Wireless Interference and Multipath TCP

Barlow-Bignell, J, da Silva, C, Gjengset, J, and Oliha, P
Supervised by Professor Mark Handley

MSc in Networked Computer Systems
University College London

September 1, 2013

Abstract

In this report we evaluate the behaviour of Multipath TCP when using multiple WiFi interfaces on a single host, specifically with regard to how self-interference affects its ability to provide reliability and improved throughput.

Our results show that Multipath TCP can improve both reliability and throughput when using multiple WiFi links for both downlink and uplink traffic. In both cases, we show that on two idle wireless networks, Multipath TCP can achieve an aggregate throughput equal to the sum of both links. We also demonstrate how Multipath TCP behaves unfairly on the uplink when competing flows are present on any of the wireless networks.

Contents

1 Introduction

The Transport Control Protocol has been one the Internet's core technologies almost since its inception, providing reliable delivery of packets across unreliable networks. However, TCP was only designed to support flows going from a single interface on one host to a single interface on another host, meaning that it cannot effectively make use of the increasing connectivity of modern devices.

Multipath TCP is a recently standardised technology that attempts to solve this problem. It extends TCP so that a single TCP connection can be distributed across any number of distinct physical links, allowing it to both provide load balancing, fail-over, and in some cases, increased throughput. It has been tested extensively on wired networks and on hosts with WiFi and 3G interfaces, but is mostly untested on devices connected to multiple WiFi networks. WiFi networks present an additional challenge as links may interfere with each other, and it is not immediately clear that putting traffic on one network will not severely impact other networks.

In this report, we evaluate the behaviour of Multipath TCP when using multiple WiFi interfaces on a single host; specifically with regard to how self-interference affects its ability to provide reliability and improved throughput. We also discuss how fair the protocol is to other clients on the WiFi networks it uses, as this is a major design goal of Multipath TCP's Coupled congestion control.

We first give some background information about TCP, WiFi networks and Multipath TCP in §2, followed by an explanation of how we performed our experiments in §3. The main body of the report is in §4, where we present our results and discuss their implications for Multipath TCP. Finally, §5 discusses our findings as a whole, as well as pointing out interesting areas for future work.

This report, our test data, and the scripts outlined in §3.4 are all available online at `http://jon.tsp.io/mscncs/`.

2 Background

2.1 TCP

The Transmission Control Protocol (TCP) is one of the core Internet protocols. It provides reliable, in-order delivery of data between two systems, and operates in full-duplex, meaning it allows data to flow in both directions simultaneously. TCP is a packet-based protocol, and uses positive acknowledgement of packets to signal successful delivery. It also provides flow and congestion control, which will be explained later in this section.

After establishing a connection between two communicating parties, TCP allows each host to pass data to it, which is then split into packets and sent over the network to the other host. The Maximum Segment Size (MSS) of a connection limits how much data TCP is allowed to put into a single packet, and TCP generally tries to ensure that every packet is exactly one MSS large to avoid sending more packets than strictly necessary. To avoid overloading the receiving host or the network, TCP uses two rate control mechanisms: flow control and congestion control.

Flow control is used to avoid sending data at a faster rate than the receiving host is able to process it. Every packet a host sends out contains an indication of the amount of buffer space available at the host, and the sender must ensure that it does not transmit data that the receiver does not have room for. This is implemented using a sliding window algorithm where each host limits the number of packets in flight at any given time, making sure to never put more data on the network than the receiver says it can handle.

Congestion control aims to prevent a TCP flow from sending faster than the network can support. Upon receiving packets, routers and other devices in a network must assign some buffer space for these incoming packets to process them before forwarding along the path, and to deal with any temporary slow-down of the next link. If a device in the network receives packets faster than it can retransmit them, this buffer will fill up, and eventually packets are dropped when the buffer is full. This packet loss indicates congestion, and TCP attempts to limit itself so that it does not congest any network device along the path, whilst using an equal, fair share of the available capacity compared to other competing flows.

The canonical congestion control algorithm is TCP New Reno[1]. This algorithm

[1]RFC 6582 - The NewReno Modification to TCP's Fast Recovery Algorithm

maintains a congestion window, similar to the receiver's flow control window, which specifies the number of packets which may be in flight concurrently without overloading the network. The sender must respect both the receive and congestion windows when deciding whether or not it may send a packet.

New Reno flows begin in a slow start phase. A small initial congestion window is chosen, and this is increased by one MSS for each packet acknowledgement received (also known as an ACK). This has the effect of increasing the congestion window size exponentially until a congestion event occurs. A congestion event is anything that indicates that a loss has occurred, and is either a timeout while waiting for an ACK, or the receipt of multiple duplicate ACKs.

Two mechanisms are used for backing off when congestion is detected. In the case of a missed ACK, the algorithm returns the congestion window to its small initial size and begins slow start again. Since the retransmission timeout in TCP must be fairly long to cater for high-latency links, a fast recovery mechanism is also implemented. When three duplicate ACKs are received, TCP considers this as another indication of congestion, and will halve its congestion window. This is called a fast retransmit, since it is intended to prevent TCP from having to wait for a timeout and re-enter slow start.

After a fast retransmit, New Reno runs in additive-increase mode, where the congestion window is increased by one MSS every round-trip time (RTT). This is implemented by the congestion window being increased by $1/cwnd$ for every received ACK, making it self-clocking. That is, TCP will not send data faster than the speed at which packets go through the bottleneck link in the path. As TCP probes for available capacity and backs off when it encounters congestion, a distinctive saw-tooth pattern of the congestion window size is typically produced.

2.2 WiFi and Interference

The 802.11 standards are a set of physical and Medium Access Control (MAC) layer specifications for implementing WiFi networks. 802.11 networks commonly operate on frequencies in the 2.4 GHz and 5 GHz bands, which are divided into a number of overlapping sub-bands, commonly known as channels. For example, the 802.11g channels in the 2.4 GHz band are 22 MHz wide and spaced 5 MHz apart, beginning with channel 1 on 2.412 GHz.

Since many of the 802.11 sub-bands are overlapping, transmissions on one channel can interfere with signals transmitted on neighbouring channels. Given that

the 2.4 GHz and 5 GHz bands are not specifically reserved for WiFi, they are also susceptible to noise from other devices such as microwave ovens and Bluetooth-enabled electronics. Wireless networks are therefore notoriously volatile, and performance can vary drastically from one location to another, and from one minute to the next.

A typical WiFi deployment consists of several "stations" connected to the network via an access point (AP). A station in 802.11 is typically a single wireless network interface card (or NIC), and a single machine can have multiple such NICs, and would thus appear as multiple stations to the AP.

The greater the number of stations connected to an AP, the higher the probability that the stations would interfere with each other as they contend for the access to the medium. Multiple stations transmitting simultaneously on interfering channels is undesirable, as it leads to corrupted packets, and thus retransmissions; lowering network utilization and consequently throughput. To prevent this, 802.11 implements carrier sense and random back-off. Before transmitting, a station will first listen to determine if another transmission is already in progress. If the medium is busy, the station will defer for a random period of time and retry. This behaviour is also used when multiple stations begin transmitting at the same time, and collide. The random back-off selected by each station reduces the probability of the stations interfering with each other on the next retry. Ideally, stations should not carrier sense, and thus defer to, stations transmitting on non-overlapping channels as this would prevent stations that could transmit simultaneously from doing so.

Carrier sense will eventually allow a single station to transmit while the others are waiting, but which station this will be is effectively random. All stations that have data to send have an equal chance of being allowed to use the medium, and this is what leads to 802.11 fairly distributing access to the wireless medium per station.

The 802.11 MAC layer also implements its own packet acknowledgements in addition to those provided by TCP. Acknowledgement packets are sent immediately after each data frame is successfully received, in a reserved time slot where no station may transmit. A station will generally retry transmitting a fixed number of times without receiving acknowledgement before dropping a packet.

This is implemented because WiFi links typically see a high loss rate unrelated to congestion. If this loss was not hidden from TCP, it would be treated as indication of congestion, and thus trigger TCP's congestion avoidance mechanisms,

significantly reducing throughput.

2.3 Multipath TCP

Devices with multiple network interfaces are becoming increasingly common. Many consumer smart phones have both WiFi and 3G interfaces, and data centre networks are often deployed such that equipment racks are connected through multiple paths. Data centres themselves are often multihomed to improve reliability, meaning that they have several points of access to the wider Internet. Multipath TCP[2] is an extension to TCP recently standardised by the IETF, which aims to improve redundancy and throughput by taking advantage of multiple paths for a single TCP flow. In order to achieve this, Multipath TCP adds additional "subflows" to a TCP connection so that data flows between every pair of network interfaces of each host, taking advantage of potentially independent paths.

The authors of Multipath TCP give three main design goals when discussing the Coupled congestion control algorithm used with Multipath TCP[3]:

improve throughput A Multipath TCP flow should perform at least as well as a single-path TCP connection would perform.

do no harm to other network users A Multipath TCP flow should not take up more capacity on any one path than if it was a single path flow using only that route – this is particularly relevant for shared bottlenecks.

balance network resources A Multipath TCP flow should balance congestion by moving traffic away from the most congested paths.

Coupled thus attempts to behave fairly when links are shared by other TCP or Multipath TCP flows. In particular, a set of Coupled subflows should not gain more throughput than a competing TCP flow. However, the Coupled algorithm also aims to utilise the available links fully when they would otherwise be idle, meaning no other competing flows are active. This is the case where Coupled can lead to improved throughput over regular TCP.

The Coupled algorithm maintains a separate congestion window for each subflow and uses the same congestion avoidance mechanisms as TCP New Reno, but links the additive increase across all subflows to ensure fairness. For each packet

[2]RFC 6824 - TCP Extensions for Multipath Operation with Multiple Addresses.
[3]RFC 6356 - Coupled Congestion Control for Multipath Transport Protocols.

acknowledgement received on subflow i, the congestion window for that subflow is increased as shown in Equation 1.

$$cwnd_i = cwnd_i + \min\left(\frac{\alpha}{cwnd_{\text{total}}}, \frac{1}{cwnd_i}\right) \tag{1}$$

$$\alpha = \frac{cwnd_{\text{total}} \cdot \max_i\left(\frac{cwnd_i}{rtt_i^2}\right)}{(\sum_i \frac{cwnd_i}{rtt_i})^2} \tag{2}$$

The parameter α specified in Equation 1 is calculated as shown in Equation 2, and controls how aggressive Coupled should be when increasing its total send rate. It is chosen such that the aggregate throughput across all subflows is equal to the throughput a TCP New Reno flow would gain on the best of the paths available. This ensures fairness with competing TCP flows when there is a shared bottleneck in the network. Additionally, the algorithm forces the congestion window of more congested links to increase at a slower rate than less congested links, which has the effect of shifting traffic onto less congested paths; effectively balancing congestion in the network.

If the available links are idle, the Coupled algorithm will allow each subflow to use the full capacity available to it. For example, with two idle links the aggregate throughput of a Multipath TCP flow should be the capacity of both links combined.

Understanding the intuition behind *why* Coupled works can be difficult solely from looking at the equation above. We will therefore try to give a less formal explanation of what Coupled is doing. As the throughput of a Coupled flow approaches the throughput a New Reno flow would get on the best link available to it, Coupled gradually decreases the aggressiveness (α) of all subflows. Doing so effectively decreases the growth rate of each flow's congestion window. The congestion window of each subflow will continue to increase, gradually filling the pipe, but it will do so more slowly than New Reno. This means that should any other flow appear with a growth rate closer to that of New Reno, its window will grow faster than the Coupled flow's window. When a congestion event occurs, the window size of both the competing flow and the Coupled flow will be halved, but since the competing flow is increasing it's window size faster, it will gain some of the available capacity previously held by the Coupled flow. This process will continue until Coupled detects that it is at risk of performing worse than regular TCP, at

which point it will go back to increasing its congestion window as aggressively as New Reno. As this happens, the flows reach an equilibrium where they both increase their congestion windows equally fast, and thus end up sharing the link evenly.

2.4 Motivation

Multipath TCP is undoubtedly a useful extension to regular TCP, but it was primarily designed for wired networks, in which links are independent, and has been demonstrated to perform well in such configurations[4].

The nature of wireless means that interfaces can interfere with each other, which is something that Multipath TCP was not designed to handle. If Multipath TCP tries to use multiple interfaces at the same time, this self-interference may prove sufficiently strong that the benefits of Multipath TCP are effectively negated.

In this paper, we aim to investigate the extent to which wireless networks interfere with each other, and how Multipath TCP behaves when used with non-independent interfaces. To explore this, we analyse its behaviour in a series of wireless experiments. Our results are presented in §4.

[4]C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP, *ACM Sigcomm 2011*

3 Methodology

In order to establish how well Multipath TCP works in its default configuration with multiple WiFi interfaces, we performed a number of experiments to measure its behaviour in various wireless environments. This section details how the project was organized and executed.

We first outline the project management strategy used in §3.1, the testing setups used for the experiments in §3.2, and the equipment and tools used in §3.3.

§3.4 then gives an overview of the different scripts used for running, analysing and graphing tests. Finally, §3.5 describes the different data sources used in our results.

3.1 Project management strategy

As we had limited time to complete the project, we decided on a small set of initial experiments to run. These experiments were intended to cover a wide range of test cases at a high level, and allowed us to determine which areas would be particularly interesting for us to explore further. We also used these tests to better understand the data we were seeing, and fine tune our test setups and metrics. Once we determined the cases we would investigate more in-depth, we planned sets of related tests which could each be completed over a few days to a week.

Due to the nature of the experiments, the majority of our time was spent running tests with all group members present. This also involved a great deal of discussion about our results. We dedicated time each week to meet more formally to discuss our progress, and to decide what direction we would take with future experiments. During this time, we also met regularly with our supervisor to discuss progress and our interpretation of results.

We initially separated this report into sections, which were distributed to each group member. We then worked individually on our allocated sections, before we rotated them for revision by the other group members. Each section was rotated and revised many times during the final write up, until all the group members had reviewed every section, and everyone was satisfied with the final document. We used video calling during this time to discuss changes to the report.

All test data from our experiments was shared using the Dropbox file sharing service, which allowed us to distribute the test results in a simple way. This also provided us with automatic backups of the test data. The various test scripts, and the report itself, were tracked using the git version control system. This allowed us

to work on different sections simultaneously and keep a history of changes, which could be visited to quickly review edits made by the other group members.

3.2 Experiment setups

The various experiment configurations we used are described in this section. Each test was generally carried out using both overlapping and non-overlapping pairs of channels in the 2.4 GHz band, and with one channel in the 2.4 GHz band and another in the 5 GHz band.

The 2.4 GHz band is more prevalent in real world deployments and so, due to time constraints, we did not perform any tests using multiple channels in the 5 GHz band only.

3.2.1 Physical environment

In all but the mobility tests in §3.2.5, the wireless clients were placed next to each other approximately 60cm apart. The APs were located 2-3 metres away, and separated by about 30cm.

All tests were run in the Computer Science department building at UCL. The building environment consists mainly of glass or plaster walls and concrete floors, and several other wireless networks were present. Other wireless experiments were also being conducted elsewhere in the building, but we do not believe these significantly affected our results.

3.2.2 Sequential TCP tests

This test setup consisted of running five tests consecutively, and then plotting all of the results in a single graph. For each test, a TCP connection was initiated from both clients to a local test server, and data was pushed through the connection as fast as the congestion control mechanism would allow. Two APs were used; one hosting network A and one hosting network B. The five tests were as follows:

1. Machine A connected to network A.
2. Machine B connected to network B.
3. Machine A connected to network A and machine B connected to network B.
4. Machine A connected to both networks.

5. Machine B connected to both networks.

The two first tests were primarily used as baselines for the other tests, to determine how well each network performed in isolation. We will refer to these as single tests.

The parallel test was used to evaluate how well two wireless clients would perform when transmitting simultaneously on each network. This was done in order to provide a reference point for how two entirely separate, regular TCP clients would utilise the two networks.

The fourth and fifth tests using multipath flows were performed with a single machine connected to both networks at the same time. These tests allowed us to determine how Multipath TCP behaves with multiple WiFi links.

Running this last configuration on both machines may seem unnecessary, but it was vital to allow us to compare the performance of a parallel test on a given channel to the Multipath TCP performance on the same channel. The reason for this is that wireless performance can vary greatly between different machines and locations. Running the dual-network test on both machines allowed us to compare the use of network A for single-flow, parallel-flow and Multipath TCP without worrying about these machine discrepancies. Note that a side-effect of this is that the Multipath TCP lines in a sequential test graph are **not** produced from the same run, and thus would not be expected to match up entirely. That said, we usually observed similar behaviour for Multipath TCP for the last two tests.

Because of this discrepancy, all of the results presented in this report which include Multipath TCP, with the exception of §4.5, are from the simultaneous TCP test setup described in the following section.

3.2.3 Simultaneous TCP tests

The simultaneous TCP tests used two APs and three wireless clients. One client used two wireless interfaces with one connected to each network. The two remaining clients were located on either side, and were connected to either of the two networks.

The test itself was a simple TCP streaming test as used in the sequential TCP tests, but running on all three clients simultaneously. This setup allowed us to more directly compare the performance of a Multipath TCP client to that of two simultaneous clients, as well as evaluate the fairness of Multipath TCP when other clients were using the network at the same time.

This test also avoids the cross-host comparison issue exhibited by the sequential tests, and consequently we mainly include results from the simultaneous tests in this report as they are more reliable.

3.2.4 UDP interrupt tests

UDP interrupt tests were run with a single client using two wireless interfaces. Each wireless interface was connected to one of the APs as in the other setups, but initially data was only transmitted on one interface. The other interface was alternated between being idle and active for fixed time slots of 30 seconds during the experiment.

The idea behind switching one interface on and off is that, ideally, if there is no interference between these networks, there should be almost no loss of throughput for the always-on interface when the alternating interface is active. If there is interference, such as with networks on the same 2.4 GHz channel, the throughput of the always-on interface should drop significantly while the other interface is active.

To avoid having the throughput be limited by factors other than the wireless medium, such as congestion control, these interrupt tests were run with UDP rather than TCP.

3.2.5 Mobility tests

For the mobility tests in §4.6, we positioned our two 2.4 GHz APs such that each AP was 4 meters from a corridor intersection. A client standing at the intersection thus had line of sight to both APs, whilst the APs themselves were isolated from each other. To reduce the range covered by each AP, their transmit power was turned down considerably so that a client located at one could barely hear the other. We then connected a client laptop to both APs and walked from one AP to the other, making a stop of 30 seconds at either AP and at the intersection. We spent approximately 10 seconds walking between the intersection and an AP to give Multipath TCP enough time to adapt to the changing environment.

3.3 Equipment and software

The WiFi interfaces used in our experiments were 2.4 GHz Wi-Pi dongles, commonly found in Raspberry Pi devices. For the 5 GHz tests, a Tenda W522U was

used. These dongles proved somewhat unreliable and would occasionally drop substantial amounts of packets or fail altogether during a test, particularly with high amounts of uplink traffic. For some experiments we were therefore forced to use the built-in wireless interfaces on our laptops to perform the tests. These exhibited more stable loss rates and generally did not fail during experiments, but also made comparing results across tests more difficult.

Our APs were two 2.4 GHz Netgear ProSafe WG103, as well as a Netgear N600 which was used exclusively for 5 GHz tests. All tests were performed using 802.11g with WPA2 encryption enabled.

The test servers were two dedicated machines connected to the UCL internal network using a gigabit switch. Since the APs were all 100 megabit only, there should be no bottleneck in the network beyond the APs. We used a variety of personal laptops as wireless clients.

To run the tests, the NetPerf performance testing tool was used to generate TCP or UDP traffic, the netem module for qdisc allowed us to emulate network delays, and the `ss` tool provided invaluable TCP socket information.

At the time of these experiments, both the servers and client laptops were running the most recent version of the Multipath TCP kernel, version 0.87, which is based on Linux 3.10.

3.4 Scripts

In order to automate common tasks such as running tests and analysing data, we developed several scripts which were then used throughout the experiments. The most interesting ones are outlined below.

Note that many of these scripts perform magic based on what wireless networks the host machine is connected to. For this paper, two servers were used, named fry and zoidberg, and three wireless networks were set up: bender-wifi (5 GHz), fry-wifi and zoidberg-wifi (both 2.4 GHz). Every test involved at least one of fry-wifi and zoidberg-wifi, and some of the scripts below use the presence of a connection to one of them as an indicator of which server should be used for tests.

mp-start and mp-congestion These two scripts enable Multipath TCP on the current machine, as well as set the appropriate congestion control algorithm on both the local machine and any remote machines it might be connected to. `mp-start` also stops any other wireless connections as well as some common

network-intensive applications such as Dropbox to prevent locally generated traffic from interfering with the tests.

mp-routes Examines the IP addresses of all active network interfaces and sets up routing tables according to the Multipath TCP configuration instructions[5].

mp-run The primary testing script for our experiments. First, logs information about test location, connected networks, nearby wireless networks, TCP configuration parameters and kernel version, amongst other details. It then starts the `mp-stats` logging daemon to record state information during the experiments, before running NetPerf for a configurable period of time. When the NetPerf test is done, the logging daemon is stopped, and any large log files are compressed. `mp-run` also supports downlink tests by spawning a local NetPerf server and running the NetPerf client on one of the server machines.

mp-stats Collects the majority of statistics during a test. By default it samples data every 500ms. It logs statistics from the wireless interfaces (signal strength, bit rate, retransmit failures), IP (bytes and packets sent) and TCP (queue sizes, RTT estimates, retransmits)

mp-int Works much the same as mp-run, but instead of running a TCP benchmark with NetPerf on all connected interfaces, it runs a UDP benchmark continuously on one interface and periodically on the other connected interface. This script implements the UDP interrupt tests described in §3.2.4.

mp-analyze Given a test directory created by `mp-run` or `mp-int`, `mp-analyze` extracts information from various log files and output a simple space-separated file for each interface (and a total) with values for everything from throughput to bit rate. This information is then used by `mp-plot` or `mp-cdf` to display graphs or other statistical information about the data.

mp-plot Given a test folder, `mp-plot` graphs every statistic generated by `mp-analyze` for every interface the given test was run with. It also performs scaling to keep all values in a 0-100 range. This will be discussed in §3.5.

mp-cdf Given tuples of test folders and APs, calculates the CDF for each corresponding interface in each test and graph them using gnuplot[6]. The script

[5]http://multipath-tcp.org/pmwiki.php/Users/ConfigureRouting
[6]http://www.gnuplot.info/

uses the statistical programming language R[7] to generate this CDF.

mp-set This script is a shortcut to avoid having to type repeated folder/AP names to plot certain data sets. It searches for all interfaces across tests connected to the same channel, and then plot each group of such interfaces using `mp-cdf`.

mp-merge Merges the data from several test sets into a single dataset.

mp-gather Simple wrapper around `mp-merge` that takes folders of test sets as arguments, extracts what WiFi networks were used and calls `mp-merge` for all related tests. For example, it can find all same-channel, coupled test sets in all its arguments, and merge them.

3.5 Metrics

To understand our results, it is important to know where the data comes from. During each test, the script `mp-stats` was run every 500ms and it was the primary source of data for every plot. The most interesting metrics are discussed below:

throughput This is calculated using the difference in the number of bytes sent on each interface (as reported by `/proc/net/dev`) and dividing it by the time since the last run of `mp-stats` using timestamps logged by `date +%s.%N`.

congestion window Retrieved from the command `ss -inot`, which shows TCP socket information. The congestion window size is reported as a number of packets, and so we multiply it by the MSS to get the real congestion window size in bytes.

roundtrip time TCP's RTT estimate is also extracted from the output of `ss`.

send queue size Printed by `ss`. This includes the size of any packets sent, but not yet ACKed by TCP. We discuss this further in §4.7.3.

TCP metrics (e.g. timeouts) This statistic is provided by `/proc/net/netstat` across all interfaces in a machine.

In order to make it easier to plot multiple metrics on a single graph, many of the graphs shown in this report are drawn using our `mp-plot` script, which scales

[7] http://www.r-project.org/

values to keep them in the range $[0, 100]$. This scaling is important to understand in order to interpret the results correctly. For example, throughput is measured in Mbps; utilization is a measure of what percentage of total throughput is sent through each interface and is displayed such that 80 is 0% and 100 is 100%, and congestion window and send queue are both plotted in kilobytes. The CDF graphs were drawn using the `mp-cdf` script which is detailed in §3.4.

3.6 A note on the reliability of wireless experiments

Running each test many times has proven very important throughout this project. Considering the uplink New Reno 2.4 GHz tests as an example, we initially suspected that Multipath TCP was consistently seeing slightly lower throughput compared to two separate flows in parallel, and we surmised that this might be because two WiFi interfaces connected to one machine might somehow lead to more cross-interface interference than with the same interfaces connected to different machines equally far apart. After running more experiments to confirm this, however, we quickly saw contradicting results where Multipath TCP was consistently faster than the parallel case. These kind of discrepancies were present in many of our tests, and are due to WiFi being extremely sensitive to positioning and timing. We therefore had to repeat many experiments to obtain consistent results.

4 Results and Evaluation

In the following sections, we will present the results of our experiments and evaluate how Multipath TCP performs on a host with multiple wireless interfaces.

§4.1 examines the effect of wireless interference on simultaneous clients irrespective of Multipath TCP. To determine to what extent interference affects the quality of the individual wireless links.

§4.2 presents the results of running a Multipath TCP connection over WiFi with New Reno congestion control, to investigate whether a Multipath TCP client with both wireless networks to itself can ever be expected to reach the same throughput as two regular TCP flows running on separate hosts on the different networks.

§4.3 evaluates the extent to which Multipath TCP behaves fairly when competing with other flows on the wireless networks. For the uplink case we also give an analysis of how and why Multipath TCP exhibits unfair behaviour in some situations. §4.4 then discusses a potential solution to this problem.

§4.5 examines the performance of Coupled Multipath TCP on idle paths, and tries to determine whether Coupled will ever allow a multipath flow fully utilize idle wireless links.

In §4.6, we present results showing how Multipath TCP reacts to changes in a mobile environment. Finally, in §4.7 we give some closing thoughts and explain several strange phenomenon we observed during our wireless experiments.

4.1 Simultaneous use of wireless interfaces

Before evaluating the performance of Multipath TCP on wireless links, we explored whether the nature of wireless permits transmitting concurrently across multiple wireless links at all. In particular, we wanted to determine:

1. Are there any inherent performance penalties when transmitting on two interfaces concurrently?

2. Can any performance benefit be gained when transmitting on two interfaces concurrently?

3. To what degree is the downlink performance affected by multiple clients using different networks simultaneously?

To answer these questions, we ran several experiments using UDP and TCP New Reno. The channel frequencies of the two networks were varied between

experiments. One network was set to channel 1 in the 2.4 GHz band, whilst the other was switched between channel 1 for same-channel tests, channel 5 or 11 for non-overlapping channel tests and a channel in the 5 GHz band for the dual band test.

4.1.1 Interference between 2.4 GHz channels

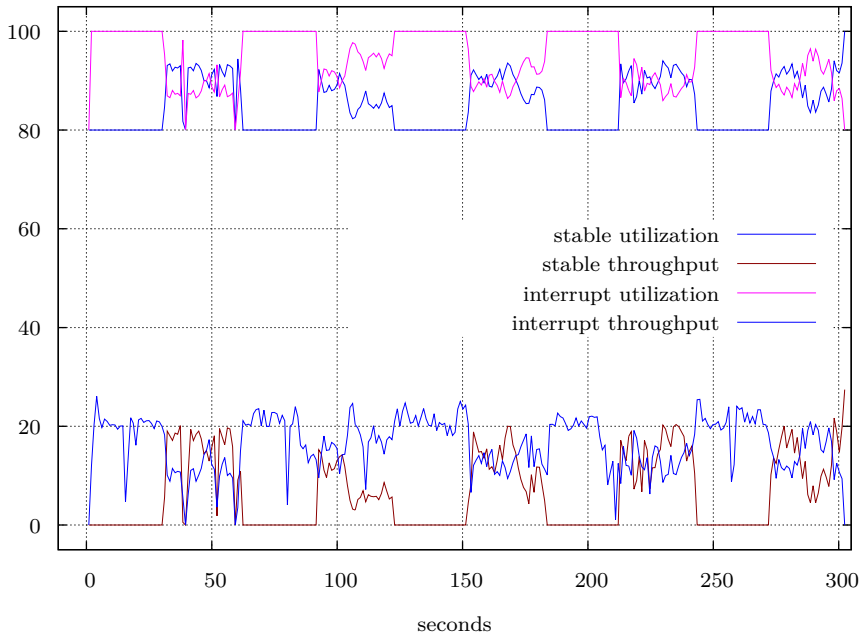


Figure 1: Non-overlapping channel interference (channels 1 and 11)

Figure 1 shows a UDP test as described in §3.2.4. The interference pattern observed here is quite interesting, as the utilization measurements show completely fair sharing of transmit time between the two networks when both interfaces are active. The wireless cards also report very few 802.11 retransmit failures.

As the two networks are operating at opposite ends of the 2.4 GHz spectrum, we would expect both interfaces to transmit without colliding, and the aggregate throughput to reach approximately the sum of the capacity of both links. The reason why this behaviour is expected is that as the networks are non-overlapping, they should not interfere with each other, and should be capable of transmitting concurrently. However, this perfect split of utilisation between the two networks implies that carrier sense is being used between the interfaces. Looking at the sum throughput, it is clear that although some performance gain is achieved, the

aggregate throughput is closer to 150% of that of a single interface than the 200% one would expect from completely independent channels.

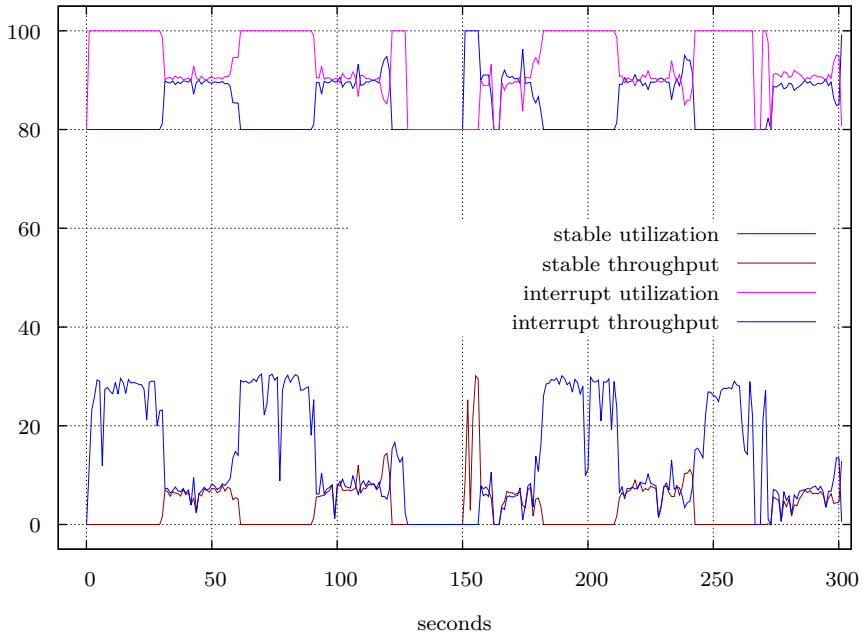


Figure 2: Same-channel interference

Figure 2 shows a similar test performed with both networks operating on the same channel in the 2.4 GHz band. The results here are perhaps even more surprising, as carrier sense should enforce fairly strict time multiplexing between two networks on the same channel. We would therefore expect the total throughput to remain unaffected, regardless of whether one or two interfaces are active. However, these results show that the **total** throughput decreases to almost 60% of that of a single link when both interfaces are active.

In both of the cases above, it seems as though carrier sense is not working quite as anticipated. We would generally expect few back-offs due to carrier sense between networks on non-overlapping channels, but clearly this is not the case; carrier sense is causing a significant number of back-offs, which shows that the non-overlapping channels are experiencing significant interference. Conversely, for networks on the same channel, we would expect carrier sense to time-multiplex fairly between the clients. Here, however, the tests show a high number of link layer transmit failures, suggesting that carrier sense is often failing to prevent collisions.

These results lead us to believe that using multiple wireless networks on the

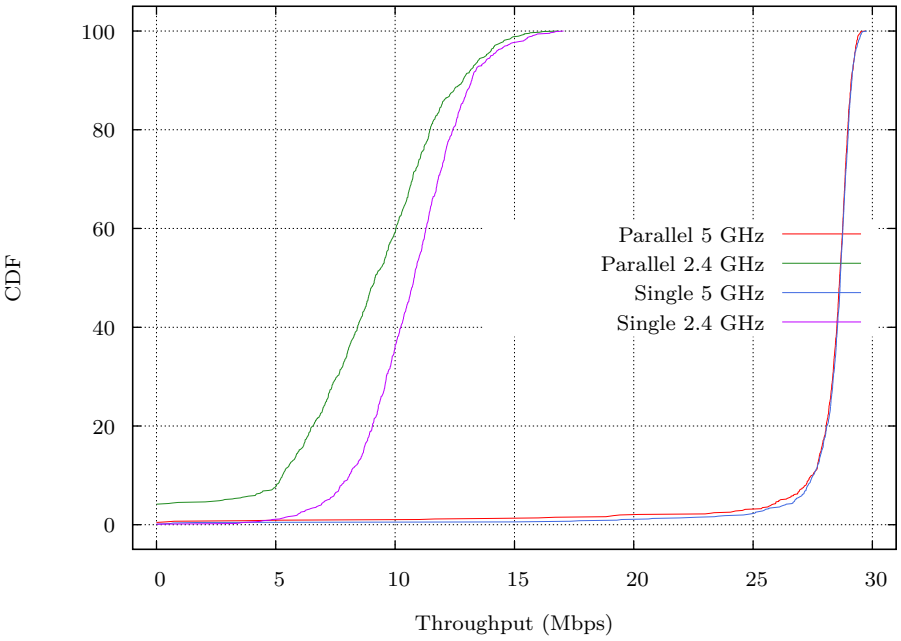
same 2.4 GHz channel may in fact lead to a performance penalty, as compared to using only a single interface. In the case where the networks are operating on non-overlapping 2.4 GHz channels, a performance benefit seems possible, although the gain may be less than one would reasonably expect.

4.1.2 Interference between bands

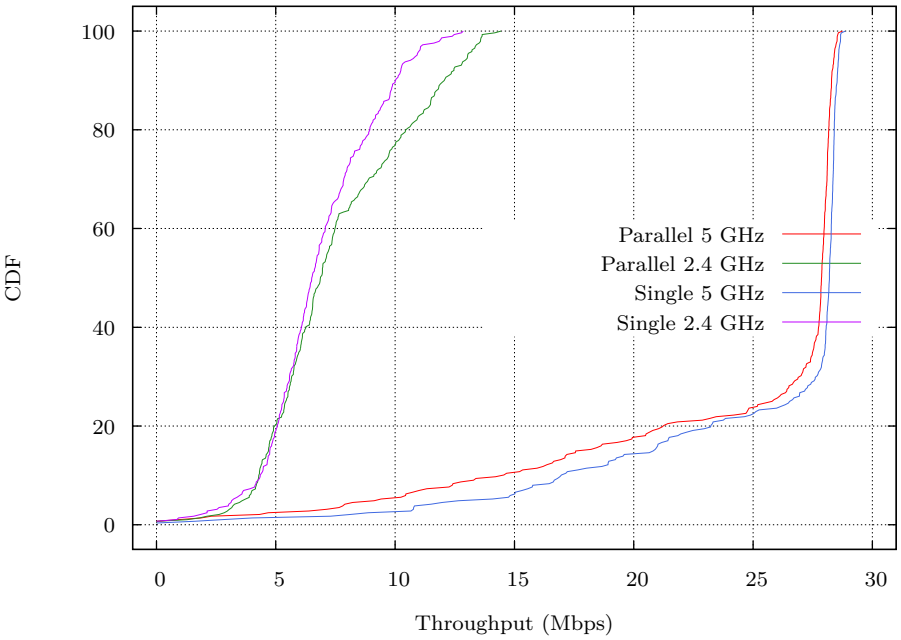
As expected, the experiments we performed with one network operating on the 2.4 GHz band and one network on the 5 GHz band showed no interference between the networks. Figure 3a shows the CDF of the throughputs for an uplink test with three of the tests described in §3.2.2, namely the parallel test and the two single interface tests. We see that the throughput in both cases is almost identical, implying that the networks on these two bands are indeed operating independently. The throughput of the 2.4 GHz parallel test is marginally lower than in the single test, but we attribute this to the tests not being run at the same time. Other tests using the same setup show the 2.4 GHz parallel throughput to be equal to that of the single on balance.

Figure 3b shows a similar test, but run on the downlink. The same lack of interference between the interfaces is evident, with the throughput of one interface being unrelated to whether the other band is in use. This graph also shows a case where the 2.4 GHz parallel is in fact performing better than its single counterpart. As mentioned above, we put this down to varying background traffic on other networks between the times of the two tests.

These results indicate that there should be no performance penalty to using multiple networks on different bands simultaneously. In fact, in the best case, the total throughput should be the sum of each links' capacity.



(a) uplink



(b) downlink

Figure 3: Cross-band interference

4.2 Wireless interference and Multipath TCP

Before discussing Multipath TCP with Coupled congestion control, we present results using Multipath TCP with New Reno congestion control operating independently for each subflow. This congestion control algorithm is known to use an unfair share of the available capacity with Multipath TCP, as it allows the congestion window of each subflow to grow as if it were an independent TCP flow[8]. These tests may seem unnecessary, but we believe they are useful to explore the subtle difference between running two wireless clients in parallel and one client with multiple wireless interfaces. Using New Reno allowed us to evaluate this separately from the load balancing performed by the Coupled algorithm.

4.2.1 Downlink

On the downlink, we would expect each Multipath TCP subflow to have an equal share of the total throughput as a regular TCP flow, as there should be no difference in interference when running both WiFi interfaces on the same machine compared to on separate machines when each interface is only transmitting ACKs. This can be seen in (a) through (c) in Figure 4. We note that in (b) the throughputs are not identical; we again attribute these discrepancies to the unstable nature of wireless networks.

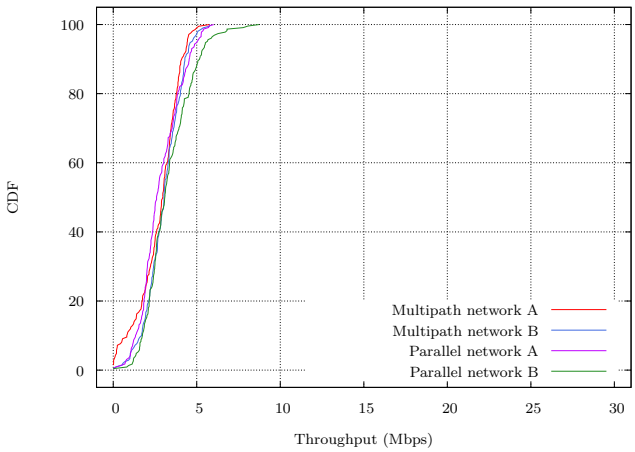
The per-subflow fairness in the downlink case is achieved in two ways. Firstly, the wireless medium should be shared equally between the two APs (assuming they both have data to send and their channels are equally busy) due to carrier sense. Additionally, two flows going through the same AP should get an equal share of its airtime. This is because TCP New Reno does not balance the load across the subflows, and therefore each sender will continue to increase their congestion window independently, limited only by the finite buffer of the AP. The effect of this is that the APs drop further incoming packets as soon as their buffers are full, and this indicates to the senders that they should adapt their rate. At this point, both flows going through a single AP have the same amount of data in that AP's buffer. It thus spends the same time in transmitting each flows' data, and thereby shares its airtime evenly between the flows. On the downlink, fairness is thus provided by a combination of carrier sense **and** the APs' queues.

[8]RFC 6356 - Coupled Congestion Control for Multipath Transport Protocols.

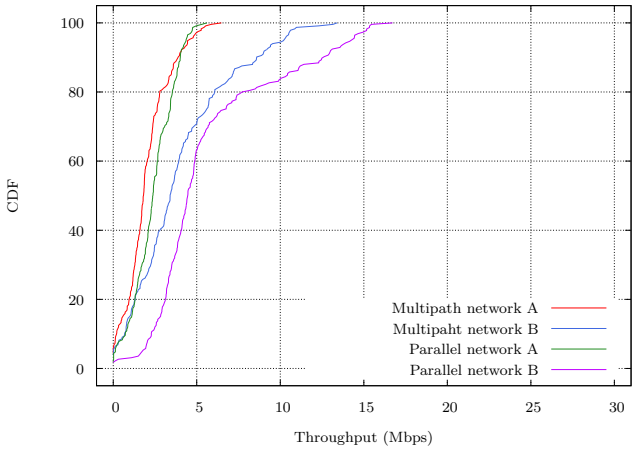
4.2.2 Uplink

Due to the unpredictable nature of WiFi networks and varying levels of background traffic, obtaining consistent results for these tests was a time consuming process. The general trend we observe in Figure 5 is the similar to that seen on the downlink; each Multipath TCP subflow gets an equal share of the total throughput to every other flow on that link. This also makes sense conceptually; on the uplink, the 802.11 MAC provides fairness **per 802.11 station** using carrier sense as explained in §2.2. Since Multipath TCP acts as one station on each network, we would expect each subflow to get half the available capacity on each network when using New Reno.

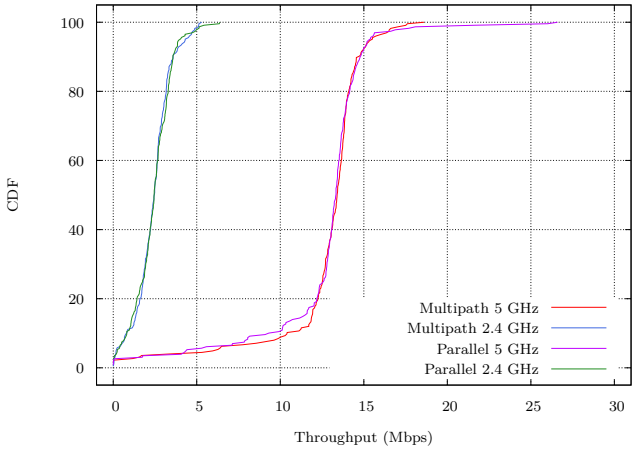
This equal distribution of transmit time gives Multipath TCP an aggregate throughput equal to the sum of half of each links' capacity, and suggests that no additional interference penalty is incurred by running both interfaces on the same client rather than separate clients. We can also reason from this that Multipath TCP with Coupled congestion control should theoretically be able to eventually reach the same throughput as parallel clients if no other flows are competing on the network; this will be examined at in §4.5.



(a) 2.4 GHz, same channel

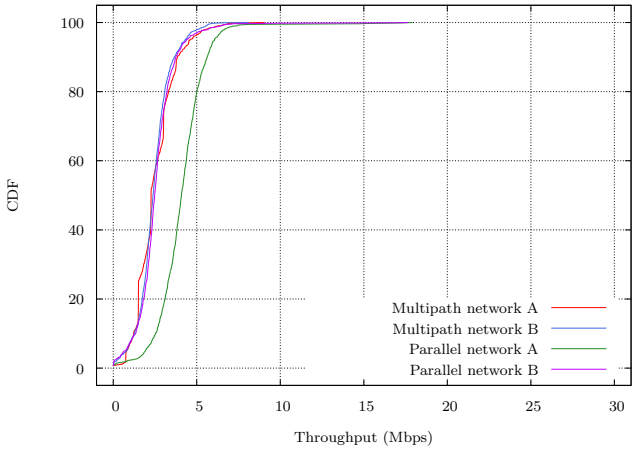


(b) 2.4 GHz, non-overlapping channels

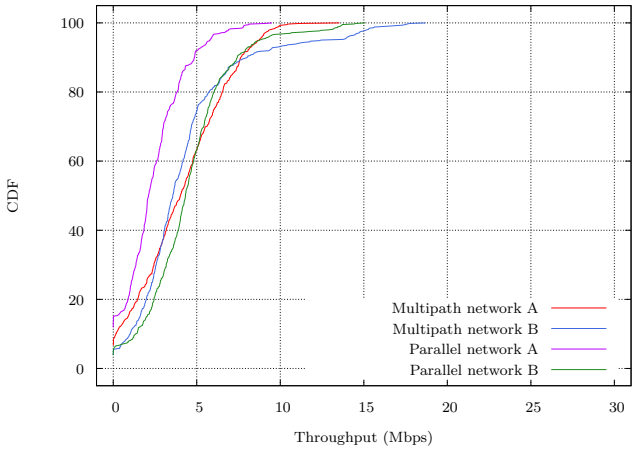


(c) 5 and 2.4 GHz

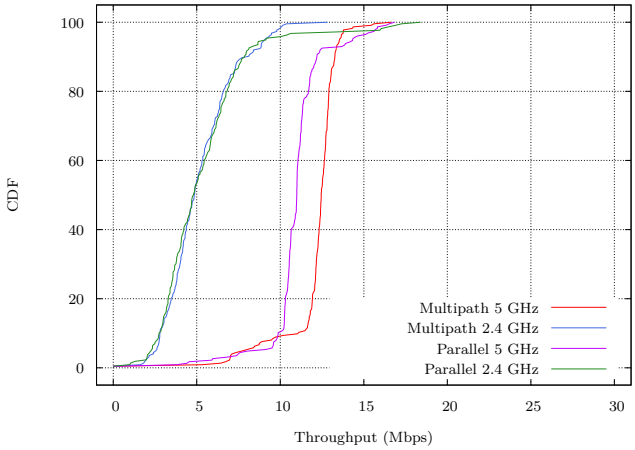
Figure 4: Downlink, New Reno



(a) 2.4 GHz, same channel



(b) 2.4 GHz, non-overlapping channels



(c) 5 and 2.4 GHz

Figure 5: Uplink, New Reno

4.3 Multipath TCP fairness

In the previous section, we showed that Multipath TCP can benefit from using multiple wireless links simultaneously. As expected, we also saw that Multipath TCP with New Reno was using an unfair share of the networks' resources; approximately half of the total available bandwidth when there was a competing flow on each network. One of the goals of Coupled congestion control is bottleneck fairness; more specifically, a Multipath TCP flow should not take more of the bottleneck capacity than any other flow. The following section evaluates the extent to which Multipath TCP with Coupled congestion control achieves fairness on wireless networks.

As in the previous section, we will evaluate Multipath TCP's behaviour on the downlink first, and then on the uplink.

4.3.1 Downlink fairness

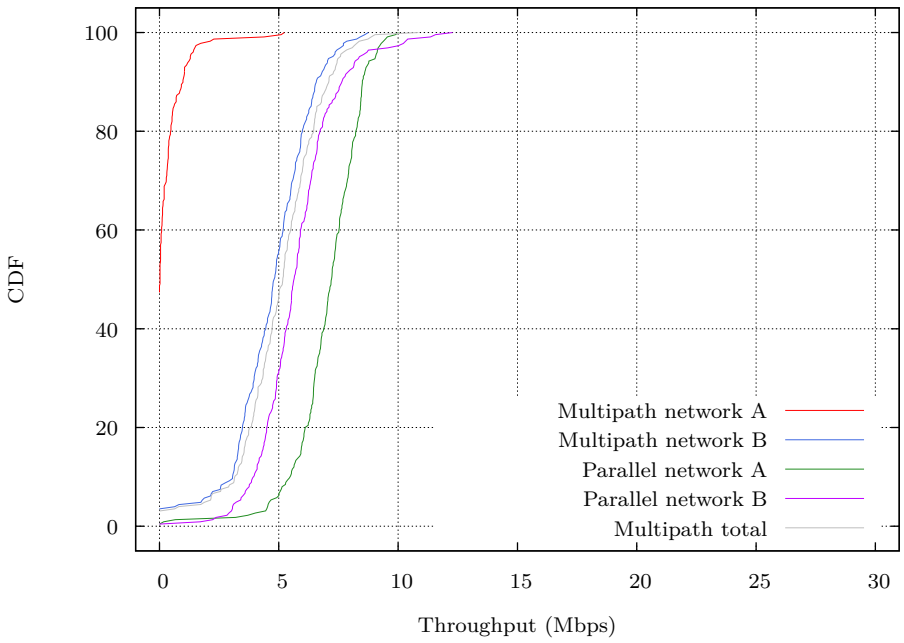


Figure 6: Traffic distribution, downlink, non-overlapping 2.4 GHz

On the downlink, the Coupled algorithm acts like regular TCP New Reno on clients with only a single interface; since they only use one subflow each, Coupled will not try to limit how aggressively it sends data. For the client with multiple interfaces, the Coupled algorithm should limit the aggregate throughput to as

explained in §2.3. Figure 6 shows that Coupled is indeed limiting one of its subflows and transmitting most of its traffic on the other. Additionally, its total throughput is close enough to that of the parallel flows that we can consider Coupled to be behaving fairly; the client with multiple interfaces is correctly taking only a third of the total network capacity, as opposed to half in the case of New Reno.

In §4.2.1 we saw that congestion feedback in the downlink case is provided to the sender by the AP dropping packets if they arrive at a faster rate than they can be sent out out on the wireless channel. The Coupled algorithm uses this feedback to correctly reduce its aggressiveness and thus avoid using an unfair share of the overall capacity. Similar results are seen when the two wireless links are operating on separate bands, as seen in Figure 7 where Coupled is again limiting itself to the throughput of the best available link.

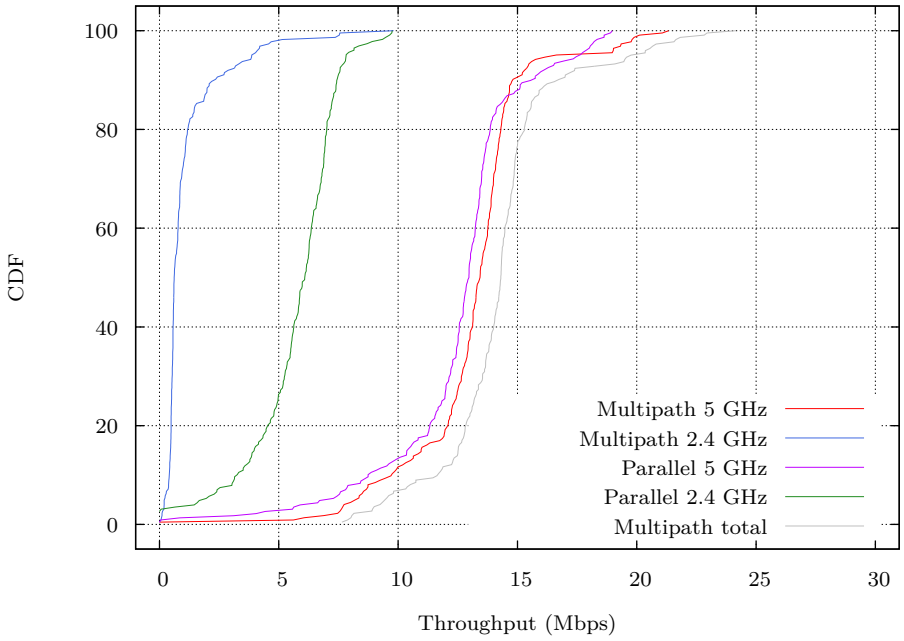


Figure 7: Cross-band downlink test with Coupled

4.3.2 Uplink fairness

Initially, we expected Multipath TCP with Coupled congestion control to perform similarly with both uplink and downlink traffic. However, we soon found that this was not the case. Most of the time plots and CDF graphs showed that each subflow was performing as well as parallel on **both** links as seen in Figure 8, indicating that the client with multiple interfaces was not behaving fairly. Instead, its subflows

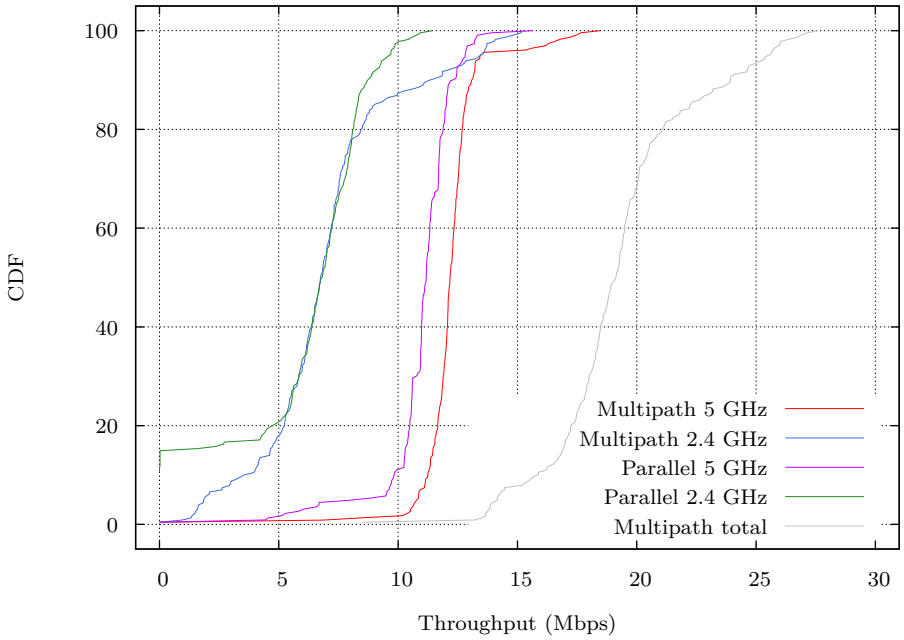


Figure 8: Traffic distribution, uplink, 5 and 2.4 GHz

were taking more of the capacity in total than any of the single-flow clients as seen with TCP New Reno in §4.2. After seeing this result in same-channel, non-overlapping and cross-band tests, we reasoned that the problem had to be related somehow to the behaviour of WiFi with uplink traffic in general. To investigate this, we looked for common elements in the time plots of several tests where Coupled was behaving unfairly.

Figure 9 shows the congestion window as a function of time for one of the subflows in one such test. In particular, the bandwidth delay product shows the approximate number of packets we expect the interface to have in flight at any given point in time. This was calculated by taking the RTT reported by `ping` and multiplying it by the average throughput we were seeing on that interface. The measured RTT was taken when the network was idle, so it was doubled for this calculation. This gives $20\text{ms} \cdot 10\text{Mbits} \approx 25\text{kB}$. The congestion window of the link is not expected to surpass this for any significant period of time, yet the graph clearly shows the congestion window is consistently much greater than this size.

Since Coupled uses the congestion window size to determine how aggressive it should be, an inflated congestion window will make it more aggressive than the environment would suggest, resulting in exactly the behaviour we see in Figure 8.

To understand what is happening here, some background knowledge about how

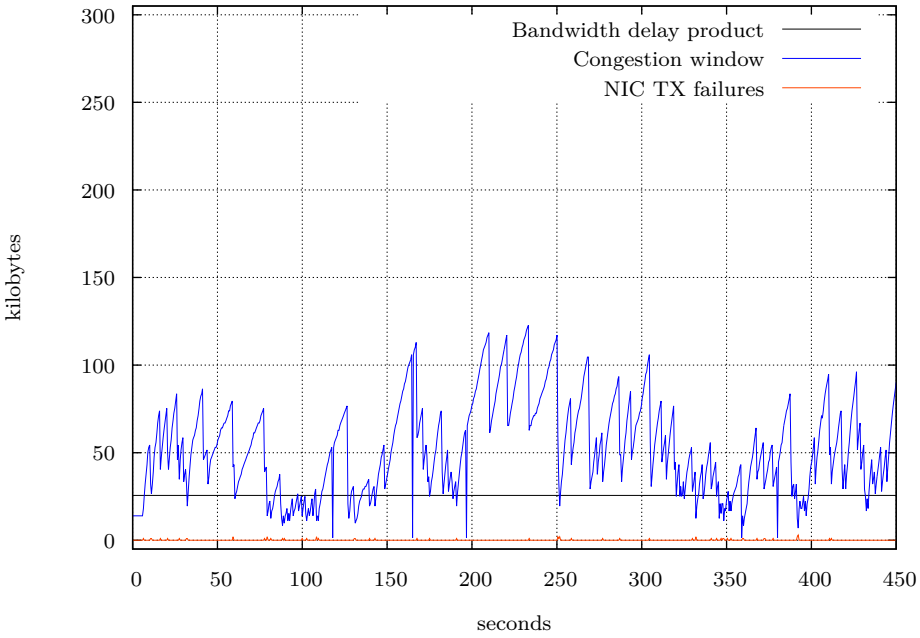


Figure 9: Congestion window size for Multipath TCP uplink connection

TCP works in Linux, and how the Linux kernel manages network queues is needed. Linux maintains the IP send queue as a list of pointers to data in the socket buffer. This list is allowed to grow very large, and so it will never overflow in practice. When TCP tries to send a packet, it will first check that there is room in the current receive and congestion windows, and only then will it pass the packet to the send queue. The NIC then takes packets from the send queue in FIFO order to transmit them onto the medium.

Ignoring the advertised receive window, the only two limiting factors to how quickly a TCP flow can send data are therefore the congestion window and the link speed of the NIC. If TCP passes packets into the queue faster than the NIC can transmit them, the send queue will simply grow, but not drop packets. TCP will consider the packet as sent, but the packet will be kept in the IP send queue until the NIC is ready to transmit it. The relevant effect of this is that TCP will include the time a packet is queued locally in its estimate of the RTT.

This behaviour is not evident on all wireless networks, however. A TCP flow will usually start experiencing loss when it overflows a buffer at a bottleneck, and this is usually not the immediate link as WiFi is generally faster than a user's Internet speed. The flow will eventually limit the number of packets it sends based on the capacity of the bottleneck, and since the capacity of the immediate link is

higher than that of the bottleneck, the NIC should be able to send at least as fast as TCP passes packets into the queue.

A mismatch of assumptions occurs when the WiFi link **is** the bottleneck of the path, such as for local connections or when the user has a very fast connection to the Internet. With the WiFi link as the bottleneck, no loss is expected to occur elsewhere on the path. The NIC will take a packet from the queue, use carrier sense and potentially retry some number of times before actually sending the packet. It will, however, very rarely actually drop a packet because it usually succeeds in sending the packet before reaching its retry limit. Since TCP does not see any loss, it will assume that the path is not congested, and so it will continue to grow its congestion window. In fact, the loss that TCP does see will likely **not** be related to congestion, but rather to faulty links or interfaces, which would cause it to back off its congestion window incorrectly.

With an ever-increasing congestion window, TCP will continue to put data in the send queue faster than the NIC can send it causing the queue to become bloated. This becomes an issue because any packets sent by other flows on the same host will now be queued behind these, increasing their delay significantly and negatively impacting applications such as VoIP, where latency is more important than throughput.

To mitigate this problem, TCP Small Queues has been implemented in the Linux kernel, effectively hard capping the number of bytes a TCP flow can keep in the IP send queue at any given point in time. Although this does not solve the underlying problem, it will limit the amount of delay a misbehaving TCP flow can inflict on other flows.

For Multipath TCP with Coupled congestion control, however, this becomes a substantial problem. As explained in §2.3, Coupled tries to not be too aggressive on any one link by throttling the growth of the congestion window. This works correctly if the congestion window directly impacts how often the NIC puts packets on the network, as throttling the growth of the congestion window will effectively lead to other clients on the network getting a larger share of the network's capacity. However, when the congestion window is sufficiently large that the NIC always has packets to send, this throttling has no effect at all. The NIC is always able to send, regardless of how quickly the congestion window grows beyond that point, which means that each **subflow** is only limited by the MAC layer fairness, which is fair **per interface**. Thus, Coupled will end up behaving exactly like New Reno, which is unfair when multiple subflows are used.

This problem is not related to the amount of interference between the different wireless networks, but rather to the lack of feedback given on wireless networks when the link capacity is reached. It is also only experienced when the WiFi link is the bottleneck, as any bottleneck further along the network path is likely to have a queue which will overflow, drop a packet, and thus signal to TCP that it should back off. In fact, this is precisely what we see in the downlink case where the queue in the AP effectively conveys how congested the wireless link is to the sender.

4.3.3 Uplink fairness on high latency links

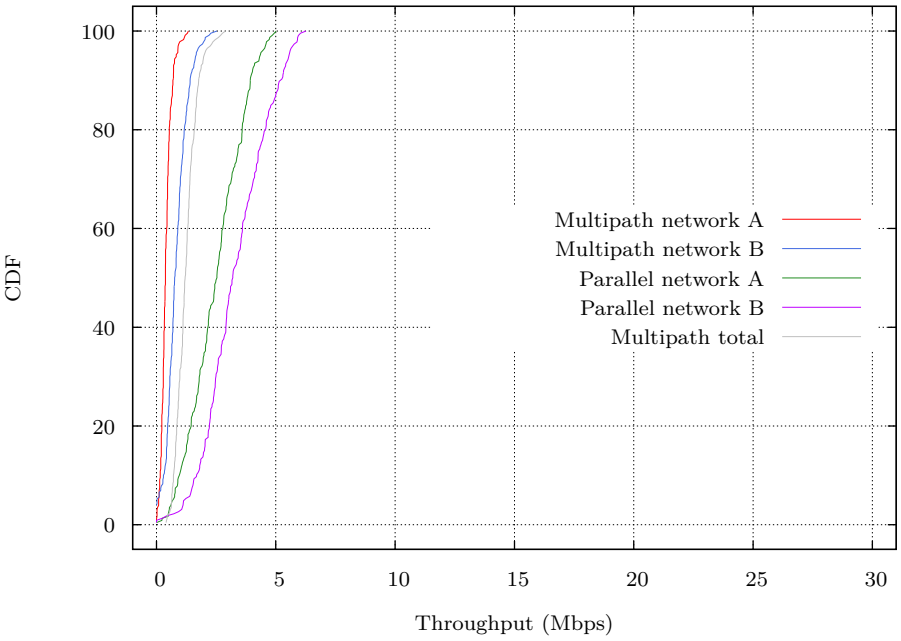
Figure 10 shows a test in which the RTT was artificially increased by 100ms, by adding delay at the IP queue level on the test server. In this case, we would not expect the higher RTT to significantly affect overall throughput, only to lower the rate of growth of the congestion window during slow-start. The CDF, however, shows that throughput is drastically reduced for all flows.

In particular, consider the time plot of one of the Multipath TCP flows in Figure 10b. The bandwidth delay product shows the product of the real RTT + 100ms and the throughput observed without an inflated RTT. Clearly, the congestion window is never approaching this value. The same is the case for the other subflow and the two parallel flows.

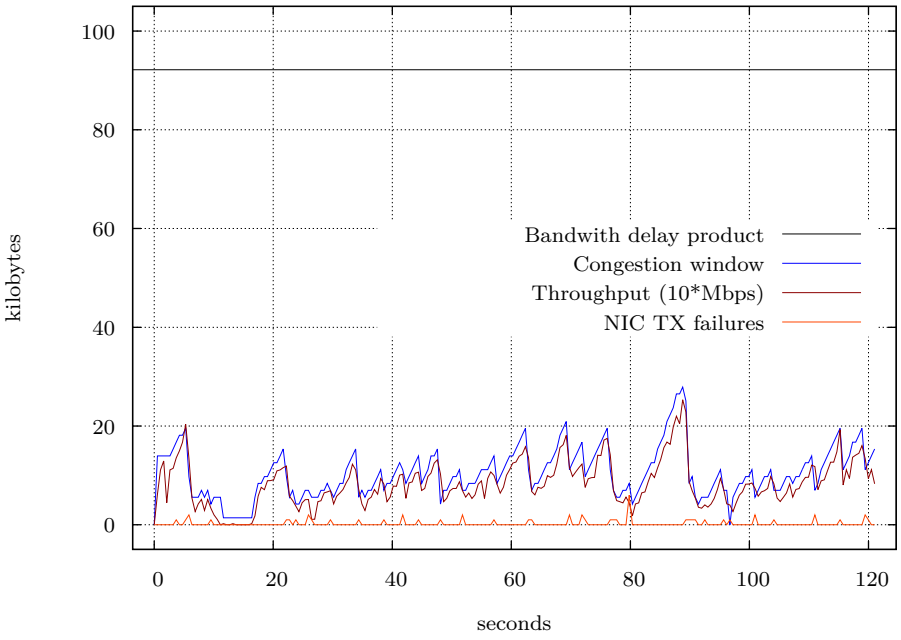
By examining the number of transmission failures reported by the NIC, we can see that the number of dropped packets is similar to that observed in tests without an artificially high RTT. This kind of constant loss causes both Coupled and New Reno to continuously halve their congestion windows. With a higher RTT, the additive increase is now too slow to bring the congestion window up to the bandwidth delay product before the next packet loss, after which the congestion window is halved again. Since TCP is now putting packets into the send queue at a much lower rate than the NIC can transmit them, the throughput is limited by the size of the congestion window. Since the window is too small, the throughput suffers.

Coupled also suffers noticeably more from this problem than New Reno. This is because Coupled is attempting to behave fairly to other flows, but is reacting to misleading signals. Coupled reduces its aggressiveness if it detects that there are competing flows, and this is again deduced from the loss rate. Since the loss rate is close to constant, Coupled believes that there are other competing flows, and

thus reduces its aggressiveness. This further limits the growth of each subflow's congestion window, which exaggerates the problem of the congestion window being too small, further reducing throughput.



(a) CDF



(b) Multipath TCP time plot

Figure 10: Uplink, non-overlapping 2.4 GHz, +100ms

4.4 Inflated congestion windows

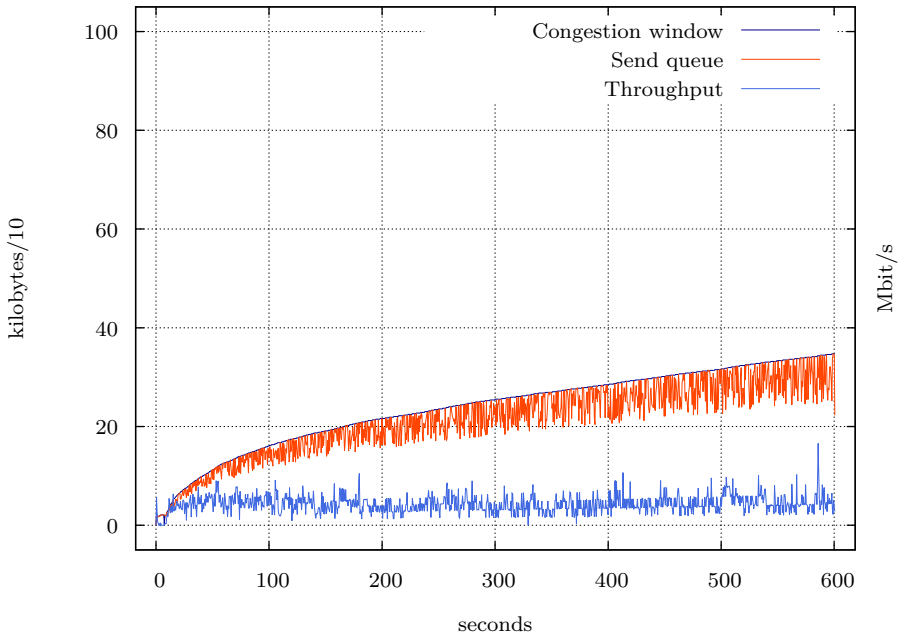


Figure 11: Unbounded congestion window growth

The issue identified in §4.3.2 will affect any connection where the bottleneck of the path is the immediate link a client is connected to, not necessarily just with WiFi links. As mentioned, it will also affect both TCP and Multipath TCP, albeit with different effects. In both cases, the congestion window will rarely be halved, and thus will grow far beyond the "correct" value of the paths' bandwidth delay products.

When the congestion window grows too large as shown in Figure 11, both TCP and Multipath TCP will continue to assume that they are being fair by continuing to put packets onto the send queue, even though these packets are drained slower than they are being put in. The effect this has on Multipath TCP and TCP was discussed in §4.3.2, but we briefly reiterate it here for reference. For TCP, this is not a problem beyond adding delay for any other flows on the host, as their packets will be queued behind an abnormally large number of packets from the bloated TCP flow. For Multipath TCP, however, it is problematic - because it will always have a packet in the queue to send, it will never yield the link voluntarily.

TCP Small Queues, which is the currently applied "fix" in the Linux kernel does not address the underlying problem, as it simply bounds the number of packets any given TCP flow can have in the queue to an arbitrary, fixed number. While

this does reduce the amount of latency that a misbehaving TCP flow can inflict on other flows on the same host, the latency will still remain somewhat inflated without careful tuning the maximum queue size. A very conservative limit could be set, but this will then be incorrect for faster links or paths with different RTTs.

We believe that the correct way of overcoming this must be to modify TCP so that it reacts to congestion which does not manifest as packet loss. Altering TCP is naturally something which should not be done lightly, and the effects of any change would need to be examined on a large scale before they are put into production use. That said, we will suggest a modification which we believe could solve this issue.

The intuition behind our change is that the growing size of the local queue could be used as another signal to TCP that there is congestion on the network. We have observed that the send queue for a normal TCP connection should usually be empty, but may occasionally grow by a small amount if the immediate link is temporarily busy. To determine if the queue is growing when it should not be, we need a way to determine how large it might reasonably grow to.

We base our suggestion on the observation that that the queue is unlikely to grow much larger than the number of bytes which could be transferred on the bottleneck link in one variance of the RTT. This limit, QL, is calculated as shown in Equation 3. For cases when the immediate link is the bottleneck, we believe that this reflects the maximum number of bytes that we should have to buffer up in order to always be able to send when the link becomes available.

$$QL = \frac{var \cdot cwnd}{RTT} \tag{3}$$

The value of var here is the RTT variance as measured by TCP. We note that for links where the immediate link is not the bottleneck, Equation 3 may not behave as desired; the sending host will now attempt to balance its local queue according to a queue further along the network path. This could potentially leave the intermediary links without data to send, and so decrease throughput. This may be overcome by measuring the send delay variance of the immediate link directly and substituting that for var in Equation 3.

Building on the intuition that QL would represent the maximum expected send queue size, we believe it would be appropriate to consider the send queue exceeding that limit as a congestion event, and consequently halve TCP's congestion window

when it happens. In theory, a single RTT should be enough to drain the send queue, and so we expect that adjusting the congestion window at most once per RTT should be sufficient.

To make this easier to implement with TCP, we also derive from Equation 3 a way to step down the congestion window size once per ACK. We want one RTT worth of ACKs to decrease *cwnd* by half, and so we can reduce the window size by *step*, as shown in Equation 4.

$$step = \frac{MSS}{2} \cdot \frac{cwnd_i}{cwnd} \tag{4}$$

Where $cwnd_i$ is set to *cwnd* any time the send queue size is smaller than QL. Thus, we update the congestion window on every received ACK:

```
on_ack():
  if length(send queue) > QL
    cwnd -= step
  else
    cwnd_i = cwnd
```

We believe this should correct TCP's congestion window size, bringing it back to the point where the send queue is usually empty. Being dynamic, it would also be a superior solution to TCP Small Queues. Finally, it could potentially correct the fairness issue we have seen with Multipath TCP over lossless links, as the congestion windows would now be correct, giving Coupled the feedback which it depends on.

Due to time constraints, we have unfortunately not been able to implement nor test this solution, and thus it is presented solely as a suggestion for future work if this problem is revisited.

4.5 Multipath TCP coupled performance

In §4.1, we showed that a performance gain is possible when using multiple wireless networks if the two networks are somewhat separated in the frequency domain. From §4.2, we know that Multipath TCP has the potential to fully saturate both of these links, and our uplink experiments in §4.3 suggest that Coupled will, if it believes it will not impact any other flow, fill its entire MAC layer share of the

network capacity. This section will attempt to determine whether Coupled will eventually fill both its wireless links if there are no other clients on the network at the same time.

For the uplink case, we already know that Coupled will eventually use the capacity of the link. Because of the lack of congestion feedback discussed in §4.3, the congestion window will grow too large, meaning there will always be packets waiting in the send queue, and thus the only limiting factor is MAC layer fairness. If no other clients are competing for access, the Coupled flow will gain all of the capacity, and thus it will saturate the link. Until this lack of feedback is somehow resolved, the results from experiments with Coupled on the uplink are predictable and not particularly interesting.

When Coupled is running for a downlink connection, however, congestion feedback **is** given, and so its behaviour when there are no competing flows is interesting. To address this, we performed a series of experiments where a parallel test was run and then immediately followed by a test where one host was connected to both hosts and running Coupled. Due to the timing difference between the parallel and Coupled tests, seeing identical throughput is unlikely, but we would be surprised to see a consistent difference in throughput between parallel and Coupled at this point.

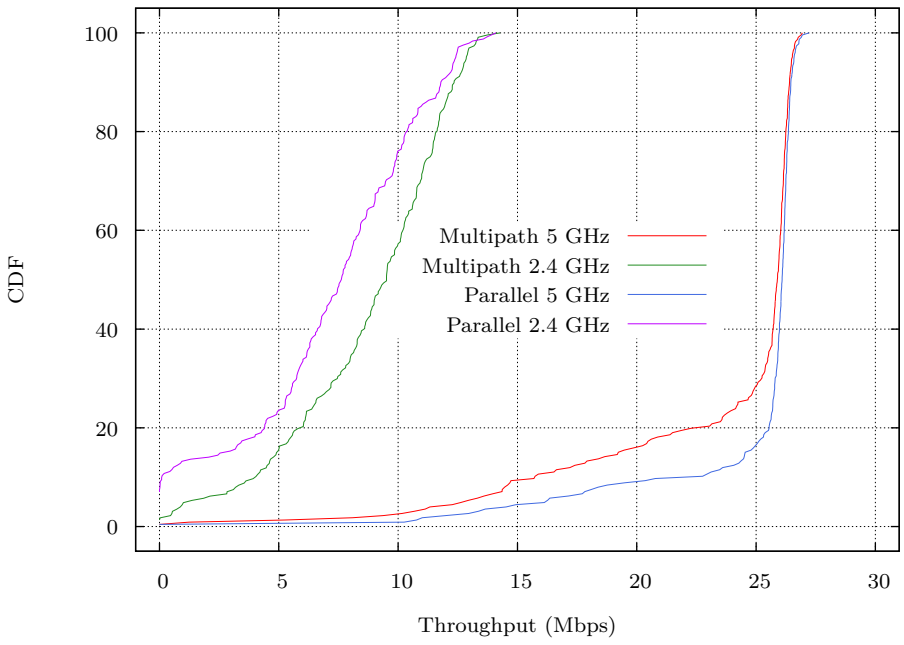


Figure 12: Coupled downlink performance, idle links

The graphs in Figure 12 show the downlink results for the aforementioned experiments, and we can see that Coupled is following the performance profile of the independent interfaces. This implies that Coupled is in fact allowing Multipath TCP to eventually use a total throughput equal to the sum of what TCP New Reno would get on each flow individually when there are no competing flows. This means that it also works as expected on wireless networks.

4.6 Reacting to change

One of the goals of Multipath TCP is to use the available links to improve reliability, and it does this partly by moving traffic away from congested paths. Considering that WiFi connectivity can change drastically as a user moves around, we felt it would be interesting to test Multipath TCP's ability to move traffic from one link to another.

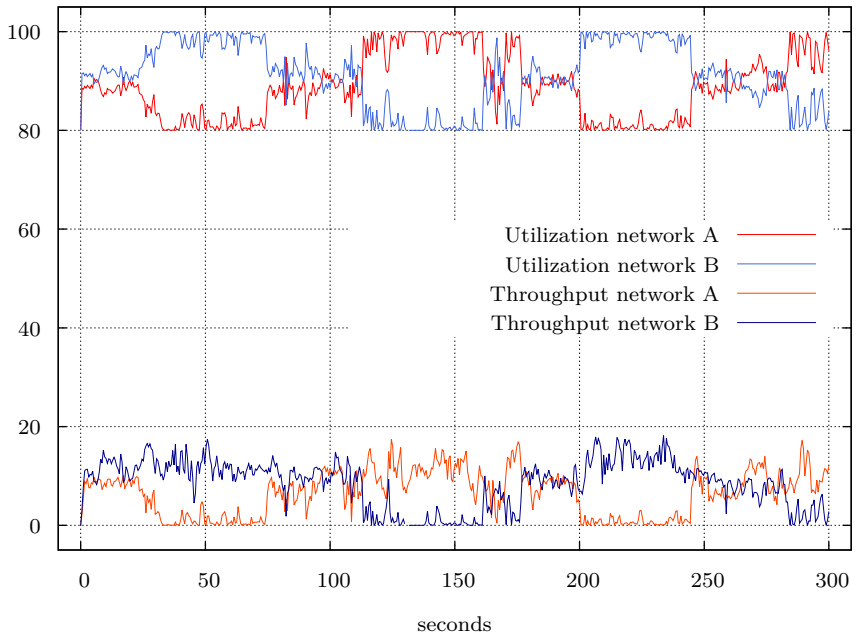


Figure 13: Traffic distribution with movement, downlink, non-overlapping 2.4 GHz

The results shown in Figure 13 show that Coupled is indeed doing exactly what we would expect. More traffic passes through one interface when the user is closer to that AP, but more importantly, it evenly distributes the traffic when the user moves back into a position where both networks are approximately equally strong.

We chose to only perform this test with downlink traffic since, as discussed

in §4.3, Coupled simply behaves like New Reno on the uplink, making the these results predictable.

4.7 Closing remarks

Throughout our experiments, we have come across several odd results, correlations and trends which we could not immediately explain. Some of these turned out to be errors, but many made complete sense after some hard thinking. This section aims to explain most of the oddities that can be observed in some of the graphs given in this report.

4.7.1 Inflated RTT

A number of our tests experienced very high RTT estimates of around 600 ms despite the network being relatively fast (i.e. we see an RTT of $\approx$ 7 ms with `ping`). The RTT also seemed to increase and decrease with the size of the send queue. As discussed in §4.3, the lack of packet loss causes unbounded growth of the congestion window, but with most of the packets actually being the local send queue. Because TCP estimates RTT based on when the packet was put into the send queue, **not** when it is actually sent by the network interface, the RTT estimate will include the time a packet spent in the queue. Since the queue is growing, so will the RTT.

4.7.2 Logarithmic growth of congestion window

Figure 11 shows the congestion window size plotted over time for one test we ran in which we were seeing very little loss. It clearly shows something resembling logarithmic growth of the congestion window rather than the familiar linear saw-tooth. To understand why this is happening, it is necessary to look closer at how the congestion window is increased.

The congestion control mechanisms used by Multipath TCP and regular TCP increase the congestion window by one MSS per RTT. They do this by increasing the congestion window by approximately $1/cwnd$ per received ACK. This works well in the expected case where increasing the congestion window will cause more packets to be sent, and thus more ACKs to be received per RTT, but is not correct when the link layer masks loss.

What happens when TCP does not see loss, as discussed in §4.3, is that the congestion window grows larger than what the network interface can handle, and

so increasing it will not cause any more packets to be sent. The number of ACKs received in an RTT will therefore remain constant. The $1/cwnd$ term, on the other hand, will grow smaller, and so the growth of the congestion window relative to the congestion window size per RTT will decrease, leading to the logarithmic growth seen in the results.

4.7.3 Congestion window and the send queue

Figure 11, and some other plots in this report, show the send queue size closely following the congestion window size through the entire test. Since we were seeing very little loss in many of our tests, and thus the congestion window was clearly growing larger than it should be, we initially reasoned that most of the bytes of the congestion window were likely to be in the host's send queue. In this case, we would expect the send queue to closely follow the congestion window, with a small gap between them representing the packets actually in flight.

However, we then noticed that we were seeing the send queue following the congestion window also when we **were** seeing loss and the congestion window was **not** inflated, such as in §4.3.3. In these cases, there should not be many packets in the send queue at all; they should mostly all be in flight.

It turned out that the reason for this is surprisingly simple; the send queue size reported by the kernel *includes unacked packets*. TCP keeps packets after sending them in case they must be re-transmitted, so until they have been ACKed, they will continue to take up space in the queue. This explains both why we were not seeing a gap between the send queue size and the congestion window in the no-loss experiments, **and** why the send queue was seemingly following the congestion window when loss was occurring.

4.7.4 Congestion window and throughput

In Figure 10b, we observe another very strange phenomenon that occurred in a number of experiments. Here, we see a curious correlation between throughput and the congestion window size. Other tests also show a correlation between the two, but it is particularly evident here as it looks like the throughput is strictly bounded by the congestion window size.

To determine why this was happening, we first tried to find similarities between the graphs that were showing this peculiar trend. It turned out that we were only seeing this in tests which showed either a high RTT with constant loss rates, or a

high amount of loss. These cases both share the feature that the congestion window is constrained from growing to the full bandwidth delay product of the path, and so no queues are expected to build up anywhere in the network. The limiting factor for the throughput thus becomes the congestion window not allowing TCP to add more packets to the send queue, even though the link is ready to send. The net effect of this is that the throughput is limited by the congestion window; whenever the congestion window grows, the throughput increases, because TCP is allowed to put more packets on the wire. If the congestion window is halved, TCP stops sending packets almost immediately, and the throughput drops.

5 Conclusion and future work

In this paper, we have presented a number of wireless experiments showing that multiplexing across multiple wireless interfaces on a single host may indeed provide advantages for certain network configurations both in terms of reliability and throughput.

Furthermore, we have shown that Multipath TCP works well when traffic is primarily downlink, which is often the case on wireless networks today, as traffic is balanced between the available wireless networks and each host uses a fair share of the total network capacity. We have also shown that it behaves unfairly with uplink traffic due to a combination of 802.11 retransmissions and effectively unbounded buffering in the network stack.

Finally, we have discussed how the uplink problems experienced by Multipath TCP are not unique to wireless links, and that they manifest also for regular TCP connections as inflated buffers. Unfortunately, due to time constraints, we have not been able to test our suggested solution to this problem, and so the question of how to properly limit queue sizes without the presence of packet loss remains an open problem.

Unfortunately, there are many potentially interesting experiments we have not been able to perform due to the limited time of the project. First and foremost, all of our experiments were performed in a building where several other wireless networks were present, making it difficult to obtain consistent results. To verify the behaviour we have observed, repeating the experiments in more controlled environments may be an appropriate next step.

Most of our experiments were also conducted with two static APs, and either two or three wireless clients each with a maximum of two interfaces. Investigating how Multipath TCP behaves when more networks are available, or with different physical distributions of APs, may yield significantly different results. As 5 GHz networks become more prevalent, testing the extent and impact of interference between 5 GHz channels could also be of value.

Additionally, we have shown that Multipath TCP experiences varying levels of interference based on the underlying network. We see that there is no interference between the 2.4 GHz and 5 GHz bands, while the interference patterns observed in §4.1 suggest that some further research on how Carrier Sense works across channels could be warranted.

Reduced throughput for flows on the same channel, inflated buffers and unfair

behaviour for uplink traffic makes the deployment of Multipath TCP for wireless clients problematic. To be useful in a wireless setting, Multipath TCP will need to be modified to correct these issues. We propose a potential solution to this in §4.4, but further research will be necessary to verify the correctness of this.

Multipath TCP is an exciting technology, which we believe has the potential to provide greater reliability and improved throughput to wireless hosts. With further research on its interaction with WiFi, we believe it could greatly improve the experience of using the Internet on mobile devices.