UNIVERSITY OF OSLO
Department of Informatics

A tandem queue distribution strategy for data subscription oriented nodes

Master thesis

Sigfred Sørensen

Network and System Administration

Oslo University College

May 20, 2012

A tandem queue distribution strategy for data subscription oriented nodes

Sigfred Sørensen

Network and System Administration
Oslo University College

May 20, 2012

Abstract

Fast data distribution is important for many businesses and services today. If data distributions like operating-system deployment, media/file distribution and patching is slow it could negatively impact productivity. The work in this thesis is aimed at improving performance of data distribution where the receiving nodes are in a data subscriber relationship. The main idea is that this could be achievable by first discovering the network topology and then traverse the network with a snake like behavior, in other words traversing each network link only once. In this thesis the focus is on exploring what performance gain there could be when using the proposed distribution strategy.

The thesis goal is approached through two main steps. First, transport protocols are used to benchmark switch duplexing performance. These benchmarks are aimed at finding out what performance can be expected and if the proposed distribution strategy is viable. Second, a proof of concept prototype based on the proposed distribution strategy is created. The prototype is compared with BitTorrent in a distribution scenario. The findings in this thesis show that there could indeed be a performance gain by using the proposed distribution strategy.

Acknowledgements

I would like to express my gratitude to the following individuals for their support during my time at Oslo University College and over the course of this master thesis.

My supervisor Tore Møller Jonassen for guiding my master thesis in the right direction, acquiring the necessary equipment and ensuring the quality of my work.

The staff at Oslo University College that has provided a good environment for learning.

My fellow students for interesting discussions, providing a good social environment and making most days at the university a joy.

My family for patience and understanding during this busy time in my life.

An extra thanks goes to my little-sister Kristine Kristiansen for her moral support. Wish you the best of luck as you now start you academic pursuit.

A special thanks goes to my girlfriend Kristin Paulsen for her support, love, patience and understanding. You have helped me improve my mathematical skills and been someone to bounce my ideas off. If writing a master thesis was a sport; you have not only been my cheerleader, but also my best team-player.

May 20, 2012
Sigfred Sørensen

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

Fast data distribution is important for many businesses and services today. If data distributions like operating-system deployment, media/file distribution and patching is slow it could negatively impact productivity.

Network and system administrators are often in charge of distributing data between computer devices. When the time comes for the network and system administrator to distribute the data, the receivers often need the same data simultaneously. Additionally the data are often distributed by a single server, which adds to the logistical challenge. In these distribution scenarios the receivers are in a data subscriber relationship, where the receivers are known and which receivers are to receive what data. The data subscribers are waiting until the day a security patch, software update or the new files are ready for distribution. Data subscriber relationship is a central topic in this thesis, and is a mandatory prerequisite for the proposed distribution strategy.

The main goal of this thesis is to explore the viability of a data distribution strategy. The distribution strategy should improve the speed of data distribution and reduce redundant traffic. The relevant distribution scenarios are when the receivers are in a data subscriber relationship. The general application domain portrayed in this thesis applies to network and system administration relevant scenarios.

In the following subsection a relevant problem scenario will be discussed, and an introduction into the proposed seeding strategy will be presented.

1.1.1 A problem scenario

Figure 1.1 illustrates how a common network topology could be structured. The task is to deploy a self installing operating-system to all computers as efficiently as possible. There are several distribution methods that can be employed to distribute the data required to install the operating-system. In this section distribution methods relevancy to the illustrated problem will be discussed.

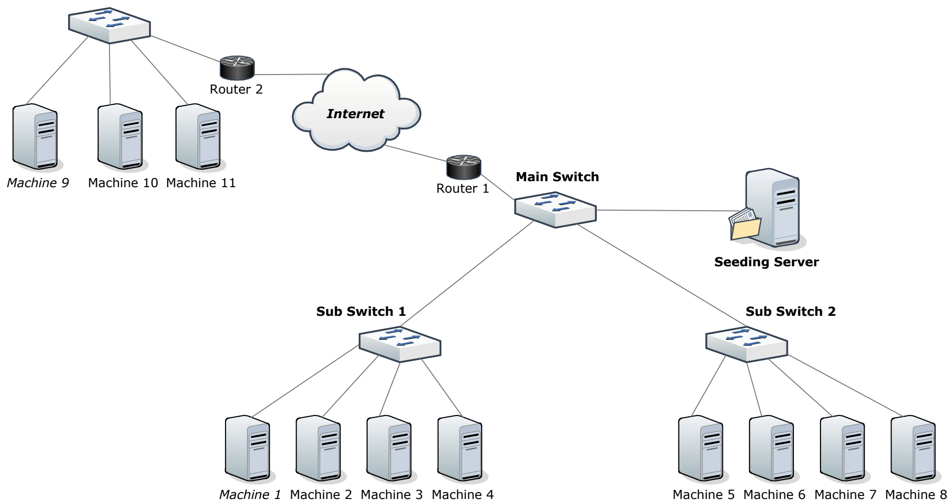


Figure 1.1: An illustration of how a common network topology could look like.

Unicast

Unicast is a commonly used data delivery method for both local-area network and over the Internet. Unicast distributes data to each receiving client individually. The required bandwidth for Unicast distribution is directly proportional to the number of receiving clients [42]. This proportionality makes the distribution-time scale linearly to the number of recipients. This means that doubling the number of recipients is likely to double the distribution-time, and is therefore not well suited for suited for data distribution scenarios where there are multiple simultaneous receivers.

IP-layer multicast

IP-layer multicast was first proposed in the early 80s, and was modelled in 1989 [8]. It is a technology that enables "one to many" and "many to many" distribution over network. IP-layer multicast is achieved by replicating the

the packets in the network devices, such that a copy of the same data is sent to each receiver. In theory IP-layer multicast should use equal amount of time to distribute data to any amount of nodes, for distribution-time this is considered optimal.

To implement IP-layer multicast it is required that all network devices support multicast as specified in [8]. This implies that every router, switch, wireless access-point, host and server needs to be multicast compliant and configured properly [5, 19, 12]. Despite much research, IP-layer multicast has problems with administrative, security and scalability issues [21, 9, 1]. These issues were presented quite some time ago, but have still not been resolved. These issues has prevented IP-layer multicast protocols to become widespread on the Internet. Additionally IP-layer multicast is a "best effort" service and does not provide any guarantee that data is delivered. This limits the possible usage scenarios to services where a lost byte here and there are insignificant. If the scenario is to distribute audio or video streaming to multiple receivers in a local-area network, IP-layer multicast is probably the best known solution. There exists research into making IP-multicast transfer reliable, but none of the efforts seem to have had any penetration into the existing market. If assuming a transfer reliable IP-layer multicast distribution, it could be used for solving the problem scenario, but every node that is not located in the local-area network would, however, still need to receive the data by Unicast methods. This would significantly slow down the distribution process.

Application-layer multicast

When distributing data with application-layer multicast each node that receives data also becomes an up-loader to the system. This greatly improves on the Unicast model and at the same time creating an alternative for IP-layer multicast. Application-layer multicast protocols use Unicast methods between hosts to emulate multicast capabilities. Because application-layer multicast uses Unicast to acheive multicast it works on the Internet. Additionally there often is no need for network configuration, making it easy to set up and use.

A common application-layer multicast protocol that use this method is BitTorrent [6]. For the given problem scenario BitTorrent would be a great candidate for distributing data efficiently between the nodes. BitTorrent is, however, not a perfect fit for the presented problem scenario. BitTorrent has much focus on seeding and choking strategies revolving around exploiters and free-riders [4]. Additionally research presented in [23] show there is much overhead traffic in the protocol associated with peer-discovery. It has also been shown that randomness of peer selection and locality-unawareness has significant impact on performance [44]. For the presented problem scenario none of these features are required and would lead to unnecessary performance loss. Additionally the scenario requires the receiving clients to install and reboot, which creates an issues with knowing when each node is done seeding.

BitTorrent is application centric, where it targets a specific need and does not suit the exact needs set by the problem scenario. This is quite common for application-layer multicast protocols, as there exists a plethora of different protocols where each tries to target the specific applications needs [20]. There has been much work done in the field of application-layer multicast protocols. The typical trend is, however, that application-layer multicast typically leads to compromises. There is often a sacrifice of efficiency for ease of deployment [16, 11].

1.2 Proposed solution

The basic of the idea is to use an application-layer multicast approach with a preplanned seeding strategy. In contrast to the application-layer multicast protocol BitTorrent, using a preplanned seeding strategy will reduce overhead associated with peer discovery. Additionally when the distribution is preplanned an optimal path can be chosen avoiding problems with locality unawareness. Lastly the receivers are considered trusted nodes, and no choking will be needed. The main goal is to create as little redundant network traffic as possible, and still maintain fast transfer speeds. This is thought to be achievable by adopting a snake like behavior when traversing the network. In other words trying to traverse the network by only traversing each network link only once, an illustration can be seen in figure 1.2 on the next page. It is thought that this distribution method could significantly improve data distribution speed, but there are some prerequisites that need to be in place. A mandatory perquisite for this distribution strategy is that the receivers are in a data subscription relationship, where the nodes are known and are waiting for the data to be distributed when it becomes available. Additionally the receives need to be trusted peers, there can be no unreliable receivers.

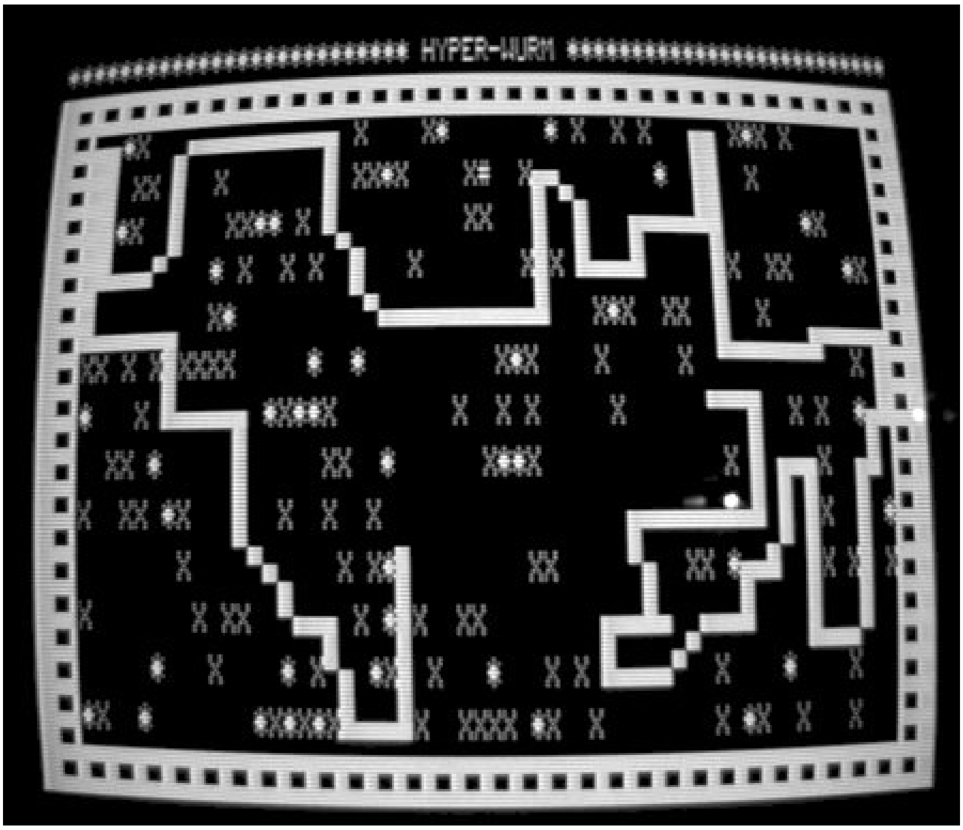


Figure 1.2: The proposed seeding strategy have similarities with the 1970s video game snake. The snakes goal is to eat food while traversing a world without hitting the walls or its own tail. As the snake eats food, the tail grows longer, making the game progressively more difficult. Damian Yerrick, 6 June 2007, *Snake on a TRS-80* [image online] Available at: http://en.wikipedia.org/wiki/Snake_(video_game) [Accessed 29 January 2012]

1.2.1 Solving the problem scenario

Using the preplanned distribution strategy it becomes important to utilize an optimal path through the network. A central feature that enables this is switch duplexing. A non-blocking full duplex Gigabit Ethernet switch which can handle 2 Gbit/s for each port. This means that the data can be served into port 1 traverse port 2,3,4,5 and then out port 1 again without any bottlenecks, see figure 1.3 for an illustration of the concept. If this assumption is true a possible optimal solution to the problem scenario could be

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \text{ then } 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \text{ and last } 9 \rightarrow 10 \rightarrow 11$$

For the given scenario Machine n would seed data to Machine $n+1$ until Machine $n+1$ reports it is done, signaling that Machine n can start installing. This effectively solves the seeding issue mentioned in section 1.1.1 on page 3.

Another path could be

$$8 \rightarrow 7 \rightarrow 6 \rightarrow 5 \text{ then } 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \text{ and last } 11 \rightarrow 10 \rightarrow 9$$

which is an equally good scenario. It is assumed that any ordering within one switch is equally fast. There is also very undesirable paths, traversing

$$11 \rightarrow 1 \rightarrow 10 \rightarrow 2 \rightarrow 9 \rightarrow 3 \text{ and so on}$$

would create significant redundant traffic between router 1 and router 2. These examples show that data distribution-time could be saved finding one of these optimal paths. There are more aspects to the proposed seeding strategy, see section 2.4 on page 26 for more details on buffer usage and end to end delay.

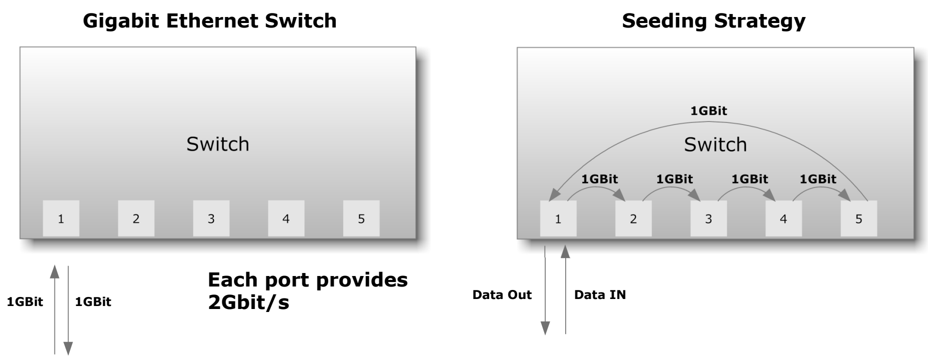


Figure 1.3: Duplexing enables the proposed seeding strategy.

1.2.2 Usage scenarios

In this section some relevant usage scenarios for the proposed distribution strategy will be presented.

Cloud infrastructure services

IaaS (Infrastructure as a Service) provides computing resources on demand. A key benefit of IaaS is the possibility to pay only for the resources that is being used, and one of these resources is bandwidth. Cost savings could be made if node data distribution could be planned. IaaS service Amazon EC2 [2] have different prices according to what region the traffic is routed, such that large number of server instances in different regions like USA and Europe should preferably not cross talk for optimal cost savings. An example on how the proposed model could be used in an IaaS distribution scenario can be seen in figure 1.4.

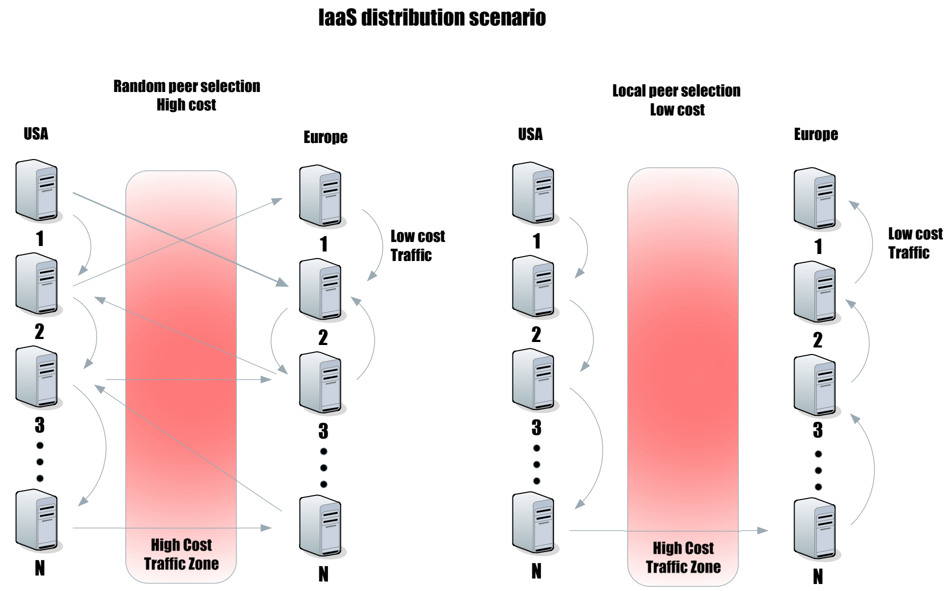


Figure 1.4: How local peer selection could save cost can be seen in this figure.

Not IP-layer multicast configured networks

Many local-area networks are not configured for IP-layer multicast. The motivation or knowledge to configure the network may be lacking. The proposed solution is intended to enable fast multicast deployments without any network configuration.

Not IP-layer multicast compliant networks

Some specialized high performance network equipment used in Internet backbone routers are not designed to support complex services such as IP-layer multicast. These routers make significant sacrifice of features in favor of performance [11]. In such conditions IP-layer multicast just does not apply, and application-layer multicast becomes the only option. The low redundancy design also assures that as little as possible of the bandwidth for other services is affected. An example of such a distribution scenario is presented in figure 1.5.

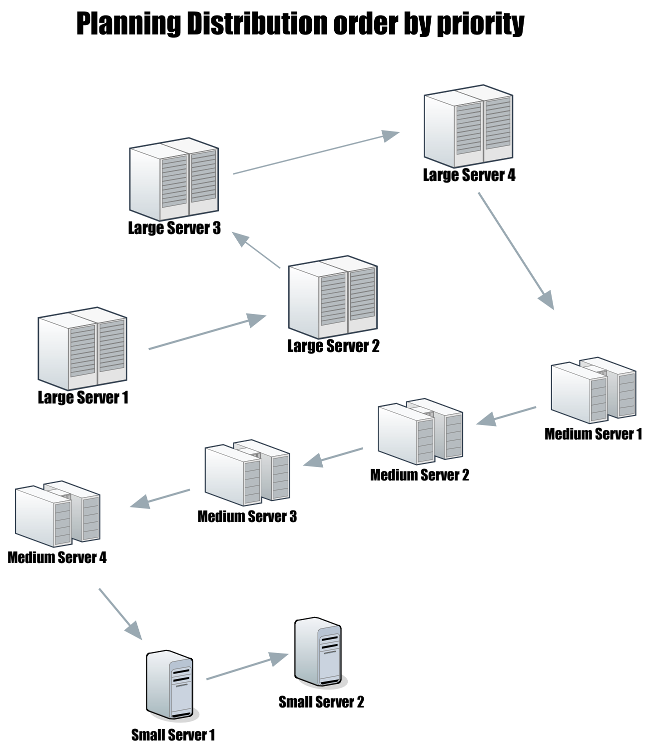


Figure 1.5: Distribution among large servers could be prioritized such that the most important nodes will receive the data before the less important nodes. An example of this type of content distribution could be movies shared amongst servers hosting video on demand services.

Internet roll-out services

The distribution strategy could be suitable for an Internet roll-out services, similar to Rocks Cluster Distribution [38]. An example of a roll-out service is illustrated in figure 1.6.

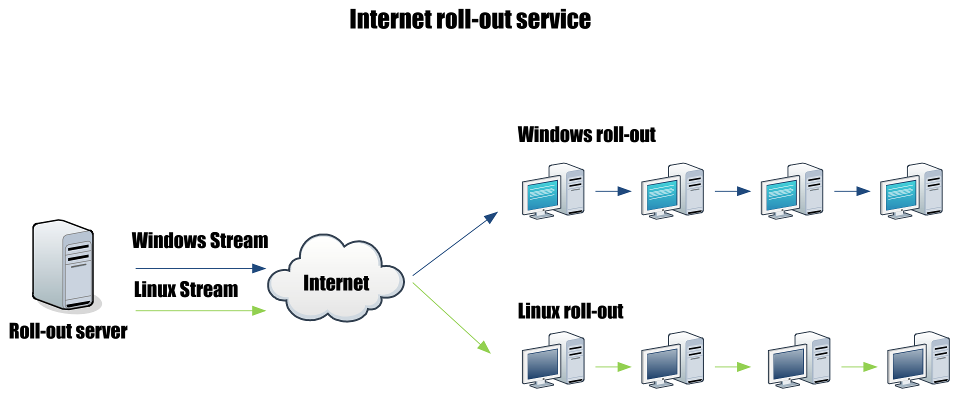


Figure 1.6: An example of how a roll-out service could be handled by the proposed distribution method. An Internet roll-out service could have a single streams for each roll-out site. Different operating-systems, distributions and versions of them would also need separate data streams.

1.2.3 Expected weaknesses

- **Wireless access-points**
  Wireless access-points could be a likely weak-point with the given seeding strategy. Given the usually slow bandwidth of wireless access-points compared to wired it is reasonable to assume wireless access-points will become a choke point. It is also expected that the data-amount combined with duplexing will create interference, further increasing the choke.

- **Scaling**
  In the 1970s snake game in figure 1.2 on page 5 the length of the snake dictates the difficulty of maintaining the current state and finding new good paths. It could be an equal scenario where it could be progressively increased difficulty of maintaining the seeding stream with increase in node count for the given distribution strategy. A slow or non functional node in the middle of a seeding chain could stop or reduce convergence-time considerably. Path recalculation and on-fly change might be needed for proper robustness.

- **Asynchronous bandwidth choke points**
  If there is asynchronous bandwidth choke points a one data-stream approach is not suited. There will be significantly reduced bandwidth utilization compared to distribution methods with multiple redundant data-streams.

1.3 Problem statement

The first goal of this thesis is to explore the proposed application-layer multicast distribution strategy, and find out if the distribution strategy could improve the speed of data distribution between nodes in network and system-administration relevant scenarios. Another goal is to try to remove all redundant data traffic from all the network links by traversing each network link only once. Effectively enforcing a one data stream policy to reduce the network footprint. Lastly the protocol should be applicable to both local network and the Internet as they are wired today.

The main idea is that this non redundant data transfer could be achievable by first discovering the network topology and then traversing the network with a snake like behavior, i.e. traversing each network link only once. The known challenges to this approach is that there are not many good mechanisms to discover the required network topology, and there needs to be a mechanism to find the optimal path when the topology is found. The problems with network discovery and optimal path calculation will not be solved in this thesis, the main goal is to explore the viability of the distribution strategy. To answer this a proof of concept model will be developed and benchmarked. To give the benchmarked results some meaningful context a comparative analysis between the proof of concept model and BitTorrent will be done. Additionally the following research questions are important for the work in this thesis.

- Application-layer multicast protocols relies on end hosts duplexing transferred data to achieve multicast. What impact does duplexing have on switch and end host performance?

- The proposed application-layer multicast protocol uses an optimal path and only one data-stream. How does such a transfer method compare in performance to random peer selecting and multiple data-stream multicast protocol such as BitTorrent?

1.4 Approach

The problem statement will be solved by two main approaches, in this section an overview is presented.

1.4.1 Finding the roof performance

Before setting up the prototype it is important to find the theoretical throughput performance. Overhead, delay and packet loss are some important factors that need to be accounted for before any benchmarking results can be assessed. Additionally the performance of the equipment that will be used needs to be tested. A large portion of this thesis will focus on measuring transport protocol throughput performance. It then becomes essential to uncover how to performance benchmark different transport protocols and the underlying network equipment. These measurements will have two purposes. First, is to uncover the roof threshold for expected performance of the developed prototype. Second, is to confirm that the proposed distribution method is possible on standard networking equipment. Getting this data will give important information about what the performance roof is, and it will in the end be the measuring stick for how well the prototype application-layer multicast protocol performs.

1.4.2 Comparative benchmark of prototype

A proof of concept model based on the proposed seeding strategy will be created. This prototype will be benchmarked according to CPU, disk and network usage. Additionally BitTorrent will be benchmarked in the same distribution scenarios. This is primarily done to give the performance measurements a reference point.

1.5 Main contributions

Key contributions are

- Finding out if the proposed distribution strategy is possible from a throughput perspective.
- Presenting the practicality and usage scenarios for the proposed distribution strategy.
- Determining the theoretical and practical roof performance of distributing data when using duplexing methods.
- Proving the concept by developing and benchmarking a working prototype.

All contributions are targeted at a single end goal, which is an attempt to improve performance of data distribution for networks with data subscription oriented nodes.

1.6 Thesis outline

The thesis is organized in the following manner:

- **Introduction**
  A short overview of project topics and its relevance. The motivation for the project is given, and the problem statement is defined in this section. Additionally a brief overview of how the problem statement will be solved is given.

- **Background and previous work**
  The theory behind benchmarking transport protocols is presented here. Placement of the thesis work within existing research will also be done here.

- **Experimental design and methodology**
  Describes how the experiments was designed to answer the problem statement, and the reasoning why these methods where chosen.

- **Results**
  The scripts and the prototype will be presented in this chapter. Additionally the results from the roof performance tests and the comparative benchmarks are presented here.

- **Analysis**
  This chapter contains the interpretation of the results.

- **Discussion and Conclusion**
  Repeatability of the experiments, likelihood of errors in the data and the viability of the results are discussed in this chapter. Lastly the thesis conclusion is presented.

Chapter 2

Background

Since a large portion of this thesis will focus on measuring transport protocol throughput performance, research into how to benchmark transport protocols had to be done. Some of this research is summarized here into the background section and is the foundation for the interpretation of the results. Additionally placement of where the work in this thesis fits within existing research will be presented in this chapter. Lastly some additional insight into the importance of buffer usage and end to end has on the proposed seeding strategy will be presented.

2.1 Placement within existing work

2.1.1 Multicast design

There exists multiple recent surveys into multicast protocols, where [11, 16] are recommended. In this section a brief and general overview is presented.

Multicasting can be segmented into primarily 3 different approaches.

- **IP-layer multicast** Is multicasting where the network devices replicates packets and sends them to a group of computers. This approach is a "best effort" service and does not provide any guarantee that data is delivered. This limits the possible usage, and it is best suited for video and audio streaming where a lost byte here and there are insignificant. There exists research into making IP-layer multicast transfer reliable, but none of the efforts have had any penetration into the existing market.

- **Application-layer multicast** Instead of replicating data at the network level, application-layer multicast utilizes end hosts to replicate and re-upload data to other hosts. This distribution method is often called peer to peer protocols.

- **Overlay multicast** Overlay multicast means that an overlay network topology is created to accommodate or improve on current multicast approaches. An example of this could be to implement IP-layer mulicast serves at different remote sites, acting as islands, which internally sync data and multicast at their respective sites.

The approaches has different strengths, the overall results from the surveys can be presented in the following table.

Metric	IP-layer	Application-layer	Overlay
Ease of deployment	low	High	medium
Bandwidth and delay efficiency	High	low	medium
Overhead efficiency	High	low	medium

Table 2.1: A highly generalized overview of the different multicast approaches.

The aim of the work in this thesis is to improve on the bandwidth, delay and overhead associated with application-layer multicast. Since there will be no work done on optimal path finding and network discovery, this will sacrifice the ease of deployment. The prototype target is to achieve the following metrics, see table 2.2.

Metric	Prototype target
Ease of deployment	medium
Bandwidth and delay efficiency	High
Overhead efficiency	High

Table 2.2: Without automatic network discovery and optimal path calculation this table represents the target metrics for the prototype.

Further work into optimal path calculation and network discovery is thought to be able to improve the ease of deployment. Since the distribution strategy is only intended for data subscription oriented nodes, the usage scenarios will, however, be limited.

2.1.2 BitTorrent performance

A deep analysis of the performance of BitTorrent is not presented in this thesis, it is used primarily as a reference point. The piece picker is a central component in BitTorrent implementations, where the strategy is to find and seed the rarest pieces into the swarm. This strategy has proven itself to be quite efficient, and BitTorrent has become a popular data distribution protocol. For more information on BitTorrent performance see survey [44].

2.1.3 Tandem queue

To utilize a tandem queue for data distribution is not new. Tandem queuing is central for all network routing. What is new for this thesis is to explore if there is any benefit to chose a tandem queue path that conforms with network junction-points such as switches and routers.

2.1.4 Network discovery and path calculation

It is thought that the primary reason for that the proposed distribution strategy has not been done before, is that there does not exist a good way to automatically discover the link-layer network devices. The intention is to use a link layer network discovery methods together with optimal path calculator to create data seeding maps for the proposed distribution strategy.

There does exists some methods to do some link-layer discovery today. Etherbat [10] for instance uses MAC spoofing to create invalid paths in the network, probes how it changed by injecting specially crafted ARP requests and checks for replies or absence of them [10]. There is also work on creating a dedicated protocol, LLTD [17] (Link Layer Topology Discovery) by Microsoft, LLDP (Link Layer Discovery Protocol) defined in IEEE 802.1AB and there exists vendor proprietary protocols such as CDP (Cisco Discovery Protocol). If LLTD, LLDP or other protocols such as CDP saturates the market, there could be promising alternatives for future implementation. There will not be done any attempt at network topology discovery in this thesis, the prototype will rely on the user to input the network topology manually.

When the network topology is found, it is though that optimal path calculation can be simplified significantly by grouping the nodes by which network junction-point they are connected to. This can be done because it is hypothesized that the distribution ordering within a switch will not affect performance. There will not be done any attempt at optimal path calculation in this thesis, the prototype will rely on the user to input the path manually.

2.2 Bits and bytes

In network communications there is a history of using the SI standard notations for Kilo, Mega, Giga and so on, where the multiplier is a decimal base. When measuring storage the the multiplier can sometimes have a binary base, which could create confusion as to the actual size. To not have any confusion, a short abbreviations table explaining the data size values used in this document was created, see table 2.3. The values in the table can be prepended to a per time unit notation, where Mbps is Megabit per second.

Abbreviation	Name	Value
b	bit	$1 \vee 0$
B	byte	8 bits
kb	kilobit	10^3 bits
Mb	Megabit	10^6 bits
Gb	Gigabit	10^9 bits
KiB	Kibibyte	2^{10} bytes
MiB	Mebibyte	2^{20} bytes
GiB	Gibibyte	2^{30} bytes

Table 2.3: A short data size abbreviations list.

2.3 How to benchmark using transport protocols

In this section theory behind how to benchmark and measure transport protocols will be presented. This theory will also be the basis for how to interpret benchmark results of network devices when using transport protocols as a benchmark tool. Ideally when benchmarking network devices proper benchmarking hardware such as the one seen in figure 2.1 on the next page would be the best tool for the job. Not having access to specialized hardware there is the alternative to benchmark with regular transport protocols such as TCP and UDP.

When benchmarking network device throughput using common network transport protocols such as TCP and UDP, there are several factors that are important to beware of. The actual effectiveness of a network transport protocol is reliant on several factors like protocol overhead, congestion control, packet loss, maximum bottleneck bandwidth and propagation-delay [32]. When the network devices reach its limits, it is nice to know what performance numbers are expected at the application-layer. Important factors that needs to be considered when benchmarking using network transport protocols are presented here. The prototype in this thesis is developed using TCP, making the main focus in this section biased towards TCP throughput performance.

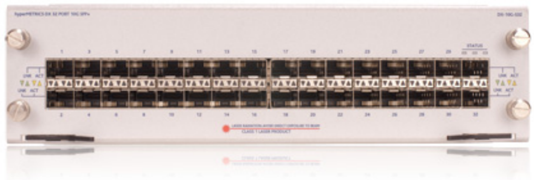


Figure 2.1: Using specialized hardware is preferable for high bandwidth testing. This hardware module from Spirent is created to be able to fully saturate high performance Ethernet network devices to their maximum capacity. *HyperMetrics dX 32-port 10G Ethernet test module* [image online] Available at: http://www.spirent.com/Solutions-Directory/Spirent-TestCenter/HyperMetrics_dX [Accessed 1 March 2012]

2.3.1 Overhead

When network devices communicate, protocols are needed for knowing things like where to send the data, where it came from, what bits are coming now and in what order. These protocols are essential for interpreting the data that are being transmitted. For benchmarking it is important to beware of the overhead that these protocols have, which will use up bandwidth, reducing the effective data payload size intended for the receiver. When reading the throughput measurements using transport protocols it is only the effective data payload which is read. It then becomes essential to account for overhead bytes before the number of bytes transferred and the actual network device performance can be determined. Different protocols have different overhead, making the possible combinations fairly large. In this text, the scope is limited by the protocols that is relevant for this thesis, which are TCP, UDP, Ethernet and IP.

Starting from the bottom, an Ethernet frame consists of header, CRC (Cyclic Redundancy Check) and payload, but on the physical link it also has a gap and preamble. Combined the Ethernet frame overhead is 12 gap + 8 preamble + 14 header + 4 CRC = 38 bytes for each frame [14]. The Ethernet payload is often referred to as the MTU (Maximum Transmission Unit) which for the original Ethernet standard is 576 bytes. The old Ethernet standard was superseded by Ethernet v2 [37] which had a standard MTU size of 1500 bytes, this decreased the associated overhead significantly. Today it is being replaced by the Gigabit Ethernet standard, which has support for "jumbo frames" with an MTU of 9000 bytes, further decreasing the overhead [25]. Enabling larger frames will decrease overall overhead, but for benchmarking purposes it will only change the target performance number slightly, which serves no purpose for the end result.

The MTU bytes are, however, not the effective payload size, the Ethernet frame also needs to carry both the IPv4 header and the transport protocol header. The TCP and IPv4 headers are not fixed in stone, requiring some extra attention when counting overhead bytes. Using IPsec [26] for instance adds 4 extra overhead bytes to the IP datagram, and timestamps option for TCP will add 12 extra overhead bytes to the TCP header. Both IP and TCP has options that make them range from 20 to 60 bytes each [36]. Taking the advised optimistic position defined in [36] the IPv4 , TCP , UDP headers are usually 20, 20 and 8 bytes respectively [34, 35, 33]. This leaves 1460 bytes for effective data when using TCP and 1472 bytes for UDP. For TCP this remaining payload can be referred to as the MSS (Maximum Segment Size) [36]. MSS is not defined for UDP, but for all intents and purposes its data must also fit into this window.

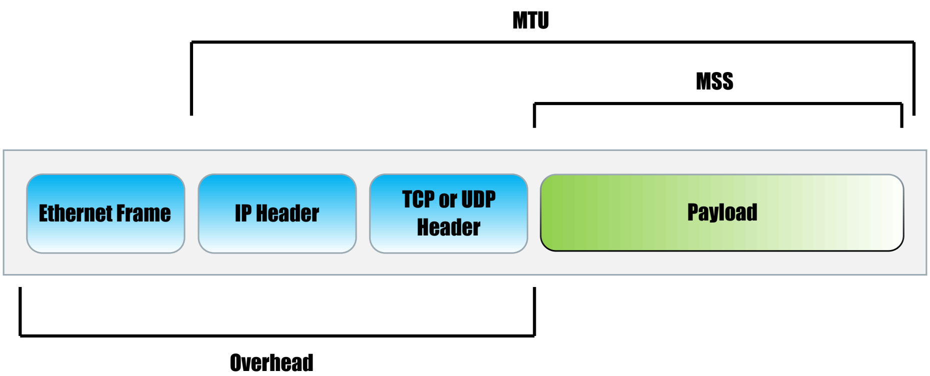


Figure 2.2: An example illustration on how network overhead eats up network bandwidth. The purpose of this illustration is to show that it is important to account for the bandwidth lost to overhead when performance benchmarking network devices.

Knowing the overhead that is associated with the protocols, the network efficiency can be calculated. It is these calculations which represents the performance targets for the benchmarks. The bandwidth efficiency can be expressed with the following equation

$$\frac{\text{MTU} - \text{IP header} - \text{Transport protocol header}}{\text{MTU} + \text{ Ethernet frame}} \cdot 100 = \text{Packet efficiency } \% \tag{2.1}$$

With an MTU of 1500 and using the optimistic values for transport protocol overhead, this would give the efficiency results of 94.9285% for TCP and 95.7087% for UDP. For a 1000 Mbps line this would translate to a theoretical maximum throughput of ~949 Mbps and ~957 Mbps for TCP and UDP respectively. See table 2.4 on the following page for more examples on common overhead combinations. These values are considered the ideal throughput values, as it considers only protocol overhead and disregards packet-loss and propagation-delay [32].

MTU	IP	Transport	Options	Calculation	Efficiency
9000	IPv4	UDP	None	$\frac{9000-28}{9000+38}$	99,2697 %
9000	IPv4	UDP	VLAN	$\frac{9000-28}{9000+42}$	99,2258 %
9000	IPv4	TCP	None	$\frac{9000-40}{9000+38}$	99,1370 %
9000	IPv4	TCP	VLAN	$\frac{9000-40}{9000+42}$	99,0931 %
9000	IPv6	UDP	None	$\frac{9000-48}{9000+38}$	99,0485 %
9000	IPv6	UDP	VLAN	$\frac{9000-48}{9000+42}$	99,0046 %
9000	IPv4	TCP	Timestamp	$\frac{9000-52}{9000+38}$	99,0042 %
9000	IPv4	TCP	Timestamp and VLAN	$\frac{9000-52}{9000+42}$	98,9604 %
9000	IPv6	TCP	None	$\frac{9000-60}{9000+38}$	98,9157 %
9000	IPv6	TCP	VLAN	$\frac{9000-60}{9000+42}$	98,8719 %
9000	IPv6	TCP	Timestamp	$\frac{9000-72}{9000+38}$	98,7829 %
9000	IPv6	TCP	Timestamp and VLAN	$\frac{9000-72}{9000+42}$	98,7392 %
1500	IPv4	UDP	None	$\frac{1500-28}{1500+38}$	95,7087 %
1500	IPv4	UDP	VLAN	$\frac{1500-28}{1500+42}$	95,4604 %
1500	IPv4	TCP	None	$\frac{1500-40}{1500+38}$	94,9285 %
1500	IPv4	TCP	VLAN	$\frac{1500-40}{1500+42}$	94,6822 %
1500	IPv6	UDP	None	$\frac{1500-48}{1500+38}$	94,4083 %
1500	IPv6	UDP	VLAN	$\frac{1500-48}{1500+42}$	94,1634 %
1500	IPv4	TCP	Timestamp	$\frac{1500-52}{1500+38}$	94,1482 %
1500	IPv4	TCP	Timestamp and VLAN	$\frac{1500-52}{1500+42}$	93,9040 %
1500	IPv6	TCP	None	$\frac{1500-60}{1500+38}$	93,6281 %
1500	IPv6	TCP	VLAN	$\frac{1500-60}{1500+42}$	93,3852 %
1500	IPv6	TCP	Timestamp	$\frac{1500-72}{1500+38}$	92,8479 %
1500	IPv6	TCP	Timestamp and VLAN	$\frac{1500-72}{1500+42}$	92,6070 %
576	IPv4	UDP	None	$\frac{576-28}{576+38}$	89,2508 %
576	IPv4	UDP	VLAN	$\frac{576-28}{576+42}$	88,6731 %
576	IPv4	TCP	None	$\frac{576-40}{576+38}$	87,2964 %
576	IPv4	TCP	VLAN	$\frac{576-40}{576+42}$	86,7314 %
576	IPv4	TCP	Timestamp	$\frac{576-52}{576+38}$	85,3420 %
576	IPv4	TCP	Timestamp and VLAN	$\frac{576-52}{576+42}$	84,7896 %

Table 2.4: In this table some efficiency values for common combination of protocol settings are listed. IPv6 does not have support for MTU less than 1280 [27], and therefore is not included in the bottom part of the table. The table is sorted by the efficiency value.

2.3.2 Bandwidth-delay product

When a network signal is sent it has to propagate through the network. This propagation speed to a node and back again is called RTT (Round Trip Time). This RTT value is important for window based protocols like TCP. TCP window size is the amount of data a sender can send to a receiver without the receiver having to acknowledge the data. This means that if the TCP window size is 65535 bytes, a sender could put 65535 bytes on the link before stopping and waiting for an acknowledgment of the received data.

This brings us back to the network propagation speed. It takes time to propagate both the bytes and the acknowledgment. This delay makes it such that there will be bytes on the physical network that has been transmitted but not yet been received. The size of how many bytes that is in the state of transit is called the BDP (bandwidth-delay product) [32]. The BDP is the product of the bandwidth and the RTT. As an example, if the bandwidth is at 1000 Mbps for a server, and there is a 2 ms RTT between the sender and receiver the BDP would be

$$BD = 1000 \cdot 10^6\text{bps} \cdot 2 \cdot 10^{-3}\text{s} = 2000 \cdot 10^3 = 2000\text{kb or} \sim 244\text{KiB}$$

The problem with TCP throughput arise when the BDP is large, and is most common on networks with high bandwidth and RTT. These networks are called LFN (Long Fat Networks), which is fittingly pronounced "ele-phan(t)". There are also problems with large BDP in local area networks when there is need for high speeds. With a receive window of 65535 bytes, all the bytes intended for the receive window will be in transit when the sender stops and waits for an acknowledgment. This creates a gap where the line is not used and both the sender and the receiver just waits for data to propagate the network link, see figure 2.3 on page 23 for an illustration of the problem. The efficiency can roughly be express by $\frac{\text{RWND}}{\text{BDP}}$, where RWND is the size of the receive window. For the specific example it would translate to $\frac{65535}{250000} = 26,2\%$ efficiency. This example show that the RTT has significant impact on TCP throughput, and it demonstrates that using correct TCP window size is important for throughput.

Originally TCP only supported a maximum of 65535 byte window, and this was later improved, see [24] with the "TCP Window Scale Option", allowing for larger than 65535 byte windows. Although increasing the window size removes much of the problems with link utilization associated with LFN and high speed networks, another problem arises. With the larger window size there is a greater risk of unintentionally congesting the network, increasing both packet loss and retransmissions. This implies that a too large increase of the window would also be also be undesirable. How much is enough? The window size is dependent on the RTT, and the RTT is dependent on other traffic, queues and chosen route, implying that it variates. To solve this variation problem the general strategy is to use the highest expected RTT to calculate the BDP. This highest expected BDP value then can be used as the TCP receive window size, being large enough to not underutilized the network link, and small enough to not unnecessarily congest traffic. Many TCP implementations use RTTM methods (Round Trip Time Measurement), for adapting to changing traffic conditions and to avoid instability known as "congestion collapse" [24]. This measurement requires the use of the extra TCP timestamp option. This timestamp adds an additional 12 bytes of overhead to the TCP header.

Mostly when using TCP these settings are taken care of by the operating-system‘s TCP/IP stack. This does, however, have an impact on performance and must be known to get accurate benchmarks calculations. When allocating the window size on on Linux it is important to know that it allocates twice as much as requested [18]. The entire window is, however, not used for purely receiving data. TCP uses some of this extra space for administrative purposes and internal kernel structures [18], thus complicating the calculations further.

If the network BDP turns out to be unusually large, it would be better to test the path with multiple TCP connections. With a line width of 1000 Mbps and an RTT of 20 ms the BDP will be 20 000 kb or ~2,44 MiB. This amount of in-transit data would be to large for a single TCP connection to test reliably [32]. In table 2.5 the minimum required TCP connections with associated RWND sizes to fill the BDP of the previous example is listed.

TCP RWND	N connections
16 KiB	153
32 KiB	77
64 KiB	39
128 KiB	20
256 KiB	10

Table 2.5: The minimum required number of TCP connections required to fill a BDP of 2,44 MiB at different RWND sizes.

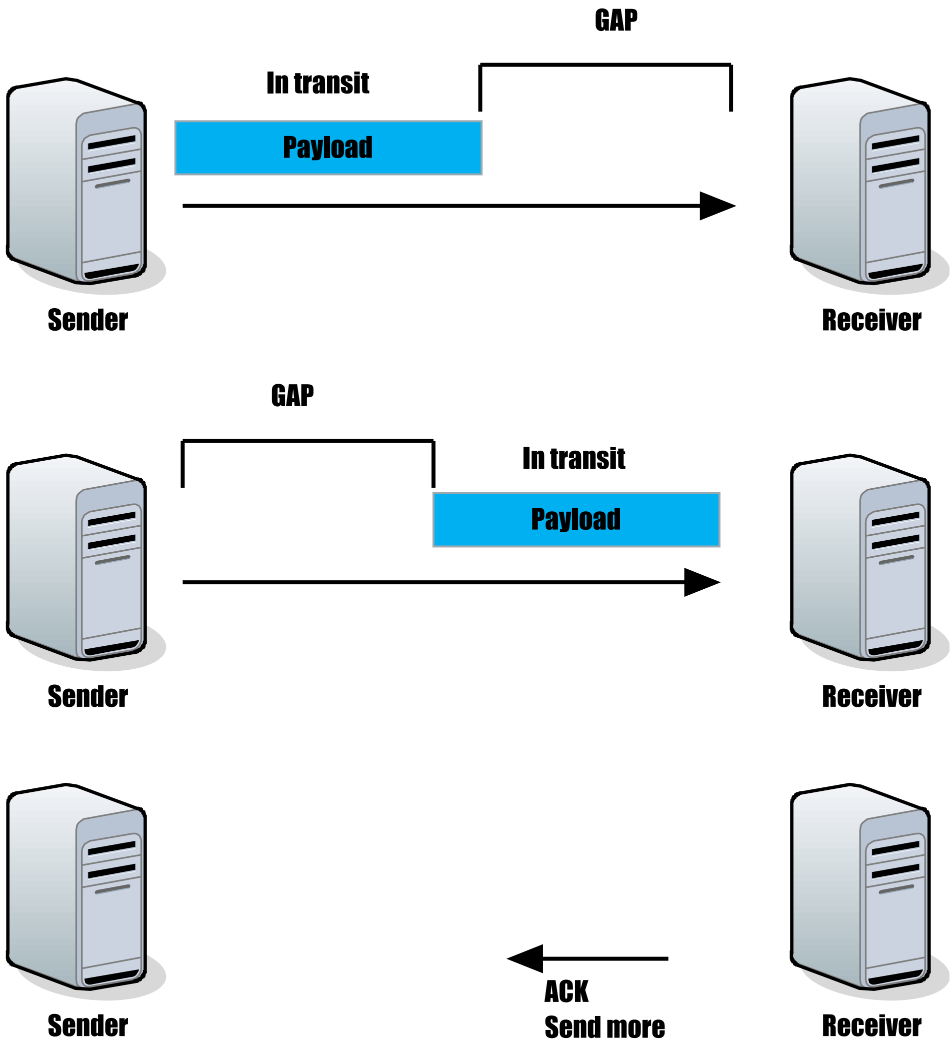


Figure 2.3: A simplified illustration of a large BDP and a small TCP window. In this example the sender prematurely fills up the entire TCP window before the receiver has received a single byte, resulting in the sender waiting for an acknowledgment before it can send more. For large data transfers this will severely impact the throughput performance.

The RWND and BDP values are the most important measures when considering TCP throughput performance. To show this importance an example table was made, see table 2.6. Not knowing these limitations could lead to errors when evaluating throughput performance values.

TCP RWND	Efficiency
16 KiB	13,1 %
32 KiB	26,2 %
64 KiB	52,4 %
128 KiB	100 %
256 KiB	100 %

Table 2.6: TCP throughput efficiency at different RWND sizes for a BDP size of 1000 kb. The BDP size is representative for line with 1000 Mbps bandwidth and 1 ms RTT.

2.3.3 Packet loss

Not all packets sent over a network reach their destination or arrive unscathed. There are a number of reasons for how a packet in transit either becomes corrupt or is lost entirely. Some common reasons being signal degradation, faulty hardware or congestion. TCP efficiency based on packet corruption and/or loss can be expressed by the following formula

$$\frac{\text{Packets sent - Packets retransmitted}}{\text{Packets sent}} \cdot 100 = \text{Efficiency }\% \tag{2.2}$$

2.3.4 Jitter

Jitter or also known as *packet delay variation*, is the difference in delay between successive packets in a data flow. Packet delay variation is important for applications with real-time voice and/or video applications, and is also important for understanding network queues as changes in delay can change the network queue dynamics [29]. For benchmarking purposes this value is important when measuring the expected quality of these real-time applications. Additionally the packet delay variation is important for the accuracy of throughput benchmarks, see section 2.3.8 on page 26 for more details.

2.3.5 Buffer delay

When running a TCP throughput test the RTT value might increase because of congestion created by traffic generated in the test. This increase in

RTT over the baseline RTT measured at non-congested conditions is called buffer delay [32]. As seen in section 2.3.2 on page 21, the RTT value is significant for the throughput performance of TCP and an increase in RTT could be significant for the end results. The buffer delay is calculated using the following formula

$$\frac{\text{Mean RTT} - \text{Baseline RTT}}{\text{Baseline RTT}} \cdot 100 = \text{Buffer delay } \% \tag{2.3}$$

Where the mean RTT value is based on the RTT values during transfer. How to measure the RTT value is defined in [32].

2.3.6 TCP equilibrium

TCP connections does not start out at maximum speed when a end to end connection is made. TCP goes through a build up process before it reaches a state called *equilibrium state*. A TCP connection goes through 3 distinct phases which is designed to ramp up throughput speed until packet loss and adjust speed accordingly. The phases are

1. Slow start phase
2. Congestion avoidance phase
3. Loss recovery phase

Some packet loss is expected in this build up process, as it is a natural result of the process of finding the throughput limit. Congestion control algorithms are a large subject, for information on how the phase processes achieve their purpose see [31]. This buildup process is relevant for the actual TCP performance, and will be a factor in TCP reliant applications. Since this process has the largest impact at the beginning of the TCP connection this slow start becomes less significant with increasing transfer size. Maximum throughput should therefore be measured when equilibrium state is reached.

2.3.7 Methodology

It is considered best practice to run full layer 2/3 tests such as described in [28] to verify the integrity of the network before running tests [32]. The test methodology can be summarized by the following three points

1. Identify the path MTU. See [30] for more information on MTU discovery methodology.
2. Find the baseline RTT and bandwidth. This step is used to provide estimates for TCP RWND size and send socket buffer size.
3. TCP connection throughput tests. Single and multiple TCP connections tests to verify the baseline network performance.

2.3.8 Accuracy of measurements

Generally it is considered not possible to make accurate TCP throughput performance measurements when the network is exhibiting unusually high packet loss and/or jitter. The guideline provided in [32] considers 5% packet loss and/or 150 ms jitter too high for accurate measurements. Because of the buffer delay, TCP throughput tests should not last less than 30 seconds, and it could be useful to test at different times of day when testing networks with underlying traffic [32].

2.4 Prototype model detailed

In the introduction section the basic idea of how to best traverse the network junction-points was presented. There are other factors that are important for the overall performance of the data distribution. Buffer usage and end to end delay is considered crucial subjects for the proposed seeding strategy. Before explaining how these metrics will affect performance a basic role definition will be presented.

2.4.1 Defining the prototype roles

At the most basic level the prototype can be seen as a tandem queue model. Tandem queue models in networks are often related to network routing, where the packets often must be be routed between two end points choosing the shortest or cheapest route. The prototype model is based on the same model, only that the aim is to visit all nodes. The prototype consists of three basic roles, which is the traffic generator, forwarder and receiver, see figure 2.4. It is the forwarding node which is of most interest, and it is the primary focus when discussing node performance in this thesis.

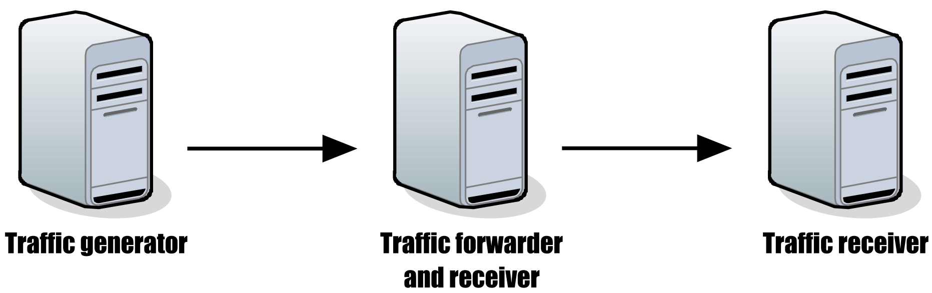


Figure 2.4: A tandem queue.

2.4.2 Buffer size

One of the aspects of the proposed model is to make use of the system buffer to utilize a temporary boost to performance until the system chokes. This choke point appears when the buffer runs out, and the system cannot receive any more data before memory space is made available. A forwarder node receive data at a specific rate, in queuing theory referred to as arrival-rate, λ. The forward node also needs to forward data, μ_f, and write to disk, μ_w. This creates a system with two effective queues, one for forwarding data and one for writing to disk. The prototype model is created such that the forward happens before the write to disk, such that the arrival-rate for the disk queue is dependent on forward data. The rate at which arrivals are served to the disk queue will be arrival-rate, λ_f. It is assumed that these internal dependent queues can be approximated using independent M/M/1 queuing models, as stated by Jackson's theorem. Some traffic shaping is expected, but it is assumed that the arrival-rates, forward-rate and write-rate can be approximated by a Poisson process. M/M/1 queuing theory states that theoretical forward utilization-rate, ρ_f, and write utilization-rate, ρ_w, can be express by the following equations

$$\rho_f = \frac{\lambda}{\mu_f} \tag{2.4}$$

$$\rho_w = \frac{\lambda_f}{\mu_w} \tag{2.5}$$

An important aspect of these equations are that queue explodes when $(\rho_w \vee \rho_f) > 1$. This is expected behavior and effectively means that the buffer usage increase as data is being transferred. When $\forall_x((\rho_f \wedge \rho_w) < 1)$ for a tandem queue consisting of x forwarding nodes, it is not likely there will be a significant buildup of queue, hence no large buffer needed. The point of the large buffer usage in the model becomes clear in the scenarios where $\exists_x((\mu_f > \mu_w) \wedge (\rho_w > 1))$, which is expected to be the norm. In the prototype it is the write-rate which is responsible for discarding queue items, and freeing up new queue spots. Assuming that $\mu_f > \mu_w$ this creates a relationship between the arrival-rate and the write-rate. This relationship is the choke point, at which point the system will have to deny any new arrivals, and the arrival-rate λ cannot become larger than the write-rate. The choke point z can be found by the following equation

$$\mu = \begin{cases} \mu_f & \text{if } (\mu_w > \mu_f) \\ \mu_w & \text{else} \end{cases}$$

$$z = \frac{q}{\lambda - \mu} \tag{2.6}$$

Where, q, is the size of the buffer. Examples of how much data can be transferred before the system chokes can be seen in figures 2.5, 2.6 and 2.7.

In the example figures the arrival-rate is fixed at 1000 Mbps and it is assumed that $\mu_w < \mu_f$.

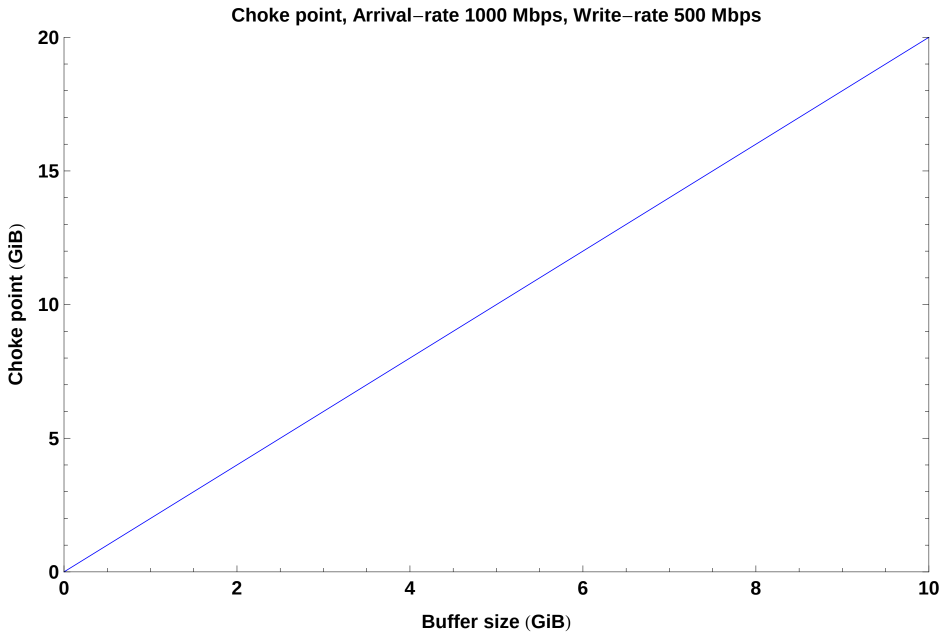


Figure 2.5: Choke point relative to the buffer size.

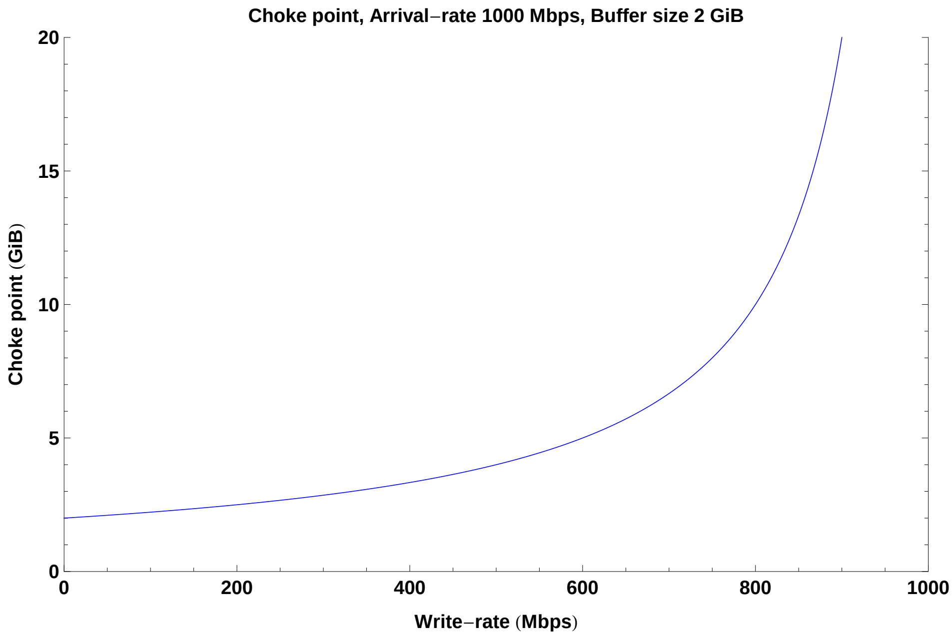


Figure 2.6: Choke point relative to the write-rate.

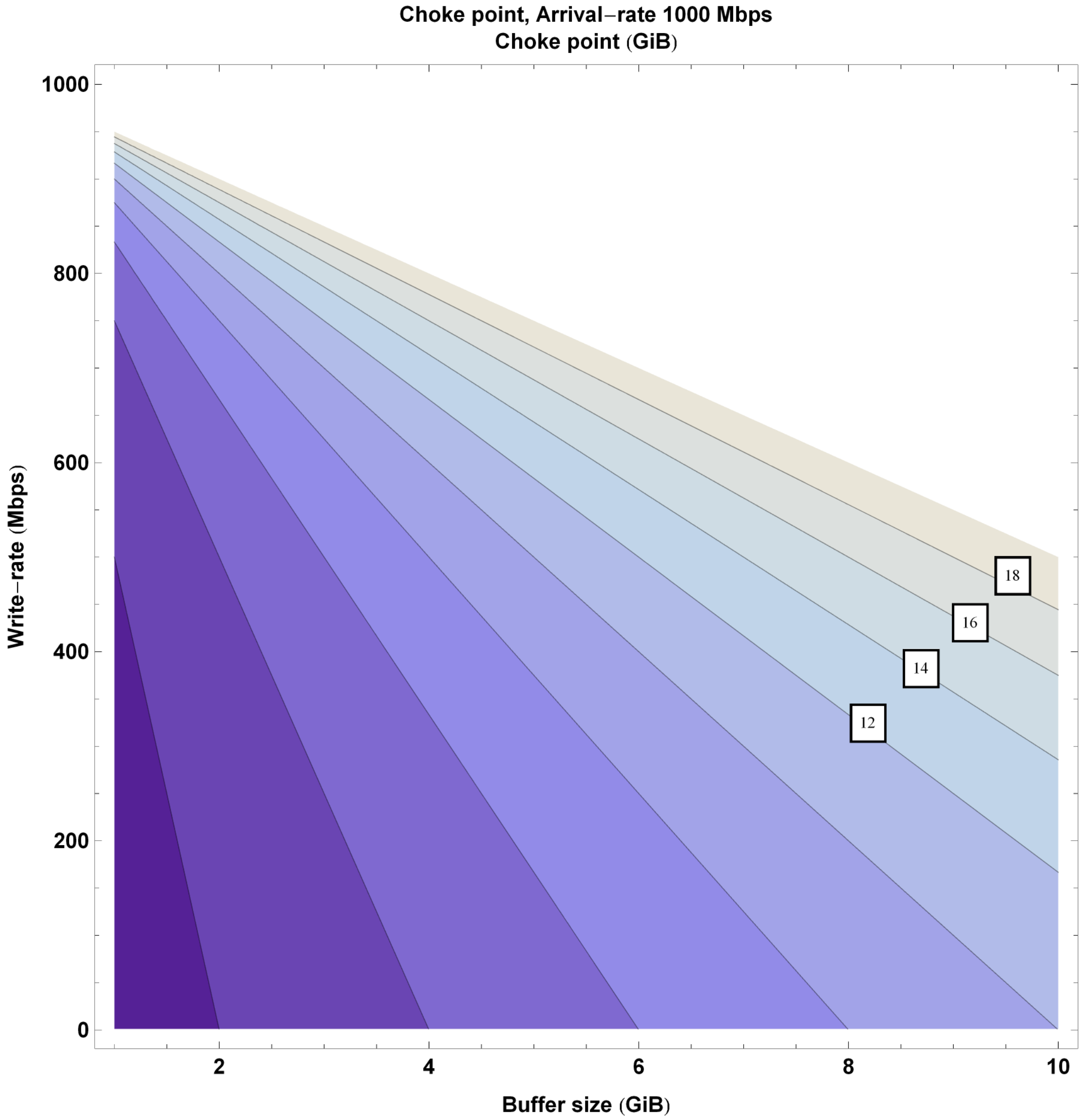


Figure 2.7: Choke point relative to write-rate and buffer size.

2.4.3 End-to-end delay and job size

Since the prototype model is a tandem queue there will be an end-to-end delay, which is the time the data needs to propagate from the first to the last node. This end-to-end delay is important for the scalability of the proposed distribution strategy. The end-to-end delay is the sum of all the nodal delays. Between each node there will be processing delay, d_p, queuing delay, d_q, transmission delay, d_t and propagation delay, d_f. The end to end delay can then be described by the following formula

$$\text{end-to-end delay} = \sum_{i=1}^{N-1} (d_p(i) + d_q(i) + d_t(i) + d_f(i)) \tag{2.7}$$

Where N is the number of nodes. It is important to notice that the routers, and switches separating the nodes also have the same delay properties. In equation 2.7 it is assumed to fall under the propagation delay between nodes. It is expected that the processing delay and the transmission will become the largest bottleneck in the distribution. It is important to notice that the end-to-end delay can create a situation where the first node has sent all data but has not been received by the last receiving node yet. This creates a scenario where it might be beneficial to divide the nodes into segments which will receive the stream in turns. If the data-stream reach 10 nodes as the original sender sends its last byte, it might be possible to segment the receiving nodes into 10 and 10 nodes. Additionally this segmentation of nodes could be combined with buffer choke avoidance, alternating between segmented nodes as the node buffer becomes saturated. These scenarios are interesting for future research, but are not out of scope for this thesis.

In the prototype model the transmitted data will be pushed into a job which contains the data buffer. The size of this buffer is important for the end-to-end delay. A job needs to be filled up before it can be forwarded, creating a transmission delay, also called the *store-and-forward delay*. The transmission delay is expected to scale linearly with increasing job buffer size, significantly increasing end-to-end delay with increasing N. Decreasing the job buffer size will increase processing overhead, as the system needs to create more jobs, queue more items and do conditionals more often, which all adds up to requiring more system resources. For fast convergence it becomes important to have as little job size as the system allows without loosing throughput caused by node resource usage. The job buffer size might have a significant role in the performance of the proposed distribution model. This implies that the system will benefit from the possibility of specifying the job buffer size, such that the transmission can be optimized according to need.

Since the job buffer size can be seen as analogous to packet-switched network delays, some of the theory in this section is based on the overview of delays in packet-switched networks presented in [15].

Chapter 3

Experimental design and methodology

3.1 Finding the roof performance

Before setting up the prototype benchmark it is important to measure the performance of the equipment that will be used in the final experiments. Getting this data will give information about what the performance roof is, and it will in the end be the measuring stick for how well the prototype application-layer multicast protocol performs. These benchmarks will be used to confirm that the proposed distribution method is possible.

3.1.1 Collecting performance data

Switch throughput performance is central for the proposed seeding strategy, therefore a good benchmarking tool will be needed. Iperf [13] was chosen as the benchmarking software to use. Iperf is developed my NLANR/DAST, and the primary function of the software is measuring the maximum TCP and UDP bandwidth performance of a network link. The variables that Iperf can report that is of interest are bandwidth, delay jitter and datagram loss. Iperf does not, however, provide with the possibility of collecting performance data from multiple simultaneous benchmarks. A wrapper to Iperf will therefore be created to get this required functionality.

3.1.2 Methodology

The first benchmark will be a baseline test with a single Cat6 cable between two nodes. The purpose of this baseline test is to uncover any potential issues with network cards or general node performance.

The full test routine designed to test the network duplexing performance can be described as $f(1,2), f(2,3) \ldots f(n-1,n), f(n,1)$ where the function

$f(x, y)$, is the logical statement: "An Iperf benchmark is run from node x to receiving node y". See figure 3.1 for an illustration of the experimental setup. This routine will effectively create a benchmark loop, such that both up and down speed will be maxed out for every node. There will be a few seconds interval between each individual node benchmark. This delay is introduced to get a more accurate measurement of an eventual choke point. SSH will be used to orchestrate the benchmarks and to collect results. The individual benchmarks will be aggregate into a single plot. The plot aggregation and benchmark orchestration requires time synchronization for accurate measurements. Synchronization to an NTP (Network time protocol) server will therefore be required before any benchmarks can be run.

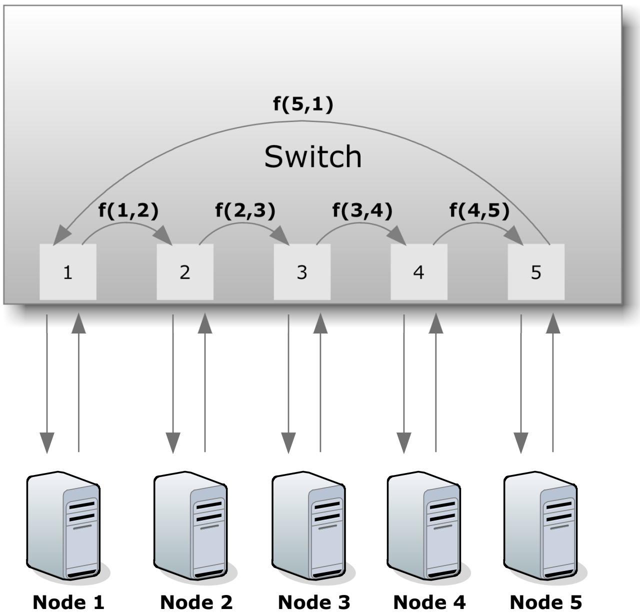


Figure 3.1: An illustration of the experimental setup for the roof performance experiments.

The prototype will be created using TCP, therefore the most important measurement will be the TCP throughput. There is a problem, however, Iperf does not support collection of packet loss or jitter when when running TCP benchmarks. Iperf does, however, support these measurements in the UDP benchmark tests, and therefore this issue will be solved by also running full UDP benchmarks.

The results from the full test routine will be measured against the baseline test using inferential statistics. It is not known if the switches will improve or degrade over the baseline performance, therefore a two tailed test will be performed to get a more rigorous statistic. The sample count is expected to be high, making it reasonable to assume that the sampling distribution of the sample mean to be normally distributed. This implies that the inferential test can be carried out by using a Z statistic. The following hypotheses statements are tested

H_0: *There is no qualitative difference between the baseline and the switch mean performance.*

H_1: *There is a qualitative difference.*

The H_0 will be assumed to be true. The significance level, α, of the test will be set to 0.05 (5%). If the P value returned by the Z-test returns a P value such that $P < \alpha$ the H_0 hypothesis will be rejected.

3.2 Prototype architecture

The architectural design of the prototype is presented in this section.

3.2.1 Libraries

The main functionality of the program will be crated using parts of the C++ Boost library [3]. The following libraries will be used.

- **Boost Asio** A cross-platform C++ library for network and low-level I/O programming.
- **Boost Program Options** A program options library that allow fetching of command-line and configuration file options
- **Boost Thread** A library that enables the use of multiple threads of execution with shared data.

3.2.2 Header

Designing an application-layer protocol always introduce extra overhead. To keep this overhead to an absolute minimum, this header is included at the start of the file transfer and is not introduced again. This header consists of 14 bytes of obligatory data and 0 to 1024 bytes for the variable length filename. After the filename is sent the file transfer starts and can be from 0 to ~8 exabytes of data. The following table show the file transfer byte-stream

<table>
<tr><td>Byte offset</td><td>0 - 2</td><td>3 - 6</td><td>7 - 14</td></tr>
<tr><td>0</td><td>Job size</td><td>Filename length</td><td>File size</td></tr>
<tr><td>15
to
1038</td><td colspan="3">Filename</td></tr>
<tr><td>Max ~8 exabytes</td><td colspan="3">File Data</td></tr>
</table>

Table 3.1: The structure of the byte-stream created by the prototype when initiating a file transfer.

3.2.3 Concurrency design

The prototype will be designed for concurrency, such that nodes with multiple processors and/or cores can utilized. The internal architecture will be created using task-servers, where each worker-thread has the responsibility of accomplishing the tasks within its assigned task-server. The internal task-servers can be divided into read, receive, send and write. Where the traffic generator "reads and sends data", the forwarder "receives, sends and writes" and lastly the receiver node "receives and writes". The task-servers for the forwarding node are illustrated in figure 3.2 on the next page. In the figure each column illustrates the tasks assigned to a worker-thread. A task must not be confused with a job. The job is the class containing the memory buffer that holds the transmission-data.

These task-servers are run in the same process such that they can share memory. This shared memory allows for sharing of job queues, and it allows the jobs to be moved and not copied from one task-server to another. This is accomplished using C++ 11 Move semantics. The key benefit from using Move semantics is that the jobs are not copied, it is only the ownership of the job that is transferred between the task-servers. The same principle is used to move jobs into and out of the queues, such that a job is only written to memory once during its lifetime in the process. This design is thought to be essential for minimizing the *store and forward delay*.

After the data has been transferred the task-servers are terminated by creating a kill-job. The kill-job is created after the last job containing transmission-data is sent. When a kill-job is received, the task-server will start shutdown procedures and the worker-threads rejoins the main-thread before the process is finally terminated.

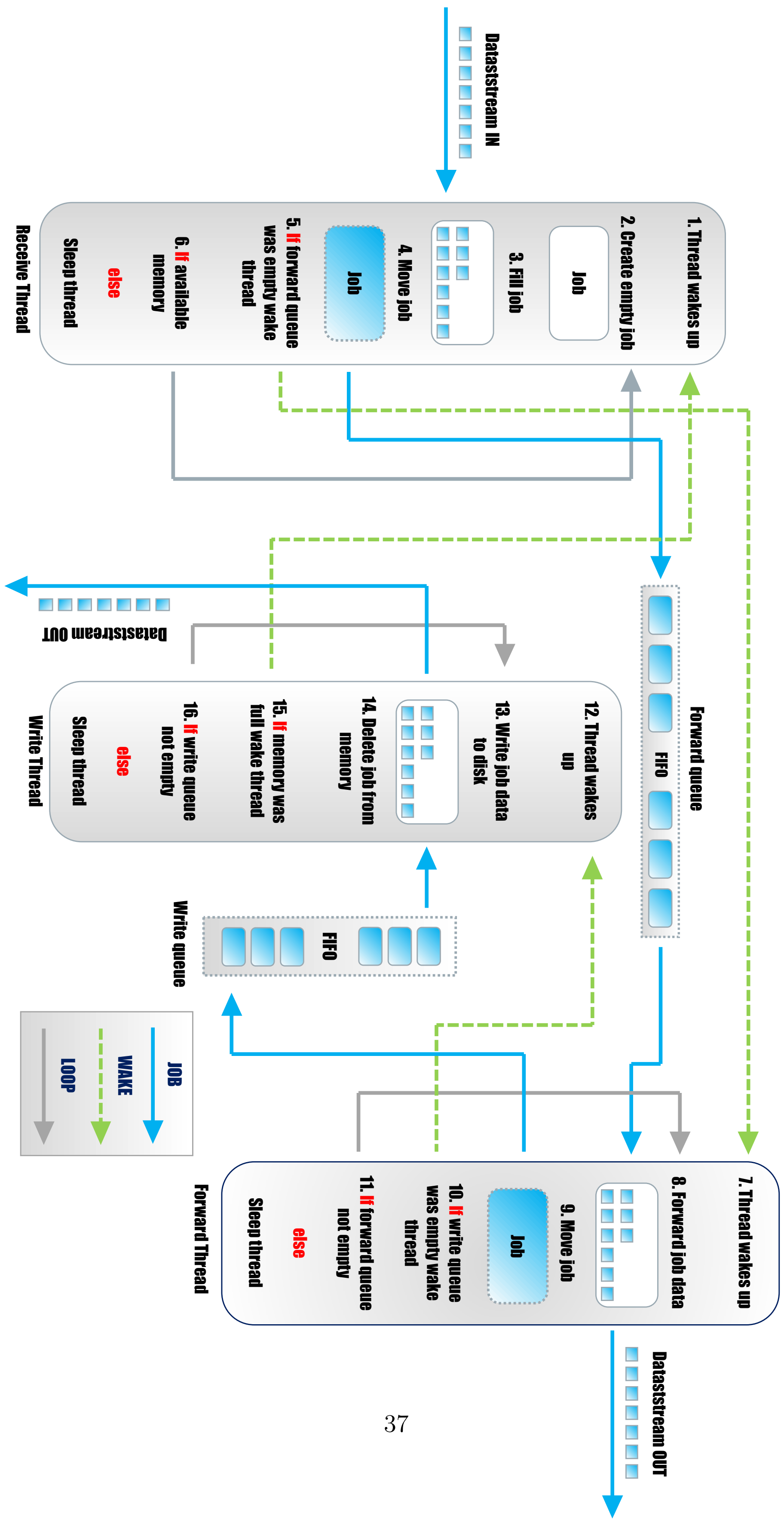

Datastream IN
1. Thread wakes up
2. Create empty job
Job
3. Fill job
4. Move job
Job
5. If forward queue was empty wake thread
6. If available memory
else
Sleep thread
Receive Thread
Forward queue
FIFO
Datastream OUT
12. Thread wakes up
13. Write job data to disk
14. Delete job from memory
15. If memory was full wake thread
16. If write queue not empty
else
Sleep thread
Write Thread
Write queue
FIFO
JOB
WAKE
LOOP
7. Thread wakes up
8. Forward job data
9. Move job
Job
10. If write queue was empty wake thread
11. If forward queue not empty
else
Sleep thread
Forward Thread
Datastream OUT

Figure 3.2

3.3 Comparative benchmarks

The prototype will be benchmarked according to CPU, storage and network usage. Additionally BitTorrent will be benchmarked in the same distribution scenarios. This is primarily done to give the performance measurements a good reference point.

3.3.1 Collecting performance data

To benchmark the different protocols a performance data collecting tool will be needed. Collectl [7] was chosen. Collectl is created for collecting system data relating to benchmarking, monitoring of system health and as a record of what the system has been doing at a certain time or period. The subsystems that Collectl can gather data from that is of interest are CPU, storage, and network. Using this tool for collecting data instead of using integrated performance monitoring will ease the the aggregation of the performance data for both protocols.

3.3.2 BitTorrent

BitTorrent was chosen as a comparative basis because it is one of the most successful application-layer multicast protocols, meaning that its performance is a good reference point and the results can easily be repeated by others.

BitTorrent has the possibility of adding nodes while a distribution is running, and it is known to scale well to a significant amount of receiving nodes. This means that the performance of the prototype has to be significantly faster than this already robust protocol before it will be relevant for usage. BitTorrent performance will be valuable in determining the worth of the prototype performance.

There exists several BitTorrent clients that can be used for comparative basis. The BitTorrent client rTorrent [39] was chosen. rTorrent was chosen for two reasons. First, that it is a client that can be run on debian linux. Second, rTtorrent is known for high efficiency in high performance networks.

3.3.3 Methodology

The main task for the comparison benchmarks will be to transfer a 10 Gbit file to all nodes in a tree network. The benchmark will be setup with two connected 5 port switches, where there are 4 individual nodes connected to each switch. This setup will simulate a tree topology. See figure 3.3 on the following page and figure 3.4 on the next page for illustrations of the experimental setups. The arrows in the figures show the expected directions of the data flow. The goal of the experiment is to see if there is a performance benefit to transferring the data distribution in an optimal path.

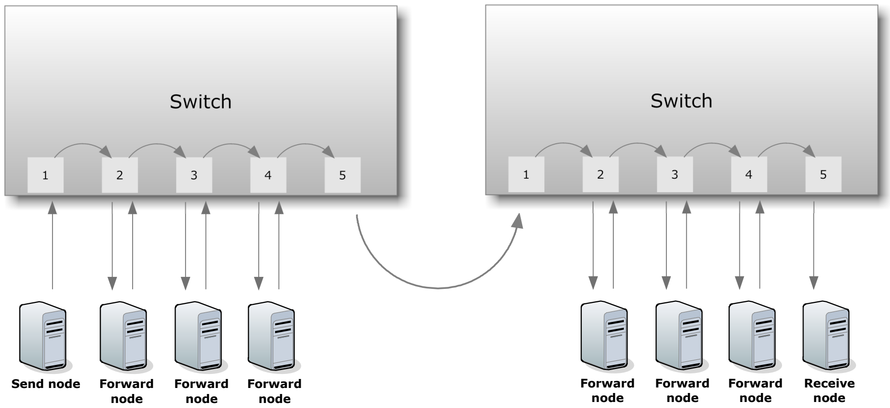


Figure 3.3: An illustration of the experimental setup for the prototype benchmarks. The data flow follows a predetermined optimal path.

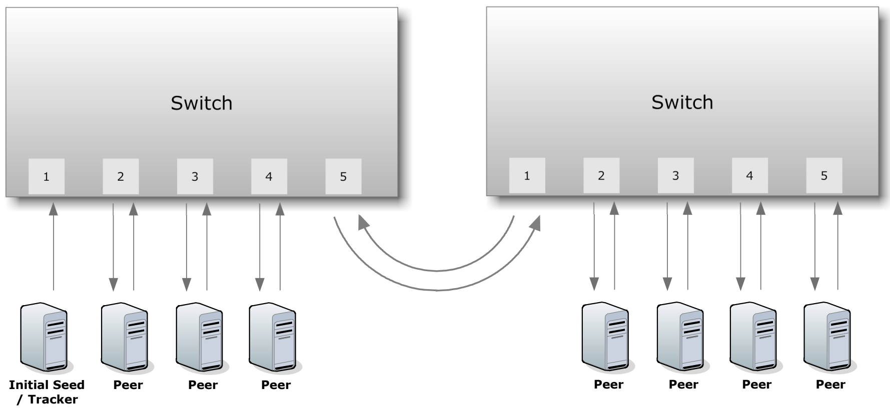


Figure 3.4: An illustration of the experimental setup for the BitTorrent benchmarks. The data flow in this experiment should go to and from all peers

Since the benchmark results are individually collected for each node, the data will need to be aggregated into a single plot. The plot aggregation and benchmark orchestration requires time synchronization for accurate measurements. Synchronization to an NTP server will therefore be required before any benchmarks can be run.

3.3.4 BitTorrent configuration

BitTorrent performance variates according to configuration, in this section the rTorrent client configuration setup is presented.

rTorrent uses the file rtorrent.rc to configure the client behavior. The following rtorrent.rc file will be used during the rTorrent benchmarks.

rtorrent.rc

```
# Global upload and download rate in KiB. "0" for unlimited.
download_rate = 0
upload_rate = 0

min_peers = 50 ; look for more peers if limit doesn't reach 50
max_peers = 500 ; if there are 500 peers, don't allow any more

# Same as above but for seeding completed torrents (-1 = same as downloading)
min_peers_seed = 10
max_peers_seed = 100

# Maximum number of simultanious uploads per torrent.
max_uploads = 25

port_range = 6881-6999 ; ports to use for listening

# Start opening ports at a random position within the port range.
port_random = yes

check_hash = no; check hash on finished torrents

# encryption settings
encryption = allow_incoming,enable_retry,prefer_plaintext

use_udp_trackers = yes ; setup client to use udp (stateless) trackers

# DHT clientless tracker
dht = yes
dht_port = 6881
peer_exchange = yes

# Session tmp file (relative dir is good, absolute is bad)
session = ./.session

# Default directory to save the downloaded torrents.
directory = ./tmp/

# Watch a directory for new torrents, and stop those that have been deleted (^d)
schedule = watch_directory,1,1,"load_start=./watch/*.torrent"
schedule = untied_directory,1,1,remove_untied=./watch/*.torrent
```

Tracker

To coordinate communication between peers a tracker is needed. Peertracker [22] was chosen for this job. Peertracker was chosen because it does not index uploading of torrents, share ratio monitoring or any other form of user management.

Tuning

- **Memory** To keep disk usage to a minimum, rTorrent needs to use system memory as caching. The amount of memory used for caching is limited to "ulimit -m" or 1GiB [40]. In the benchmark setup "ulimit -m" will be set to unlimited, such that rTorrent can use up to 1GiB of memory. This is the same amount of memory available as used for the prototype benchmarks.

- **check_hash disabled** Preliminary testing revealed that rTorrent disconnects all peers at torrent completion when check_hash option is enabled. The node does not seem to continue seeding to the disconnected peers after hash check is completed, but rather leaves it to the tracker to resolve the issue. When most nodes complete fairly simultaneously this could lead to some nodes being disconnected from most other peers. The "abandoned" peer/s must then time-out and re-connect to tracker before the conflict can be resolved. Disabling this feature enhanced average convergence-time significantly.

- **super-seeding disabled** When using super-seeding all connections are closed for the initial seeder after the swarm shows two distributed complete copies [41]. The initial seeder re-joins the swarm and are reconnected with normal seeding connection after this. There are so few peers in the benchmark setup that the peer disconnection is considered too be more detrimental to performance than what performance could be gained by enabling super-seeding.

3.3.5 Prototype configuration

Tuning

- **Memory** For the prototype to perform well, system memory will need to be available for use. 1GiB of memory will be the maximum amount of memory that the prototype will be able to use. This is the same amount of memory available as used in the rTorrent benchmark.

- **Job size** The job size will be set to 1460 bytes in the comparative benchmarks. It is expected that this setting will use more CPU power than what a larger job size would, but it is not expected to become a limiting factor.

3.4 Test equipment

Here is a list of the equipment used in the experiments in this thesis

3.4.1 Node hardware

All experiments in this thesis has been carried out on nodes with equivalent specifications. Here follows the node specifications

- **Model:** Apple MacBook 2.4 (Mid-2010)
- **CPU:** Intel Core 2 duo P8600 2,4 GHz
- **Memory:** 2 GiB DDR3 SDRAM 1066 MHz, 1,74 GiB usable by the system as 256 MiB is reserved for graphics.
- **Network card:** Nvidia Nforce 10/100/1000
- **Disk:** 250 GB Serial ATA (5400 RMP)

All nodes where connected via Cat6 certified cables. Debian GNU/Linux 6.0 2.6.32-5-amd64 was used as operating-system for the nodes.

3.4.2 Network devices

Here follows the network equipment that was benchmarked in the roof performance tests.

Device	Certified speeds	Full duplex	Ports
HP V1405C-5	10/100 Mbit	Yes	5
Dlink DGS-1005D	10/100/1000 Mbit	Yes	5
Netgear GS605	10/100/1000 Mbit	Yes	5
Netgear ProSafe GS105	10/100/1000 Mbit	Yes	5
Cisco SD2005	10/100/1000 Mbit	Yes	5
3Com 3CGSU05	10/100/1000 Mbit	Yes	5
Cisco SG 100D-08	10/100/1000 Mbit	Yes	8
3Com 3CGSU08	10/100/1000 Mbit	Yes	8

Table 3.2: A table of the network equipment that was benchmarked.

The list consist of a mix of consumer and business-grade switch equipment. The amount of devices is considered fair enough to uncover if there is a general trend when looking at switch throughput performance. The 100 Mbit switch was included in the list to uncover if there was any issues with node network performance.

3.4.3 Network configuration

The benchmarks was run with the following network configuration

Test	MTU	Overhead/Options	Ideal per node	Ideal 5 port	Ideal 8 port
TCP	1500	IPv4, Timestamp	941.48 Mbps	4705.74 Mbps	7529.19 Mbps
UDP	1500	IPv4, None	957.09 Mbps	4785.44 Mbps	7656.70 Mbps

Table 3.3: The ideal throughput values for the benchmark configurations. The throughput values are rounded to the nearest two decimal places.

In table 3.3 the target performance values for the benchmarks in the following sections are listed.

Chapter 4

Results

4.1 Finding the roof performance

4.1.1 Iperf wrapper

In this section the developed Iperf wrapper is presented. The wrapper was successfully developed with the required functionality defined in the methodology section. The full Perl script can found in appendix 7.1 on page 134.

To execute the script the following command can be used

```
bench.pl -i iplist -s 30 -t 60 -o results/ -x 10.0.0.10
```

The command will result in 30 sample benchmarks run for 60 seconds each. The option -x defines which NTP server to synchronize time with. The input file *iplist* will determine which hosts to benchmark and option -o defines which folder to save the output results.

iplist full

```
10.0.0.10
10.0.0.11
10.0.0.12
```

Here is a sample of how a typical output from the script looks like

output sample

```
SSH Password: **********
Client: 10.0.0.10
Client: 10.0.0.11
Client: 10.0.0.12

Server: 10.0.0.10
Server: 10.0.0.11
Server: 10.0.0.12
```

```
Client connect retries: 0

Server connect retries: 0

Trying to sync 10.0.0.10 to 10.0.0.10
10.0.0.10 is ntp server,
skipping synchronization of 10.0.0.10
Trying to sync 10.0.0.11 to 10.0.0.10
time server 10.0.0.10 offset 0.175051 sec
Trying to sync 10.0.0.12 to 10.0.0.10
time server 10.0.0.10 offset 0.183668 sec

### SAMPLE 1 OF 30 ###
Host 0 connects to 1
iperf -s  (10.0.0.11)
iperf -c 10.0.0.11 -y C -i 1 -t 62 (10.0.0.10)
Waiting for: 2
Host 1 connects to 2
iperf -s  (10.0.0.12)
iperf -c 10.0.0.12 -y C -i 1 -t 62 (10.0.0.11)
Waiting for: 2
Host 2 connects to 0
iperf -s  (10.0.0.10)
iperf -c 10.0.0.10 -y C -i 1 -t 62 (10.0.0.12)
Waiting for: 2
Waiting for execution to finish..
```

The script will run benchmarks for 2 seconds more than specified in the script input, and the resulting output is cropped to the specified length. It was uncovered during preliminary testing that Iperf some times fail to print results for the last executed second, and therefore the 2 second margin was added to prevent uneven length data.

The results are output to files separated by protocol, host and sample. The resulting output to file from one host/sample could be as follows

T-h1-s1.csv sample

```
20120315193526,10.0.0.10,58556,10.0.0.11,5001,3,0.0-1.0,118136832,941094656
20120315193527,10.0.0.10,58556,10.0.0.11,5001,3,1.0-2.0,117809152,941473216
20120315193528,10.0.0.10,58556,10.0.0.11,5001,3,3.0-4.0,117686272,941490176
20120315193529,10.0.0.10,58556,10.0.0.11,5001,3,4.0-5.0,110395392,883163136
```

Invoking the script with option -h will print the correct usage of the script, and give explanation of the csv output fields.

Usage full

```
Usage: bench.pl [OPTIONS] --in iplist

DESCRIPTION

A wrapper to iperf that can benchmark multiple nodes simultaneously.
The benchmark results are output in a csv file format.

GENERIC OPTIONS
```

```
  -h, --help      Display Usage information
  -i, --in        File with list of IP addresses to benchmark
  -o, --out       File to output sampled data
  -t, --time      Time in seconds to sample
  -w, --wait      How long to wait before adding another node to the benchmark
  -s, --samples   How many benchmarks to run
  -x, --sync      Specify NTP server sync address
  -r, --window    Specify TCP recieve window size

UDP OPTIONS

  -u, --udp       Invoke UDP test
  -b, --bandwidth [m|g] Specify UDP bandwidth target

EXPLANATION OF CSV OUTPUT FIELDS

TCP:
        Field 1:  Timestamp
        Field 2:  From host
        Field 3:  From port
        Field 4:  Target host
        Field 5:  Target port
        Field 6:  ID
        Field 7:  Time interval
        Field 8:  Bytes transferred
        Field 9:  Bits per second over interval

UDP:
        Field 1:  Timestamp
        Field 2:  From host
        Field 3:  From port
        Field 4:  Target host
        Field 5:  Target port
        Field 6:  ID
        Field 7:  Time interval
        Field 8:  Bytes transferred
        Field 9:  Bits per second over interval
        Field 10: Jitter in milliseconds
        Field 11: Lost datagrams over interval
        Field 12: Total datagrams over interval
        Field 13: Lost datagrams in % over interval
        Field 14: Datagrams delivered out of order
```

4.1.2 Baseline Cat6, TCP

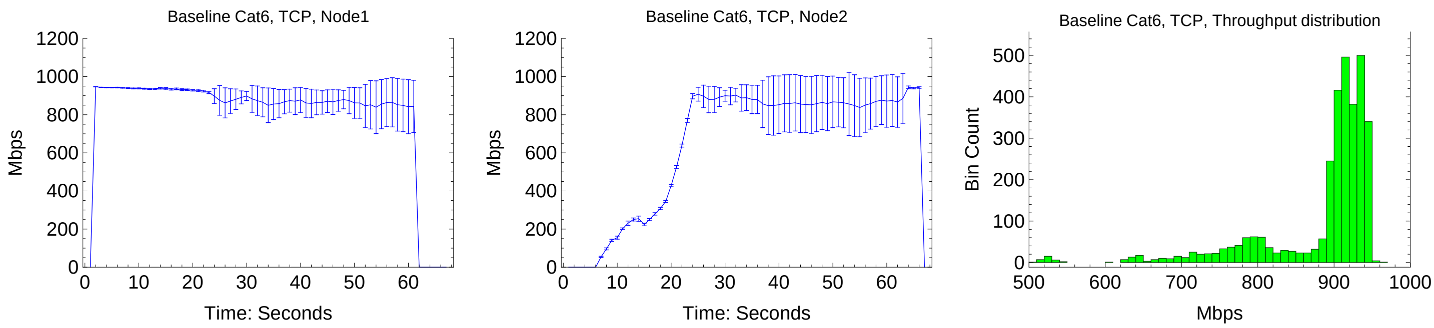


Figure 4.1: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second. In the histogram to the right the benchmark throughput distribution is shown.

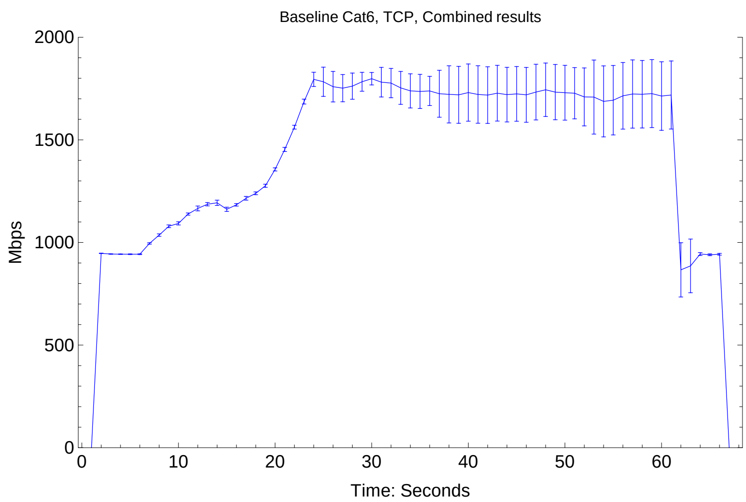


Figure 4.2: The results for the 2 nodes combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second.

4.1.3 Baseline Cat6, UDP

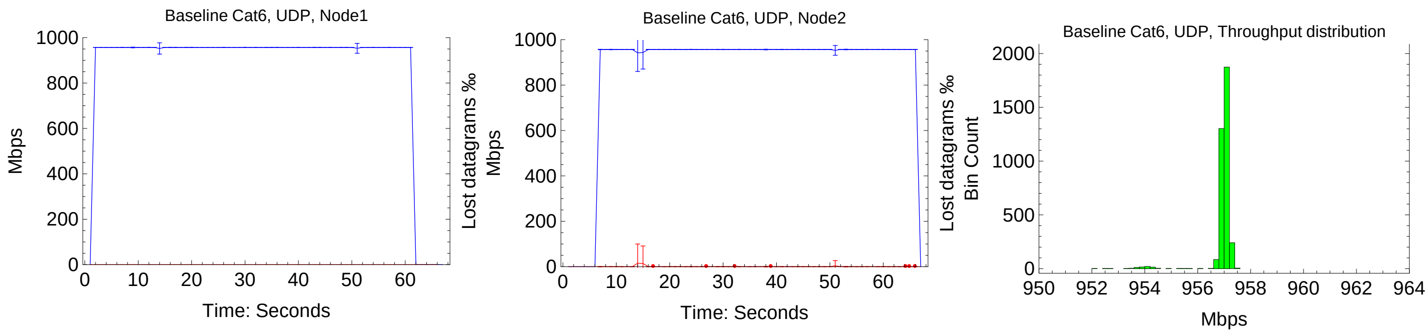


Figure 4.3: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second. In the histogram to the right the benchmark throughput distribution is shown.

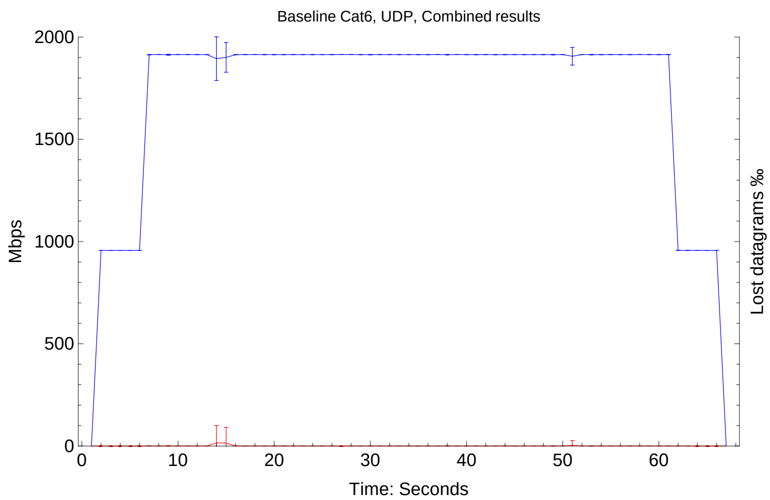


Figure 4.4: The results for the 2 nodes combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second.

4.1.4 HP V1405C-5, TCP

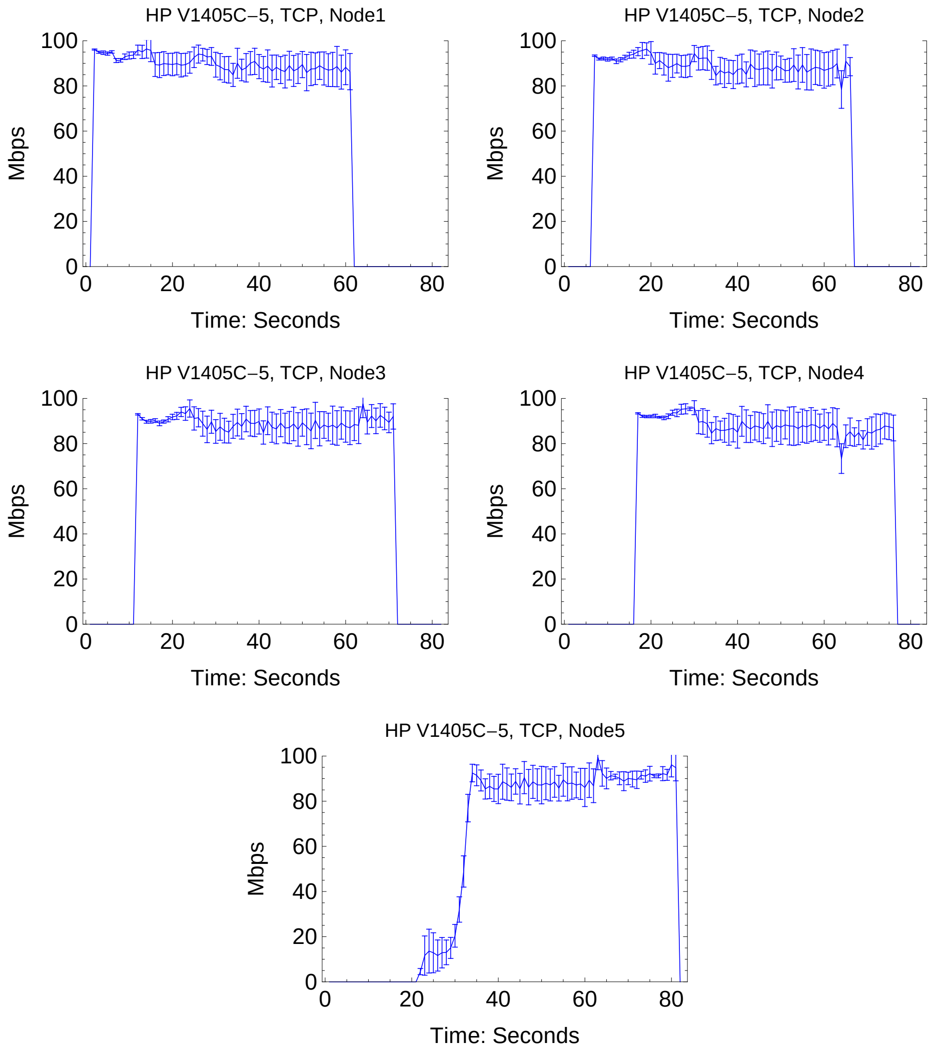


Figure 4.5: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

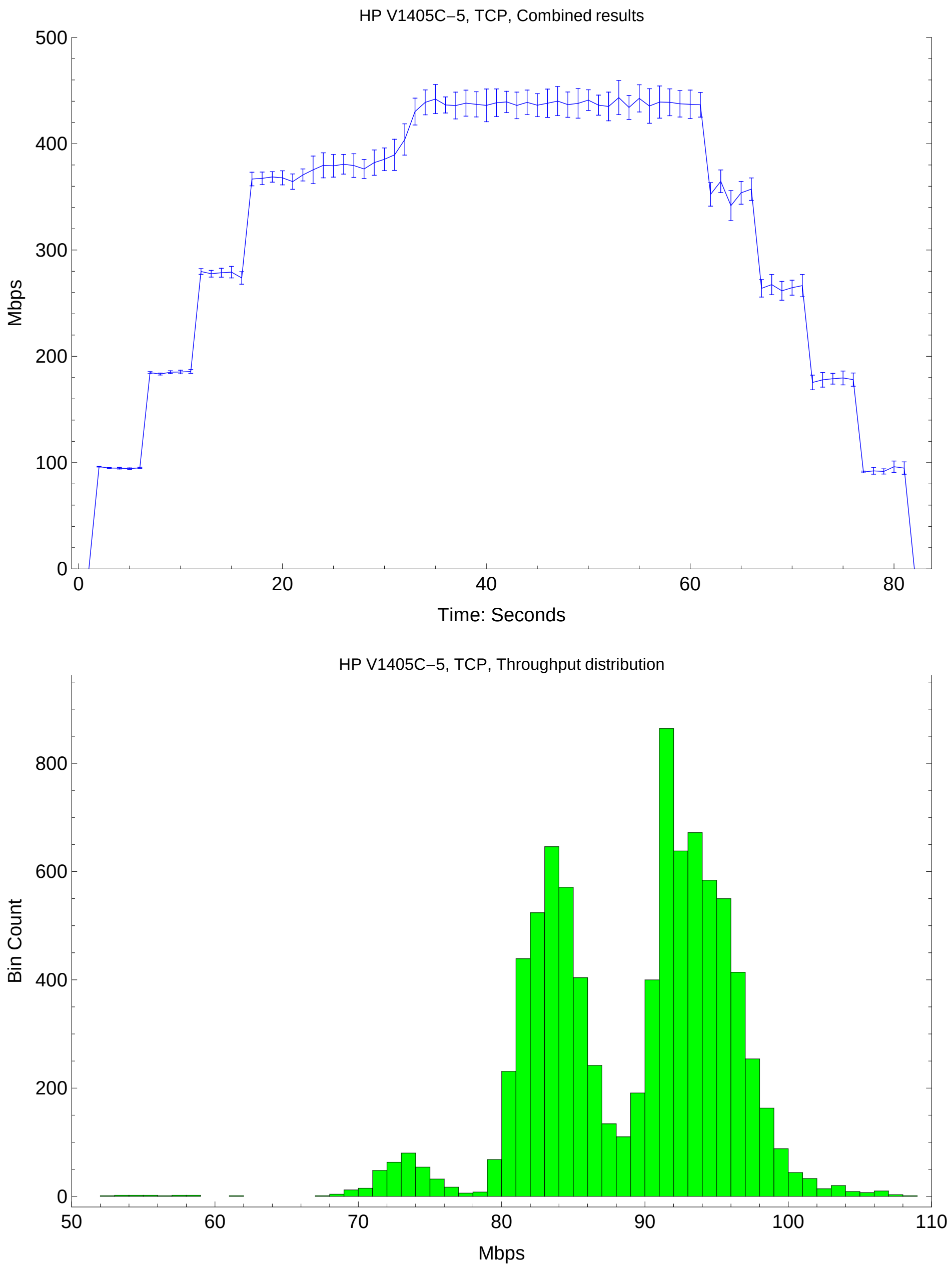


Figure 4.6: In the top graph the results for the 5 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.5 HP V1405C-5, UDP

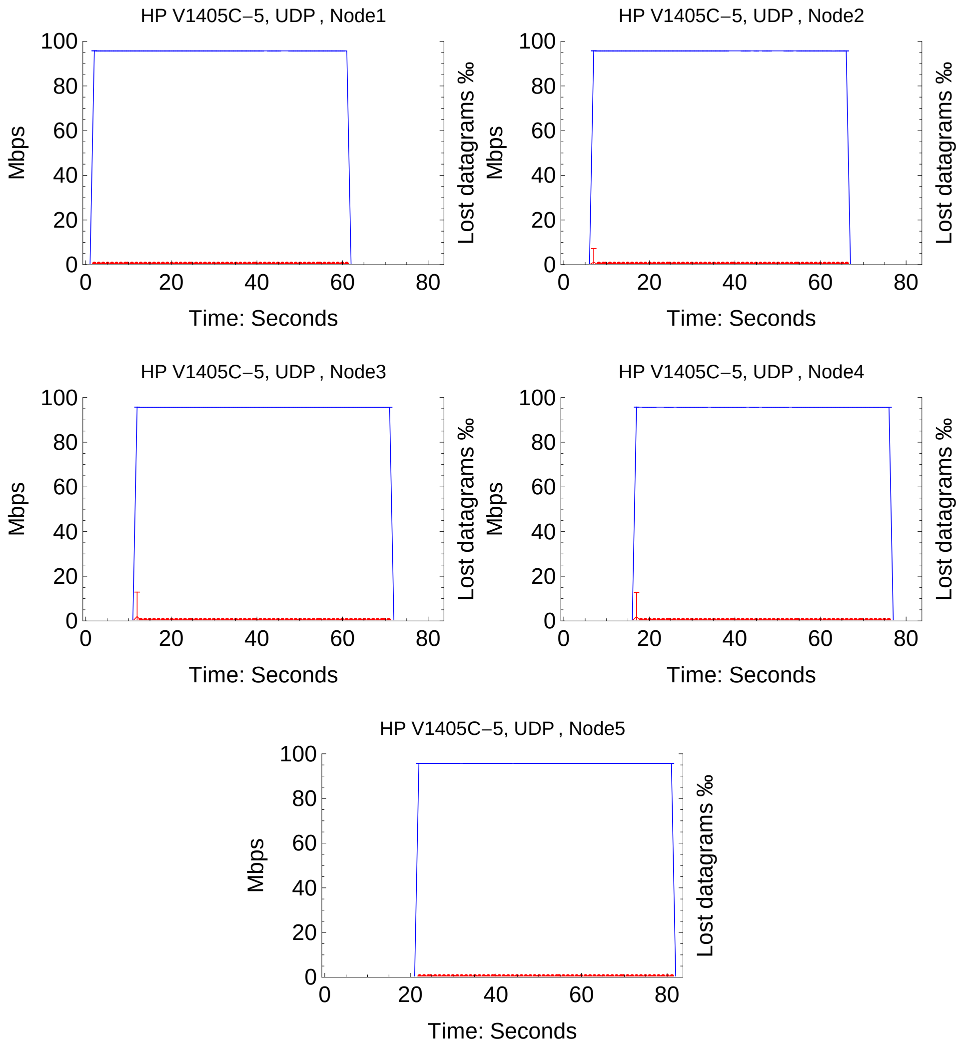


Figure 4.7: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

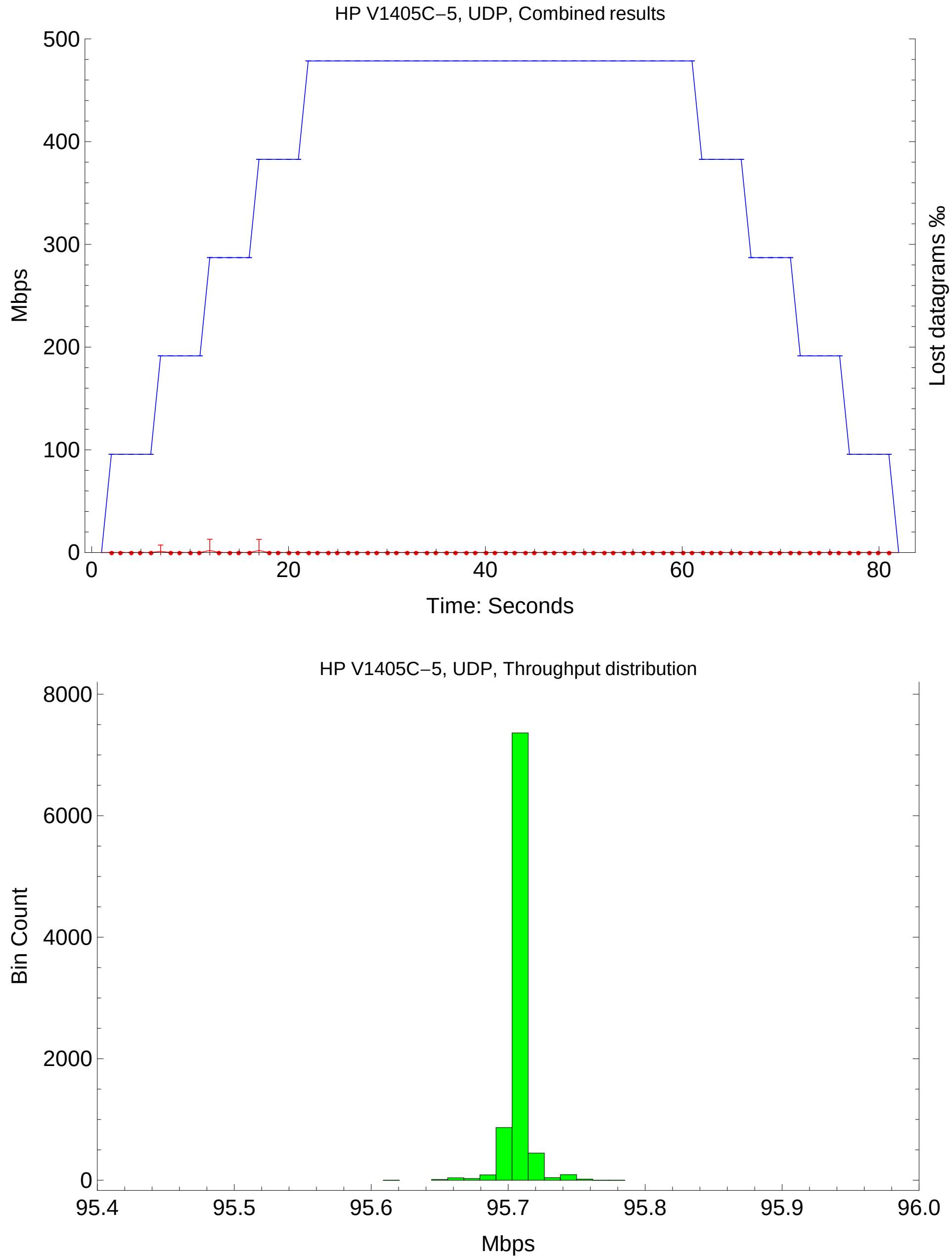


Figure 4.8: In the top graph the results for the 5 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.6 Dlink DGS-1005D, TCP

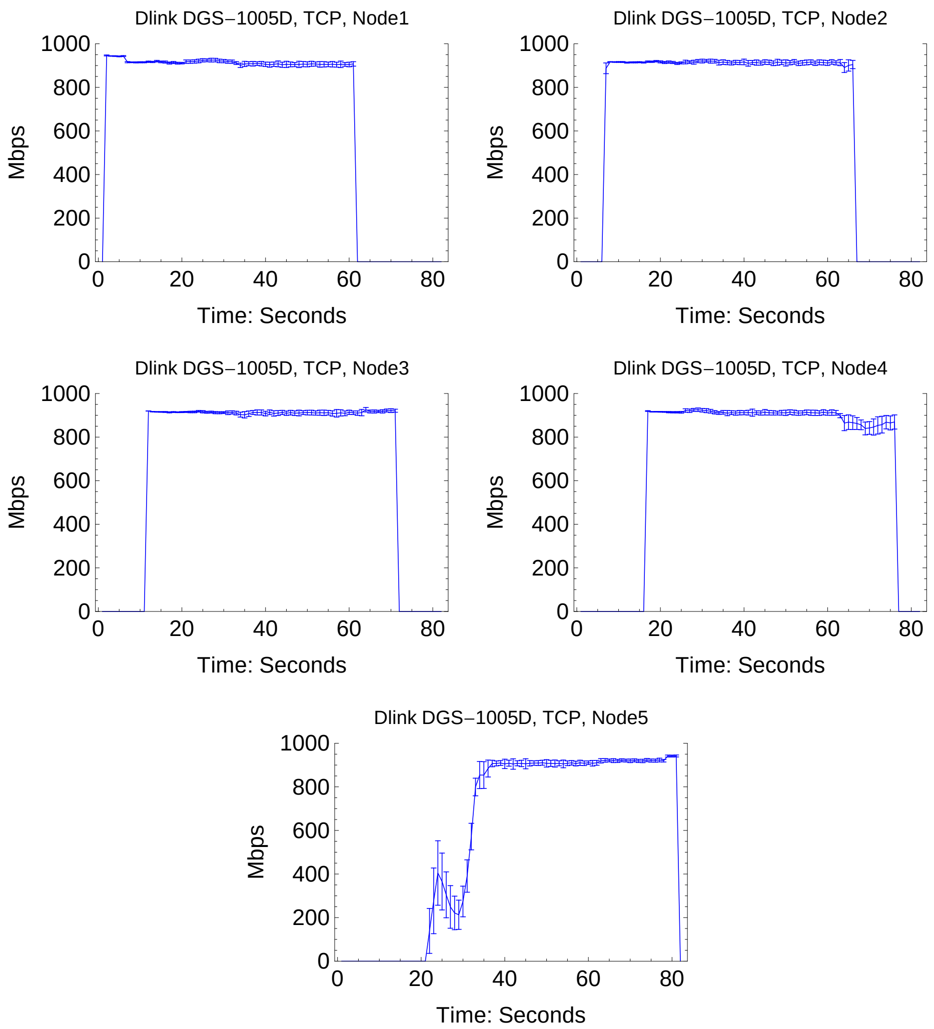


Figure 4.9: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

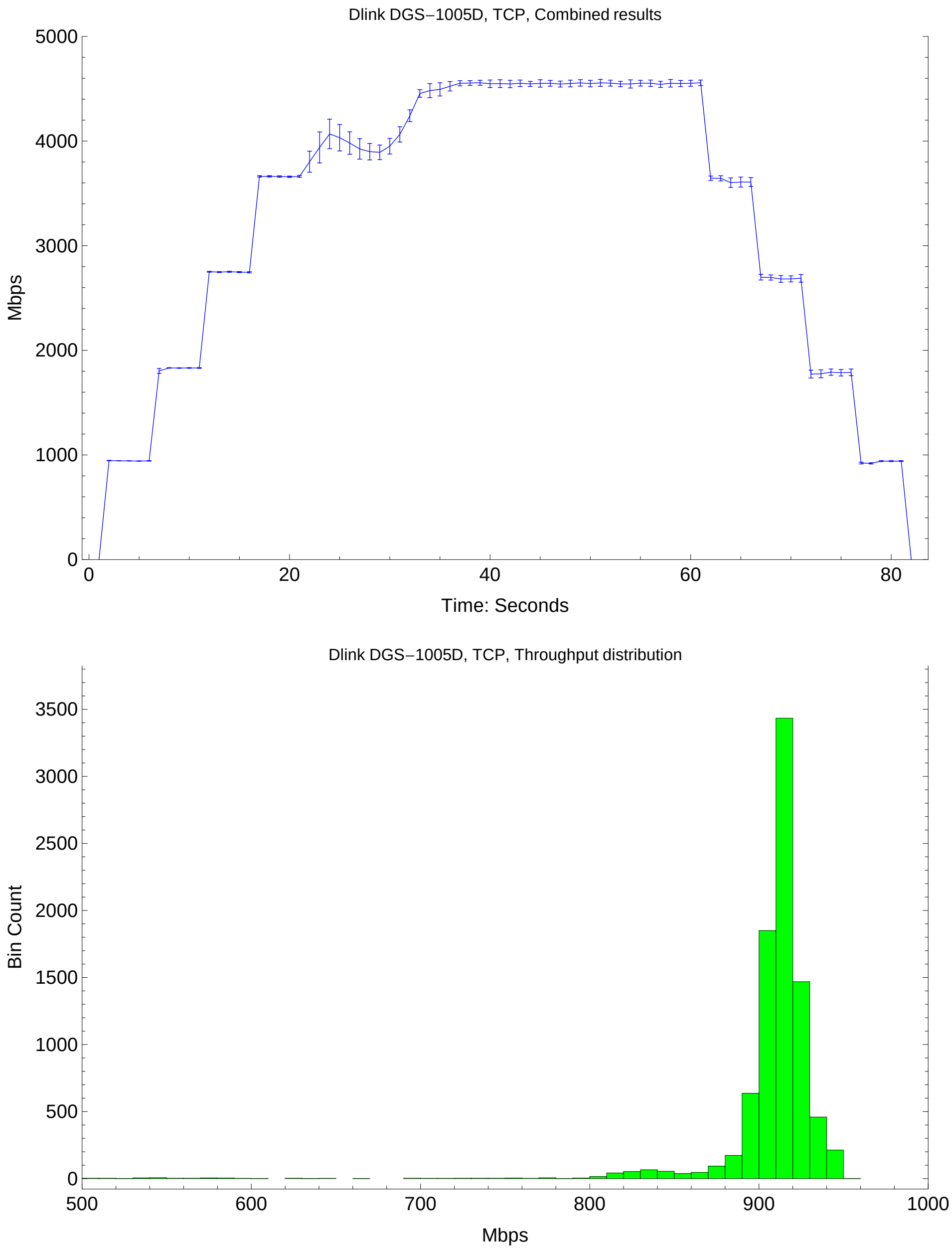


Figure 4.10: In the top graph the results for the 5 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.7 Dlink DGS-1005D, UDP

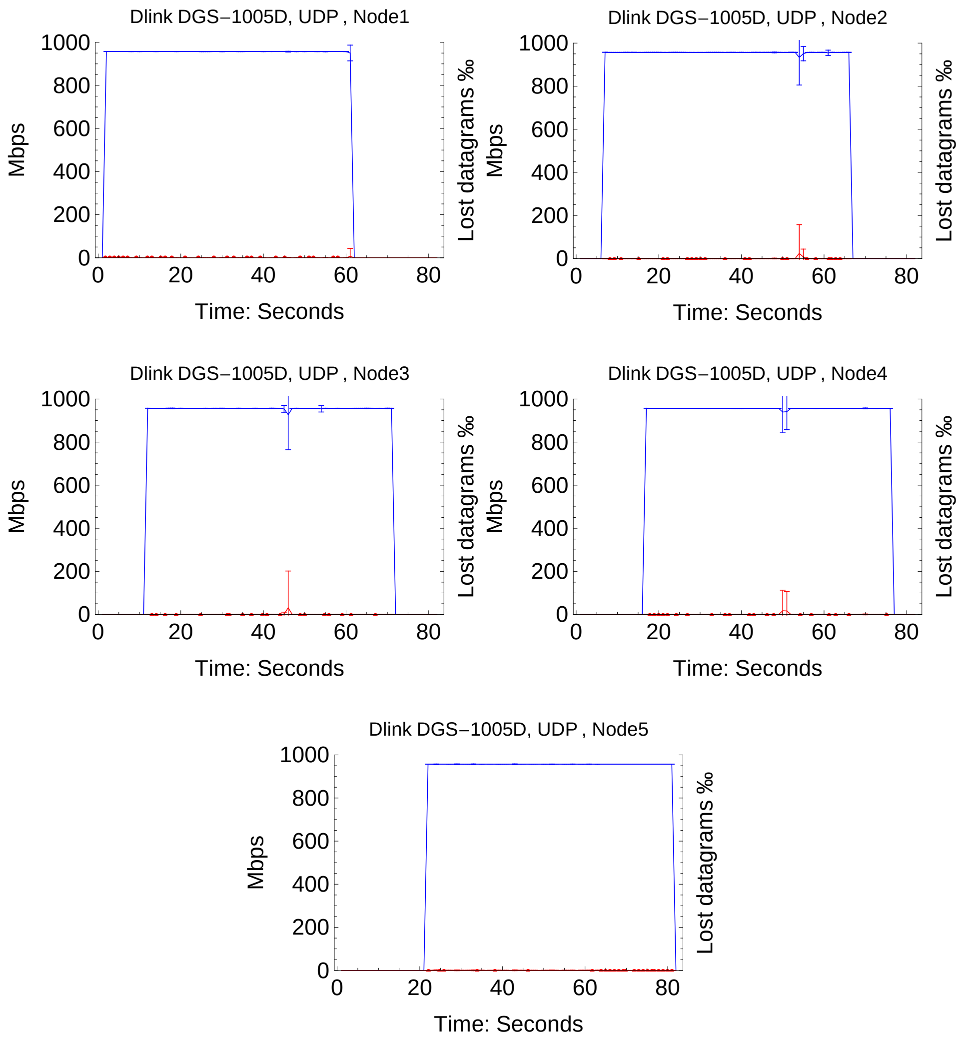
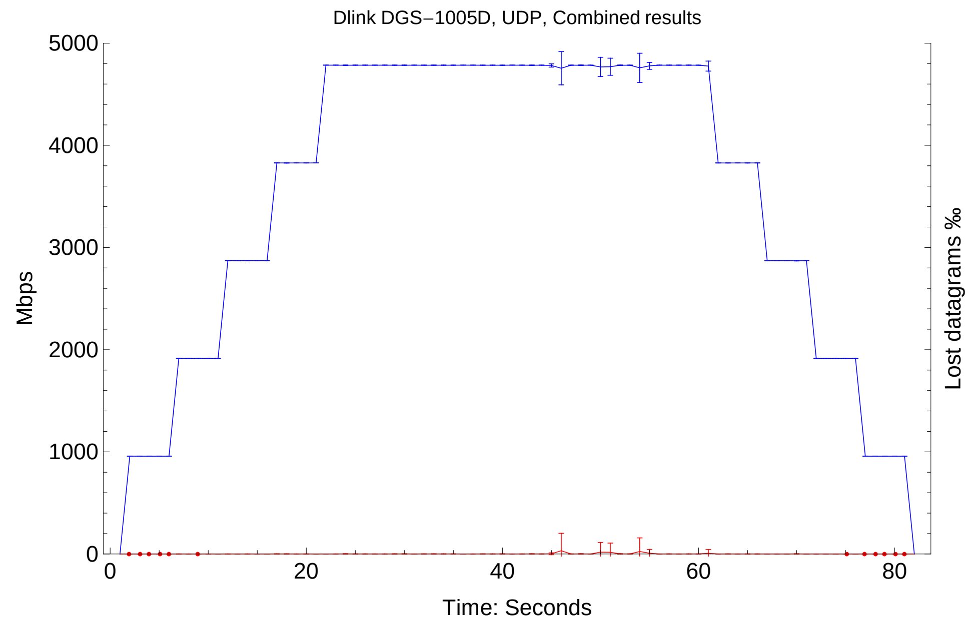


Figure 4.11: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.



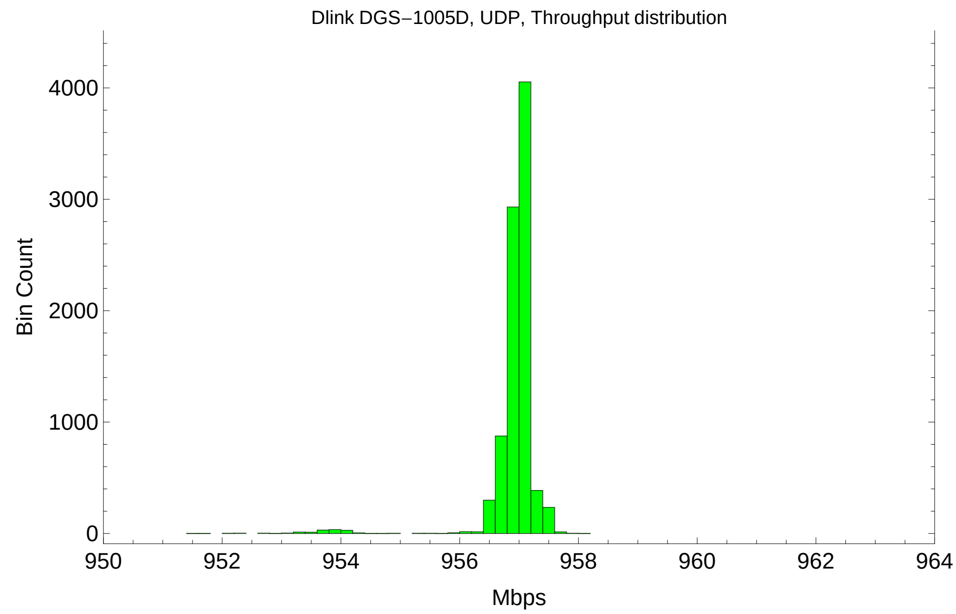


Figure 4.12: In the top graph the results for the 5 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.8 Netgear GS605, TCP

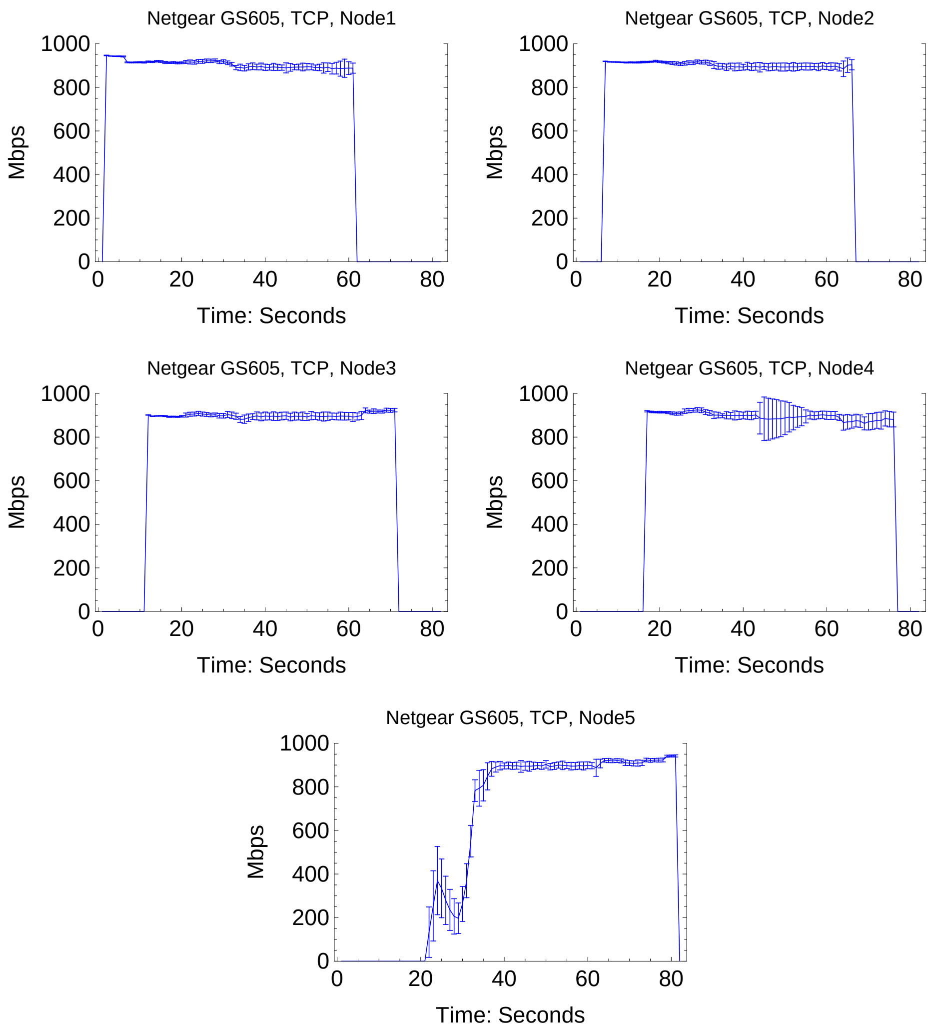


Figure 4.13: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

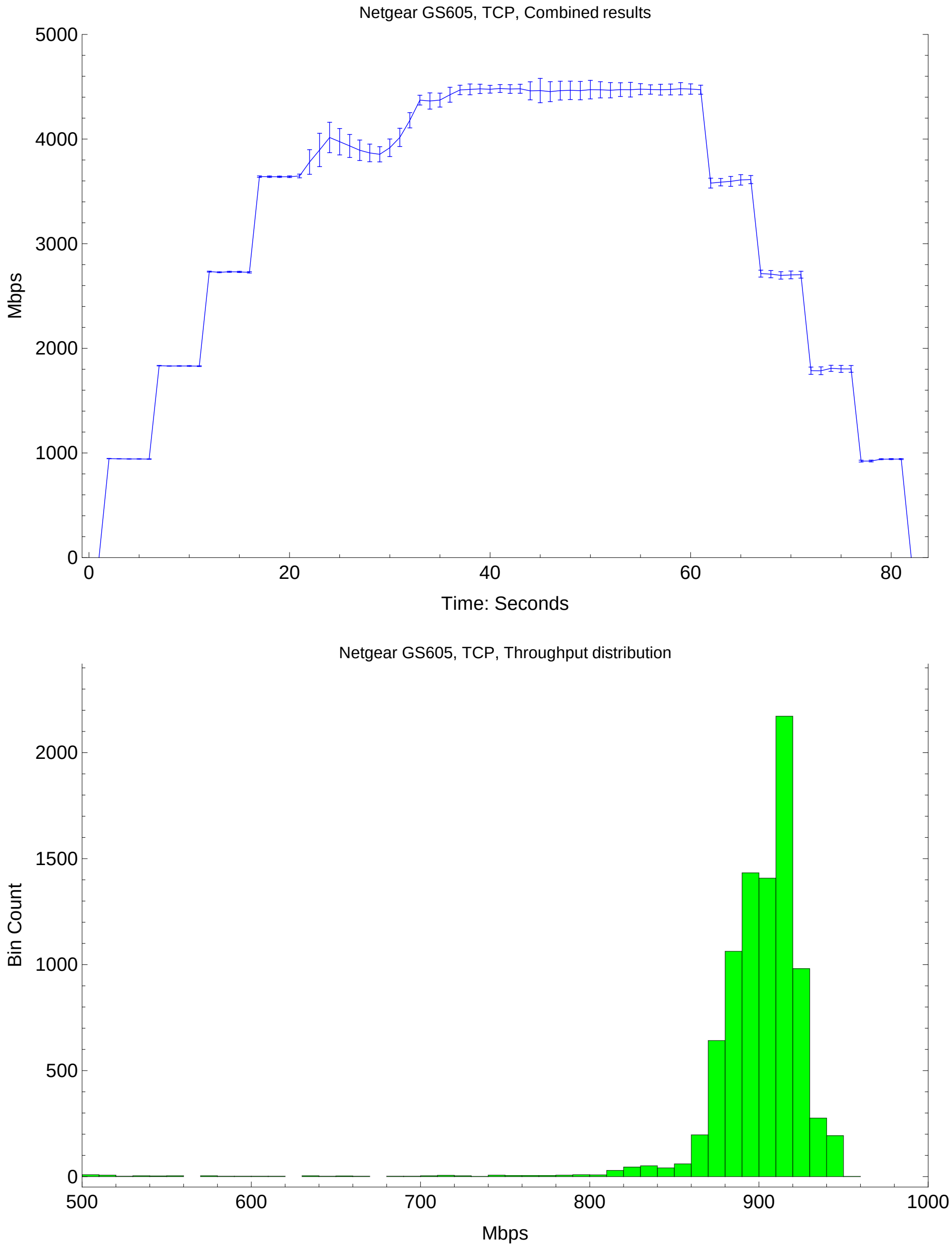


Figure 4.14: In the top graph the results for the 5 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.9 Netgear GS605, UDP

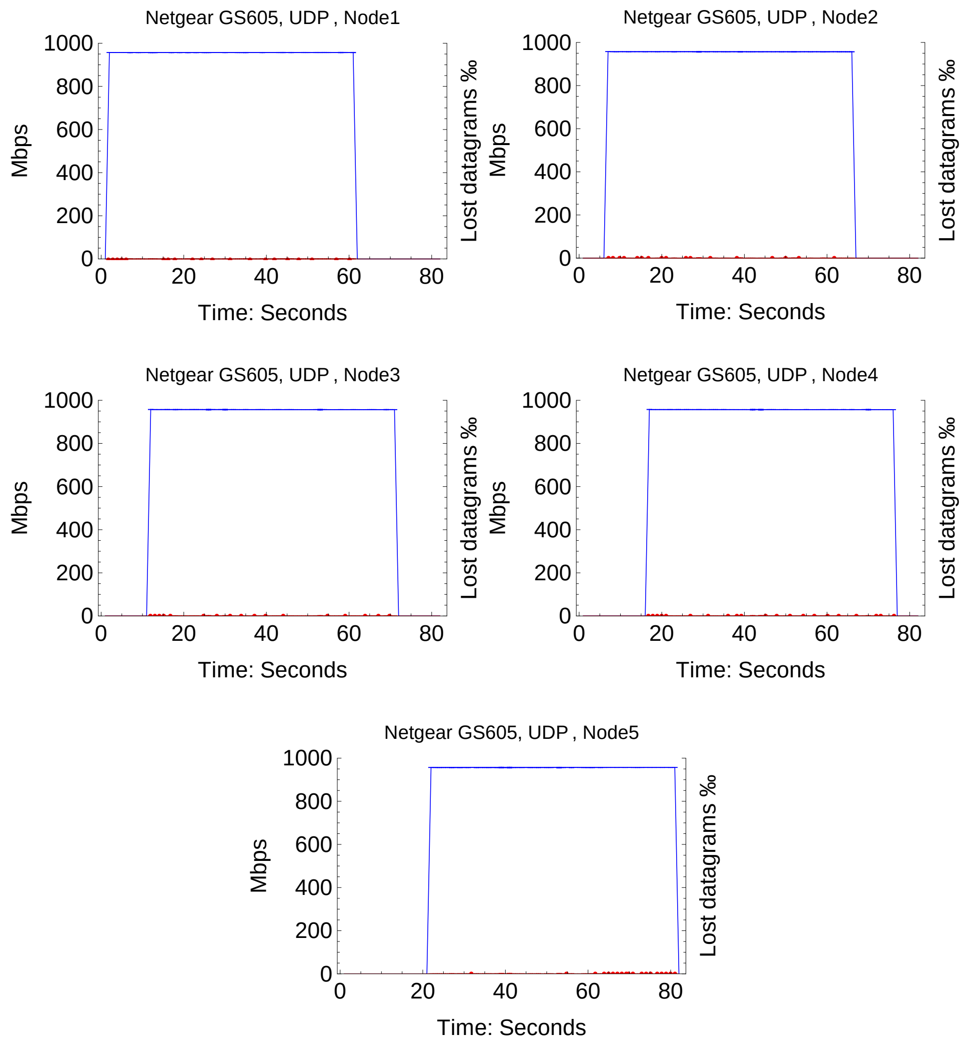
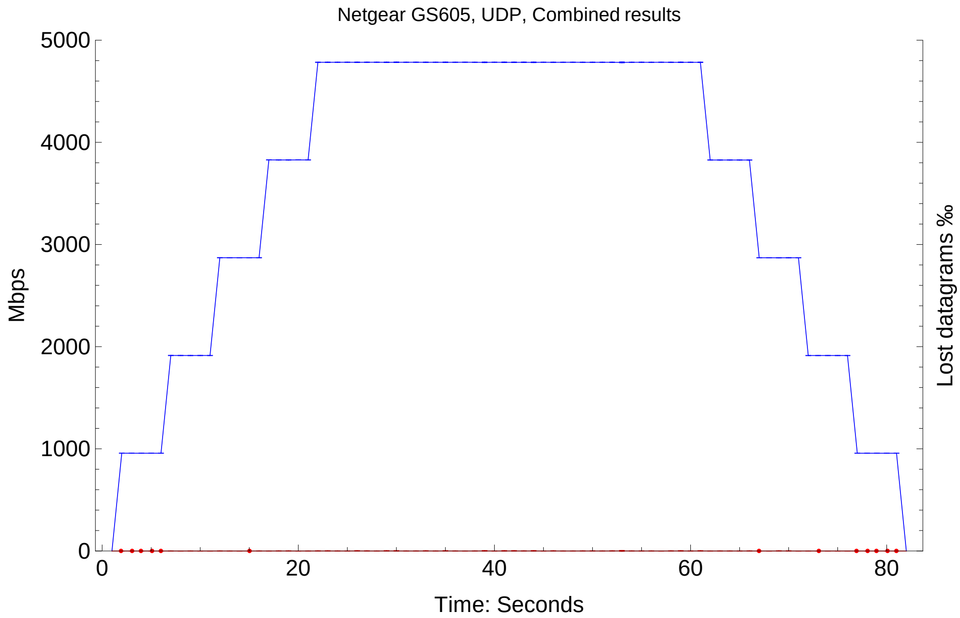


Figure 4.15: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.



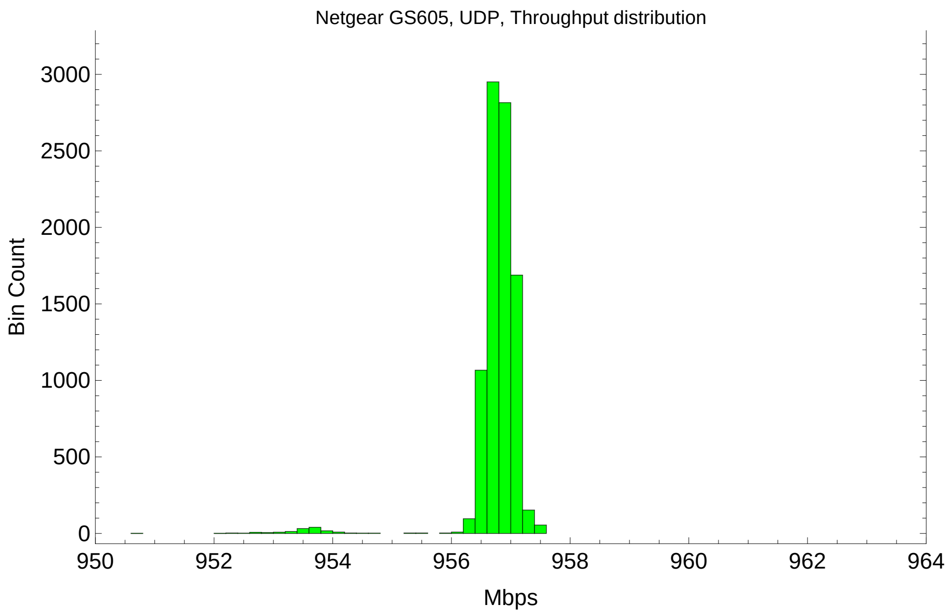


Figure 4.16: In the top graph the results for the 5 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.10 Netgear ProSafe GS105, TCP

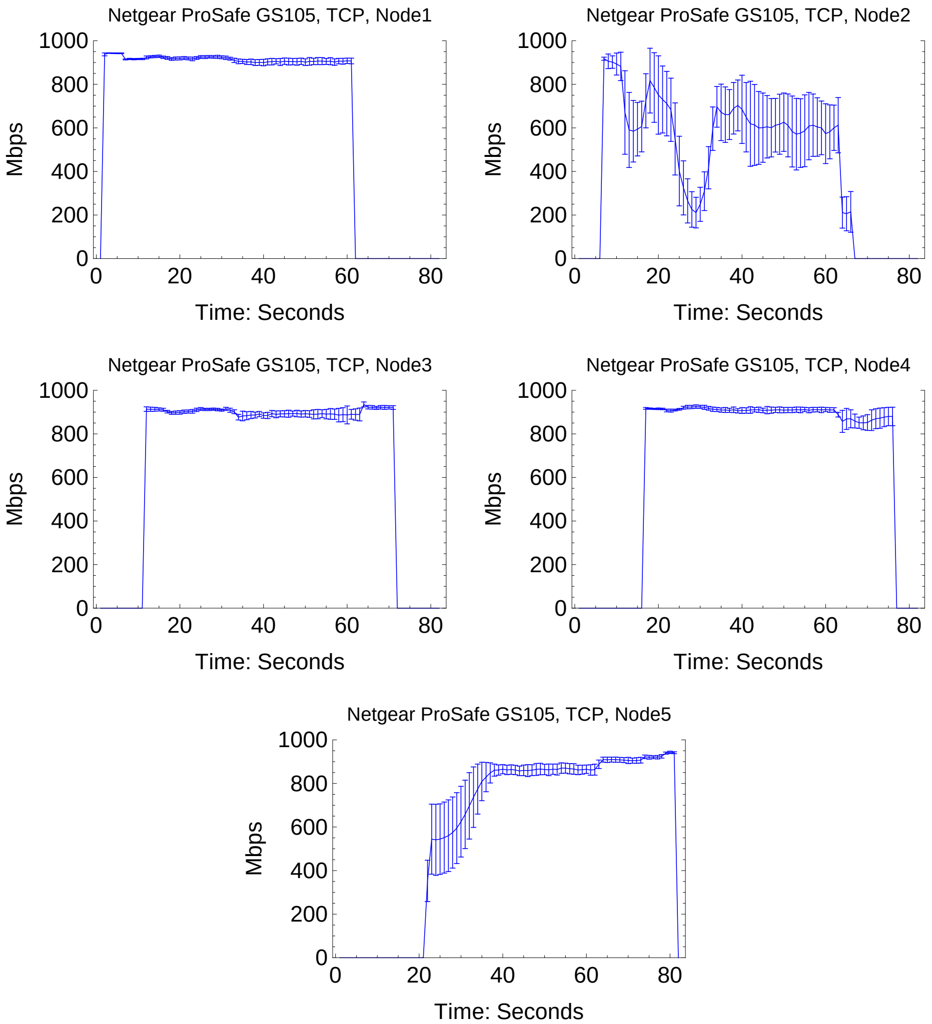
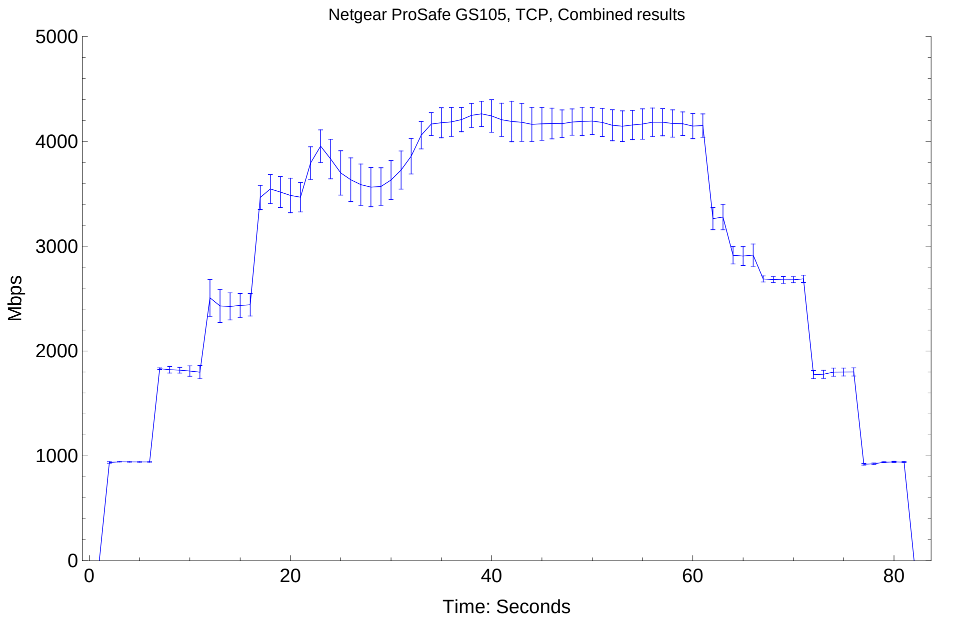


Figure 4.17: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.



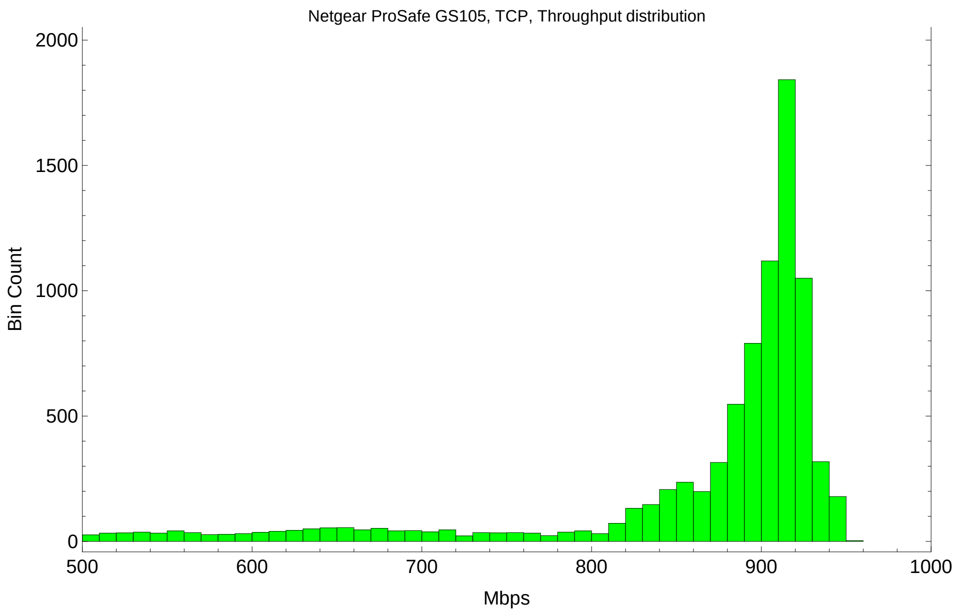


Figure 4.18: In the top graph the results for the 5 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.11 Netgear ProSafe GS105, UDP

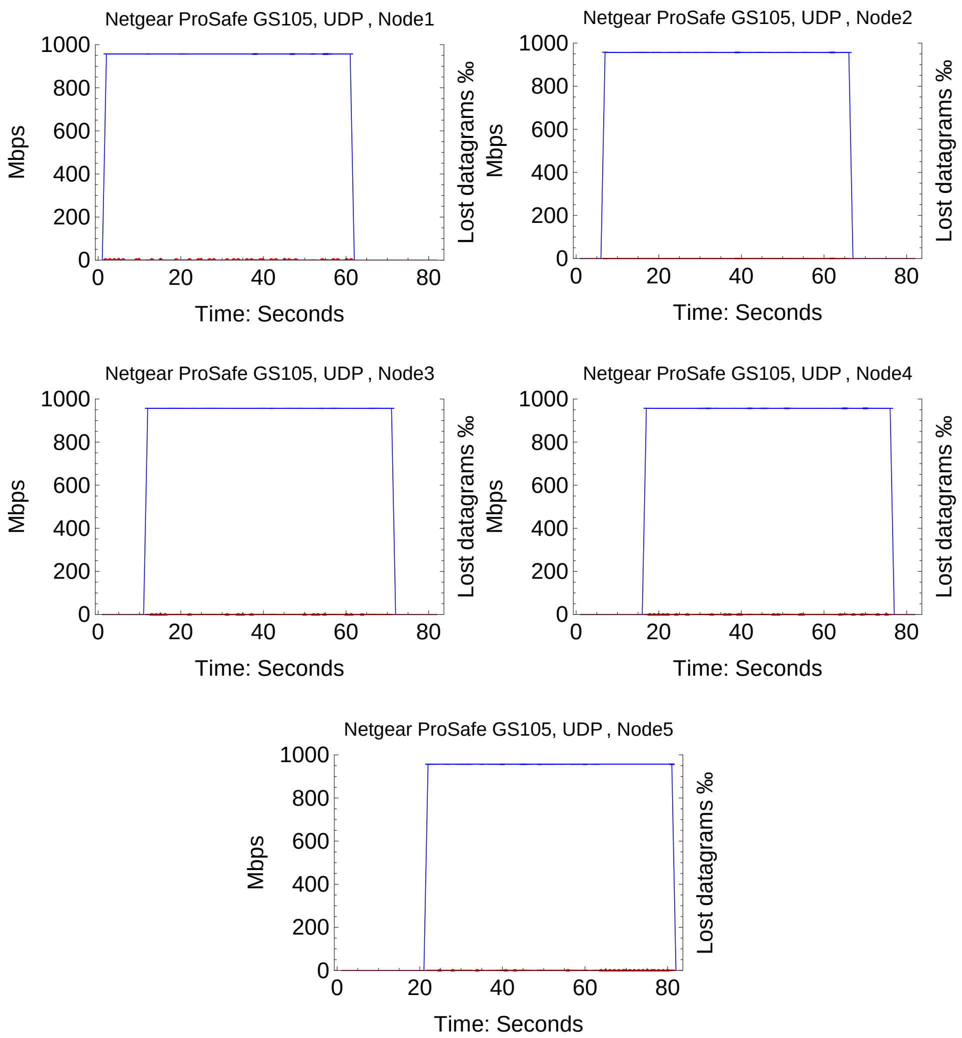


Figure 4.19: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

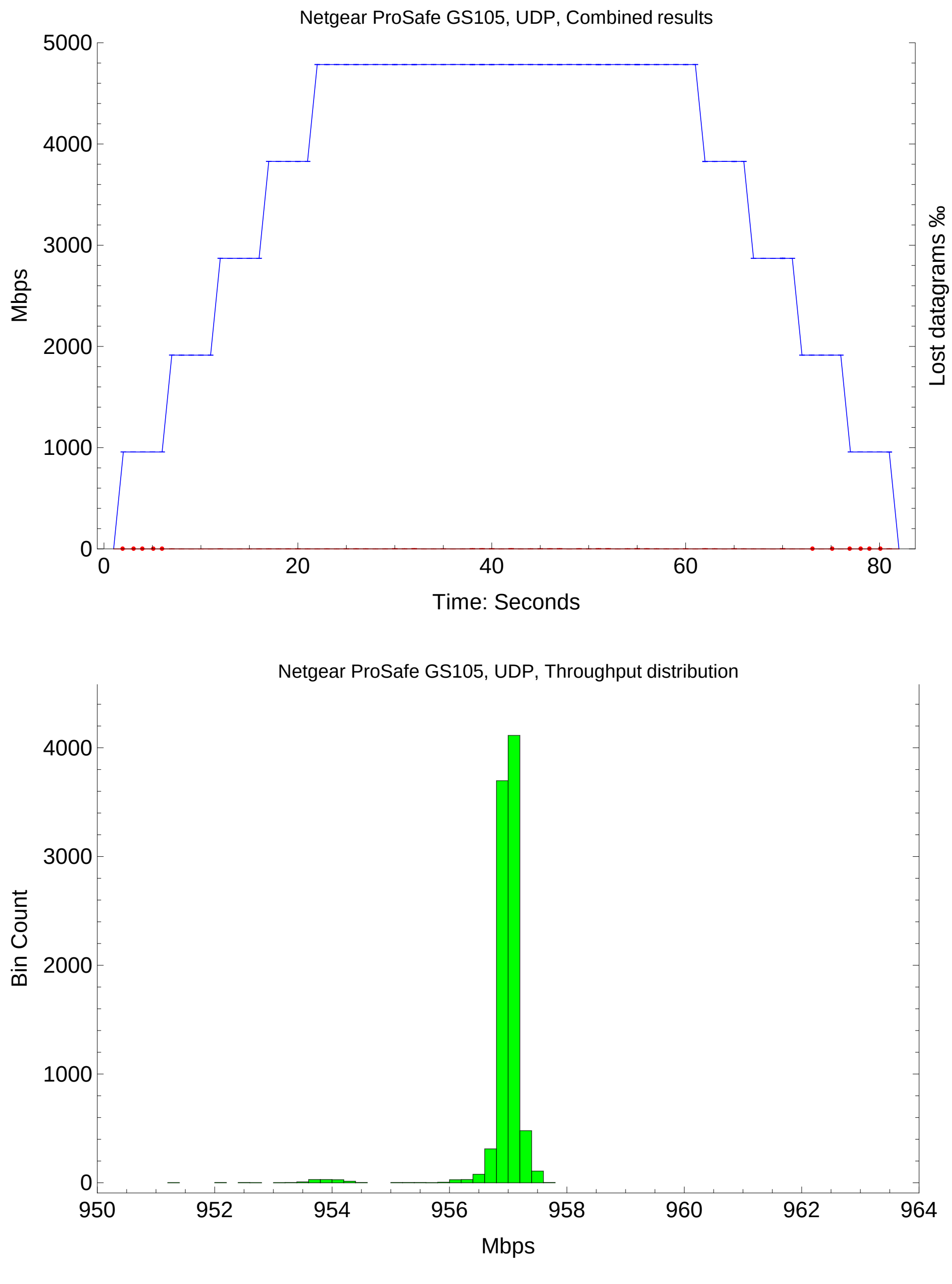


Figure 4.20: In the top graph the results for the 5 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.12 Cisco SD2005, TCP

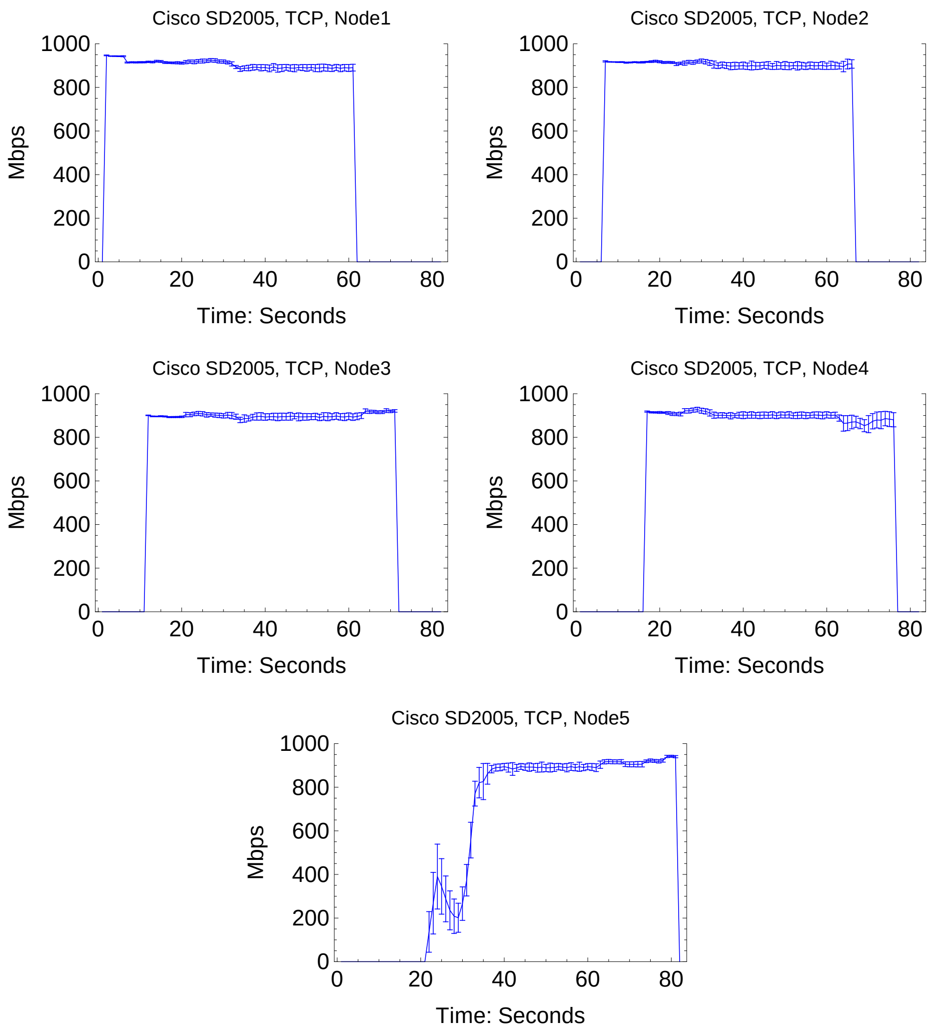
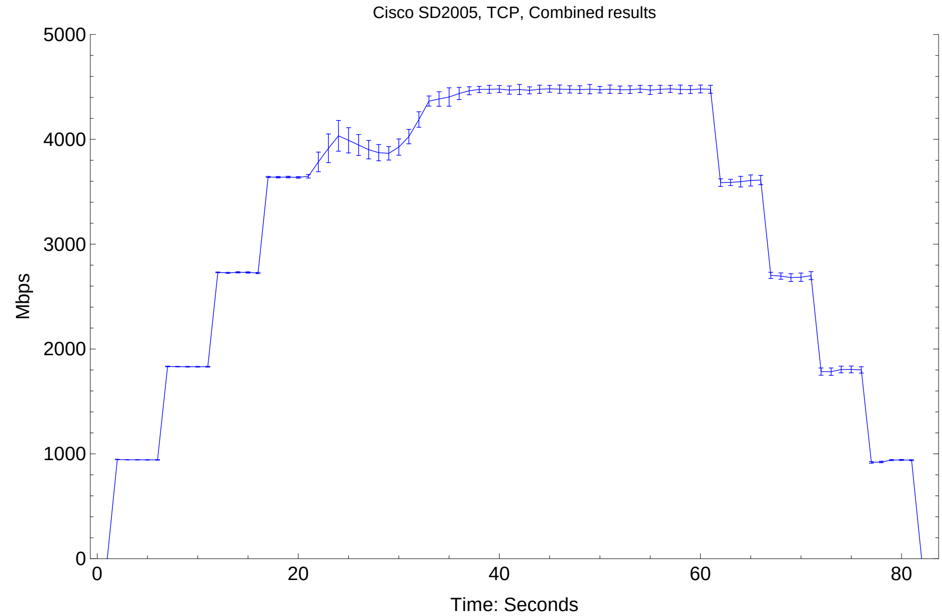


Figure 4.21: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.



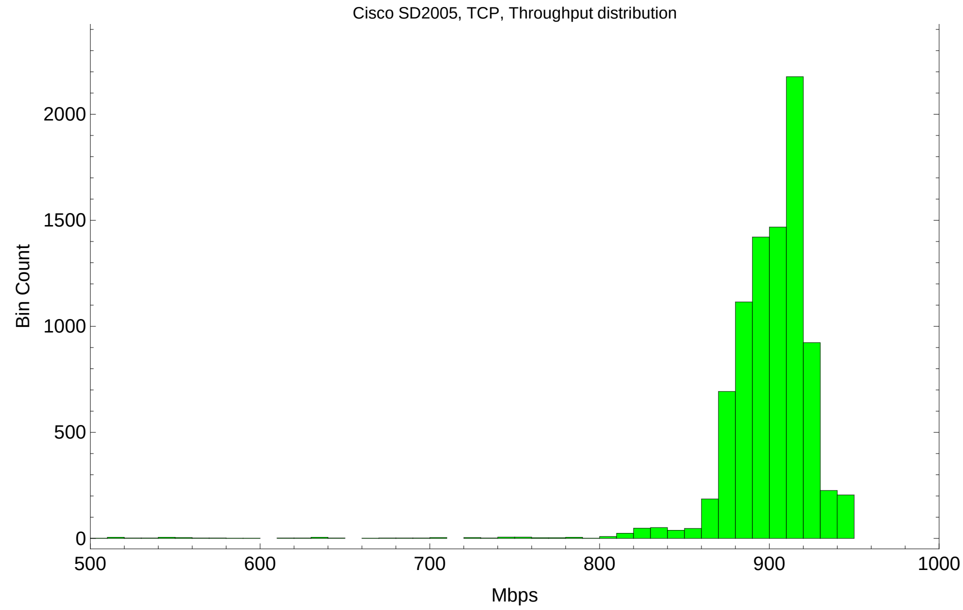


Figure 4.22: In the top graph the results for the 8 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.13 Cisco SD2005, UDP

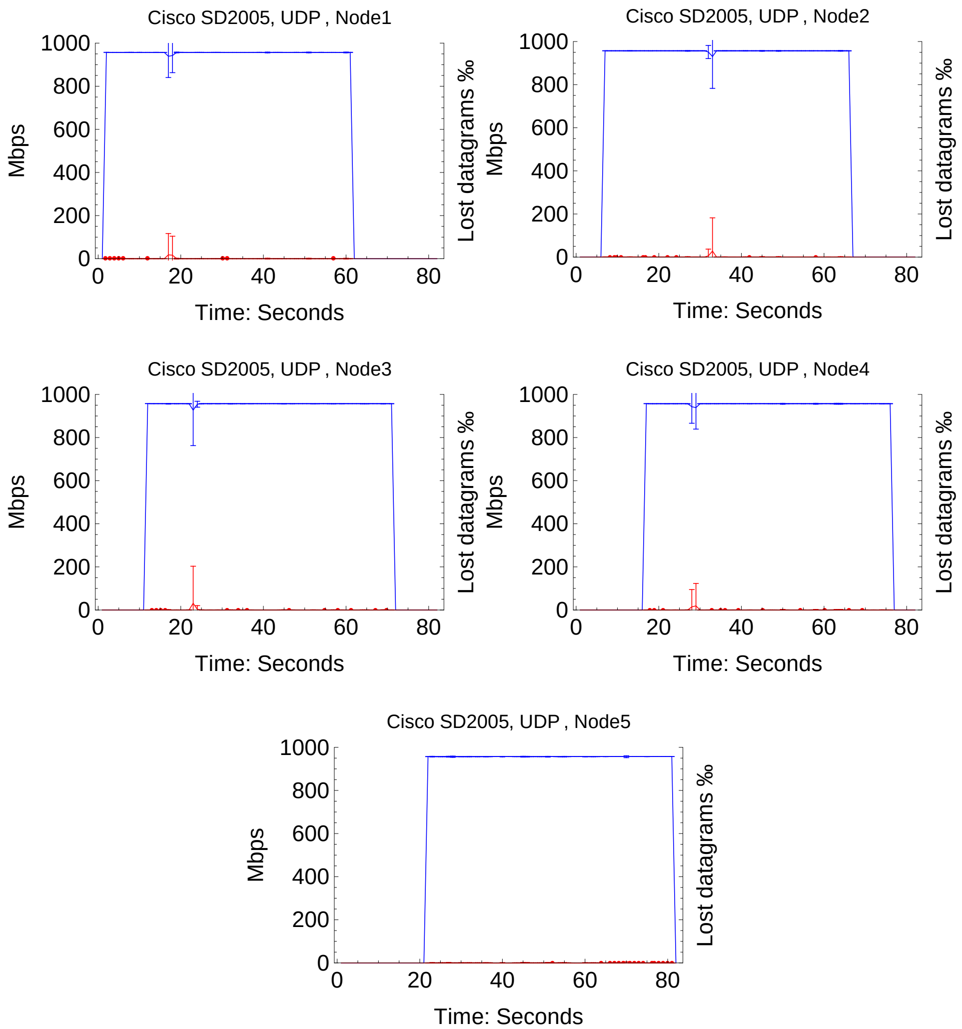


Figure 4.23: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

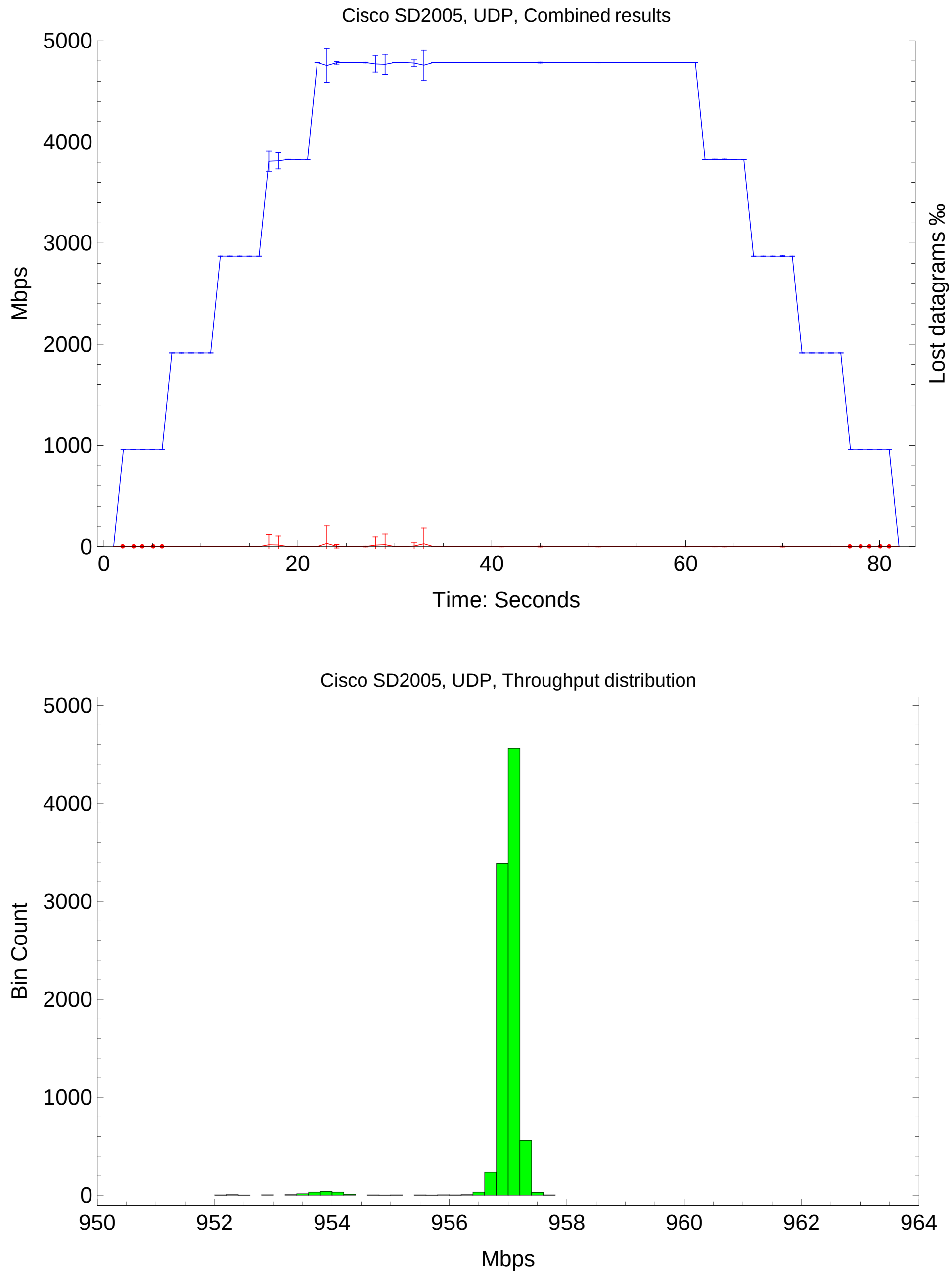


Figure 4.24: In the top graph the results for the 5 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.14 3Com 3CGSU05, TCP

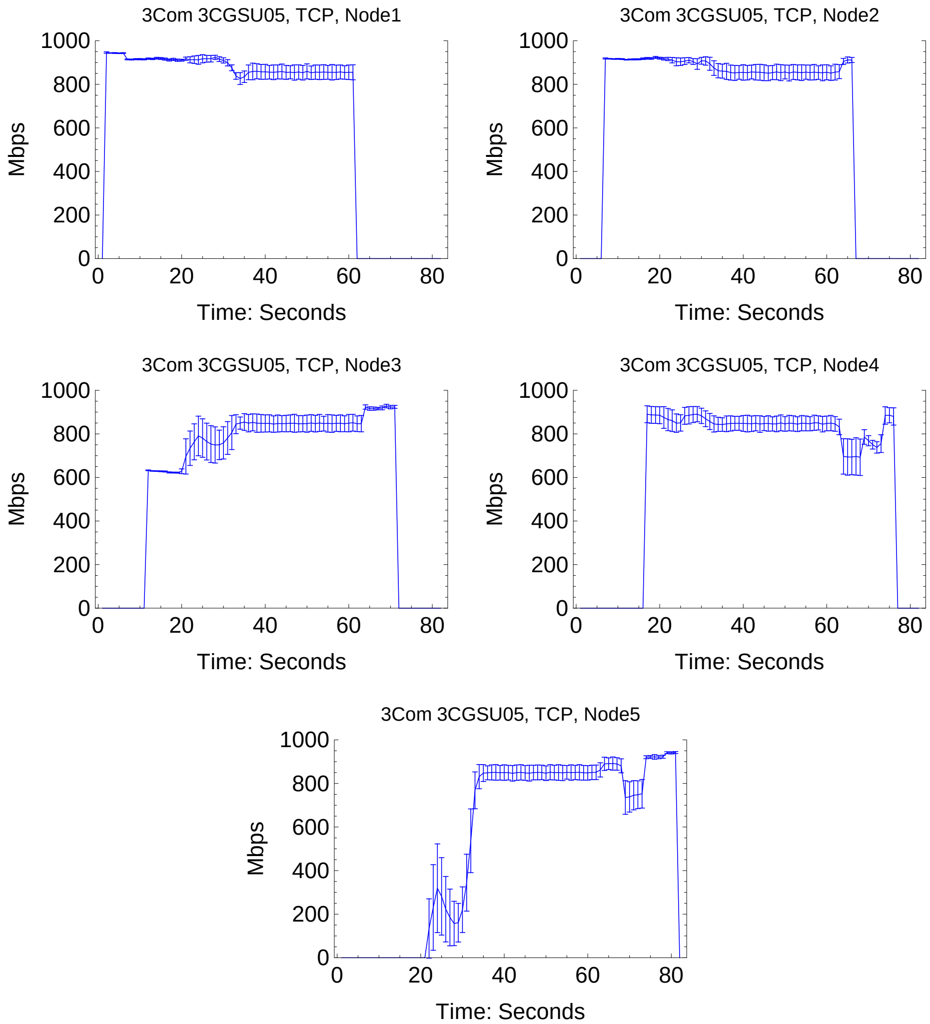
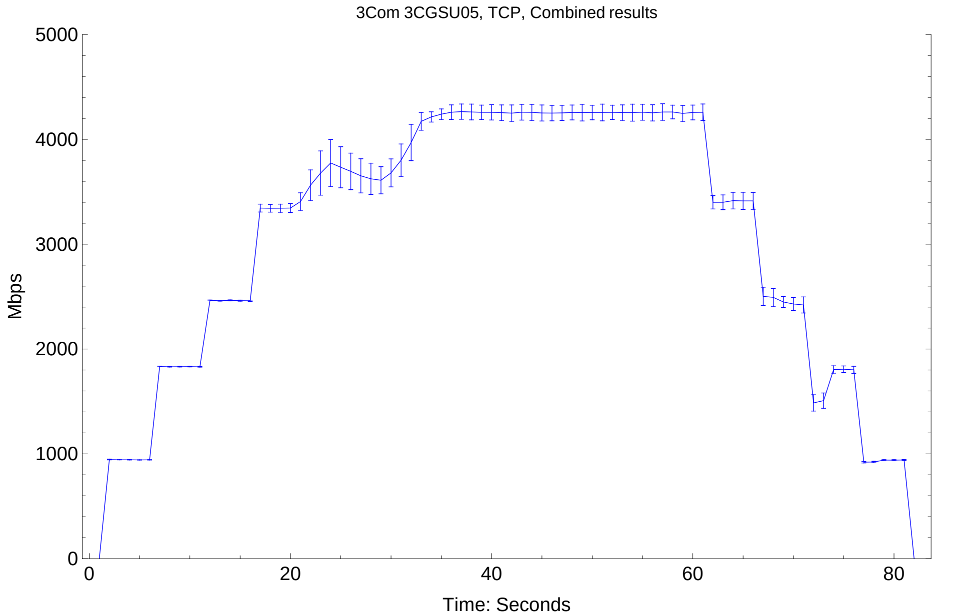


Figure 4.25: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.



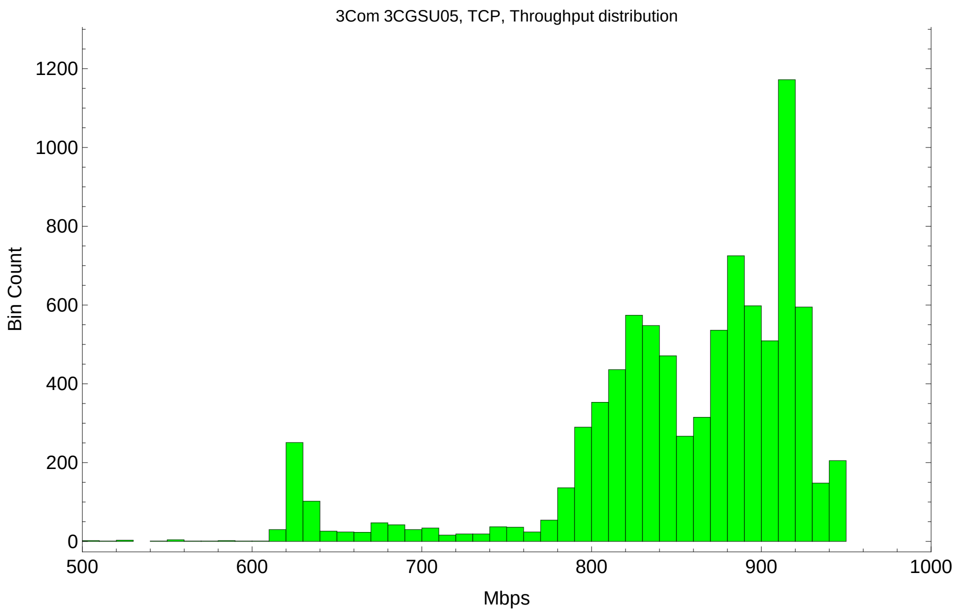


Figure 4.26: In the top graph the results for the 8 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.15 3Com 3CGSU05, UDP

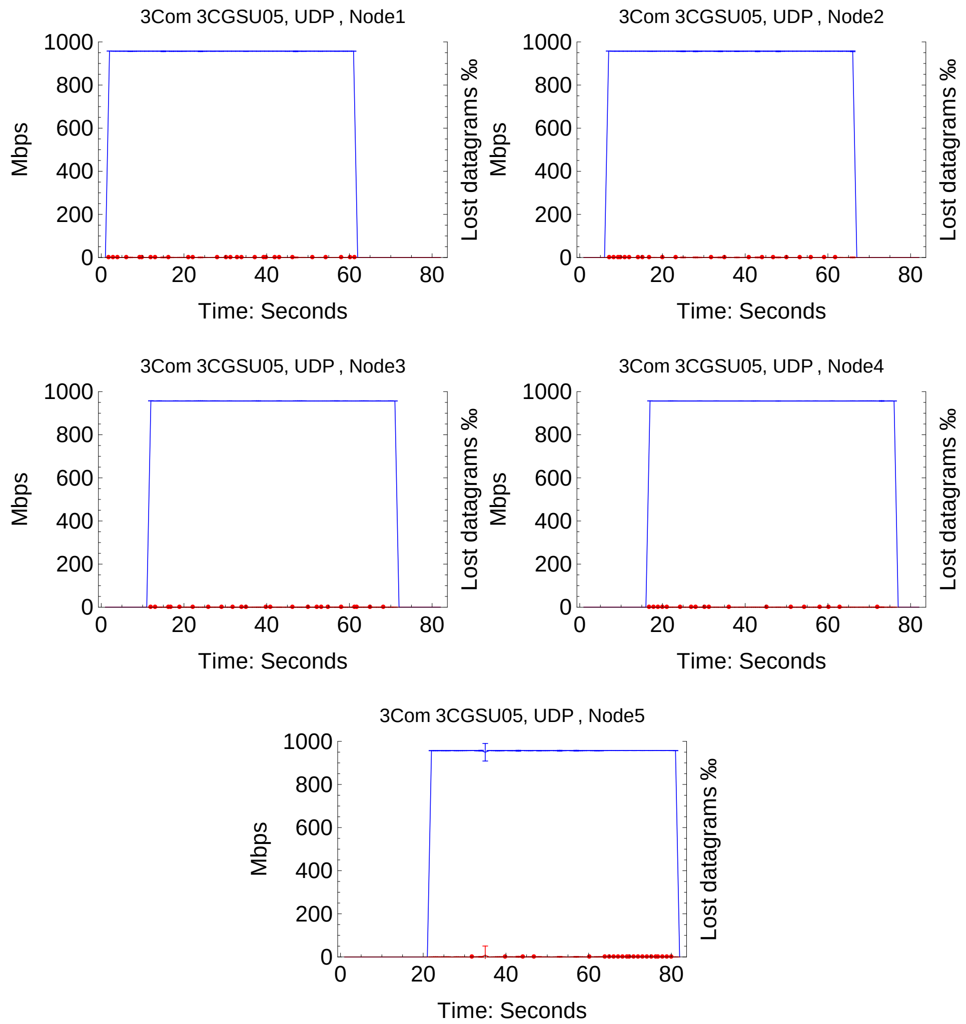


Figure 4.27: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

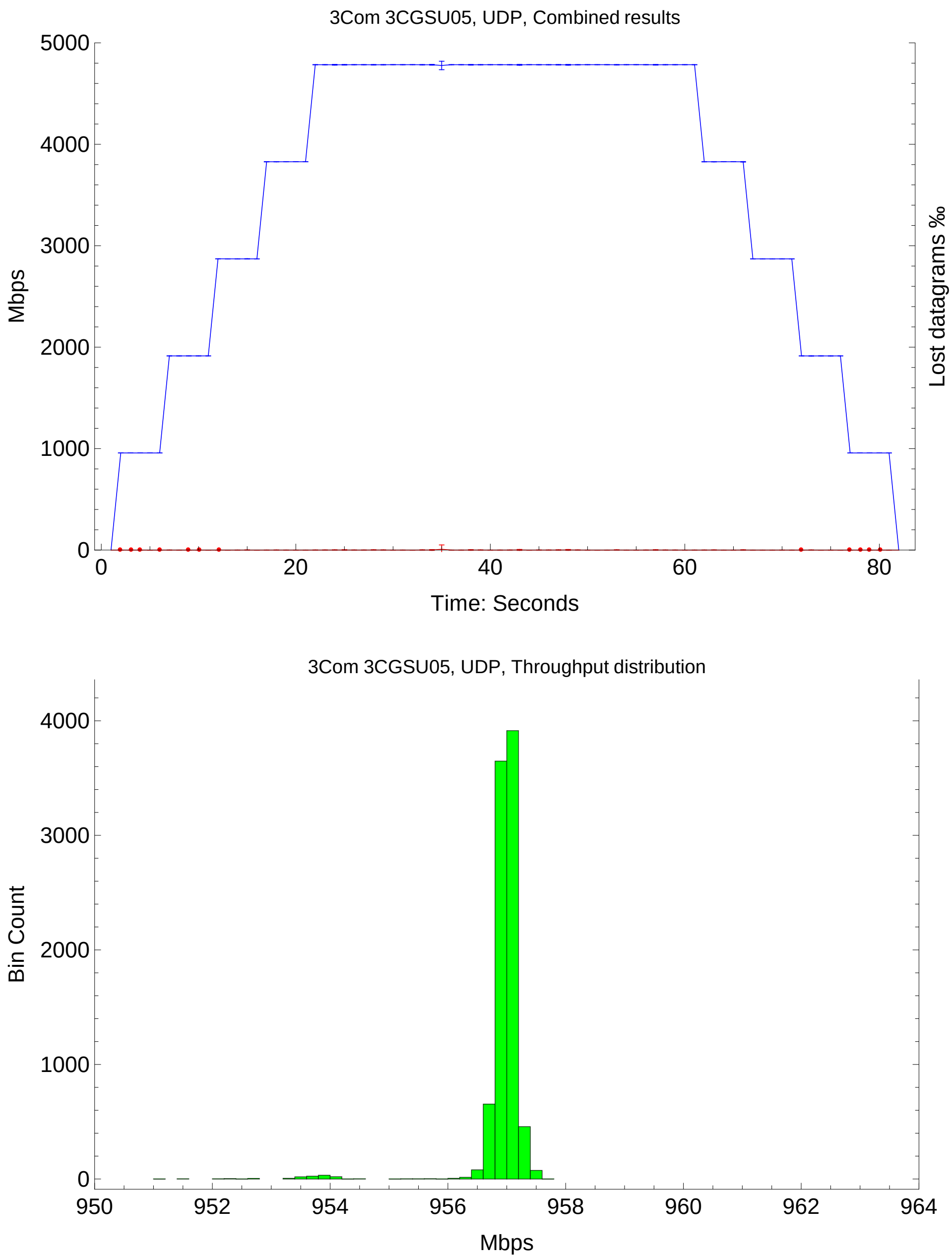


Figure 4.28: In the top graph the results for the 5 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.16 Cisco SG 100D-08, TCP

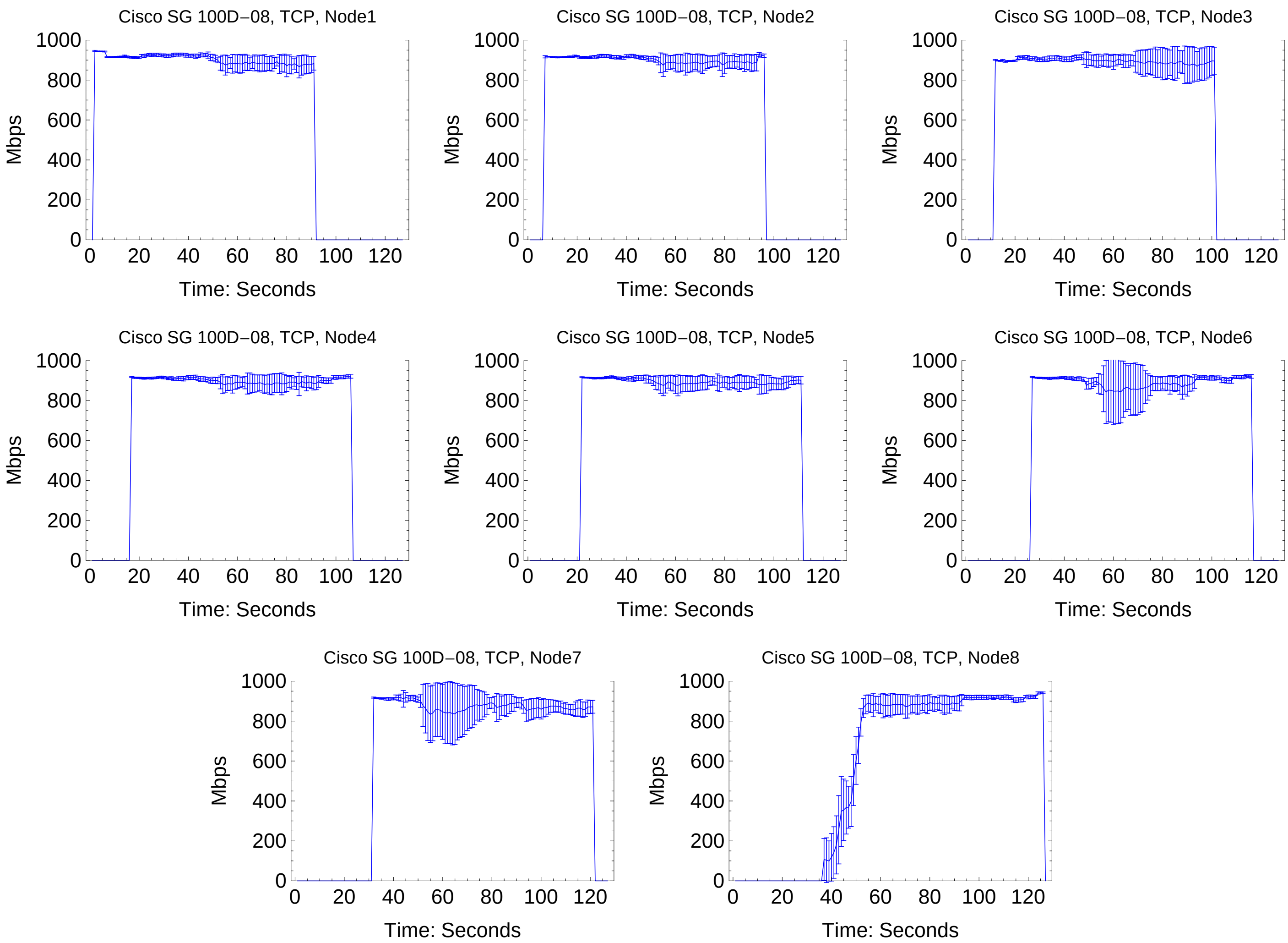


Figure 4.29: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

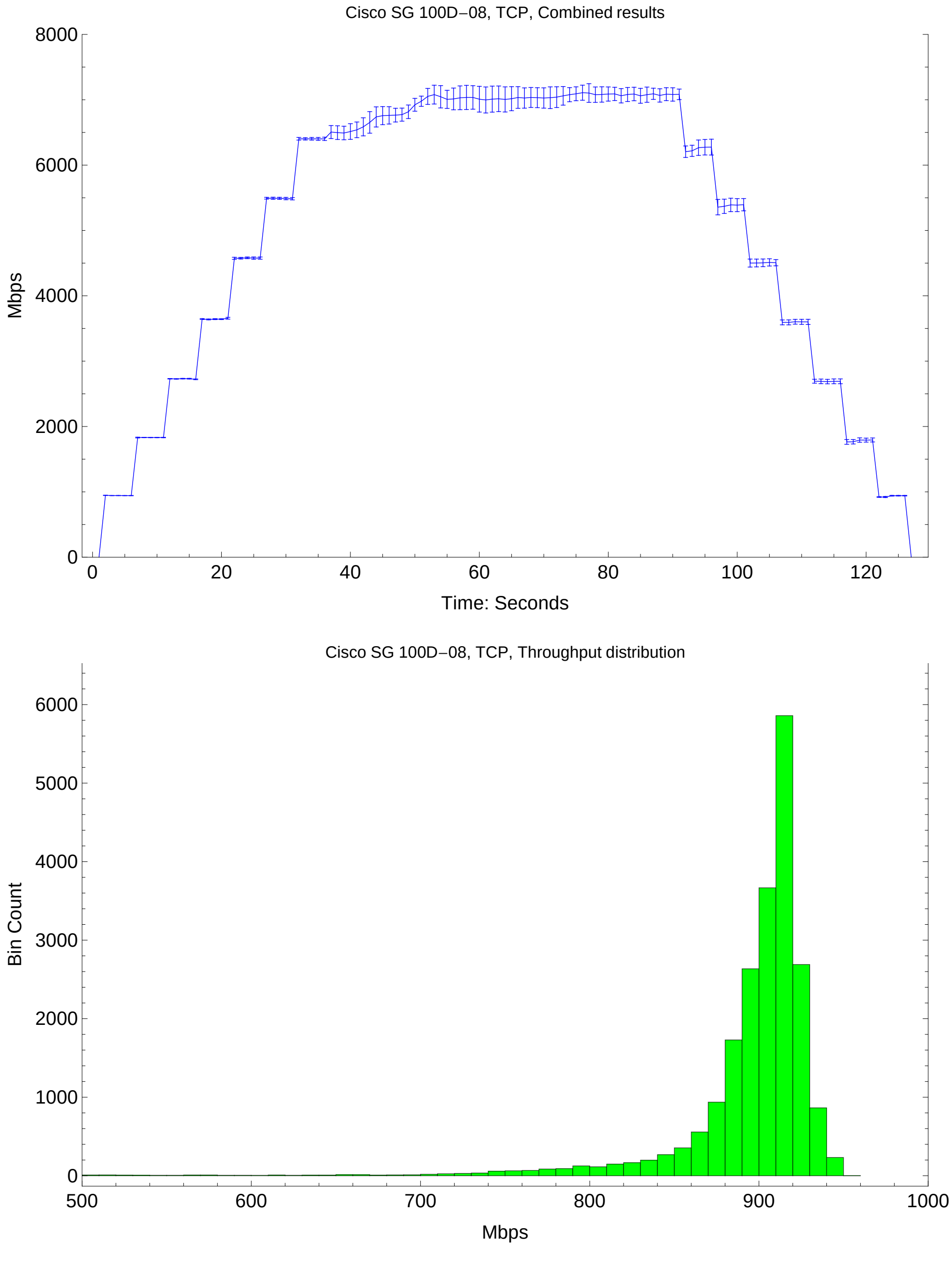


Figure 4.30: In the top graph the results for the 8 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.17 Cisco SG 100D-08, UDP

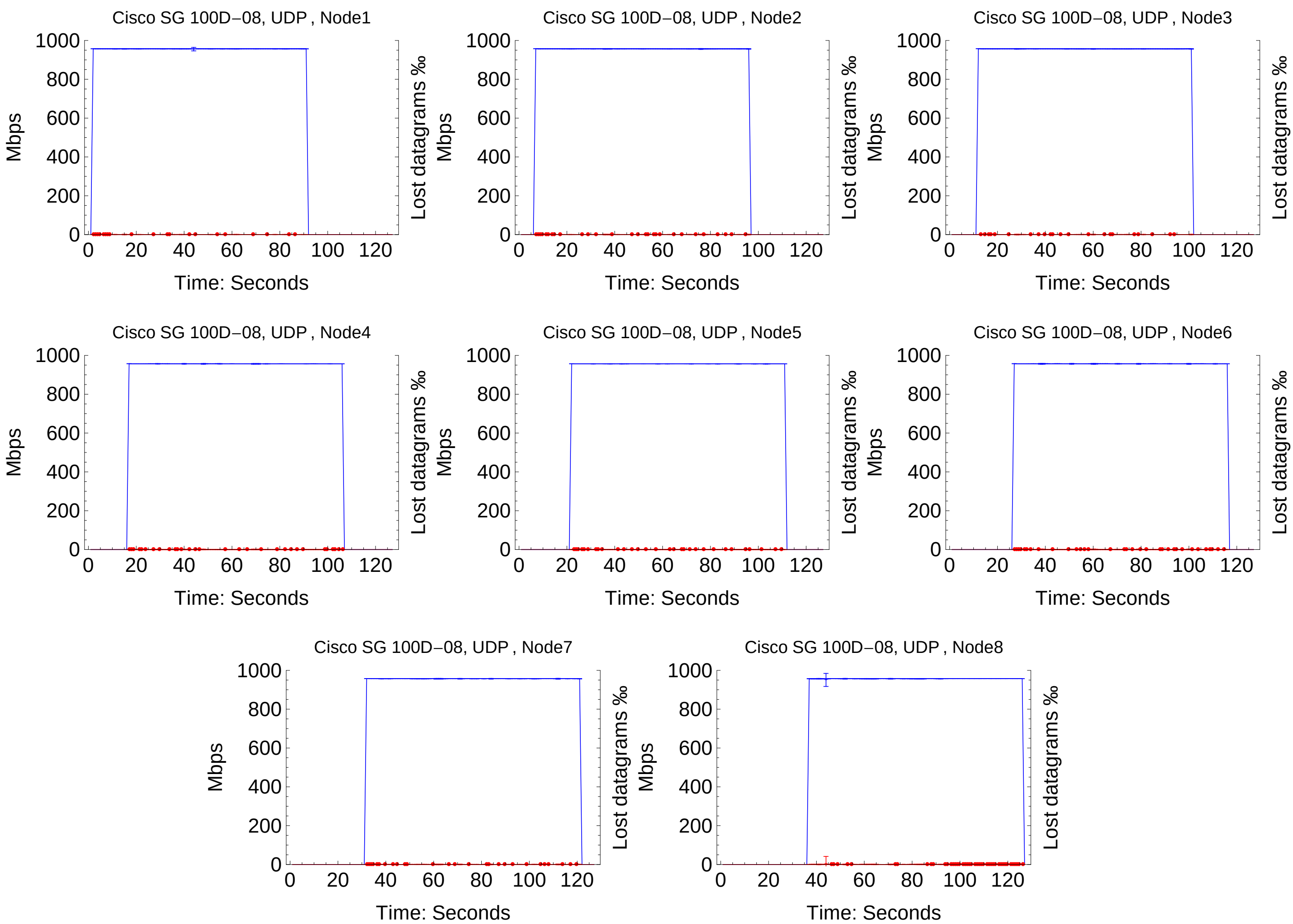


Figure 4.31: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

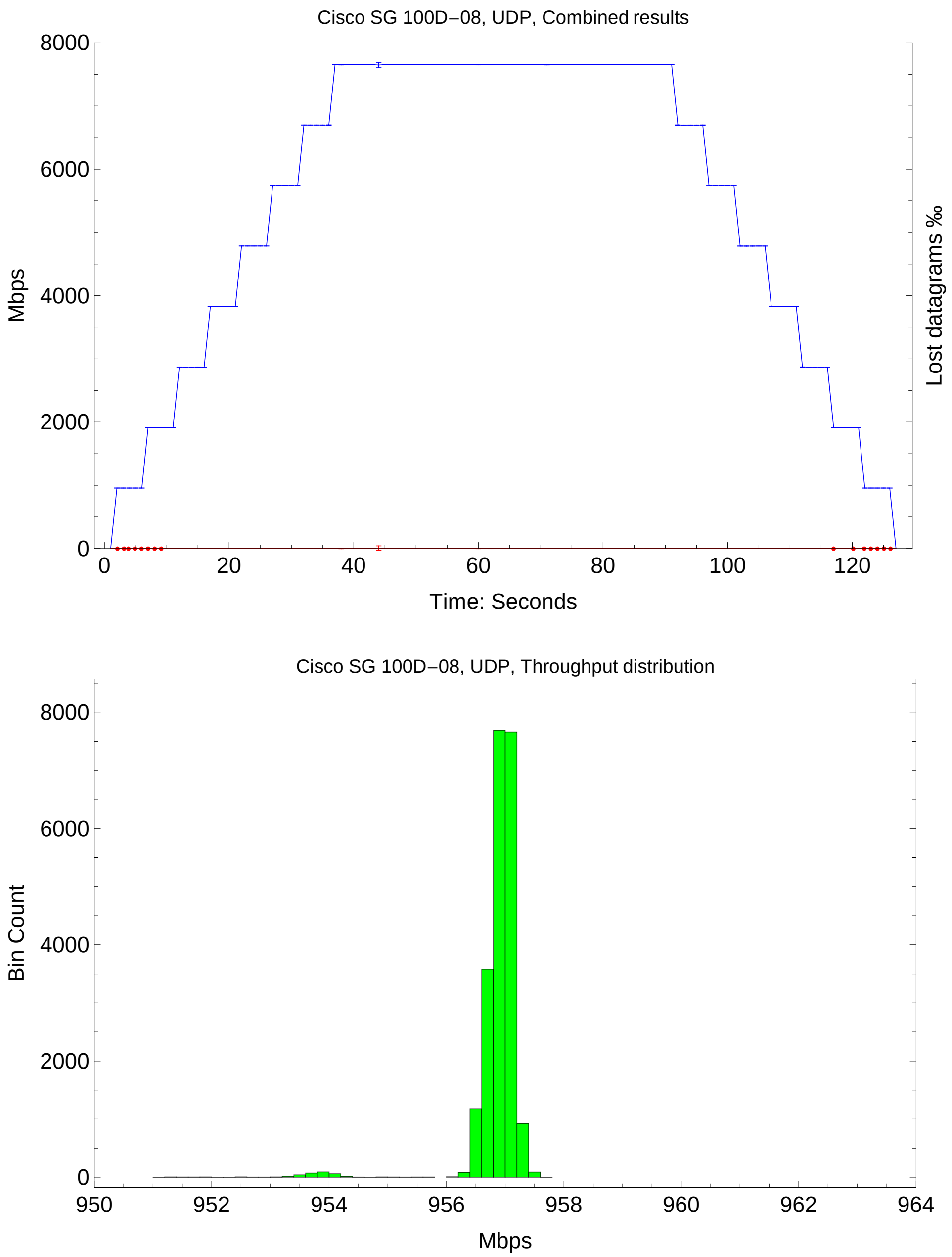


Figure 4.32: In the top graph the results for the 8 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.18 3Com 3CGSU08, TCP

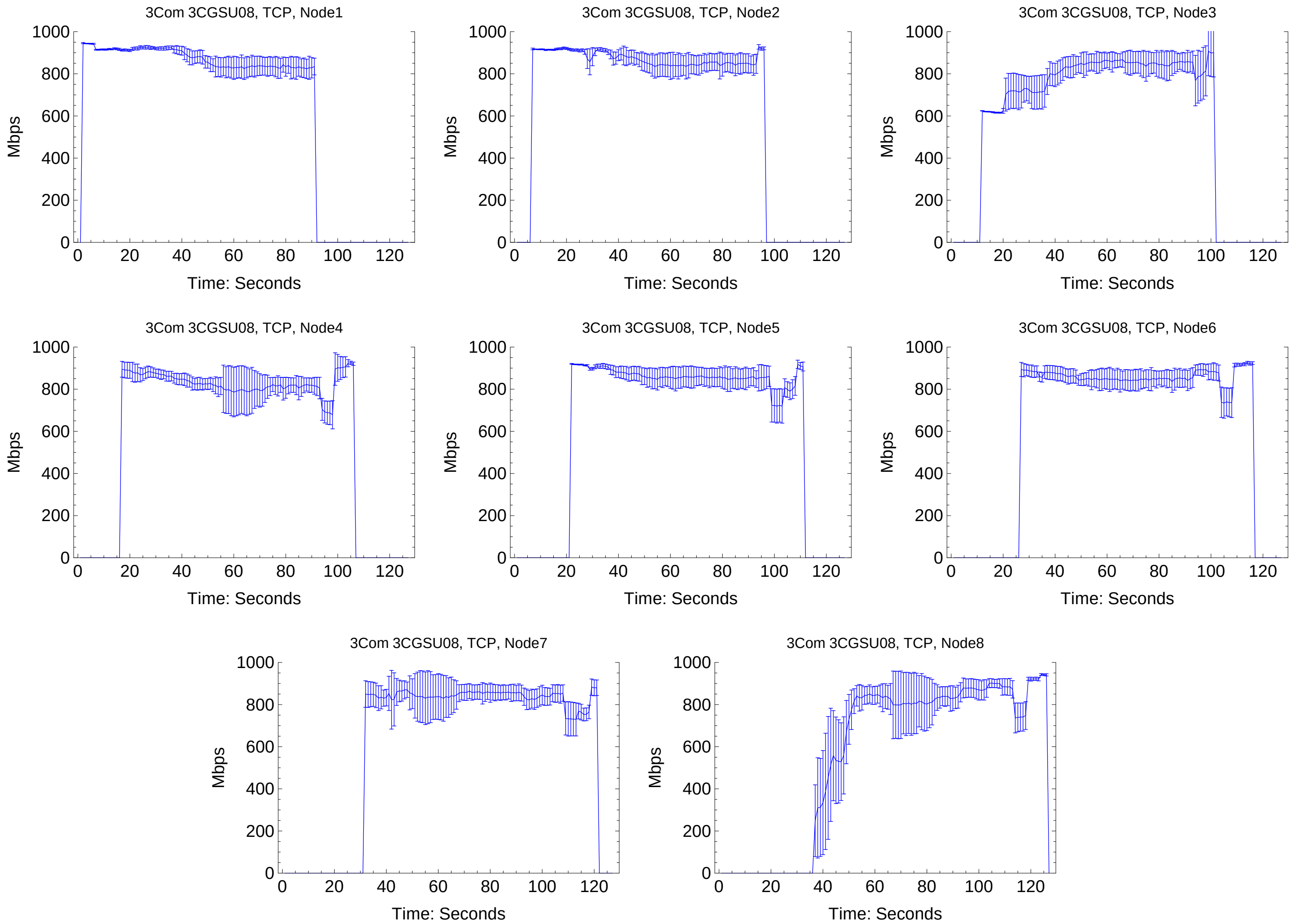
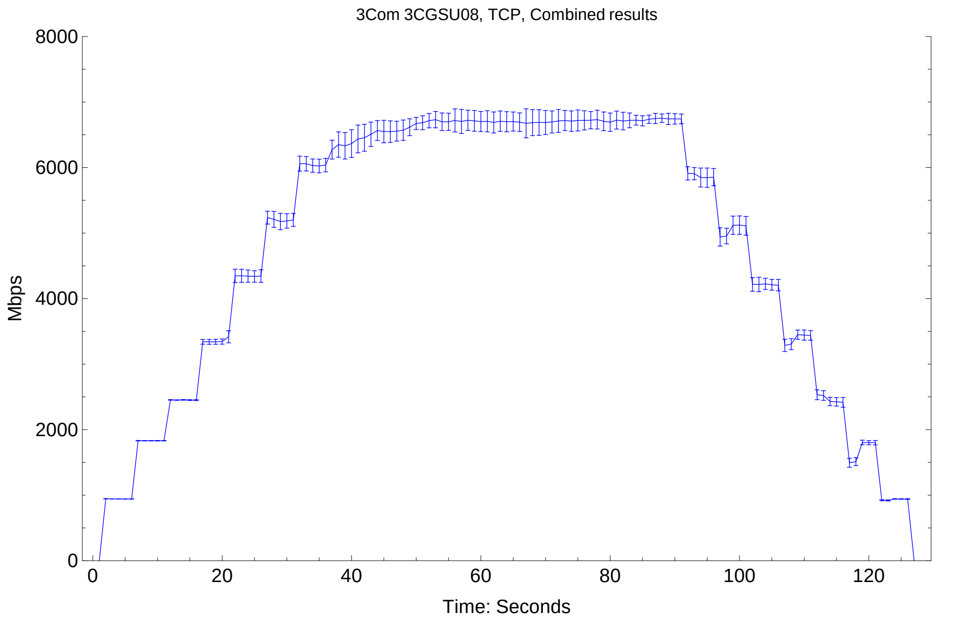


Figure 4.33: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.



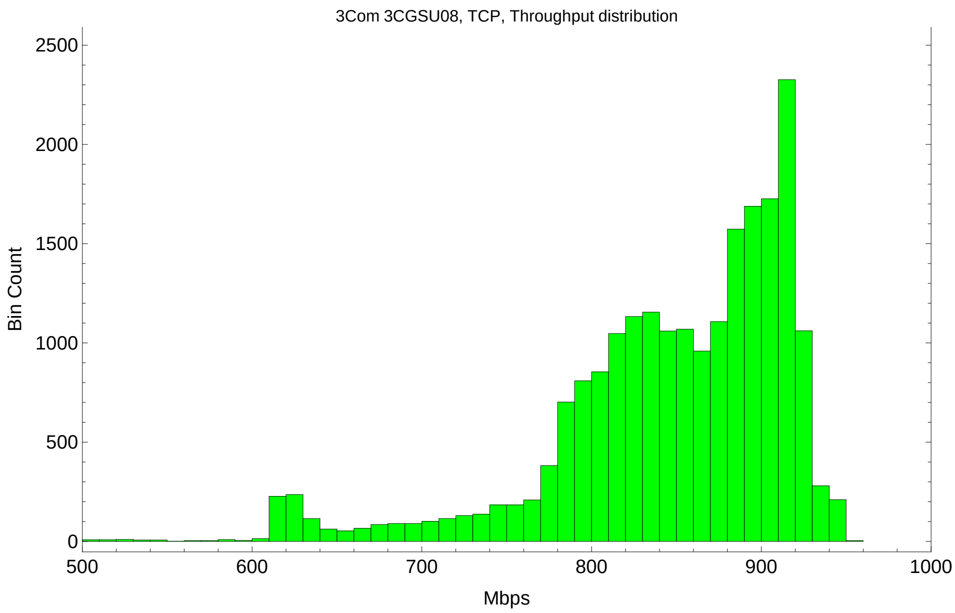


Figure 4.34: In the top graph the results for the 8 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.19 3Com 3CGSU08, UDP

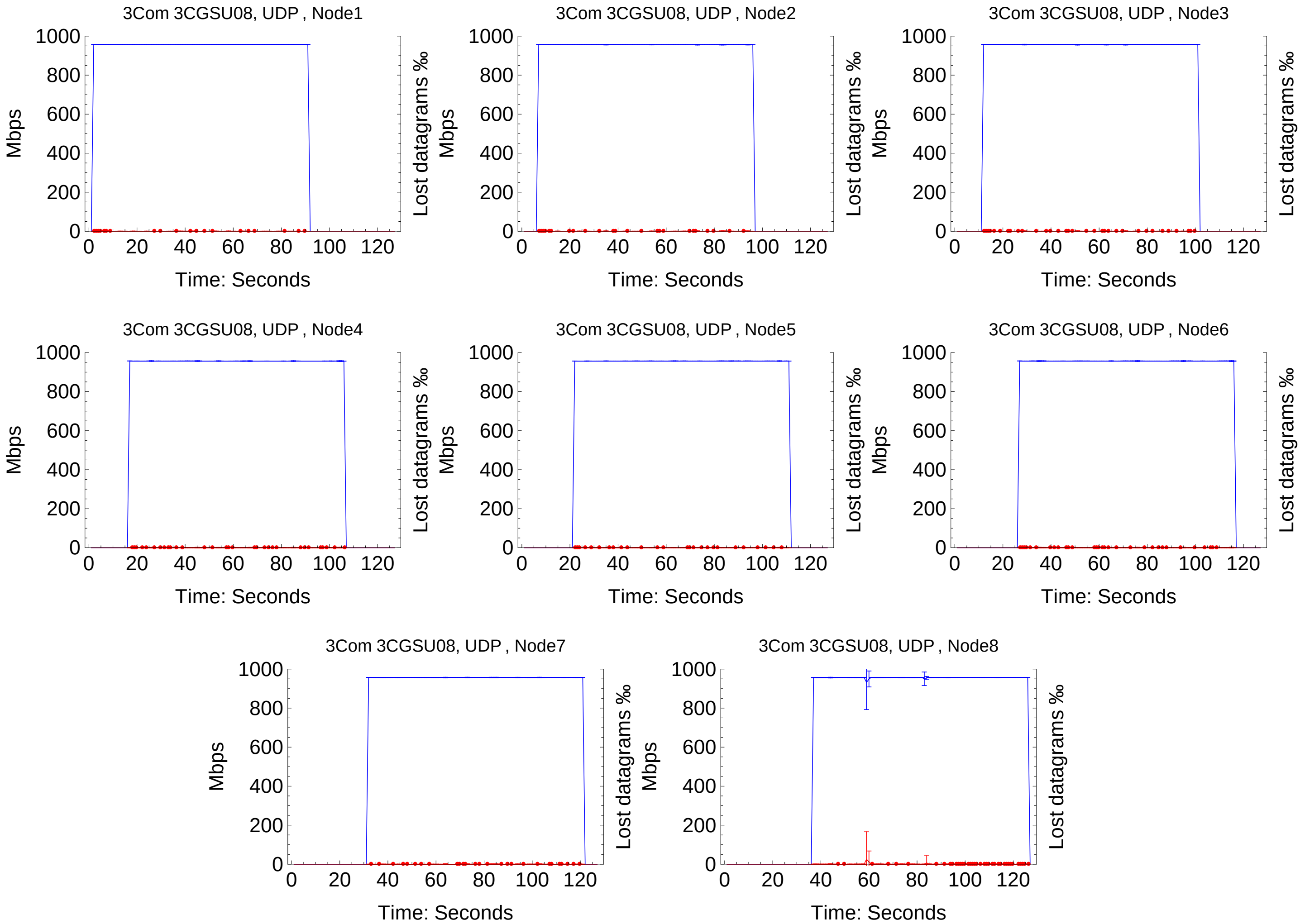


Figure 4.35: Benchmark results for the individual nodes. There is a 5 second interval between the startup of each node. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

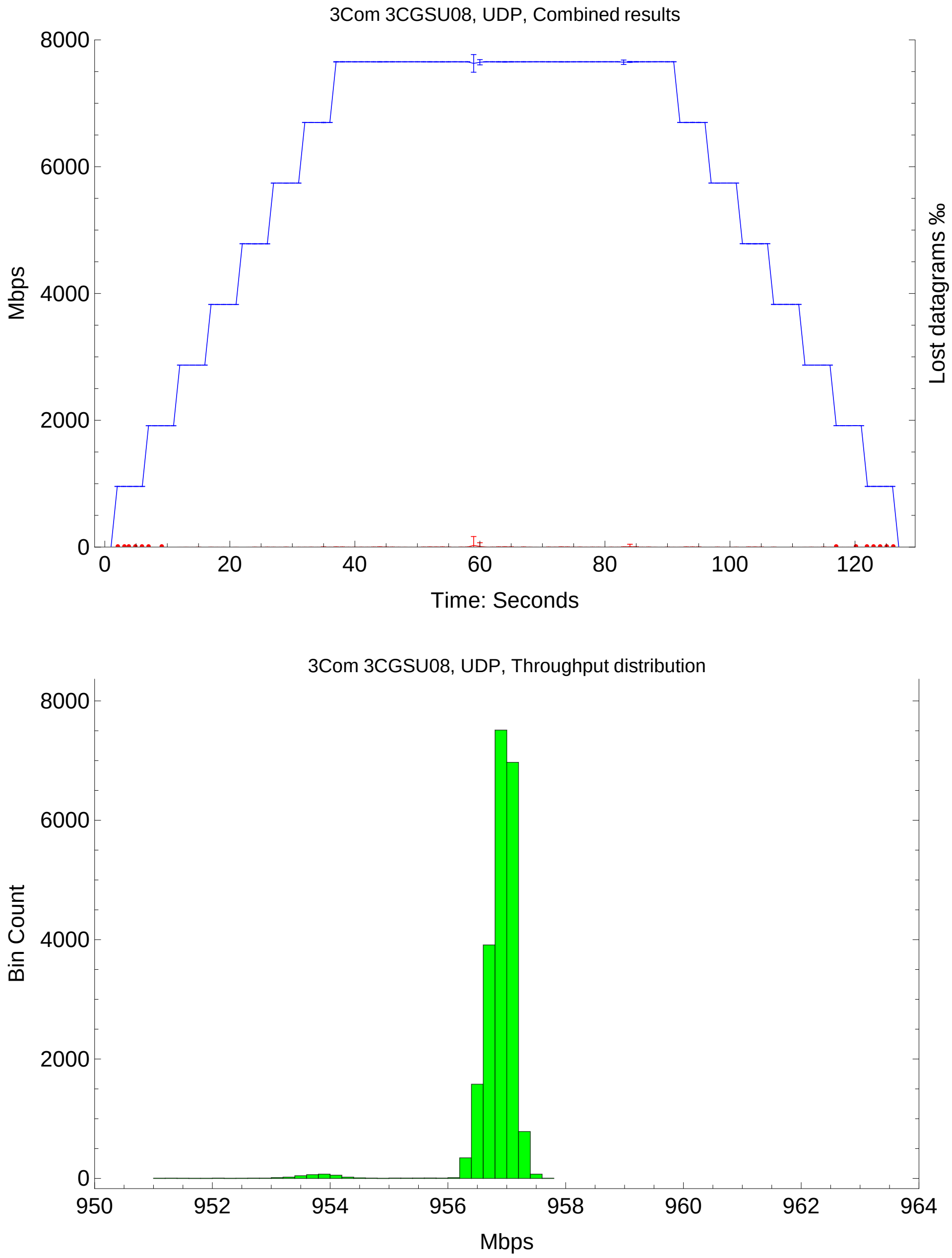


Figure 4.36: In the top graph the results for the 8 nodes are combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second. In the bottom histogram the benchmark throughput distribution is shown.

4.1.20 TCP throughput performance statistics

Device	Mean	Median	Mode	Max	S	95% CI
Baseline Cat6	802.373	908.296	941.425	968.36	232.71	794.769, 809.978
HP V1405C-5	86.569	90.702	91.750	108.134	15.029	86.257, 86.880
Dlink DGS-1005D	888.526	913.375	915.866	950.141	118.292	886.082, 890.970
Netgear GS605	878.530	904.069	916.390	950.206	121.830	876.013, 881.048
Netgear ProSafe	826.057	900.071	918.487	955.908	170.618	822.532, 829.583
Cisco SD2005	879.225	903.610	916.914	949.813	119.329	876.759, 881.691
3Com 3CGSU05	831.486	869.761	917.438	949.223	137.774	828.639, 834.333
Cisco SG 100D-08	879.378	909.050	915.341	949.223	120.935	877.403, 881.353
3Com 3CGSU08	839.663	865.796	916.914	947.585	107.530	837.906, 841.419

Table 4.1: A table with the TCP throughput statistics.

Device	Sample Count	$\bar{x} - \bar{y}$	$S_{\bar{x}}$	$S_{\bar{x}-\bar{y}}$	P value
Baseline Cat6	3600	N/A	3.879	N/A	N/A
Dlink DGS-1005D	9000	+ 86.153	1.247	5.125	0.
Netgear GS605	9000	+ 76.157	1.284	5.163	0.
Netgear ProSafe	9000	+ 23.684	1.798	5.677	0.00003
Cisco SD2005	9000	+ 76.852	1.258	5.136	0.
3Com 3CGSU05	9000	+ 29.113	1.452	5.331	0.
Cisco SG 100D-08	21600	+ 81.238	0.694	4.573	0.
3Com 3CGSU08	21600	+ 37.737	0.663	4.542	0.

Table 4.2: A table with the statistical significance of the difference between the TCP throughput sample mean over the TCP baseline sample mean.

ANOVA	DF	Sum of squares (SoS)	$\frac{SoS}{DF}$	F Statistic	P value
SSB	4	3.14×10^7	7.85×10^6	430.535	0.
SSW	88193	8.20×10^8	18233.4	N/A	N/A
SST	88199	8.52×10^8	N/A	N/A	N/A

Device	Deviates from	P value, less than
Dlink DGS-1005D	3Com 3CGSU05 Netgear GS605 Cisco SD2005 Netgear ProSafe	0.01
Netgear GS605	3Com 3CGSU05, Netgear ProSafe	0.01
Cisco SD2005	3Com 3CGSU05, Netgear ProSafe	0.01

Table 4.3: The ANOVA F-test statistics for the 5 port switches, and the Bonferroni post hoc test results revealing which devices significantly differs and at what significance level. Abbreviations: SSB (Sum of Squares Between), SSW (Sum of Squares Within), SST (Sum of Squares Total) and DF (Degrees of Freedom).

4.1.21 UDP throughput performance statistics

Device	Mean	Median	Mode	Max	S	95% CI
Baseline Cat6	956.626	957.041	956.982	957.523	10.625	956.279, 956.973
HP V1405C-5	95.702	95.703	95.703	95.773	0.008	95.702, 95.702
Dlink DGS-1005D	956.579	957.005	957.099	958.040	14.356	956.282, 956.876
Netgear GS605	956.759	956.817	957.099	957.546	0.579	956.747, 956.771
Netgear ProSafe	956.944	957.005	957.076	957.711	0.544	956.933, 956.955
Cisco SD2005	956.514	957.041	957.099	957.652	16.516	956.173, 956.855
3Com 3CGSU05	956.904	956.994	957.088	957.711	2.425	956.854, 956.954
Cisco SG 100D-08	956.859	956.970	957.099	957.605	1.428	956.839, 956.878
3Com 3CGSU08	956.782	956.958	957.099	957.605	5.589	956.708, 956.857

Table 4.4: A table with the UDP throughput statistics.

Device	Sample Count	$\bar{x} - \bar{y}$	$S_{\bar{x}}$	$S_{\bar{x}-\bar{y}}$	P value
Baseline Cat6	3600	N/A	0.1771	N/A	N/A
Dlink DGS-1005D	9000	− 0.0471	0.1513	0.3284	0.8860
Netgear GS605	9000	+ 0.1328	0.0061	0.1832	0.4684
Netgear ProSafe	9000	+ 0.3178	0.0057	0.1828	0.0822
Cisco SD2005	9000	− 0.1122	0.1741	0.3512	0.7493
3Com 3CGSU05	9000	+ 0.2782	0.0256	0.2027	0.1699
Cisco SG 100D-08	21600	+ 0.2324	0.0097	0.1868	0.2135
3Com 3CGSU08	21600	+ 0.1563	0.0380	0.2151	0.4675

Table 4.5: A table with the statistical significance of the difference between the UDP throughput sample mean over the UDP baseline sample mean.

ANOVA	DF	Sum of squares (SoS)	$\frac{SoS}{DF}$	F Statistic	P value
SSB	7	1599.94	228.563	3.819	0.00037
SSW	91792	5.4931×10^6	59.8429	N/A	N/A
SST	91799	5.4947×10^6	N/A	N/A	N/A

Device	Deviates from	P value, less than
Cisco SD2005	Netgear ProSafe	0.01
Cisco SD2005	Cisco SG 100D-08 , 3Com 3CGSU05	0.05
Dlink DGS-1005D	Netgear ProSafe	0.05

Table 4.6: The ANOVA F-test statistics, and the Bonferroni post hoc test results revealing which devices significantly differs and at what significance level. Abbreviations: SSB (Sum of Squares Between), SSW (Sum of Squares Within), SST (Sum of Squares Total) and DF (Degrees of Freedom).

4.1.22 Jitter statistics

Device	Mean	Median	Mode	Max	S	95% CI
Baseline Cat6	0.0166	0.015	0.015	1.98	0.0331	0.0156, 0.0177
HP V1405C-5	0.2299	0.208	0.151	0.422	0.0738	0.2283, 0.2314
Dlink DGS-1005D	0.0168	0.015	0.015	1.987	0.0311	0.0161, 0.0174
Netgear GS605	0.0162	0.015	0.015	0.058	0.0058	0.0161, 0.0163
Netgear ProSafe	0.0161	0.015	0.015	0.063	0.0056	0.0160, 0.0162
Cisco SD2005	0.0171	0.015	0.015	1.985	0.0376	0.0163, 0.0178
3Com 3CGSU05	0.0163	0.015	0.015	0.055	0.0059	0.0162, 0.0164
Cisco SG 100D-08	0.0162	0.015	0.015	0.055	0.0058	0.0162, 0.0163
3Com 3CGSU08	0.0163	0.015	0.015	2.114	0.0154	0.0161, 0.0165

Table 4.7: A table with the jitter statistics.

Device	Sample Count	$\bar{x} - \bar{y}$	$S_{\bar{x}}$	$S_{\bar{x}-\bar{y}}$	P value
Baseline Cat6	3600	N/A	0.00055	N/A	N/A
HP V1405C-5	9000	+ 0.21322	0.00078	0.00133	0.
Dlink DGS-1005D	9000	+ 0.00014	0.00033	0.00088	0.8752
Netgear GS605	9000	− 0.00041	0.00006	0.00061	0.5088
Netgear ProSafe	9000	− 0.00049	0.00006	0.00061	0.4263
Cisco SD2005	9000	+ 0.00043	0.00040	0.00095	0.6504
3Com 3CGSU05	9000	− 0.00033	0.00006	0.00061	0.5910
Cisco SG 100D-08	21600	− 0.00040	0.00004	0.00059	0.4959
3Com 3CGSU08	21600	− 0.00032	0.00011	0.00066	0.6246

Table 4.8: A table with the statistical significance of the difference between the jitter sample mean over the baseline jitter sample mean.

ANOVA	DF	Sum of squares (SoS)	$\frac{SoS}{DF}$	F Statistic	P value
SSB	7	0.00712	0.00102	2.9017	0.00494
SSW	91792	32.1774	0.00035	N/A	N/A
SST	91799	32.1845	N/A	N/A	N/A

Device	Deviates from	P value, less than
Cisco SD2005	Netgear ProSafe, 3Com 3CGSU08, Cisco SG 100D-08	0.05

Table 4.9: The ANOVA F-test statistics, and the Bonferroni post hoc test results revealing which devices significantly differs and at what significance level. Abbreviations: SSB (Sum of Squares Between), SSW (Sum of Squares Within), SST (Sum of Squares Total) and DF (Degrees of Freedom).

4.1.23 Datagram loss statistics

Device	Mean	Median	Mode	Max	S	95% CI
Baseline Cat6	0.0360	0	0	46.184	1.0642	0.0012, 0.0707
HP V1405C-5	0.0017	0	0	5.982	0.0954	-0.0003, 0.0037
Dlink DGS-1005D	0.0419	0	0	93.438	1.4933	0.0110, 0.0727
Netgear GS605	0.0073	0	0	1.082	0.0565	0.0061, 0.0084
Netgear ProSafe	0.0081	0	0	1.171	0.0554	0.0069, 0.0092
Cisco SD2005	0.0542	0	0	94.011	1.7254	0.0186, 0.0899
3Com 3CGSU05	0.0097	0	0	23.300	0.2529	0.0045, 0.0149
Cisco SG 100D-08	0.0084	0	0	19.214	0.1438	0.0064, 0.0103
3Com 3CGSU08	0.0133	0	0	76.900	0.5839	0.0055, 0.0211

Table 4.10: A table with the datagram loss statistics.

Device	Sample Count	$\bar{x} - \bar{y}$	$S_{\bar{x}}$	$S_{\bar{x}-\bar{y}}$	P value
Baseline Cat6	3600	N/A	0.0177	N/A	N/A
HP V1405C-5	9000	$-$ 0.0343	0.0010	0.0187	0.0675
Dlink DGS-1005D	9000	+ 0.0059	0.0157	0.0335	0.8602
Netgear GS605	9000	$-$ 0.0287	0.0006	0.0183	0.1173
Netgear ProSafe	9000	$-$ 0.0279	0.0006	0.0183	0.1276
Cisco SD2005	9000	+ 0.0183	0.0182	0.0359	0.6109
3Com 3CGSU05	9000	$-$ 0.0263	0.0027	0.0204	0.1979
Cisco SG 100D-08	21600	$-$ 0.0276	0.0010	0.0187	0.1401
3Com 3CGSU08	21600	$-$ 0.0227	0.0040	0.0217	0.2969

Table 4.11: A table with the statistical significance of the difference between the UDP datagram loss mean over the UDP baseline mean.

ANOVA	DF	Sum of squares (SoS)	$\frac{SoS}{DF}$	F Statistic	P value
SSB	7	23.1283	0.00102	5.1079	0.
SSW	91792	59375.7	0.64685	N/A	N/A
SST	91799	59398.8	N/A	N/A	N/A

Device	Deviates from	P value, less than
Cisco SD2005	3Com 3CGSU05 Netgear GS605 Netgear ProSafe Cisco SG 100D-08 3Com 3CGSU08	0.01
Dlink DGS-1005D	Cisco SG 100D-08	0.05

Table 4.12: The ANOVA F-test statistics, and the Bonferroni post hoc test results revealing which devices significantly differs and at what significance level. Abbreviations: SSB (Sum of Squares Between), SSW (Sum of Squares Within), SST (Sum of Squares Total) and DF (Degrees of Freedom).

4.2 Presenting the prototype

4.2.1 The program

The prototype was successfully created with the required functionality. The full source of the program can be found in appendix 7.3 on page 141. The program must be compiled before execution, and the makefile is provided in appendix 7.2 on page 141.

The main functionality of the program has been crated using parts of the C++ Boost library [3]. The program was created using boost version 1.48.0, and the following libraries where used.

- **Boost Asio** A cross-platform C++ library for network and low-level I/O programming.
- **Boost Program Options** A program options library that allow fetching of command-line and configuration file options
- **Boost Thread** A library that enables the use of multiple threads of execution with shared data.

For optimization of performance the program is heavily reliant on C++11 move semantics. Using R value references and Move semantics avoids unnecessary copying of data in memory, making the program more efficient. This feature requires the program to be compiled with GCC 4.6 and the -std=c++0x option for enabling the appropriate C++11 features. The intention was to transition to GCC 4.7 when it was released. The GCC 4.7 has been released at the time of writing, but there are compiling issues with the current released boost versions 1.48.0 and 1.49.0. When the boost library is released with full compatibility with GCC 4.7 the transition is expected to not cause any issues. Using GCC 4.7 the -std=c++0x option must be switched with the corresponding GCC 4.7 option -std=c++11 when compiled.

To execute the program the following commands can be used. Where the IP addresses of the nodes are 10.0.0.20, 10.0.0.21 and 10.0.0.22.

10.0.0.20

```
race -S 10.0.0.21 --file [PATH]
```

10.0.0.21

```
race -F 10.0.0.22
```

10.0.0.22

```
race -R
```

The command will result in the specified file being transferred from node 10.0.0.20 to node 10.0.0.21 and forwarded to 10.0.0.22.

Next a typical output of the program. This particular sample is taken from the virtualized environment that was used during development of the program.

Forward node sample output

```
Forward behavior invoked, main thread waiting at barrier
Receiving data..
Writing to disk..
Forwarding data..
recv_job: 1336071772.084879 1336071772.093053 1336071772.100391 1336071772.106883
recv_start: 1336071772.082438
recv_end: 1336071778.776722
Data received: avg receive speed 1493.811736 Mbit/s for 6.694284s
write_start: 1336071772.082439
write_end: 1336071778.777146
Write finished: avg write speed 1493.717350 Mbit/s for 6.694707s
fwd_job: 1336071772.088926 1336071772.093650 1336071772.100920 1336071772.107304
fwd_start: 1336071772.083712
fwd_end: 1336071778.777164
Forward finished: avg forward speed 1493.997417 Mbit/s for 6.693452s
Total exectution time: 6.69572 seconds
```

Invoking the program with option -h will print the correct usage.

Usage

```
Commandline options:

Generic options:
  -h [ --help ]                    Prints usage
  -v [ --version ]                 Prints version
  -b [ --buffer_size ] arg (=600)  Buffer size in MiB

Send options:
  -S [ --send ] arg                Invoke send behavior, specify host address
  -p [ --s_port ] arg (=9000)      Data out to port
  -f [ --file ] arg                Specify send file
  -c [ --job_size ] arg (=1460)    Define send/write/receive chunk size

Forward options:
  -F [ --forward ] arg             Invoke forward behavior, specify host address
  -i [ --fr_port ] arg (=9000)     Data in port
  -o [ --fs_port ] arg (=9000)     Data out to port

Receive options:
  -R [ --receive ]                 Invoke receive behavior
  -r [ --r_port ] arg (=9000)      Data in port
```

4.3 Comparative benchmarks

4.3.1 rTorrent throughput performance

Receive throughput

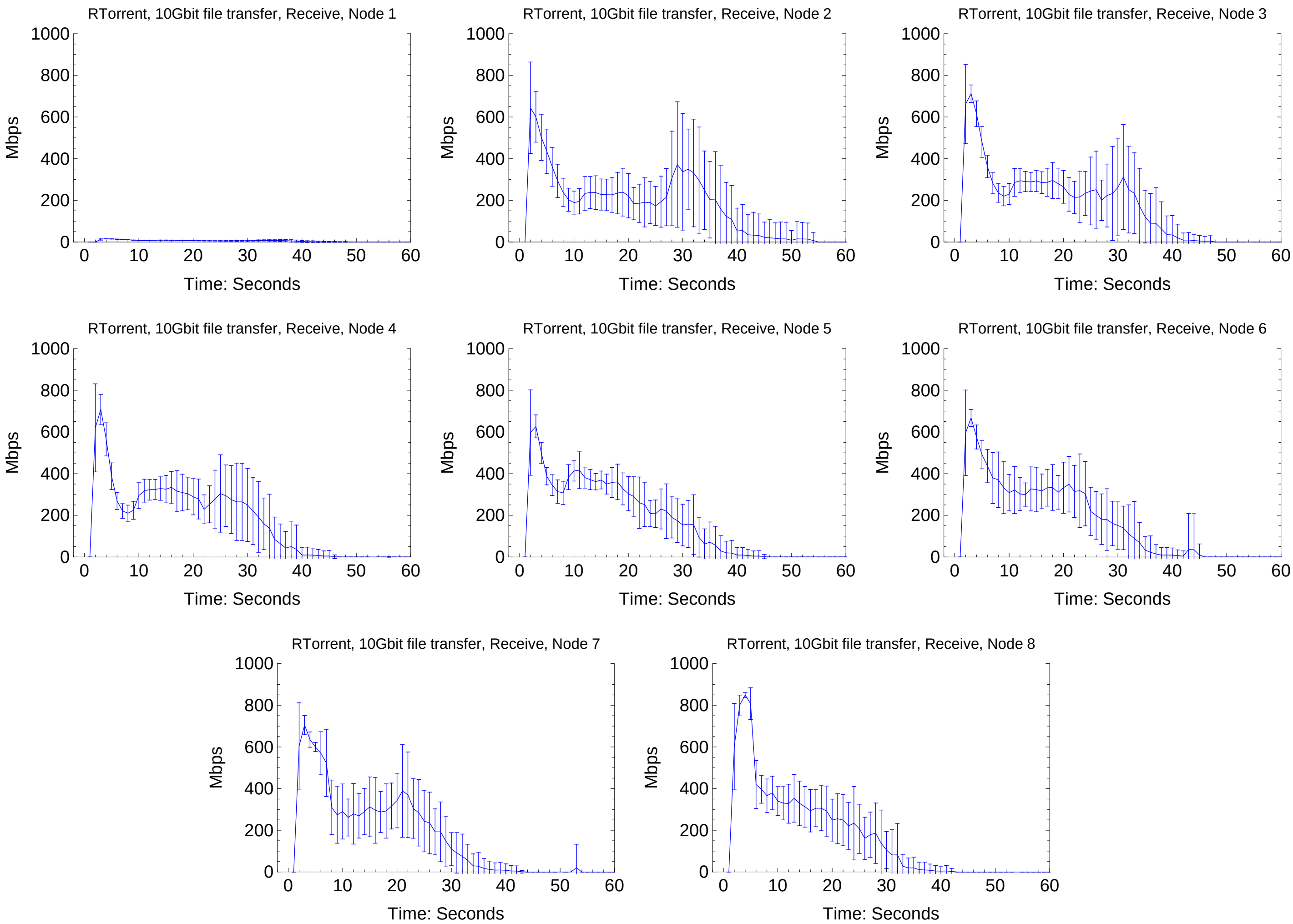


Figure 4.37: rTorrent receive throughput results for the individual nodes. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

Forward throughput

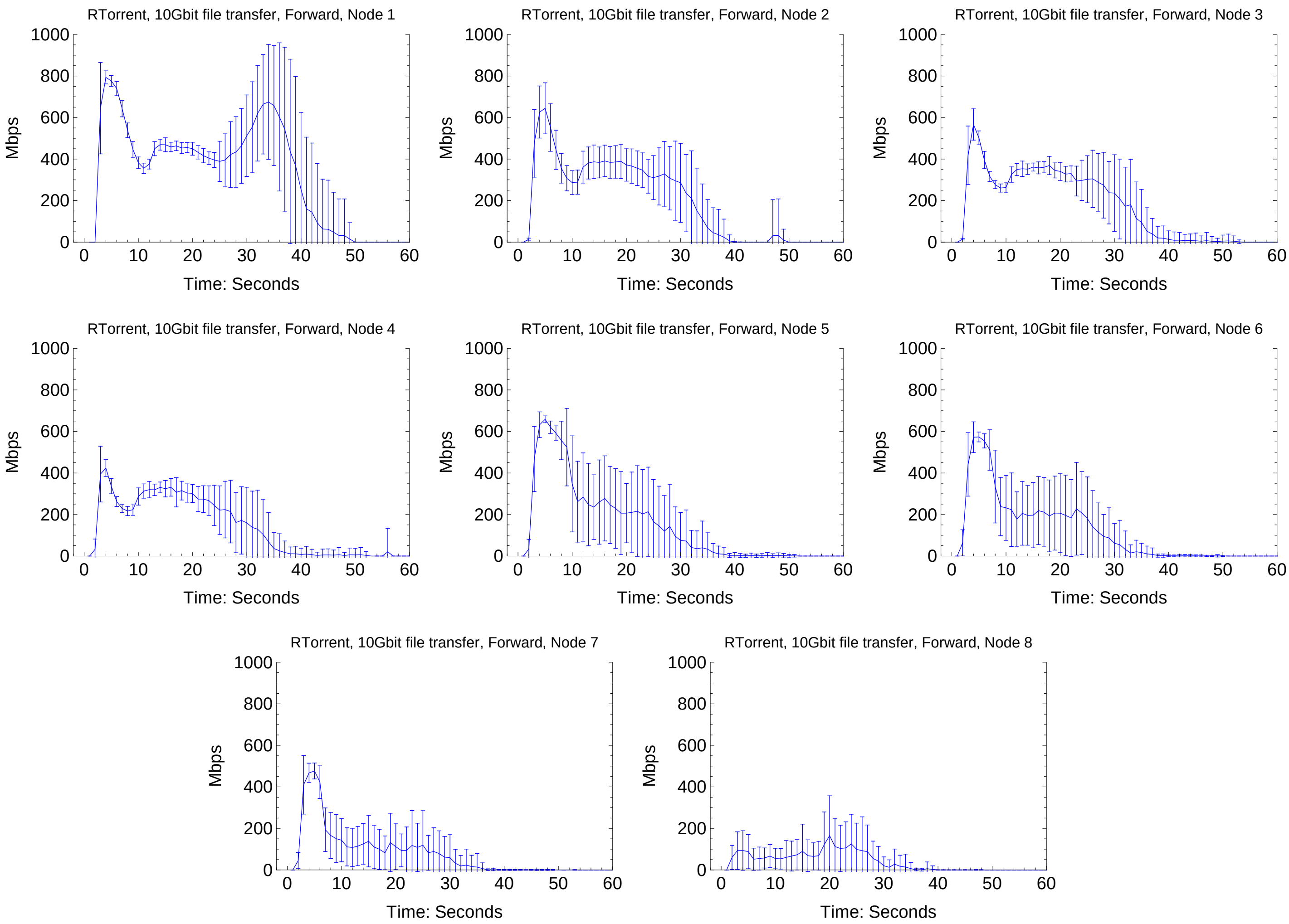


Figure 4.38: rTorrent forward throughput results for the individual nodes. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

Throughput aggregated

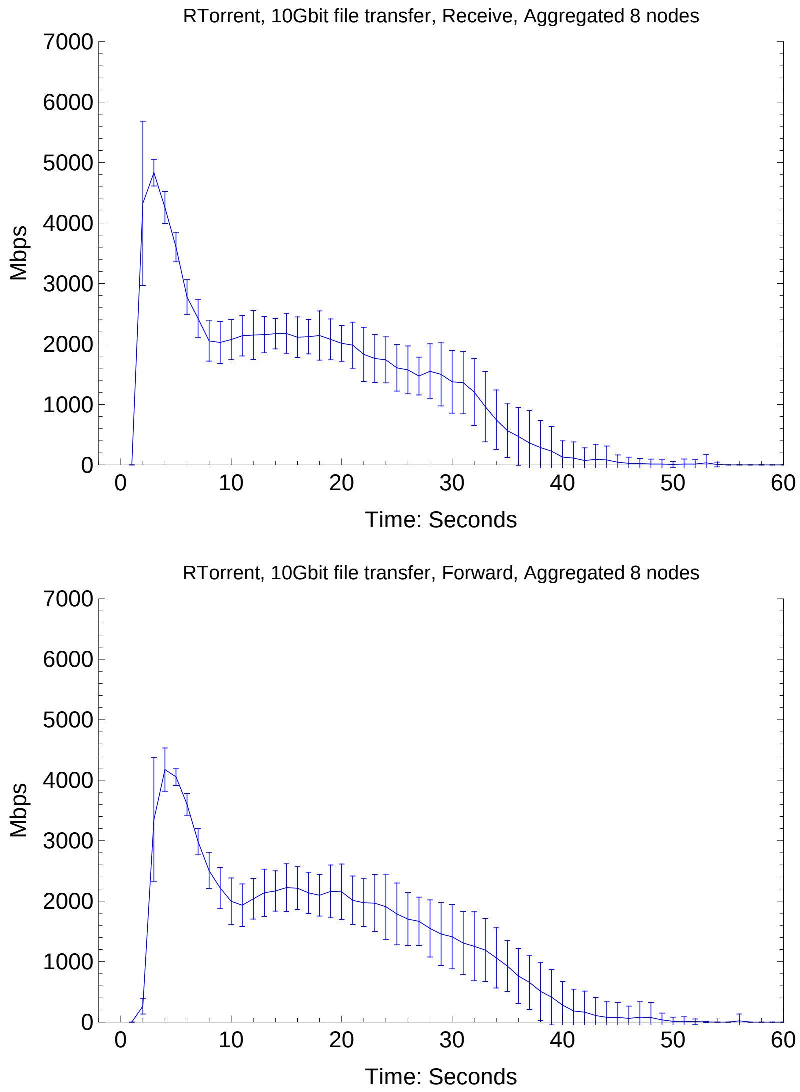


Figure 4.39: The rTorrent throughput results for the 8 nodes combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second.

4.3.2 Prototype throughput performance

Receive throughput

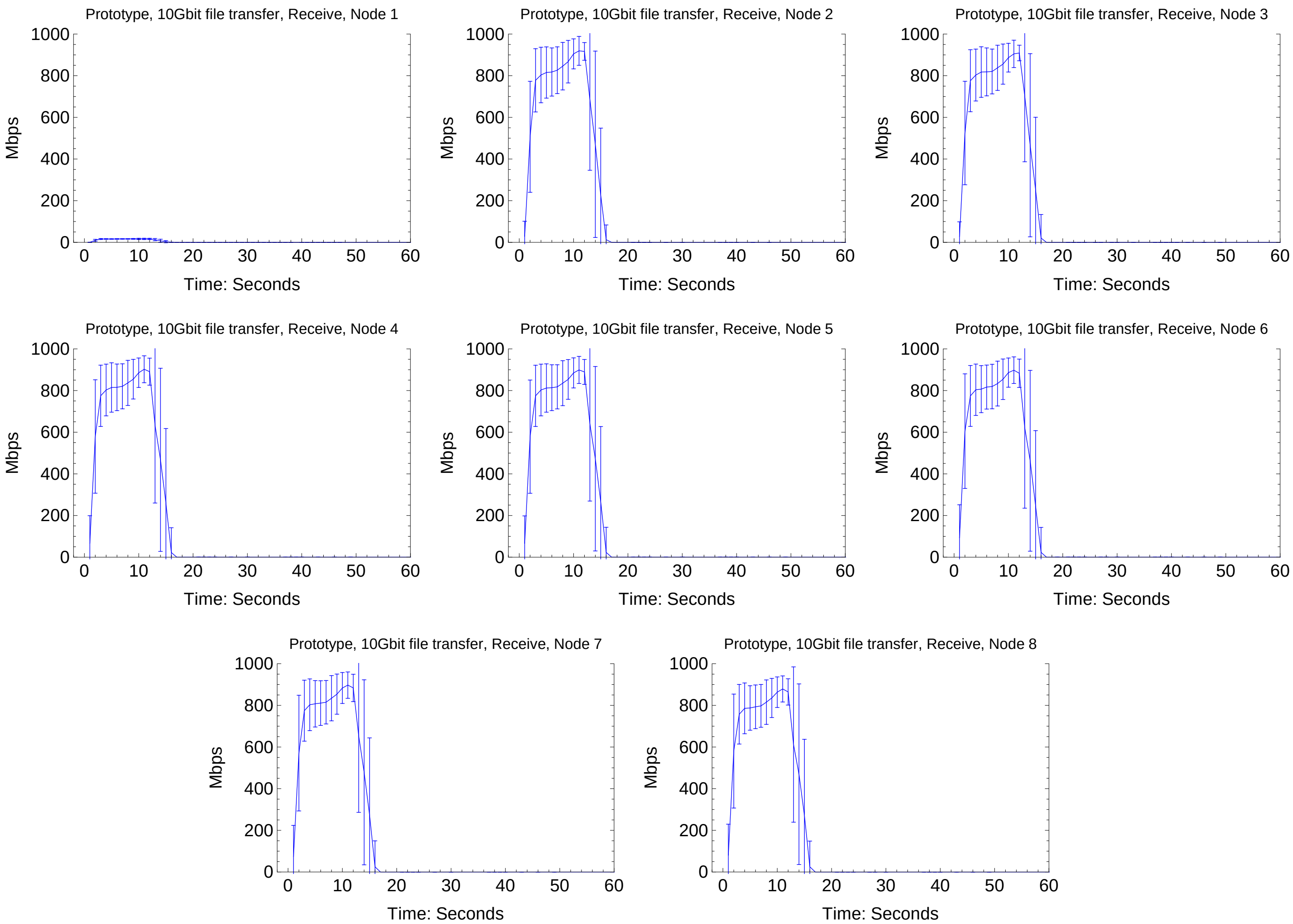


Figure 4.40: Prototype receive throughput results for the individual nodes. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

Forward throughput

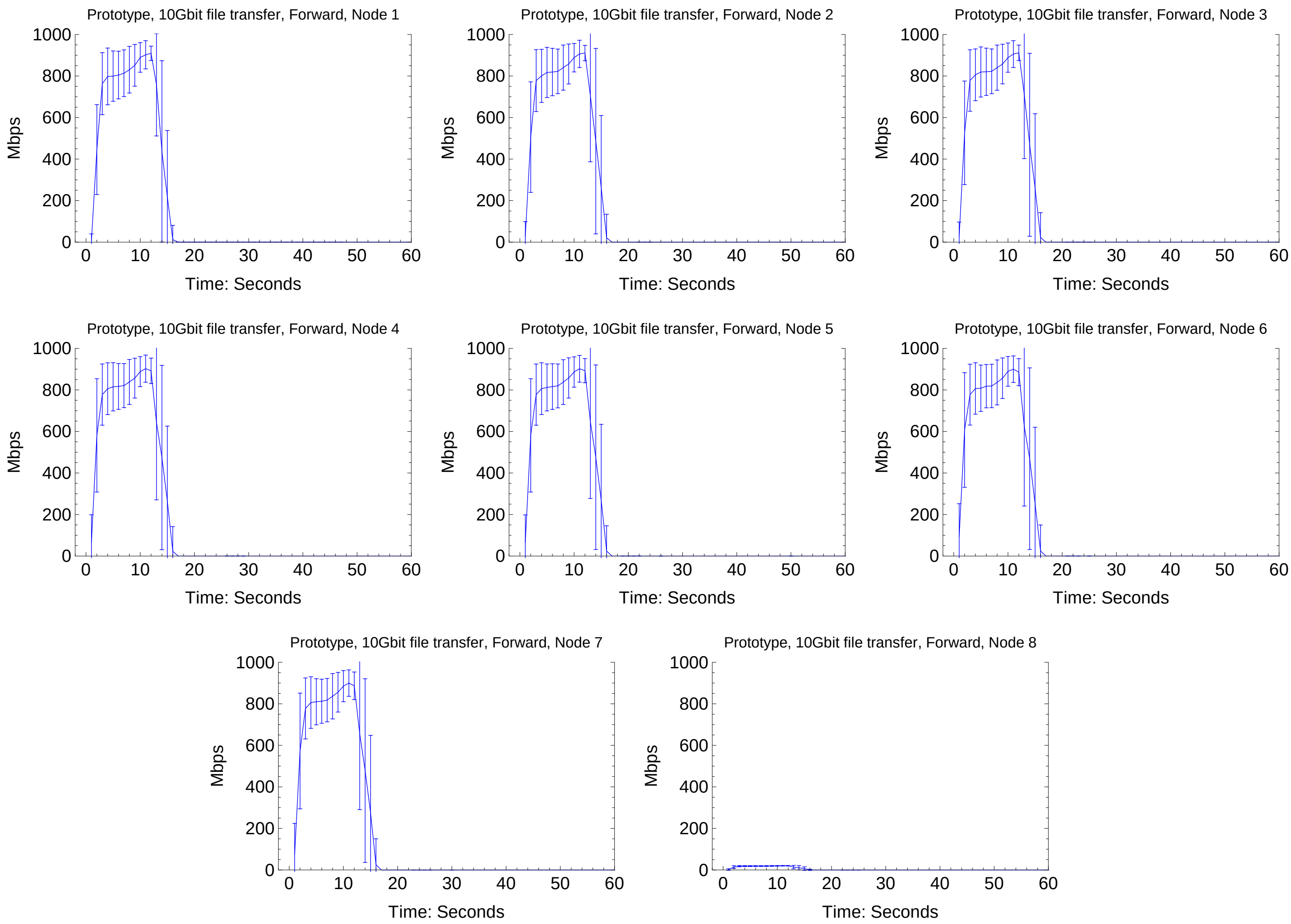
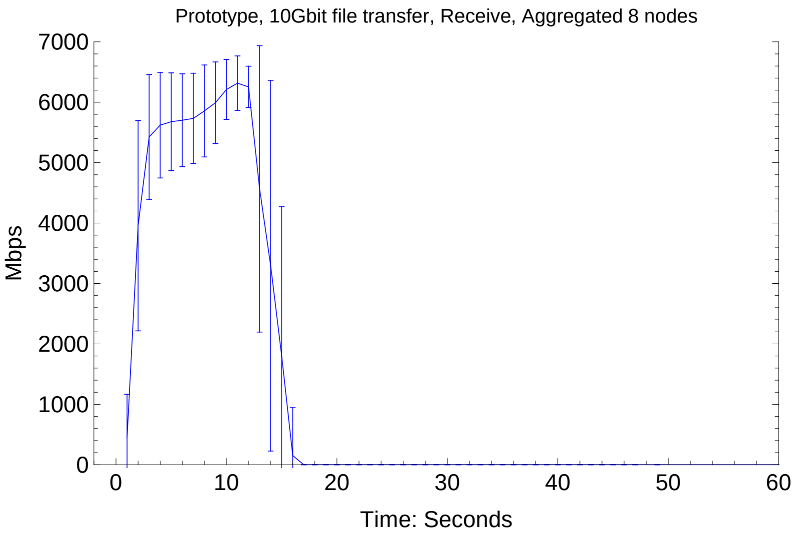


Figure 4.41: Prototype forward throughput results for the individual nodes. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

Throughput aggregated



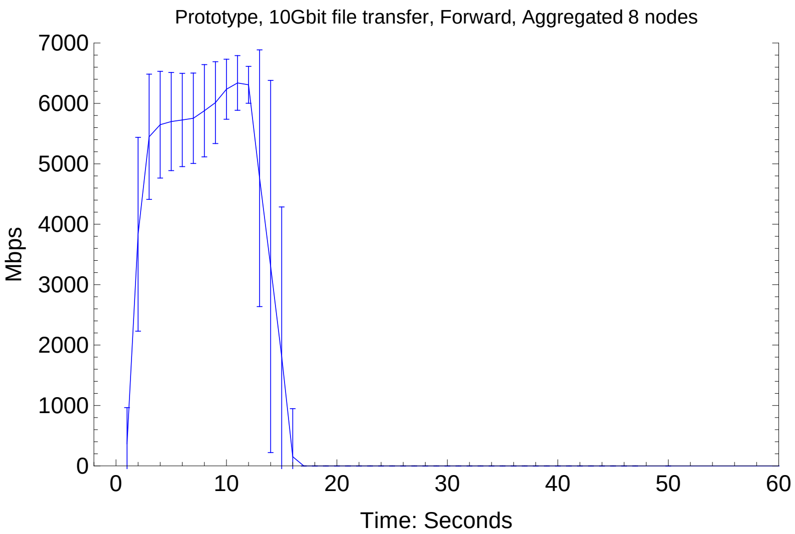


Figure 4.42: The prototype throughput results for the 8 nodes combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second.

4.3.3 rTorrent storage performance

Storage read performance

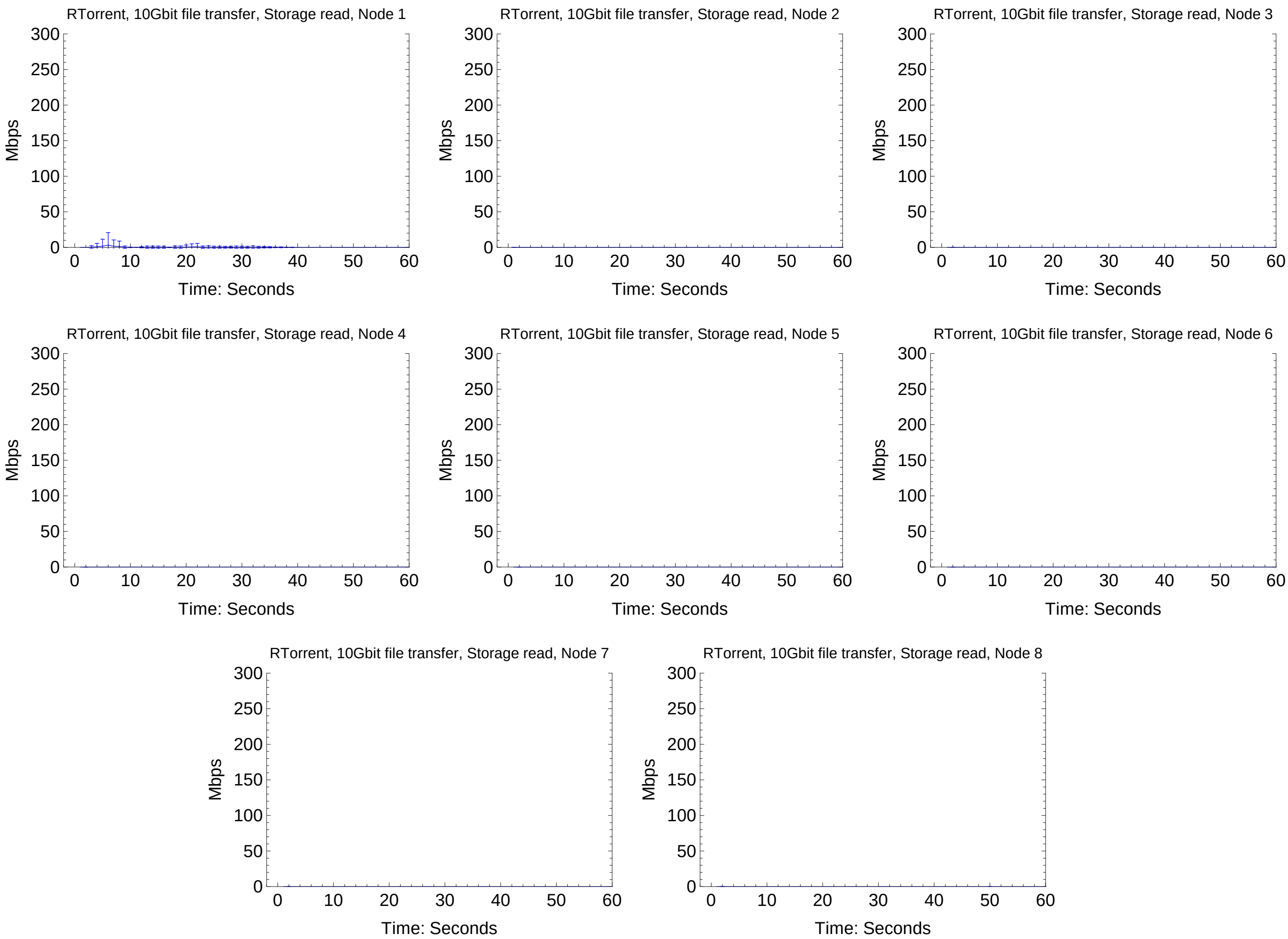


Figure 4.43: rTorrent storage read performance results for the individual nodes. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

Storage write performance

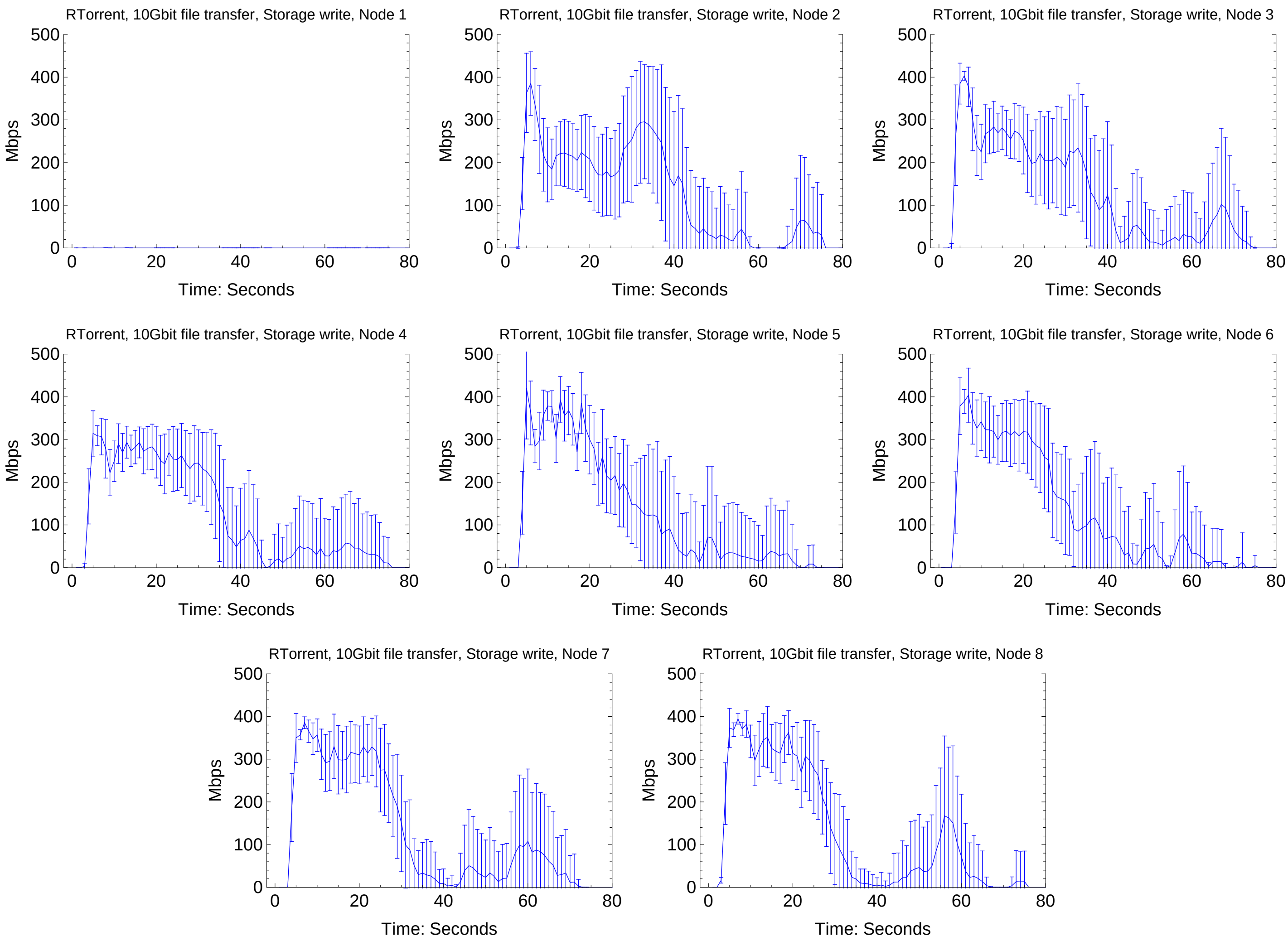


Figure 4.44: rTorrent storage write performance results for the individual nodes. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

Storage performance aggregated

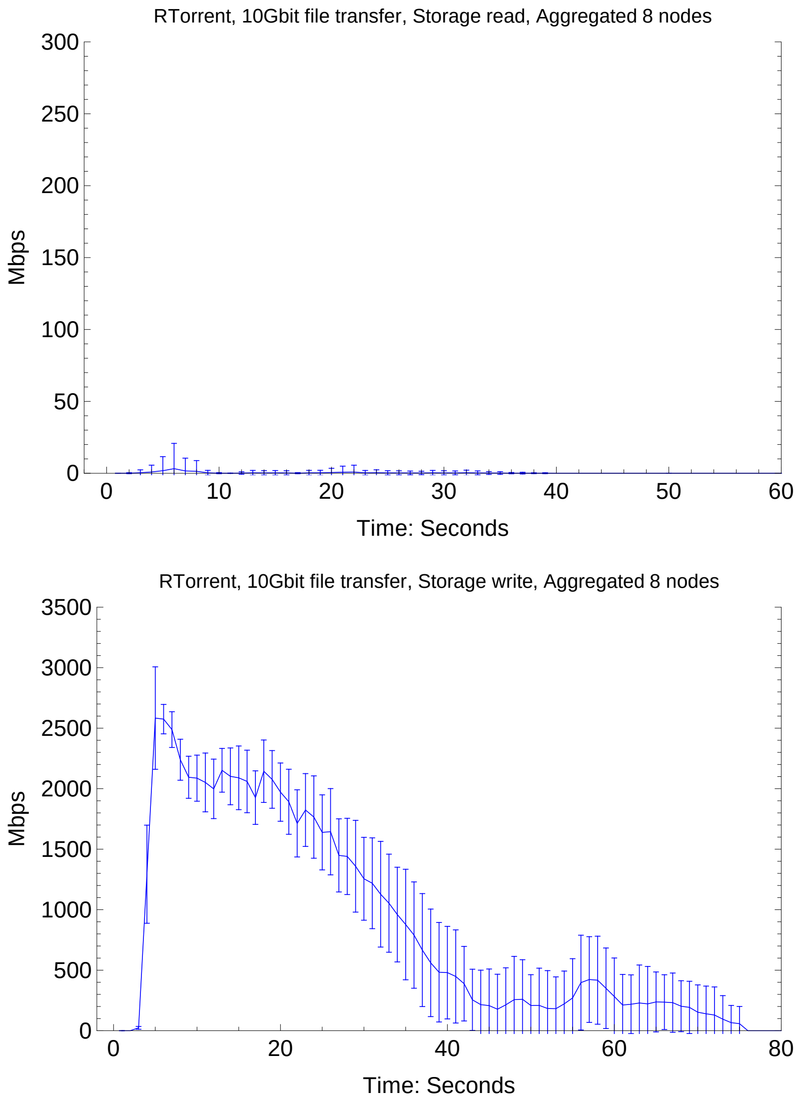


Figure 4.45: The rTorrent storage performance results for the 8 nodes combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second.

4.3.4 Prototype storage performance

Storage read performance

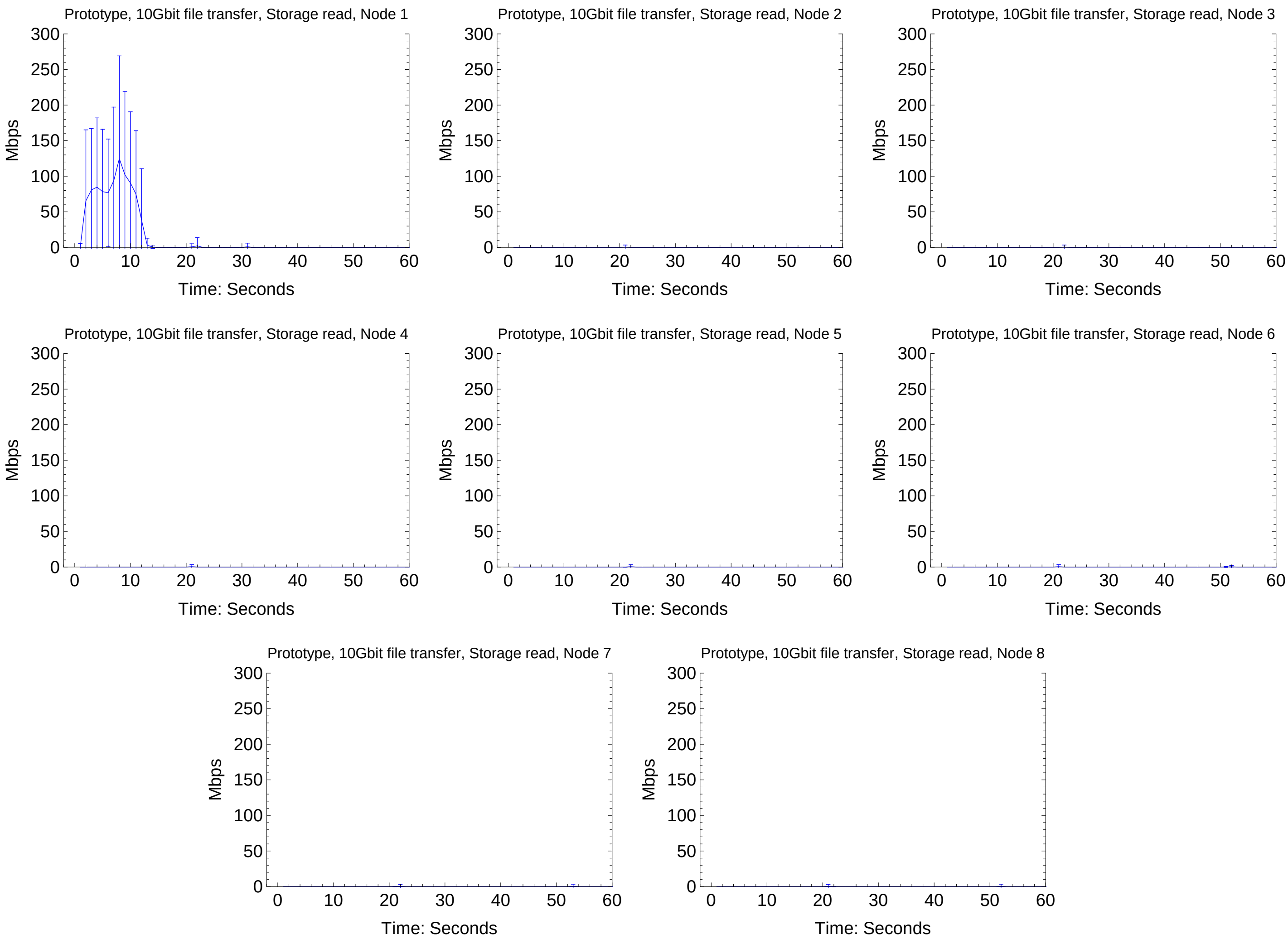


Figure 4.46: Prototype storage read performance results for the individual nodes. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

Storage write performance

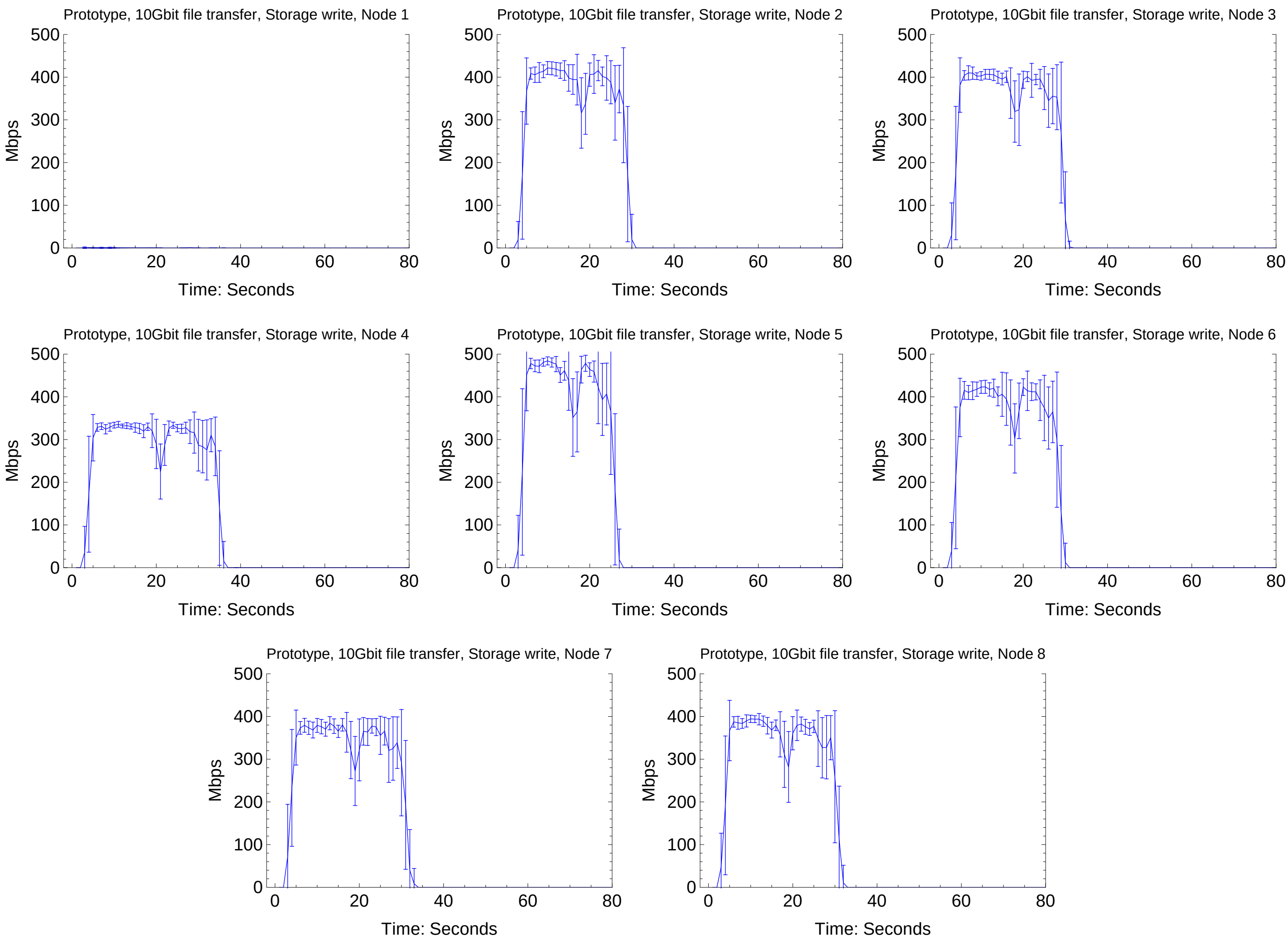
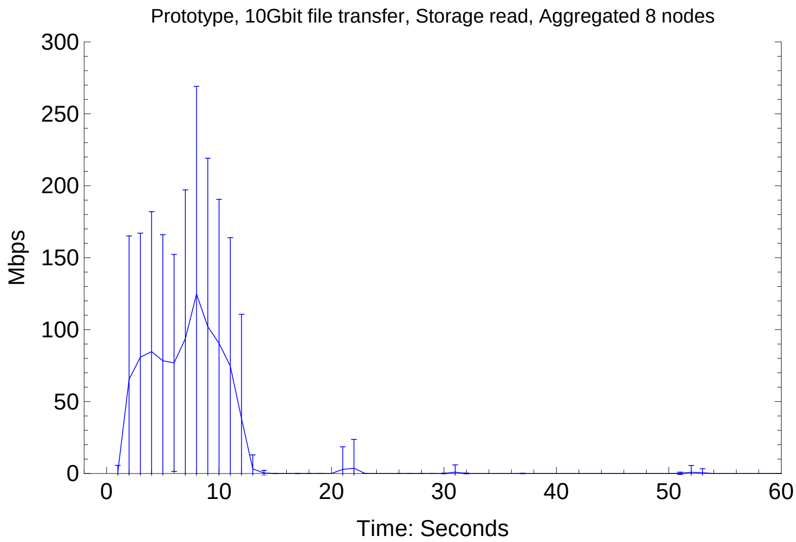


Figure 4.47: Prototype storage write performance results for the individual nodes. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

Storage performance aggregated



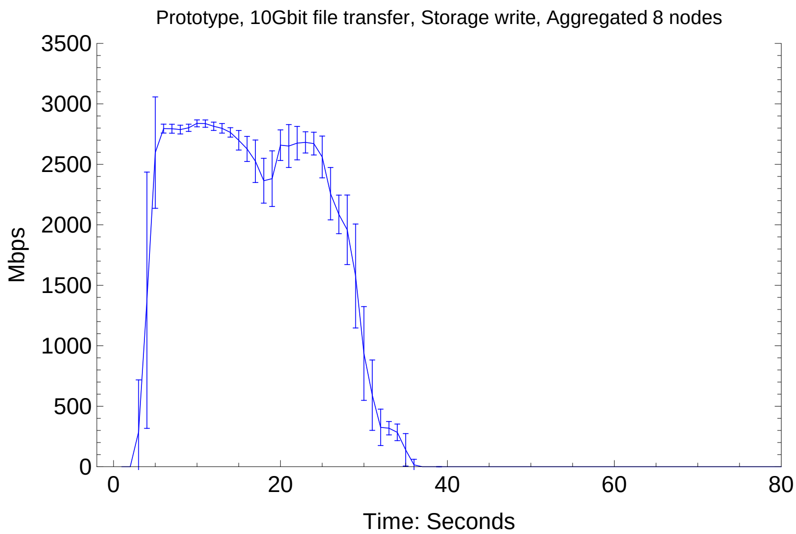


Figure 4.48: The prototype storage performance results for the 8 nodes combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second.

4.3.5 rTorrent CPU usage

rTorrent individual node CPU usage

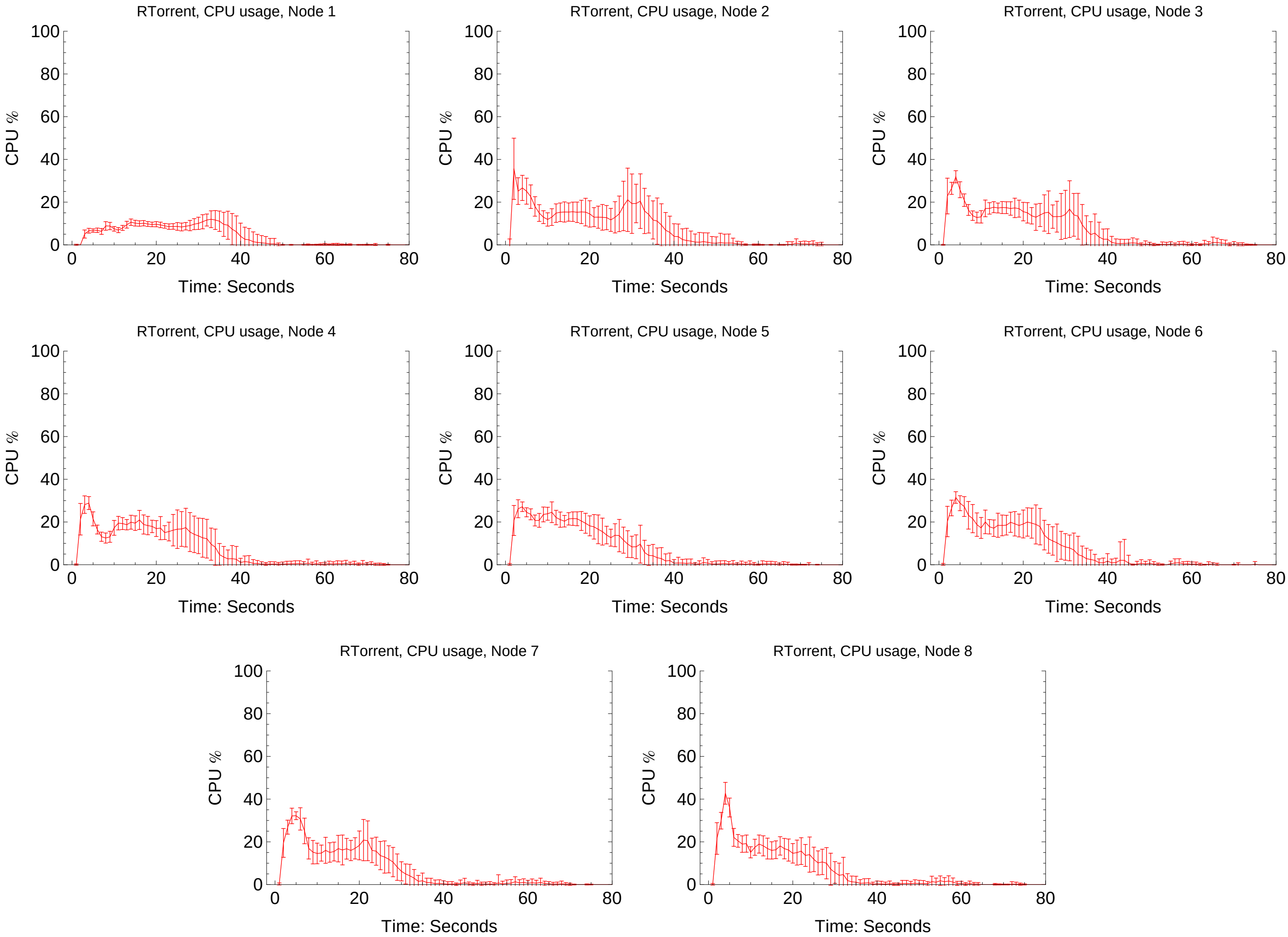


Figure 4.49: rTorrent CPU usage for the individual nodes. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

rTorrent aggregated CPU usage

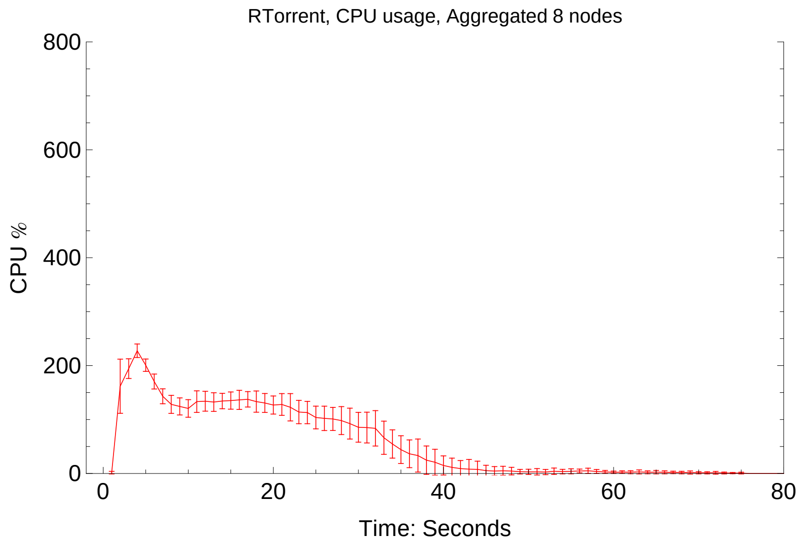


Figure 4.50: rTorrent CPU usage for the 8 nodes combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second.

4.3.6 Prototype CPU usage

Prototype individual node CPU usage

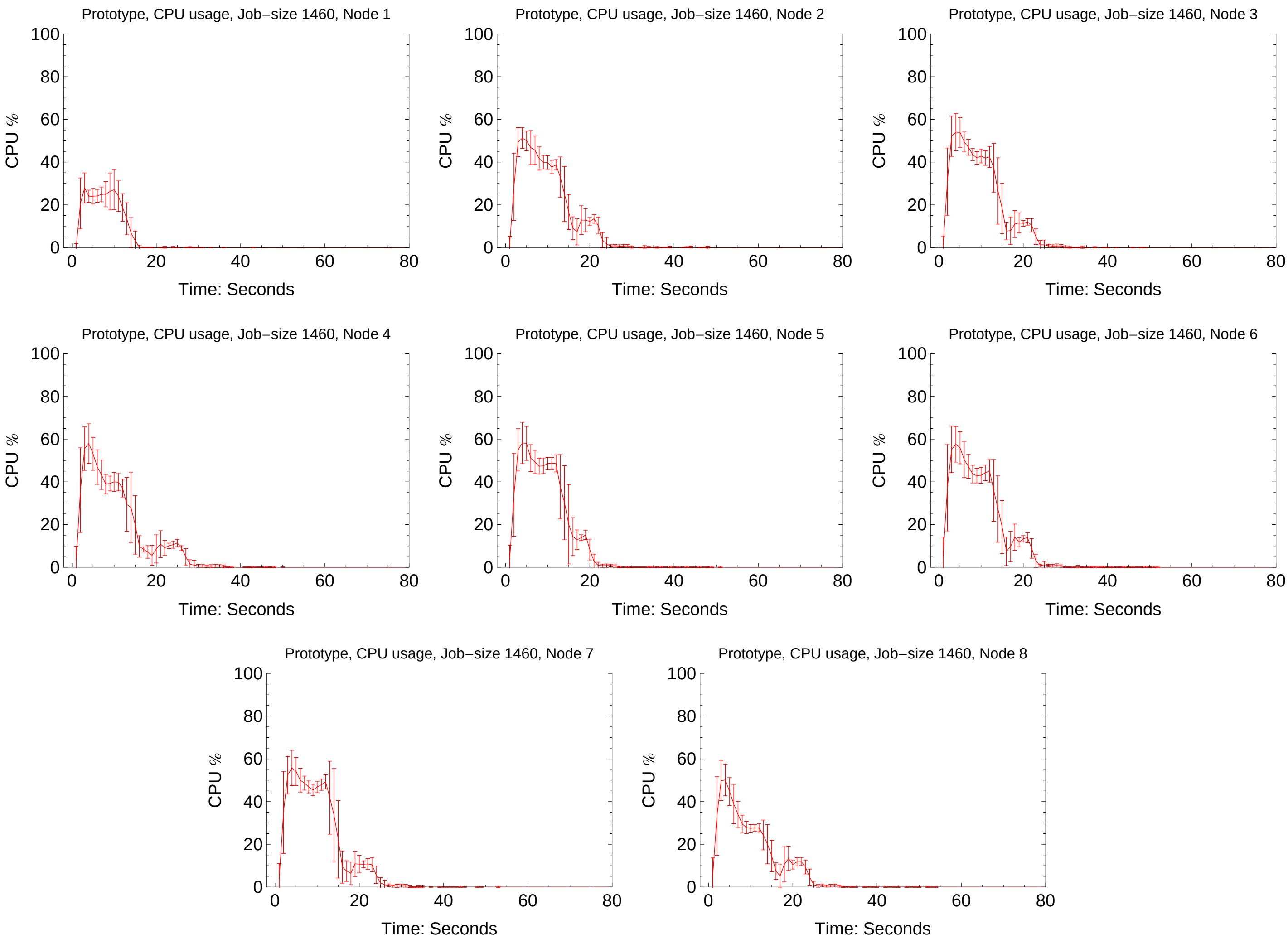


Figure 4.51: Prototype CPU usage for the individual nodes. The test plots are aggregated from 30 samples, and the standard deviation is plotted for every second.

Prototype aggregated CPU usage

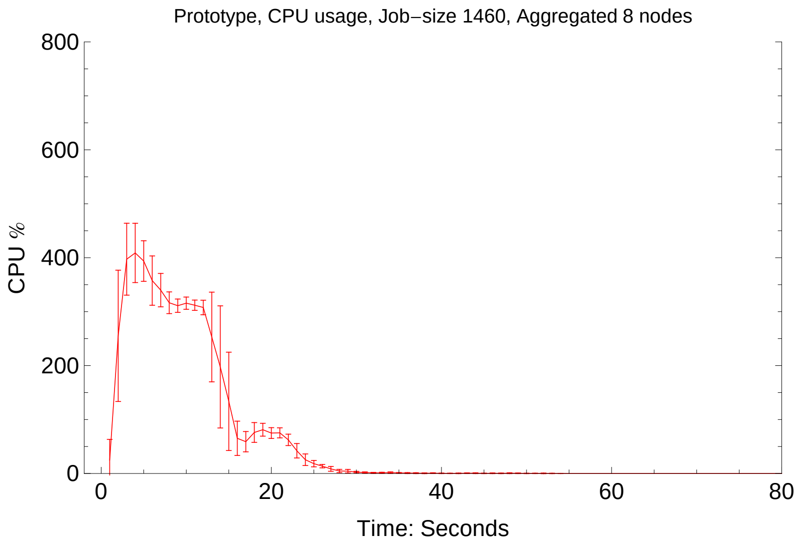


Figure 4.52: The prototype CPU usage for the 8 nodes combined into a single plot. The plot data is aggregated from 30 samples per node, and the standard deviation is plotted for each second.

4.4 Prototype scalability measurements

4.4.1 Throughput

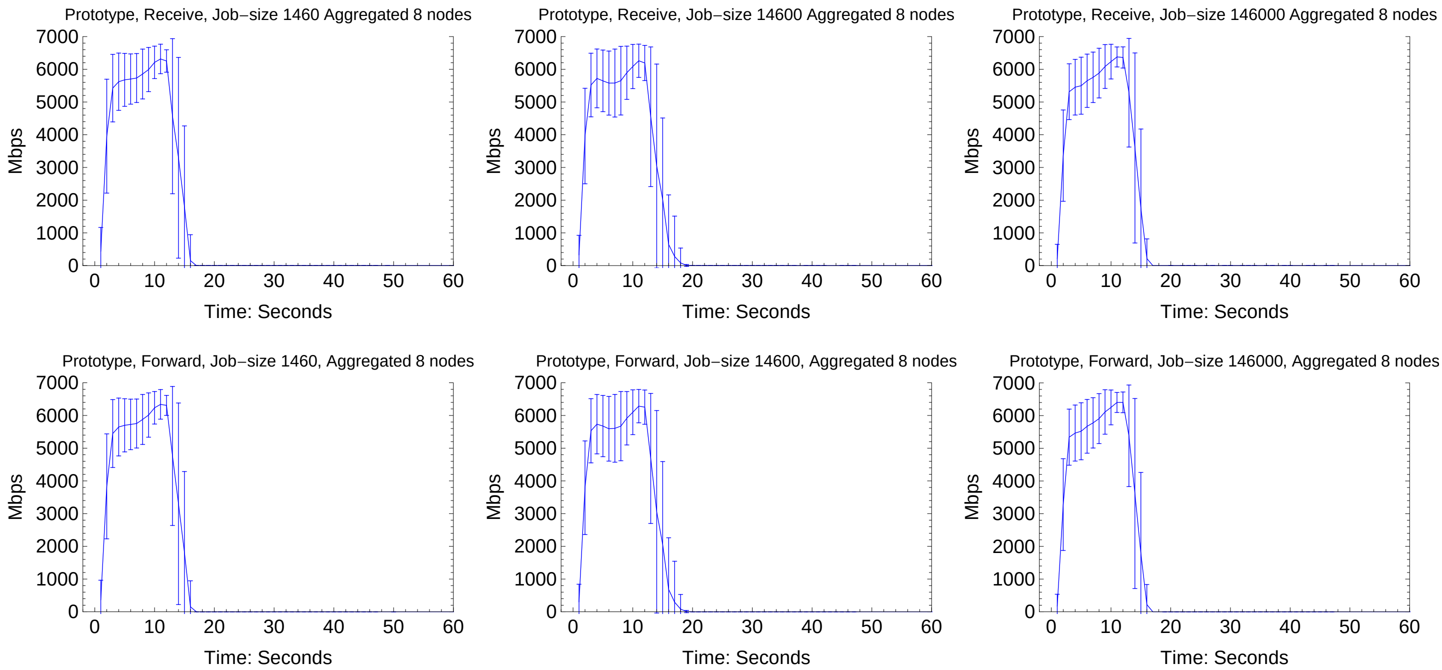


Figure 4.53: The prototype throughput performance at different job-size. The plots are aggregated results from full size experiments. The job-size is a count of the number of bytes for each job. The test plots are aggregated from 30 samples each, and the standard deviation is plotted for every second.

4.4.2 Storage performance

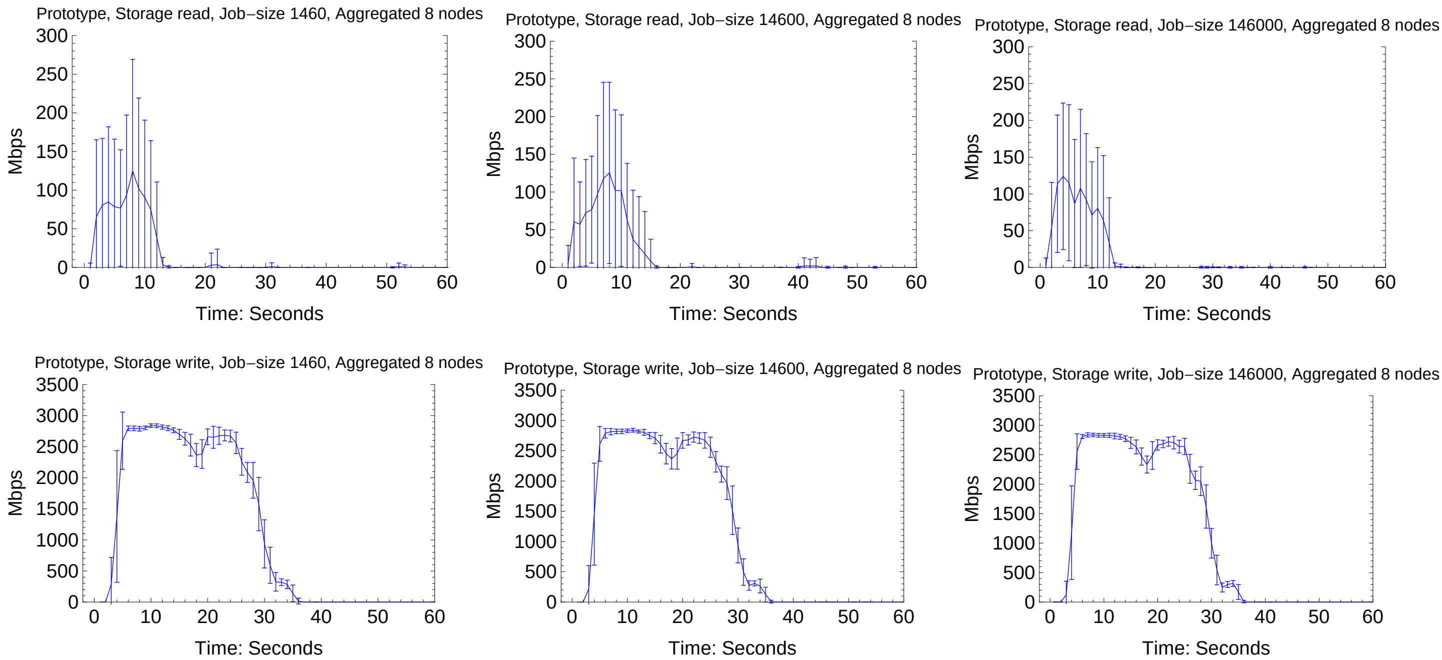


Figure 4.54: The prototype storage performance at different job-size. The plots are aggregated results from full size experiments. The job-size is a count of the number of bytes for each job. The test plots are aggregated from 30 samples each, and the standard deviation is plotted for every second.

4.4.3 CPU usage

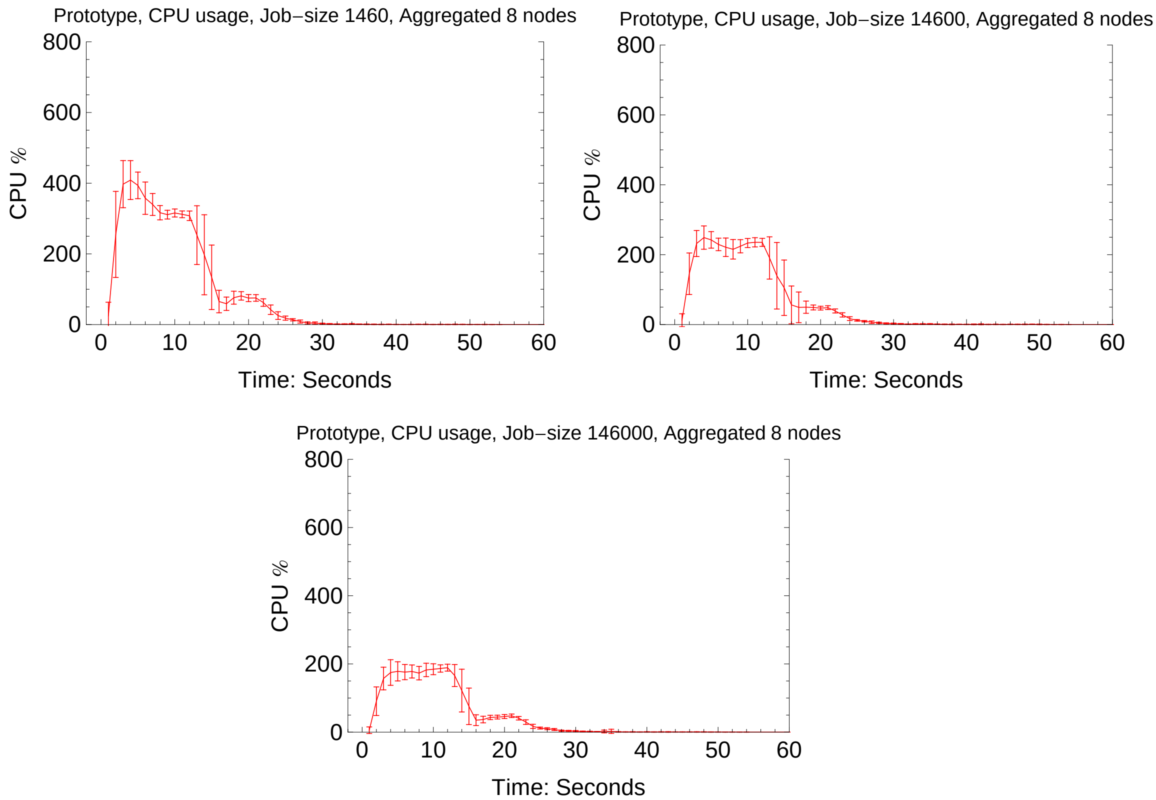


Figure 4.55: The prototype CPU usage at different job-size. The plots are aggregated results from full size experiments. The job-size is a count of the number of bytes for each job. The test plots are aggregated from 30 samples each, and the standard deviation is plotted for every second.

4.4.4 Prototype delay measurements

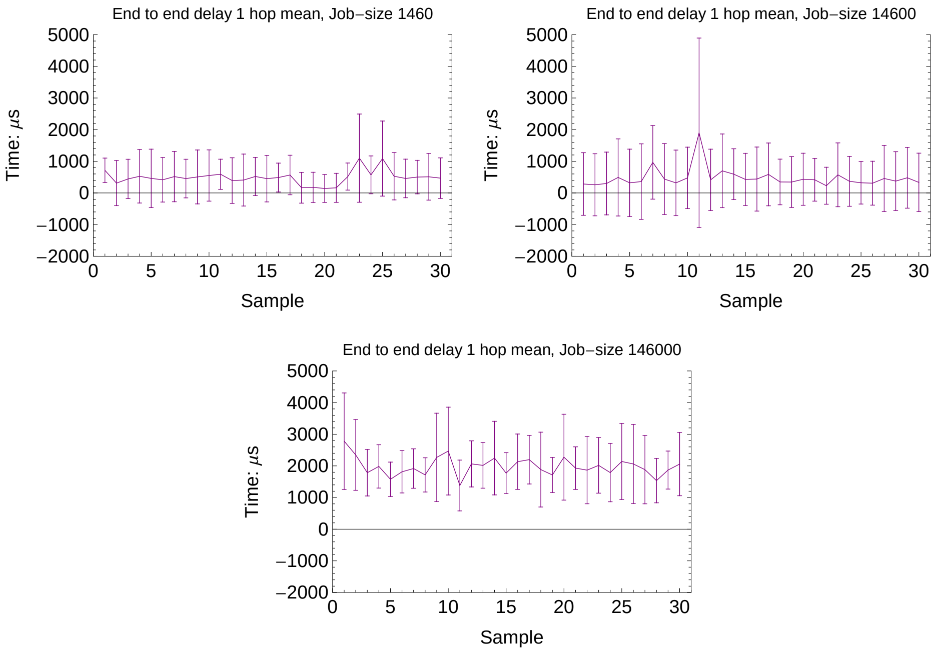


Figure 4.56: The prototype delay measurements at different job-sizes. The plots are aggregated results from full size experiments. The job-size is a count of the number of bytes for each job. The test plots are aggregated from 30 samples each, and the standard deviation is plotted for every second.

Job-size	Sample Count	Mean	Median	Max	S	95 % CI
1460	600	489.848	267.000	3531.000	674.260	435.788, 543.909
14600	600	474.735	288.500	7199.000	1025.330	392.527, 556.943
146000	600	1982.285	1721.000	7199.000	998.194	1902.250, 2062.320

Table 4.13: A table with delay measurement statistics. Values are rounded to a presentable level.

Chapter 5

Analysis

5.1 Finding the roof performance

The data from the experiments aimed at pinpointing the roof throughput performance of the proposed distribution strategy is analyzed here.

5.1.1 TCP throughput performance

The combined results for the baseline TCP test between two nodes without any mediating network devices can be seen in figure 4.2 on page 47. It can be seen from the figure that the connection from the first node to the second achieve maximum throughput in less than one second. When the second node establishes a connection the TCP connection immediately congests. TCP equilibrium is reached approximately 20 seconds after the returning TCP stream is initialized. The time delay until TCP equilibrium is significant and is outside expected performance.

The baseline test does, however, not represent the performance characteristics observed when the nodes are connected to a switch. A good representation of this can be seen in figure 4.14 on page 58. There seem to be a clear and distinct increases in speed between each consecutive TCP connection until the last node completes the transfer ring. When the last node connects to the initial starting node, TCP congestion avoidance seem to kick in. This performance pattern can be recognized among all tested network switches. Some deviating results can be seen in node 2 in figure 4.17 on page 61, node 3 in figure 4.25 on page 69 and node 3 figure 4.33 on page 77. On closer inspection it can be observed that two of the mentioned deviations from two different switches within the 3Com brand show similar deviation pattern. Another observation is that they are consistent in their deviation pattern. This behavior cannot be explained by TCP congestion, the throughput seem too flat and too consistent.

Some further work was done trying to find the root cause of these deviations. A physical examination revealed that all affected switches has one of the ports visually separated from the rest, which indicates that the port is indented to be used as an up-link port. In figure 5.1 an example of this can be seen. This suggests that the port has a functionality that the others do not.

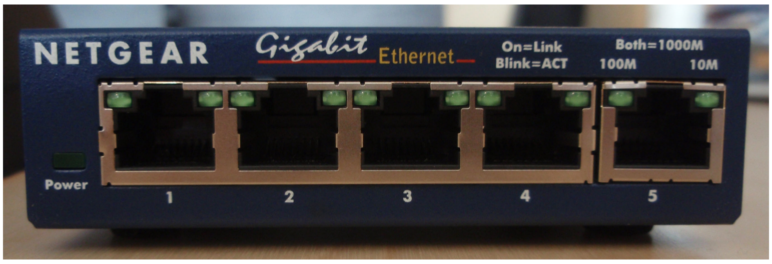


Figure 5.1: The Netgear ProSafe GS105 visually separates the rightmost port from the other ports. *Netgear ProSafe GS105* [Photograph] [Taken 5 April 2012]

It was found that the switches that showed this deviation has QoS (Quality of Service) listed as a feature. QoS is a resource reservation feature, which often is related to real-time streaming of multimedia UDP traffic. QoS would also explain why this deviation is only seen in the TCP benchmarks. Based on this observation, a likely explanation is that the deviation pattern is a TCP rate-limiting feature targeted at reserving resources for high priority UDP data traffic.

The performance statistics for the TCP benchmark reveals that there is a significant difference in mean TCP throughput and the ideal performance value. Overhead suggest an ideal transfer rate of 941.482 Mbps. This difference is expected because of the slow buildup process TCP goes through when trying to reach equilibrium state. Since the mean value is significantly affected by outliers in the data, the mode better represents the equilibrium state performance. The baseline test has a mode of 941.425 Mbps, which is 0.057 Mbps below the ideal performance target. TCP acknowledgments has not been accounted for, and the slight deviation that is measured is smaller than expected. None of the network switches achieve a mode value close to this target. This gap in performance here is expected to be mostly caused by the congestion appearing when the last node completes the transfer ring. In the performance graphs there seems to be a trend that there are some speed disturbance happening for each consecutive initiated TCP transfer, and the largest impact is when the last node completes the transfer ring.

The results return by the Z-tests, see table 4.2 on page 81, all gave a P such that $P < \alpha$. This is within the preset significance level, and the similarities between the baseline is rejected. The dissimilarity can be clearly seen in figure 4.2a on page 47. The difference observed was considered to large to give a good comparative basis between the switches, resulting in the need for post hoc tests. A more generalized ANOVA F-test test was setup with the following hypotheses

H_0*: There is no difference between switch TCP throughput performances.*

H_1*: There is a difference.*

The H_0 will be assumed to be true. The significance level, α, of the test will be set to 0.05 (5%). If the P value returned by the F statistic returns a P value such that $P < \alpha$ the H_0 hypothesis will be rejected. It is only the 5 port switches that will be tested in this follow up test. This is done because the congested port over non congested port ratio is significantly different for the 8 and 5 port switches. The results for the 5 port tests are expected to translate to the 8 port switches.

The results of the follow up test can be seen in table 4.3 on page 82. The ANOVA F-test returned a P value such that $P < \alpha$. This is within the preset significance level, therefore H_0 is rejected. The rejection of H_0 makes it fair to assume that the observed differences in the measurements are caused by a qualitative difference between the switches. The qualitative difference is most prominent between the high performing non rate-limited switches relative to the rate-limited ones.

5.1.2 UDP throughput performance

The performance statistics for UDP performance reveal that there is little deviation from the ideal throughput performance for all benchmarked switches. The ideal throughput performance when accounting for the overhead in the benchmarks is 95.7087%. All registered mode values are less than 0.02% percent points away from the ideal target. The largest difference can be seen in the standard deviation, where Dlink DGS-1005D and Cisco SD2005 show increased deviation over the baseline. The rest of the switches show a decrease in throughput deviation over the baseline, with Netgear GS605 and Netgear ProSafe GS105 standing out in this regard. When looking at the performance graphs, and the datagram loss statistics, it is clear that this throughput deviation is directly related to datagram loss. An inspection of the deviation for Dlink DGS-1005D when excluding error prone segments showed that it is the errors that cause the throughput deviation, and not the throughput deviation that causes the errors. The Dlink DGS-1005D switch has the highest recorded max throughput. The Cisco SD2005 switch show similar tendency but does not show an abnormally high max value. An inspection of the throughput values close to errors showed that there is no boost in throughput before or after recorded errors. It is worth noting that the plotted standard deviation in the graphs might give an impression that the throughput deviates with positive and negative values. This is an incorrect depiction, as it is consistently negative deviation that is seen in the recorded data.

The results return by the Z-tests, see table 4.5 on page 83, all gave a P such that $P > \alpha$. This is not within the preset significance level, and the similarities between the baseline cannot be rejected. In the TCP throughput performance test it was revealed that the baseline did not properly represent the switch throughput performance. Some of this trend can also be seen here, as the baseline has a fairly large confidence interval despite the large sample count. The difference seen was considered to large to give a good comparative basis between the switches, resulting in the need for post hoc tests. A more generalized ANOVA F-test test was setup with the following hypotheses

H_0: *There is no difference between measured switch UDP throughput performances.*

H_1: *There is a difference.*

The results of the follow up test can be seen in table 4.6 on page 84. The ANOVA F-test returned a P value such that $P < \alpha$. This is within the preset significance level, therefore H_0 is rejected. The rejection of H_0 makes it fair to assume that the observed differences in the measurements are caused by a qualitative difference between the switches.

5.1.3 Jitter

The jitter statistics seen in table 4.7 on page 85 does not show any surprising results. The measured values are well within the range which is required for accurate measurements. In the UDP throughput performance measurements there was observed some deviations that was caused by data loss. A manual inspection of the data revealed that this does not seem to be true for the jitter measurements. The deviations here does not seem to be caused by the errors, as high and low jitter seem evenly spread across the data and there was found no pattern that could link high, medium or low jitter values to data errors.

The results returned by the Z-tests, see 4.8 on page 85 all gave a P such that $P > \alpha$. This is not within the preset significance level, and the similarities between the baseline cannot be rejected. Similar to the UDP throughput performance data the baseline has a fairly large 95% confidence interval despite the large sample count. The difference seen was considered to large to give a good comparative basis between the switches, resulting in the need for post hoc tests. A more generalized ANOVA F-test test was setup with the following hypotheses

H_0: *There is no difference between the observed jitter.*

H_1: *There is a difference.*

The results of the follow up test can be seen in table 4.9 on page 86. The ANOVA F-test returned a P value such that $P < \alpha$. This is within the preset significance level, therefore H_0 is rejected. The rejection of H_0 makes it fair to assume that the observed differences in the measurements are caused by a qualitative difference between the switches.

5.1.4 Errors

In table 4.10 on page 87 it can bee seen that DGS-1005D and Cisco SD2005 show an increase in errors over the baseline. The rest show an improvement, where Netgear GS605 and Netgear ProSafe GS105 stand out in this regard.

The results returned by the Z-tests, see table 4.11 on page 87, all gave a P such that $P > \alpha$. This is not within the preset significance level, and the similarities between the baseline cannot be rejected. Similar to the UDP throughput performance and Jitter data the baseline has a fairly large 95% confidence interval despite the large sample count. The difference seen was considered to large to give a good comparative basis between the switches, resulting in the need for post hoc tests. A more generalized ANOVA F-test test was setup with the following hypotheses

H_0: *There is no difference between the observed loss measurements.*

H_1: *There is a difference.*

The results of the follow up test can be seen in table 4.12 on page 88. The ANOVA F-test returned a P value such that $P < \alpha$. This is within the preset significance level, therefore H_0 is rejected. The rejection of H_0 makes it fair to assume that the observed differences in the measurements are caused by a qualitative difference between the switches.

5.1.5 RTT and BDP

The test environment did not measure any RTT value large enough for the BDP to have any significance on TCP performance.

5.1.6 Throughput distribution and inter arrival-rate

The UDP throughput distributions all seem to be roughly normally distributed. There is little deviation seen in the UDP throughput distributions and the mean is close to the theoretical ideal performance. In the TCP throughput distributions unimodal, bimodal and even multi-modal tendencies can be seen. The bimodal and multi-modal tendencies seem to be caused primarily by switches where there is observed some TCP throughput throttling. Ignoring this throttling behavior the general TCP throughput performance can be described by a unimodal distribution with a negative skew. In figure 5.2 on the next page an example of a unimodal throughput distribution and a complementing histogram of the inter-arrival times can be seen. From the inter-arrival histogram it can be observed that it is simply just a mirror of the throughput distribution. Visually the inter-arrival distribution looks similar to a Poisson distribution. The inter-arrival distribution is, however, not formed by a Poisson arrival process. Since the TCP throughput goes through a ramp-up process a positive skew in the arrival distribution is expected. Further more the arrivals are not formed by a random process, and is therefore likely to be deterministic. Despite of this assumed deterministic arrival process, it is assumed that a Poisson arrivals can be used as a crude theoretical approximation.

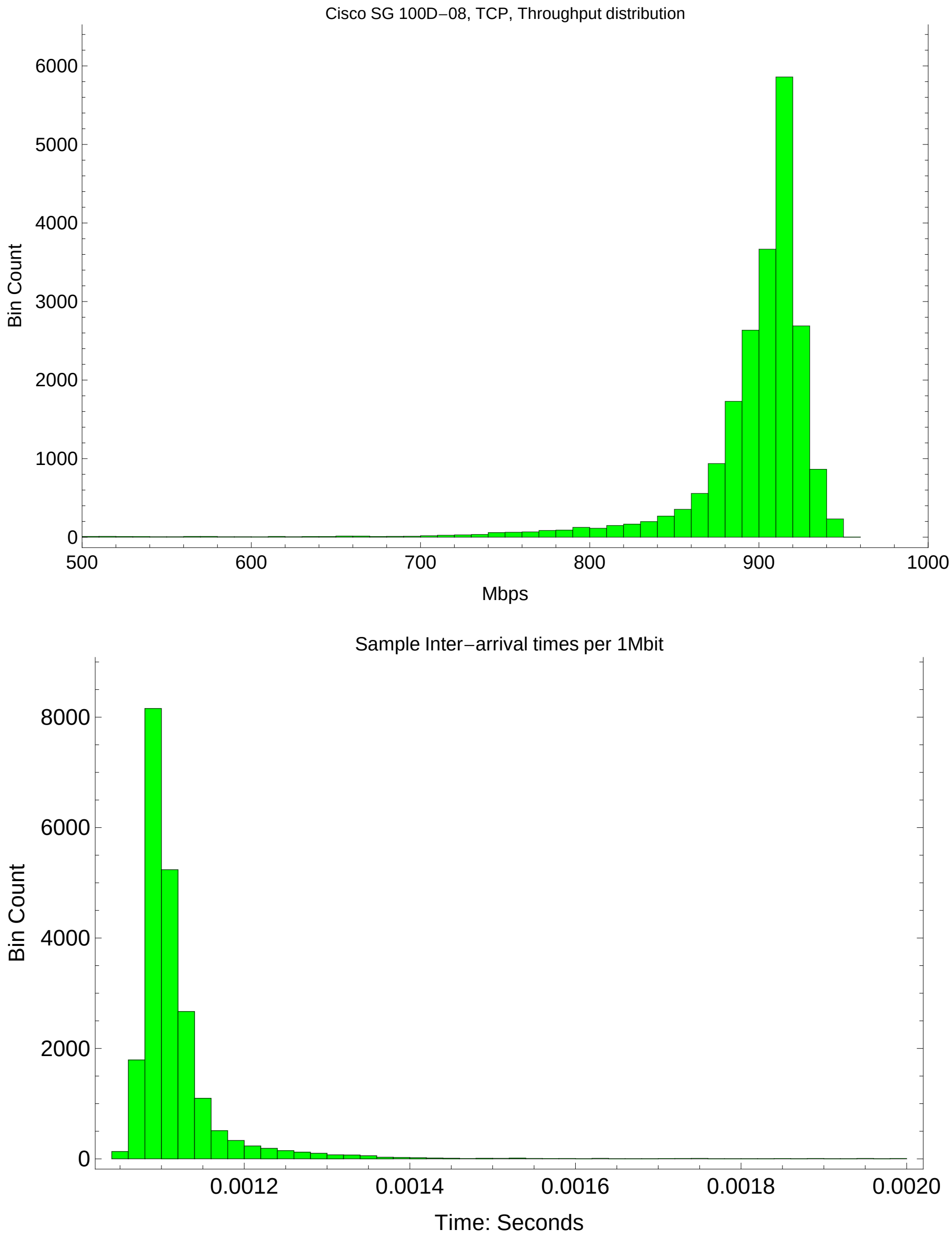


Figure 5.2: A histogram of the throughput distribution for the Cisco SG 100D-08 switch can be seen in the top graph. The graph is unimodal and the peak seem to be right before the ideal maximum TCP throughput. In the bottom graph it can be seen that the inter-arrival times are a mirror image of the throughput distribution.

5.2 Comparative benchmarks

5.2.1 Throughput performance

rTorrent

In the graphs in figure 4.37 on page 91 it can be seen that there is a clear early spike in receive and forward performance. The receive and forward are naturally linked, as one cause the other. This early performance is temporary, and performance overall drops after this initial performance spike. The drop in performance is followed by a slow increase in performance until some nodes have received the entire data, which results in a rapid decline in overall throughput. It assumed that this slow increase in throughput would continue up to at least the performance of the early spike if the file transfer process had lasted longer.

Towards the end of the file transfer there is a decrease in overall throughput. The reason for this behavior is that there will be less nodes to receive the data when nodes reach completion. Additionally the file is segmented and the segments are assigned to different nodes, resulting in fewer and fewer nodes seeding to each receiver as the file transfer reaches completion.

The nodes are not started exactly simultaneously, which result is that there will be a similar distribution of early peers for each benchmark sample. It is hypothesized that this is the reason for the observed early performance spike. The nodes are initiated sequentially, meaning that node 2 will be the first to receive data, node 3 will contact tracker having node 1 and node 2 as peers, node 4 receiving node 1, 2 and 3 as peer and so on. This pattern makes it reasonable to assume that it is highly likely to emerge an early tandem queue seeding pattern. If there was a tandem queue, the data that node 8 receives will already be spread across all other 7 nodes, resulting in low forward throughput for node 8. This is the pattern seen for the node 8 graph in figure 4.38 on page 92, which supports the proposed explanation. After a few seconds all nodes are aware of all other peers and the performance advantage of the early tandem queue dissolves, explaining the drop in performance.

Overall the performance is lower than what was expected, as the network is highly underutilized for most of the duration of the transfer.

Prototype

In the graphs in figure 4.40 on page 94 it can observed that the distinct throughput pattern found in the roof performance tests are not entirely translated to the prototype. The roof performance tests have distinct fast starts, and little to no congestion before the last node. The prototype have a slower initial start, and all nodes share the congestion evenly. The entire capacity of the switches was not used in these benchmarks, therefore it was not expected to be any observed congestion.

Compared to the rTorrent benchmarks the graphs show little difference in how long time it takes to reach the initial peak performance. The prototype manges to reach a high throughput quite early, and show a slow overall speedup until the end of the transfer. What separates the rTorrent and the prototype is that the prototype manages to maintain high throughput during the entire duration of the transfer. The prototype convergence-time is approximately half of what is observed when distributing with rTorrent.

It appears to be higher stress on the switch when the TCP transfers are initiated simultaneously. In the roof performance tests it was uncovered that in cases where the transfer originates outside the switch and does not end in the same switch the last node will experience congestion. The results indicate that there is a change in performance characteristics based on how much time there is between consecutive transfers, two possible scenarios is assumed to be likely. The additional congestion observed for the last node in the roof performance tests could be evenly distributed across the nodes, possibly negating the hypothesized cascading effect of the lower throughput for one node. The other option is that the congestion behavior seen in the last node could appear for all participating nodes, which would result in a significant decrease of overall performance. Additional testing would be required to confirm how this scenario would play out.

Based on the performance seen in previous benchmarks the throughput results for the prototype was a little slower than expected. Considering the small amount nodes, an average network throughput improvement from rTorrent to the prototype of 10 - 50% was expected. The observed increase in average throughput was more than 100%, resulting in less than half the convergence-time when using the prototype. The difference is expected to increase with increasing amount of nodes and network bottlenecks, as long as the seeding chain of the prototype is maintained properly.

5.2.2 Storage performance

rTorrent

In the graphs in figure 4.43 on page 97 it can be seen that there is almost no storage reads for any of the participating nodes. This suggest that the seeding node has cached most of the file in memory. The performance of the caching used in all the peers was unexpectedly efficient.

The write performance seen in the graphs in figure 4.44 on page 98 does show that the write of the file to storage lasts for a significant amount of time after the file has been received in memory. This period lasts approximately for 10-20 seconds after the file has been received. The write-rate is considered to be efficient and high performing during the first half of the transfer. Towards the last half of the file transfer the write-rate slows down significantly, resulting in a an low average write-rate.

Overall the absence of reads is considered significantly more effective than what was expected. The write performance was slower than expected.

Prototype

In the graphs in figure 4.46 on page 100 it can be seen that there are almost no storage reads for any of the participating nodes, except for the initial seeding node. The amount of storage reads observed for the initial seeding node is significantly lower than the size of the file transfer would suggest. This indicates that the operating-system has cached some of the file in memory.

The write performance seen in the graphs in figure 4.47 on page 101 show that the write of the file to storage lasts for a significant amount of time after the file has been received in memory. This period lasts approximately for 10-20 seconds after the file has been received. The write-rate does maintain a high performance during the entire transfer. The prototype does, however, not achieve maximum write-rate throughout the entire transfer, as there seems to be a dip in performance right after the entire file is received in memory. This performance drop is not maintained over time and show the characteristics of a short performance hiccup. In node 4 there seems to be a lower performing storage device than what is in the rest of the nodes, which does affect the appearance of the aggregated results. Although not as apparent, this slower performing storage device can also be recognized in the rTorrent benchmark results.

The read performance of the prototype is better than what was expected, which was primarily caused by the amount in memory caching done by the operating-system. It was unexpected that rTorrent outperformed the prototype on storage reads. It was expected that rTorrent would discard more of the file from memory and that it would result in more storage reads. The time it takes to write the file to disk after the file has been received in memory seems to be equal for both protocols. The storage write performance of the prototype is, however, considered to be performing better. When the prototype has received the file in memory it has only written approximately half of the file to storage, and maintain what seems to be close to maximum write-rate over the entire duration. This is not the case for rTorrent, and the prototype achieves an approximate double the average write-rate over rTorrent through the duration of the transfer. This is primarily caused by the decrease in write-rate seen in the torrent distribution towards the end of the file transfer.

5.2.3 CPU usage

rTorrent

In the graphs in figure 4.49 on page 103 it can be seen that the CPU usage is not a limiting factor in the benchmarks. The CPU usage seem to follow the throughput rates, where it seems that there is a higher CPU usage caused by receiving data than transmitting. This difference can be seen when comparing node 1 and node 8 graphs. The CPU usage results are within the expected range.

Prototype

In the graphs in figure 4.51 on page 105 it can be seen that the CPU usage is not a limiting factor in the benchmarks. The CPU usage seem to follow the throughput rates, where it seems that there is a higher CPU usage caused by receiving data than transmitting. This difference can be seen when comparing node 1 and node 8 graphs. The CPU usage results are within the expected range. Compared to the rTorrent CPU usage, there is little difference. The CPU usage of the prototype is closely linked with the specified job size, see section 5.3.3 on the next page for more details.

5.3 Scalability measurements

The scalability of the prototype is hypothesized to be closely linked with the job size specification. In this section the prototype performance characteristics at different job sizes are analyzed. Since the maximum segment size of the network that was benchmarked was 1460 bytes, the scalability benchmarks are a multiple of this size, 1x, 10x and 100x respectively.

5.3.1 Throughput performance

In the graphs in figure 4.53 on page 107 there seem to be little change in throughput characteristics depending on the benchmarked job sizes. This is, however, not going to hold true for all job sizes. Imagine transferring a file with a job size of the entire file, which would not be effective. It is expected that there is no use in straying outside the job size range of 1460 bytes to 1 MiB. Experience while developing the prototype indicated that larger job sizes was especially useful for throughput performance in virtualized environments. In virtualized environments the network performance is often only restricted by memory and CPU performance, therefore throughput will be limited by how job size affects those parameters, see section 5.3.3.

5.3.2 Storage performance

In the graphs in figure 4.54 on page 108 it can be seen that the storage performance characteristics remains mostly unchanged depending on the benchmarked job size.

5.3.3 CPU usage

In the graphs in figure 4.55 on page 109 it can be seen that the CPU usage is closely linked to the specified job size. The CPU usage is close to halved by increasing the job size with a multiple of 100x of 1460. Increasing the job buffer size decrease processing overhead, as the system needs to create less jobs, queue less items and do conditionals less often, which all adds up to requiring less system resources. This does, however, not scale well. The reason for this is that the CPU usage wasted on creating jobs, queuing and conditionals will be dwarfed by the CPU usage required to receive and transmit the data as the job size is increased beyond 100x of 1460.

5.3.4 Delay measurements

In the graphs in figure 4.56 on page 110 it seen that 1 hop end to end delay seem to be linked to the specified job size. In table 4.13 on page 110 the 1460 and 14600 job sizes seem to be have no statistically significant difference. It is, however, expected that with a higher sample count there will be lower average delay when using the 1460 byte job size. In these experiments the impact of the transmission delay has on the end to end delay seems to be too small to notice between 1460 and 14600 bytes. When looking at the job size of 146000 bytes there does seem to be an increase in transmission delay, and is thought to be the reason for the observed increase in the end to end delay. For a job size of 1460 bytes there is a 95% confidence interval of 435.788 - 543.909 μs. These results are considered an estimate because of the inherent low accuracy of the node clocks used for calculating these delays. It is this inaccuracy of the internal node clocks that cause the high standard deviation of up to 1000 μs (1 millisecond).

The end to end delay measurements show performance well above what was expected. Assuming an average delay of 500 μs or 0.0005 seconds, the data-stream could have reached 25 000 nodes within 12.5 seconds, and if the performance characteristics manage to maintain an 800 Mbps average throughput across a large number of nodes, the 10 Gbit data-stream could be within memory of 25 000 nodes after 25+ seconds.

Chapter 6

Discussion and conclusion

6.1 Finding the roof performance

6.1.1 TCP throughput performance

In the performance statistics there was found to be speed disturbances for each consecutive initiated TCP transfer. These disturbances are considered negligible for all consecutive TCP transfers except for the last connection, where the disturbance could have a significant performance degrading effect for the proposed seeding strategy. Additionally the last initiated TCP transfer suffered significant performance penalty caused by TCP congestion avoidance. Since the seeding strategy relies on a tandem queue, the performance hit could have a cascading effect for all the succeeding nodes. This is only an issue when the data stream does not originate or end in the switch. This issue is easily solved by leaving one of the switch ports unused. It is also hypothesized that using a transfer reliable UDP based protocol with less aggressive congestion avoidance than TCP could alleviate or solve this performance issue. This could be an interesting subject for further research. The overall the TCP results are considered a healthy foundation for the proposed seeding strategy.

6.1.2 UDP throughput performance

In the performance statistics there was found to be little difference in the throughput performance of the switches. There was also some differences in the amount of throughput deviation, and was correlated to be caused by datagram loss. Where some switches improved upon the baseline performance, and others did worse. It is unknown what caused the errors. It is hypothesized that the reason could be a qualitative difference in the interpretation and creation of the transmission signal. If it was a buffer dependent issue, it is assumed that it is not likely that there would be observed improvement over the baseline. The test results might suggest that using a transfer reliable UDP based protocol with less aggressive congestion avoidance than TCP could improve overall performance of the seeding strategy. Using UDP based transfer protocol could also avoid observed throughput rate-limiting.

6.1.3 TCP and UDP comparison

As a hardware benchmarking tool the TCP and UDP tests did show a significant difference. The TCP benchmarks did not manage to reveal any of the switch performance figures because of too unstable overall performance. It seems that UDP is a more precise and reliable tool for finding the true bit pushing power of the switch. Although UDP was shown to be a more reliable benchmarking tool, the TCP test results ended up being more valuable in discovering weaknesses and the expected performance of the distribution strategy.

6.1.4 Repeatability

The script that was created and used for benchmarking is included as an appendix. It is expected that the script will work on all versions of Iperf. This compatibility is, however, not tested and using Iperf version 2.0.4 is recommended for full reliability. The output results is reliant on accurate synchronization of time, making the setup of an local NTP server recommended for accurate measurements.

6.1.5 Likelihood of errors in the data

Each benchmark consists of 30 individual experiments where each experiment has a run time of at least 60 seconds. These data where then aggregated into the combined results for the benchmark. This is considered a fair amount of data. Most of the results that where seen was within expected behavior, and the deviations that where found in the benchmarks was connected with a possible explanation. Overall the likelihood of errors in the observed data is considered small.

6.1.6 Weaknesses in the experimental design

Some important weaknesses are listed here.

- The number of switches tested in this report is only a small subset of existing consumer switch hardware that exists. Testing a larger sample base would create a better basis for generalizing switch performance. There could also be undiscovered issues that is not represented in the sampled hardware in this thesis.

- The benchmarks in this thesis uses only a single type of network-card. The distribution strategy requires full duplex bandwidth performance, which is considered a significant load. It is not expected that there is equal quality in performance between different vendor network-cards. This makes it reasonable to assume that there could exist network-cards that might not have full performance under this specific type of load.

- Only a single operating-system is tested. Hence only a single TCP/IP stack is tested. TCP/IP stacks might not be created equal, and it is assumed that the TCP/IP stack tested is representative for all general TCP/IP stacks. This assumption might be wrong, and there could different behavior seen based on different implementations of the TCP/IP stack. Further testing would be required to confirm this.

- Iperf did not support collecting error data during TCP benchmarks. This created a gap in the collected data that could have given valuable information.

6.1.7 Alternative approaches

There are many ways the problem statement could have been solved, here are some of the alternative approaches that was considered.

- Using a single machine with several network cards would increase the time accuracy. Additionally using a single machine would ease the collection of performance data. This approach was rejected because of the possibility of saturating a system bottleneck.

- Using hardware test platforms dedicated for benchmarking network equipment could improve testing reliability. This approach was rejected because the lack of access and funds to such hardware.

- It is assumed that the low cost switch performance translates up to larger and more expensive switches. Benchmarks of assorted switches with 24+ ports would have been valuable.

- Wireless access-points has been hypothesized as being a likely weak point of the proposed distribution strategy. Benchmarks of low and high end wireless access-points will be needed to confirm if this is true, and to what extent.

6.1.8 Surprising results

In this section there will be made an attempt to explain the surprising results in the roof performance tests.

Baseline not representative for switch performance

In the baseline test the returning TCP connection showed immediate congestion. An expected result from this would be to see congestion when receiving and forwarding data when connected to a switch. This is not what is seen in the results, and the congestion is delayed until the last connection of the benchmark transfer ring. It is consistent by that it is always the last connection in the transfer ring that gets congested. This is, however, not considered a likely explanation as it would suggest that the congestion is intentional. It is hypothesized that the reason for the congestion in the baseline test could be caused by a poor interpretation and/or creation of the transmission signal. If the switch interpretation and creation of the transmission signal is better it would improve stability when it is being used to mediate the traffic. This further would suggest that that the congestion seen in the baseline and the switch benchmarks are unrelated. Testing would be required to confirm the proposed explanation for this phenomena.

Measured UDP errors not translating too poor TCP performance

The measured errors in the UDP tests does not directly translate to poor TCP throughput performance. There is logical link between high data loss ratio and poor TCP performance, as data loss could cause TCP congestion avoidance to trigger. The Dlink DGS-1005D and Cisco SD2005 showed high ratio of data loss, but was nonetheless among the best performers in the TCP tests. A possible cause for the difference could be that the TCP benchmark does not maintain such a high data transmission-rate over a long enough duration to provoke errors in the same degree as observed in the UDP benchmarks. Further testing would be required to confirm this hypothesis.

6.1.9 Viability of the results

When looking at the performance strictly from a throughput perspective, the results from the benchmarks show results which indicate that the distribution strategy is viable. No further work will be done to confirm surprising results or deviations, as the main objective of the roof performance benchmarks is considered to be fulfilled.

6.2 Comparative benchmarks

6.2.1 Repeatability

The full source code and make file for the prototype is included as appendix, making it possible to repeat the experiments. Note that the prototype does not have any robustness or security measures and is therefore unsuitable for use outside a lab environment.

6.2.2 Likelihood of errors in the data

Each benchmark consists of 30 individual experiments. These data where then aggregated into the combined results for the benchmark. This is considered a fair amount of data. There was observed slight deviation in write performance in a single node and the effectiveness of the rTorrent caching was unexpected. All results where within plausible range, and the mentioned deviations are considered insignificant. Overall the likelihood of errors in the data is considered to be small.

6.2.3 Weaknesses in the experimental design

Some important weaknesses in the experimental design are listed here.

- The experiments are configured such that the strength of the prototype distribution strategy is shown. Using a non bottlenecked scenario is likely to close the performance gap some.

- It is assumed that rTorrent represents a high performance BitTorrent client. No testing towards differentiating BitTorrent clients has been done, and there is a high chance that there exists a client that could perform better in the given benchmark scenario.

- It is assumed that Peertracker represents a high performance BitTorrent tracker. No testing towards differentiating BitTorrent trackers has been done, and it is likely there exists better performing trackers.

- rTorrent does have a fair amount of configuration options available. Not all configuration options and combinations of these was thoroughly tested, there certainly could exist a more optimal setup then what was used in the experiments.

6.2.4 Alternative approaches

There are many ways the problem statement could have been solved, here are some of the alternative approaches that was considered.

- In the introduction section of this thesis IP-layer multicast was introduced as an alternative distribution method. Comparative benchmarks against IP-layer multicast are needed for full assessments of the value of the prototype performance.

- BitTorrent was chosen as a comparative basis because it is the most commonly used application-layer multicast protocol. It is highly likely that there exists application-layer multicast protocols that would rival the prototype more on performance.

6.2.5 Viability of the results

The prototype performed slightly below what was expected based on the results from the roof performance tests. The performance was not far from utilizing the network to its full potential, and overall the results are considered good. As stated in the methodology, the prototype was expected to have better performance than BitTorrent in the configured experiment. Since the prototype does have smaller feature set than BitTorrent and fewer available usage scenarios, the performance had to be significantly better to be a viable alternative. The experiments revealed that the prototype outperformed the BitTorrent distribution on overall throughput and storage write performance metrics by a fair margin. The prototype does sacrifice ease of deployment for performance, but the performance difference is considered large enough for the prototype to be a viable alternative in the proposed usage scenarios.

6.3 Scalability measurements

6.3.1 Repeatability

The prototype source is included as appendix, and the job size can be adjusted by using the command-line argument job_size. The scalability measurements does, however, require some extra effort for reliable results. In addition to requiring a NTP server for time synchronization, the delay measurements will also require setup of clock disciplining. The clock drift on most regular computer clocks are relatively large, the measurements will not be possible if the clocks are not disciplined for at least some hours.

6.3.2 Likelihood of errors in the data

In the scalability measurements the accuracy of the internal computer clock became an issue. During clock disciplining it was observed errors up to ± 0.5 milliseconds. There was still high inaccuracy in the clocks relative to the delays that needed to be measured after disciplining. The clock inaccuracy was the prime reason for the high standard deviation seen in the end to end delay measurements. A high sample count does not combat this effectively because the node count becomes the limiting factor. The high inaccuracy of the clocks made it impossible to determine if there was a difference between the delays when using 1460 or 14600 byte job sizes. A difference was expected, therefore the measured delays are considered to be of limited accuracy.

6.3.3 Weaknesses in the experimental design

Some important weaknesses in the experimental design are listed here.

- The experiments does only have 8 participating nodes, and only 6 of those are forwarding nodes. The end to end delay measurements are then calculated between the 6 forwarding nodes, which gives only 5 similar data points for each benchmark. It is the first and last node clock drift that becomes significant when averaging out the delay. Thus increasing the node count would have given significantly better accuracy in the end to end delay measurements.

- There could be issues in with scalability that is not presenting itself in low node count experiments. Extrapolating from the results of 8 nodes is not considered confirming of high scalability. Large node count experiments would be required to confirm the small scale test results.

6.3.4 Viability of the results

The end to end delay of the prototype is considered the single most important measurement of how well the prototype scales to increasing number of receiving nodes. As the delay approaches 0, the scalability of the prototype approach infinity. Assuming that the measurements in this report show the true average delay, a single node can be traversed in approximate 0.0005 seconds. Assuming an average throughput of 800 Mbps, a 10 Gbit transfer will last for 10000/800 = 12.5 seconds. Using the measured delay the data will have traversed 12.5/0.0005 = 25000 nodes within 12.5 seconds. Further assuming the transfer characteristics can be maintained across this amount of nodes, the last node will receive the data within another 12.5 seconds plus the time of header exchange and writing to disk. A 10 Gbit file can then be transferred to 25000 equal stable performing nodes like those used in the benchmarks in 25+ seconds. Some large assumptions are made in these calculations, but the potential of the prototype is clearly shown. Additional testing is certainly needed for confirmation of large scale scalability, but the small scale show promising results.

6.4 Future work

For the prototype to become a usable product, further work with robustness, security and NAT traversal will be needed. Additionally the prototype needs to be implemented with a controller that can update node routes, collect availability status and initiate transfers. Converting the current prototype from a command-line tool into a library would be preferable before development of a graphical user interface. The current prototype sacrifice ease of deployment for performance. Implementation and research into link-layer discovery and optimal-path calculation could significantly improve ease of deployment. It is hypothesized that an implementation of a transfer reliable UDP based transport protocol over TCP could improve performance further, especially for networks exhibiting a large BDP. UDT [43] is considered a good candidate for a slot-in replacement for the current TCP implementation in the prototype.

6.5 Conclusion

In this thesis a data distribution strategy for nodes in a data subscriber relationship was proposed. The main idea was that data could efficiently be transferred to all nodes in a network by traversing the network with a snake like behavior, in other words traversing each network link only once.

In the problem statement the following goal was defined

"Explore the proposed application-layer multicast distribution strategy, and find out if the distribution strategy could improve the speed of data distribution between nodes in network and system administration relevant scenarios"

When working to solve the problem statement the following tasks where accomplished

- A roof performance test was devised and the results showed that the proposed distribution strategy was viable from a throughput perspective.
- A prototype which used the proposed distribution strategy was created. This prototype was benchmarked and was found to be a little less efficient than the roof performance tests would suggest.
- The prototype and the BitTorrent client rTorrent was benchmarked with 8 nodes in a tree topology scenario. The results showed that the prototype was more than twice as fast in average network throughput and storage write performance.
- Experiments to uncover scalability of overall throughput performance of the prototype where devised, in which the most important measure was the end to end delay. By extrapolating the end to end delay results to larger node counts it was found that there was potential for significant performance.

The results has shown that there could indeed be a performance gain by using the presented distribution strategy in the proposed usage scenarios.

Bibliography

[1] K.C. Almeroth. "The evolution of multicast: from the MBone to interdomain multicast to Internet2 deployment". In: *Network, IEEE* 14.1 (2000), pp. 10 –20. ISSN: 0890-8044. DOI: 10.1109/65.819167.

[2] *Amazon Elastic Compute Cloud (Amazon EC2)*. http://aws.amazon.com/ec2/.

[3] *Boost Library*. http://www.boost.org.

[4] Xinuo Chen and S.A. Jarvis. "Analysing BitTorrent's Seeding Strategies". In: *Computational Science and Engineering, 2009. CSE '09. International Conference on*. Vol. 2. Aug. 2009, pp. 140 –149. DOI: 10.1109/CSE.2009.140.

[5] Cisco. *Planning for IP Multicast in Enterprise Network*. http://www.cisco.com/en/US/tech/tk828/technologies_white_paper09186a0080092942.shtml.

[6] Bram Cohen. *BitTorrent*. http://www.bittorrent.com.

[7] *Collectl*. http://collectl.sourceforge.net.

[8] S. Deering. *Host Extensions for IP Multicasting*. http://www.ietf.org/rfc/rfc1112.txt. 1989.

[9] C. Diot et al. "Deployment issues for the IP multicast service and architecture". In: *Network, IEEE* 14.1 (2000), pp. 78 –88. ISSN: 0890-8044. DOI: 10.1109/65.819174.

[10] *Etherbat - Ethernet topology discovery (Documentation)*. http://www.cryptonix.org/projects/etherbat/#documentation.

[11] M. Hosseini et al. "A Survey of Application-Layer Multicast Protocols". In: *Communications Surveys Tutorials, IEEE* 9.3 (2007), pp. 58 –74. ISSN: 1553-877X. DOI: 10.1109/COMST.2007.4317616.

[12] IBM. *Deployment Guide Series: Tivoli Provisioning Manager for OS Deployment V5.1*. http://www.redbooks.ibm.com/redbooks/pdfs/sg247397.pdf. 2007.

[13] *Iperf*. http://sourceforge.net/projects/iperf/.

[14] James F. Kurose and Keith W. Ross. *Computer Networking: A Top Down Approach, 4th edition*. Pages 471 - 475, Ethernet Frame Structure. Addison-Wesley, 2007.

[15] James F. Kurose and Keith W. Ross. *Computer Networking: A Top Down Approach, 4th edition*. Pages 33 - 45, Delay , Loss and Throughput in Packet-Switched Networks. Addison-Wesley, 2007.

[16] L. Lao et al. "A comparative study of multicast protocols: top, bottom, or in the middle?" In: *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*. Vol. 4. 2005, 2809 –2814 vol. 4. DOI: 10.1109/INFCOM.2005.1498567.

[17] *Link Layer Topology Disvoery (LLTD)*. http://msdn.microsoft.com/en-us/library/cc233983(v=prot.10).aspx.

[18] *Linux Programmer's Manual: TCP protocol*. http://www.kernel.org/doc/man-pages/online/pages/man7/tcp.7.html.

[19] Microsoft. *About Multicast for Operating System Deployment*. http://technet.microsoft.com/en-us/library/cc431418.aspx. 2008.

[20] K. Obraczka. "Multicast transport protocols: a survey and taxonomy". In: *Communications Magazine, IEEE* 36.1 (1998), pp. 94 –102. ISSN: 0163-6804. DOI: 10.1109/35.649333.

[21] M. Ohmori, K. Okamura, and K. Araki. "Design of scalable interdomain IP multicast architecture". In: *Information Networking, 2001. Proceedings. 15th International Conference on*. 2001, pp. 819 –824. DOI: 10.1109/ICOIN.2001.905591.

[22] *Peertracker*. http://code.google.com/p/peertracker/.

[23] Jiayin Qi et al. "Analyzing BitTorrent Traffic Across Large Network". In: *Cyberworlds, 2008 International Conference on*. Sept. 2008, pp. 759 –764. DOI: 10.1109/CW.2008.150.

[24] *RFC 1323 - TCP Extensions for High Performance*. http://tools.ietf.org/html/rfc1323.

[25] *RFC 2147 - TCP and UDP over IPv6 Jumbograms*. http://tools.ietf.org/html/rfc2147.

[26] *RFC 2401 - Security Architecture for the Internet Protocol*. http://tools.ietf.org/html/rfc2401.

[27] *RFC 2460 - Internet Protocol, Version 6 (IPv6) Specification*. http://tools.ietf.org/html/rfc2460.

[28] *RFC 2544 - Benchmarking Methodology for Network Interconnect Devices*. http://tools.ietf.org/html/rfc2544.

[29] *RFC 3393 - IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)*. http://tools.ietf.org/html/rfc3393.

[30] *RFC 4821 - Packetization Layer Path MTU Discovery*. http://tools.ietf.org/html/rfc4821.

[31] *RFC 5681 - TCP Congestion Control*. http://tools.ietf.org/html/rfc5681.

[32] *RFC 6349 - Framework for TCP Throughput Testing*. http://tools.ietf.org/html/rfc6349.

[33] *RFC 768 - User Datagram Protocol*. http://tools.ietf.org/html/rfc768.

[34] *RFC 791 - INTERNET PROTOCOL*. http://tools.ietf.org/html/rfc791.

[35] *RFC 793 - TRANSMISSION CONTROL PROTOCOL*. http://tools.ietf.org/html/rfc793.

[36] *RFC 879 - The TCP Maximum Segment Size and Related Topics*. http://tools.ietf.org/html/rfc879.

[37] *RFC 895 - A Standard for the Transmission of IP Datagrams over Experimental Ethernet Networks*. http://tools.ietf.org/html/rfc895.

[38] *Rocks Cluster*. http://www.rocksclusters.org/wordpress/.

[39] *rTorrent*. http://libtorrent.rakshasa.no.

[40] *rTorrent performance tuning*. http://libtorrent.rakshasa.no/wiki/RTorrentPerformanceTuning.

[41] *rTorrent super seeding*. http://libtorrent.rakshasa.no/wiki/RTorrentInitialSeeding.

[42] Henning Schulzrinne, Radu State, and Saverio Niccolini. *Principles, systems and applications of IP telecommunications : Services and Security of next generation networks : second international conference IPTcomm 2008*. Page 42. 2008.

[43] *UDT: Breaking the Data Transfer Bottleneck*. http://udt.sourceforge.net.

[44] R.L. Xia and J.K. Muppala. “A Survey of BitTorrent Performance”. In: *Communications Surveys Tutorials, IEEE* 12.2 (2010), pp. 140 –158. ISSN: 1553-877X. DOI: 10.1109/SURV.2010.021110.00036.

Chapter 7

Appendices

7.1 Appendix: Iperf wrapper (Perl)

bench.pl

```
#!/usr/bin/perl

# --> Include packages
# --------------------
use Getopt::Long;
use strict;
use warnings;
use threads;
use Net::SSH::Expect;
use Term::ReadKey;

# --> Init variables
# ------------------
my $HELP;
my $OUT;
my $IN;
my @IP_LIST;
my $PASSWORD;
my $TIME;
my $U;
my $B;
my $WAIT;
my $SAMPLES;
my $SYNC;
my $WIN;

# --> Handle flags and arguments
# ------------------------------
GetOptions('h|help' => \$HELP,
           'u|udp' => \$U,
           'x|sync=s' => \$SYNC,
           'b|bandwidth=s' => \$B,
           'o|out=s' => \$OUT,
           'i|in=s' => \$IN,
           't|time=s' => \$TIME,
           'w|wait=s' => \$WAIT,
           's|samples=s' => \$SAMPLES,
           'r|window=s' => \$WIN);

# Print help message if -h is invoked
```

```
if ($HELP){
    usage();
    exit 0;
}

if(!$TIME)
{
    $TIME = 32;
}
else
{
    $TIME = $TIME + 2;
}

if(!$WAIT)
{
    $WAIT = 2;
}

if(!$B)
{
    $B = "957m";
}

if(!$SAMPLES)
{
    $SAMPLES = 1;
}

if($WIN)
{
    $WIN = "-w $WIN";
}
else
{
    $WIN = "";
}

if(!$IN)
{
    print "Option --in is mandatory\n";
    usage();
    exit 0;
}
else
{
    open(FILE,"<", "$IN") or die "\nCannot open file $IN $!\n";
    while ( <FILE> )
    {
        push @IP_LIST, trim($_);
    }
}

# Password prompt
my $key = 0;
my $password = "";

print "\nSSH Password: ";

my $index = 0;
ReadMode(4);
while(ord($key = ReadKey(0)) != 10)
```

```
{
    if(ord($key) == 127 || ord($key) == 8) {
        # DEL/Backspace was pressed
        # 1. Remove the last char from the password
        if ($index > 0)
        {
            chop($PASSWORD);
            # 2 move the cursor back by one
            print "\b \b";
            $index--;
        }
    } elsif(ord($key) < 32) {
        # Do nothing
    } else {
        $PASSWORD = $PASSWORD.$key;
        print "*";
        $index++;
    }
}
ReadMode(0); # Reset the terminal

# --> Main script content
# -----------------------

my @servers_ssh;
my @clients_ssh;

print "\n";
# Create ssh connectors for iperf clients
foreach (@IP_LIST)
{
    push @clients_ssh, Net::SSH::Expect->new (
        host=> $_,
        password=> "$PASSWORD",
        user=> 'root',
        raw_pty => 1
    );
    print "Client: $_\n";
}

print "\n";
# Create ssh connectors for iperf servers
foreach (@IP_LIST)
{
    push @servers_ssh, Net::SSH::Expect->new (
        host=> $_,
        password=> "$PASSWORD",
        user=> 'root',
        raw_pty => 1
    );
    print "Server: $_\n";
}

# Connect
print "\n";
foreach(@clients_ssh)
{
    my $max_retry_count = 10;
    my $retry_count = 0;
    while(1){
        my $rc = eval{$_->login();};
        last if defined $rc;
```

```
        last if $retry_count >= $max_retry_count;
        $retry_count++;
        sleep 1;
    }
    print "Client connect retries: $retry_count\n";
    if($retry_count >=$max_retry_count)
    {
        print "Connection refused, script exited\n";
        exit 0;
    }
}

# Connect
print "\n";
foreach(@servers_ssh)
{
    my $max_retry_count = 10;
    my $retry_count = 0;
    while(1){
        my $rc = eval{$_->login();};
        last if defined $rc;
        last if $retry_count >= $max_retry_count;
        $retry_count++;
        sleep 1;
    }
    print "Server connect retries: $retry_count\n";
    if($retry_count >=$max_retry_count)
    {
        print "Connection refused, script exited\n";
        exit 0;
    }
}

my $command = $U ? "iperf -s -u -i 1 -y C" : "iperf -s $WIN";
my $max = scalar(@IP_LIST);

print "\n";

if($SYNC)
{
    foreach(@clients_ssh)
    {
        print "Trying to sync $_->{'host'} to $SYNC\n";
        if ($_->{'host'} eq $SYNC){
        print "$SYNC is ntp server,\nskipping synchronization of $SYNC\n"; next;
        }
        $_->exec("stty raw -echo");
        $_->send("ntpdate $SYNC");
        $_->waitfor('adjust',10) or die "Could not sync time to $SYNC, exited\n";
        print $_->eat($_->peek(0));
        print "\n";
    }
}

for(my $sample = 1; $sample <= $SAMPLES; $sample++)
{
    print "### SAMPLE $sample OF $SAMPLES ###\n";
    for(my $b = 0; $b < $max; $b++)
    {
        my $g;

        my $client = $clients_ssh[$b];
```

```
        if($b == ($max-1)){$g = 0;} else {$g = $b+1;}
        my $server = $servers_ssh[$g];

        print "Host $b connects to $g \n";

        $server->exec("stty raw -echo");
        $client->exec("stty raw -echo");

        $server->send ($command);
        sleep(1);

        print "$command ($IP_LIST[$g])\n";

        if($U)
        {
            $client->send ("iperf -c $IP_LIST[$g] -u -b $B -t $TIME");
            print "iperf -c $IP_LIST[$g] -u -b $B -t $TIME ($IP_LIST[$b])\n";
        }
        else
        {
            $client->send ("iperf -c $IP_LIST[$g] -y C -i 1 -t $TIME");
            print "iperf -c $IP_LIST[$g] -y C -i 1 -t $TIME ($IP_LIST[$b])\n";
        }

        print "Waiting for: $WAIT\n";
        sleep($WAIT);
    }

    print "Waiting for execution to finish..\n";
    sleep($TIME);

    print "\n";
    for(my $b = 0; $b < $max; $b++)
    {
        my $g;

        my $client = $clients_ssh[$b];
        if($b == ($max-1)){$g = 0;} else {$g = $b+1;}
        my $server = $servers_ssh[$g];

        my $chunk;
        my @content;
        if($U)
        {
            $chunk = $server->peek();
            @content = split('\n',$server->eat($chunk));
        }
        else
        {
            $chunk = $client->peek();
            @content = split('\n',$client->eat($chunk));
        }

        splice(@content,($TIME-2));

        my $s = 1;
        foreach (@content)
        {
            print "Line $s: $_\n";
            $s++;
        }

```

```
        my $prot = $U ? "U" : "T";
        my $hos = $b+1;
        # Output to file
        if ($OUT){
            open(OUT,">$OUT$prot-h$hos-s$sample.csv");

            foreach (@content)
            {
                print OUT "$_\n";
                $s++;
            }

            close(OUT);
        }

        # Terminate iperf server
        $server->send("\cC"); # Ctrl-C
        print "\n";
    }
}

# Close ssh connection
foreach(@clients_ssh)
{
    $_->close();
}

# Close ssh connection
foreach(@servers_ssh)
{
    $_->close();
}

# --> Functions
# -------------

sub trim{
    my $string = shift;
    $string =~ s/^\s+|\s+$//g;
    return $string;
}

# Prints the correct use of this script
sub usage{
print <<"USAGE";

    Usage: bench.pl [OPTIONS] --in iplist

    DESCRIPTION

        A wrapper to iperf that can benchmark multiple nodes simultaneously.
        The benchmark results are output in a csv file format.

    GENERIC OPTIONS

        -h, --help\tDisplay Usage information
        -i, --in\tFile with list of IP addresses to benchmark
        -o, --out\tFile to output sampled data
        -t, --time\tTime in seconds to sample
        -w, --wait\tHow long to wait before adding another node to the benchmark
        -s, --samples\tHow many benchmarks to run
        -x, --sync\tSpecify NTP server sync address
```

```
        -r, --window\tSpecify TCP recieve window size

    UDP OPTIONS

        -u, --udp\tInvoke UDP test
        -b, --bandwidth\t[m|g] Specify UDP bandwidth target

    EXPLANATION OF CSV OUTPUT FIELDS

    TCP:
        Field 1:  Timestamp
        Field 2:  From host
        Field 3:  From port
        Field 4:  Target host
        Field 5:  Target port
        Field 6:  ID
        Field 7:  Time interval
        Field 8:  Bytes transferred
        Field 9:  Bits per second over interval

    UDP:
        Field 1:  Timestamp
        Field 2:  From host
        Field 3:  From port
        Field 4:  Target host
        Field 5:  Target port
        Field 6:  ID
        Field 7:  Time interval
        Field 8:  Bytes transferred
        Field 9:  Bits per second over interval
        Field 10: Jitter in milliseconds
        Field 11: Lost datagrams over interval
        Field 12: Total datagrams over interval
        Field 13: Lost datagrams in % over interval
        Field 14: Datagrams delivered out of order

USAGE
}
```

7.2 Appendix: Prototype makefile (BASH)

make.sh

```
#!/bin/bash

g++-4.6 -O2 -march=native -std=c++0x -o race racetrack.cpp -pthread -lboost_thread
-lboost_system -lboost_program_options -static -static-libgcc
```

7.3 Appendix: Prototype source (C++)

racetrack.cpp

```
#include <stdio.h>
#include <fstream>
#include <iostream>
#include <boost/thread.hpp>
#include <boost/asio.hpp>
#include <boost/program_options.hpp>
#include <chrono>
#include <deque>
#include <condition_variable>
#include <sstream>

namespace po = boost::program_options;
using boost::asio::ip::tcp;
using namespace std;

struct Job
{
   char* buf;
   int size;
   bool kill;
};

struct Header
{
   static const int head_size = 14;
   char head[14];
   int chunk_size;
   char filename[1024];
   short filename_length;
   int64_t filesize;
};

template<class T>
class JobQueue
{
   deque<T> _queue;
   condition_variable _cond;
   mutex _mutex;

 public:

   void put(T && job)
   {
        {
           lock_guard<mutex> lck(_mutex);
           _queue.push_front(move(job));
        }
      _cond.notify_one();
```

```
   }

   T receive()
   {
      unique_lock<mutex> lck(_mutex);
      _cond.wait(lck,[this]{return !_queue.empty();});
      T job = move(_queue.back());
      _queue.pop_back();
      return job;
   }
};

class JobCounter
{
   int _max_in_queue;
   int _in_queue;
   int _total_job_count;
   int _jobs_created;
   condition_variable _cond;
   mutex _mutex;

 public:

   void setValues(int max_in_queue, int total_job_count)
   {
      lock_guard<mutex> lck(_mutex);
      _max_in_queue = max_in_queue;
      _total_job_count = total_job_count;
   }

   bool inc()
   {
      unique_lock<mutex> lck(_mutex);
      _in_queue++;
      _jobs_created++;
      _cond.wait(lck,[this]{return (_in_queue < _max_in_queue);});
      return (_jobs_created < _total_job_count);
   }

   void dec()
   {
        {
           lock_guard<mutex> lck(_mutex);
           _in_queue--;
        }
      _cond.notify_one();
   }
};

void receive(
        boost::promise<Header> & header_received,
        boost::promise<
                std::chrono::time_point<
                        std::chrono::high_resolution_clock>> & send_start_receive,
        int port, JobQueue<Job> & sendqueue,
        JobCounter & jobcounter,
        int buffer_size
        )
{
   int n;
   Header h;

```

```
    boost::asio::io_service io_service;
    tcp::acceptor acceptor(io_service, tcp::endpoint(tcp::v4(),port));

    tcp::socket socket(io_service);
    acceptor.accept(socket);

    send_start_receive.set_value(std::chrono::high_resolution_clock::now());

    // Read header
    int bytes_received = 0;
    while(bytes_received < h.head_size)
    {
       n = boost::asio::read(socket,boost::asio::buffer(
                                        h.head + bytes_received,
                                        h.head_size - bytes_received));
       bytes_received += n;
    }

    memcpy(&h.chunk_size,h.head,4);
    memcpy(&h.filename_length,h.head+4,2);
    memcpy(&h.filesize,h.head+6,8);

    // Read filename
    bytes_received = 0;
    while(bytes_received < h.filename_length)
    {
       n = boost::asio::read(socket,boost::asio::buffer(
                                        h.filename + bytes_received,
                                        h.filename_length - bytes_received));
       bytes_received += n;
    }

    header_received.set_value(h); // Send prms

    int chunks_to_receive = h.filesize / h.chunk_size;
    int left = h.filesize % h.chunk_size;
    buffer_size = buffer_size * 1048576; // MiB

    jobcounter.setValues((buffer_size/h.chunk_size), chunks_to_receive);

    stringstream out (stringstream::in | stringstream::out);
    out << "recv_job: ";
    int i = 0;

    auto start = std::chrono::high_resolution_clock::now();
    cout << "Receiving data.." << endl;

    do
    {
       Job j;
       char* buffer = new char[h.chunk_size];

       int bytes_received = 0;
       while(bytes_received < h.chunk_size)
       {
          n = boost::asio::read(
                         socket,
                         boost::asio::buffer(
                                        buffer + bytes_received,
                                        h.chunk_size - bytes_received));
          bytes_received += n;
       }
```

```

      // Record arrivaltimes of first 4 jobs
      if(i < 4)
      {
        auto pkttime = std::chrono::high_resolution_clock::now();
        out << fixed << std::chrono::duration<double>(
                        pkttime.time_since_epoch()).count() << " ";
        i++;
      }

      j.buf = move(buffer);
      j.size = h.chunk_size;
      sendqueue.put(move(j));

   }while(jobcounter.inc());

   // Receive leftover bytes that did not fill up a chunk
   if(left > 0)
   {
      Job j;
      char* buffer = new char [left];

      int bytes_received = 0;
      while(left > bytes_received)
      {
                n = boost::asio::read(
                              socket,
                              boost::asio::buffer(
                                              buffer + bytes_received,
                                              left - bytes_received));
                bytes_received += n;
      }

      j.buf = move(buffer);
      j.size = left;
      sendqueue.put(move(j));
   }

   Job j;
   j.kill = true;
   sendqueue.put(move(j)); // Send kill job

   auto end = std::chrono::high_resolution_clock::now();
   double sec = std::chrono::duration<double>(end - start).count();

   // OUTPUT
   double transfer_speed = (((double)(h.filesize*8)/1000000)/sec);
   out << endl;
   out << "recv_start: ";
   out << fixed << std::chrono::duration<double>(
                start.time_since_epoch()).count() << endl;
   out << "recv_end: ";
   out << fixed << std::chrono::duration<double>(
                end.time_since_epoch()).count() << endl;
   out << "Data received: avg receive speed ";
   out << transfer_speed << " Mbit/s" << " for " << sec << "s" << "\n";

   cout << out.str();
   socket.close();
}

void write_to_disk_server(
```

```
                boost::shared_future<Header> & header,
                JobQueue<Job> & writequeue,
                JobCounter & jobcounter)
{
   header.wait();
   Header h = header.get();

   auto start = std::chrono::high_resolution_clock::now();
   cout << "Writing to disk.." << endl;

   h.filename[h.filename_length] = '\0';

   fstream filedata(h.filename, ios::in | ios::out | ios::binary | ios::trunc);

   while(true)
   {
      Job j = writequeue.receive();
      if(j.kill){break;};
      filedata.write(j.buf ,j.size);

      delete[] j.buf;
      jobcounter.dec();
   }

   filedata.close();

   auto end = std::chrono::high_resolution_clock::now();
   double sec = std::chrono::duration<double>(end - start).count();

   // OUTPUT
   double write_speed = (((double)(h.filesize*8)/1000000)/sec);
   stringstream out (stringstream::in | stringstream::out);

   out << "write_start: ";
   out << fixed << std::chrono::duration<double>(
                start.time_since_epoch()).count() << endl;
   out << "write_end: ";
   out << fixed << std::chrono::duration<double>(
                end.time_since_epoch()).count() << endl;
   out << "Write finished: avg write speed ";
   out << write_speed << " Mbit/s" << " for " << sec << "s" << endl;

   cout << out.str();
}

void forward_data_server(
                boost::shared_future<Header> & header,
                JobQueue<Job> & sendqueue,
                JobQueue<Job> & writequeue,
                string port,
                string host)
{
   int n;

   header.wait();
   Header h = header.get();

   // Establish tcp connection options
   boost::asio::io_service io_service;
   tcp::resolver resolver(io_service);
   tcp::resolver::query query(tcp::v4(), host ,port);
   tcp::resolver::iterator endpoint_iterator = resolver.resolve(query);
```

```
    tcp::socket socket(io_service);

    // Establish a connection.
    boost::asio::connect(socket, endpoint_iterator);

    // Send header
    int bytes_sent = 0;
    while(bytes_sent < h.head_size)
    {
       n = boost::asio::write(
                        socket,
                        boost::asio::buffer(
                                       h.head + bytes_sent,
                                       h.head_size - bytes_sent));
       bytes_sent += n;
    }

    // Send filename
    bytes_sent = 0;
    while(bytes_sent < h.filename_length)
    {
       n = boost::asio::write(
                        socket,
                        boost::asio::buffer(
                                       h.filename + bytes_sent,
                                       h.filename_length - bytes_sent));
       bytes_sent += n;
    }

    stringstream out (stringstream::in | stringstream::out);
    out << "fwd_job: ";
    int i = 0;

    auto start = std::chrono::high_resolution_clock::now();
    cout << "Forwarding data.." << endl;

    while(true)
    {
       Job j = sendqueue.receive();
       if(j.kill){writequeue.put(move(j));break;}

       int bytes_sent = 0;
       while(bytes_sent < j.size)
       {
          n = boost::asio::write(
                        socket,
                        boost::asio::buffer(
                                       j.buf + bytes_sent,
                                       j.size - bytes_sent));
          bytes_sent += n;
       }

       // Record arrivaltimes of first 4 jobs
       if(i < 4)
       {
          auto pkttime = std::chrono::high_resolution_clock::now();
          out << fixed << std::chrono::duration<double>(
                        pkttime.time_since_epoch()).count() << " ";
          i++;
       }

       writequeue.put(move(j));
```

```
   }

   socket.close();

   auto end = std::chrono::high_resolution_clock::now();
   double sec = std::chrono::duration<double>(end - start).count();

   double forward_speed = (((double)(h.filesize*8)/1000000)/sec);
   out << endl;
   out << "fwd_start: ";
   out << fixed << std::chrono::duration<double>(
             start.time_since_epoch()).count() << endl;
   out << "fwd_end: ";
   out << fixed << std::chrono::duration<double>(
             end.time_since_epoch()).count() << endl;
   out << "Forward finished: avg forward speed ";
   out << forward_speed << " Mbit/s" << " for " << sec << "s" << endl;

   cout << out.str();
}

void send_data(
        boost::shared_future<Header> & header,
        JobQueue<Job> & sendqueue,
        string port,
        string host,
        JobCounter & jobcounter)
{
   int n;

   header.wait();
   Header h = header.get();

   // Establish tcp connection options
   boost::asio::io_service io_service;
   tcp::resolver resolver(io_service);
   tcp::resolver::query query(tcp::v4(), host ,port);
   tcp::resolver::iterator endpoint_iterator = resolver.resolve(query);
   tcp::socket socket(io_service);

   // Establish a connection.
   boost::asio::connect(socket, endpoint_iterator);

   // Send header
   int bytes_sent = 0;
   while(bytes_sent < h.head_size)
   {
      n = boost::asio::write(
                      socket,
                      boost::asio::buffer(
                                      h.head + bytes_sent,
                                      h.head_size - bytes_sent));
      bytes_sent += n;
   }

   // Send filename
   bytes_sent = 0;
   while(bytes_sent < h.filename_length)
   {
      n = boost::asio::write(
                      socket,
                      boost::asio::buffer(
```

```
                                          h.filename + bytes_sent,
                                          h.filename_length - bytes_sent));
        bytes_sent += n;
    }

    stringstream out (stringstream::in | stringstream::out);
    out << "sent_job: ";
    int i = 0;

    auto start = std::chrono::high_resolution_clock::now();
    cout << "Sending file.." << endl;

    while(true)
    {
        Job j = sendqueue.receive();
        if(j.kill){break;}

        int bytes_sent = 0;
        while(bytes_sent < j.size)
        {
          n = boost::asio::write(
                          socket,
                          boost::asio::buffer(
                                          j.buf + bytes_sent,
                                          j.size - bytes_sent));
          bytes_sent += n;
        }

        // Record arrivaltimes of first 4 jobs
        if(i < 4)
        {
           auto pkttime = std::chrono::high_resolution_clock::now();
           out << fixed << std::chrono::duration<double>(
                          pkttime.time_since_epoch()).count() << " ";
           i++;
        }

        delete[] j.buf;
        jobcounter.dec();
    }

    socket.close();

    auto end = std::chrono::high_resolution_clock::now();
    double sec = std::chrono::duration<double>(end - start).count();

    double send_speed = (((double)(h.filesize*8)/1000000)/sec);
    out << endl;
    out << "send_start: ";
    out << fixed << std::chrono::duration<double>(
                          start.time_since_epoch()).count() << endl;
    out << "send_end: ";
    out << fixed << std::chrono::duration<double>(
                          end.time_since_epoch()).count() << endl;
    out << "Send finished: avg send speed ";
    out << send_speed << " Mbit/s" << " for " << sec << "s" << endl;

    cout << out.str();
}

void read_from_file(
        boost::promise<Header> & header_created,
```

```
        boost::promise<
                std::chrono::time_point<
                        std::chrono::high_resolution_clock>> & send_start_receive,
        int chunk_size,
        string file,
        JobQueue<Job> & sendqueue,
        JobCounter & jobcounter,
        int buffer_size)
{
   send_start_receive.set_value(std::chrono::high_resolution_clock::now());

   Header h;
   h.chunk_size = chunk_size;

   fstream filedata(file.c_str(), ios::in | ios::binary | ios::ate);
   int64_t filesize = filedata.tellg();
   filedata.seekg(0, ios::beg);

   short filename_size = file.length();

   memcpy(h.head, &chunk_size,4);
   memcpy(h.head + 4, &filename_size,2);
   memcpy(h.head + 6, &filesize,8);
   strcpy(h.filename, file.c_str());
   memcpy(&h.chunk_size, h.head,4);
   memcpy(&h.filename_length, h.head+4,2);
   memcpy(&h.filesize, h.head+6,8);

   header_created.set_value(h);
   int chunks_to_read = filesize / chunk_size;
   int left = filesize % chunk_size;
   buffer_size = buffer_size * 1048576; // MiB

   jobcounter.setValues((buffer_size / chunk_size), chunks_to_read);

   auto start = std::chrono::high_resolution_clock::now();
   cout << "Reading file.." << endl;

   do
   {
      Job j;
      char* buffer = new char [h.chunk_size];

      filedata.read(buffer,h.chunk_size);

      j.buf = move(buffer);
      j.size = h.chunk_size;
      sendqueue.put(move(j));

   }while(jobcounter.inc());

   // Receive leftover bytes that did not fill up a chunk
   if(left > 0)
   {
      Job j;
      char* buffer = new char [left];

      filedata.read(buffer,left);

      j.buf = move(buffer);
      j.size = left;
      sendqueue.put(move(j));
```

```
    }

    Job j;
    j.kill = true;
    sendqueue.put(move(j)); // Send kill job

    filedata.close();

    auto end = std::chrono::high_resolution_clock::now();
    double sec = std::chrono::duration<double>(end - start).count();

    stringstream out (stringstream::in | stringstream::out);

    double read_speed = (((double)(h.filesize*8)/1000000)/sec);
    out << "read_start: ";
    out << fixed << std::chrono::duration<double>(
                start.time_since_epoch()).count() << endl;
    out << "read_end: ";
    out << fixed << std::chrono::duration<double>(
                end.time_since_epoch()).count() << endl;
    out << "Read finished: avg read speed ";
    out << read_speed << " Mbit/s" << " for " << sec << "s" << endl;

    cout << out.str();
}

int main(int argc, char *argv[])
{
    //--> INIT VARIABLES
    //------------------

    const string version = "23.04.12";
    int port;
    string fwdport; // Must be string
    string fwdhost;
    int chunk_size;
    string file;
    int buffer_size;

    JobQueue<Job> sendqueue;
    JobQueue<Job> writequeue;
    JobCounter jobcounter;

    // Promise of sending header
    boost::promise<Header> send_header;
    boost::shared_future<Header> header;
    header = send_header.get_future();

    // Promise of sending start message
    boost::promise<
         std::chrono::time_point<
                std::chrono::high_resolution_clock>> send_start_receive;

    boost::unique_future<
         std::chrono::time_point<
                std::chrono::high_resolution_clock>> start_receive;

    start_receive = send_start_receive.get_future();

    //--> HANDLE INPUT
    //----------------

```

```
    // Declare the supported commandline options.
    po::options_description o_generic("Generic options", 1024);
    o_generic.add_options()
      ("help,h", "  Prints usage")
      ("version,v","  Prints version")
      ("buffer_size,b",
          po::value<int>(&buffer_size)->default_value(600),
          "  Buffer size in MiB")
    ;

    // Declare the supported commandline options.
    po::options_description o_receive("Receive options", 1024);
    o_receive.add_options()
      ("receive,R", "      Invoke receive behavior")
      ("r_port,r",
          po::value<int>(&port)->default_value(9000),
          "      Data in port")
     ;

    // Declare the supported commandline options.
    po::options_description o_forward("Forward options", 1024);
    o_forward.add_options()
      ("forward,F",
          po::value<string>(&fwdhost),
          "     Invoke forward behavior, specify host address")
      ("fr_port,i",
          po::value<int>(&port)->default_value(9000),
          "     Data in port")
      ("fs_port,o",
          po::value<string>(&fwdport)->default_value("9000"),
          "     Data out to port") // Must be string, do not change
     ;

    // Declare the supported commandline options.
    po::options_description o_send("Send options", 1024);
    o_send.add_options()
      ("send,S",
          po::value<string>(&fwdhost),
          "    Invoke send behavior, specify host address")
      ("s_port,p",
          po::value<string>(&fwdport)->default_value("9000"),
          "    Data out to port")
      ("file,f",
          po::value<string>(&file),
          "    Specify send file")
      ("job_size,c",
          po::value<int>(&chunk_size)->default_value(1460),
          "    Define send/write/receive chunk size")
     ;

    // Options for print
    po::options_description cmd_options("Commandline options");
    cmd_options.add(o_generic).add(o_send).add(o_forward).add(o_receive);

    po::variables_map vm;
    po::store(po::parse_command_line(argc, argv, cmd_options), vm);
    po::notify(vm);

    // Print usage if help is invoked
    if (vm.count("help"))
    {
       cout << cmd_options << endl;
```

```
        return 1;
    }

    // Print version if invoked
    if (vm.count("version"))
    {
        cout << "Development version: " << version << endl;
        return 1;
    }

    //--> MAIN EXECUTION
    //------------------

    if(vm.count("forward"))
    {
        boost::thread th_read(
                            &receive,
                            std::ref(send_header),
                            std::ref(send_start_receive),
                            port,std::ref(sendqueue),
                            std::ref(jobcounter),
                            buffer_size);

        boost::thread th_forward(
                            &forward_data_server,
                            std::ref(header),
                            std::ref(sendqueue),
                            std::ref(writequeue),
                            fwdport,fwdhost);

        boost::thread th_disk(
                            &write_to_disk_server,
                            std::ref(header),
                            std::ref(writequeue),
                            std::ref(jobcounter));

        cout << "Forward behavior invoked, main thread waiting at barrier" << endl;

        th_read.join();
        th_forward.join();
        th_disk.join();
    }
    else if(vm.count("receive"))
    {
        boost::thread th_read(
                            &receive,
                            std::ref(send_header),
                            std::ref(send_start_receive),
                            port,std::ref(sendqueue),
                            std::ref(jobcounter),
                            buffer_size);

        boost::thread th_disk(
                            &write_to_disk_server,
                            std::ref(header),
                            std::ref(sendqueue),
                            std::ref(jobcounter));

        cout << "Receive behavior invoked, main thread waiting at barrier" << endl;

        th_read.join();
        th_disk.join();
```

```
    }
    else if(vm.count("send"))
    {
       boost::thread th_read(
                          &read_from_file,
                          std::ref(send_header),
                          std::ref(send_start_receive),
                          chunk_size,
                          file,std::ref(sendqueue),
                          std::ref(jobcounter),
                          buffer_size);

       boost::thread th_send(
                          &send_data,std::ref(header),
                          std::ref(sendqueue),
                          fwdport,fwdhost,
                          std::ref(jobcounter));

       cout << "Send behavior invoked, main thread waiting at barrier" << endl;

       th_read.join();
       th_send.join();
    }
    else
    {
       cout << "No behavior was invoked" << endl;
       cout << cmd_options << endl;
       return 1;
    }

    auto start = start_receive.get();
    auto end = std::chrono::high_resolution_clock::now();
    double sec = std::chrono::duration<double>(end - start).count();

    cout << "Total exectution time: " << sec << " seconds" << endl;

    return 0;
 }
```