

***RePT*: Resource Pooled TCP**

Multi-Source, Multi-Path TCP with Linear Network Coding

Nathanael Davison

Corpus Christi College

Supervised by Prof. Jon Crowcroft

December 26, 2017

Submitted to the Department of Computer Science in partial
fulfilment of the requirements for

Part III Computer Science

at the

University of Cambridge



**UNIVERSITY OF
CAMBRIDGE**

Abstract

Resource Pooling is a simple and widely used technique by which a collection of resources is made to behave as a single, pooled resource. The Internet of today presents many opportunities for resource pooling, which are yet to be exploited. The most popular content accessible through the Internet is served from many, redundant locations, and multiple redundant paths exist between hosts and data sources. Furthermore, the field of network coding has arisen, providing opportunities for pooling data across packets and flows.

This dissertation proposes, describes, and evaluates Resource Pooled TCP, *RePT* (pronounced ‘Repeat’). *RePT* is the latest step in the attempt to produce an evolution of the traditional TCP protocol, introducing concepts of multiple-sources, multiple-paths, and network coding. At the heart of *RePT* is an ‘inversion’ of the architecture of TCP, with data being included with acknowledgements, produced in response to a request. This inversion removes the need for low-latency coordination between sources, which would otherwise be necessary to avoid sources transmitting the same blocks. In this way we prevent unnecessary data being sent through the network.

RePT is shown to perform at least as well as traditional TCP in all situations, and is able to provide significant benefits when resource pooling is available. We show that *RePT* is able to increase the load on paths in response to the loss of one or more connections to a source, resulting in a 6 times reduction in the time to transfer a file when a source breaks. *RePT* is able to dynamically shift traffic placed on each path in response to dynamic network conditions. Specifically, when a source or path becomes heavily loaded, more traffic is placed on less loaded paths in order to reduce additional load on the heavily loaded path. Thirdly, pooling data across packets is shown to reduce the average time to decode batches

Words in text: 11,988

Declaration

I, Nathanael Davison of Corpus Christi College, being a candidate for Part III in Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date: December 26, 2017

This dissertation is copyright ©2017 Nathanael Davison.

All trademarks used in this dissertation are hereby acknowledged.

Contents

1	Introduction	3
1.1	Main Contributions	5
2	Background and Related Work	9
2.1	Multi-Path Transport	9
2.1.1	Multi-path <i>Routing</i>	10
2.2	Multi-Source Transport	11
2.3	Network Coding	12
2.3.1	MAC-Independent Opportunistic Routing and Encod- ing (MORE)	13
2.4	Network Coding in P2P and Multi-Path Technologies	14
3	Multi-Source, Multi-Path TCP with Linear Network Coding	17
3.1	Overview of <i>RePT</i>	17
3.1.1	<i>RePT</i> Flow Establishment	19
3.1.2	Data Transfer	20
3.2	<i>RePT</i> Details	23
3.2.1	<i>RePT</i> Requester	23
3.2.2	<i>RePT</i> Responder	31
3.2.3	<i>RePT</i> Forwarder	33
3.3	Structured Coding Approach	35
4	Experimental Evaluation	39
4.1	Simulation Environment	40
4.2	Effect of Multiple Sources and Paths	41

4.2.1	Losing Connection to a Source	41
4.2.2	Heavily Loaded Path(s)	45
4.3	Effect of Network Coding	45
4.4	Combining Multiple Sources, Multiple Paths, and Network Coding in <i>RePT</i>	48
4.4.1	<i>RePT</i> 's Overhead	49
5	Summary and Conclusions	53
5.1	Future Work	54
5.1.1	Kernel Level Implementation	54
5.1.2	Mechanisms for Choosing Best Paths and Sources . . .	55
5.1.3	Comparison of Structured and Random Coding Approaches	56
5.1.4	Opportunities for Coding	56
5.1.5	Additional Use Cases	57
5.1.6	Security Implications of <i>RePT</i>	57
A	Proof of Linear Independence of Rows	59
B	Project Proposal	63
B.1	Introduction, Approach and Outcomes	63
B.2	Workplan	66

List of Figures

1.1	MPTCP cut in half. The \rightarrow 's here contain switches, routers, proxies, and middleboxes which can implement recoding of packets from the same or different flows.	5
2.1	Subset of geographical locations of YouTube servers. Collected using PingTools GeoPing utility [1].	15
3.1	<i>RePT</i> requester establishing first connection with first source.	19
3.2	<i>RePT</i> requester with a connection for each path to each responder.	21
3.3	<i>RePT</i> Packet Format.	22
3.4	<i>RePT</i> forwarder forming linear combinations of uncoded packets.	34
4.1	Network Topology in Simulation.	41
4.2	Connection starting up across multiple paths and sources, and losing connection to first source. First column shows case of 1 path to each source, second column shows 2 paths to first source, and 1 path to second source. Error bars show 95% confidence interval.	42
4.3	Source Breaking under Traditional TCP	43
4.4	Connection to first source becomes heavily loaded, reducing throughput, and so traffic is shifted onto connection to second source.	46

4.5	Effect of Batch Size on number of additional requests and time to transfer a file (excluding code/decode time). Error bars show 95% confidence interval.	47
4.6	Average time between first request for a batch and receiving all <i>BatchSize</i> packets, relative to TCP	48
4.7	Time to transfer a 1MB file, relative to traditional TCP-Vegas, under different numbers of paths, sources, and batch sizes. . .	50
B.1	MPTCP cut in half. The →'s here contain switches/routers/proxies/middleboxes which can implement recoding of packets from the same or different flows.	65

List of Tables

3.1	Notation used in discussion of <i>RePT</i>	18
4.1	Details of device used to perform Network Emulation	40
4.2	Time to Transfer 1MB in simulation when connection is lost to a source.	45

Chapter 1

Introduction

When considering communication networks, a primary goal is to achieve the maximum flow of information, as allowed by fundamental limits. A concept often used to approach the maximum flow across a network is ‘resource pooling’; a simple and widely used technique by which a collection of resources is made to behave as a single, pooled resource. Since the earliest days of the ARPAnet, new technologies have been developed and deployed improving quality of service by exploiting practical properties and implementing new ideas from a wide range of fields. Many of these technologies apply the resource pooling principle to maximise information flow. Packet switching to achieve statistical multiplexing, traffic engineering to balance load, and multi-homing sites for improved failure resilience are all examples of resource pooling [2].

As the Internet continues to develop, new opportunities for resource pooling continue to present themselves. The most popular content on the Internet is often served from many, redundant locations, with streaming services like Netflix and YouTube replicating content across many geographically diverse locations. In addition to this redundancy of data sources, the modern Internet often has multiple redundant paths between hosts and data sources, with modern devices having access to multiple wireless interfaces, and multi-homing the norm for server farms [3]. This redundancy in the Internet,

which has arisen in the past decade, presents new opportunities for resource pooling which are yet to be exploited.

Attempts have been made to translate these theoretical opportunities for resource pooling into practical protocols which can be deployed on the Internet. An early example of a multiple-source approach to data transfer came from application-layer peer-to-peer protocols such as BitTorrent, which provides data from multiple sources simultaneously, allowing users to join a ‘swarm’ of hosts which upload to, and download from, each other. In addition, it has been shown that resource pooling by striping data over multiple paths can provide simple and often near optimal load balancing [4]. Multi-Path TCP (MPTCP) came out of this observation, allowing multiple paths to be used simultaneously by a single transport connection. MPTCP has been shown to provide significantly higher throughput than traditional TCP; in addition to improved mobility [3].

Alongside these developments in the structure of the Internet, developments in the field of ‘network coding’ have created new avenues for resource pooling, applying traditional coding theory at the level of flows. Coding across data is a concept widespread in today’s communication systems at the link level, achieving data rates approaching the capacity of the additive white Gaussian noise channel [5,6]. In their seminal paper, Ahlswede et al. introduced coding at the level of packets and flows, allowing the mixing of data at intermediate nodes in the network [7]. In this way, network coding achieves a form of resource pooling by spreading data across both packets and flows. This facilitates a form of compression, reducing the number of transmissions, and provides a mechanism to overcome packet loss [8,9].

Despite these many opportunities for resource pooling in the modern Internet, little technology exists to exploit it. The transmission control protocol (TCP) remains by far the most widely used transport protocol on the Internet, yet is essentially a point-to-point, single-path protocol. This dissertation aims to present and evaluate an approach to exploiting these new opportunities for resource pooling in the modern Internet. Beyond accessing content replicated on the web, such a protocol would have application in a num-

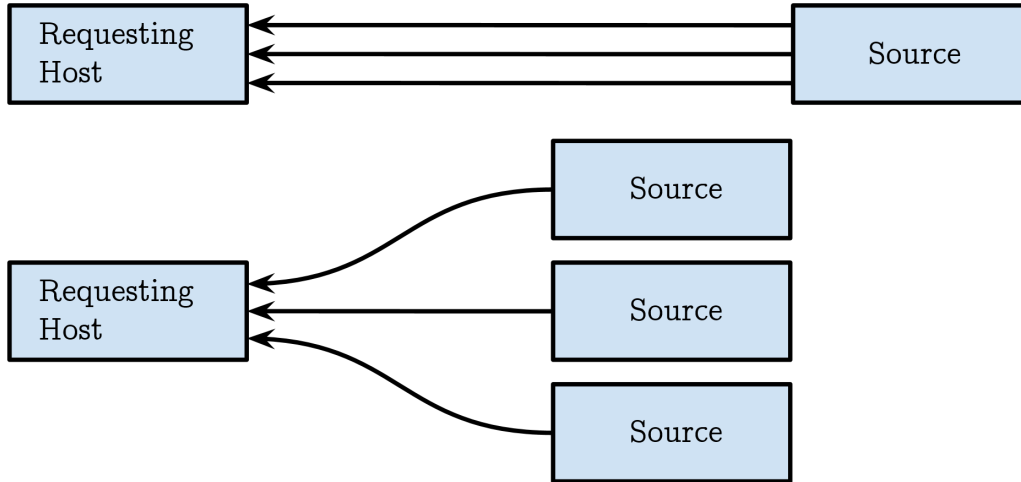


Figure 1.1: MPTCP cut in half. The \rightarrow 's here contain switches, routers, proxies, and middleboxes which can implement recoding of packets from the same or different flows.

ber of diverse use cases, such as avoiding hotspots within datacentres, and mitigating bottlenecks slow uplinks in the edge if the internet.

1.1 Main Contributions

This dissertation proposes, describes, and evaluates Resource Pooled TCP, *RePT* (pronounced ‘Repeat’). *RePT* is an evolution of the traditional TCP protocol, introducing concepts of multiple-sources, multiple-paths, and network coding. With *RePT*, a multi-homed host, requesting content which is replicated across a number of sources, is able to connect to multiple sources simultaneously, and use multiple paths to each source. The sources send coded blocks of data, with intermediate nodes encouraged to recode data in the network. To a certain extent, *RePT* can be viewed as multi-path TCP ‘cut in half’ as shown in figure 1.1, with the addition of network coding.

At the heart of *RePT* is an ‘inversion’ of the architecture of TCP. Rather than the source transmitting each block of the file in turn, the host looking to obtain the file sends requests for data within a section of the file to each

source. The sources construct linear combinations of data within the indicated section of the file, and appends the coded data to the acknowledgement for the request packet. As such, rather than the data being sent from ‘source’ to ‘sink’, with acknowledgements in reply, we have requests being sent from ‘sink’ to ‘source’, acknowledgements being sent from ‘source’ to ‘sink’, with data being appended to these acknowledgements. This approach removes the need for low-latency coordination between sources, which would otherwise be necessary to avoid sources transmitting the same blocks.

The main contributions of this work are as follows:

1. The design and description of an evolution of TCP allowing the connection to multiple sources simultaneously, using multiple simultaneous paths, and allowing for coding and recoding of packets within the network.
2. Evaluation of the protocol in simulation. The results of this evaluation reveal the following findings.
 - On average, *RePT* *always* performs at least as well as traditional TCP-Vegas, and much better in situations where resource pooling can be exploited.
 - *RePT* is able to increase the load on paths in response to the loss of one or more connections to a source, resulting in 6 times reduction in the time to transfer a file when a source breaks.
 - *RePT* is able to shift traffic along each path in response to dynamic network conditions. Specifically, when a source or path becomes heavily loaded, more traffic is placed on less loaded paths.
 - In order to gain significant benefits from network coding, batches of at least 16 blocks should be used in order to sufficiently improve the time to decode each batch.

The rest of the dissertation is organised as follows. Chapter 2 establishes the context for this work, and introduces several related works. This dissertation draws from previous work on multi-pathing, and network coding; introduced

in sections 2.1 and 2.3 respectively. Chapter 3 provides a description of the *RePT* protocol, presenting an overview of the architecture of the protocol and descriptions of each component.

Chapter 4 presents the results of an evaluation of the protocol in simulation, showing the advantages of *RePT* in a number of situations, and quantifying the gains that can be gained using resource pooling techniques. Finally, chapter 5 summarises the contributions of this dissertation, concludes it, and discusses future work related to this dissertation.

Chapter 2

Background and Related Work

In this chapter, we introduce the reader to the basic concepts and related work in protocols supporting multi-sourcing, multi-pathing, and network coding. Much of the work in this dissertation can be seen as identifying useful elements of previous work and formulating it in the context of the general Internet.

2.1 Multi-Path Transport

The modern Internet is inherently multi-path, with modern devices having access to multiple wireless interfaces, and multihoming the norm for server farms [3]. Despite this, traditional TCP, a protocol designed in the 1970s, remains a single-path, point-to-point protocol. Once a TCP connection is established, the connection is bound to the IP addresses of the two communicating hosts. If either of these addresses changes, the connection will break.

This approach is clearly sub-optimal, neglecting opportunities improve throughput by the efficient usage of resources, and increase the resilience of connections by supporting multiple concurrent paths. As such, multi-path extensions to the Internet transport protocols have been proposed as a means of

realising some of the goals of resource pooling. An early attempt at multipath transport was the Stream Control Transmission Protocol (SCTP), standardised in 2000, and originally intended for use in telecommunication signalling [10]. The primary object of SCTP was to support redundancy and mobility for multi-homed devices; however, in its original form SCTP did not support the *simultaneous* use of multiple paths. While SCTP provided many of the same features as TCP, the protocol was not compatible with TCP and required applications to actively choose to use it rather than another transport protocol. Due to these limitations, SCTP never saw widespread use.

The Internet Engineering Task Force (IETF) created a Multipath TCP working group in 2009, from which emerged the Multi-Path TCP protocol (MPTCP) [11]. MPTCP is a specific transport protocol that instantiates the Multipath TCP concept, improving on the work of earlier attempts such as SCTP, with the objective of wide-scale deployment. In addition to the functional goals of improving throughput and resilience through resource pooling, the IETF working group have established additional ‘compatibility goals’ which aim to support the deployment in today’s Internet by providing compatibility with existing applications, networks, and other network users [12]. This evolutionary, rather than revolutionary, approach to protocol development has led to the adoption of MPTCP in a few applications, most notably the Siri application on iPhones, iPads and Macs; performing uninterrupted handover between wireless networks [13].

2.1.1 Multi-path *Routing*

SCTP and MPTCP exploit the multi-homing of servers and the availability of multiple technologies as devices to provide multi-path transport. However, efforts are being made to produce extensions to traditional routing algorithms allowing the installing of multiple routes over multiple paths in routing tables. Equal-Cost Multipath in OSPF is an example of this, allowing traffic be distributed across paths whenever several equal-cost routes to the destination exists [14]. Similarly, Van Beijnum et al. have proposed changes to

BGP’s path selection and dissemination rules, allowing for the use of a wider selection of paths concurrently [15]. These proposals operate at the IP-layer, and work in tandem with transport-layer solutions.

2.2 Multi-Source Transport

In addition to being inherently multi-path, for a considerable proportion of traffic the Internet is also a multi-source environment. The most popular items on many streaming services are manyfold replicated, both within each datacentre and across many datacentres. Figure 2.1 shows the geographical locations of a subset of the YouTube datacentres. When a host first makes a request for the content, a frontend decides the best source under the current conditions, using some heuristic. The host then establishes a TCP connection with the chosen source, and remains connected to this source for the duration of the connection. In this way load balancing between sources is provided on the granularity of connections, however, the potential advantages of performing this on the granularity of packets is clear.

A number of peer-to-peer (P2P) file sharing protocols exist, which might be placed into the category of ‘multi-source’ transport. Cohen’s BitTorrent protocol, described in 2001, remains one of the most influential such protocols [16]. In BitTorrent, networks of coordinating users are created for every set of files. The file being distributed is divided into a number of blocks, referred to as ‘pieces’. As each peer receives a new piece of the file, it becomes a source of that piece for other peers, introducing additional ‘sources’ into the network. When a new host joins the network, they are provided with the ‘torrent descriptor file’, providing meta-data such as the number of pieces of the file and which hosts are able to provide which pieces. Hosts form a “swarm” of hosts, downloading blocks they do not have from other sources, and uploading blocks of data to their peers when required [17]. In this way data is gathered from a number of sources simultaneously, although in practice there is only one ‘true’ source. While BitTorrent is a multi-source

protocol, the multiple sources are artificially created by having clients become servers for files they are downloading, and does not exploit the redundancy extant within the Internet. Furthermore, in BitTorrent all peers are assumed to be equal, with sources chosen uniformly at random. As such, there is no dynamic load balancing based on network conditions.

2.3 Network Coding

The central concept of Network Coding is the mixing of data during transport, rather than treating each segment as a distinct object which must be delivered intact. The mathematical underpinnings of Network coding were established by Ahlswede et al., who demonstrated that the information rate from a source to a set of nodes can reach the minimum of the individual max-flow bounds, through coding [7]. Li et al. furthered this work by providing a constructive proof that this bound can be reached using only linear functions when mixing packets together [18], while Ho et al. demonstrated random linear functions sufficed [19].

Without coding, each packet confers information for the data segment it contains. In contrast, with network coding, each received packet provides information about *every* data segment coded together. These data segments can be decoded once sufficient independent linear combinations have been received. Specifically, when using network coding, the source transports linear combinations of data segments, which can be decoded once sufficient linear combinations have been delivered. Therefore, it is possible to anticipate packet loss in the network and introduce redundancy to overcome this packet loss, without needing to know which specific packets will be lost. Furthermore, in multi-path and multi-cast situations, the total number of transmissions can be significantly lowered by having intermediate nodes combine and transmit linear combinations of the packets they receive [18].

The field of network coding is still in its infancy, and is yet to be adopted in many commercial products - leading some to question whether the technique

is practically useful [20]. However, an extension to TCP which includes network coding, called TCP/NC, was proposed by Sundararajan et al. in 2012 [8]. A significant feature of this work is the redefinition of TCP acknowledgements. Rather than indicating the receipt of a packet, an ACK indicates the receipt of a ‘degree-of-freedom’, such that once sufficient degrees-of-freedom have been received, the original packet can be recovered. The authors demonstrate the throughput advantages of their scheme under a number of different loss rates. Researchers have been able to demonstrate the application of network coding for a variety of areas, including as an alternative to forward error correction and ARQ in traditional wireless networks [9, 21], reducing buffer delays [22, 23], and increasing bandwidth in single-hop wireless multicast transmission [24]. Beyond improving data rates in networks, network coding has been used to provide secrecy and resistance to replay attacks [25, 26], and in distributed storage [27, 28].

2.3.1 MAC-Independent Opportunistic Routing and Encoding (MORE)

The majority of recent research into network coding has been oriented towards wireless mesh networks. To construct a wireless mesh-network, an administrator need only scatter a few commodity nodes over the covered area, and all other configuration performed in software [29]. Similarly, ad-hoc mesh-networks can be formed with mobile devices, such as cell phones. This simplicity comes at the cost of reduced link quality, with physical phenomena leading to bit errors and corruption [30]. The broadcast nature of mesh networks means they are highly amenable to network coding, proving many opportunities to combat the drawbacks of the technology, and reducing the number of transmissions necessary [18].

This is the focus of MORE (MAC-Independent Opportunistic Routing and Encoding), proposed by Chachulski et al. [31]. Nodes in a mesh network running MORE randomly mix packets before they are forwarded, ensuring that routers that hear the same transmissions do not forward duplicate pack-

ets. When combined with opportunistic routing techniques [32], this allows MORE to deliver the opportunistic routing gains while maintaining clear architectural abstractions between the transport and link layers. MORE defines details such as the packet format and operation of intermediate nodes, demonstrating the practical integration of network coding into a transport layer protocol without modifying lower-level technologies. However, it is not possible to directly translate the operation of MORE, or any other mesh network protocol, to the general Internet.

2.4 Network Coding in P2P and Multi-Path Technologies

Since 2015, Microsoft have used a peer-to-peer protocol to deliver updates for Windows 10 [33]. Details of this protocol are not available, but it is likely based on Goldberg and Gkantsidis’ ‘Avalanche’ protocol [34]. Avalanche operates in a similar manner to BitTorrent, with hosts forming a swarm, simultaneously communicating pieces of the source file to one another. An important feature of BitTorrent is the replication of “locally rare” pieces first, however, Goldberg and Gkantsidis argue the system often fails to identify the optimal piece to replicate first. To combat this Avalanche uses network coding to distribute linear combinations of pieces of the file, rather than the distinct blocks. As introduced in section 2.3, this means any piece uploaded by a peer can be of use to *any* other peer. Rather than needing a copy of each individual piece to complete, a host simply needs sufficient independent linear combinations to be able to decode. Goldberg and Gkantsidis demonstrate the gains that can be gained using this technique, although their results have received criticism [35].

CTCP (“Coded TCP with Multiple Paths”) was proposed by Kim et al. in 2012 as an attempt to incorporate the advantages of multiple paths and network coding into TCP - maintaining TCP’s good features such as congestion control and reliability [36]. The authors draw on Sundararajan et al.’s

TCP/NC [8], however, rather than introducing coding as a shim layer, coding is directly included in the transport layer. In this way, coding concepts can be included directly into the congestion control algorithm, introducing a novel token-based congestion control mechanism. The authors demonstrate the advantages of CTCP over traditional TCP in environments with multiple paths, and high packet-loss.












Location	Avg Time	Status
Amsterdam	26 ms	
Bangalore	14 ms	
Cape Town	1 ms	
Frankfurt	9 ms	
London	6 ms	
Moscow	17 ms	
New York	1 ms	
New Zealand	26 ms	
San Francisco	10 ms	
Singapore	110 ms	
Toronto	14 ms	

Figure 2.1: Subset of geographical locations of YouTube servers. Collected using PingTools GeoPing utility [1].

Chapter 3

Multi-Source, Multi-Path TCP with Linear Network Coding

In this chapter, we present a description of *RePT*, our multi-source, multi-path, network coded evolution of TCP. We first provide, in section 3.1, a high-level walkthrough of the behaviour of a *RePT* data transfer. We then describe the details of algorithms used and the operation of individual components in section 3.2. A reference implementation has been produced, implementing the behaviour described here.

A summary of the notation used is provided in table 3.1

3.1 Overview of *RePT*

RePT is a transport protocol, designed for the transport of content which may or may not be replicated at multiple diverse locations, to a host which may or may not be multihomed. For example, such a situation might arise for a user watching a YouTube video on their mobile phone. The content may be replicated amongst a subset of the YouTube servers, and the user's mobile phone able to connect through both WiFi and 3G networks. As a transport protocol, *RePT* achieves all the same goals as TCP (reliability,

Table 3.1: Notation used in discussion of *RePT*

Symbol	Interpretation
i	Batch Number
j	Block within batch, $j \in \{1 \dots n\}$
k	Coded data block within batch $j \in \{1 \dots m\}$
n	Size of batches
m	Number of requests sent for each batch $m \geq n$
b_{ij}	The $j^{\text{th}} \in \{1 \dots n\}$ native block of batch i
d_{ik}	The $k^{\text{th}} \in \{1 \dots m\}$ coded block of batch i
c_{kj}	The $k^{\text{th}} \in \{1 \dots m\}$ coefficient of block j

flow control, etc. [37]), while exploiting additional techniques to maximise resource pooling within the network.

The primary components of a *RePT* connection are:

- A ‘Requester’: the host trying to download a file.
- One or more ‘Responder(s)’: the sources who have a copy of the file requested.
- One or more ‘Forwarder(s)’: *RePT*-capable intermediate nodes in the network.

Each of these components, and how they interact, is shown in figure 3.2. The requester operates one or more independent ‘connections’ - one for each path to each source. The connections perform standard TCP operations along their path to the responder, including congestion control and providing reliable (see section 3.2.1.3), in order ¹, delivery of packets. To a non-*RePT* intermediate node in the network, each path between requester and a responder appears independent; having the appearance of a stream of TCP packets and acknowledgements with consecutive sequence numbers. This meets the ‘network-compatibility goal’ formulated by the MPTCP working group [12], and is desirable for the widespread adoption of such a protocol. If there is

¹When data is mixed together there is no concept of ‘order’. However, packets in a TCP stream have sequence numbers which represent an order, and so the connection maintains this ordering.

more than one connection, the requester acts as a mediator between the connections, identifying the next blocks to be requested and decoding received data.

3.1.1 *RePT* Flow Establishment

We imagine a host who wishes to access a piece of content replicated in multiple locations. This host learns the location of the first responder using the same technology as in the current Internet - a Domain Name System (DNS) lookup. Many large content providers use a front-end DNS server to direct requests to the ‘optimal’ server at that time, according to some heuristic. This configuration is shown in figure 3.1, in which a host is requesting a data item replicated at 2 sources. The host initiates a *RePT* requester, which in turn initiates a connection to the first responder using the standard SYN, SYN+ACK, ACK exchange of TCP. The initial SYN packet from the requester will include the identifier of the desired content, a unique ‘flow’ identifier, and will set the *RePT* flag within the TCP header. If the source does not support *RePT*, then the requester will default to traditional TCP.

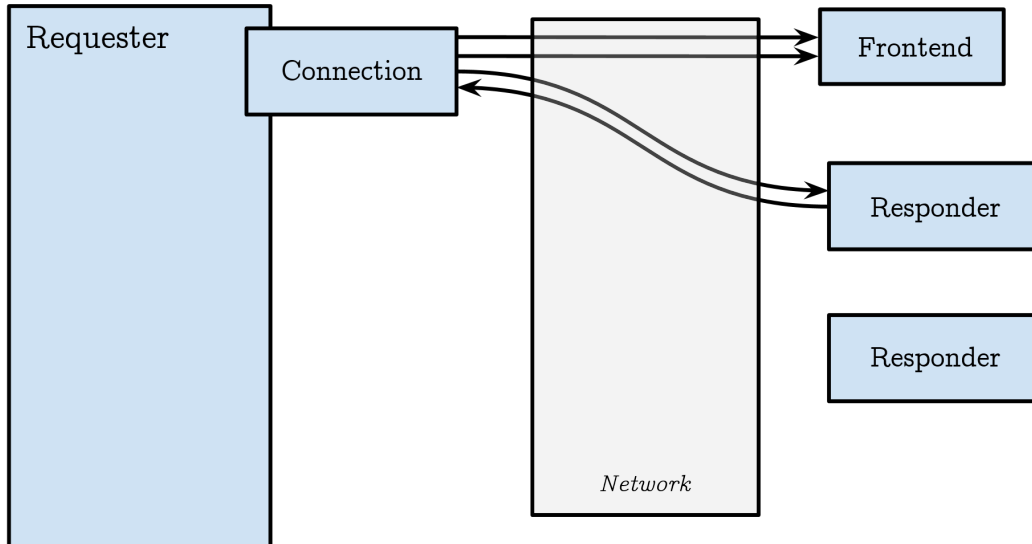


Figure 3.1: *RePT* requester establishing first connection with first source.

However, if the source is able to support *RePT*, the **SYN+ACK** generated in response to the **SYN** will contain a *RePT* Information Packet. This packet contains the size of the file requested, and the details of other sources able to serve the file. The responder implicitly interprets the file requested as a series of data blocks of a fixed size (by default 800 bytes, although the final block may be smaller), and groups these blocks into batches of a fixed size n (by default $n = 16$, although the final batch may be smaller). It may be necessary for negotiation of these and other parameters, such as whether any sources should be preferred to others, to occur, however, at present default values are assumed. *RePT* contains a number of layer violations, requiring application-level information such as file sizes and identifiers. However, this information can be gathered by the socket, such that applications need not be altered in order to use *RePT*.

The **SYN+ACK** packet, along with the *RePT* Information Packet is sent through the network from the responder to the requester. There may be one or more *RePT* forwarders on the path between the requester and responder. A forwarder which sees a **SYN+ACK** for a *RePT* connection, initialises a buffer for the flow, identified using the unique flow ID in the packet header, and prepares to perform recoding of packets in the network.

The connection delivers the *RePT* information packet to the requester, which is used to finish initialising the flow. The requester then initialises connections over additional paths and to additional sources, again using the standard **SYN**, **SYN+ACK**, **ACK** exchange of TCP. Any forwarders on these additional paths initialise buffers for the flow as before. If any forwarder lies on the paths of multiple sub-flows, these sub-flows share buffers, maximising opportunities for recoding. The result of this initialisation is shown in 3.2.

3.1.2 Data Transfer

Once a *RePT* flow has been set up, consisting of one or more connections, the transfer of the data can begin. The format of packets in *RePT* is shown in figure 3.3. To a non-*RePT* capable node in the network, these packets

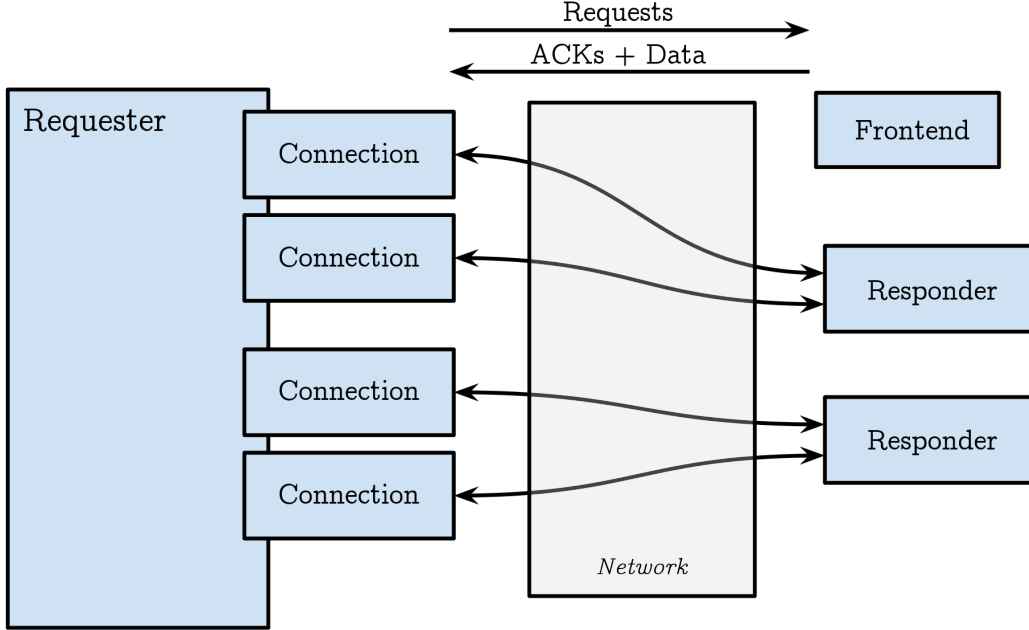


Figure 3.2: *RePT* requester with a connection for each path to each responder.

appear as standard TCP packets with a number of options.

A connection is only permitted to transmit requests if the congestion control allows. Whenever it is permitted to transmit, the connection sends a *RePT* packet, containing a request for a linear combination of blocks in a given batch². In the *RePT* header (figure 3.3), the flow ID is set to the unique identifier generated by the first connection, and the batch being requested is recorded. Packet type is set to 1, indicating it is a request packet. Since this is a request packet, no data is transported, and so the length fields are set to 0, and the code vector and data fields left empty.

On receipt of a request packet, the responder generates an acknowledgement as if it were standard TCP: verifying the checksum, generating a new header, and copying across the fields from the received packet as specified in the TCP standard [37]. Once this is complete, the responder uses the flow

²This ‘lowest-RTT approach’ is a trade-off of complexity to efficiency, based on Paasch et al.’s evaluation of MPTCP schedulers [38]. It is left to further work to investigate alternatives.

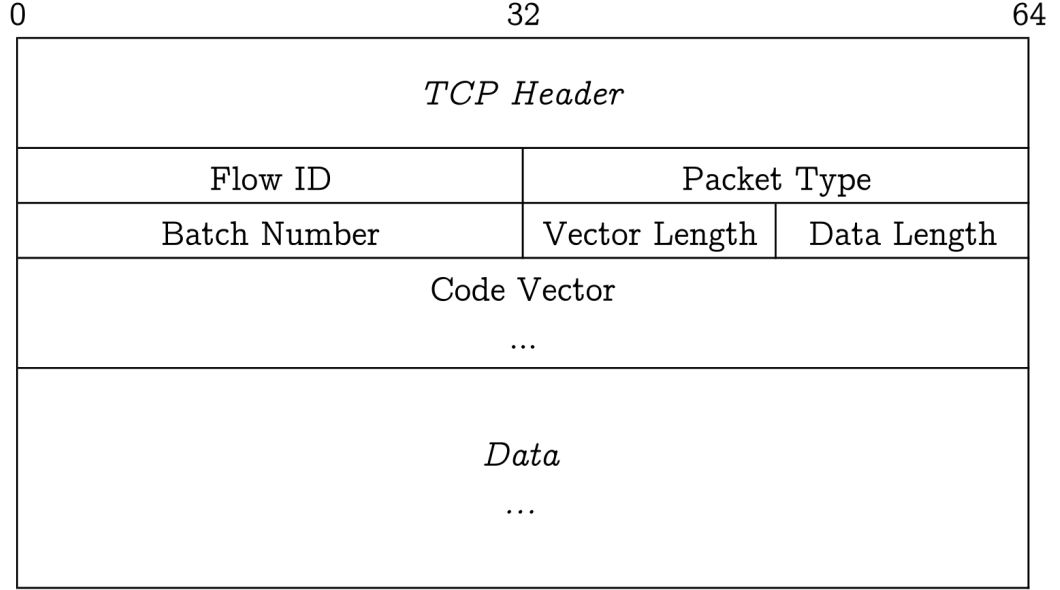


Figure 3.3: *RePT* Packet Format.

identifier to access the file being transmitted, and reads every block b_{ij} of the batch i specified in the request. The bytes of the file constituting the blocks are interpreted as integers, multiplied by coefficients chosen uniformly at random from a finite field, and added together to produce a random linear combination of the blocks. The result of this operation is a coded data block $d = \sum_j c_{kj} b_{ij}$, where c_{kj} is the randomly generated coefficient of block b_{ij} . The coded data d is placed in the data field of the *RePT* packet. A code vector is generated, consisting of n block-coefficient pairs, relating to the randomly generated coefficients c_{kj} and the number of their corresponding blocks within the batch. The coded data block d and the code vector are placed in their corresponding fields in the *RePT* packet, and their lengths recorded. The packet is then sent through the network.

Forwarder nodes listen to all transmissions. When a forwarder hears a response packet, it checks whether the packet contains new information; that is it checks whether the code vector is linearly independent from the packets previously received for the batch. Checking for independence is done using simple algebra (section 3.2.1.5). Packets which contain new information are added to a flow buffer. On the arrival of a new packet, the new data

block is added, with a random coefficient, to a pre-coded combination of the buffered blocks in the batch, and this recoded block transmitted. This new data block overwrites the data field of the received packet. The code vector, vector length, and data length fields are overwritten as appropriate for the new linear combination.

On receipt of a response packet at the requester, the connection first performs congestion control operations; verifying the checksum, checking for lost or reordered packets, and updating the congestion window. The packet is then passed to the requester. The requester verifies the data is linearly independent using the same technique as forwarders (section 3.2.1.5), and if so stores the coded block d and its corresponding code vector in a buffer for the batch. If n packets have been received for the batch, then the batch can be decoded using LU decomposition [39], and the decoded blocks output. Once every batch has been decoded, the requester closes all connections using the standard TCP **FIN**, **FIN+ACK**, **ACK** exchange, and terminates. Forwarders which hear a **FIN** for a flow for which they have a buffer initialised delete their buffers for the flow.

3.2 *RePT* Details

In the previous section we presented an overview of the behaviour of a *RePT* flow. We now present a more in-depth view of the individual operation of components, including description of algorithms used and implementation details.

3.2.1 *RePT* Requester

The majority of the complexity of *RePT* lies within the requester, which mediates between a number of connections, generates requests for linear combinations of blocks within a batch, and decodes received linear combinations.

Algorithm 1: *RePT* requester algorithm for deciding which batch to request.

```

batch ← batchTimeout()
if batch ≠ -1 then
  | return batch
end
batch ← nextBatch
if batch ≤ batchesInFile then
  | nextBatchReqs ← 1.1 · (nextBatchReqs +  $\frac{p_{path}}{1-p_{path}}$  - 1)
  | if nextBatchReqs < 0.5 then
  | | nextBatch ← nextBatch + 1
  | | nextBatchReqs ← nextBatchReqs + n
  | end
else
  | complete ← True
  | for i ∈ {0, ..., batchesInFile} do
  | | if receivedBatches[i] ≥ n then
  | | | batch ← i
  | | | complete ← False
  | | | break
  | | end
  | end
  | if complete then
  | | return null
  | end
end
return batch

```

Within each connection the requester must estimates packet loss-rate, ensure reliable delivery of packets, and perform congestion control.

3.2.1.1 Batch to Request

Whenever a connection can transmit a request, it calls the *nextBatch()* method of the requester, which returns the number of the next batch to be requested. Pseudocode of this method is shown in algorithm 1.

The role of this method is to select the batch for which we will send a

request. When network coding, packet carry independent linear combination of data, and so we can inject additional, redundant packets into the network to overcome any packets loss. For a connection with an average loss rate of p , it can be shown that a redundancy factor $R \sim \frac{1}{1-p}$ is necessary to overcome the losses [8]. Therefore, when generating the code vector, the requester takes each batch in turn and makes $n \cdot \frac{1}{1-p}$ requests for each batch in order to overcome any packet loss. The efficiency this approach depends on batch size, and uncoded packets can be thought of as having a batch size of 1. With small batches, there will be high variance in the number of packets dropped per batch, and so injecting additional packets is unlikely to be effective.

However, in a multipath environment the end-to-end packet loss will be different on each path, and may change over the course of data transmission. Therefore, it is not possible to pre-compute the redundancy factor necessary, since we do not know how many requests from each batch will be sent on each path, or the drop rate at any given time. To achieve this property, the number of requests to be sent is initialised to n ; the minimum number of combinations needed to decode the batch. Then, whenever a request is generated to be sent on a given path, the redundancy necessary to account for the packet loss on that path, $R \sim \frac{1}{1-p_{path}}$, is calculated, and additional requests for the batch sent as necessary. In this way, the redundancy is the weighted sum of the redundancy of each path, where the weights are the proportion of requests sent on each path. To the best of our knowledge, such an algorithm to dynamically calculate redundancy across multiple paths has not been described previously.

Each connection estimates its average packet loss, using the method described in section 3.2.1.2. However, this estimated value is an average, and at any point in time the instantaneous packet loss will vary from the average. *RePT* has two mechanisms to account for this. First, the redundancy is multiplied by a scale factor, by default 1.1, to account for instantaneous packet loss higher than the average. This will increase the number of packets transmitted which do not contain useful information, but reduce the latency to decode each batch. Second, the first line of the *nextBatch()* method checks whether

there are any batches for which the initial $n \cdot R$ requests have been made, but insufficient linear combinations have been received within a timeout proportional to the round-trip time of the slowest connection. If this is the case, an additional request for the batch is made. Once requests for every batch have been made, the requester continues making requests for any batches for which insufficient data has been received, until sufficient linear combinations have been received for all batches. Once every batch has at least n coded packets, the connections are terminated.

3.2.1.2 Estimating Packet Loss

The *RePT* requester depends on knowing the end-to-end packet loss rate on each path, p_{path} , in order to calculate the necessary redundancy. However, p_{path} is rarely known before time, and may fluctuate over time and space. Therefore, each connection estimates the drop rate of its path, using the following formula:

$$p_{path} \leftarrow p_{path} \cdot (1 - \mu)^{losses+1} + (1 - (1 - \mu)^{losses})$$

where μ is a smoothing factor. This formulation is a modified version of the exponential smoothing technique, extended to account for the fact each acknowledgement represents a success, and zero or more losses. The data series can be thought of as a sequence of 0's and 1's, where 0 indicates a packet loss, and 1 a successful delivery. Now assume there were $losses$ number of packets lost. If $losses = 0$, then the update equation becomes:

$$p_{path} \leftarrow p_{path} \cdot (1 - \mu) + 0$$

If $losses = 1$, the same update equation becomes:

$$p_{path} \leftarrow p_{path} \cdot (1 - \mu)^2 + \mu = (1 - \mu)[p_{path} \cdot (1 - \mu) + 0] + \mu$$

which is identical to executing exponential smoothing over two data points - one lost and one acknowledged. We can repeat this idea for $losses > 1$ to

obtain the formula presented above.

3.2.1.3 Reliable Communication

A primary feature of TCP is the reliable and in-order delivery of packets. Specifically, “*TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the Internet communication system*” [37]. This is achieved in TCP using checksums, sequence numbers, and acknowledgements (ACKs). If an ACK is not received within a timeout interval, the data is retransmitted. The sequence numbers are used to correctly order packets received out of order.

With network coding, there is no concept of packet order, since every packet is a linear combination of multiple blocks. Furthermore, *RePT* sends additional requests to overcome any packet loss in the network, removing the need to retransmit packets lost in transmission. However, the network compatibility goal of the MPTCP specification emphasises the need for individual connections to behave, from the perspective of intermediate nodes, as if they were traditional TCP connections [12]. As such, if a *RePT* requester detects loss in the network, it will retransmit a packet with the same sequence number as in traditional TCP. However, this newly retransmitted packet *is not necessarily* identical to the one that was lost. Instead, the requester generates a new request, calling the *nextBatch()* method (section 3.2.1.1). The *nextBatch()* method is designed to overcome this packet loss in the number of requests it makes for each batch, and so this new request is entirely independent to the one that was lost. The sequence number of the lost packet is assigned to this new request, maintaining the appearance that it is a retransmission of a previous packet.

In TCP, if the ACK for the original packet, which the requester had thought was lost, is later received, then the retransmitted packet will have been unnecessary and waste bandwidth. In *RePT*, the data received in response to both the original and retransmitted request is a different linear combination, and so both can be used for decoding.

3.2.1.4 Congestion Control

Congestion control, along with network coding, is one of the primary components of *RePT*, allowing resource pooling in response to changing network conditions on the granularity of packets. To achieve this each connection must maintain its own congestion control state (i.e. congestion window *cwnd*). However, if we were to simply run a congestion control algorithm from traditional TCP independently on each connection, then the *RePT* flow would be able to gain more than its ‘fair share’ at any bottleneck links traversed by more than one of its connections, as measured by Jain’s fairness index [40]. Furthermore, it is desirable that a *RePT* flow using multiple sources and paths transfers more traffic using the least congested of its connections, achieving the resource pooling, while achieving cumulative throughput of at least the level of traditional TCP.

In section 1.1 we described *RePT* as ‘MPTCP cut in half.’ In practice, this means a number of the congestion control algorithms developed for MPTCP are also applicable to *RePT*. With the initial proposal of MPTCP, Raicu et al. proposed a congestion control algorithm based on the principles of TCP-Reno [41]. However, more recently Cao et al. developed *wVegas*, an MPTCP congestion control algorithm based on TCP-Vegas [42]. Cao et al. demonstrated that *wVegas* is more sensitive to changes in network congestion than loss-based algorithms. Therefore, *RePT* uses the *wVegas* algorithm for congestion control.

3.2.1.5 Verifying Linear Independence of Packets

When a new packet is received, checking whether the coded data is useful implies checking whether the received packet is linearly independent of the set of packets from the same batch already stored at the node. This is the same at both forwarders and the requester.

Each node keeps code vectors of the packets in its buffer as a triangular $(n \times n)$ matrix M , with some of the rows missing. For each stored row,

the smallest index of a non-zero element is distinct. To check if the code vector of a newly received packet is linearly independent, standard Gaussian Elimination is used so that consecutive elements of the vector become 0. If the vector is linearly independent, then one element will not be zeroed due to a missing row, and the modified vector can be added to the matrix in that empty slot. This operation is shown in algorithm 2, where u is the received vector, M is the triangular matrix as described above, and n is the size of the vector (the batch size).

Algorithm 2: Checking for linear independence of a vector u

```

for  $i \in \{1, \dots, n\}$  do
  if  $u_i \neq 0$  then
    if  $M_i$  exists then
       $u \leftarrow u - M_i u_i$ 
    else
      admit the modified block to memory
       $M_i \leftarrow \frac{u}{u_i}$ 
      return True
    end
  end
end
return False

```

This complexity of this operation is $\mathcal{O}(n^2)$, and is used at both the requester and forwarder nodes. As such it is desirable to use small coefficients, which reduce the time of each individual operation. However, it is worth noting that this operation does not touch the coded data block, and only operates on the code vector. A similar approach is proposed by Chachulski et al. [31].

3.2.1.6 Decoding Packets

To understand the decoding of packets, it is necessary to understand how packets are encoded. Although not performed explicitly, the coding operation can be thought of in terms of matrix operations. The responders collectively divide the file into a series of ‘blocks’, and interpret each of these blocks as an integer. These blocks are then grouped into batches of size n , and

interpreted as vectors $B_i = (b_{ij})$ where b_{ij} is the $j^{\text{th}} \in \{1, \dots, n\}$ block of batch i . Coding is then performed across each batch. To perform coding, the responder multiplies the randomly generated code vector C_k , by the vector of blocks B_i ; generating the coded data block d_{ik} . This can be written in vector form as:

$$C_k \cdot B_i = d_{ik}$$

$$\begin{pmatrix} c_{k1} & c_{k2} & \dots & c_{kn} \end{pmatrix} \cdot \begin{pmatrix} b_{i1} \\ b_{i2} \\ \dots \\ b_{in} \end{pmatrix} = d_{ik}$$

In section 3.2.1.1 we noted that the code vectors can be thought of as rows of an $(m \times n)$ coding matrix C , where $m \geq n$. Therefore, the coded data d_{ik} are the elements of the vector which results when we multiply this coding matrix C with the vector of blocks $B_i = (b_{ij})$; generating a vector of coded data $D_i = (d_{ik})$. This can be written as matrix form as:

$$C \cdot B_i = D_i$$

$$\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_{i1} \\ b_{i2} \\ \dots \\ b_{in} \end{pmatrix} = \begin{pmatrix} d_{i1} \\ d_{i2} \\ \dots \\ d_{im} \end{pmatrix}$$

The responder sends through the network the code vector C_k along with its corresponding coded data block d_{ik} . Due to packet loss in the network, some of these packets will be lost, and so the requester will only receive a subset the transmitted coded data and code vectors. However, once n pairs have been received the requester can construct the vector D'_i , the elements of which are a subset of the original coded data vector D_i , along with the corresponding

$(n \times n)$ coding matrix C' , the rows of which are a subset of the rows of the code matrix C . With these constructed, the original data blocks B_i can be recovered by taking the inverse of the matrix C' , and multiplying this by the coded data vector D'_i . This can be written in matrix form as:

$$C'^{-1} \cdot D'_i = B_i$$

$$\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}^{-1} \cdot \begin{pmatrix} d_{i1} \\ d_{i2} \\ \dots \\ d_{in} \end{pmatrix} = \begin{pmatrix} b_{i1} \\ b_{i2} \\ \dots \\ b_{in} \end{pmatrix}$$

In this way, the original data blocks can be recovered from the coded data blocks. Due to the linear independence of rows, the matrix C is guaranteed to be invertible, and the ordering of the received rows does not matter - as long as the pairs of coded blocks and code vectors maintain the same ordering within D'_i and C' respectively.

In practice, decoding at the requested is performed using LU decomposition, rather than matrix inversion [39]. However, it is useful to think in terms of matrix operations to conceptualise the coding and decoding operations.

3.2.2 *RePT* Responder

Due to the fact that most of the complexity lies in the Requester, the *RePT* responder is very simple, simply performing the operations instructed by the packets received from the requester. On receipt of a request, the responder first verifies the checksum, and then generates a TCP ACK packet, generates a linear combination of blocks from the requested batch, and sends these to the requester.

Each connection at a responder is treated independently, even if the connections are part of the same *RePT* flow, using multiple paths from the requester.

3.2.2.1 Generating Response Packet

On receipt of a packet, the responder must generate an acknowledgement packet in order to facilitate the operation of the reliable transport and congestion control algorithms at the requester. This process is the same as in standard TCP, with the responder indicating the sequence number of the packet it received, and the sequence number of the next packet it is expecting. The sequence number of the next expected packet will usually be 1 greater than the sequence number of the packet being acknowledged, however if any packets have been lost or reordered this may not be the case. Details can be found in the TCP standard [37] (or any good networking textbook).

3.2.2.2 Coding Operations

Section 3.2.1.6 provided a conceptual description of the coding and decoding operations, in terms of multiplying a code matrix by the vector of blocks in the batch. In practice, this operations is distributed between all sources. On receipt of a packet, the responder reads each block from the file, interprets the bytes as an integer, multiplies each by a random coefficient, and adds them together. As such, the coding matrix C described in section 3.2.1.6 is never explicitly constructed. Rather, the rows of the matrix are constructed at various times and at different responders across the network. Ho et al. demonstrated that when generating random coefficients, if coefficients are chosen from a sufficiently large finite field \mathbb{F} , the probability of obtaining linearly independent combinations (and therefore ensuring the packet confers new information) approaches 1 [19]. As such, there is no need for coordination between the responders during the transmission of the data.

Blocks of the file are by default 800 bytes. Therefore, when interpreting these blocks as unsigned integers the resulting numbers are often *extremely* large, lying within a range of $2^{8 \times 800} - 1 \approx 10^{1927}$ numbers. Schoolbook long multiplication has a complexity of $\mathcal{O}(n^2)$, where n is the number of digits of the inputs, and so it may be desirable to use smaller block sizes in order to

reduce memory and processing overheads. For this reason, it is also desirable to use small coefficients, preventing a significant increase in block size due to multiplication.

3.2.3 *RePT* Forwarder

RePT forwarders are switches, routers, proxies, and middleboxes which allow and encourage the recoding of data blocks in the network in a way which is largely invisible to other nodes in the network. Forwarders listen to all transmissions. If a forwarder hears a packet for a flow they are participating in, the forwarder checks if the packet contains new information using the method described in section 3.2.1.5, and replaces the data and code vector in the received packet with a new, pre-encoded packet, before forwarding the packet to the next hop.

3.2.3.1 Pre-Encoding Packets

Each forwarder maintains a buffer of received data blocks and code vectors for each flow it is participating in. The forwarder also maintains a pre-encoded random linear combination of coded data blocks it has previously received for each batch. Note that a linear combinations of coded data blocks will also be a linear combination of the corresponding uncoded blocks. In particular, assume that the forwarder has previously buffered data blocks of the form $d_{ik} = \sum_j c_{kj} b_{ij}$, where b_{ij} is the j^{th} block of the batch i^{th} batch of the file. The linear combination of a set of data blocks is $d'_{il} = \sum_k r_{lk} d_{ik}$, where r_{lk} 's are random numbers. This can then be expressed in terms of blocks of the file as follows:

$$d'_{il} = \sum_k (r_{lk} \sum_j c_{kj} b_{ij}) = \sum_j (\sum_k r_{lk} c_{kj}) b_{ij}$$

Similarly, the new code vector can be calculated as a linear combination of the previous code vectors. Therefore, the recoded packet is a linear combination

of the uncoded data blocks. This method avoids the need to decode the original blocks at each node, significantly reducing the complexity of the operations performed at forwarders.

Whenever the forwarder receives a packet it adds the received coded data block to the pre-encoded packet, and forwards this through the network. The forwarder then pre-encodes a new packet using the buffered coded data blocks.

3.2.3.2 Enabling Network Coding on non-*RePT* Connections

It was proposed during development that forwarders could be used to introduce certain features of *RePT* when communicating with responders which otherwise would be not *RePT* compatible. In section 3.1 we noted that if a responder is not *RePT* compatible, then communication defaults to standard TCP. However, if a forwarder lies on the path then it can act as a proxy, storing a buffer of data blocks from the responder and generating linear combinations in response to requests as shown in figure 3.4. The forwarder can effectively maintain 2 separate connections, one *RePT* connection generating linear combinations in response to requests from the requester, and one standard TCP connection receiving data from the responder.

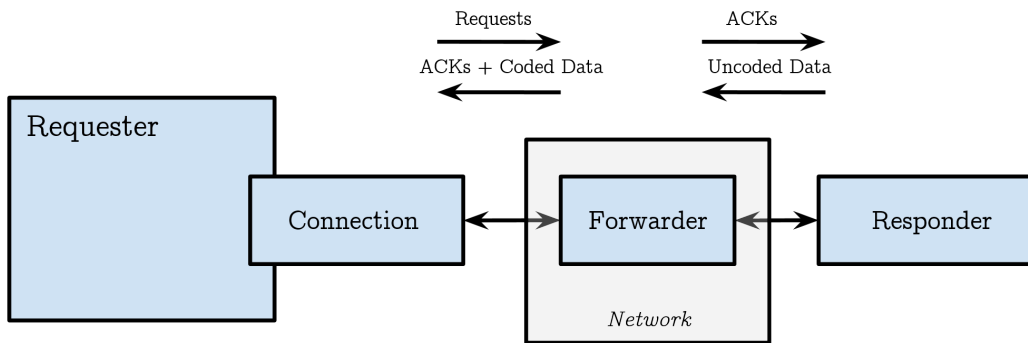


Figure 3.4: *RePT* forwarder forming linear combinations of uncoded packets.

Exploration of this proposal is left to further work (section 5.1, and not considered any further here.

3.3 Structured Coding Approach

During the development of *RePT* we explored a structured coding approach, as opposed to the random coding approach described above. Rather than having the responder use random coefficients to construct a linear combination of blocks within a batch indicated by the requester, we propose having the code vector generated at the requester in a systematic way, and communicated to the responder in the request packet. The responder then constructs the coded data block using the code vector provided, and transmits this data block through the network. The requester already knows the code vector, and so the code vector need not be transmitted with the data.

This approach has many potential advantages. The random approach works since, as the size of the field coefficients are chosen from increases, the probability of generating linearly independent vectors approaches 1 [19]. However, due to the need to send the code vector along with the data it is undesirable to use coefficients from a very large field, since this will reduce the proportion of bandwidth carrying coded data. As previously mentioned, small coefficients are also desirable to reduce the complexity of coding operations and checking linear independence at forwarders and the requester. To mitigate the impact of large coefficients, we use random coefficients from a smaller field, increasing the likelihood of generating code vectors which are not linearly independent. The mixing of packets in intermediate nodes, as described in section 3.2.3, will mitigate this, but introduces additional latency and resource demands in the network.

A structured approach allows us to limit the size of coefficients, and so the code vector, while guaranteeing coded blocks are linearly independent. In this way, we remove the need for the recoding of packets in the network, since data will always be linearly independent. Furthermore, the code vector is included with the request packet, rather than the response. This reduces the size of the response packet, at the cost of a larger request packet, potentially allowing for the use of large block sizes and reducing the total number of transmissions. Reducing the variation in packet sizes in the network is desirable to reduce

the variation of queue sizes in the network, allowing more accurate network measurements.

To achieve these properties, we replace calls at the requester to the *getBatch()* method, which returns the next batch to request as described in section 3.2.1.1, with calls to a *getCodeVector()* method, pseudocode for which is provided in algorithm 3. The function returns the code vector to be communicated to the responder.

Algorithm 3: *RePT* requester algorithm for generating a code vector.

```

batch  $\leftarrow$  getBatch()
coefficientVector[]  $\leftarrow$  generateCoefficients(batch)
firstBlock  $\leftarrow$  batch  $\cdot$  n
for i  $\in$  {0, ..., n - 1} do
    | codeVector[i]  $\leftarrow$  (firstBlock + i, coefficientVector[i])
end
return codeVector

```

The *getCodeVector()* method chooses the batch to request in the same way as previously, calling the *getBatch()* method shown in algorithm 1. However, line 2 of algorithm 3 calls a function *generateCoefficients()*, which returns a list of *n* coefficients; one for each block in the batch. This method generates linearly independent code vectors, such that the coefficients are sufficiently small. To the best of our knowledge, this problem has not been addressed previously in the literature.

We approach the problem as follows. As discussed in section 3.2.1.6, the coefficients of the linear combinations can be thought of as the rows of an $(m \times n)$ matrix *C*, where $m \geq n$ is the number of requests, including any redundancy, made for a batch. In order to decode the received linear combinations, it is necessary to obtain an invertible $n \times n$ matrix *C'*, such that the rows of *C'* are a subset of those of *C*. For this matrix to be invertible, every row must be linearly independent. In order to guarantee code vectors with this property, elements c_{kj} of the matrix *C* are generated by the formula:

$$c_{k,j} = k^{j-1} \mod p$$

where p is a prime such that $p \geq m$. Proof that all rows are linearly independent is provided in appendix A.

Clearly, no element in C exceeds $p - 1$, and so setting $p = 257$ allows us to generate coefficients which fit inside a byte, and to generate a matrix C with dimensions $(256 \times n)$. For each batch, the number of packets transmitted, including redundancy, is calculated as $n \times \frac{1}{1-p_{drop}}$, where p_{drop} is the average end-to-end loss rate (section 3.2.1.1). With 256 rows of C we are able to generate independent linear combinations for a loss rate of:

$$n \cdot \frac{1}{1 - p_{drop}} \leq 256 \Rightarrow p_{drop} \leq 1 - \frac{n}{256}$$

For $n = 16$, $p_{drop} \leq 0.9375$, and for $n = 64$, $p_{drop} \leq 0.75$. These loss rates are significantly higher than can be expected in any practical network. However, if a requester ever sends over 256 requests for a batch, it can default to generating random coefficients as demonstrated by Ho et al. [19]. The matrix C can be precomputed and stored, avoiding performing a large number of expensive operations at runtime.

To the best of our knowledge, such a method for generating linearly independent vectors with coefficients within a certain range has not been proposed previously in the literature. It is left to further work to compare when it is more efficient to use a structured or random approach (section 5.1).

Chapter 4

Experimental Evaluation

In this section we evaluate the resource pooling achieved in *RePT*, focussing on demonstrating the ability of the protocol to use resource pooling to provide an improved quality of service in a manner not possible for traditional TCP. A reference implementation of *RePT* has been produced in Java and is used within an emulation of a network to demonstrate the practical use of *RePT* and quantify the advantages that can be attained. We evaluate the multi-sourcing, multi-pathing, and network coding features of *RePT* individually, before reviewing their combined impact. This evaluation is performed with respect to the performance of traditional TCP-Vegas within the simulation. The primary findings of our experiments are as follows:

- On average, *RePT* *always* performs at least as well as traditional TCP-Vegas, and much better in situations where resource pooling can be exploited.
- *RePT* is able to increase the load on paths in response to the loss of one or more connections to a source, resulting in 6 times reduction in the time to transfer a file when a source breaks.
- *RePT* is able to dynamically shift traffic placed on each path in response to dynamic network conditions. Specifically, when a source or path becomes heavily loaded, more traffic is placed on less loaded paths,

avoiding traffic on the heavily loaded path.

- In order to gain significant benefits from network coding, batches of at least 16 packets should be used. Batches of less than 16 packets increases the total time to transfer a file while only marginally improving the time to decode each batch

4.1 Simulation Environment

The evaluation of *RePT* is performed in an emulation of a network, running between processes on a host machine. The details of the host device can be found in table 4.1.

Table 4.1: Details of device used to perform Network Emulation

Property	Detail
Operating System	Ubuntu 14.04.5 LTS
Processor	Intel Core i5 760
Cores	4
Clock Speed	2.8 GHz
Memory (RAM)	8 GiB
Java Version	Open JDK 7 Ubuntu

We simulate the transfer of a 1 MB file replicated at 2 geographically distinct locations to a host which is itself able to communicate using 2 distinct technologies. Each path between requesting host and source contains a number of *RePT*-capable routers able to perform recoding in the network. Configuring the simulation in this way resembles the common situation of a user accessing a piece of content which is replicated in several places online, using a device which has access to multiple communication technologies.

The network topology is shown in figure 4.1, where squares are used to denote end hosts, and crossed circles used to denote routers implementing *RePT*. The arrows may contain additional switches, routers, proxies, and middle-boxes which do not implement *RePT*. From the perspective of these intermediate nodes, each flow is an independent stream of TCP Packets. Unless

otherwise stated, the end-to-end throughput, latencies, and packet loss rates are assumed to be the same on each path.

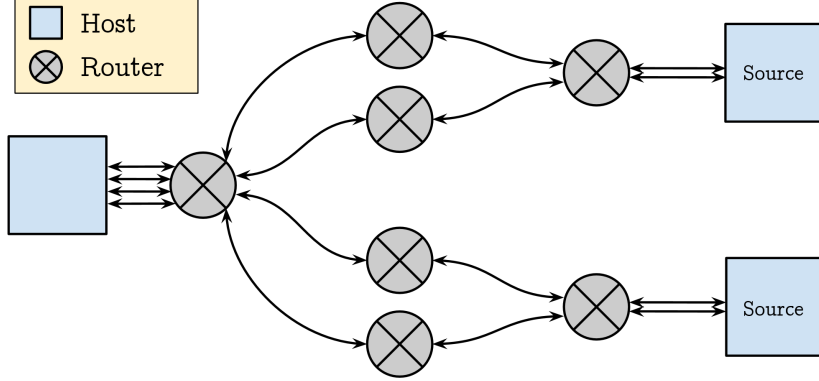


Figure 4.1: Network Topology in Simulation.

4.2 Effect of Multiple Sources and Paths

We begin by looking at the resource pooling achieved through multi-sourcing and multi-pathing. The use of multiple sources and paths allows us to load balance on the granularity of individual packets, rather than on the granularity of connections. This allows connections to adjust as network conditions change, and to maintain communication even when a path to a source is lost. In this way, the overall quality of service of communications on the Internet is improved, as links are pooled. In addition, this allows improved mobility, as users can disconnect from old sources and paths, and connect to new sources and paths as they move.

4.2.1 Losing Connection to a Source

Figures 4.2a and 4.2b show the total throughput of a *RePT* flow, and the throughput of the individual sub-flows, as the flow starts up and then loses connection to a source. The first column of figures shows the case in which

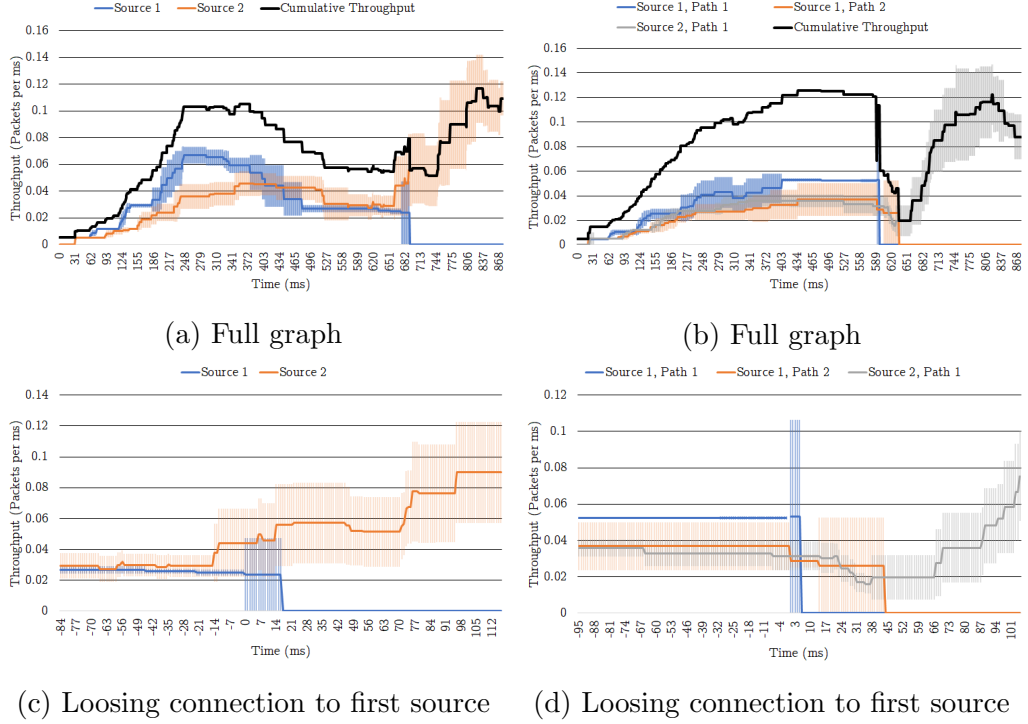


Figure 4.2: Connection starting up across multiple paths and sources, and loosing connection to first source. First column shows case of 1 path to each source, second column shows 2 paths to first source, and 1 path to second source. Error bars show 95% confidence interval.

a single path is available to each source, while in the second column, 2 paths are available to the first source, while a single path is available to the second. Throughput is calculated using the formulation $\text{Throughput} = \frac{\text{Window Size}}{\text{RTT}} \times \text{Packet Drop Rate}$. Since *RePT* maintains its congestion window in packets, the result of this equation is a rate of packets, although this can be converted to bytes by multiplying by the block size (set to 800 bytes in this simulation). We note that this does not represent the *goodput* of the connection; that is the graphs show the rate packets are transmitted, and not whether the data within the packets contains ‘useful’ information. The proportion of packets containing ‘useful’ information is briefly discussed in section 4.3.

In figures 4.2a and 4.2b we see that after the first connection starts up, there is a delay of approximately 1 round-trip time before the subsequent

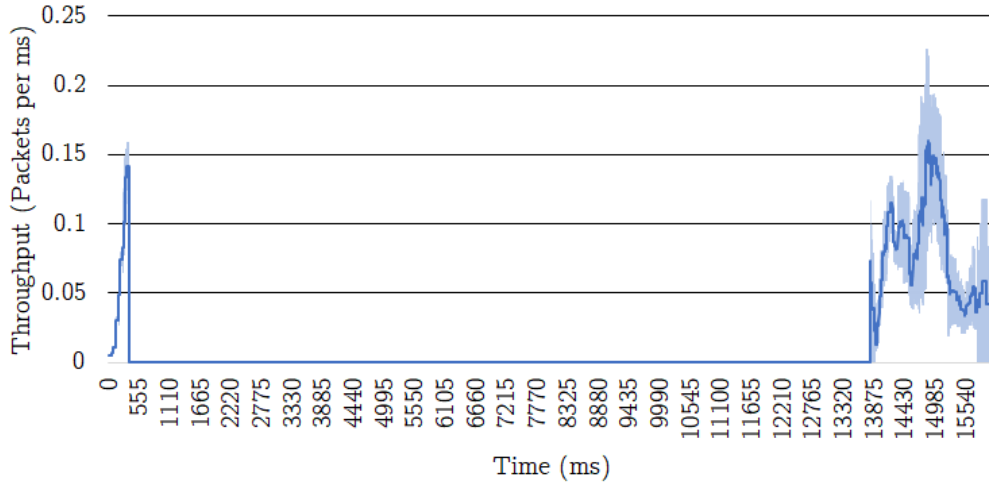


Figure 4.3: Source Breaking under Traditional TCP

connections begin. This is the result of receiving the addresses and paths available to each source as part of the **SYN+ACK** of the first connection - as described in section 3.1. Once each connection has established, data is approximately evenly distributed across each path, as desired. In simulation, each path has the same drop rate and latency, and this is reflected in the throughput on each path.

Approximately 600ms after the *RePT* flow begins, all communication with the first source is lost. This might be caused by a number of phenomena, such as denial of service attacks, natural disasters, or mobility at the source. The behaviour of *RePT*'s load balancing is evident in the behaviour of the connection to the second source, which remains active throughout. The throughput to the second source dramatically increases in response to the loss of connectivity to the first source, accounting for the loss of throughput on the paths to the first source. Visually, we can see that the cumulative throughput of all connections initially falls as connectivity is lost, but quickly returns to a level approximately equivalent to its condition before connectivity was lost.

Figures 4.2c and 4.2d focus on the throughputs around the time at which the source stop responding, with time denoted relative to the time at which the source breaks. There is a noticeable delay between the time at which

the source ‘breaks’ and the time at which the throughput of the paths to that source drops. This is a result of throughput being calculated using the current congestion window and measured round-trip time. These parameters are not reset until triggered by a timeout, resulting in a delay in detection of broken sources. This delay could be reduced by reducing the timeout time, however this would increase the probability of false positives, triggering when the packets are delayed, but the source is still available. Regardless of the delay, we see that for the duration of the connection, data is constantly being transferred, and the throughput on the surviving path can account for the loss in throughput on the other path.

If further sources were available, then *RePT* would be able to initiate new connections in order to return to a state of pooling resources across a number of paths and sources. Once these new connections had been established the load on each path would return to being evenly distributed across paths, according to the network properties at that time. The time taken to establish connections to new sources is the same as the time it would take for a traditional TCP connection to re-establish a connection to the source. However, for the length of this time *RePT* is able to continue receiving data at a high rate, while traditional TCP would be unable to receive any data until a new connection had been established. The behaviour of traditional TCP can be seen in figure 4.3. The improvement of *RePT* over traditional TCP is quantified in table 4.2. Speedups of 6.611 and 4.968 are achieved relative to traditional TCP. The case of using 2 paths to the source which breaks can be seen to be slightly slower, since the throughput on both paths to the source is lost when connectivity to the source is lost, and so the amount the other path must account for is larger. Despite this, it is still advantageous to maintain multiple distinct paths to each source in order to maximise resource pooling.

Configuration	Time to Transfer (s)	Speedup over TCP
Traditional TCP	16.092	1
2 Sources, 1 Path to Each	2.888	6.611
2 Sources, 2 Paths to first	3.851	4.958

Table 4.2: Time to Transfer 1MB in simulation when connection is lost to a source.

4.2.2 Heavily Loaded Path(s)

A less extreme but perhaps more common situation is that in which a source or path becomes heavily loaded during communication. In this case it is desirable to offload requests from the heavily loaded connection, on to other less heavily loaded connections. A demonstration of this behaviour in *RePT* can be seen in figure 4.4. Two sub-flows are established as part of a *RePT* flow, connecting to 2 different sources. After a period of time, the path to the first source becomes heavily loaded, increasing latency and packet loss rate on the path, and reducing the throughput available through that sub-flow. Therefore, the throughput of the second flow increases to account for this.

In the same way as was the case for a broken source, there is a slight delay between the time when the throughput of the first path is reduced, and when the throughput of the second path increases to account for it. Again, this delay might be reduced by reducing timeouts at the requester, but this may lead to incorrectly identifying delays or losses as a heavily loaded path.

4.3 Effect of Network Coding

Next, we look at the resource pooling achieved by network coding. Network coding, which pools data across packets in the flow, is used to inject additional packets into the network in order to overcome packets loss, without needing to know specifically which packet will be dropped. In order to reduce the time a user must wait before being able to decode their data, the file is divided into ‘batches’, and coding only occurs across blocks within these

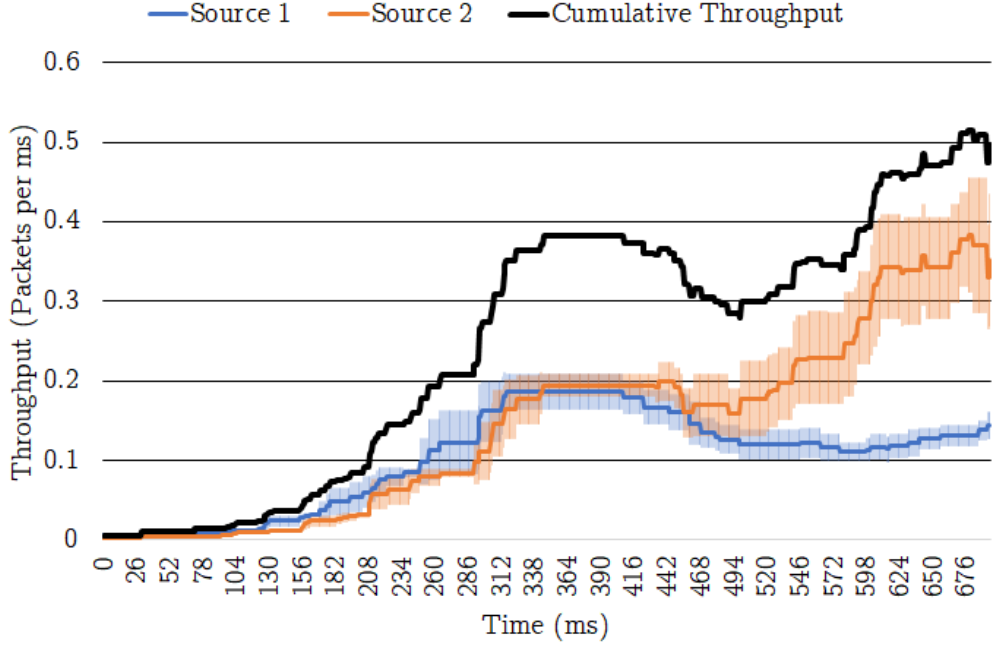
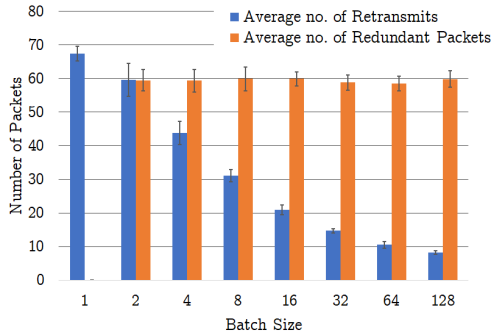


Figure 4.4: Connection to first source becomes heavily loaded, reducing throughput, and so traffic is shifted onto connection to second source.

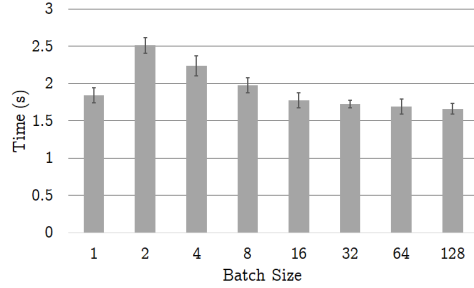
batches. This has the added advantage of reducing memory and processing overheads at both endpoints and intermediate nodes. We expect larger batches to more efficiently account for packet loss, since by the weak law of large numbers, as we send more packets, the proportion which are dropped will converge to the actual packet loss.

To quantify the effect of network coding in overcoming packet loss, we measure the number of additional requests made after all initial requests (including redundancy) have been made, for batch sizes ranging from 2 to 128. The number of sources and paths is set to 1, so that the experiment isolates the effect of network coding within *RePT*. The number of additional requests represents the number of packet losses which the injection of additional coded packets was not able to overcome.

Figure 4.5a shows the average number of redundant packets sent, and the average number of additional requests necessary once all initial requests have



(a) Number of Additional Requests



(b) Time to Transfer File

Figure 4.5: Effect of Batch Size on number of additional requests and time to transfer a file (excluding code/decode time). Error bars show 95% confidence interval.

been made. For a batch size of 1, the protocol behaves similarly to standard TCP, and so no redundancy is introduced. We see that the same amount of redundancy is introduced at each batch size. This is to be expected, as the redundancy is proportional to the packet drop rate, and independent of the batch size. It is quite clear that as the batch size increases, the number of additional requests decreases. For smaller batch sizes, increasing the batch size has a pronounced effect; between batch sizes of 2 and 4 there is a 26% reduction in number of additional requests. However, increasing batch size from 64 to 128 only provides a 22% reduction in number of requests.

Figure 4.5b shows the average time to complete transfer of the file. To isolate the effect of the resource pooling we exclude the time to code and decode packets, which is dependent on the host devices. Code/decode cost is briefly discussed in section 4.4. Introducing network coding with small batch sizes dramatically *increases* the time to complete file transfer, since a large number of redundant packets are being sent, but the probability these packets account for packets which are lost remains small. However, as the batch size increases, the redundant packets are more effective at overcoming packet loss, and so the total time to transfer the file decreases.

The role of network coding in *RePT* is to decrease the latency for individual

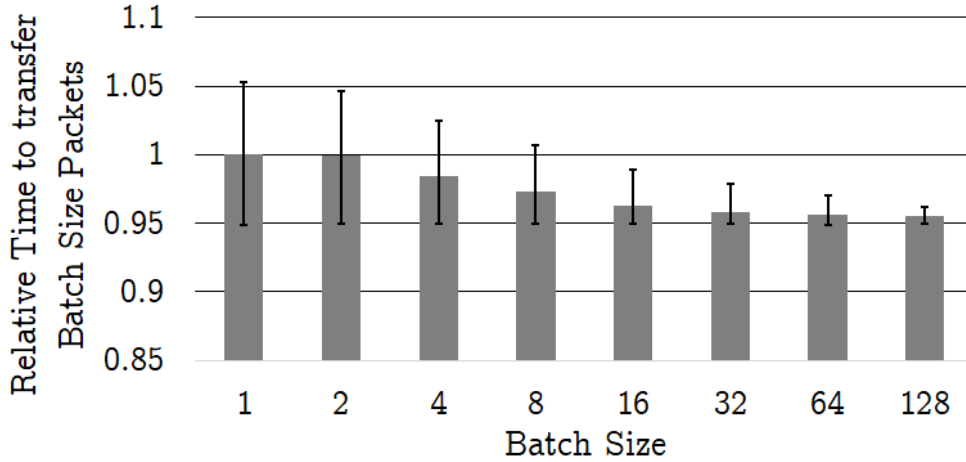


Figure 4.6: Average time between first request for a batch and receiving all *BatchSize* packets, relative to TCP

batches, as is desirable when streaming content. The effect of this is shown in figure 4.6, which shows the time between the first request for a batch being sent, and *BatchSize* packets being received, relative to TCP. We see that using a large batch size decreases the average time taken to receive an individual batch. This is as expected, since rather than having to wait for losses to be detected and then retransmitting packets, we send additional packets to overcome the packet loss in advance. An additional advantage of network coding is a reduction in the variance of packet loss, since this no longer depends on the proportion of packets lost and just on variance in the network.

4.4 Combining Multiple Sources, Multiple Paths, and Network Coding in *RePT*

Finally, we look at the operation of multi-sourcing, multi-pathing, and network coding in tandem. The impact of each of these resource pooling techniques has looked at individually, and so we focus on showing how they interact, and the combined effect of their use.

Figure 4.7 shows, for a number of configurations of *RePT*, the relative time to transfer our 1MB file relative to traditional TCP-Vegas under static network conditions with low packet loss and little other traffic. We see first that under no configuration tested does *RePT* perform worse than TCP; the time to transfer the file is lowered in each case. This meets the functional goal established out by the MPTCP working group of “doing no worse than TCP”. Multi-sourcing, multi-pathing, and network coding all contribute to this reduction. When using multiple connections, the combined throughput across all paths of *RePT* is defined to be the same as the throughput which traditional TCP would achieve on the ‘best’ sub-flow. Having a set of paths to choose from increases the chances of the ‘best’ sub-flow being better than the average of all paths, and so the combined throughput is on average higher than achieved in a single-path approach. The effect of network coding on transfer time was explored in section 4.3.

4.4.1 *RePT*’s Overhead

When discussing network coding in section 4.3 we disregarded the cost of creating the linear combinations, verifying linear independence of coded data, and decoding batches of data. The reasoning behind this was to isolate the advantages of resource pooling, provided by network coding, from the overheads introduced, which are influenced by the processing power and efficiency of the implementation. However, the analysis in figure 4.7 does include these additional costs, and we see a marked increase in transfer time when moving from a batch size of 16 to a batch size of 64. A significant factor of this increase is the construction of the linear combinations of data. In this operation, blocks of 800 bytes are interpreted as integers and multiplied by the coefficients specified in the code vector (section 3.2.1.6). Being 800 bytes long, these integers are extremely large values, and so multiplication and adding together many of them becomes a complex operation. Furthermore, verifying the linear independence of packets at both the forwarders and requester (section 3.2.1.5), and the recoding of packets in the network,

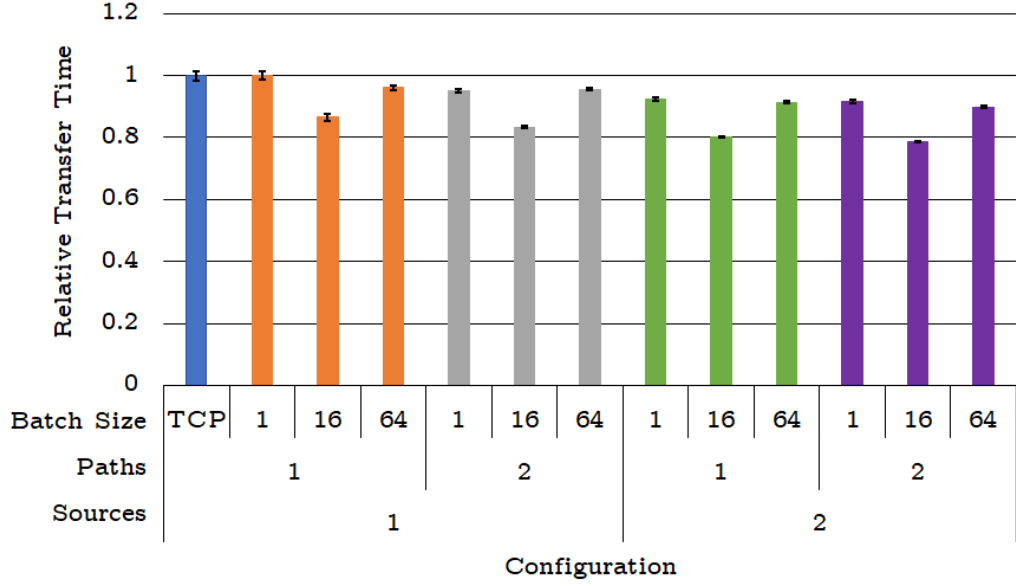


Figure 4.7: Time to transfer a 1MB file, relative to traditional TCP-Vegas, under different numbers of paths, sources, and batch sizes.

introduces an overhead both in time and memory. Intermediate nodes in the modern Internet do not have access to large, fast memory, and so network coding will introduce significant overheads for these nodes. With that being said, moves are being made to introduce additional processing in the network, notably Microsoft’s ‘catapult’ [43]. These factors all contribute to increasing the total time to transfer the file, reducing the throughput of the channel. However, the advantage of network coding comes from reducing the time to decode each batch - not having to wait for timeouts to detect packet loss. Therefore, many situations exist in which network coding will be highly beneficial, notably when timely delivery of data is important.

Other than the overhead associated with network coding, *RePT* introduces additional state at the end points related to each sub-flow. In addition to the normal state to maintain a TCP connection, information on the file size, negotiated parameters (such as batch size), and available sources and paths must be maintained at both requester and responder, and communicated through the network at start-up. Furthermore, each additional flow

introduces the same amount of state as a single TCP connection. This may provide opportunities for denial of service attacks in which a malicious adversary simply initiates a large number of *RePT* connections in order to consume resources. An investigation of the security implications of *RePT* is left to further work.

Chapter 5

Summary and Conclusions

Resource pooling is a powerful idea with broad application in the field of networking. In the context of the Internet, properties such as statistical multiplexing through packet switching, traffic engineering to balance load, and multi-homing sites for improved failure resilience, which may at first sight seem unrelated, all into the category of resource pooling techniques. However, the opportunities for resource pooling in the Internet have by no means been exhausted, while developments in the field of applied mathematics have produced new forms of resource pooling which previously did not exist. In this dissertation, we have drawn together several techniques and explored new directions for resource pooling in the Internet, placing these in the context of a transport level protocol compatible with current Internet architecture. Specifically, we have explored transferring data from multiple sources simultaneously, the use of multiple paths to each source, and pooling data between packets using network coding. We have presented the design of a practical protocol, *RePT*, which incorporates these ideas, and have developed and evaluated a reference implementation.

The main contributions of this dissertation are:

- The introduction and description of multi-sourcing, multi-pathing, and network coding, and how these techniques are relevant to the current

architecture of the Internet.

- The development of *RePT*, a protocol exploiting multi-sourcing, multi-pathing, and networking coding in the modern Internet, while maintaining compatibility with intermediate nodes with no knowledge of *RePT*.
- Simulations demonstrating the situations in which a protocol able to exploit multi-sourcing, multi-pathing, and network coding provide significant benefits.

This dissertation makes a case for the exploration of new resource pooling techniques as a paradigm to improve quality of service on the Internet. Widespread deployment of a protocol like *RePT* would allow network resources to be maximised, improving global quality of service without the need for investment in additional resources.

5.1 Future Work

There are several avenues for future work, developing on the ideas explored in this dissertation.

5.1.1 Kernel Level Implementation

The reference implementation developed here and used to perform evaluation in simulation is implemented in userspace over UDP. This enabled us to test *RePT* without having to modify the kernel’s protocol stack; minimising complexity during development and ensuring correct implementation within the time frame permitted. As such, the reference implementation is effective in its role as a ‘proof of concept’, but is not appropriate for standardisation or widespread deployment.

One of the most important remaining tasks is the implementation of a kernel-space version of *RePT*. A kernel implementation of each component would

allow evaluation of the protocol in realistic scenarios on the Internet, rather than in simulation as has been the case thus far. A service such as PlanetLab [44] could be used to deploy of a an international network of *RePT* sources, and routers able to perform the recoding of packets in the network.

5.1.2 Mechanisms for Choosing Best Paths and Sources

In its current implementation, *RePT* initiates connections in the order they are listed in the *RePT* information packet received with the `SYN + ACK` (section 3.1). This is an appealing method as it allows the administrator of the sources to control which ones the requesting host connects to, and the order in which is connects to these sources. However, this approach does not allow the fine-grained choosing of the best sources based on current network congestions.

When choosing peers to exchange data with, the BitTorrent protocol uses a random approach. The central ‘tracker’ chooses a set of random peers to which a new host connects. If the number of connections drops below a certain threshold, the requesting host requests a list of further nodes to which it can connect [16]. This approach is simple and provides an effective, decentralised distribution of connections in the network. However, it has been noted that additional information, such as the ISPs through which a connection passes, could be used to achieve higher data rates for participants [45].

The challenge for BitTorrent is the distributed nature of the protocol, significantly reducing opportunities for information sharing in the network. This is not the case for *RePT*, in which sources are usually all controlled by a single administrator, and are regularly exchanging information. Furthermore, most sources in BitTorrent are other clients which are likely to be lacking significant resources. This is not the case for *RePT*, where we can expect fewer sources, each with significant resources at their disposal. These differences present many opportunities to explore choosing the best set of sources to connect to at any time.

5.1.3 Comparison of Structured and Random Coding Approaches

Chapter 3 presented two possible methods for generating code vectors recording the linear combinations of packets to use. The first of these involved the responder randomly generating coefficients from a large field, and sending the code vector along with the encoded data. The second approach used a structured coding approach, with the requester indicating the coefficients to use from a matrix, chosen to have the properties necessary to be inverted (section 3.3). This structured coding approach has the potential to remove the need for recoding in the network. A route for further work is to compare these approaches, quantifying in which situations each is most effective. Such a study might also include other coding schemes such as Kim et al.’s block coding [36], and the fountain codes used by Parisi et al. [23].

5.1.4 Opportunities for Coding

There are two types of network coding at the level of flows: *intra-flow* coding and *inter-flow* coding. Intra-flow coding only encodes packets from the same flow, and is the form of coding used within *RePT*. Inter-flow coding allows the coding of packets from different flows, in addition to coding within flows. By definition, inter-flow coding provides higher resource pooling than intra-flow coding, and so is an interesting route for further development.

COPE, described by Katti et al. [46], presents and quantifies the advantage of an approach to inter-flow coding in the context of wireless mesh networks. A key feature of the protocol is the broadcast nature of wireless communication. This is clearly not transferable to the general Internet, where communication is in most cases point-to-point. However, the development of multi-sourcing and multi-pathing begins to provide opportunities for coding between flows. We can imagine a situation might arise in which a set of intermediate nodes lie on the paths of multiple flows, and are able to code together, and later decode, packets from different flows invisibly to the end points.

5.1.5 Additional Use Cases

This dissertation has focused on the use case of accessing popular content on the web. However, a protocol such as *RePT* has application in a number of diverse use cases. As well as being replicated across many locations, content is often also replicated within datacentres. Raicu et al. have previously demonstrated the application of MPTCP within datacentres [47], and so an avenue for further work is to extend this to multi-sourcing and network coding using *RePT*.

A second use case of *RePT* is in tandem with peer-to-peer protocols such as BitTorrent. In its current format, BitTorrent makes no distinction between the throughputs of links to individual peers. Introducing concepts from *RePT* would help mitigate the impact on slow uplinks in the edge of the network.

5.1.6 Security Implications of *RePT*

In section 4.4.1 we made reference to the opportunity for denial of service attacks, provided by *RePT*. This is due to the fact that a *RePT* connection maintains significantly more state than a traditional TCP connection, and so an attacker might be able to overwhelm a *RePT* responder by initiating a large number of *RePT* connections, using multiple paths with spoofed IP addresses. Furthermore, since *RePT* violates both the layers of the OSI Internet model, and the end-to-end principle, higher level protocols are not a practical feasible mechanism for protecting against these attacks. Further work is needed to investigate the impact of widespread adoptions of a protocol such as *RePT*, and the mechanisms available to mitigate malicious adversaries.

Appendix A

Proof of Linear Independence of Rows

Given a $(m \times n)$ matrix C , $m \geq n$, the elements of which are defined by the formula

$$(c_{ij}) = i^{j-1} \mod p$$

we present an inductive proof that no $(n \times n)$ matrix C' formed by choosing n rows of C randomly without replacement has a determinant of 0. This is equivalent to showing that every matrix C' is invertible.

Proof. It can be seen that the matrix C' will have the form:

$$C' = \begin{pmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n+1} & x_{n+1}^2 & \dots & x_{n+1}^{n-1} \end{pmatrix}$$

Theorem: For any $n \in \mathbb{Z}$, the determinant of C' is calculated as:

$$\begin{vmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n+1} & x_{n+1}^2 & \dots & x_{n+1}^{n-1} \end{vmatrix} = \left(\prod_{1 \leq i < j \leq n} x_j - x_i \right)$$

We prove by induction on n .

$$\det(C') = \begin{vmatrix} 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ 1 & x_3 & x_3^2 & \dots & x_3^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n+1} & x_{n+1}^2 & \dots & x_{n+1}^n \end{vmatrix}$$

Subtracting the first row from every other row we obtain the following. The value of the determinant is unchanged.

$$= \begin{vmatrix} 1 & x_1 & x_1^2 & \dots & x_1^n \\ 0 & x_2 - x_1 & x_2^2 - x_1^2 & \dots & x_2^n - x_1^n \\ 0 & x_3 - x_1 & x_3^2 - x_1^2 & \dots & x_3^n - x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & x_{n+1} - x_1 & x_{n+1}^2 - x_1^2 & \dots & x_{n+1}^n - x_1^n \end{vmatrix}$$

Since the elements of the first column are all zero except the first, the determinant of the matrix is equivalent to the determinant of the matrix without

the first row or column.

$$= \begin{vmatrix} x_2 - x_1 & x_2^2 - x_1^2 & \dots & x_2^n - x_1^n \\ x_3 - x_1 & x_3^2 - x_1^2 & \dots & x_3^n - x_1^n \\ \vdots & \vdots & \ddots & \vdots \\ x_{n+1} - x_1 & x_{n+1}^2 - x_1^2 & \dots & x_{n+1}^n - x_1^n \end{vmatrix}$$

We divide each row by its first element, and multiplying the determinant by each of these to ensure it is not changed.

$$= (x_2 - x_1) \cdot (x_3 - x_1) \cdot \dots \cdot (x_{n+1} - x_1) \begin{vmatrix} 1 & x_2 + x_1 & \dots & \sum_{k=1}^n x_2^{n-k} x_1^{k-1} \\ 1 & x_3 + x_1 & \dots & \sum_{k=1}^n x_3^{n-k} x_1^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n+1} + x_1 & \dots & \sum_{k=1}^n x_{n+1}^{n-k} x_1^{k-1} \end{vmatrix}$$

In general, the $(i, j)^{\text{th}}$ element of the above matrix is $\sum_{k=1}^j x_{j+1}^{j-k} x_1^{k-1}$. We multiply this by the matrix with 1's on the main diagonal, $-x_1$'s on the diagonal above, and 0's elsewhere. The determinant of this matrix is 1 and so does not change the result of the equation.

$$= \prod_{k=2}^{n+1} (x_k - x_1) \left| \begin{pmatrix} 1 & x_2 + x_1 & \dots & \sum_{k=1}^n x_2^{n-k} x_1^{k-1} \\ 1 & x_3 + x_1 & \dots & \sum_{k=1}^n x_3^{n-k} x_1^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n+1} + x_1 & \dots & \sum_{k=1}^n x_{n+1}^{n-k} x_1^{k-1} \end{pmatrix} \begin{pmatrix} 1 & -x_1 & 0 & \dots & 0 \\ 0 & 1 & -x_1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix} \right|$$

The $(i, j)^{\text{th}}$ element of the product of the matrices above is:

$$\begin{aligned}
& -x_1 \cdot \sum_{k=1}^{j-1} x_{i+1}^{j-1-k} x_1^{k-1} + 1 \cdot \sum_{k=1}^j x_{i+1}^{j-k} x_1^{k-1} \\
&= -\sum_{k=1}^{j-1} x_{i+1}^{j-1-k} x_1^k + \sum_{k=2}^j x_{i+1}^{j-k} x_1^{k-1} + x_{i+1}^{j-1} \\
&= -\sum_{k=2}^{j-1} x_{i+1}^{j-k} x_1^{k-1} + \sum_{k=2}^j x_{i+1}^{j-k} x_1^{k-1} + x_{i+1}^{j-1} \\
&= x_{i+1}^{j-1}
\end{aligned}$$

Therefore:

$$\begin{aligned}
& \prod_{k=2}^{n+1} (x_k - x_1) \left| \begin{pmatrix} 1 & x_2 + x_1 & \dots & \sum_{k=1}^n x_2^{n-k} x_1^{k-1} \\ 1 & x_3 + x_1 & \dots & \sum_{k=1}^n x_3^{n-k} x_1^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n+1} + x_1 & \dots & \sum_{k=1}^n x_{n+1}^{n-k} x_1^{k-1} \end{pmatrix} \begin{pmatrix} 1 & -x_1 & 0 & \dots & 0 \\ 0 & 1 & -x_1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix} \right| \\
&= \prod_{k=2}^{n+1} (x_k - x_1) \left| \begin{pmatrix} 1 & x_2 & \dots & x_2^{n-1} \\ 1 & x_3 & \dots & x_3^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n+1} & \dots & x_{n+1}^{n-1} \end{pmatrix} \right| \\
&= \prod_{1 \leq i < j \leq n} (x_j - x_i)
\end{aligned}$$

□

The above result implies that the determinant of the matrix is 0 if and only if $x_i = x_j$ for some $i \neq j$. Therefore, by making the x_i distinct, the determinant is guaranteed to be non-zero.

Since the integers modulo p where p is prime form a field we retain the above result when taking remainders on division by p .

Appendix B

Project Proposal

MultiSource MultiPath TCP using Random Linear Codes

A Part III project proposal

N. Davison (*nd369*), Corpus Christi College

Project Supervisor: Prof Jon Crowcroft

B.1 Introduction, Approach and Outcomes

A lot of data in the Internet is replicated in many places. For example the most popular items on streaming services like Netflix, YouTube and iPlayer are many fold replicated, both within each datacentre and across many datacentres. Meanwhile TCP is essentially a single-path protocol: when a TCP connection is established, the connection is bound to the IP addresses of the two communicating hosts. This means current tools can only load balance at the single request level, and current protocols are point-to-point.

It has been shown that resource pooling, by striping data over multiple paths, can provide much simpler and often near optimal load balancing. MultiPath

TCP (MPTCP) came out of this observation, allowing multiple paths to be used simultaneously by a single transport connection. Raiciu et al. demonstrated that MPTCP allows us to approach the optimal throughput in a datacentre [47]. Key et al. point out that bittorrent, which provides sources from multiple servers (a swarm), also helps on the other side of the equation [4].

In this work, I plan to combine multiple pathing, as in MPTCP, with multiple sourcing, as in bittorrent/swarms. With MPTCP we have a single source and a single destination with many paths between them. The protocol I intended to implement can be thought of as MPTCP cut in half - each path originating from a different source - visualised in figure B.1. This multi-source multi-path TCP (MSMPTCP) allows load balancing on the granularity of individual packets, rather than connections. It also allows improvements in terms of robustness and throughput. The work will likely consist of starting from an existing MPTCP implementation [48] and adding functionality for multiple sources.

However MSMPTCP, like TCP, will suffer in performance when there are losses. I plan to use a third technique called network coding to maximise the degrees of freedom that resource pooling can enjoy. Network coding spreads data over both packets and flows - allowing and encourages mixing of data at intermediate nodes. This allows us to cope with (or rather, remove) traffic imbalances and therefore reduce hot spots in the network. Network Coding has been shown to increase throughput and robustness against failures and erasures [49]. Kim et al. have presented Coded TCP using multiple paths (CTCP) and show that when losses are injected, CTCP maintains significantly higher throughput than single-path TCP both without and with coding [36]. Therefore I plan to use MSMPTCP to create more opportunities for coding and approach the optimum efficiency.

The results of this work will be the design and implementation of a protocol for allowing multi-path multi-source TCP with linear coding. I plan to evaluate the protocol in simulation, and if possible in practice, and gather data about the throughput gain of the protocol.

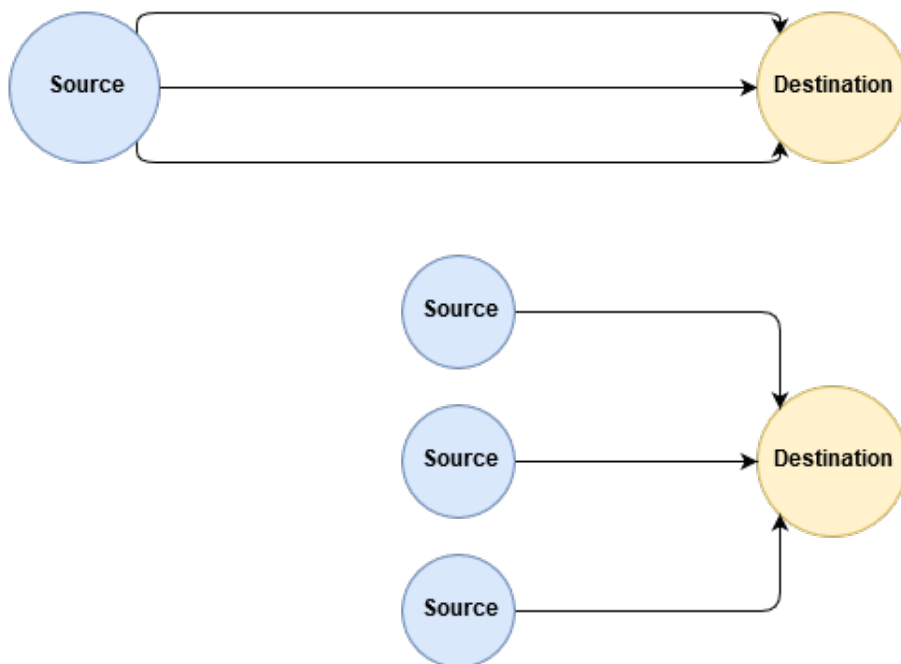


Figure B.1: MPTCP cut in half. The \rightarrow 's here contain switches/routers/proxies/middleboxes which can implement recoding of packets from the same or different flows.

B.2 Workplan

Weeks 1 & 2 - 28/11/2016 to 11/12/2016: Holiday work for Michaelmas modules. This is an essay and annotated bibliography for Network Architectures and a survey paper for Chip Multiprocessors.

Week 3 - 12/12/2016 to 18/12/2016: Reading existing work on MPTCP and Coding

Week 4 - 19/12/2016 to 25/12/2016: Exploring Network Simulation Environments. This week includes Christmas and so is likely to be less productive.

Weeks 5 to 6 - 26/11/2016 to 08/01/2017: Getting existing MPTCP implementation(s) to work

Week 7 - 09/01/2017 to 15/01/2017: Designing Coded MSMPTCP protocol. This week will be largely dominated by revision for a Chip Multiprocessors exam on 16th Jan

Weeks 8 to 12 - 16/01/2017 to 19/02/2017: Implementing a multi-source extension to multipath TCP

Weeks 13 to 17 - 20/02/2017 to 26/03/2017: Implementing coding in multisource multipath TCP

Weeks 18 to 21 - 27/03/2017 to 23/04/2017: Evaluating the gains from using coding with multiple sources. *Progressive review*

Weeks 22 to 26 - 24/04/2017 to 28/05/2017: Writing dissertation

Weeks 27 to 28 - 29/05/2017 to 06/06/2017 Contingencies. *Project Deadline - 03/06/2017. Presentations - 06/06/2017*

Bibliography

- [1] StreamSoft. Pingtools network utilities. https://play.google.com/store/apps/details?id=ua.com.streamsoft.pingtools&hl=en_GB, 2017.
- [2] Damon Wischik, Mark Handley, and Marcelo Bagnulo Braun. The resource pooling principle. *ACM SIGCOMM Computer Communication Review*, 38(5):47–52, 2008.
- [3] Olivier Bonaventure, Mark Handley, Costin Raiciu, et al. An overview of multipath tcp. *USENIX login*, 37(5), 2012.
- [4] Peter Key, Laurent Massoulié, and Don Towsley. Path selection and multipath congestion control. *Communication of the ACM*, 54(1):109–116, January 2011.
- [5] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 1949.
- [6] Sae-Young Chung, G David Forney, Thomas J Richardson, and Rüdiger Urbanke. On the design of low-density parity-check codes within 0.0045 db of the shannon limit. *IEEE Communications letters*, 5(2):58–60, 2001.
- [7] Rudolf Ahlswede, Ning Cai, S-YR Li, and Raymond W Yeung. Network information flow. *IEEE Transactions on information theory*, 46(4):1204–1216, 2000.
- [8] Jay Kumar Sundararajan, Devavrat Shah, Muriel Médard, Szymon Jakubczak, Michael Mitzenmacher, and Joao Barros. Network coding meets tcp: Theory and implementation. *Proceedings of the IEEE*, 99(3):490–512, 2011.

- [9] MinJi Kim, Jason Cloud, Ali ParandehGheibi, Leonardo Urbina, Kerim Fouli, Douglas Leith, and Muriel Médard. Network coded tcp (ctcp). *arXiv preprint arXiv:1212.2291*, 2012.
- [10] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000. Obsoleted by RFC 4960, updated by RFC 3309.
- [11] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Proposed Standard), January 2013.
- [12] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Proposed Standard), March 2011.
- [13] Olivier Bonaventure and Seo SungHoon. Multipath tcp deployments. *IETF Journal*, 12(2), 2016.
- [14] J. Moy. OSPF Version 2. RFC 2328 (INTERNET STANDARD), April 1998. Updated by RFCs 5709, 6549, 6845, 6860, 7474.
- [15] Iljitsch Van Beijnum, Jon Crowcroft, Francisco Valera, and Marcelo Bagunlo. Loop-freeness in multipath bgp through propagating the longest path. In *Communications Workshops, 2009. ICC Workshops 2009. IEEE International Conference on*, pages 1–6. IEEE, 2009.
- [16] Bram Cohen. Bittorrent - a new p2p app. *Yahoo eGroups*, 2001.
- [17] Bram Cohen. The bittorrent protocol specification. http://www.bittorrent.org/beps/bep_0003.html, feb 2017.
- [18] Shuo-Yen Robert Li, Raymond W Yeung, and Ning Cai. Linear network coding. *IEEE transactions on information theory*, 49(2):371–381, 2003.
- [19] Tracey Ho, Ralf Koetter, Muriel Medard, David R Karger, and Michelle Effros. The benefits of coding over routing in a randomized setting. 2003.
- [20] Mea Wang and Baochun Li. How practical is network coding? In *Quality of Service, 2006. IWQoS 2006. 14th IEEE International Workshop on*, pages 274–278. IEEE, 2006.
- [21] Peter Larsson and Niklas Johansson. Multi-user arq. In *Vehicular Technology Conference, 2006. VTC 2006-Spring. IEEE 63rd*, volume 4, pages 2052–2057. IEEE, 2006.

- [22] Sandeep Bhadra and Sanjay Shakkottai. Looking at large networks: Coding vs. queueing. In *INFOCOM*, 2006.
- [23] George Parisis, Toby Moncaster, Anil Madhavapeddy, and Jon Crowcroft. Trevi: Watering down storage hotspots with cool fountain codes. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 22. ACM, 2013.
- [24] Dong Nguyen, Tuan Tran, Thinh Nguyen, and Bella Bose. Wireless broadcast using network coding. *IEEE Transactions on Vehicular technology*, 58(2):914–925, 2009.
- [25] Ning Cai and Raymond W Yeung. Secure network coding. In *Information Theory, 2002. Proceedings. 2002 IEEE International Symposium on*, page 323. IEEE, 2002.
- [26] Tracey Ho, Ben Leong, Ralf Koetter, Muriel Médard, Michelle Effros, and David R Karger. Byzantine modification detection in multicast networks using randomized network coding. In *Information Theory, 2004. ISIT 2004. Proceedings. International Symposium on*, page 144. IEEE, 2004.
- [27] Mohammad Hamed Firooz and Sumit Roy. Data dissemination in wireless networks with network coding. *IEEE Communications Letters*, 17(5):944–947, 2013.
- [28] Anxiao Jiang. Network coding for joint storage and transmission with minimum cost. In *Information Theory, 2006 IEEE International Symposium on*, pages 1359–1363. IEEE, 2006.
- [29] John Bicket, Daniel Aguayo, Sanjit Biswas, and Robert Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *Proceedings of the 11th annual international conference on Mobile computing and networking*, pages 31–42. ACM, 2005.
- [30] Daniel Aguayo, John Bicket, Sanjit Biswas, Glenn Judd, and Robert Morris. Link-level measurements from an 802.11b mesh network. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 121–132. ACM, 2004.
- [31] Szymon Chachulski, Michael Jennings, Sachin Katti, and Dina Katabi. *Trading Structure for Randomness in Wireless Opportunistic Routing*, volume 37. ACM, 2007.

- [32] Sanjit Biswas and Robert Morris. Opportunistic routing in multi-hop wireless networks. *ACM SIGCOMM Computer Communication Review*, 34(1):69–74, 2004.
- [33] Tom Warren. Microsoft to deliver windows 10 updates using peer-to-peer technology. *The Verge*, 2015.
- [34] Christos Gkantsidis and Mitch Goldberg. Avalanche: File swarming with network coding. *Microsoft Research*, 2005.
- [35] Bram Cohen. Comments on avalanche. <http://bramcohen.livejournal.com/20140.html>, 2005. Accessed 16/05/2017.
- [36] MinJi Kim, Ali ParandehGheibi, Leonardo Urbina, and Muriel Meedard. Ctcp: Coded tcp using multiple paths. *arXiv preprint arXiv:1212.1929*, 2012.
- [37] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.
- [38] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. Experimental evaluation of multipath tcp schedulers. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*, pages 27–32. ACM, 2014.
- [39] Th Banachiewicz. Méthode de résolution numérique des équations linéaires, du calcul des déterminants et des inverses, et de réduction des formes quadratiques. *Bulletin International de l'Académie Polonaise des Sciences et des Lettres, Série A*, pages 393–404, 1938. In French.
- [40] Raj Jain, Dah-Ming Chiu, and William R Hawe. *A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System*, volume 38. Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984.
- [41] Costin Raiciu, Damon Wischik, and Mark Handley. Practical congestion control for multipath transport protocols. *University College London, London/United Kingdom, Tech. Rep*, 2009.
- [42] Yu Cao, Mingwei Xu, and Xiaoming Fu. Delay-based congestion control for multipath tcp. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–10. IEEE, 2012.
- [43] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric

- for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.
- [44] Larry L. Peterson and David Cule. Planetlab. <http://www.planet-lab.org/>, 2017.
 - [45] Ruchir Bindal, Pei Cao, William Chan, Jan Medved, George Suwala, Tony Bates, and Amy Zhang. Improving traffic locality in bittorrent via biased neighbor selection. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 66–66. IEEE, 2006.
 - [46] Sachin Katti, Hariharan Rahul, Wenjun Hu, Dina Katabi, Muriel Médard, and Jon Crowcroft. Xors in the air: Practical wireless network coding. In *ACM SIGCOMM computer communication review*, volume 36, pages 243–254. ACM, 2006.
 - [47] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 266–277. ACM, 2011.
 - [48] Multipath tcp resources. <http://www.multipath-tcp.org>. Accessed: 2016-11-05.
 - [49] Ralf Koetter and Muriel Médard. An algebraic approach to network coding. *IEEE/ACM Transactions on Networking (TON)*, 11(5):782–795, 2003.