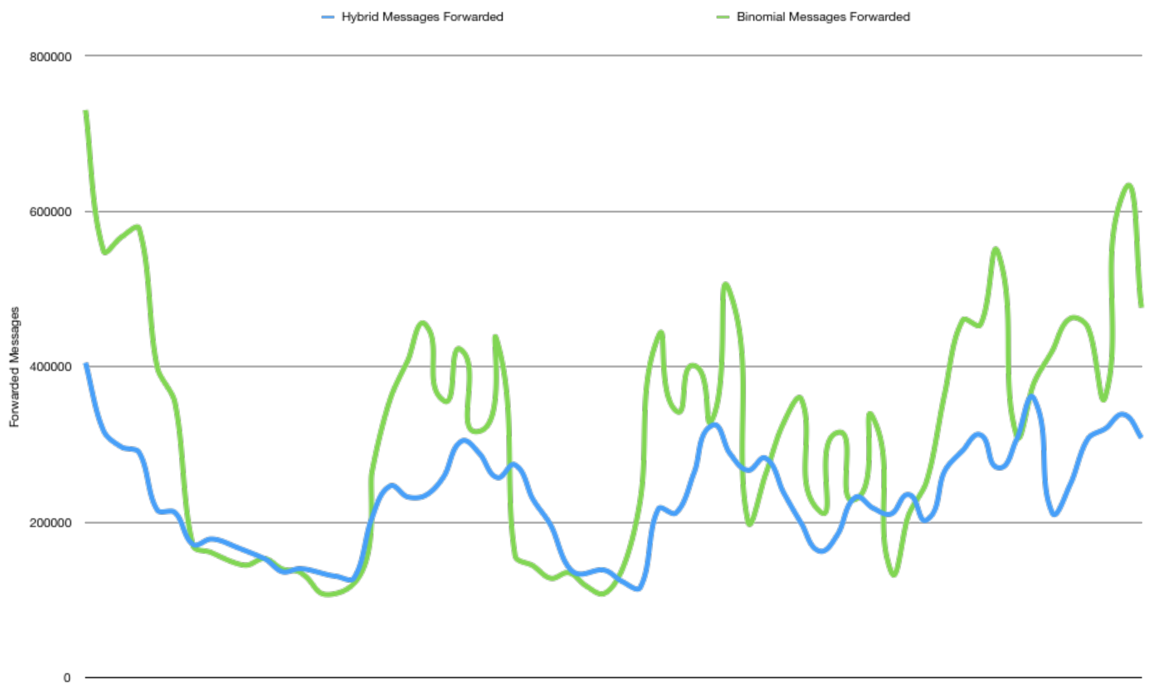

Figure 4.6. Variance in the number of messages forwarded between nodes for every approach, data points are per instance ID

lized. The instances introduced less variance between them in the number of messages forwarded. This effect is taking place since more congested routes will be relieved with direct connections, that will relieve all the intermediate hops for that particular route. This effect is great for load balancing. See Figure 4.5 and Figure 4.6.

4.3.1.2 Time Benefits.

This reading is for the benefit in time for the overall processing time; the time elapsed for all the instances to finish working.

As like forwarded messages count, no Time benefit was recorded when the data were uniformly distributed over the nodes. This was expected since the whole-system average diameter diminished. Moreover, around 1%-2% reduction in performance was noticed. We believe this is due to TCP reconnecting overhead.

With skewed data distribution and on a stressed enough machine, the time benefit recorded around 5% - 10% reduction in total processing time, see Table 4.3.

4.3.1.3 Other readings.

Some improvements were noticed with average and median latency. The latency improvements were not consistent over the whole system, since the machines where all on the same switch.

As we already know, transferring data over the network needs to use CPU and memory. Sending fewer messages over the network will save the CPU, and memory usage. Moreover, it will reduce the congestion over the network resources, hence saving network bandwidth for other messages and other queries to be sent over. This leads to better utilizing the CPU for the actual work that matters and increasing the network efficiency.

Table 4.3. Time benefit with different configurations

Messages Count	Messages Size	Instances Count	Connections Count	Data Skewed	Delays Used	Binomial Avergage Time (Sec)	Hybrid Average Time (Sec)	Time Benefit (Sec)	Benefit Percentage	Clarifications
300K	64KB	60	12	Yes	50 ms	557	502	55	10	Delays, messages size, and enough instances were sufficient to stress the system
300K	64KB	60	10	Yes	No	507	466	41	8	Without the Delays, the performance boost was less
300K	256KB	40	10	Yes	100 ms	928	867	61	7	Delays, and bigger messages were sufficient to stress the system despite the use of fewer instances
300K	64KB	60	16	No	No	322	352	-3	-1	With uniformly distributed dataset, no performance gain, -1% drop in time is due to TCP reconnecting overhead
300K	64KB	60	12	No	No	318	324	-6	-2	2% drop in time due to TCP reconnecting overhead

CHAPTER 5
CONCLUSION

In data-parallel systems that scale up to thousands of nodes where data shuffling operations are necessary for the job processing, network configuration is critical to optimize resources utilization, system performance, system balance, and system scalability. Data-buffering and messages chunk size to be sent over the network are critical parameters that need to be tweaked to improve performance and resources utilization. With small size messages, higher CPU utilization, higher latencies, and lower throughput were noticed. On the other hand, extremely large messages also yielded higher latencies and unstable throughput for the overall system. The network connections concurrency degree is another major factor that can affect system performance and scalability. Hence, keeping an unlimited number of concurrent connections is impossible. CPU, memory, and state management will suffer as the network connections count increases; eventually, the hosting machine will fail. Keeping only a few concurrent connections yielded lower throughput. On the other extreme, too many connections produced higher CPU utilization, higher latencies, and unstable throughput. Not to forget, the more network connections, the more memory is consumed.

Network overlayed topologies such as Binomial Graph and Torus enforce a limited number of concurrent network connections while keeping the average diameter as short as possible. The value of the limit on the number of simultaneous network connections must be well evaluated and carefully selected to maximize performance and scalability. Implementing these topologies expands system scalability. However, with these topologies, some destinations are farther than the others; they maintain higher diameters to be reached. When these routes are congested, more system resources are utilized to forward the messages over the hops to the destination. By using the hybrid approach and make use of direct connections when necessary to relieve the congested routes is advantageous to reduce the amount of data repeatedly sent over the network. As a consequence, The system performance will be boosted, and the resources will be better utilized.

The decrease in the node degree in favor of the direct connections pool causes a reduction in the average diameter of the network framework graph. Hence, with evenly distributed data there are fewer benefits. Contrariwise, with data that are skewed and not evenly partitioned data, the hybrid approach is saving great fractions of the wasted (rerouted/forwarded) messages and awarding a decent performance boost. Reducing the number of messages forwarded leads to a reduction in CPU and memory utilization, that can be utilized for other tasks on a busy system.

Even though the experiments showed us that the hybrid approach increased the system load balancing, but the hybrid approach working mechanism could affect the nodes' degree equality, where some receiver nodes might have too many incoming connections. Hence, getting the communication graph to be irregular. This could happen only where the data are highly skewed, and there are a few values that are excessively repeated. However, a simple solution idea for this problem can be granting the receiver nodes the control to accept or reject the direct connections requests, in the case of being overwhelmed and already received enough incoming direct connections.

CHAPTER 6
FUTURE WORK AND RECOMMENDATIONS

The tool we described in Chapter 3 can be used with topologies that enforce a limit on the number of concurrent connections to configure the limit depending on the machine's resources correctly.

Although the idea described in Chapter 4 proved that it could improve performance and resources utilization, it also demonstrated that the direct connections portion of the number of connections allowed on the system is critical. The reduction in the number of connections (the topologies degree limit) will lead to possible longer average route length, that will affect all the routes. Depending on the network utilization, and the data heuristics, the direct connections portions can be decided and reserved. Conducting more experiments to optimize this ratio can be advantageous.

Rerouting algorithm proposed in this dissertation is a lightweight algorithm. More sophisticated algorithms can be tested to employ this hybrid approach better. Using previously executed queries heuristics to anticipate how could this query be executed to utilize the direct connections better. The currently implemented algorithm decides its routes based on local congestion information, it tries to optimize locally. Rerouting based on global information by using information from other nodes to assess the whole overlayed network congestion can turn up better rerouting decisions. Another benefit of using global, and comprehensive knowledge about the communications framework is to employ the direct connections within the original overlayed topologies. Information about the direct connections on each other node can be used to exploit the direct connections further and reduce the number of hops to distant destinations.

BIBLIOGRAPHY

[1] M. Asay, "Why the world's largest Hadoop installation may soon become the norm," 2014. [Online]. Available: https://www.techrepublic.com/article/why-the-worlds-largest-hadoop-installation-may-soon-become-the-norm/

[2] Hadoop, Apache, "Powered by Apache Hadoop." [Online]. Available: https://wiki.apache.org/hadoop/PoweredBy#Y

[3] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] A. Hadoop, "Hadoop," 2009.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS operating systems review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google file system.* ACM, 2003, vol. 37, no. 5.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on.* Ieee, 2010, pp. 1–10.

[9] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, no. 2007, p. 21, 2007.

[10] J. Arnold, B. Glavic, and I. Raicu, "HRDBMS: Combining the Best of Modern and Traditional Relational Databases," *Illinois Institute of Technology, Department of Computer Science, PhD Oral Qualifier*, 2015.

[11] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[12] T. Angskun, G. Bosilca, and J. Dongarra, "Binomial graph: A scalable and fault-tolerant logical network topology," in *International Symposium on Parallel and Distributed Processing and Applications.* Springer, 2007, pp. 471–482.

[13] Y. Saad and M. H. Schultz, "Topological properties of hypercubes," *IEEE Transactions on computers*, vol. 37, no. 7, pp. 867–872, 1988.

[14] B. Khasnabish, "Topological properties of Manhattan street networks," *Electronics Letters*, vol. 25, no. 20, pp. 1388–1389, 1989.

[15] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken *et al.*, "Blue Gene/L torus interconnection network," *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 265–276, 2005.

[16] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, “Managing data transfers in computer clusters with orchestra,” *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.

[17] Amazon Web Services, “Amazon EC2 Pricing.” [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[18] M. J. Karol, “Optical interconnection using shufflenet multihop networks in multi-connected ring topologies,” in *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4. ACM, 1988, pp. 25–34.

[19] S. Ohring and S. K. Das, “Folded petersen cube networks: New competitors for the hypercubes,” in *Parallel and Distributed Processing, 1993. Proceedings of the Fifth IEEE Symposium on.* IEEE, 1993, pp. 582–589.

[20] K. N. Sivarajan and R. Ramaswami, “Lightwave networks based on de Bruijn graphs,” *IEEE/ACM Transactions on Networking (TON)*, vol. 2, no. 1, pp. 70–79, 1994.

[21] G. Panchapakesan and A. Sengupta, “On a lightwave network topology using kautz digraphs,” *IEEE Transactions on Computers*, vol. 48, no. 10, pp. 1131–1137, 1999.

[22] J. R. Goodman and C. H. Sequin, “Hypertree: A multiprocessor interconnection topology,” *IEEE Transactions on Computers*, no. 12, pp. 923–933, 1981.

[23] S. Campbell, M. Kumar, and S. Olariu, “The hierarchical cliques interconnection network,” *Journal of Parallel and Distributed Computing*, vol. 64, no. 1, pp. 16–28, 2004.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network.* ACM, 2001, vol. 31, no. 4.

[25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[26] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing.* Springer, 2001, pp. 329–350.

[27] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, “Tapestry: A resilient global-scale overlay for service deployment,” *IEEE Journal on selected areas in communications*, vol. 22, no. 1, pp. 41–53, 2004.

[28] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, “ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table,” in *Parallel & distributed processing (IPDPS), 2013 IEEE 27th international symposium on.* IEEE, 2013, pp. 775–787.

[29] D. DeWitt and J. Gray, “Parallel database systems: the future of high performance database systems,” *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[30] IITDBGroup, “HRDBMS.” [Online]. Available: https://github.com/IITDBGroup/HRDBMS

[31] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, “Spark sql: Relational data processing in spark,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[32] L. Lin, V. Lychagina, W. Liu, Y. Kwon, S. Mittal, and M. Wong, “Tenzing a sql implementation on the mapreduce framework,” 2011.

[33] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, “Hive: a warehousing solution over a map-reduce framework,” *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[34] Wikipedia contributors, “Nagle’s algorithm — Wikipedia, the free encyclopedia,” 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Nagle%27s_algorithm&oldid=840406204