An Evaluation of Bottleneck Bandwidth Round Trip Time for Improved Congestion Control in Diverse Network Environments

Jonathan Starkey

2019

BSc Computing Final Year Project

Bournemouth University
Faculty of Science & Technology
Department of Computing and Informatics

Abstract

This project will compare other TCP congestion control algorithms alongside Bottleneck Bandwidth and Round-trip propagation time (BBR), to discern where the BBR algorithm could be applied constructively, and evaluate in a range of environments the algorithm's behaviour with regard to attributes commonly measured with research into similar transport protocols evaluations (throughput, fairness, adaptability etc).
Using network simulator 3 (ns-3) the project replicated published results before building a framework for BBR evaluation and mimicking complex network environments with different networking attributes (number of flows, rates of traffic, paths of traffic). With further work the paper could also research the implementation of the BBR algorithm in a range of modern networking environments (mobile, tethering etc).

This paper draws on the history of TCP and importance of congestion control in networks. Successfully, this paper highlights the experiences of BBR on given network topologies. Evaluating throughput and fairness BBR simulations are set up alongside CUBIC and NewReno simulations enabling critical analysis of the BBR algorithm and critique of its performance compared to CUBIC and NewReno.

Dissertation Declaration

I agree that, should the University wish to retain it for reference purposes, a copy of my dissertation may be held by Bournemouth University normally for a period of 3 academic years. I understand that once the retention period has expired my dissertation will be destroyed.

Confidentiality

I confirm that this dissertation does not contain information of a commercial or confidential nature or include personal information other than that which would normally be in the public domain unless the relevant permissions have been obtained. In particular any information which identifies a particular individual's religious or political beliefs, information relating to their health, ethnicity, criminal history or sex life has been anonymised unless permission has been granted for its publication from the person to whom it relates.

Copyright

The copyright for this dissertation remains with me.

Requests for Information

I agree that this dissertation may be made available as the result of a request for information under the Freedom of Information Act.

Signed:
Name: Jonathan Paul Starkey
Date:
Programme: BSc Computing

Original Work Declaration

This dissertation and the project that it is based on are my own work, except where stated, in accordance with University regulations.

Signed:

Contents

1 Introduction **1**

1.1 Problem Definition 1

1.2 Problem Objectives 2

1.2.1 Objectives, aims and success criteria 2

1.3 Research Rationale 4

1.4 Project Plan 4

1.5 Risk Analysis 5

2 Background Study **6**

2.1 TCP Congestion Control 6

2.1.1 TCP Underlying Algorithms 7

3 Literature Review **9**

3.1 Delay Based Congestion Control 9

3.2 Loss Based Congestion Control 11

3.2.1 Active Queue Management (AQM) 11

3.2.2 Explicit Congestion Notification (ECN) 13

3.2.3 Congestion Based Congestion Control 13

3.3 Bandwidth 14

3.3.1 Ethernet (single flow) 15

3.3.2 Wireless 16

3.4 Fairness 16

3.4.1 Intra protocol flows 18

3.4.2 Inter protocol flows 19

3.5 Reaction to network changes 21

3.5.1 New flows joining the network 22

4 Methodology **26**

4.1 Development Methodology 26

4.1.1 Available Methodologies 26

4.1.2 Chosen Methodology 27

4.2 Development Software 27

5 Analysis and Design **28**
5.1 Topologies 28
5.1.1 Line Topology 28
5.1.2 Modified Line Topology 28
5.1.3 2 by 2 Horizontal Dumbbell 29
5.1.4 2 by 2 Vertical Dumbbell 29
5.1.5 3 by 3 Horizontal Dumbbell 30
5.1.6 Line Topology - One New Flow 30
5.1.7 Line Topology - Two New Flows 31
5.2 Coding 31
5.2.1 Simulation Coding 32
5.2.2 Results Coding 32

6 Results **34**
6.1 Replication of Results 34
6.1.1 BBR - Claypool 34
6.1.2 BBR - Shiyao 35
6.1.3 Cubic and NewReno 36
6.2 Bandwidth 40
6.3 Fairness 45
6.3.1 Delayed Joining Flows 53
6.3.2 Multiple Joining Flows 59
6.4 Summary 62
6.4.1 Results 62
6.4.2 Artefact 63

7 Conclusion **64**
7.1 Evaluation 64
7.2 Future Work 65

Appendices **69**

List of Figures

2.1 TCP Slow Start and Congestion Avoidance 8

3.1 TCP Reno undergoing congestion window resizing 12
3.2 BBR - Kleinrock's (A) Optimal Point for Inflight Data 14
3.3 BBR Global Ethernet Results (Tongyu Dai, 2018) 15
3.4 BBR 4G mobile Results (Tongyu Dai, 2018) 16
3.5 BBR 4G LTE Results (Claypool, 2018) 17
3.6 50-ms RTT Average Goodput 87.3 Mbps, 10-ms RTT Average Goodput 6.3 Mbps (Shiyao, 2017) 18
3.7 IETF 99 Update of BBR convergence (Cardwell, July 2017) 19
3.8 Slide 16 - Throughput vs Time (Cardwell, 2017) 20
3.9 Slide 17 - Throughput vs Time (Cardwell, 2017) 21
3.10 Slide 18 - Throughput vs Time (Cardwell, 2017) 22
3.11 BBR adapting to network changes (Scholz, 2017) 23
3.12 BBR adapting to new flows (Cardwell, 2016) 24
3.13 BBR adapting to new flows in relation to Probe RTT of existing flows (Scholz, 2017) 24
3.14 BBR adapting to new flows during and after Probe RTT of existing flows (Scholz, 2017) 25

5.1 Line topology (3 nodes) 28
5.2 2 competing flows - modified line topology 29
5.3 2x2 horizontal dumbbell topology 30
5.4 2x2 vertical dumbbell topology 30
5.5 3x3 horizontal dumbbell topology 31
5.6 Introducing a new flow 31
5.7 Introducing multiple new flows 32

6.1 Claypool Replication Results 35
6.2 Shiya Replication Results 36
6.3 Claypool Replication Results CUBIC 37

6.4 Claypool Replication Results NewReno 38
6.5 Shiya Replication Results CUBIC 39
6.6 Shiya Replication Results NewReno 40
6.7 Results for One vs One Horizontal Dumbbell over BBR 41
6.8 Results for One vs Two Horizontal Dumbbell over BBR 42
6.9 Results for One vs One Vertical Dumbbell over BBR 43
6.10 Results for One vs One Vertical Dumbbell over CUBIC 44
6.11 Results for 3 flow 3x3 Dumbbell over BBR 45
6.12 Results for One vs One Horizontal Dumbbell over BBR 46
6.13 Results for One vs Two Horizontal Dumbbell over BBR 47
6.14 Results for One vs Two Vertical Dumbbell over BBR 48
6.15 Results for 3 flow 3x3 Dumbbell over CUBIC 49
6.16 Results for 3 flow 3x3 Dumbbell over NewReno 50
6.17 Results for joining flow on node 2 over BBR 51
6.18 Results for joining flow on node 2 over CUBIC 52
6.19 Results for joining flow on node 2 over NewReno 53
6.20 Results for 200ms delayed joining flow joining on node 2 running BBR . . . 55
6.21 Results for 400ms delayed joining flow joining on node 2 running BBR . . . 56
6.22 Results for 600ms delayed joining flow joining on node 2 running BBR . . . 57
6.23 Results for 800ms delayed joining flow joining on node 2 running BBR . . . 58
6.24 Results for 1000ms delayed joining flow joining on node 2 running BBR . . 59
6.25 Results for joining flows on nodes 1 and 2 running BBR 60
6.26 Results for joining flows on nodes 1 and 2 running CUBIC 61
6.27 Results for joining flows on nodes 1 and 2 running NewReno 62

1 Original plan time sheet 81
2 Updated time sheet 82

Chapter 1

Introduction

The internet provides a platform to serve and consume content through a vast array of multimedia formats simultaneously. Whilst there is more information on the internet than any one person can process, the information we desire must be delivered near instantly for a user to consider consuming. Progress to improve the speed and efficiency of connections is thus never ending, with an increasing number of ever more capable devices connected to the internet, more diverse and demanding connection conditions arise. Keeping up with this demand allows business to provide services to more clients and applies pressure on academics to constantly investigate and experiment with different implementations for networking protocols and topologies.
This study investigates and experiments with a proposed new TCP algorithm implementation named BBR through the use of simulations and theory. Using NS-3 and python this study aims to provide performance metrics given a range of environmental factors.
Furthermore this document sets out the engineering of such a program so that it may be repeated, replicated and reproduced as defined by ACM (Bonaventure, 2016) with the further aim of extension if the algorithm is revised. Featured are details for the requirements to tackle such a task and logically justifies the approach and methodology.
The artefact comprises of software solution that can aid network engineers and academics understand the properties of said new implementation and freely compare with past and future algorithms, without the expenditure commonly occurred when revising network architectures.

1.1 Problem Definition

Bandwidth requirements for internet users is greatly increasing across many countries around the world. However these improvements are irrelevant if the network pipe (the connection between devices that dictates the maximum bandwidth) is not fully utilised. Other TCP congestion control algorithms fail to maintain the desired pipe utilisation ca-

pacity, known as Kleinrock's optimal operating point (kleinrock, 2979).

As well as data in flight utilisation, TCP congestion control algorithms are responsible for maintaining data delivery at the edge router. Current the internet suffers from buffer bloat - large packet queues in router memory due to processor speed or memory capacity bottlenecks resulting in the drop of incoming packets until the queue clears.

Current TCP implementations are heavily critiqued, whilst some perform well in a given field no current algorithm dominates in all fields (eg. An algorithm may have high throughput on Ethernet connections but only because its bandwidth sharing fairness is very low). Due to how TCP congestion control algorithms have evolved over time, interpretation of congestion means algorithms do not operate sufficiently when sharing bandwidth.

It is therefore important that BBR be tested in a range of fields to better understand the strengths and weaknesses when operating in simulations comparable to the real world. It is only when weaknesses are discovered and fully understood can improvements be made to the algorithm.

In order to collaborate this effort, a common framework must be shared for researchers to reproduce and extend the work of others. This investigation focuses on BBR algorithm's throughput, intra-flow and inter-flow fairness and reactions to changes in the network using open source software written in python and C++.

1.2 Problem Objectives

The following describe the project objectives and provide aims to monitor progress. These aims act as milestones and allow for retrospective assessment towards the end of the project. They align with Gnatt charts accessible in the appendix.

1.2.1 Objectives, aims and success criteria

Objective 1 - Research BBR algorithm
Aim 1: Gain understanding of algorithm operation/implementation.
Aim 2: Gain access to sites/groups where BBR discussion occurs.
Success Criteria 1 - Hold an understanding of algorithm operation and current strengths and weaknesses. Hold an understanding of what other researches are currently investigating.

Objective 2 - Set up a network simulator environment for working with BBR algorithm.
Aim 1: Investigate current open source network simulators.
Aim 2: Investigate possible network simulators compatibility with BBR.
Aim 3: Ensure simulator provides means to modify and create custom networks.

Aim 4: Ensure simulator provides means to incorporate other congestion control algorithms.
Aim 5: Ensure simulator can replicate results already published.
Success Criteria 2 - Have a simulator that can run BBR alongside previous TCP congestion control algorithms and be able to create custom network configurations using said simulator.

Objective 3 - Investigate BBR throughput
Dependencies - Objectives 1 and 2 completed first.
Aim 1: Set up custom networks with varying number of flows.
Aim 2: Set up custom scripts to extract results.
Aim 3: Analyse results and compare with published results of others.
Aim 4: Suggestion improvements to BBR algorithm from results.
Success Criteria 3 - Have a number of scripts that replicate the results of others and produce results for custom environments. Have a list of suggested improvements based on my findings.

Objective 4 - Investigate BBR inter/intra protocol fairness
Dependencies - Objectives 1 and 2 completed first.
Aim 1: Set up custom networks with varying number of flows.
Aim 2: Set up custom scripts to extract results.
Aim 3: Analyse results and compare with published results of others.
Aim 4: Suggestion improvements to BBR algorithm from results.
Success Criteria 4 - Have a number of scripts that replicate the results of others and produce results for custom environments. Have a list of suggested improvements based on my findings.

Objective 5 - Investigate BBR behaviour with varying flow paths
Dependencies - Objectives 1 and 2 completed first.
Aim 1: Set up custom networks with varying number of flow paths.
Aim 2: Set up custom scripts to extract results.
Aim 3: Analyse results and compare with published results of others.
Aim 4: Suggestion improvements to BBR algorithm from results.
Success Criteria 4 - Have a number of scripts that replicate the results of others and produce results for custom environments. Have a list of suggested improvements based on my findings.

Objective 6 - Investigate BBR behaviour during network changes
Dependencies - Objectives 1 and 2 completed first.

Aim 1: Set up custom networks that allow for run time changes in bottleneck.
Aim 2: Set up custom networks that allow for run time changes in number of flows.
Aim 3: Set up custom networks that allow for run time changes in delayed joining flows.
Aim 4: Analyse results and compare with published results of others.
Aim 5: Suggestion improvements to BBR algorithm from results.
Success Criteria 6 - Have a number of scripts that modify the network during simulation run time. Compare results with that of published results. Have a list of suggested improvements based on my findings.

1.3 Research Rationale

The BBR algorithm is undergoing researched currently at many universities and technology companies such as Google. As such the majority of theoretical research will be conducted reading articles. This will also be the source for results when validating the simulator. Due to its preliminary nature of BBR, the business gain can not be fully researched and the full cost of developing such an algorithm are yet known. However BBR has gained enough popularity to be included in the Linux kernel since version 4.9 and is therefore free to implement on supporting Linux machines. The intention of this project is to support the work already under investigation and to provide a software solution that would aid in the further research of both BBR and alternative TCP congestion control algorithms.
Firstly this software solution will feature no financial barrier - it will use open source software and scripts will be freely accessible. Running a simulator, it not will require purchasing of additional hardware and will run on most modern computers. This ensures repeatability, replicability and reproducibility (Bonaventure, 2016). Secondly the software will be user friendly. The simulator code will be written in C++ and will conform to the coding conventions set out by ns-3 (NS-3, 2019).

1.4 Project Plan

Gnatts charts can be seen in appendix. These charts detail the time estimations for deconstructed objectives in chronological order.

1.5 Risk Analysis

Table 1.1 in the appendix highlights potential risks during research and suggested solutions.

Chapter 2

Background Study

The following research highlights research currently undertaken in congestion control. The primary focus will be on the algorithms behaviour in different networking configurations and deriving results for later comparison to the simulation.

2.1 TCP Congestion Control

The mid 1970s featured the arrival of TCP - a transport layer internet suite protocol designed to provide reliability, error checking and ordering via a connection window strategy. (Vinton, 1974).
In the late 1980s research to improve TCP lead to the addition of congestion control algorithms (Jacobson, 1988) to reduce packet drop rates. The algorithms originally developed by Jacobson Et al. adhered to one observation: ”The flow on a TCP connection ... should obey a ’conservation of packets’”. This conservation principle simply states that where a connection exists, the volume of packets entering the connection must be equal to volume exiting the connection.
In order to ensure such equilibrium connections can not rely solely on ACK packets (acknowledgement packets that confirm when a packet has been received). Such packets are sent on the receiving of data and, due to the conservation of packets principle, would mean the initial connection establishment packets could never be sent. Secondly in the event that a packet is dropped due to time out or buffer overflow, no ACK packet is received. Thus ACK packets alone are not a sufficient indicator for network congestion control. However early on in TCP, ACK packet monitoring was the main method for congestion detection.
Jacobson Et al. proposes a number of changes to TCP. Using both mean and variation of round trip time to indicate they calculate suitable time out periods for transmission of packet. They also set out improvements for congestion window resizing to aid in congestion avoidance. In addition they explain the algorithms that form part of the TCP RFC

standards - RFC 2581 (Allman, 1999) and RFC 5681 (Allman, 2009) which obsoletes RFC 2581. At the time however there was no congestion detection, ”Only in gateways, at the convergence of flows, is there enough information to control sharing and fair allocation. Thus, we view the gateway ’congestion detection’ algorithm as the next big step”.

2.1.1 TCP Underlying Algorithms

With the further creation of TCP congestion control algorithms came changes to implementations of slow start, congestion avoidance, fast retransmit, and fast recovery. To implement these algorithms, two variables are created and stored per connection. Using the window principle, the congestion window (cwnd) limits the sender data that can be transmitted into the network pipe before receiving an ACK packet. The receiver's advertised window (rwnd) likewise limits the amount of outstanding data at the destination. Where the minimum of cwnd/rwnd value dictates a bottleneck.
When a TCP connection opens a socket in a network with unknown conditions TCP implementations will slowly probe the network to determine the available capacity, in order to avoid congesting the network with an inappropriately large burst of data. The slow start algorithm is used for this purpose at start of a connection establishment, or after recovering from packet drop. At the beginning of slow start the cwnd is calculated by exponentially increasing the window size until it exceeds threshold. Over time as the connection continues to successfully send packets the cwnd is increased additively. As the receiver successfully acknowledges packets, so too does the rwnd (receiver window) grow. Similarly when packet loss is detected the cwnd is reduced to allow congestion to alleviate.
Distinct implementations of TCP modify the rate of adjustment of cwnd and rwnd differently, usually as a result of what environments an implementation wishes to target:- Low latency, high bandwidth, equal fairness. Additive Increase/Multiplicative Decrease (AIMD) refers to the behavior of these algorithms when resizing the cwnd. Take this example (Sangtae Ha, 2005) with TCP NewReno additive increase of cnwd per ACK:
”For instance, if the bandwidth of a network path is 10 Gbps and the RTT is 100 ms, with packets of 1250 bytes, the BDP of the path is around 100,000 packets. For TCP to grow its window from the midpoint of the BDP, say 50,000, it takes about 50,000 RTTs which amounts to 5000 seconds (1.4 hours). If a flow finishes before that time, it severely under-utilizes the path.”.
This example highlights that particular algorithms may only be suited for particular network environments, a consideration that must be factored when designing simulations later in the project.
Another per connection variable is the slow start threshold (ssthresh). This variable is

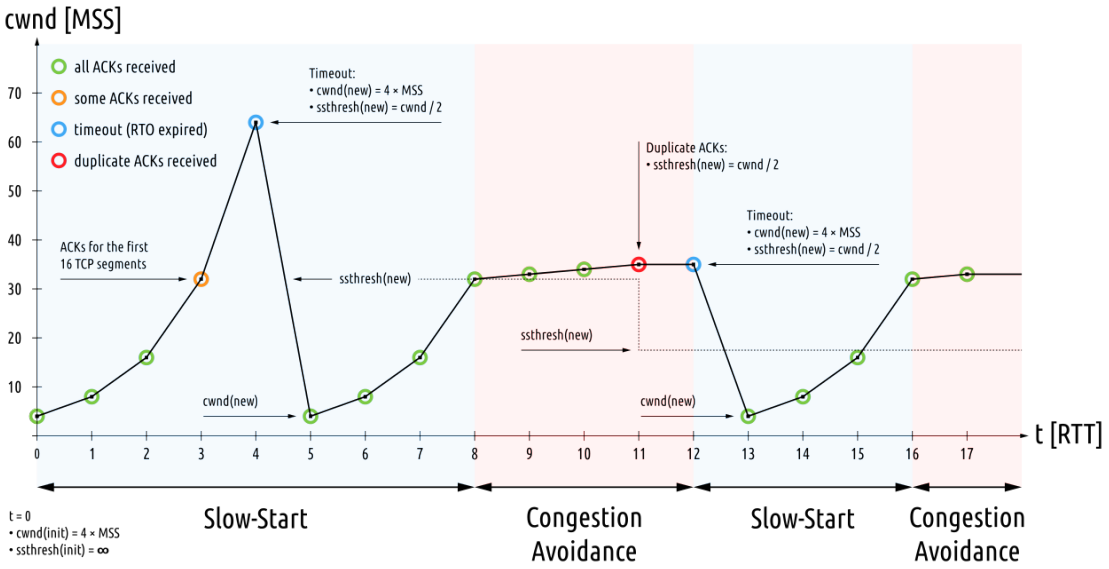


Figure 2.1: TCP Slow Start and Congestion Avoidance

used to determine whether the slow start or congestion avoidance algorithm is used. By setting ssthresh to the largest available window size and reducing it when congestion occurs, the network dictates the maximum slow start threshold size and provides cwnd with a target value. Whilst cwnd is less than ssthresh TCP algorithms stay in slow start and continue to increase the window size. When cwnd is more than or equal to ssthresh, algorithms must enter congestion avoidance. This interaction can be see in figure 2.1 (Wikimedia commons, 2014).

Fast retransmit and fast recovery work together. The fast retransmit algorithm uses the arrival of 3 duplicate ACKs as an indication that a segment has been lost. After receiving 3 duplicate ACKs, the cwnd undergoes multiplicative decrease and retransmission of the perceived missing segment occurs. 3 duplicate ACKs is required to avoid misinterpretation between the source and destination nodes, as the process for re-ordering out-of-order packets at the destination spews multiple ACKs.

After the retransmitted packet is sent the ”fast recovery” algorithm manages the transmission of packets until a non-duplicate ACK arrives. Essentially this process regards the number of duplicated ACKs as a sign of packet arrival at the receiver, something we know is true from the conservation of packets principle. TCP processes these additional duplicate ACKs as if they had acknowledged data correctly, the sequence number is updated and subsequent packets can now be sent, in the expected order.

Using the fast retransmit and fast recovery an algorithm can reduce the cwnd, allow congestion to alleviate and continue transmitting packets - the connection does not have to undertake another slow start phase.

Chapter 3

Literature Review

The following research highlights research currently undertaken in congestion control. The primary focus will be on the algorithms behaviour in different networking configurations and deriving results for later comparison to the simulation.

3.1 Delay Based Congestion Control

Delay based refers to one such technique of congestion control. With delay based implementations measurements record the expected and actual round trip times to indicate congestion based on the presence of delay. One example of TCP Delay based congestion control algorithms is TCP Vegas. "Vegas measures congestion at a link by its queueing delay, and that of a path by the end–to–end queueing delay (without propagation delay). A Vegas source computes the queueing delay from the round trip time and the estimated propagation delay, and attempts to set its rate to be proportional to the ratio of propagation to queueing delay" (Low, 2000).
TCP Vegas measures the difference between expected and actual throughput (δ) based on RTT delays. When δ is lower than a low threshold (α), the path is not congested and the sending rate in increased. When δ is larger than an upper threshold (β), the sending rate is reduced to alleviate any congestion. If δ is between α and β Vegas maintains the current sending rate. To calculate the expected throughput Vegas divides the current congestion window by the minimum RTT, which contains the delay if there is currently no congestion. For each round trip time, TCP-Vegas computes the actual throughput by dividing the number of packets sent by the sampled RTT. TCP Vegas is highly suited for environments demanding low packet loss. "Vegas' strategy is to adjust the source's sending rate (congestion window) in an attempt to keep a small number of packets buffered in the routers along the transmission path" (Low, 2000).
In Low's paper NS-2 simulator and a dumbbell topology are used to model, compare and evaluate TCP Vegas. Using the dumbbell topology and ftp servers on each node pair,

Lows configuration has a number of benefits:- All flows will have the same path distance, sharing the bottleneck allows for fairness measurements, measurements can be taken before and after the bottleneck (ie at each router), ACK packets must travel the inverse of the route on return (symmetric), the option for flows to have staggered start or start in unison and furthermore additional bottlenecks can be applied between nodes and routers.
Low provides the simulation environmental variables and compares the results with an expected model, however there is no justification for the chosen values or explanation of the model. There is also no provision of NS-2 scripts. Low finds the simulation behaves as per the model. "The model predicts that all connections receive an equal share (1200KBps) of the bottleneck link and the simulations confirm this." (Low, 2000). During the investigations for this project similar simulation validation must occur. This project will likewise provide environmental variables supplemented with justification.
Low details the benefit of a delay based algorithm by comparison to a loss based algorithm TCP Reno. Vegas' delay based algorithm will not cause packet loss once a connection has reached equilibrium, due to its nature of measuring RTT for congestion. Low states "This is in contrast to TCP Reno which constantly probes the network for spare capacity by linearly increasing its window until packets are lost, upon which the window is multiplicatively decreased. Thus, by carefully extracting congestion information from observed round trip time and intelligently reacting to it, Vegas avoids the perpetual cycle of sinking into congestion and recovering from it." This is a key delimiter between delay based and loss based congestion control and again highlights that particular algorithms may only be suited for particular network environments.
The paper also discusses the use of Random Exponential Marking (REM) alongside Vegas. "REM is a congestion control mechanism ... consists of a link algorithm and a source algorithm. The link algorithm computes the link price and feeds it back to sources through packet marking." REM is an example of an active queue management algorithm. AQM algorithms are discussed in further section of this document.
FAST, another delay based congestion control, determines the current congestion window size given the round-trip delays and packet losses over a path. FAST uses a rate-pacing multiplier to update the sending rate after every other RTT. FAST estimates the queuing delay of the path using RTTs and if the delay is well below a threshold, it increases the window aggressively and if it gets closer to the threshold, the algorithm slowly reduces the increasing rate. This is similar to the behaviour exhibited by TCP Reno in figure 3.1. When the delay increases beyond the threshold: first FAST slowly decreases the window and then aggressively decreases the window. For packet losses however FAST behaves like loss based algorithms, halving the congestion window and enters loss recovery.

3.2 Loss Based Congestion Control

Loss based congestion control refers to another technique used for congestion control. With loss based congestion control algorithms the focus is primarily on detecting packet loss as the congestion signal.
“About 30,000 Internet web servers, only 3.31% to 14.4% of web servers are still using TCP Reno, whereas, 14.5% to 25.66% are using TCP Compound and 46.92% are using TCP CUBIC.” (Mudassar Ahmad, 2018). As stated the majority of TCP implementations currently running are all loss based. Packet loss is indicated either through monitoring of packet return time against RTT, thus indicating a time out, or by recording the sequence of ACKs of packets sent around the same time as the suspected lost packet.
TCP CUBIC is derived from Binary Increase Congestion control (BIC), an algorithm with focus on stability and scalability. CUBIC is the most commonly used algorithm, as previously stated by Ahman et. al. above, featuring improvements over BIC most notably its cubic functions for window resizing - an explanation for the ’CU’ in CUBIC. “Instead of a linear increase, CUBIC TCP uses a cubic function. The inflection point of the cubic function is usually shifted to the value on which the last packet loss occurred. This means that the slope of the function is very low around this point but increases with distance to this point. This leads to an improved scalability. Instead of 0.5 (Reno) CUBIC TCP reduces its CWnd merely to a factor of 0.7 after a packet loss.” (Mark Hock, 2017). CUBIC is another example of a TCP algorithm developed for specific networking environments.

Figure 3.1 shows the congestion window resizing of a connection over TCP Reno (Tanenbaum, 2011). A number of TCP algorithms exist and the behaviour they exhibit is heavily dictated by their adaptation of the congestion window. A common theme is to reduce the window size by 0.5, as stated by Hock and seen in Tanenbaum’s graph. This image resembles 2.1, but shows the TCP states specific to Reno.

3.2.1 Active Queue Management (AQM)

Active queue management was introduced as a proposed solution to complement the congestion avoidance algorithms RFC 2309 section 1 (Braden, 1998). AQM was designed to address two major issues:- Lock-out and full queues.
Lock out occurs when a number of original connections fully occupy the queue, a new connection arrives but is unable to successfully append packets to the already full queue and as a result the original connections monopolize the queue space. The reason for such a situation lies in the routers behaviour when the queue is full. Routers undergo ’tail drop’ when the queue is full, dropping incoming packets that can not fit in the buffer. This is known as passive queue management (PQM). Lock-out can be solved with the use of two alternative queue disciplines ”random drop on full” or ”drop front on full”. Under

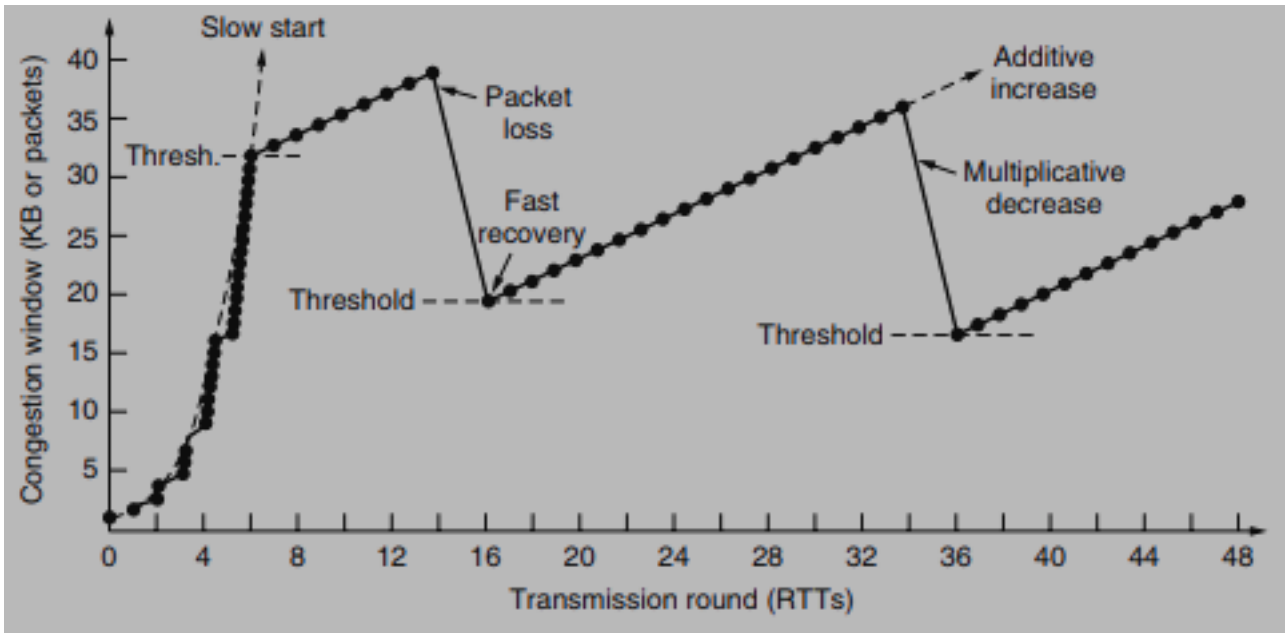


Figure 3.1: TCP Reno undergoing congestion window resizing

the 'random drop on full' discipline, a router drops a randomly selected packet from the queue, rather than the incoming packet, providing space for the newly arrived packet. Likewise the 'drop front on full' discipline drops the front most packet in the queue to provide the required queue space for the newly arrived packet. Both of these solve the lock-out problem.

Full queues occur when the router's buffer is fully utilised. Not only do full queues lead to packet drop and lock-out it also causes increased delay for all other connections sharing the buffer. Due to the nature of internet traffic, full queues can often cause packet drops when a burst of traffic arrives to the network without adequate buffer storage space. AQM resolves the issue of full queues (and lock outs) by running on the receivers router and prematurely dropping packets before the queue is full. Whilst using or not using AQM results in packet loss, not using AQM would result in more packets dropped when a queue does overflow, impacting the congestion model for every connection that processes the packet drop as congestion.

A further side effect of such heavy packet loss at full buffer utilisation, something mitigated by AQM, is the possibility of congestion control synchronisation. If a number of connections were to receive a loss signal at the same time, theoretically they would enter slow start (or similar such mechanism) simultaneously, an undesired effect (Baker, 2015).

AQM selects packets to be dropped allowing for more educated processing than simply the first, last or random packet. This selection process could be used to target based on quality of service (QoS) or to target aggressive connections that appear to not be using congestion avoidance or are presenting unfairness to other connections in some way.

Active queue management can separate policies of queuing or dropping packets from the

policies for indicating congestion. Routers running AQM use the Congestion Experienced (CE) codepoint in a packet header as an indication of congestion, instead of relying solely on packet drops. This has the potential of reducing the impact of loss on latency-sensitive flows (Ramakrishnan, 2001).

3.2.2 Explicit Congestion Notification (ECN)

AQM uses CE codepoint only when using ECN. 'CE packet' is used to denote a packet that has the CE codepoint set. The use of the CE codepoint in ECN allows the TCP connection source to receive the packet, reducing the potential for excessive delays due to both retransmission and the subsequent resizing of cwnd thereof. Such a codepoint exists at bits 6 and 7 in the IPv4 header TOS field. "Upon the receipt by an ECN-Capable transport of a single CE packet, the congestion control algorithms followed at the end-systems MUST be essentially the same as the congestion control response to a *single* dropped packet. For example, for ECN-Capable TCP the source TCP is required to halve its congestion window for any window of data containing either a packet drop or an ECN indication." (Ramakrishnan, 2001).

3.2.3 Congestion Based Congestion Control

BBR takes an alerted approach to other TCP algorithms and as such can not be defined under loss or delay based approaches. Mark (Hock, 2017) proposed "congestion-based" in contrast to loss based or delay based. Whereas loss based congestion control requires packet loss, BBR tries to match the bottleneck in an attempt not to cause packet loss by keeping in flight data at kleinrock's optimal operating point (see figure 3.2). Hock also states "BBR is neither delay-based nor loss-based and it ignores packet loss as congestion signal. It also does not explicitly react to congestion, whereas congestion window-based approaches often use a multiplicative decrease strategy to reduce (data in flight)". However BBR can still compare to other TCP algorithms as BBR must still implement the underlying TCP algorithms in discussed previously in subsection 2.1.1. For example, the slow start phase for bandwidth probing, called 'startup' in BBR, likewise to other algorithms doubles the sending rate per round trip until a queue occurs. Where relevant this document will highlight differences in implementations in order to evaluate BBR against other algorithms.

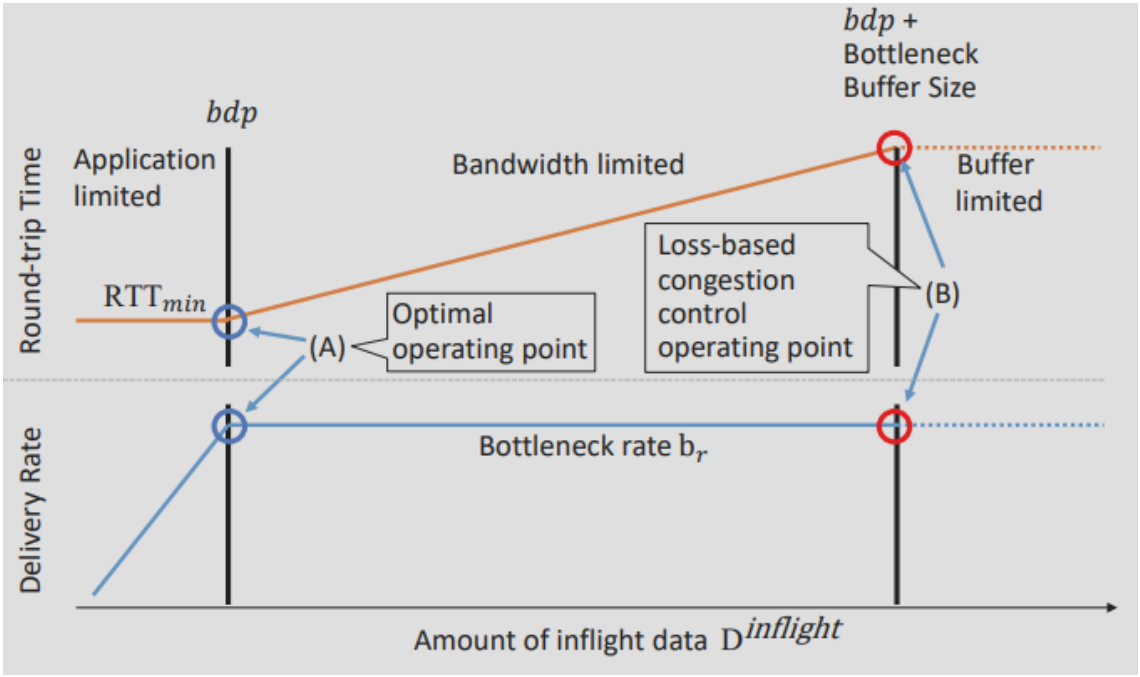


Figure 3.2: BBR - Kleinrock's (A) Optimal Point for Inflight Data

3.3 Bandwidth

Bandwidth utilisation, notably throughput, is one factor that TCP algorithms aim to improve. The problem plaguing bandwidth utilisation is the inability to achieve a balance between full pipe utilisation and over saturation leading to back off and further under utilisation. Often this utilisation must occur whilst competing flows exist, making changes to available bandwidth. Every algorithm has a different approach to bandwidth utilisation.
BBR enters a state known as "ProbeBW". In ProbeBW BBR undertakes gain cycling. Gain cycling "helps BBR flows reach high throughput, low queuing delay, and convergence to a fair share of bandwidth" (Internet Congestion Control Research Group, 2017). This process is a continuous cycle intended to dynamically control pacing rate, using the pacing gain with values: 1.25, 0.75 and 1.
To start the pacing gain is set to 1.25 for the duration required to make this change to all transmitting traffic. This probing checks for more available bandwidth for the duration of one RTT or until packet loss. To mitigate this influx, pacing rate is set to 0.75 (as pacing gain of 1 is equal to the bottleneck bandwidth, 1.25 pacing results in a queue) for the duration again of one round trip and the queue is drained. A pacing rate of 1 to maintain data inflight at the bottleneck bandwidth and provide a number of states (6 at pacing rate of 1) for gain cycle randomisation.

TABLE I: Average throughput [Kbps] and RTT [ms] to America, China, Japan, New Zealand and Switzerland.

	BBR		Cubic		Vegas	
Country	Rate	RTT	Rate	RTT	Rate	RTT
USA	1381	270	1557	260	662	314
CHN	6040	88	6344	68	4074	167
JPN	2599	137	2997	168	2602	150
NZL	802	342	1095	365	660	449
CHE	901	321	1097	316	848	315

Figure 3.3: BBR Global Ethernet Results (Tongyu Dai, 2018)

3.3.1 Ethernet (single flow)

Tongyu Dai Et al. (2018) published results (figure3.3) of a number of non competing sender nodes in different countries running different TCP algorithms. Figure 3.3 shows the average throughput and RTT for five countries over 300 seconds. It is clear that CUBIC trumps BBR in throughput for every country and likewise BBR trumps Vegas. However all algorithms remain competitive in RTT. Whilst such an experiment provides valuable results, such a setup (over multiple countries) does not allow for traffic monitoring at multiple hops along the way, nor does it provide a custom network for alternative configurations or replication.

Hock Et al. research found that ”The intended behavior of BBR can be nicely observed if only a single flow is active at the bottleneck link... (one) BBR flow can fully utilize the provided link capacity. However, throughput drops occur regularly (every 10s), caused by the ProbeRTT phase.” Figure 4 of Hock’s paper supports this statement.
When conducting tests Hock provides all the framework to replicate the results and justifications throughout. Hock uses the dumbbell topology, testing both 1Gbps and 10 Gbps links. Hock’s results also highlight the ’goodput’ of BBR in figures 7 and 8 of their paper. Goodput is a measure of throughput minus any retransmitted packets and overhead, and is an important statistic to measure the sending success rate of an algorithm. Hock finds that a single BBR flow behaves as per the models expectations with both large and small buffers. ProbeBW and ProbeRTT phases, and subsequent effects, are both apparent in the results graph.

TABLE II: Average throughput [bps] and RTT [ms] in 4G mobile network.

	Ave. Throughput	Ave. RTT
BBR	3256	128
Cubic	2997	129
Vegas	4285	94

Figure 3.4: BBR 4G mobile Results (Tongyu Dai, 2018)

3.3.2 Wireless

Tongyu Dai Et al. (2018) published results displaying that BBR performs similarly to Cubic, in terms of both throughput and delay (3.3). However Vegas performs best in 4G mobile network, obtaining 1.3× throughput in comparison to BBR and Cubic (3.4). The results are presented as averages from a 10 minute recording ”on the Fourth Ring Road of Beijing and employed a laptop connected with a 4G mobile to investigate BBR, Cubic and Vegas”. This experiment represents true use cases that would occur when connected to wireless networks:- 4G mobile internet, moving target, user equipment handovers occurring, user equipment contained within a car and downloading for a duration of 10 minutes. However the experiments are not repeated and due to the exact nature of the ring road and lacking details on the exact route and distance between senders and receivers, are unlikely to be replicable or reproducible by others.

Claypool Et al. (2018) also published results for wireless networks, using the simulation software NS3. They compare BBR prime (AKA ”BBR’” - the BBR implementation for NS3) to CUBIC. Figure 3.5 shows three figures from their paper. The top figure shows the topology used. The second figure shows the throughput and RTT against time. In this second figure we see that BBR’ is comparable in average throughput but outperforms CUBIC, with almost half the average RTT. In the bottom figure we see throughput and RTT again but against user equipment distance. Again BBR’ outperforms CUBIC in terms of throughput and significantly trumps CUBIC in regards to RTT when the user equipment is moving further away from the source.

3.4 Fairness

Fairness is an attribute assessed when TCP algorithms share the bandwidth. Fairness, ideally, dictates that all flows sharing bandwidth must do so evenly. A simple example:

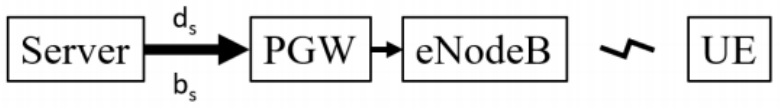


Figure 9: Wireless Topology

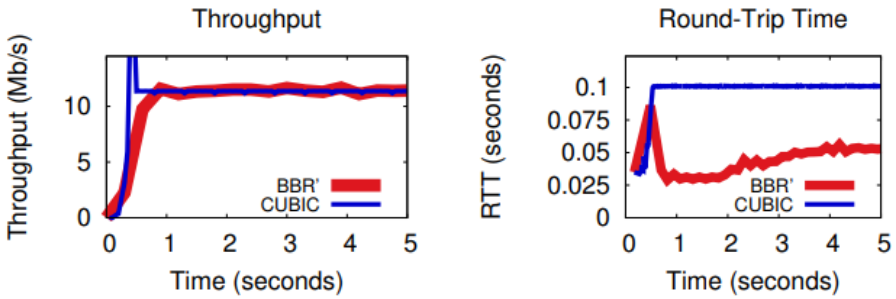


Figure 10: 4G LTE Network (UE Distance is 5k)

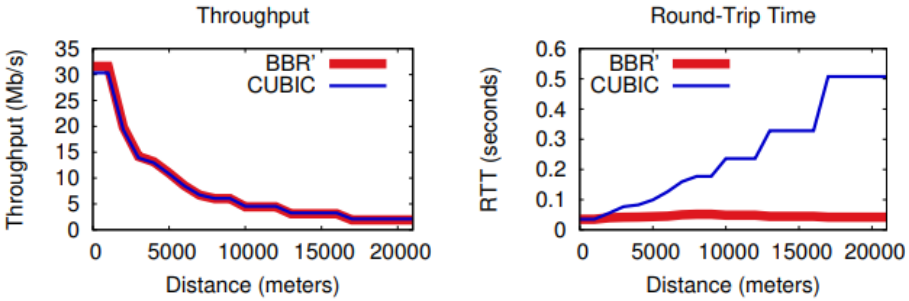


Figure 11: 4G LTE Network versus UE Distance

Figure 3.5: BBR 4G LTE Results (Claypool, 2018)

Two flows should share 50% whereas three flows would expect to share 33.33%. BBR claims it ”converges toward a fair share of the bottleneck bandwidth whether competing with other BBR flows or with loss-based congestion control” (Cardwell, 2016).

Tongyu Dai (2018) results lead to the findings that *”When the queue size is small, TCP is starved by BBR. The small queue size limits TCP to accumulate packets so that packet loss occurs quickly. Once TCP halves its congestion window in front of packet loss, BBR grabs remainded bandwidth to starve it. In contrast, when the queue size is large, TCP starves BBR. TCP is always trying to fulfill network queue until packet loss occurs. Even it halves its window size sometimes, the number of inflight packets are still much larger than BDP, resulting in starving BBR.”*

To better understand the fairness of BBR, the following two subsections discuss BBR

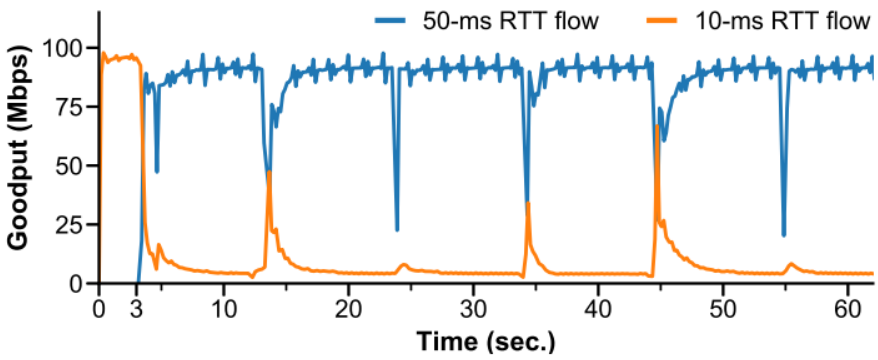


Figure 3.6: 50-ms RTT Average Goodput 87.3 Mbps, 10-ms RTT Average Goodput 6.3 Mbps (Shiyao, 2017)

fairness between BBR flows and again between BBR and other flows running alternative TCP implementations.

3.4.1 Intra protocol flows

Intra flow is the presence of competing flows baring the same TCP algorithm.
Shiyao Ma Et al. (2017) researched intra flow behavior of BBR and noted that ”a BBR flow with longer RTT dominates a competing flow with shorter RTT ... We blame it on BBR’s connivance at sending an excessive amount of data when probing bandwidth. Specifically, the amount of data is in proportion to RTT, making long RTT flows overwhelming short ones”. The paper shows goodput over a 100Mbps bottleneck occupied by two BBR flows with RTTs of 50 and 10 milliseconds. Supporting their statement, the 10ms flow averages only 6.3Mbps whereas the 50ms flow averages 87.3 Mbps (see figure 3.6). Further experiments, all of which use the dumbbell topology, see the 10ms flow against a 15ms and a 30ms flow separately, finding bandwidth utilisation of the 10ms flow at less than 25% and 10% respectively. Figure 6 of Shiyao Ma’s paper shows results of such an experiment with a varying number of flows.
The team discuss how this knowledge of precedence to longer RTT flows could be exploited. By artificially adding RTT latency a malicious receiving node could occupy bandwidth required for other flows. Such an exploit risks violating net neutrality ethics. This can be seen in figure 7 of the Shiyao Ma Et al. paper referenced. Such a precedence also highlights that BBR is inverse to efforts made in designing the internet as fast as possible, if the slowest connection or longest path receives more bandwidth, there is no need for shortest path routing algorithms.

Hock (2017) observes multiple BBR flows effect on RTT. Hock observers that ”the

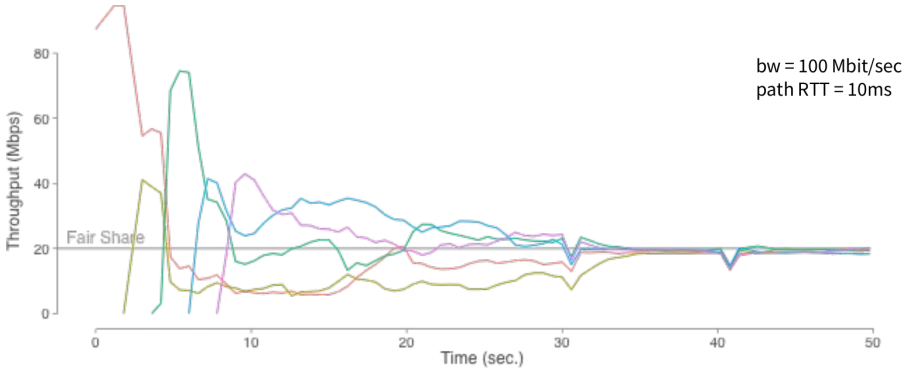


Figure 3.7: IETF 99 Update of BBR convergence (Cardwell, July 2017)

RTT doubles if (at least) two BBR flows with the same RTT min share a bottleneck". This is due to BBR's probeRTT phase. When predicting the minimum RTT BBR requires clearing of the buffer queue, however when other flows are present this can not be controlled, leading to an overestimation of the RTT and consequently the in flight cap, causing larger queueing delays per additional flow.

Google inc. (Cardwell, March 2016) published results of two competing BBR flows with short and long minimum RTT values. Supporting Shiyao Ma's paper, they also observed that "BBR flows w/ higher RTT have an advantage; but BBR flow with 64x higher minRTT only has ¡ 4x higher bw". This consolation highlights the precedence to higher RTT values and again we can observe violation of net neutrality.

The Google team later produce a throughput versus time graph (Cardwell, July 2017) showing that when multiple flows exist with the same RTT the convergence between multiple BBR flows leads to an even fairness for all flows (Figure 3.7). This again reinforces that bias exists where variant RTT values are present.

3.4.2 Inter protocol flows

Inter flow is the presence of competing flows baring different TCP algorithms.

Tongyu Dia Et al. (2018) research also covered inter flow competition between BBR and CUBIC. The research evaluates the fairness of two flows over a single fibre cable, with five different queue sizes. The group finds that in order for both flows to evenly share the bandwidth the queue size must be equal to 300 milliseconds. A value of less than 300 milliseconds (25ms,50ms,100ms) saw BBR take a majority of the bandwidth, whilst values above (1000ms) saw CUBIC take about 70% of the bandwidth.

Hock (2017) finds similar results to Tongyu, when the buffer is large (large enough to create queue sizes close to the 1000ms previously mentioned) CUBIC will occupy the bandwidth bottleneck considerably more than BBR (see Hock's figure 15). When the buffer is small BBR takes a majority of the bandwidth whilst CUBIC occupies around 25%.

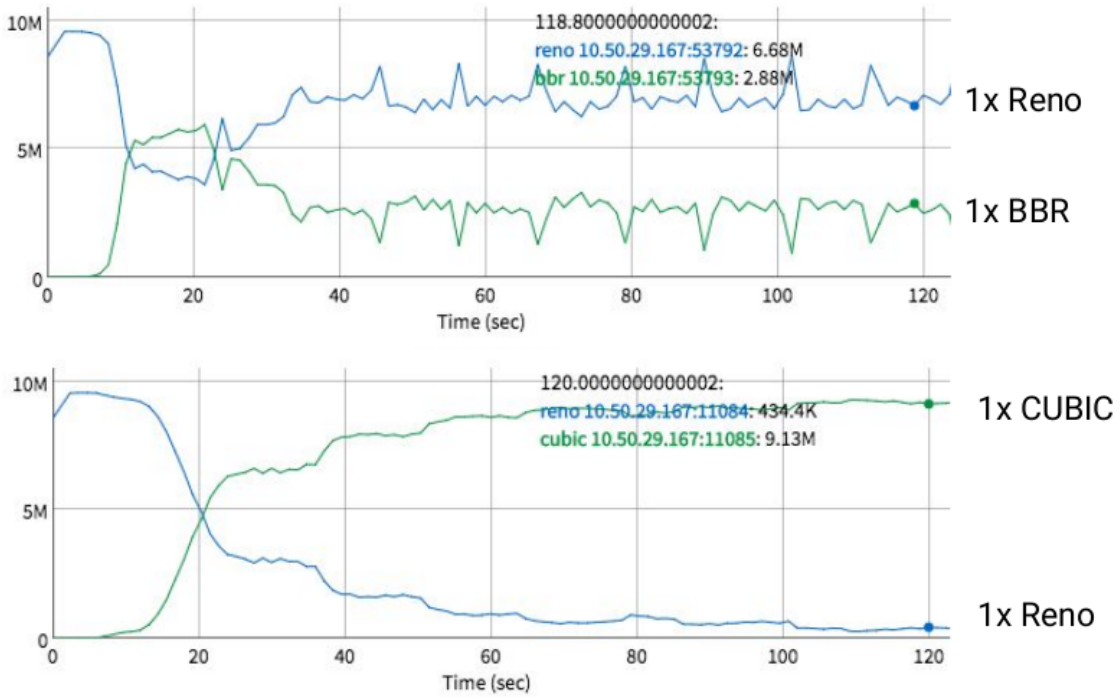


Figure 3.8: Slide 16 - Throughput vs Time (Cardwell, 2017)

Google Inc. after developing BBR published preliminary findings of the algorithm in comparison to CUBIC (Cardwell, 2016). The team at Google found that with 8 concurrent flows, CUBIC experienced increased latency when buffer size increased whereas BBR was unaffected. They found that when sharing large buffers with a CUBIC flow, BBR initially could not compete with CUBIC however BBR recovers after about 40 seconds and proceeds to fairly compete with CUBIC.

When BBR is deployed for WAN TCP traffic at Google the team observed the following:- 2% lower search latency on google.com, 13% larger Mean Time Between Rebuffers on YouTube, 32% lower RTT on YouTube, Loss rate increased from 1% to 2% (Cardwell, 2017).

The Google team also published a number of comparisons between BBR, CUBIC and Reno with different flow quantities (figures 3.8, 3.9 and 3.10). Keeping the environment the same throughout:- 10Mbps bandwidth, 40ms RTT, 1MB buffer (Cardwell, 2017). The results show that, with the exception of 1 BBR versus 1 Reno (figure3.8 top), both CUBIC and BBR out perform Reno regardless of the number of Reno flows present. These figures highlight the struggle for fairness that exists between inter protocol flows, even two loss based flows (figure3.8 bottom) do not fairly negotiate bandwidth.

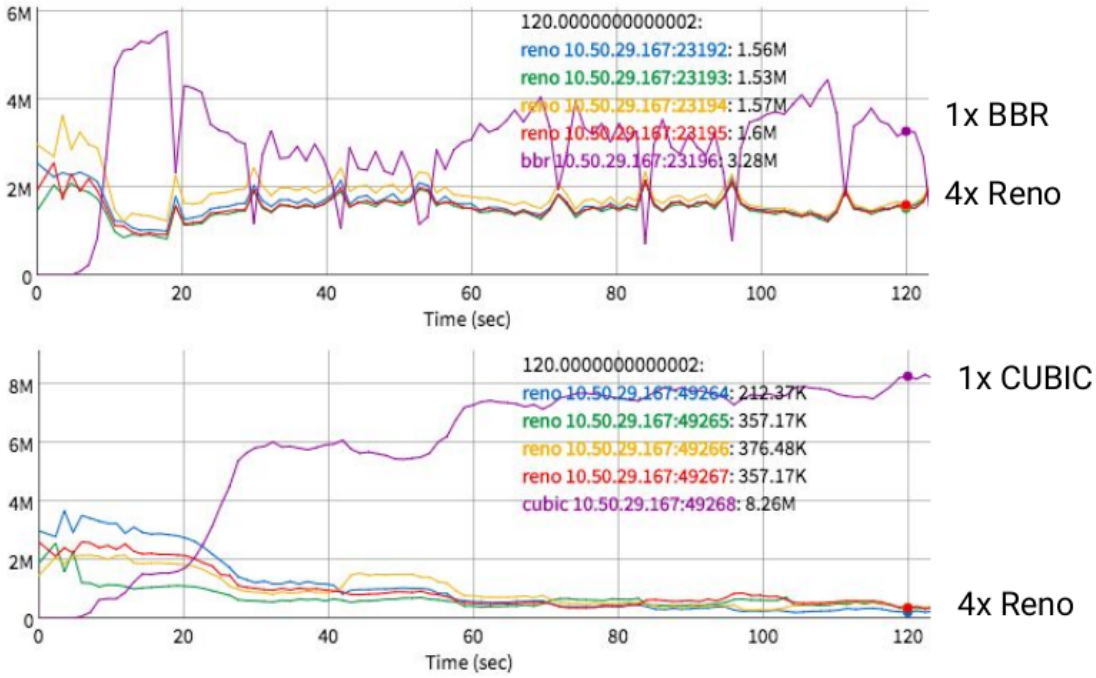


Figure 3.9: Slide 17 - Throughput vs Time (Cardwell, 2017)

3.5 Reaction to network changes

Changes to network paths happen for a number of reasons:- A path is no longer accessible, the bandwidth allowance has changed due to other flows joining or leaving the network, a new shorter path is discovered, or in the case of mobile networks the UE (User Equipment) experiences handover and a new path is forged.

Dominik Scholz (2018) researched BBR's ability to adapt to changing network environments. Figure 3.11 shows a single BBR flow with an RTT of 40 ms and 10 Mbit/s bottleneck, under changing network conditions. This figure shows that sending rate, RTT and inflight values align with the bottleneck bandwidth, RTprog and BDP respectively. ProbeRTT phases can be seen every 10 seconds, starting in the first second. When the bottleneck bandwidth is doubled BBR is able to adapt the sending rate without building a significant queue, as noted by the very minimal RTT increase. After the bandwidth is reduced to its original rate, the in flight data is still delivered at the old bandwidth rate and a queue forms, increasing the RTT estimation significantly until the queue is drained in the following five seconds. BBR demonstrates its ability to recover from buffer bloat and continues to match the expected sending rate, RTT and in flight values.

When the RTprop value in tripled at 56 seconds BBR cannot adapt immediately. As stated the ProbeRTT phase occurs on the first second every ten seconds - resulting in ProbeRTT at 51 seconds and again at 61 seconds. Since the tripling of RTT occurs half way between these times (56 seconds) BBR cannot immediately detect the change. When RTprop increases, ACK packets require more time to arrive, increasing the inflight

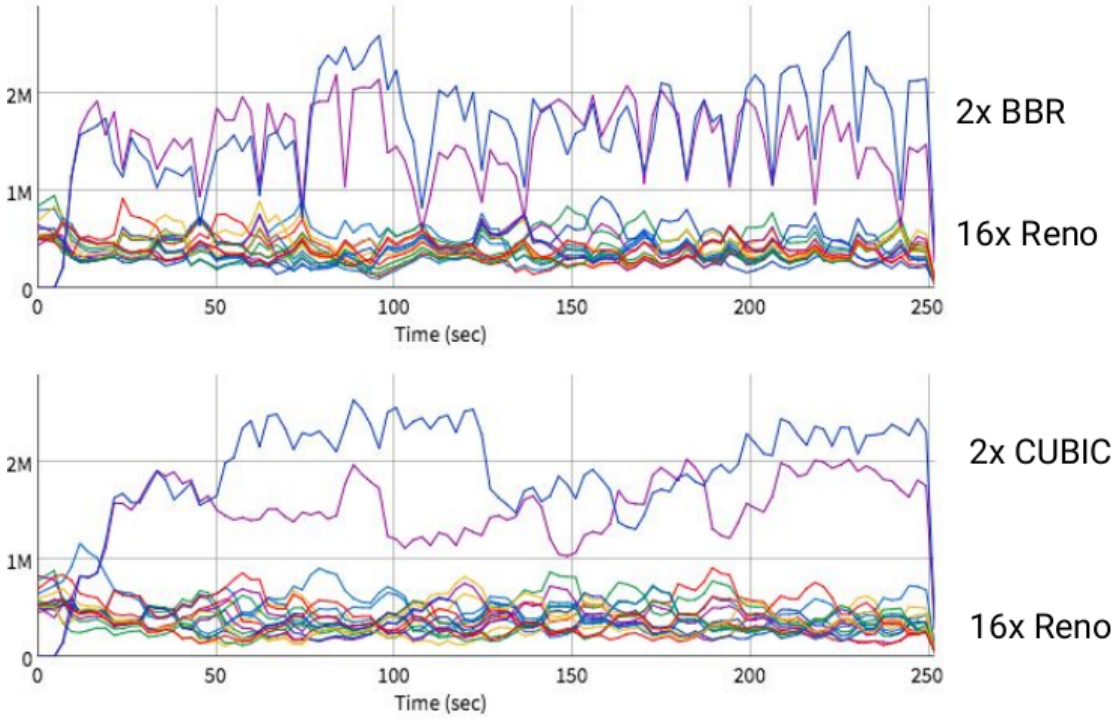


Figure 3.10: Slide 18 - Throughput vs Time (Cardwell, 2017)

data until the congestion window is saturated. BBR limits its sending rate until the BtlBw estimate expires, a new lower BDP is calculated and the congestion window is reduced. After both the probeBW and probeRTT are complete BBR learns about the new values and increases the sending rate again to match available bandwidth. When the RTT is then decreased at 76 seconds, to its original value, BBR responds immediately matching the inflight data to the BDP.

3.5.1 New flows joining the network

As an internet user can expect multiple connections simultaneously. To truly test fairness more realistic experiments should include multiple flows, preferably starting at different times.

Hock experiments with multiple inter protocol flows, two BBR and two CUBIC flows, using a small buffer. The experiment starts two flows, BBR and CUBIC, at the same time, with results similar to Hock's one BBR verses one CUBIC flow. However when the second BBR flow is established after 98 seconds, the two BBR flows quickly reach respectable fairness but the CUBIC flow drops to close to zero. The introduction of another CUBIC flow at 123 seconds does not effect the low CUBIC throughput. Hock repeats the experiment on both 1Gbps and 10Gbps connections and receives the same results.

Cardwell (2016) investigates the convergence of multiple BBR flows joining a network at different intervals. These results (figure 3.12) show that BBR flows can converge to a

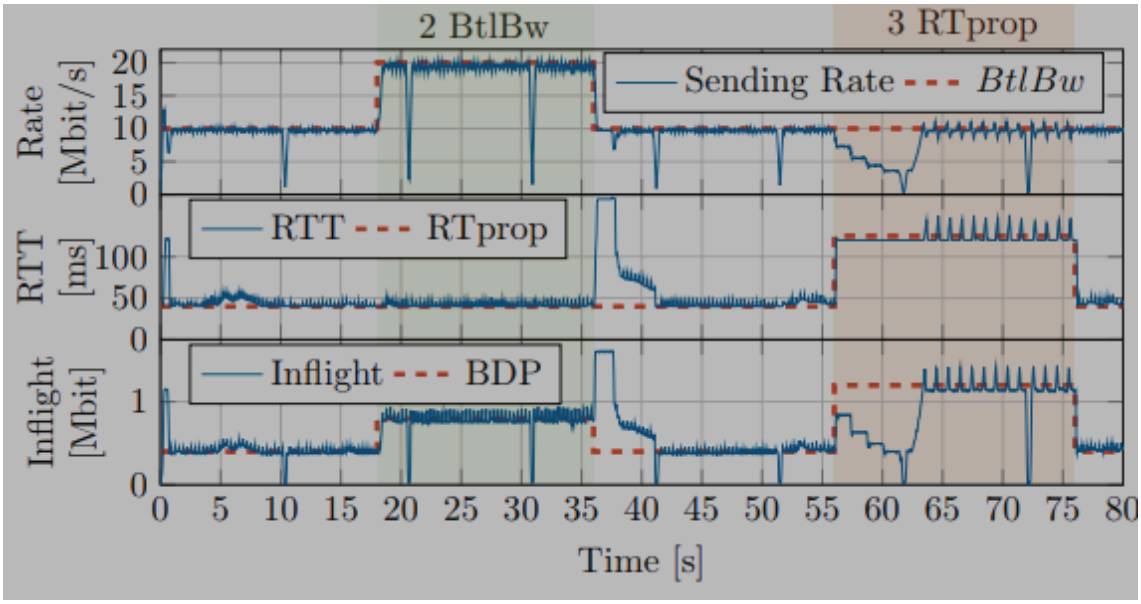


Figure 3.11: BBR adapting to network changes (Scholz, 2017)

fair share by relinquishing bandwidth to smaller flows. The relinquish and subsequent convergence of flows is a results of BBR gain cycling. Gain cycling occurs during the probeBW phase and is designed to vary the pacing gain, reducing it to allow smaller flows to compete and enlarging it to occupy the available bandwidth.

Scholz (2017) also conducts experiments into the effect of additional flows, with the same RTT time, joining the network at different time periods. Scholz deduces that while BBR flows can synchronise, the time it takes to achieve equal fairness is heavily reliant on the timing of flows already present (figure 3.13.A). This figure displays results when four synchronised BBR flows are joined by an additional BBR flow. This show the optimal timing for a joining flow is during the probeRTT phase. As existing flows have drained the queue the new flow can correctly measure the RTT alongside the existing flows and thus synchronise. Similar to behaviours seen in the inter flow fairness section 3.4.2, one flow can dominate others when its BDP is too high due to incorrect/outdated RTT value. Figure 3.13.B shows that the optimal flow arrival timing must factor the startup phase of the new flow. By joining at 1.7 seconds before the next probeRTT the new flow undertakes startup whilst the existing flows drain ready for probeRTT. As a results the new flow overestimates the bottleneck and suppresses the existing flows. Given an extra 0.3 seconds (2 seconds before probeRTT) the new flow can complete startup and drain a majority of its queue for the existing flows to measure a more representable RTT, syncing the flows but slightly overestimating the bottleneck. In contrast joining at 1.7 seconds before probeRTT sees fairness stabilising above 95% only after 22 seconds, whereas 2 seconds before improves this values to only 10 seconds. Scholz remarks that ”Considering the prevalence of short-lived flows in the Internet, this high value (22

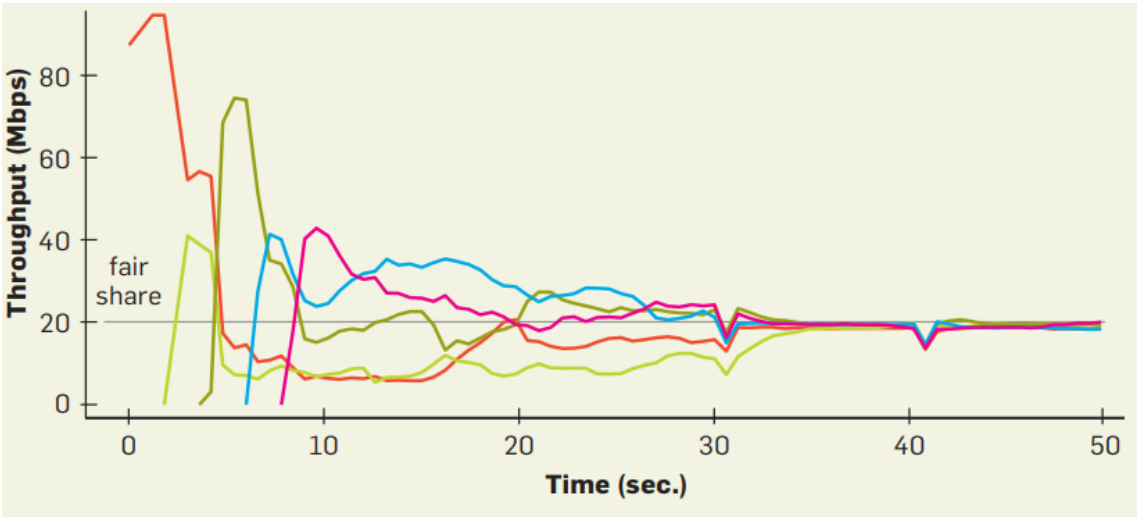
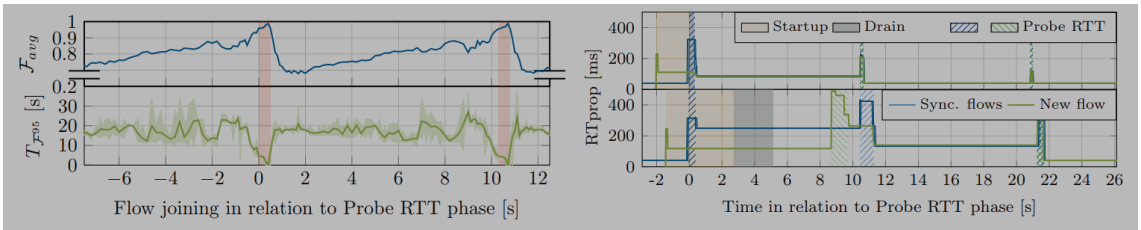


Figure 3.12: BBR adapting to new flows (Cardwell, 2016)



(a) Join during different times of the Probe RTT cycle. Red area marks Probe RTT phases.

(b) Correlation between Startup/Drain and Probe RTT for joining 2 s and 1.7 s before next Probe RTT phase

Figure 3.13: BBR adapting to new flows in relation to Probe RTT of existing flows (Scholz, 2017)

seconds) poses a significant disadvantage of TCP BBR".

Figures 3.14.A and 3.14.B supports the findings Scholz demonstrates above. With a flow (A) joining during the probeRTT phase we see convergence occurring significantly faster, reaching 100% fairness after 55 seconds, then a flow joining immediately after probeRTT (B).

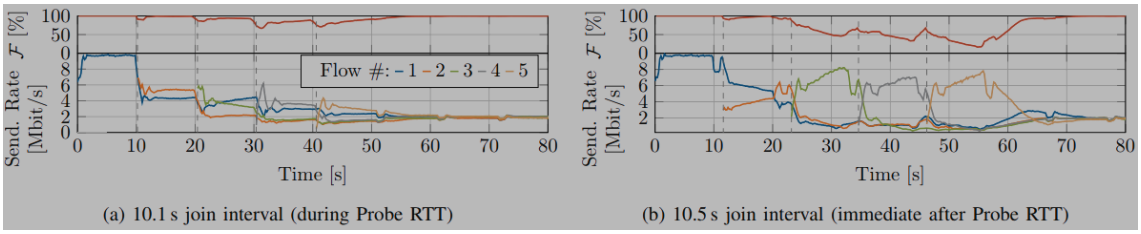


Figure 3.14: BBR adapting to new flows during and after Probe RTT of existing flows (Scholz, 2017)

Chapter 4

Methodology

4.1 Development Methodology

4.1.1 Available Methodologies

Many design methodologies exist that dictate the interaction between developers, users, contractors and the software its self. As this is a research project, not all of the above entities exist and thus a methodology would need to focus on the aims set forward by the project rather than communication between entities.
Methodologies such as agile and Dynamic Systems Development Methodology (DSDM) allow for the completion of tasks based on a priority (such as MoSCoW). Whilst such a priority system could be implemented, it would not suit the investigative nature of the research as it is not yet clear the potential of BBR algorithm. As this is an individual project, methodologies that require greater than one person are not an option (such as eXtreme Programming or Joint Application Development).
Linear methodologies, such as waterfall, are not suited to this research project. Whilst many objectives are similar and could be conducted in parallel, benefits to tackling one objective at a time are clearly present:- Improvements to the use of the software as knowledge increases, continuous time management on completion of every objective and the acknowledgement of meeting requirements before the project is close to an end.
Rapid Application Development (RAD) is an agile software development methodology. RAD undertakes an iterative approach to building a software solution. By specifying the deconstructed objectives upfront with clear time frames, (Prashanth Kumarl and Prashanth, 2014) developers are encouraged to start development promptly. Such a solution is ideal for this project, having already defined the deconstructed objectives in the form of aims (see section 1.2.1). However such a methodology risks poorly defined objectives impacting the whole project. This risk is mitigated due to the isolation of objectives into logical groupings and continuous discussions with the project supervisor.

4.1.2 Chosen Methodology

RAD has been chosen for this project due to the solo element of the programming and the time frame to create artefact. It is crucial that the project is completed as the time frame can not be manipulated. Using the aims, focus and effort can be applied based on the remaining work, complexity and remaining time. As stated the objectives are similar, that is they all run experiments against the simulator before evaluating BBR, and would be conducted iteratively.

4.2 Development Software

BBR research can be conducted using a variety of methods. Other research into BBR includes using Mininet (Scholz, 2018) or running experiments with physical hardware (Tongyu Dai, 2018). This research evaluates BBR using NS-3 Network Simulator V3 (NS-3, 2018).
NS-3 is well known simulator in academics, it is open source and written in c++ and python. Whilst simulators are not fully representable of real life, Claypool et al. believe that ”generally, the performance results for BBR and BBR’ look quite similar, with nearly the same round-trip times, bandwidths and bytes inflight”. Figures 4,5 and 6 of his paper support this statement through comparisons of BBR and BBR’.
Using version 3.27 of NS-3 ensures compatibility with BBR’ module written and released by Claypool (Claypool, 2018). This version is also compatible with a CUBIC module (Levasseur, 2014) allowing for algorithm comparison. Choosing an older version of NS-3 provides protection against release bugs, ensures documentation exists and allows compilation on a stable operating system. This research uses Ubunutu 16.04 LTS, again an older version, which is open source and considered highly stable.
Both modules were then evaluated to ensure they were patched into NS-3 correctly. This was done in the case of BBR through validation with Claypool’s BBR for NS-3 results (claypool, 2018). For CUBIC, a python script is supplied to validate the patch.

Chapter 5

Analysis and Design

5.1 Topologies

The network topology designs can be seen in figures 5.1 to 5.7. Each design details the network topology (nodes n0 to nX), bottleneck link and direction of flow(s).

5.1.1 Line Topology

Figure 5.1 is set up to test BBR's performance and compare against other algorithms. This topology sees computers connected over a single hop, simulating a communication of two machines over LAN connection with n1 acting as a router. This topology allows for initial comparison of BBR to other algorithms and validation against Claypool (2018).

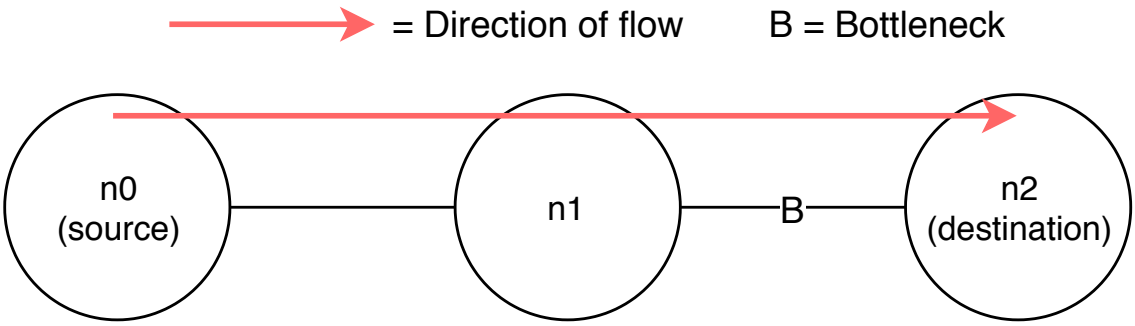


Figure 5.1: Line topology (3 nodes)

5.1.2 Modified Line Topology

Figure 5.2 topology allows competing flows to occupy the bottleneck by the addition of node n3. Conducting a secondary flow on an alternative node will test BBR's ability to

sync with other BBR flows. Due to an ns3 limitation, additional nodes are also required to modify attributes such as bandwidth and delay on each flow individually. As research has already indicated, BBR does not share the bandwidth bottleneck fairly. This topology will continue this research to evaluate BBR's fairness.

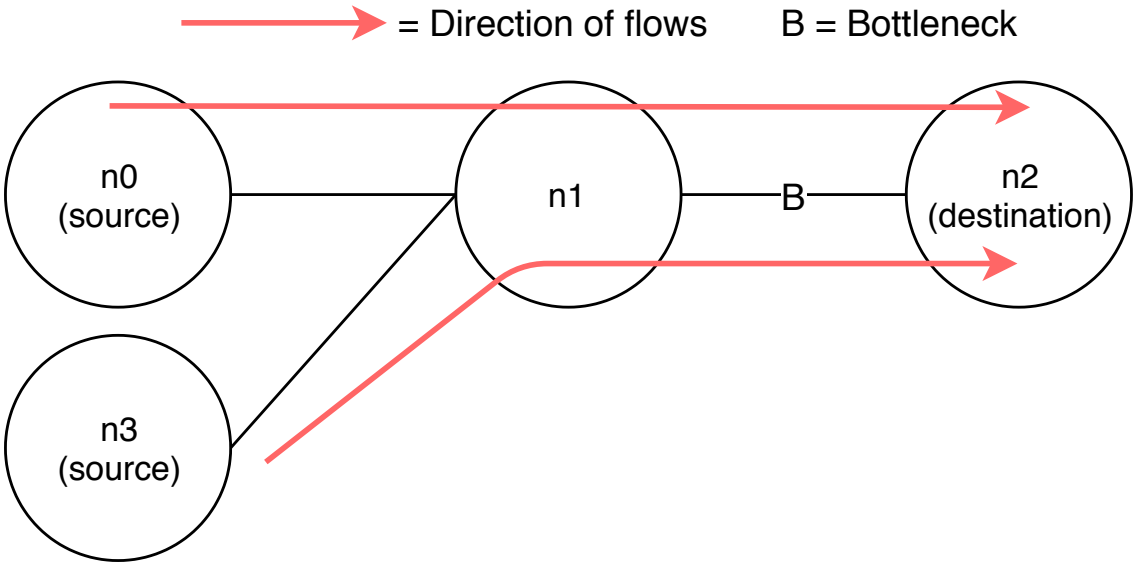


Figure 5.2: 2 competing flows - modified line topology

5.1.3 2 by 2 Horizontal Dumbbell

Figure 5.3 features a 2 by 2 dumbbell topology. The dumbbell topology is commonly used (Scholz, 2018) (Hock, 2017) as closely resembles real world network toplogies, provides one communal bottleneck and is highly scalable. This setup allows for one or more flows to exist on a path:- multiple flows can compete within the same path, between different paths or both.

5.1.4 2 by 2 Vertical Dumbbell

Figure 5.4 features a modified dumbbell topology with flows travelling top down. Similar to 5.3 this topology, investigates BBR's behaviour to bidirectional flows over the bottleneck. Whilst not often tested, the effect of ACK clock and possible ACK suppression may present alternative findings.

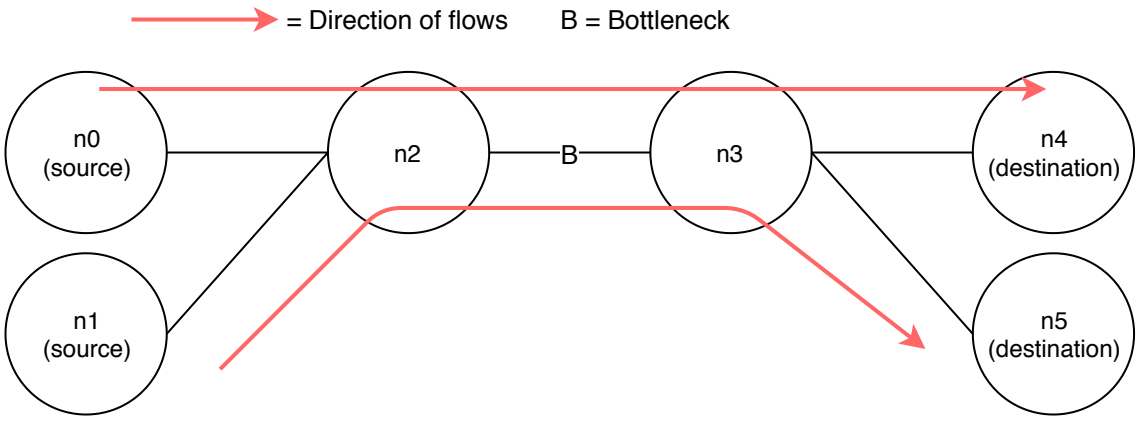


Figure 5.3: 2x2 horizontal dumbbell topology

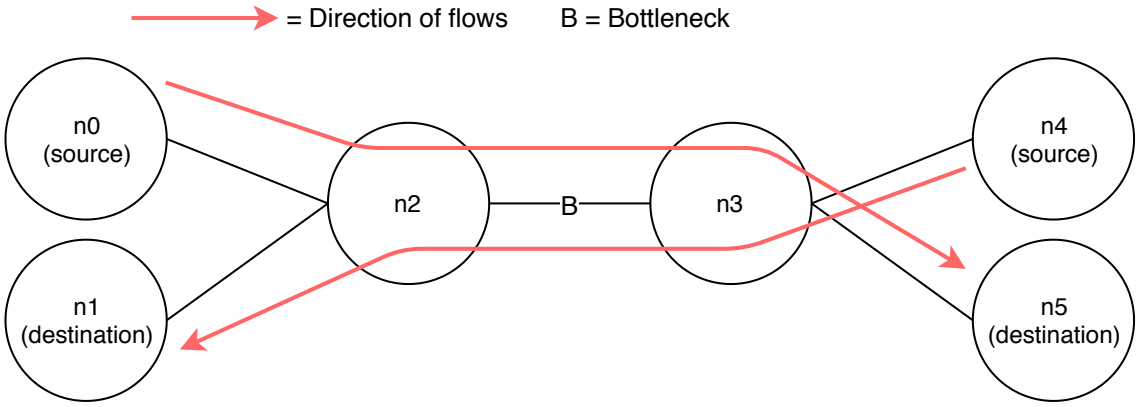


Figure 5.4: 2x2 vertical dumbbell topology

5.1.5 3 by 3 Horizontal Dumbbell

Figure 5.5 similar to 5.3 features a dumbbell topology. This topology allows for 3 alternative algorithms to compete over paths with one-to-one or one-to-many relationship, all likely scenarios given that different operating systems run different algorithms and flows of varying implementations can originate from any one of the nodes. In this topology multiple algorithms can be seen competing and multiple algorithm implementation types (loss based, delay based and congestion based) observed, unveiling possible bias towards homogeneous implementation types.

5.1.6 Line Topology - One New Flow

As stated in 3.5.1 BBR does not behave as desired when subsequent flows arrive. This topology allows the investigation of delayed flows effect on BBR by introducing a flow on n2, after a given time, before the bottleneck. Using this 4 node line allows comparison both before (n1) and after (n3) introduction of the new flow(s).

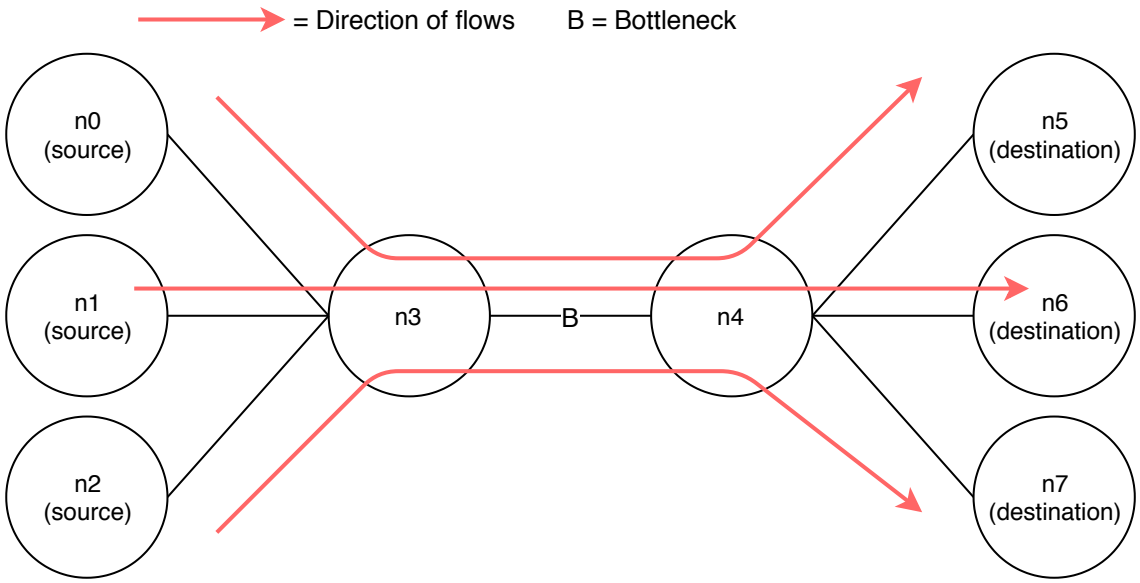


Figure 5.5: 3x3 horizontal dumbbell topology

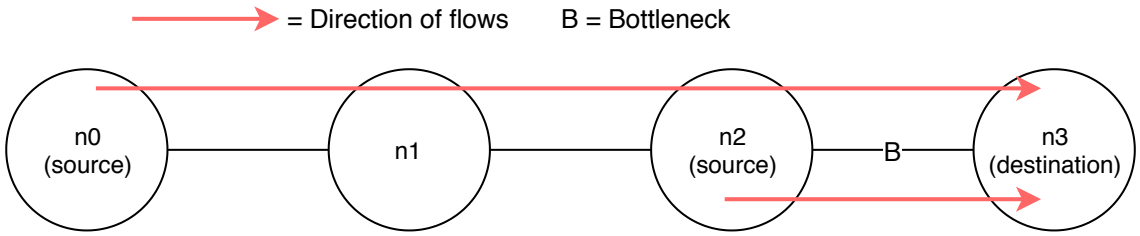


Figure 5.6: Introducing a new flow

5.1.7 Line Topology - Two New Flows

Continuing the research from 5.1.6 this topology evaluates BBR's performance when many flows arrive. By using two separate nodes to deliver additional flows this topology resembles scenarios common in real world networks.

5.2 Coding

Project related code can be found in the appendix and artefact. Code is written in C++ (NS-3), python (parsing) and plain text (GNUPLOT scripting). When writing the code the topology designs above were followed.
For every simulation a folder is created containing the simulation code, results per flow, results in graph form, the python script to parse this per flow data and the GNUPLOT

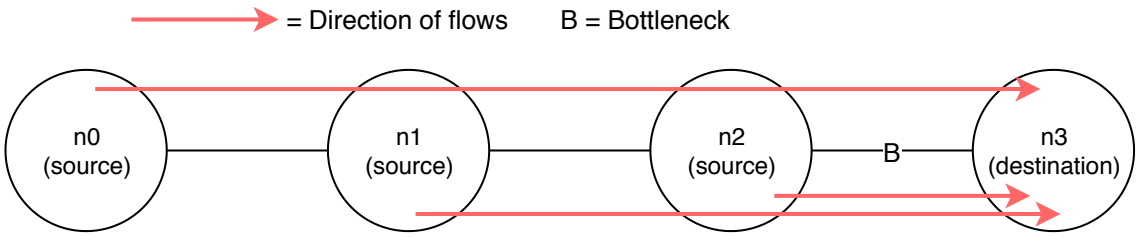


Figure 5.7: Introducing multiple new flows

script to generate said graph. This structure allows future users full access to code enabling modification and execution of individual simulations and the generation of results at whim, ensuring replicability. This logical grouping also allows for additional simulations to be created without increased complexity - simply create another topology, add an accompanying folder and start coding.

5.2.1 Simulation Coding

For each script an ascii interpretation of the design was created in the header comment, showing the network topology, bottleneck path, IP addresses and subnets. Constants for the simulation are declared foremost, such as available bandwidth, path delay, router queue size, packet size, output booleans and TCP socket implementation. TCP implementations can be changed by adjusting the commented lines found at the bottom of the constants list. Unfortunately the software is not able to run multiple TCP versions on one script. This is a limitation of the software.
Every script contains a main function that builds the simulation environment and runs the simulation as per NS-3 guidelines (NS-3, 2019). For each script the following ordered processes must occur:- creation of nodes, setup connections between nodes, configuration of connections, allocation of IP addresses, installation of applications, optional output file specification and finally simulation start/stop/destroy commands. Every script is commented to distinguish each process.

5.2.2 Results Coding

Python was used to interpret the output from the simulation - a custom tracefile format (.tr) that can be treated as a CSV file when delimited by empty spaces rather than commas. Custom written Python scripts filtered NS-3 produced outputs to be used for GNUPLOT scripts. The python scripts make the following assumptions:- Only packets marked with 'r' (denoting a received packet (NS-3, 2019)) are parsed, packets not equal to "PACKET_SIZE" constant set in the NS-3 script are ignored, duplicate packets are ignored and

packets sent outside of the specified simulation duration time are ignored, the user knows the IP and port numbers needed to parse. An instance of this script can be found in the appendix.
Using GNUPLOT scripts, parameters of the graph are scripted including:- title, axis labels, axis ranges, output file and data to be plotted. GNUPLOT scripts consume data per flow, before combining into one graph. GNUPLOT scripts are written using GNUPLOT syntax and executed over command line to produce scalable vector graphics (.svg) graph files.

Chapter 6

Results

All processed simulation results in the form of graphs supplied in this chapter and in the artefact. All unprocessed results, in the form of traceroute files, can be found in the artefact submission.
The comparisons below are between BBR and two loss based algorithms:- CUBIC and NewReno. NewReno, an improvement over Reno, is highly stable TCP implmentation. CUBIC is amongst the most popular, ”46.92%” of web servers run CUBIC (Mudassar Ahmad, 2018) and therefore a logical comparable TCP candidate.
In this chapter, references to throughput signifies enqueued networks packets multiplied by the packet size over time represented in Megabits per second. Fairness refers to the acquisitions of equal throughput over the duration of the simulation, given the number of flows and bottleneck bandwidth.

6.1 Replication of Results

6.1.1 BBR - Claypool

This topology is focused on replication of results from other works to verify the simulator. Figure 6.1 shows replication of Claypool (Claypool, 2018). The simulation acquires the same throughput as Claypool records. From the figure can also be inferred the nature of BBR, seen at the start BBR experiences exponential growth until the first congestion event and subsequent drain at around 0.5 seconds. Achieving 18/20Mbps BBR successfully saturated the bottleneck at Kleinrock’s optimal point.

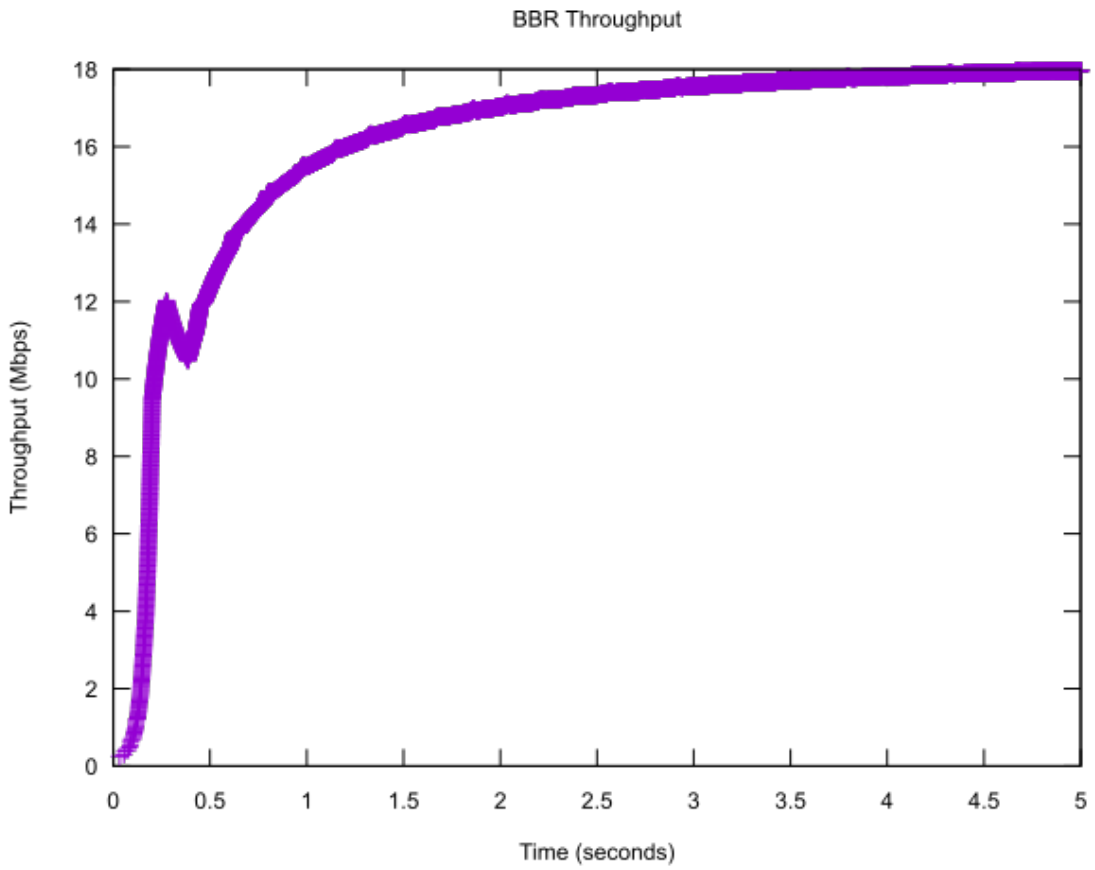


Figure 6.1: Claypool Replication Results

6.1.2 BBR - Shiyao

Figure 6.2 shows replication of Shiyao (Shiyao, 2017). Whilst Shiyao runs this simulation at a factor of 10 larger, the results demonstrate the same findings.

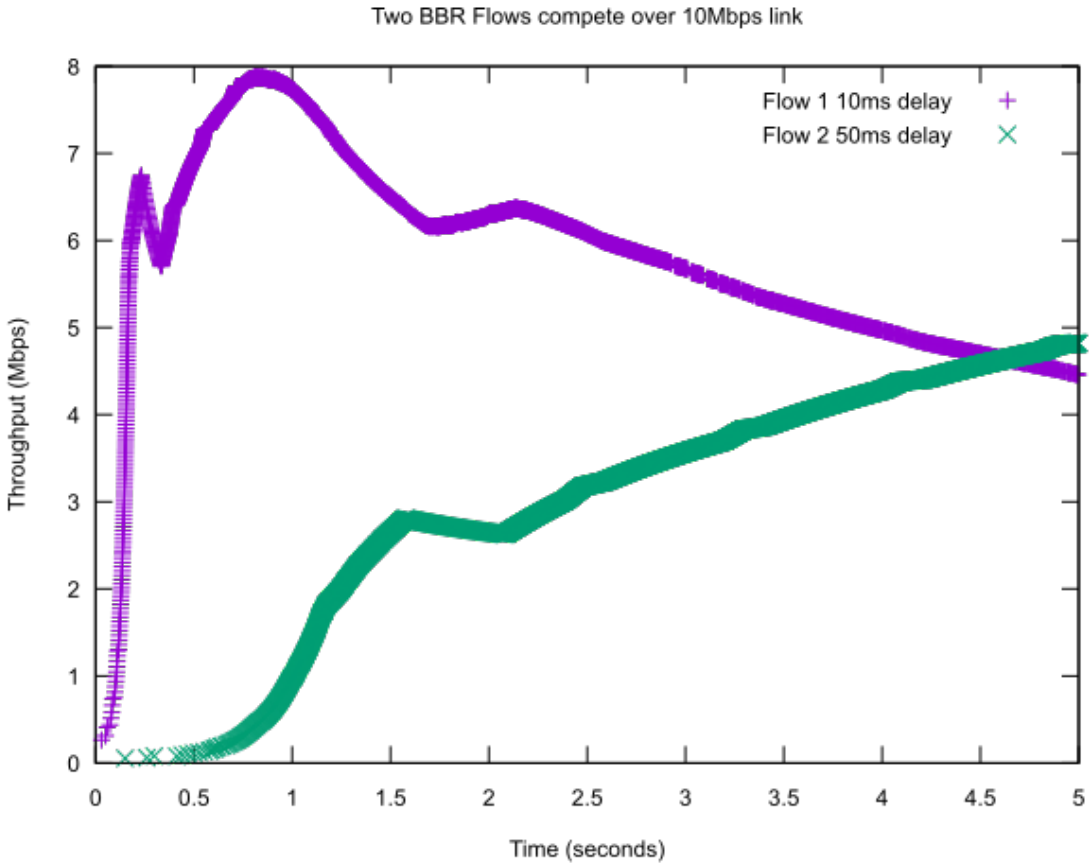


Figure 6.2: Shiya Replication Results

6.1.3 Cubic and NewReno

In contrast the replication simulations are compared to the same simulations using CUBIC (figure 6.3) and NewReno (figure 6.4) TCP algorithms. For Claypool's replication, both CUBIC and NewReno show higher throughput by around 0.2Mbps or 1%.
When compared to Shiyao's results for BBR, BBR shows fairness equilibrium at 4.6 seconds whereas neither CUBIC (figure 6.5) or NewReno (figure 6.6) achieve fairness.

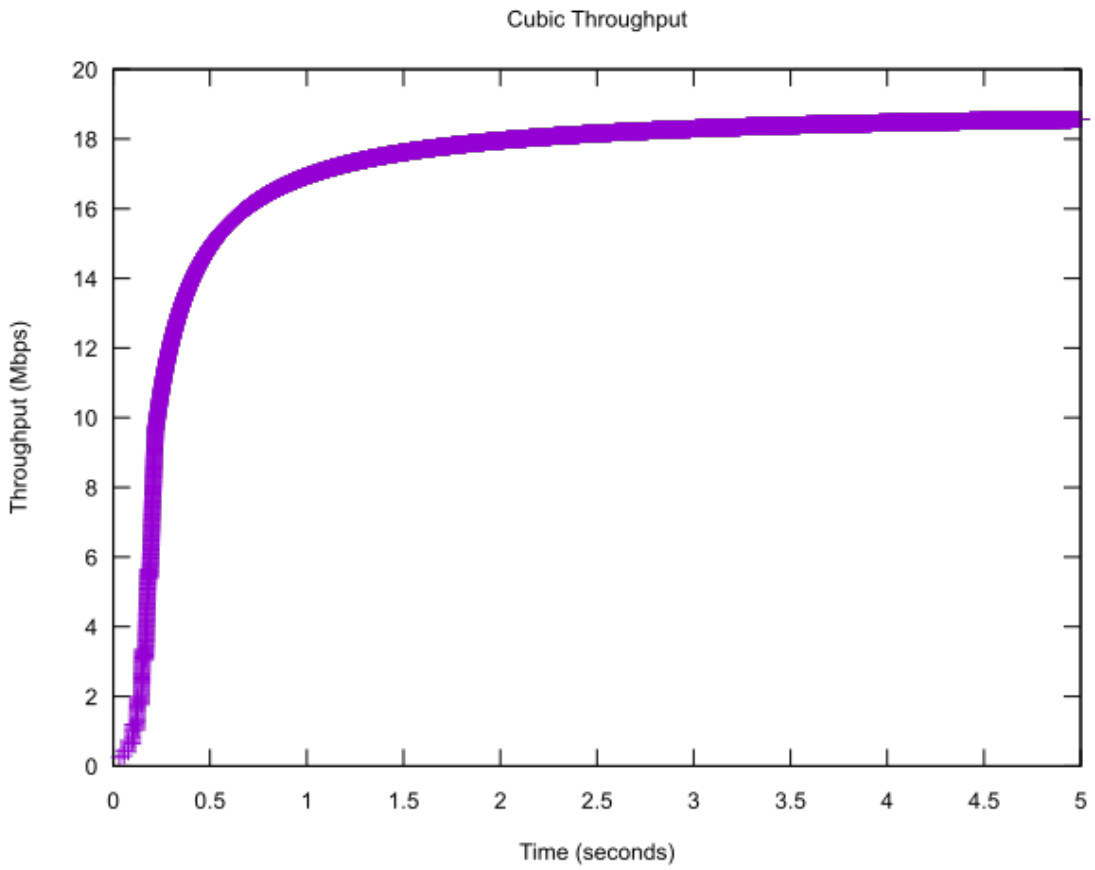


Figure 6.3: Claypool Replication Results CUBIC

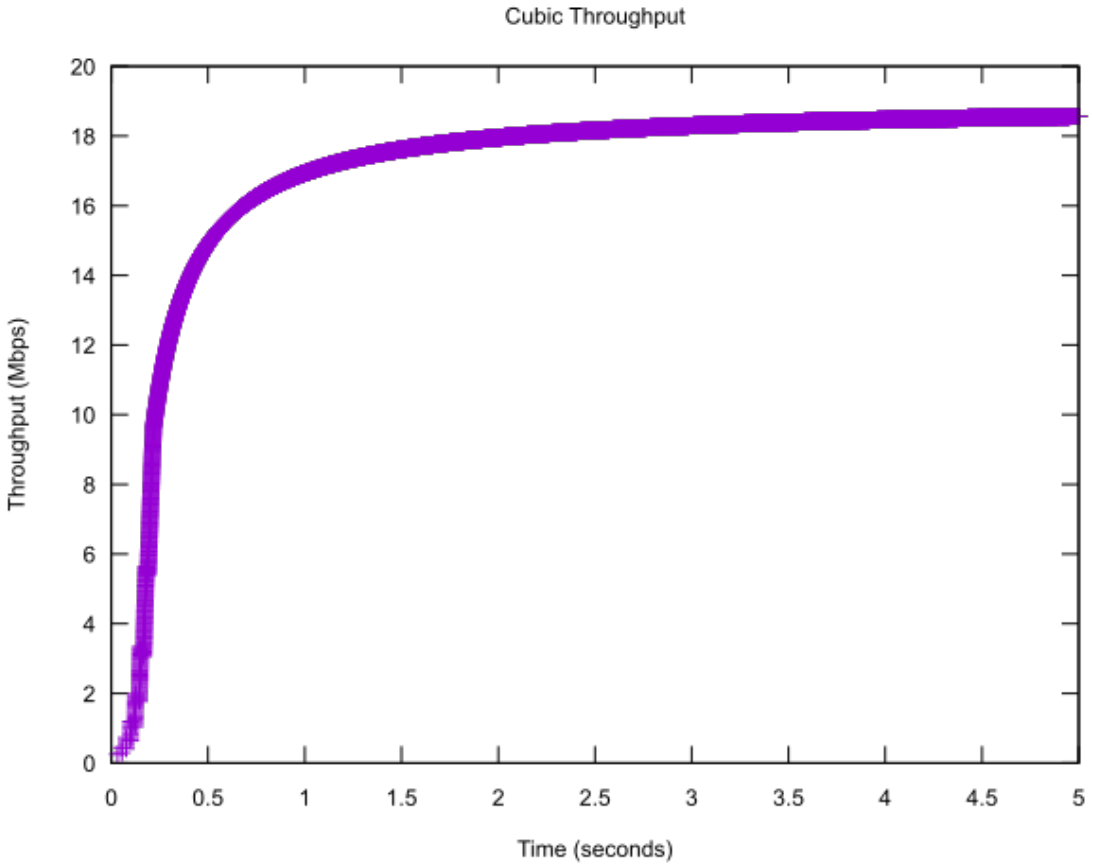


Figure 6.4: Claypool Replication Results NewReno

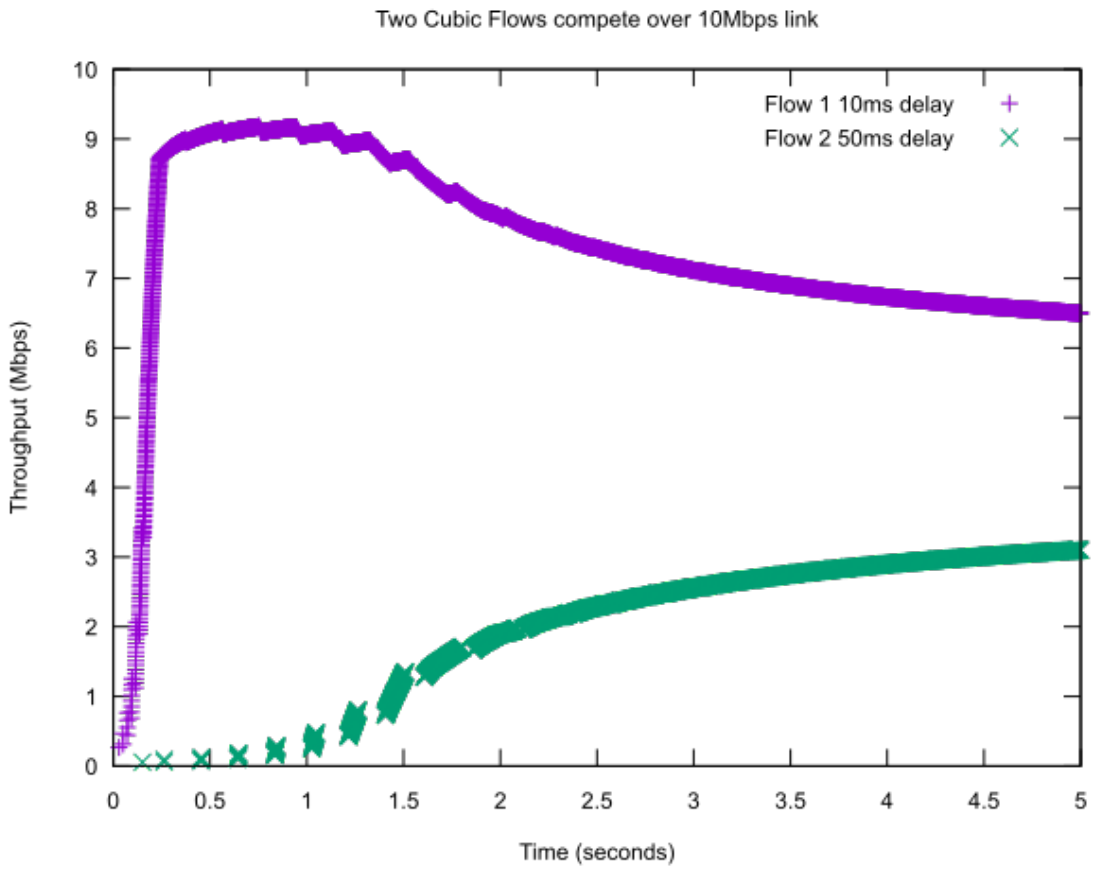


Figure 6.5: Shiya Replication Results CUBIC

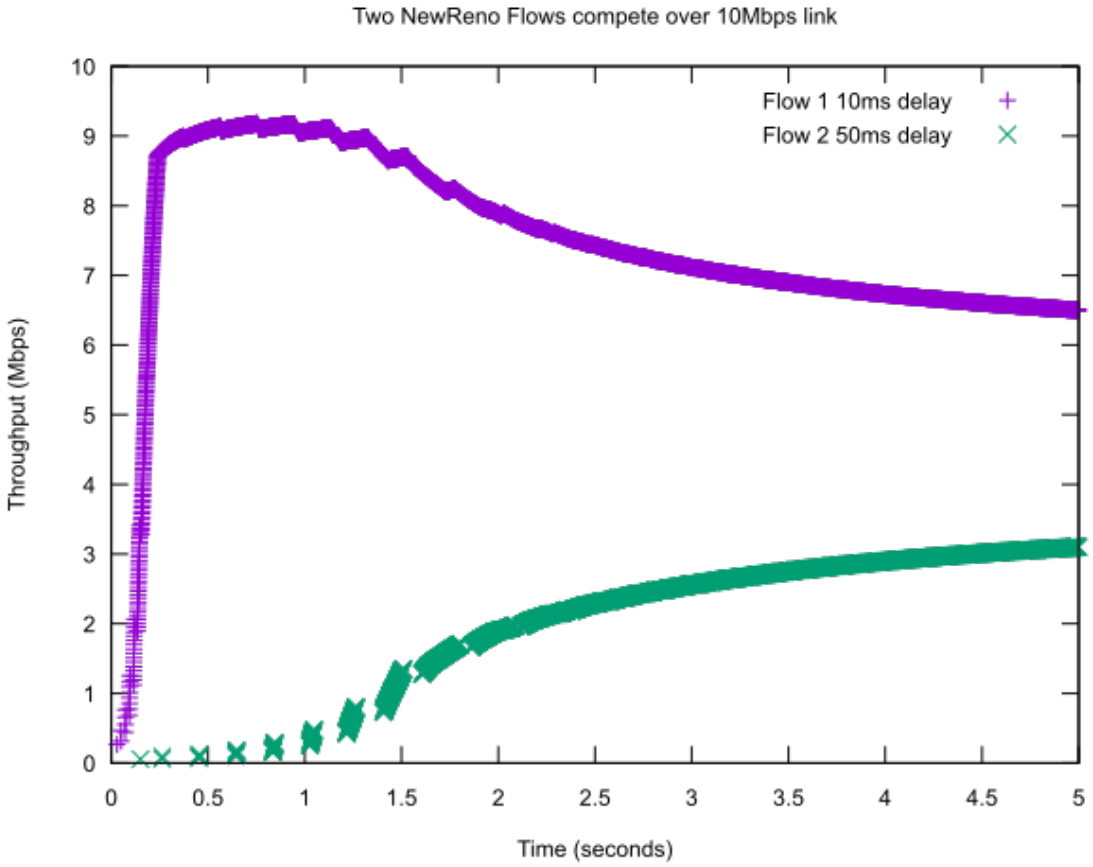


Figure 6.6: Shiya Replication Results NewReno

6.2 Bandwidth

Figures 6.12 and 6.13 show that BBR is able to saturate a bottleneck with multiple flows at or just below Klienrock's optimal point within 1.5 seconds of joining the network. Having probed at 1.2 seconds and with 3.8 further seconds to undertake both probe states again, it is clear from both figures that the BBR flows would plateau for the duration of the simulation.

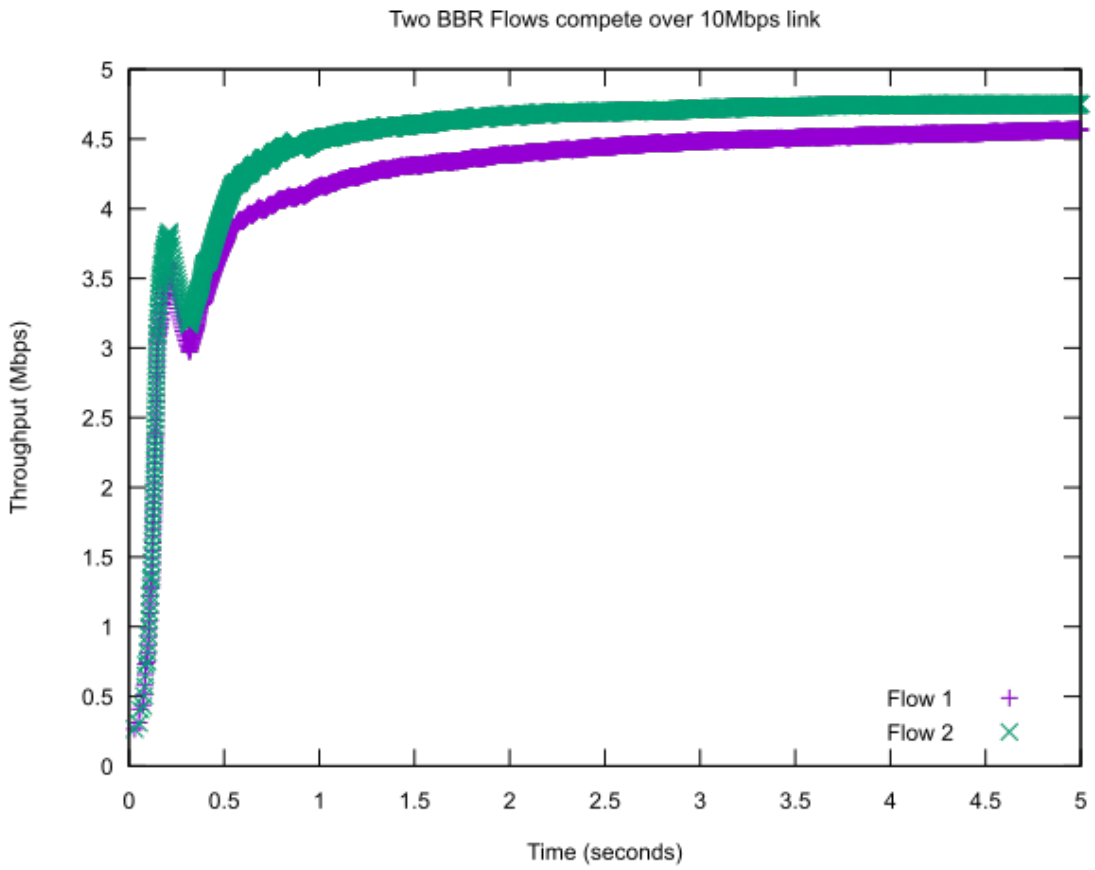


Figure 6.7: Results for One vs One Horizontal Dumbbell over BBR

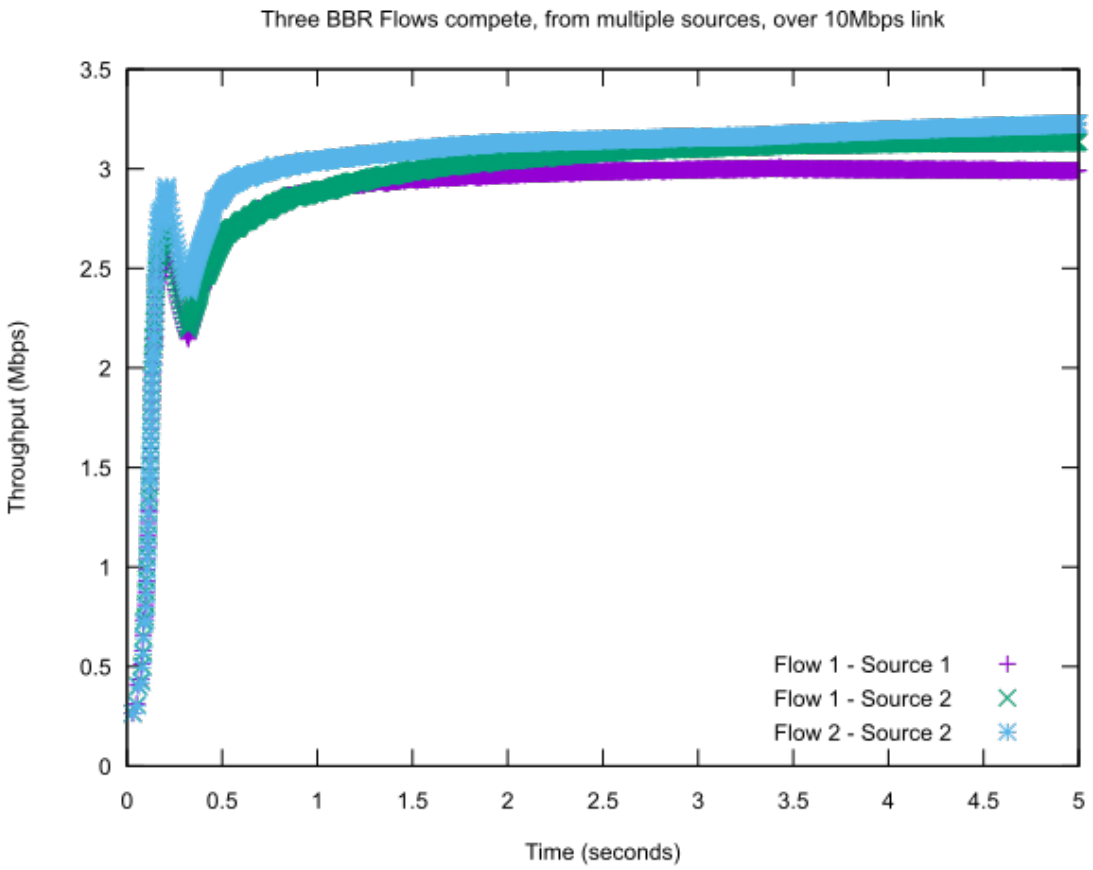


Figure 6.8: Results for One vs Two Horizontal Dumbbell over BBR

The third topology used highlights BBRs ability over a dumbbell topology. Figure 6.9 shows BBR undertaking probing RTT in frequent intervals, noted by the small bandwidth drops lasting around 200ms. Probe RTT happens after probing bandwidth, hence the shallow peak trough pattern. Figure 6.10 shows the same throughput recorded for CUBIC in comparison. CUBIC achieves the same throughput but unlike BBR, also achieves full fairness with a Jain's fairness index value of 1.

In comparison over the 5 seconds of simulation on vertical dumbbell at 10Mbps bottleneck, two flows of BBR transport 84.833Mb (out of the possible 100Mb) whereas two flows of CUBIC traffic achieve 88Mbps total throughput. The remaining 15.2Mb is possibly outstanding as buffered packets in the receivers client or packets still in-flight, packets that were not marked by the simulation as delivered and thus not recorded. This is due in part by how BBR calculated bottleneck bandwidth, using the in-flight transmission medium and buffer as part of the path to maintain full path saturation and thus constant delivery rate.

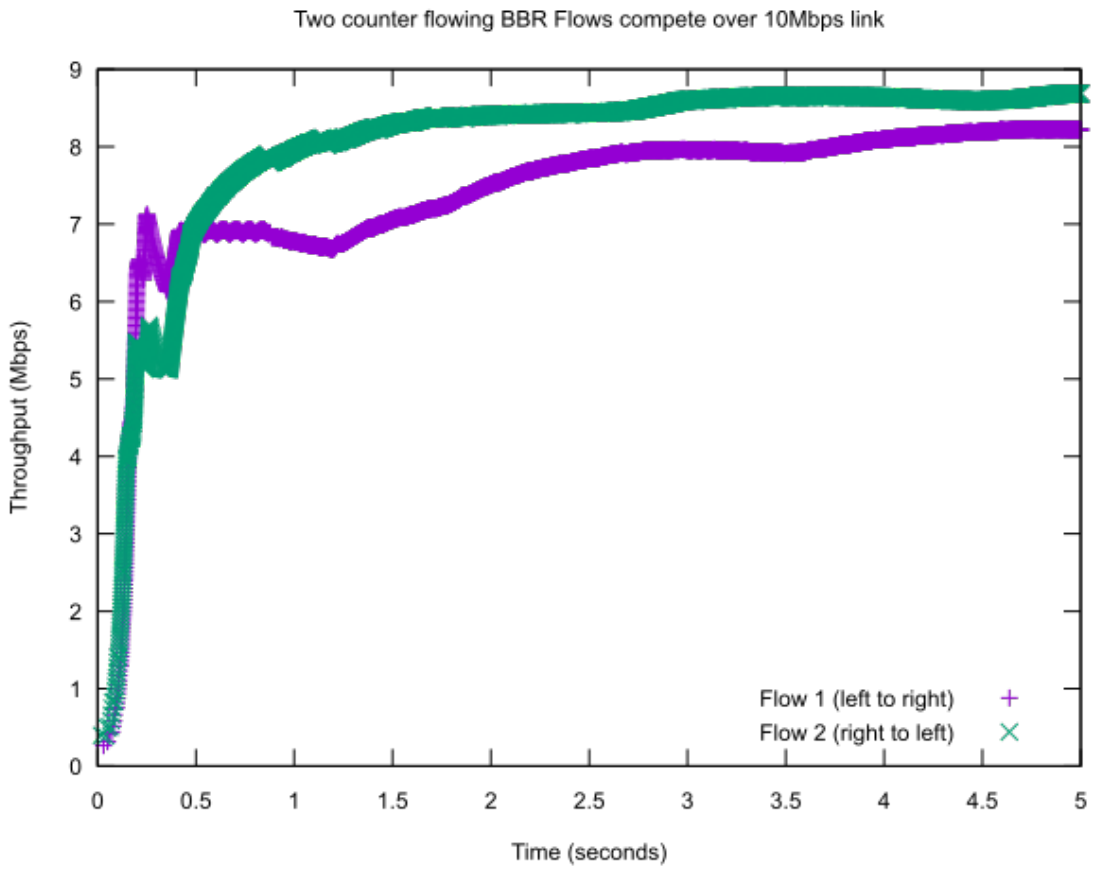


Figure 6.9: Results for One vs One Vertical Dumbbell over BBR

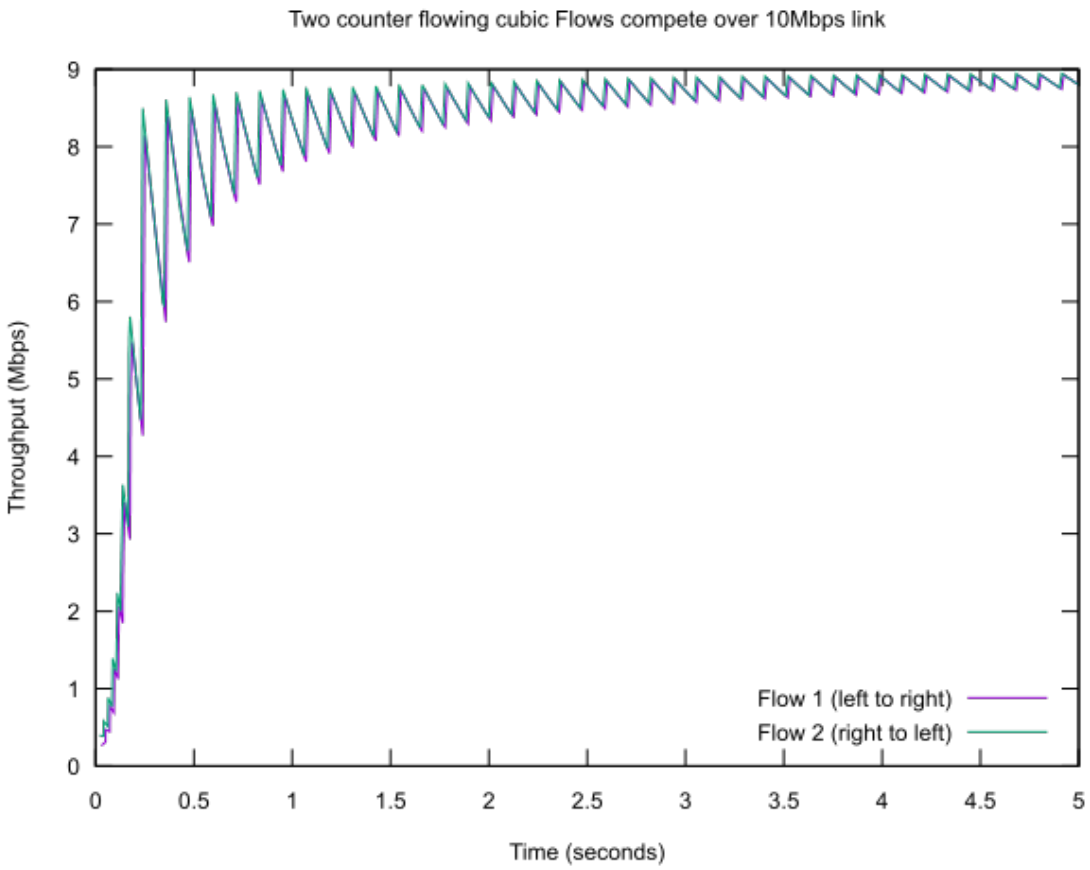


Figure 6.10: Results for One vs One Vertical Dumbbell over CUBIC

Figure 6.11 shows BBR In another dumbbell topology. In these simulations again BBR flows synchronise, probing for bandwidth per flow at around 0.2 seconds before the RTT probing phase at around 0.4 seconds and subsequent regrowth to the previously calculated bottleneck. This graph demonstrates what a stable connection would ideally experience using BBR.

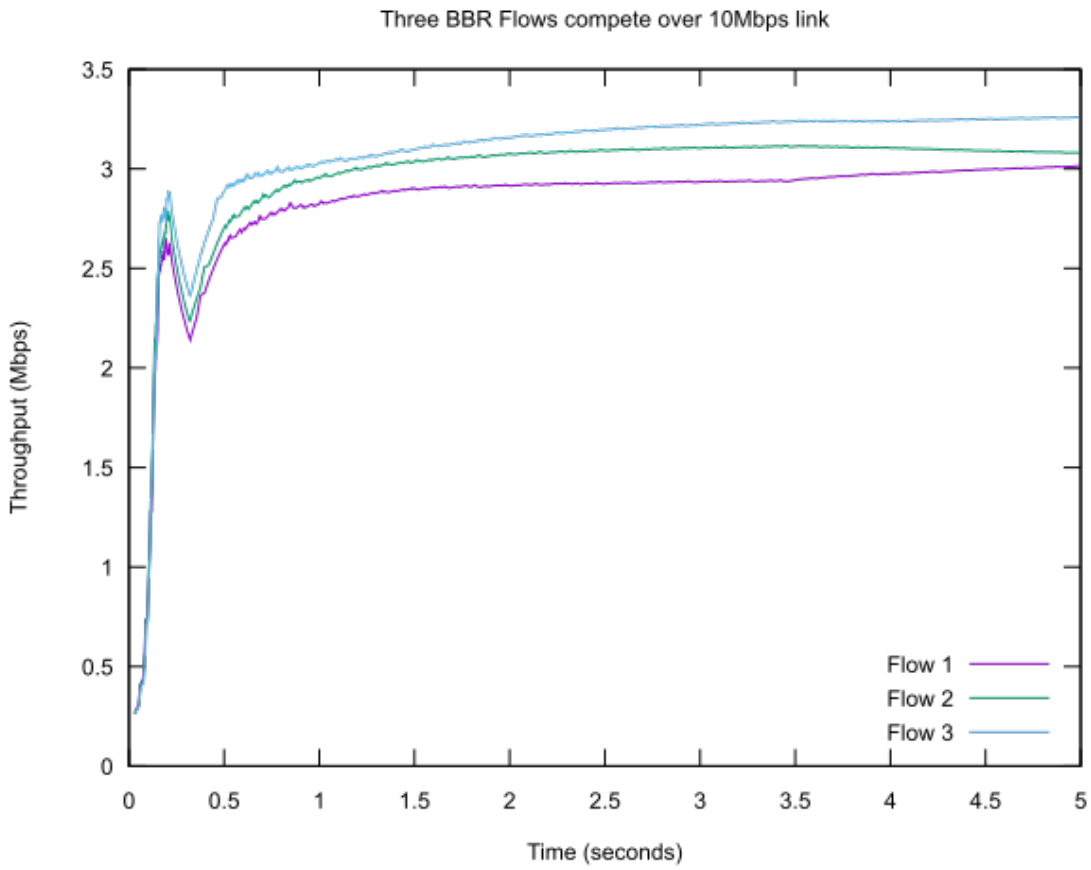


Figure 6.11: Results for 3 flow 3x3 Dumbbell over BBR

The flows in figure 6.11 have a combined throughput of 46.77Mb (out of 50Mb). However BBR is exceeded by both CUBIC (figure 6.15) and NewReno (figure 6.16) with total throughputs of 49.74Mb and 49.66Mb respectively. This is due to BBRs flow synchronising at a lower throughput to CUBIC and NewReno.

6.3 Fairness

Figures 6.12 and 6.13 show that BBR is able to adapt the number of flows to the bottleneck. As seen in both figures at around 1.5 seconds in, BBR synchronisation occurs and the flows start to share the bottleneck. However figure 6.13 shows a minimal bias, when the number of flows from one source out weights another, the source with the least number of flows is suppressed. This is likely caused in two parts, firstly synchronisation of the two flows as they originate from the same source. Secondly, due to the synchronisation of said flows, incorrect estimation by the single flow of the RTT due to congestion caused by the dominate flows.

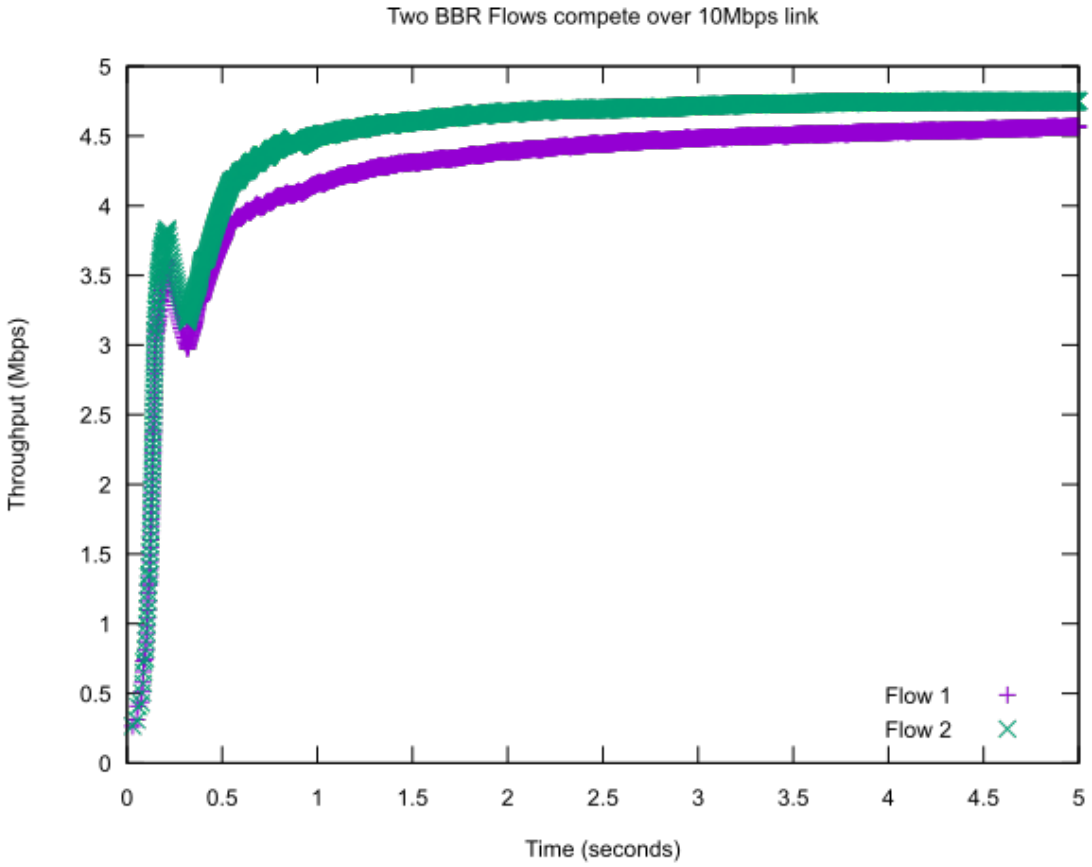


Figure 6.12: Results for One vs One Horizontal Dumbbell over BBR

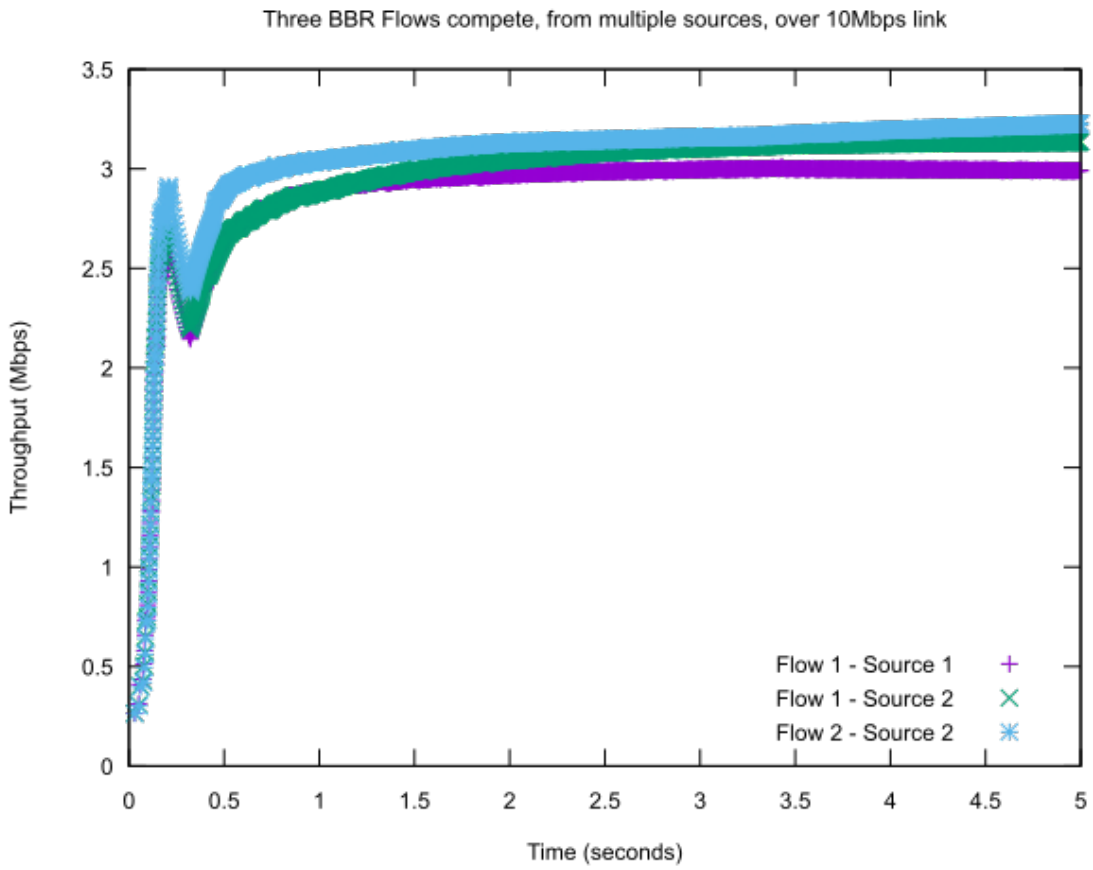


Figure 6.13: Results for One vs Two Horizontal Dumbbell over BBR

The third topology evaluates BBRs ability to maintain the optimal operating point over a bidirectional bottleneck. BBRs algorithm relies on timed monitoring of congestion to dictate a bottleneck and in doing so records the pace of packet delivery (dictated by ACK clock). When flows travel along the same path they use the same pacing and BBR synchronises these flows, causing the single flow to dominate the bandwidth (as seen in figure 6.14).

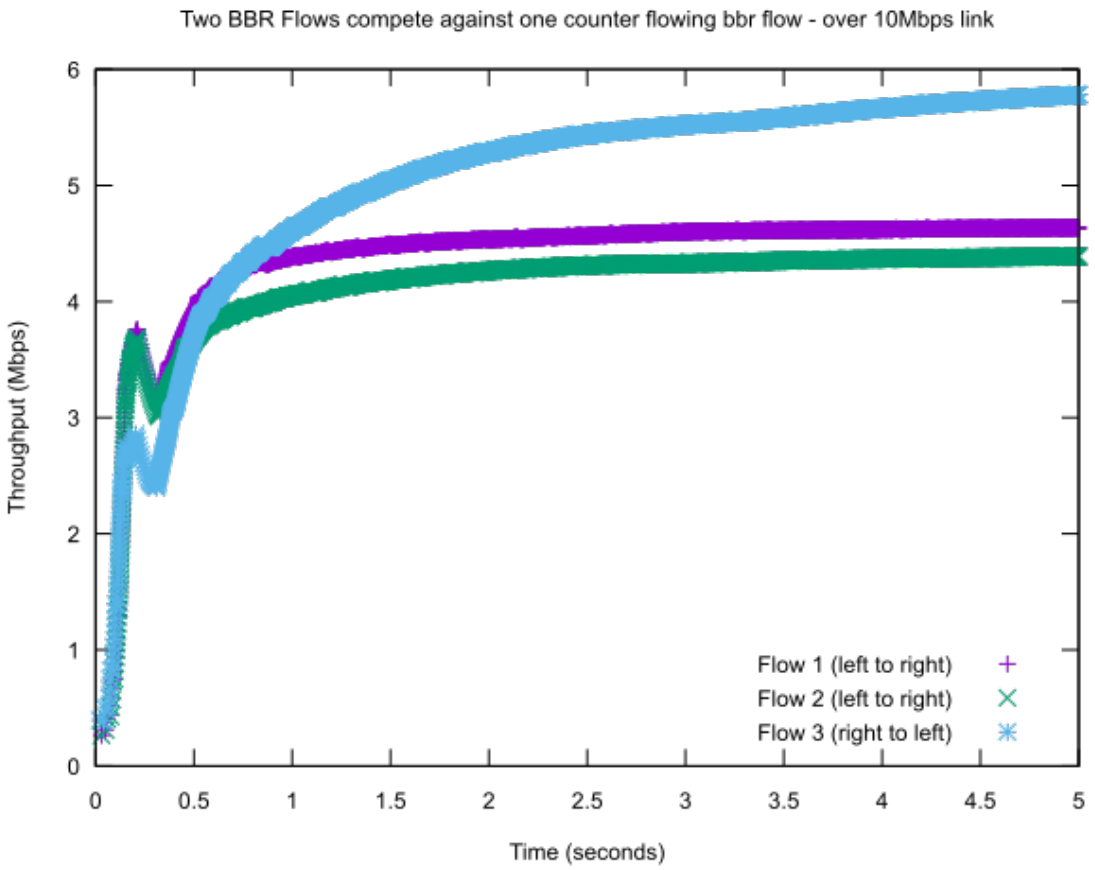


Figure 6.14: Results for One vs Two Vertical Dumbbell over BBR

When two apposing flows are present BBR (figure 6.9) closes the fairness gap further after every probe cycle is complete, indicating that BBR can handle bidirectional bottlenecks. CUBIC (figure 6.10) outperforms BBR in this topology, again CUBIC is able to achieve full fairness.

In contrast to figure 6.11 both CUBIC and NewReno show unfairness for the first 3 seconds of the simulations (figures 6.15 and 6.16), both showing almost 20% difference of the overall throughput at 0.4 seconds, easing into an equal fairness that can be observed from the 3 second mark onward.

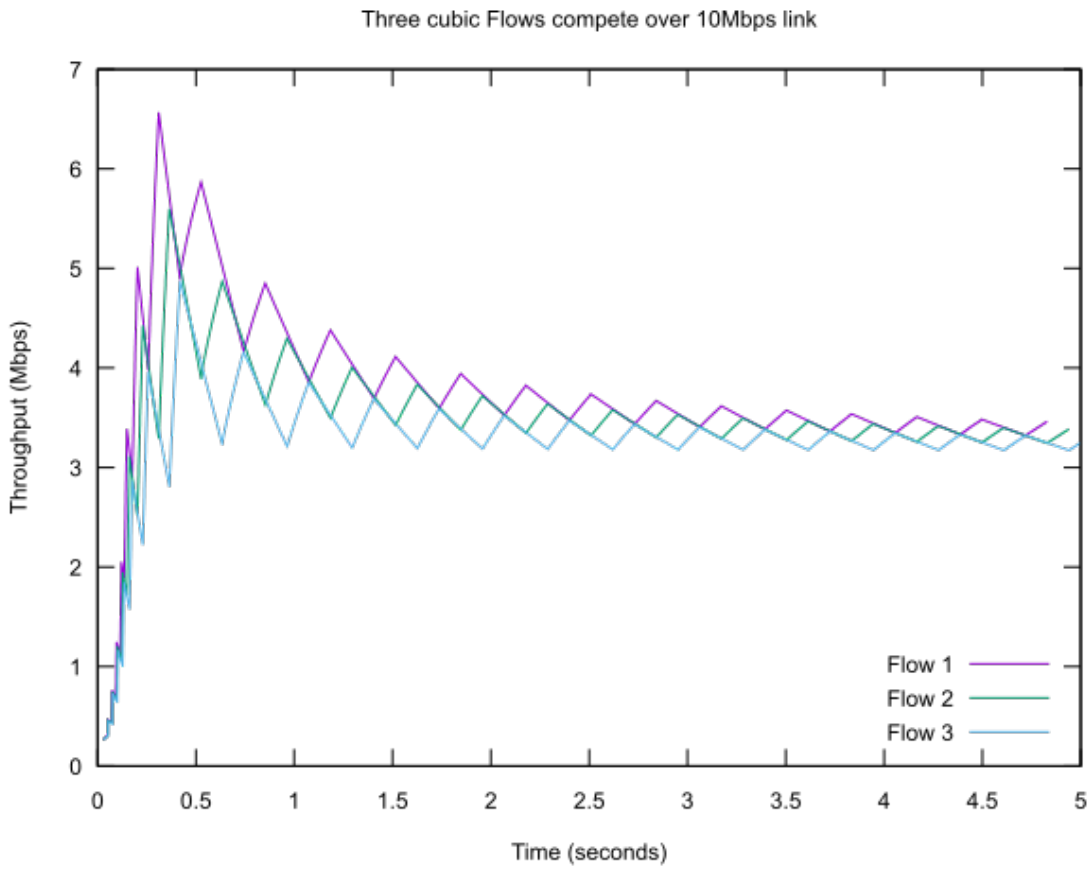


Figure 6.15: Results for 3 flow 3x3 Dumbbell over CUBIC

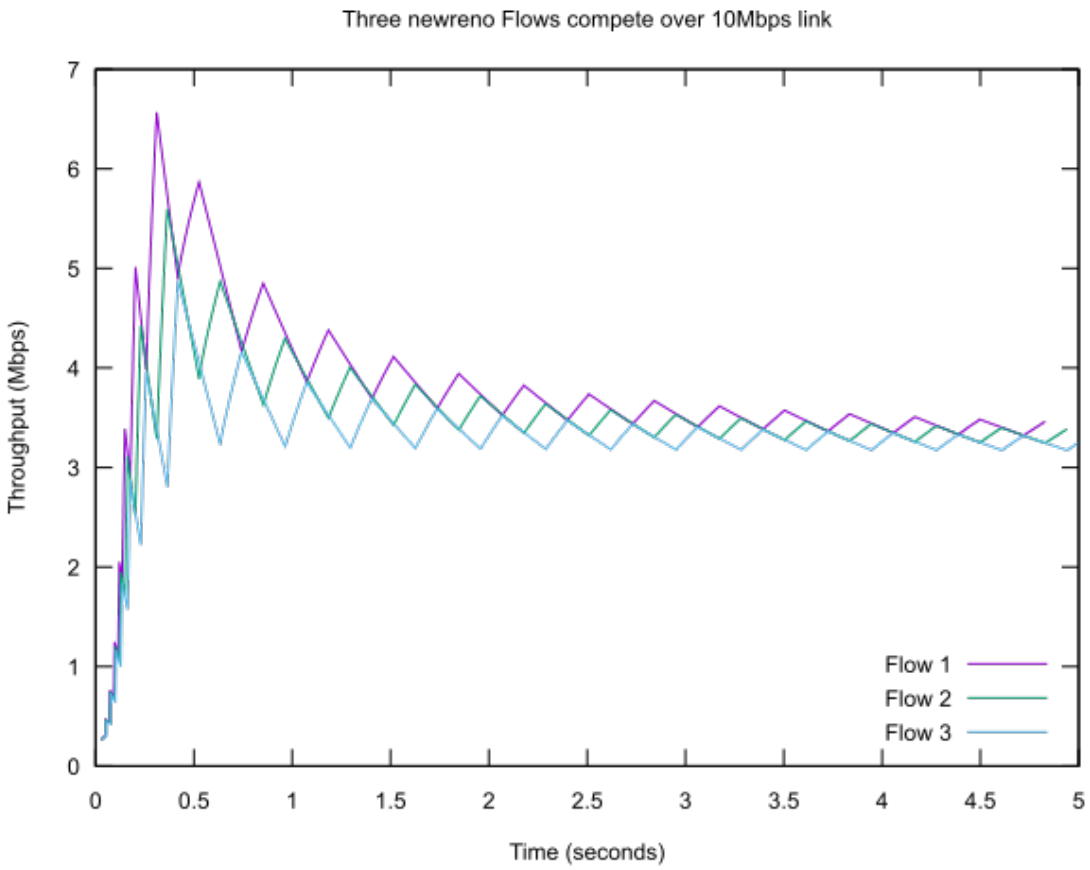


Figure 6.16: Results for 3 flow 3x3 Dumbbell over NewReno

Figures 6.17, 6.18 and 6.19 show the result of an additional flow joining the network from another source. Figures 6.17 and 6.18 demonstrate that both CUBIC and BBR favour the shortest path (joining flow - as it joins at node 2 closest to the destination node 3), calculating close to the bottleneck bandwidth before the original flow reaches the destination. However the original flow (longest path to destination - from node 0 to node 3) is able to gain considerable throughput and reach fairness half way through the 5 second simulation. It is clear that the two flows do synchronise, but without the ability to track BBRs internal variables it is unclear why the original flow is considerable more aggressive at acquiring throughput and why after 2.5 seconds the flows do not maintain equal fairness. This is an anomaly that could be investigates with additional time and bespoke networking software.

Whilst CUBIC behaviour mirrors that of BBR, NewReno appears to favour the joining flow in a manner expected by BBR. NewReno experiences the same joining flow spike, reaching close to 8Mbps before idling at a stable 6-7Mbps. The original flow, having a longer path, obtains a throughput inverse of the joining flow, idling at 3-4Mbps at 2.5 seconds after growing at around 1Mbps every second.

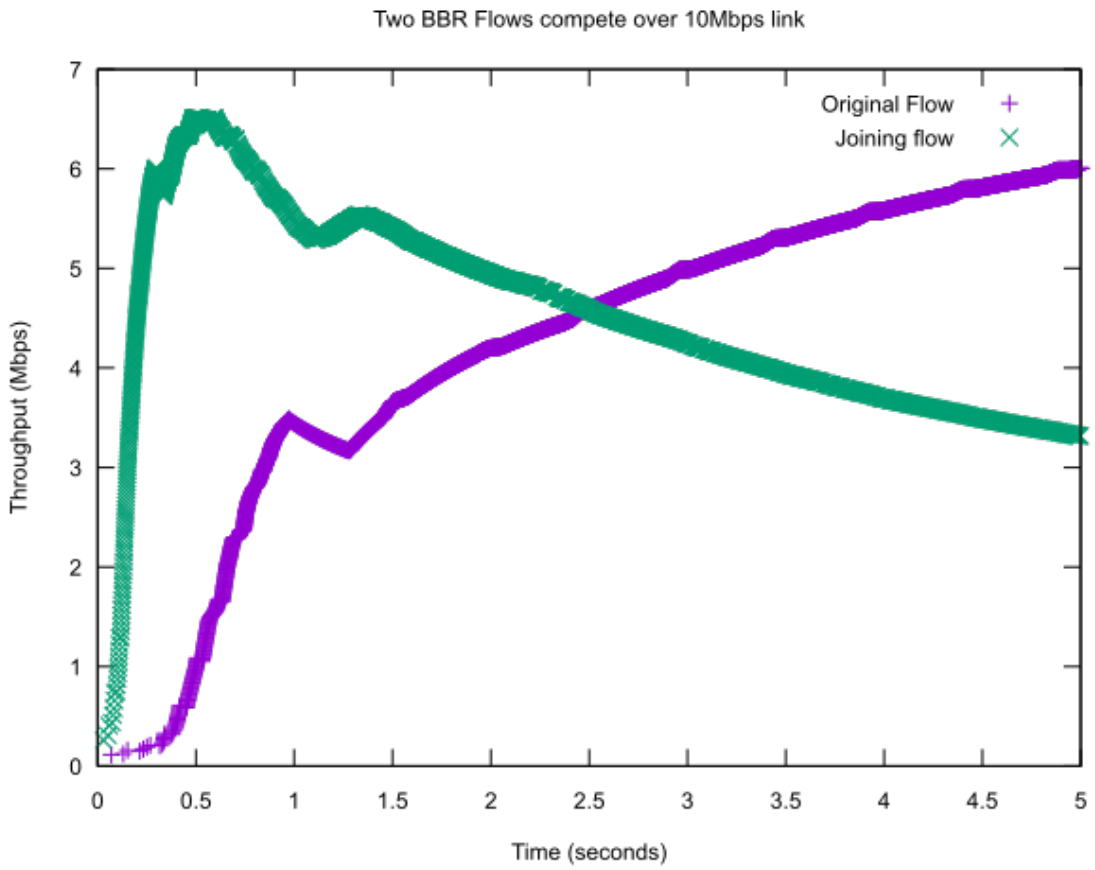


Figure 6.17: Results for joining flow on node 2 over BBR

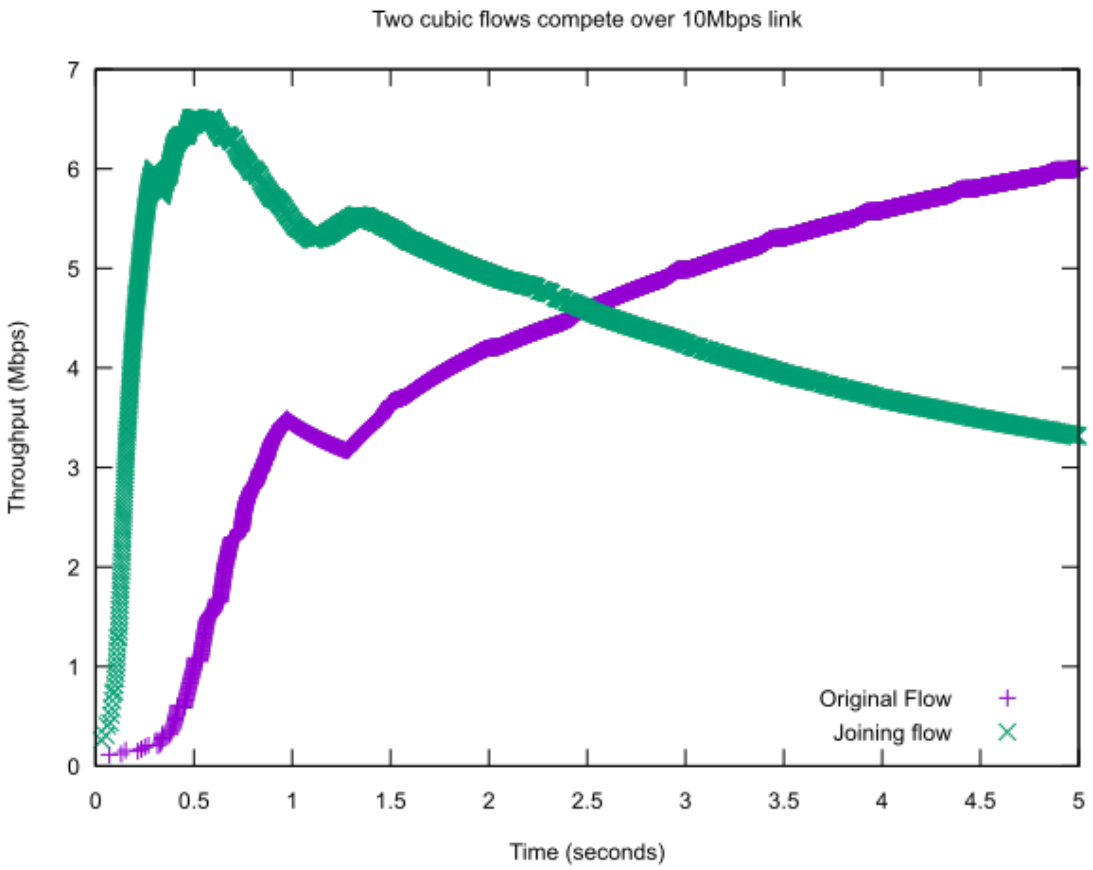


Figure 6.18: Results for joining flow on node 2 over CUBIC

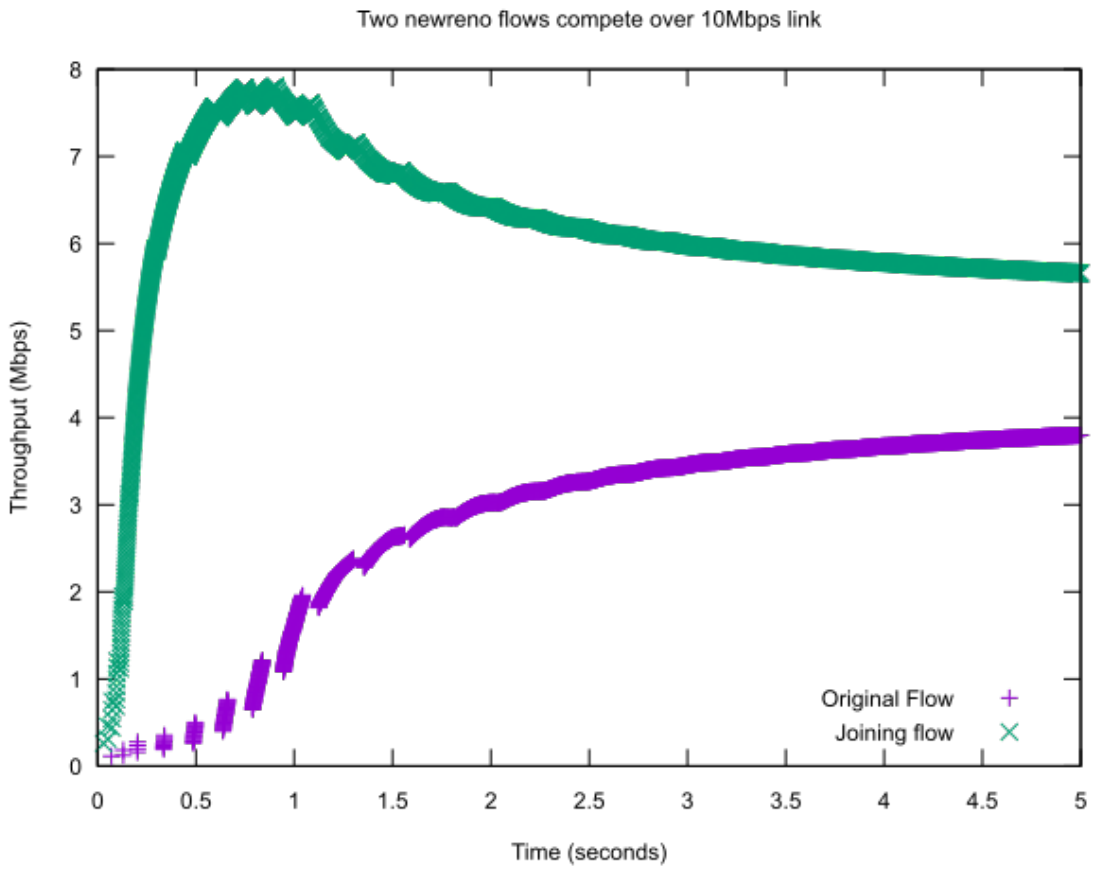


Figure 6.19: Results for joining flow on node 2 over NewReno

6.3.1 Delayed Joining Flows

For comparison with figure 6.17, figures 6.20 to 6.24 show the joining flow arriving at increments of 200ms later than the original flow. This topology negates the effects of shorter paths as well as resembling more likely scenarios - that flows enter and exit networks at random intervals. Another reason for choosing 200ms intervals, BBR's *probeRTTInterval* internal variable, which dictates the duration of an RTT probe, is 200ms which ensures that the first probe RTT is complete before a flow joins the network.
Figure 6.20 shows the BBR 200ms delayed flow is able to use the exponential startup to quickly reach similar throughput to the original flow. Unfortunately poor timing of the joining flows sees it probe RTT just as the original flow probes for bandwidth at 1 second. At 1.4 seconds the opposite happens, the original flow probes for RTT just before the joining flow undertakes probe bandwidth. These two opposing events sees the delayed joining flow overestimate the RTT and underestimate the available bandwidth, leading to heavy suppression by the original flow. This is a critical flaw of BBR that was observed also by Scholz (2017), that in order to be resolved, would require BBRs awareness of other flows internal state variables.

Figures 6.21 to 6.24 all show a common trend. By joining late the delayed flow is able to occupy a fair share of the bandwidth. The original flow undertakes both probing states before the delayed flow is able to compete. This leads to the original flow predicting higher throughput then is fair but predicting an accurate RTT. Consuming a considerable amount of bandwidth the original flow causes the delayed flow to inaccurately predict an RTT time - the path and buffer are not cleared by the dominate flow. As a result the flows will never achieve equal fairness.

This is a considerable issue for BBR, the unfairness presented here can have a large impact in industry. By suppressing connections with low RTT those unable to pay for the fastest connections will be hindered. This principle is known as net neutrality and refers to the right of all users to have equal access to network resources irrelevant of their infrastructure or geographical location.

Whilst Cardwell (2016) is able to demonstrate that multiple delayed joining BBR flows can reach equal fairness after 20 seconds, he negates the specific entry times of such flows and ignores the industrial nature of networks - 20 seconds of network unfairness is enough to dissuade customers not to return.

To improve BBR more research must be done introducing flows with delays of a range of 500 to 3000ms at intervals of 50ms. This would seek to address the opposing probe state situation seen in the first 1.5 seconds of figure 6.20.

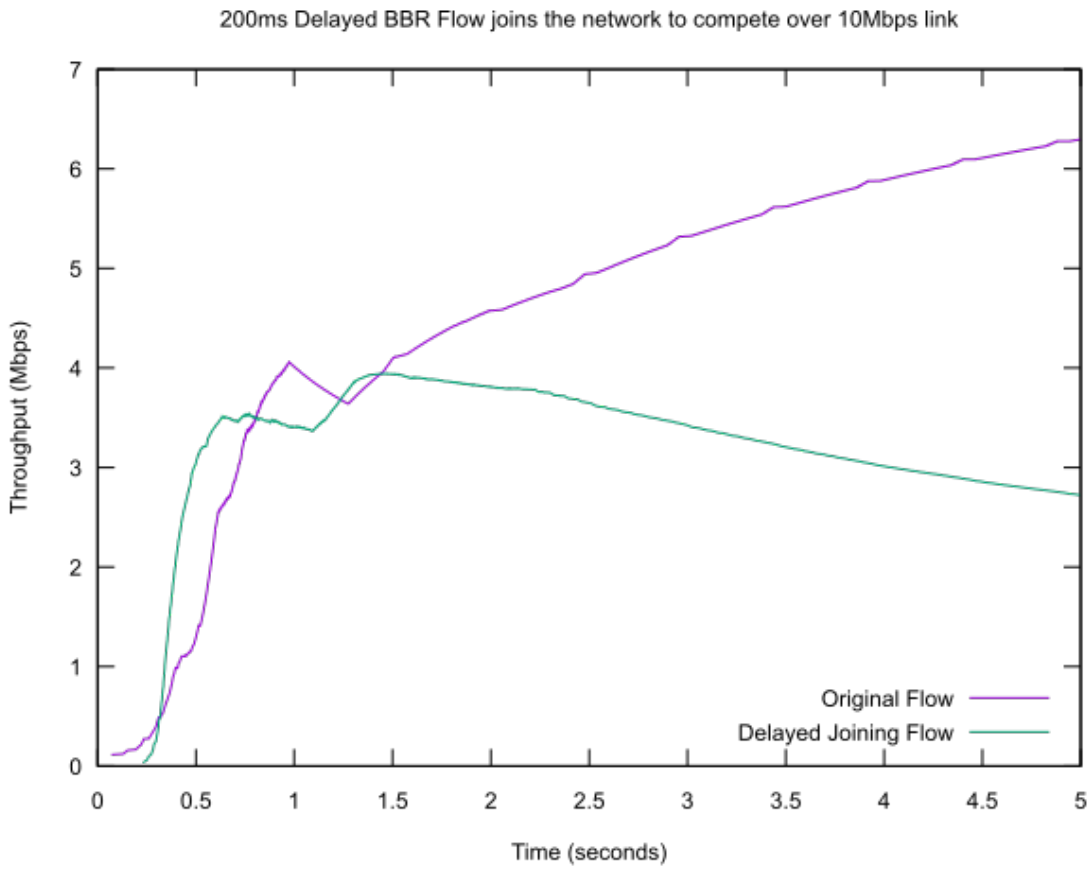


Figure 6.20: Results for 200ms delayed joining flow joining on node 2 running BBR

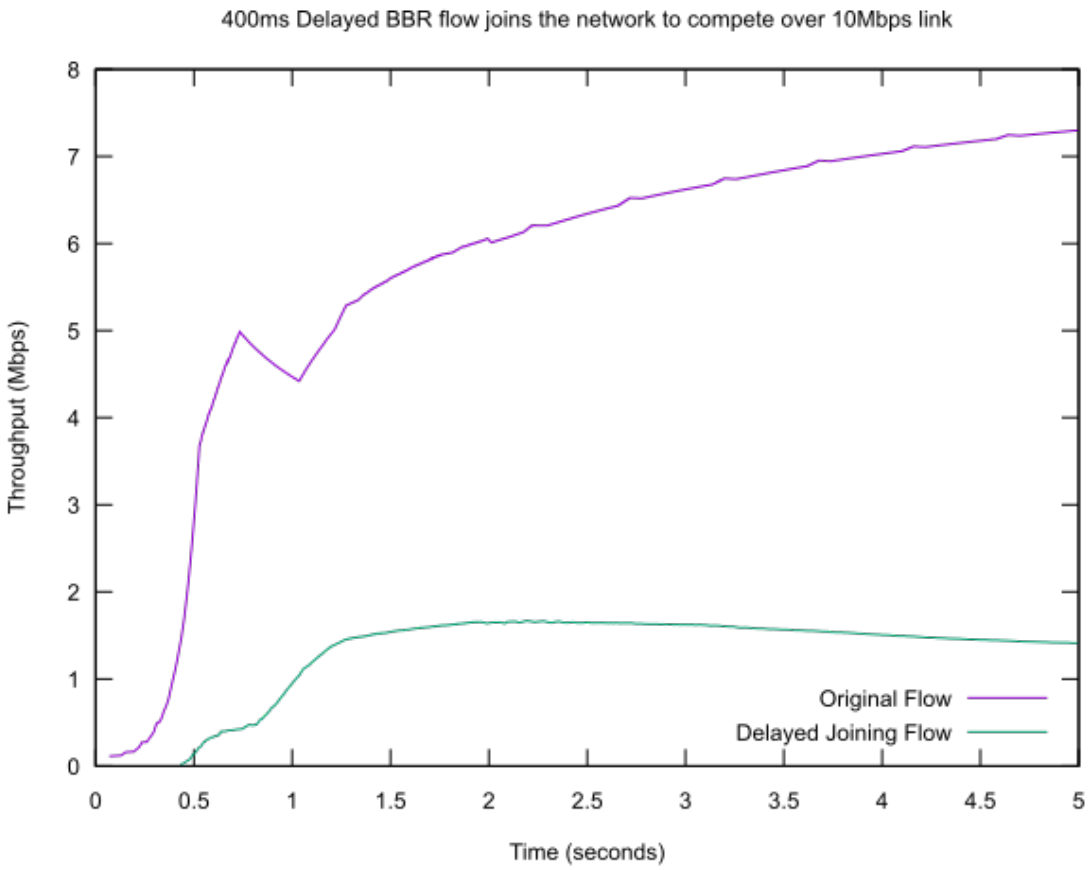


Figure 6.21: Results for 400ms delayed joining flow joining on node 2 running BBR

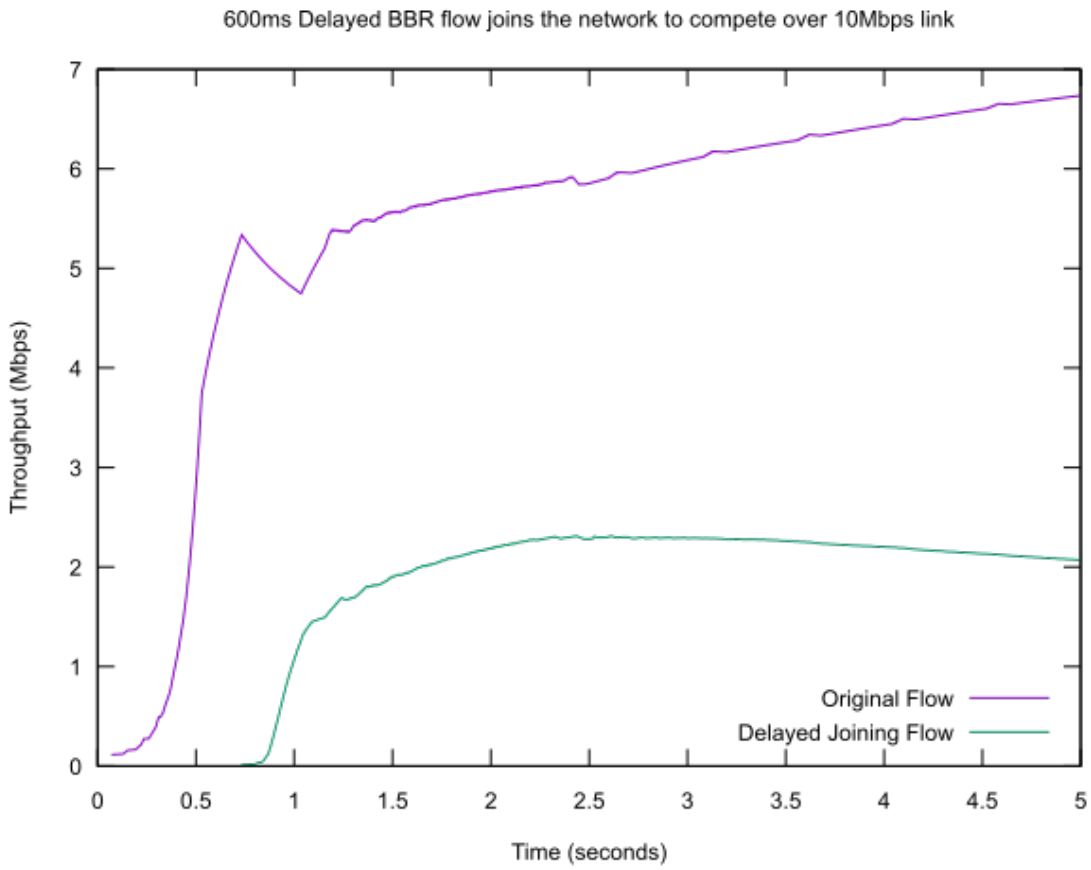


Figure 6.22: Results for 600ms delayed joining flow joining on node 2 running BBR

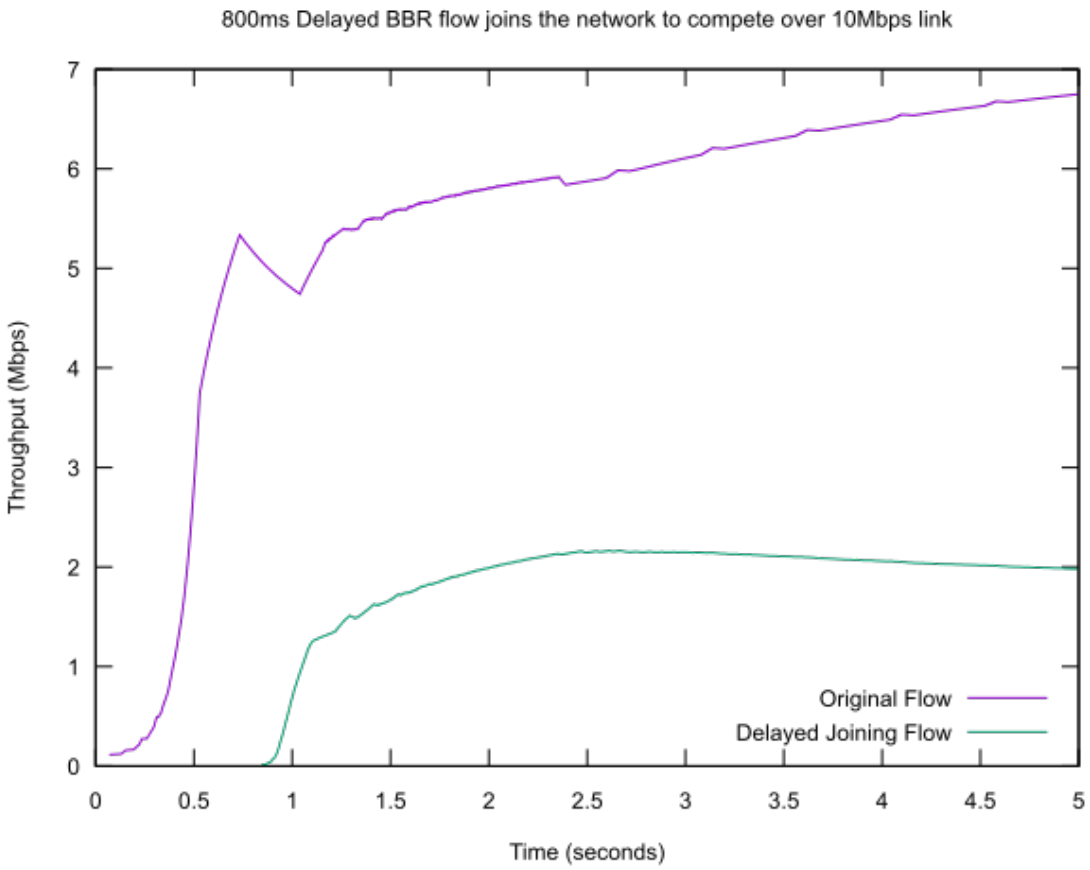


Figure 6.23: Results for 800ms delayed joining flow joining on node 2 running BBR

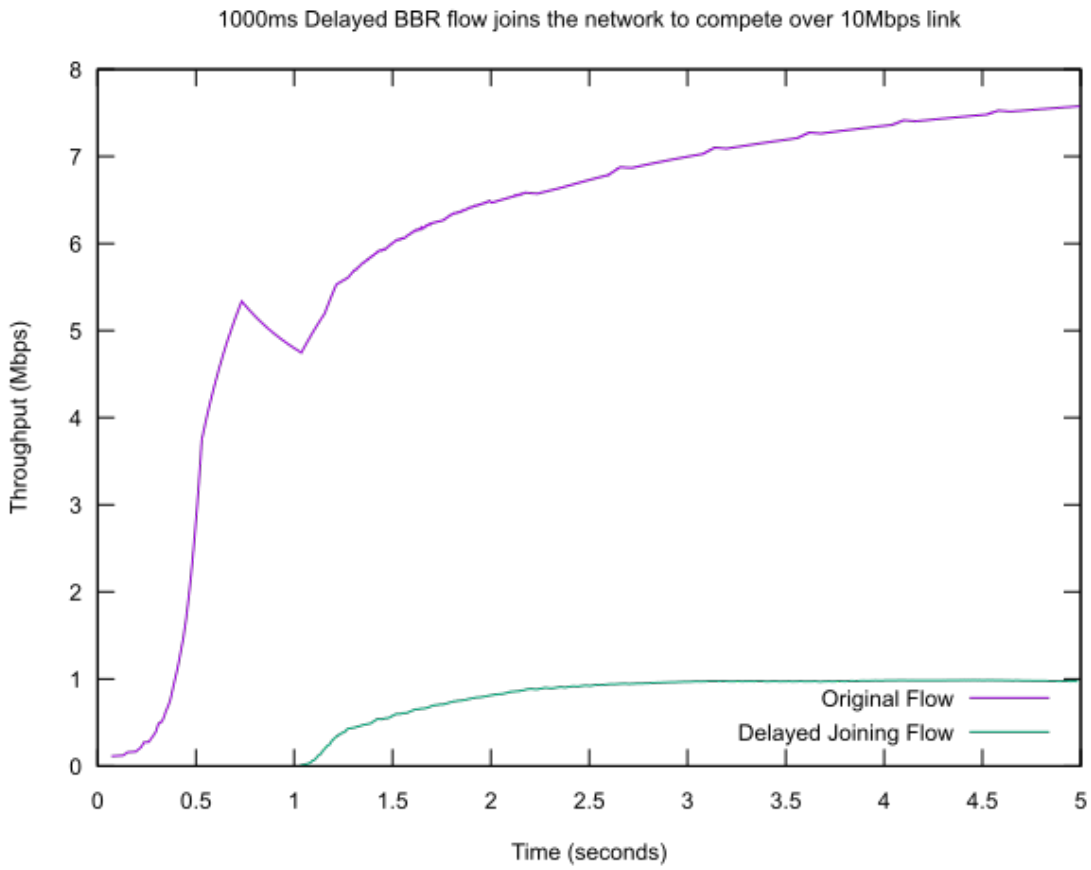


Figure 6.24: Results for 1000ms delayed joining flow joining on node 2 running BBR

6.3.2 Multiple Joining Flows

The flows in these simulations join the network at the same time (all flows start at 0.0 seconds) from individual nodes. This experiment replicates what might occur after the resolution of a network outage, many connections starting from multiple sources simultaneously. Figures 6.25, 6.26 and 6.27 display the results. Figure 6.25 shows the BBR is not capable of handling such events. Not only does no fairness resolution occur, flows do not synchronise probing events and the end result is considerably different throughput for all 3 flows.

Given the results it is clear that two BBR flows will overlap in a given time dependant on the available throughput. In figure 6.17 the overlap occurs at 2.5 seconds, whereas in figure 6.25 the overlap occurs at 3.5 seconds, an additional seconds due to less available throughput because of the additional joining flow. Given more time this would be a relationship worth pursuing.

CUBIC and NewReno however operate in the same regard. Not only do they demonstrate bias towards the shorter path (as BBR did to start), they never reach equal fairness during the simulation and finish with similar separation to BBR.

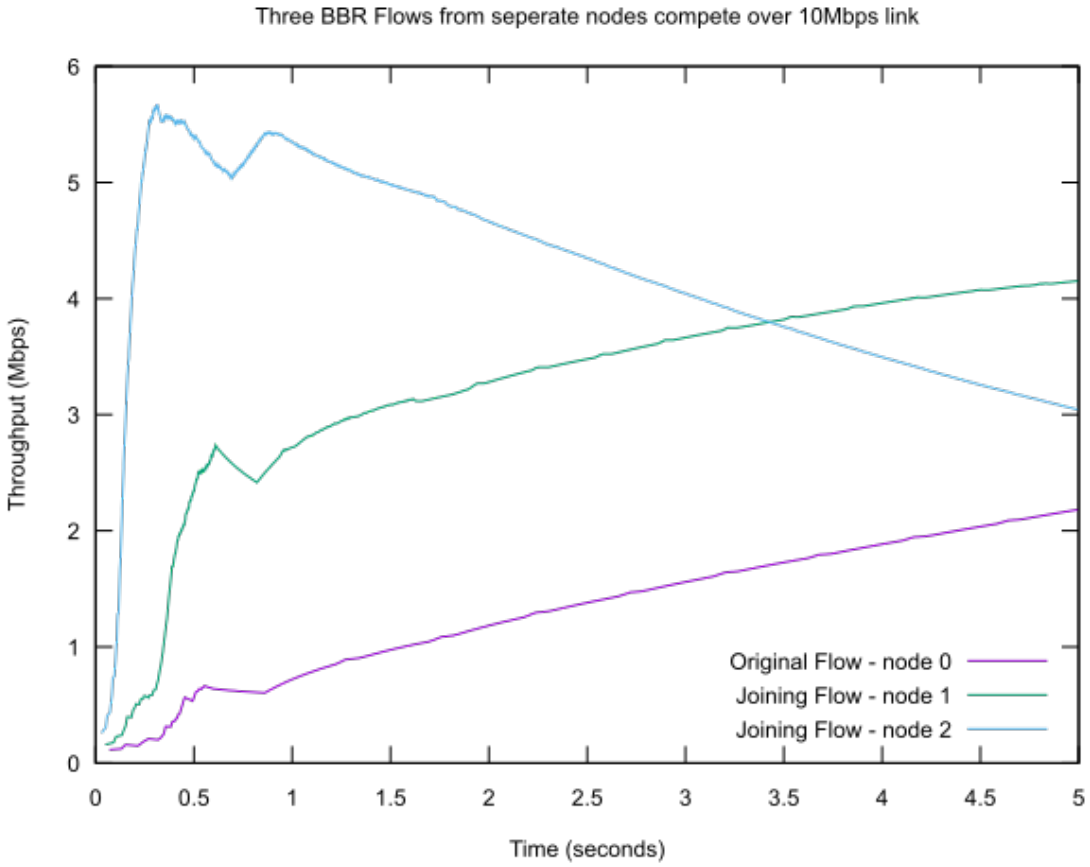


Figure 6.25: Results for joining flows on nodes 1 and 2 running BBR

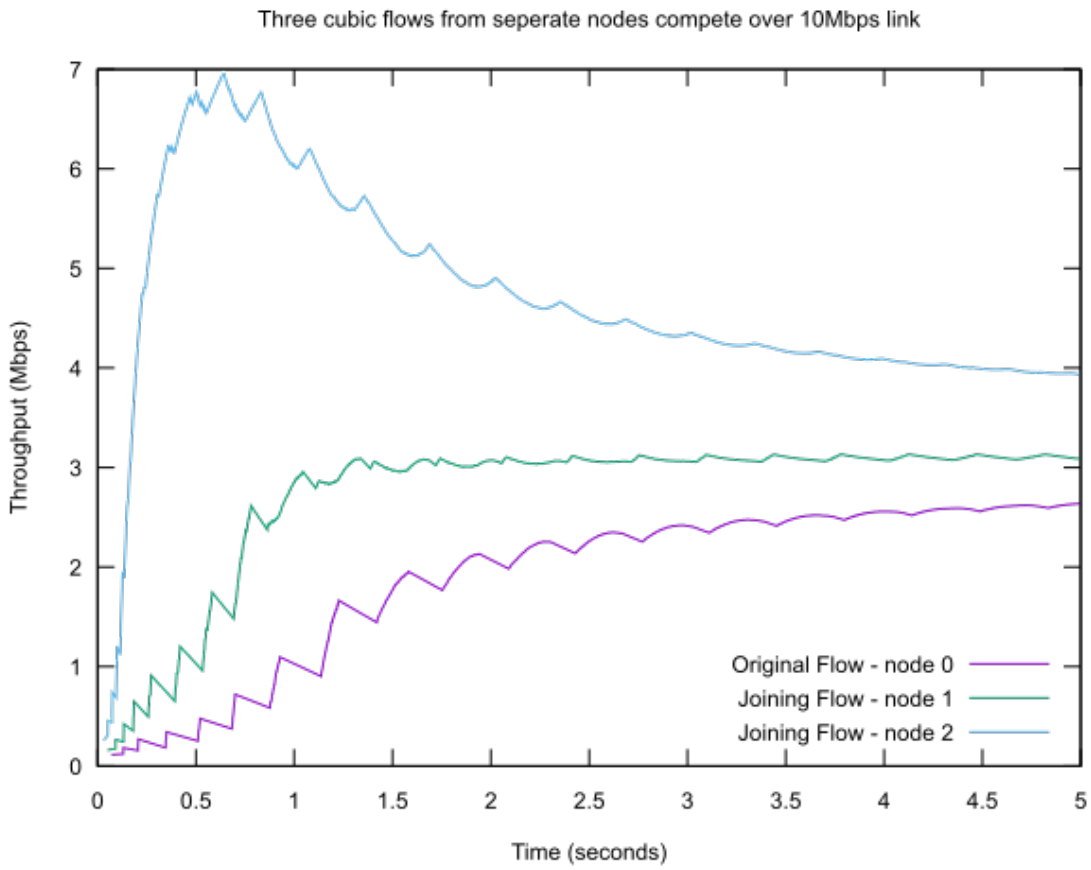


Figure 6.26: Results for joining flows on nodes 1 and 2 running CUBIC

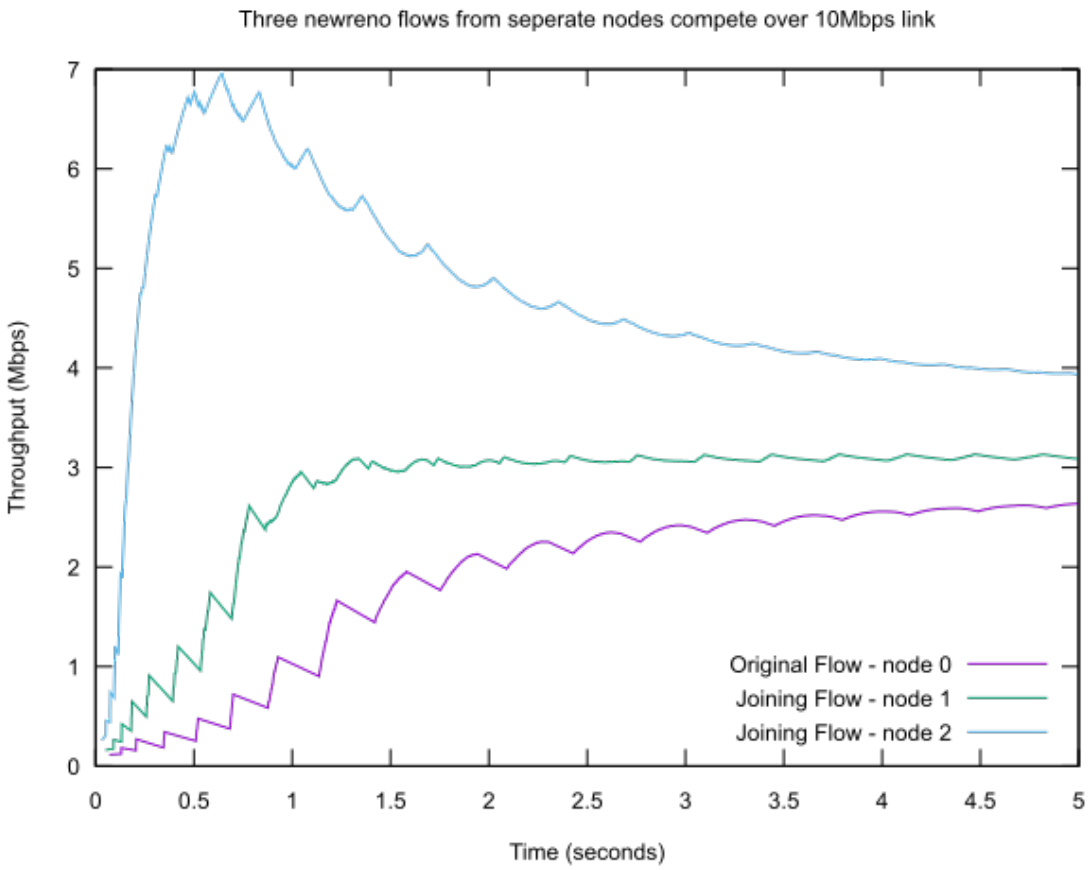


Figure 6.27: Results for joining flows on nodes 1 and 2 running NewReno

6.4 Summary

6.4.1 Results

BBR as a candidate for TCP congestion control is theoretically promising but this research shows that BBR struggles to improve over loss based congestion control algorithms. BBR can achieve Kleinrock's optimal saturation point over a given path, but only in non challenging conditions where other flows have the same RTT and path. Currently BBR matches CUBIC and NewReno in its ability to manage multiple directional flows, share the bandwidth and redistributing the throughput. However no algorithm provides complete advantageous characteristics over another and fairness, to the extent desired for net neutrality, is still absent. Further investigations that analyse the internal state variables of BBR at run time are required to suggest suitable changes to the implementation.

6.4.2 Artefact

The artefact is a simulation environment that analyses the BBR algorithm for use in modern networks. Using the artefact researchers can evaluate BBR against a vast number of other TCP algorithms or simply interpret the current findings again through re-use of the traceroute and parsing scripts supplied with every simulation run. The use of global constants allows full configuration of the simulations, accompanied by comments to give clarity to all code. In using professional coding standards the simulation scripts are easy to interpret, used alongside the parsing and plotting scripts, these enable users to create or manipulate existing simulations and quickly generate results to be graphed.

Chapter 7

Conclusion

7.1 Evaluation

The implementation of the artefact achieves its main goal, to evaluate the BBR algorithm and supply a *testbed* for future research to users of any technical level. Repeatability and reproducibility are important and the artefact's implementation supports this in its structure, keeping separate the simulation scripts from the parsing and plotting scripts, the .cc files can be reused in other related works. NS-3 was the most suitable simulator for this, allowing logical separation between the simulator and the simulation (scripts).
The simulator (NS-3) did however produce a number of issues. Not only was the debugging and testing complex and time consuming, but a limitation of the software (its inability to run multiple TCP types at runtime) dramatically reduced the number of simulations available and thus hindered the analysis of BBR, limiting experiments to only intra-protocol simulations and comparisons. This impacted the project, other research evaluates BBR's behaviour with other TCP algorithms because this is highly important when BBR is released and considerably more representable of modern networks - which may see many flows with varying TCP algorithms.
The RAD methodology has worked well in this project due to the time frame that was given. RAD helped to develop simulations that were thorough but not excessive, keeping me within the time frames set out at the start of the project. Time management was monitored using spreadsheet software, the original plan (appendix 1) set out weekly constraints for objectives which, by using an live updated copy, allowed tracking of overall project progress (appendix 2). The only issue with the chosen methodology was the user feedback, having no end user feedback, my feedback was restricted to the project supervisor and compiler affirmations.
In section 1.2.1 the aims and objectives of this project were set out. Objectives 1 has been completed, and the success criteria can seen in section 2 and section 3. Objective 2 can be seen in section 4.2. Objectives 3,4 and 5 success criteria can be seen in section

6, however objective 4 is only partially complete (intra-protocol is done inter-protocol is not) due to the software limitation mentioned above. Objective 6 is not complete, again due to limitations in the software as well as a lack of time and understanding. Objective 6 sets out to investigate BBR during network changes, something that requires the understanding ability to control the internal state variables of BBR through modifications (known as patches) of NS-3 simulator itself. The issues with objectives 4 and 6 were risks highlighted as possibilities on the risk analysis - see appendix 1 ”Simulations are outside of the capability of the software” however the solution provide was vague, ambiguous and too time consuming - ”Investigate the software to be used and suggest other software or alternative implementations of the simulation” and did little to aid the project.

7.2 Future Work

This project was built with continuation in mind and simulations could easily be extended to scale for more nodes, more flows or both. Other works could look at modifying the global variables of the simulations scripts, which were declared as constants to allow easy and quick change to attributes such as:- bottleneck size, bottleneck locations, bandwidth of paths entering/exiting the bottleneck and path delays. These constants were used for example for adjusting the delay in the delayed starting flow simulations. Possible future work could include parsing of the trace files, already present in the artefact, to analyse network attributes such as latency and goodput for greater indepth evaluation of Bandwidth Bottleneck Round trip time.

WORD COUNT: 10,590

Bibliography

Ahmad, M., Hussain, M., Abbas, B. and Aldabbas, O., 2018. “end-to-end loss based tcp congestion control mechanism as a secured communication technology for smart healthcare enterprises” [online]. Available From: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=8294190 [Accessed 16 November 2018].

Allman, M., Glenn, N., Paxson, V., ACIRI-ICSI and Stevens, W., 1999. Tcp congestion control. Available from: https://tools.ietf.org/html/rfc2581 [Accessed 7 February 2019].

Allman, M., Paxson, V., ICSI, Blanton, E. and University, P., 2009. Tcp congestion control. Available from: https://tools.ietf.org/html/rfc5681 [Accessed 7 February 2019].

Baker, F., Systems, C., Fairhurst, G. and of Aberdeen, U., IETF Recommendations Regarding Active Queue Management. 2015. Available from: https://tools.ietf.org/html/rfc7567 [Accessed 11 February 2019].

Bonaventure, O., 2016. ”april 2016: Editor’s message” [online]. ACM Digital Library. Available From: http://www.sigcomm.org/node/3897 [Accessed: 05 February 2019].

Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J. and Zhang, L., 1998. Recommendations on queue management and congestion avoidance in the internet. Available from: https://tools.ietf.org/html/rfc2309 [Accessed 9 February 2019].

Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H. and Jacobson, V., 2016a. Bbr: Congestion-based congestion control. *ACMqueue*, 14 (5).

Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H. and Jacobson, V., 2016b. Bbr congestion control. Available from: https://www.ietf.org/proceedings/97/slides/slides-97-iccrg-bbr-congestion-control-02.pdf [Accessed 1 March 2019].

Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H. and Jacobson, V., July 2017. Bbr congestion control: An update. Available from:

https://www.ietf.org/proceedings/98/slides/slides-98-iccrg-an-update-on-bbr-congestion-control-00.pdf [Accessed 3 March 2019].

Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., Swett, I., Iyengar, J., Vasiliev, V. and Jacobson, V., March 2017. Bbr congestion control: Ietf 99 update. Available from: https://www.ietf.org/proceedings/99/slides/slides-99-iccrg-iccrg-presentation-2-00.pdf [Accessed 4 March 2019].

Cardwell, N., Cheng, Y., Yeganeh, S. H., Jacobson, V. and inc, G., 2017. "bbr congestion control". Internet Congestion Control Research Group. Available from: https://tools.ietf.org/html/draft-cardwell-iccrg-bbr-congestion-control-00 [Accessed 10 February 2019].

Cerf, V. and Kahn, R., 1974. "a protocol for packet network intercommunication". *IEEE Transactions on Communications*, 22 (5), 637–648.

Claypool, M., Chung, J. W. and Li, F., 2018. Bbr' – an implementation of bottleneck bandwidth and round-trip time congestion control for ns-3. *WNS3 '18 Proceedings of the 10th Workshop on ns-3*, 1, 1–8. Available from: https://dl.acm.org/citation.cfm?id=3199903 [Accessed 25 February 2019].

Crichigno, J., Csibi, Z., Bou-Harb, E. and Ghani, N., 2018. "impact of segment size and parallel streams on tcp bbr". *2018 41st International Conference on Telecommunications and Signal Processing (TSP)*. Available from: https://ieeexplore.ieee.org/document/8441250/ [Accessed 20 February 2019].

Dai, T., Zhang, X. and Guo, Z., 2018. Poster abstract: Analysis and experimental investigation of bbr. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 1–2.

Fleshgrinder, 2014. "file:tcp slow-start and congestion avoidance.svg". Available from: https://commons.wikimedia.org/wiki/File:TCP_Slow-Start_and_Congestion_Avoidance.svg [Accessed 15 February 2019].

Ha, S., Rhee, I. and Xu, L., 2005. "cubic: A new tcp-friendly high-speed tcp variant" [online]. Available From: https://www.cs.princeton.edu/courses/archive/fall16/cos561/papers/Cubic08.pdf [Accessed 14 February 2019].

Hock, M., Bless, R. and Zitterbart, M., 2017. "experimental evaluation of bbr congestion control". Available from: https://ieeexplore.ieee.org/document/8117540/ [Accessed 10 November 2018].

Jacobson, V., 1988. "congestion avoidance and control" [online]. Available From: https://ee.lbl.gov/papers/congavoid.pdf [Accessed: 10 February 2019].

Kleinrock, L., 1979. ”power and deterministic rules of thumb for probabilistic problems in computer communications.” [online]. Available From: https://www.lk.cs.ucla.edu/data/files/Kleinrock/Power and Deterministic Rules of Thumb for Probabilistic.pdf [Accessed 05 February 2019].

Kumarl, B. P. and Prashanth, Y., 2014. Improving the rapid application development process model. Available from: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7056962 [Accessed 9 March 2019].

Levasseur, B., 2014. Available from: http://perform.wpi.edu/downloads/cubic/ns-3.27.tar.gz [Accessed 7 March 2019].

Low, S., Peterson, L. and Wang, L., 2000. ”understanding tcp vegas: Theory and practice” [online]. Available From: ftp://ftp.cs.princeton.edu/reports/2000/616.pdf [Accessed 10 February 2019].

Ma, S., Jiang, J., Wang, W. and Li, B., 2017. Fairness of congestion-based congestion control: Experimental evaluation and analysis. Available from: https://arxiv.org/pdf/1706.09115.pdf [Accessed 28 February 2019].

NS-3, 2018. Older releases. Available from: https://www.nsnam.org/releases/ns-allinone-3.27.tar.bz2 [Accessed 5 March 2019].

NS-3, 2019a. ”conceptual overview” [online]. Available From: https://www.nsnam.org/develop/contributing-code/coding-style/ [Accessed: 12 February 2019].

NS-3, 2019b. ”parsing ascii traces” [online]. Available From: https://www.nsnam.org/docs/release/3.9/tutorial/tutorial$_2 3.html Parsing - Ascii - Traces [Accessed : 16 March 2019]$.

Ramakrishnan, K., Networks, T., Floyd, S., ACIRI, Black, D. and EMC, 2001. The addition of explicit congestion notification (ecn) to ip. Available from: https://tools.ietf.org/html/rfc3168 [Accessed 14 February 2019].

Scholz, D., Jaeger, B., Schwaighofer, L., Raumer, D., Geyer, F., and Carle, G., 2018. Towards a deeper understanding of tcp bbr congestion control. Available from: https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/IFIP-Networking-2018-TCP-BBR.pdf [Accessed 3 March 2019].

Tanenbaum, A. and Wetherall, D., 2011. *Computer Networks*. 5th edition. New Jersey USA: Prentice Hall.

Appendices

Risks	Solutions
Simulations are outside of the capability of the software.	Investigate the software to be used and suggest other software or alternative implementations of the simulation.
Changes to design or understanding of the BBR protocol.	Allow time throughout the project to research the most recent implementation of the BBR protocol as it is highly likely to undergo changes whilst the project is conducted.
All aims can not be achieved in the required time frame.	Set a priority list to establish precedent to allow revoking of less important aims within a given objective.
Unforeseen Circumstances (eg natural disasters, power outages, corruption of data etc.).	Loss or corruption of data, either via accidental damage or a naturally occur event, can be mitigated by keeping a backup of all digital data (from data collect by simulations to the code that runs the simulations themselves). This includes the report documentation.
Affected by illness.	Set small margin of error in time planning to account for this possibility.
Artefact is infeasible	Complete the artefact with only the feasible features.

Table 1: Risk Analysis

Below contains the code for parsing the traceroute files for use in graph plots. PORT-NUMBER and IPADDR represents the target node of which the user desires throughput statistics.

```
import csv

# seconds counters
previousCounter = 0.0
counter = 5

# list of sequence numbers
# used to remove duplicate rows
seqList = []

# packet counter to calculate throughput
numberOfBytes = 0.0

with open('./replication/Replication.of.shiyao.Arxiv.figure1.tr', newline='') as csvfile:
rowreader = csv.reader(csvfile, delimiter=' ', quotechar='"')
for row in rowreader:
```

```
# get only recieved packets
# from a given host
# to differentiate flows we just isolate with port number where required
if row[0].startswith('r') and row[28] == IPADDR and row[34] == PORTNUMBER:

# prevent duplicate packets
if row[18] not in seqList:
seqList.append(row[18])

# check qualifying packets - should show payload on last element
if len(row) == 43:
# checking payload here (last element on 0-starting index)
if row[42] == "(size=1000)":

# group in 40ms batches if required for GNUPLOT
if float(row[1]) <= counter and float(row[1]) >= previousCounter:
numberOfBytes = numberOfBytes + float(1000)

# calculate throughput based on packet size 1000 bytes
# (set in configuration 'PACKET_SIZE' = 1000)
# numberOfPackets * 1052 = throughtput Bytes
throughput = numberOfBytes
throughput *= 8 # convert to bits
throughput /= 1000000 # convert to Megabits
throughput /= (float(row[1]))
#print("bytes: {} \n throughtput: {}".format(numberOfBytes, throughput))
print("{},{}".format(float(row[1]), throughput))
```

The below code is run on the simulator. It simulates network topologies and the flow specific information information.

```
//
// Network topology
//
//            10.0.0.x                192.168.1.x                 10.2.0.x
//       10.0.0.1    10.0.0.2   192.16.1.1     192.168.1.2   10.2.0.1     10.2.0.2
//
//       n0 -------------(n1/router) --------------- (n5/router)------------ n2
//       n3 -------------(         )                 (         )------------ n4
//       n7 -------------(         )                 (         )------------ n6
//
//       10.3.0.1    10.3.0.2                                 10.4.0.1    10.4.0.2
//            10.3.0.x                                             10.4.0.x
//
//       10.7.0.1    10.7.0.2                                 10.6.0.1    10.6.0.2
//            10.7.0.x                                             10.6.0.x
//
// - 3 Flows from n0, n3 and n7 using BulkSendApplication over BBR
// - Start and stopped at the same time
// - Tracing of queues and packet receptions to file "*.tr" and
//   "*.pcap" when tracing is turned on.
//

// System includes.
#include <string>
#include <fstream>

// NS3 includes.
#include "ns3/core-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
#include "ns3/network-module.h"
#include "ns3/packet-sink.h"
#include "ns3/ipv4-global-routing-helper.h"

using namespace ns3;
```

```
// Constants.
#define ENABLE_PCAP      false     // Set to "true" to enable pcap
#define ENABLE_TRACE     true     // Set to "true" to enable trace
#define BIG_QUEUE        2000      // Packets
#define QUEUE_SIZE       100       // Packets
#define START_TIME       0.0       // Seconds
#define STOP_TIME        5.0       // Seconds
#define S_TO_R_BW        "100Mbps" // Server to router BW
#define S_TO_R_DELAY     "10ms"
#define DELAY_50         "50ms"     // 50ms delay
#define R_TO_C_BW        "100Mbps"  // Router to client BW
#define R_TO_C_DELAY     "0ms"
#define R_TO_R_BW        "10Mbps"
#define R_TO_R_DELAY     "0ms"
#define PACKET_SIZE      1000      // Bytes.

// Uncomment one of the below.
//#define TCP_PROTOCOL     "ns3::TcpBbr"
//#define TCP_PROTOCOL     "ns3::TcpNewReno"
//#define TCP_PROTOCOL     "ns3::TcpCubic"

// For logging.

NS_LOG_COMPONENT_DEFINE ("main");
////////////////////////////////////////////////
int main (int argc, char *argv[]) {

  ////////////////////////////////////////
  // Turn on logging for this script.
  // Note: for BBR', other components that may be
  // of interest include "TcpBbr" and "BbrState".
  LogComponentEnable("main", LOG_LEVEL_INFO);

  //  enable sript printing
  CommandLine cmd;
  cmd.Parse (argc, argv);

  ////////////////////////////////////////
```

```
// Setup environment
Config::SetDefault("ns3::TcpL4Protocol::SocketType",
                   StringValue(TCP_PROTOCOL));

// Report parameters.
NS_LOG_INFO("TCP protocol: " << TCP_PROTOCOL);
NS_LOG_INFO("Server to Router Bwdth: " << S_TO_R_BW);
NS_LOG_INFO("Server to Router Delay: " << S_TO_R_DELAY);
NS_LOG_INFO("Router to Client Bwdth: " << R_TO_C_BW);
NS_LOG_INFO("Router to Client Delay: " << R_TO_C_DELAY);
NS_LOG_INFO("Router to Router Bwdth: " << R_TO_R_BW);
NS_LOG_INFO("Router to Router Delay: " << R_TO_R_DELAY);
NS_LOG_INFO("Packet size (bytes): " << PACKET_SIZE);

// Set segment size (otherwise, ns-3 default is 536).
Config::SetDefault("ns3::TcpSocket::SegmentSize",
                   UintegerValue(PACKET_SIZE));

// Turn off delayed ack (so, acks every packet).
// Note, BBR' still works without this.
Config::SetDefault("ns3::TcpSocket::DelAckCount", UintegerValue(0));

////////////////////////////////////////
// Create nodes.
NS_LOG_INFO("Creating nodes.");
NodeContainer nodes;
// n0=source, n3=source#2, n7=source#3
// n1=router, n5=router#2
// n2=sink, n4=sink#2, n6=sink#3
nodes.Create(8);

////////////////////////////////////////
// Create channels.
NS_LOG_INFO("Creating channels.");
NodeContainer n0_to_n1 = NodeContainer(nodes.Get(0), nodes.Get(1));
//NodeContainer n1_to_n2 = NodeContainer(nodes.Get(1), nodes.Get(2));
NodeContainer n3_to_n1 = NodeContainer(nodes.Get(3), nodes.Get(1));
NodeContainer n1_to_n5 = NodeContainer(nodes.Get(1), nodes.Get(4));
NodeContainer n5_to_n2 = NodeContainer(nodes.Get(5), nodes.Get(2));
```

```
NodeContainer n5_to_n4 = NodeContainer(nodes.Get(5), nodes.Get(4));
NodeContainer n7_to_n1 = NodeContainer(nodes.Get(7), nodes.Get(1));
NodeContainer n5_to_n6 = NodeContainer(nodes.Get(5), nodes.Get(6));

////////////////////////////////////////
// Create links.
NS_LOG_INFO("Creating links.");

// n0 to n1
int mtu = 1500;
PointToPointHelper p2p;
p2p.SetDeviceAttribute("DataRate", StringValue (S_TO_R_BW));
p2p.SetChannelAttribute("Delay", StringValue (S_TO_R_DELAY));
p2p.SetDeviceAttribute ("Mtu", UintegerValue(mtu));
NetDeviceContainer devices1 = p2p.Install(n0_to_n1);

// n3 to n1
p2p.SetDeviceAttribute("DataRate", StringValue (S_TO_R_BW));
p2p.SetChannelAttribute("Delay", StringValue (S_TO_R_DELAY));
p2p.SetDeviceAttribute ("Mtu", UintegerValue(mtu));
NetDeviceContainer devices2 = p2p.Install(n3_to_n1);

// n7 to n1
p2p.SetDeviceAttribute("DataRate", StringValue (S_TO_R_BW));
p2p.SetChannelAttribute("Delay", StringValue (S_TO_R_DELAY));
p2p.SetDeviceAttribute ("Mtu", UintegerValue(mtu));
NetDeviceContainer devices5 = p2p.Install(n7_to_n1);

//n1 to n5
p2p.SetDeviceAttribute("DataRate", StringValue (R_TO_R_BW));
p2p.SetChannelAttribute("Delay", StringValue (R_TO_R_DELAY));
p2p.SetDeviceAttribute ("Mtu", UintegerValue(mtu));
NS_LOG_INFO("Router queue size: "<< QUEUE_SIZE);
p2p.SetQueue("ns3::DropTailQueue",
             "Mode", StringValue ("QUEUE_MODE_PACKETS"),
             "MaxPackets", UintegerValue(QUEUE_SIZE));
NetDeviceContainer routers = p2p.Install(n1_to_n5);

// n5 to n2
```

```
p2p.SetDeviceAttribute("DataRate", StringValue (R_TO_C_BW));
p2p.SetChannelAttribute("Delay", StringValue (R_TO_C_DELAY));
p2p.SetDeviceAttribute ("Mtu", UintegerValue(mtu));
NS_LOG_INFO("Router queue size: "<< QUEUE_SIZE);
p2p.SetQueue("ns3::DropTailQueue",
             "Mode", StringValue ("QUEUE_MODE_PACKETS"),
             "MaxPackets", UintegerValue(QUEUE_SIZE));
NetDeviceContainer devices3 = p2p.Install(n5_to_n2);

// n5 to n4
p2p.SetDeviceAttribute("DataRate", StringValue (R_TO_C_BW));
p2p.SetChannelAttribute("Delay", StringValue (R_TO_C_DELAY));
p2p.SetDeviceAttribute ("Mtu", UintegerValue(mtu));
NS_LOG_INFO("Router queue size: "<< QUEUE_SIZE);
p2p.SetQueue("ns3::DropTailQueue",
             "Mode", StringValue ("QUEUE_MODE_PACKETS"),
             "MaxPackets", UintegerValue(QUEUE_SIZE));
NetDeviceContainer devices4 = p2p.Install(n5_to_n4);

// n5 to n6
p2p.SetDeviceAttribute("DataRate", StringValue (R_TO_C_BW));
p2p.SetChannelAttribute("Delay", StringValue (R_TO_C_DELAY));
p2p.SetDeviceAttribute ("Mtu", UintegerValue(mtu));
NS_LOG_INFO("Router queue size: "<< QUEUE_SIZE);
p2p.SetQueue("ns3::DropTailQueue",
             "Mode", StringValue ("QUEUE_MODE_PACKETS"),
             "MaxPackets", UintegerValue(QUEUE_SIZE));
NetDeviceContainer devices6 = p2p.Install(n5_to_n6);

////////////////////////////////////////
// Install Internet stack.
NS_LOG_INFO("Installing Internet stack.");
InternetStackHelper internet;
internet.Install(nodes);

////////////////////////////////////////
// Add IP addresses.
NS_LOG_INFO("Assigning IP Addresses.");
Ipv4AddressHelper ipv4;
```

```
ipv4.SetBase("10.0.0.0", "255.255.255.0");
Ipv4InterfaceContainer i0i1 = ipv4.Assign(devices1);

ipv4.SetBase("10.3.0.0", "255.255.255.0");
Ipv4InterfaceContainer i3i1 = ipv4.Assign(devices2);

ipv4.SetBase("10.7.0.0", "255.255.255.0");
Ipv4InterfaceContainer i7i1 = ipv4.Assign(devices5);

ipv4.SetBase("191.168.1.0", "255.255.255.0");
Ipv4InterfaceContainer i1i5 = ipv4.Assign(routers);

ipv4.SetBase("10.2.0.0", "255.255.255.0");
Ipv4InterfaceContainer i5i2 = ipv4.Assign(devices3);

ipv4.SetBase("10.4.0.0", "255.255.255.0");
Ipv4InterfaceContainer i5i4 = ipv4.Assign(devices4);

ipv4.SetBase("10.6.0.0", "255.255.255.0");
Ipv4InterfaceContainer i5i6 = ipv4.Assign(devices6);


Ipv4GlobalRoutingHelper::PopulateRoutingTables();

//////////////////////////////////////
// Create apps.
NS_LOG_INFO("Creating applications.");
NS_LOG_INFO("  Bulk send.");

// Well-known port for server.
uint16_t flow1port = 911;
//uint16_t flow2port = 912;

// Source n0.
// first sending flow
BulkSendHelper source("ns3::TcpSocketFactory",
                      InetSocketAddress(i5i2.GetAddress(1), flow1port));
// Set the amount of data to send in bytes (0 for unlimited).
source.SetAttribute("MaxBytes", UintegerValue(0));
```

```
source.SetAttribute("SendSize", UintegerValue(PACKET_SIZE));
ApplicationContainer apps = source.Install(nodes.Get(0));

// Source n3
// second sending flow
BulkSendHelper source2("ns3::TcpSocketFactory",
                       InetSocketAddress(i5i4.GetAddress(1), flow1port));
source2.SetAttribute("MaxBytes", UintegerValue(0));
source2.SetAttribute("SendSize", UintegerValue(PACKET_SIZE));
apps = source2.Install(nodes.Get(3));

// Source n7
// third sending flow
BulkSendHelper source3("ns3::TcpSocketFactory",
                       InetSocketAddress(i5i6.GetAddress(1), flow1port));
source3.SetAttribute("MaxBytes", UintegerValue(0));
source3.SetAttribute("SendSize", UintegerValue(PACKET_SIZE));
apps = source3.Install(nodes.Get(7));

apps.Start(Seconds(START_TIME));
apps.Stop(Seconds(STOP_TIME));

// Sink (at end nodes).
// sink for flow 1
PacketSinkHelper sink("ns3::TcpSocketFactory",
                       InetSocketAddress(Ipv4Address::GetAny(), flow1port));
apps = sink.Install(nodes.Get(2));
apps.Start(Seconds(START_TIME));
apps.Stop(Seconds(STOP_TIME));
Ptr<PacketSink> p_sink = DynamicCast<PacketSink> (apps.Get(0));

// sink for flow 2
PacketSinkHelper sink2("ns3::TcpSocketFactory",
                       InetSocketAddress(Ipv4Address::GetAny(), flow1port));
apps = sink2.Install(nodes.Get(4));
apps.Start(Seconds(START_TIME));
apps.Stop(Seconds(STOP_TIME));
Ptr<PacketSink> p_sink2 = DynamicCast<PacketSink> (apps.Get(0));
```

```
// sink for flow 3
PacketSinkHelper sink3("ns3::TcpSocketFactory",
                       InetSocketAddress(Ipv4Address::GetAny(), flow1port));
apps = sink3.Install(nodes.Get(6));
apps.Start(Seconds(START_TIME));
apps.Stop(Seconds(STOP_TIME));
Ptr<PacketSink> p_sink3 = DynamicCast<PacketSink> (apps.Get(0));

////////////////////////////////////////
// Setup tracing (as appropriate).
if (ENABLE_TRACE) {
  NS_LOG_INFO("Enabling trace files.");
  AsciiTraceHelper ath;
  p2p.EnableAsciiAll(ath.CreateFileStream("
  ./results/top4-3v3-dumbbell/3v3-dumbbell-newreno.tr"));
}
if (ENABLE_PCAP) {
  NS_LOG_INFO("Enabling pcap files.");
  p2p.EnablePcapAll("shark", true);
}

////////////////////////////////////////
// Run simulation.
NS_LOG_INFO("Running simulation.");
Simulator::Stop(Seconds(STOP_TIME));
NS_LOG_INFO("Simulation time: [" <<
            START_TIME << "," <<
            STOP_TIME << "]");
NS_LOG_INFO("---------------- Start ----------------------");
Simulator::Run();
NS_LOG_INFO("---------------- Stop -----------------------");

////////////////////////////////////////
// Ouput stats.
// Flow 1
NS_LOG_INFO(" ");
NS_LOG_INFO("Flow1.");
NS_LOG_INFO("Total bytes received: " << p_sink->GetTotalRx());
double tput = p_sink->GetTotalRx() / (STOP_TIME - START_TIME);
```

```
  tput *= 8;          // Convert to bits.
  tput /= 1000000.0;  // Convert to Mb/s
  NS_LOG_INFO("Throughput: " << tput << " Mb/s");

  // Flow 2
  NS_LOG_INFO(" ");
  NS_LOG_INFO("Flow2.");
  NS_LOG_INFO("Total bytes received: " << p_sink2->GetTotalRx());
  double tput2 = p_sink2->GetTotalRx() / (STOP_TIME - START_TIME);
  tput2 *= 8;          // Convert to bits.
  tput2 /= 1000000.0;  // Convert to Mb/s
  NS_LOG_INFO("Throughput: " << tput2 << " Mb/s");

  // Flow 3
  NS_LOG_INFO(" ");
  NS_LOG_INFO("Flow3.");
  NS_LOG_INFO("Total bytes received: " << p_sink3->GetTotalRx());
  double tput3 = p_sink3->GetTotalRx() / (STOP_TIME - START_TIME);
  tput3 *= 8;          // Convert to bits.
  tput3 /= 1000000.0;  // Convert to Mb/s
  NS_LOG_INFO("Throughput: " << tput3 << " Mb/s");

  // Done.
  NS_LOG_INFO(" ");
  NS_LOG_INFO("---------------- Destroy ------------------------");
  Simulator::Destroy();
  NS_LOG_INFO("Done.");
  return 0;
}
```

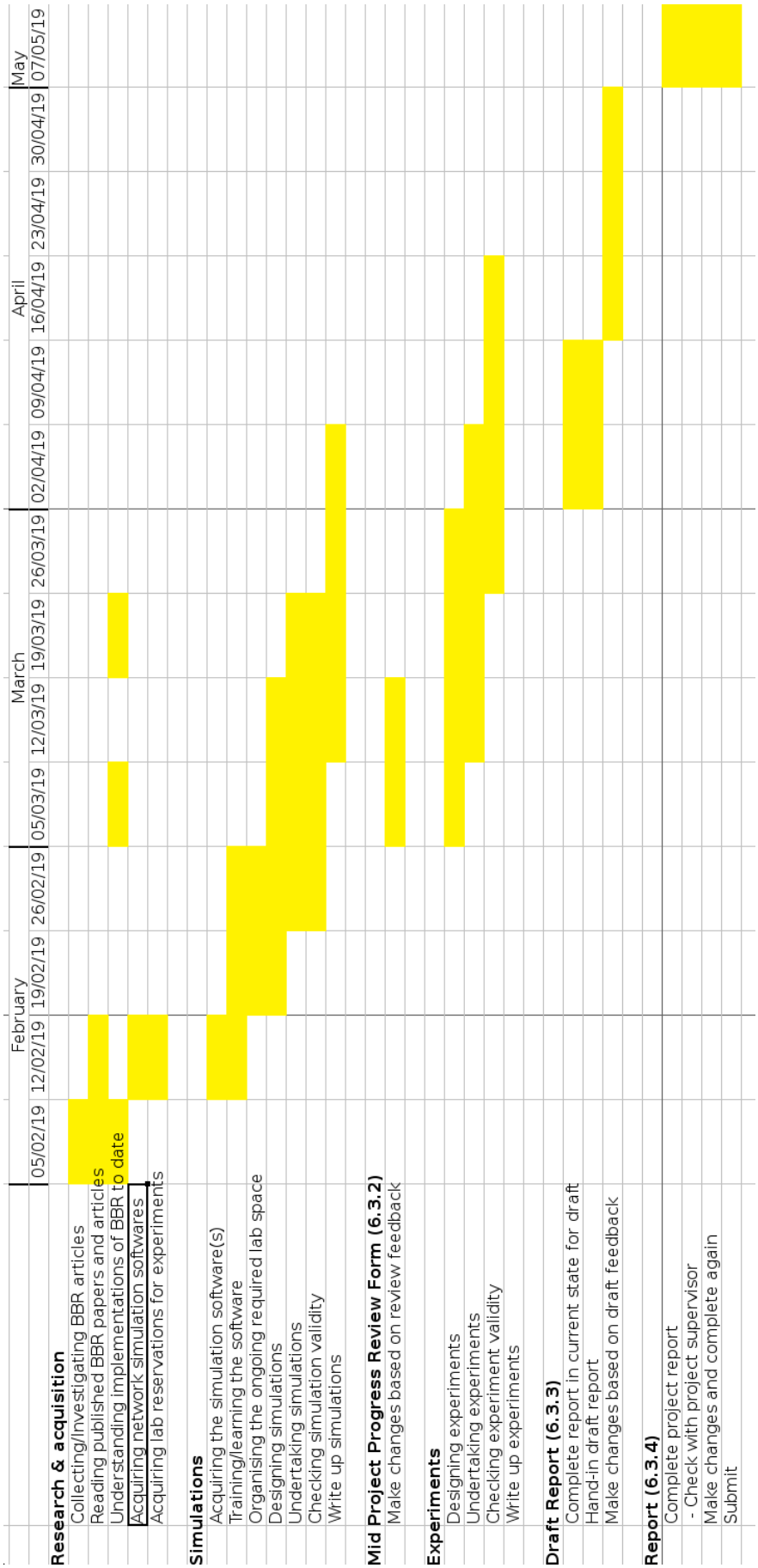


Figure 1: Original plan time sheet

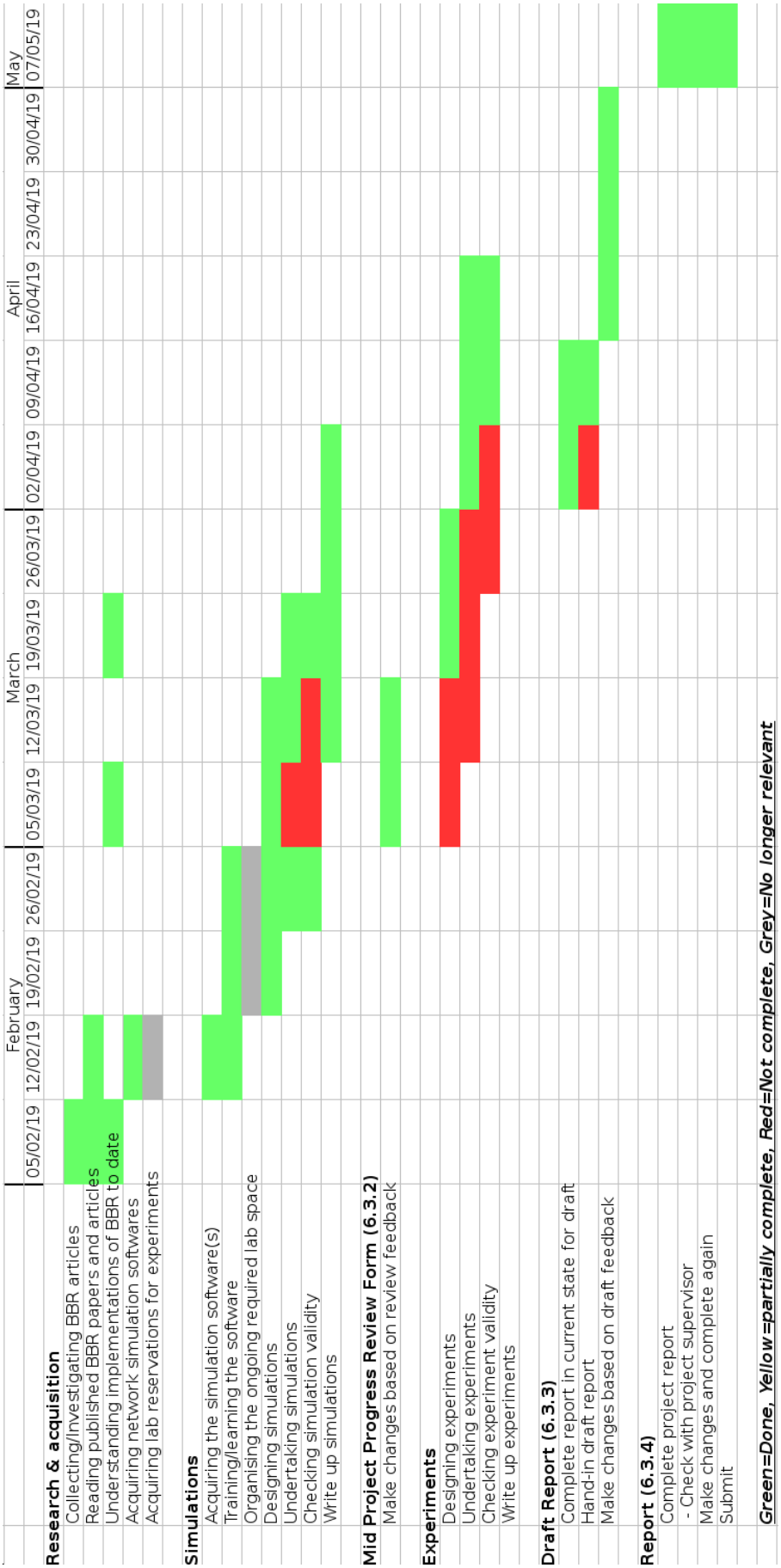


Figure 2: Updated time sheet