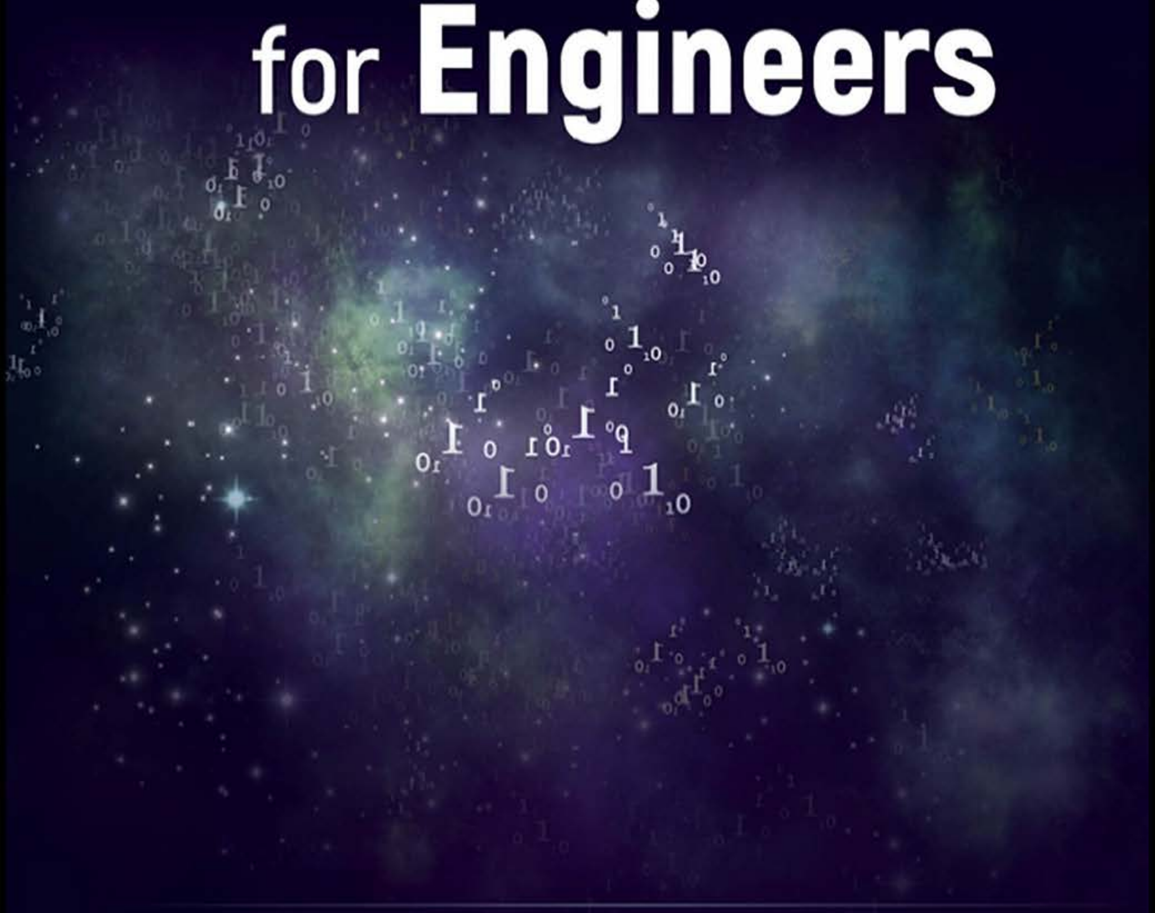
Network Coding for Engineers



Muriel Médard • Vipindev Adat Vasudevan
Morten Videbæk Pedersen • Ken R. Duffy

IEEEPress

WILEY

Network Coding for Engineers

Network Coding for Engineers

Muriel Médard
Massachusetts Institute of Technology
and Optimum
United States

Vipindev Adat Vasudevan
Massachusetts Institute of Technology
United States

Morten Videbæk Pedersen
Steinwurf APS
Denmark

Ken R. Duffy
Northeastern University
USA

IEEE*Press*
WILEY

Library of Congress Cataloging-in-Publication Data Applied for:

Hardback ISBN 9781394217274

Cover Design: Wiley
Cover Image: Courtesy of Vivian Conner

Set in 9.5/12.5pt STIXTwoText by Straive, Chennai, India

In memory of Ralf Kötter and Nuala Bennett Kötter.

Contents

List of Figures

List of Tables

About the Authors

Muriel Médard

Muriel Médard is the NEC Professor of Software Science and Engineering in the Electrical Engineering and Computer Science (EECS) Department at the Massachusetts Institute of Technology (MIT), where she leads the Network Coding and Reliable Communications Group in the Research Laboratory for Electronics, and Chief Scientist for Steinwurf, which she has co-founded. She obtained three Bachelor's degrees (EECS in 1989, Mathematics in 1989, and Humanities in 1991), as well as her M.S. (1991) and Sc.D. (1995), all from MIT. Muriel is a Member of the German National Academy of Sciences Leopoldina (elected in 2022), the American Academy of Arts and Sciences (elected in 2021), and the US National Academy of Engineering (elected in 2020); a Fellow of the US National Academy of Inventors (elected in 2018); and a Fellow of the Institute of Electrical and Electronics Engineers (elected in 2008). She holds Honorary Doctorates from the Technical University of Munich (2020), from the University of Aalborg (2022), and from the Budapest University of Technology and Economics (BME; 2023).

Muriel was co-winner of the MIT 2004 Harold E. Edgerton Faculty Achievement Award and was named a Gilbreth Lecturer by the US National Academy of Engineering in 2007. She received the 2017 IEEE Communications Society Edwin Howard Armstrong Achievement Award and the 2016 IEEE Vehicular Technology James Evans Avant Garde Award. Muriel was awarded the 2022 IEEE Kobayashi Computers and Communications Award. She received the 2019 Best Paper Award for *IEEE Transactions on Network Science and Engineering*, the 2018 ACM SIGCOMM Test of Time Paper Award, the 2009 IEEE Communications Society and Information Theory Society Joint Paper Award, the 2009 William R. Bennett Prize in the Field of Communications Networking, the 2002 IEEE Leon K. Kirchmayer Prize Paper Award, as well as nine conference paper awards. Most of her prize papers are co-authored with students from her group.

Muriel has served as technical program committee co-chair of ISIT (twice), CoNext, WiOpt, and WCNC and of many workshops. She has chaired the IEEE Medals committee and served as a member and chair of many committees, including as inaugural chair of the Millie Dresselhaus Medal. She was Editor-in-Chief of the *IEEE Journal on Selected Areas in Communications* and of the *IEEE Transactions on Information Theory*. She has served as an editor or a guest editor of many IEEE publications, including the *IEEE Transactions on Information Theory*, the *IEEE Journal of Lightwave Technology*, and the *IEEE Transactions on Information Forensics and Security*. Muriel was on the inaugural steering committees for the *IEEE Transactions on Network Science* and for the *IEEE Journal on Selected Areas in Information Theory*. She was elected president of the *IEEE Information Theory Society* and served on its board of governors for a dozen years.

Muriel received the inaugural MIT Excellence in Postdoctoral Mentoring Award (2022) and in 2013 the inaugural MIT EECS Graduate Student Association Mentor Award, voted by the students. She was recognized nationally as a Siemens Outstanding Mentor (2004) for her work with high school students. She set up the Women in the Information Theory Society (WithITS) and Information Theory Society Mentoring Program, for which she was recognized with the 2017 IEEE Aaron Wyner Distinguished Service Award. She served as an undergraduate Faculty in Residence for 7 years in two MIT dormitories (2002–2007). Muriel was elected by the faculty and served as a member and later chair of the MIT Faculty Committee on Student Life and as the inaugural chair of the MIT Faculty Committee on Campus Planning. She was chair of the Institute Committee on Student Life. She has served on the Board of Trustees of the International School of Boston, for which she was treasurer. She served on the Nokia Bell Labs Technical Advisory Board.

Muriel has over 80 US and international patents awarded, the vast majority of which have been licensed or acquired. For technology transfer, she has co-founded CodeOn, Steinwurf, and Optimum.

Muriel has supervised over 40 master's students, over 20 doctoral students, and over 25 postdoctoral fellows.

Vipindev Adat Vasudevan

Vipindev Adat Vasudevan is a postdoctoral associate at the Massachusetts Institute of Technology (MIT) with the Network Coding and Reliable Communications Group. He received his B.Tech. degree in Electronics and Communication Engineering from the Mahatma Gandhi University, Kerala, India, in 2014; his M.Tech. degree in Computer Engineering (Cyber Security) from the National Institute of Technology, Kurukshetra, India, in 2017; and his Ph.D. in Information and Communication Technologies from the University of Vigo, Spain, in 2022. He has previously worked as a Marie Curie Fellow in the Wireless Telecommunications Laboratory of the University of Patras, Greece, and as a research officer in the Commission for Higher Education Reforms constituted by Kerala State Higher Education Council, India. His research interests include but are not limited to network coding, network security, 5G and beyond networks, and the Internet of Things. He is an active member of the IEEE.

Morten Videbæk Pedersen

Morten Videbæk Pedersen is the Chief Technology Officer (CTO) at Steinwurf ApS, a company he co-founded in 2011 as a spin-off from Aalborg University. Under his leadership, Steinwurf has developed high-performance software solutions for communication and storage systems, bringing theoretical academic research into practical, commercial applications. As CTO, Morten has overseen software architecture, tooling, quality assurance, product development, technical sales, and team management.

Morten holds a Ph.D. in Wireless Communication from Aalborg University, where he also completed his M.Sc. with honors. He has experience as a postdoctoral researcher at the Department of Electronic Systems, Aalborg University, focusing on low-complexity network coding algorithms and cooperative networking protocols. His academic journey included time as a visiting student at the Massachusetts Institute of Technology (MIT).

An avid software enthusiast, Morten has contributed to several scientific software libraries, including Kodo, a high-performance network coding library used by industry and research institutions worldwide. He has received numerous accolades, including the Nokia Developer Champion award in 2010.

Morten is passionate about software engineering, staying updated with the latest developments, and enjoys building innovative solutions that push the boundaries of technology.

Ken R. Duffy

Ken R. Duffy is a professor at Northeastern University with a joint appointment in the Department of Electrical and Computer Engineering and the Department of Mathematics. His primary research focus is on the interdisciplinary application of probability and statistics in science and engineering. Algorithms he has developed have been implemented in digital circuits and in DNA. He received his B.A. (mod) in Mathematics in 1996 and his Ph.D. in Applied Probability in 2000, both awarded by Trinity College Dublin. He was previously a professor at the National University of Ireland, Maynooth, where he was the Director of the Hamilton Institute, an interdisciplinary research center, from 2016 to 2022. He was one of three co-directors of the Science Foundation Ireland Centre for Research Training in Foundations of Data Science, which has funded more than 120 Ph.D. students.

He is a co-founder of the Royal Statistical Society's Applied Probability Section (2011), co-authored a cover article of Trends in Cell Biology (2012), is a winner of the best paper award at the IEEE International Conference on Communications (2015), the best paper award from *IEEE Transactions on Network Science and Engineering* (2019), the best research demo award from COMSNETS (2022), and the best demo award from COMSNETS (2023). He is an associate editor of *IEEE Transactions on Information Theory* and *IEEE Transactions on Molecular, Biological, and Multi-Scale Communications*.

Preface

This book is the result of several years of research, teaching, and product design. The goal to have a book that can be a teaching tool at different levels and a real guide for the practitioner has had a slow genesis. Teaching tutorials, as well as networking, coding, algorithms and information theory classes at MIT and at Northeastern University, has shaped the understanding of the best way to present the material to balance didactic utility with depth. Working with companies implementing network coding has guided the selection of what concepts are necessary and what framing of architectural issues is truly useful to enable network coding to improve network performance. The result is a book where the main philosophical thread is that the material is curated in a way that we have ascertained to be beneficial for engineers, whether they be students or practising professionals. Indeed, one of the core joys of engineering is being involved in constant learning. We hope that this book fosters such learning.

January 6th, 2025 *M.M., V.A.V., M.V.P., and K.R.D.*
Massachusetts, USA & Aalborg, Denmark

Acknowledgments

This book has been in the works for many years, ever since the publication of the algebraic network coding work by Ralf Kötter and Muriel Médard. The insight and contributions of Ralf permeate this book and many of the key algebraic insights. The many tutorials he and Muriel undertook over the years before his untimely passing cemented the foundations on which this book is built.

We thank the many collaborators, colleagues, and students who have participated, directly or indirectly, in the journey of network coding or in the creation of the body of knowledge and understanding in this book, including Vitaly Abdrashitov, Szymon Acedanski, Varun Aggarwal, Ebad Ahmed, Chang Wook Ahn, Tor Aulin, Hari Balakrishnan, João Barros, Olivier Bonaventure, Viveck Cadambe, Ning Cai†, Flavio Calmon, Christopher Chang, Clifford Choute, Jason Cloud, Alejandro Cohen, Supratim Deb, Bikash Dey, Michelle Effros, Elona Erez, Attila Eryilmaz, Soheil Feizi, Frank Fitzek, Kerim Fouli, Christina Fragouli, Mario Gerla†, Steluta Gheorghiu, Andrea Goldsmith, Bernhard Haeupler, Keesook Han, Babak Hassibi, Janus Heide, Tracey Ho, Wenjun Hu, Siddarth Jaggi, Szymon Jakubczak, Coleen Josephson, Ton Kalker, David Karger, Mohammad Karzand, Dina Katabi, Sachin Katti, Minji Kim, Wonsik Kim, Michael Langberg, Hyunjoo Lee, Doug Leith, Ben Leong, Luisa Lima, Daniel Lucani, Desmond Lun, Nancy Lynch, Martin Maier, Ali Makhdoumi, Ivana Maric, Athina Markopoulou, Michael Mitzenmacher, Marie-José Montpetit, Peter Musial, Rafael d'Oliveira, Una-May O'Reilly, Asuman Ozdaglar, Payam Pakzad, Ali ParandehGheibi, Joon-Sang Park, Giovanni Pau, Ilias Politis, Narayan Prakash, Nayak Ratnakar, Harihan Rahul, Arman Rezaee, Parastoo Sadeghi, Devavrat Shah, Jun Shi, Shirley Shi, Saurabh Shintre, Chres Sørensen, Fabio Soldo, Emina Soljanin, Alex Sprintson, Milica Stojanovic, Jay-Kumar Sundararajan, Surat Teerapittayanon, Emre Telatar, Alberto Lopez Toledo, Danail Traskov, Christos Tselios, Tiago Vinhoza, Ming Xiao, Edmund Yeh, Yunjung Yi, Linda Zeger, Weifei Zeng, and Fang Zhao.

January 6th, 2025

M.M., V.A.V., M.V.P., and K.R.D.
Massachusetts, USA & Aalborg, Denmark

Acronyms

5G	Fifth Generation of Wireless Networks
AC-RLNC	Adaptive and Causal Random Linear Network Coding
AES	Advanced Encryption Standard
AIMD	Additive Increase and Multiplicative Decrease
API	Application Programming Interface
ARQ	Automatic Repeat-reQuest
BDP	Bandwidth-Delay Product
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CTCP	Coded TCP
D2D	Device-to-Device
DoF	Degree of Freedom
EW	Effective Window
FB-FEC	Feedback-Based Forward Erasure Correction
FEC	Forward Erasure Correction
FIFO	First In – First Out
FlEC	Flexible Erasure Correction
FPGA	Field Programmable Gate Array
Gbps	Gigabits per Second
GCD	Greatest Common Divisor
GF	Galois Field
GPU	Graphics Processing Unit
HTTP	Hyper Text Transfer Protocol
HUNCC	Hybrid Universal Network Coding Cryptosystem
IETF	Internet Engineering Task Force
i.l.c.	Integer Linear Combination
IP	Internet Protocol
kB	kilo Bytes
MB	Mega Bytes

Mbps	Megabits per Second
MIT	Massachusetts Institute of Technology
MTU	Maximum Transmission Unit
NC	Network Coding
P2P	Point-to-Point
PMF	Probability Mass Function
PQUIC	Pluginized QUIC
QUIC	Quick UDP Internet Connections
r.v.	random variable(s)
RLC	Random Linear Code
RLNC	Random Linear Network Coding
RREF	Reduced-Row Echelon Form
RTT	Round Trip Time
SIMD	Single Instruction, Multiple Data
SR-ARQ	Selective Repeat ARQ
SRTT	Effective Round Trip Time
SWNC	Sliding Window Network Coding
TCP	Transmission Control Protocol
TCP/NC	Network Coding with TCP
TD	Triple-duplicate
TO	Time-out
UDP	User Datagram Protocol

1

Introduction

The objective of packet networks, such as the Internet, is to reliably transport information from sources to receivers. While data is packaged for communication, which may involve compressing it and adding headers describing destinations as well as other information, traditionally the data streams themselves are treated as immutable as they traverse the network, in the sense that the information of individual streams is kept separate and intact throughout their transit. The simple premise of Network Coding (NC) is that data can be readily algebraically manipulated and that making use of that fact results in a relaxation of many networking problems, which can be leveraged to vastly improve performance in a number of distinct dimensions.

As a basic thought experiment, if the network wishes to transport two pieces of information, A and B, it could instead transport A and $A+B$ and allow the receiver to solve a basic set of linear equations, i.e. $B=(A+B)-A$, to recover both. Why would that be useful? Think of a setting where three packets are transmitted across a network, but one will get lost in transit, as can happen in networks such as the Internet. If the network transmits A, B, and $A+B$, no matter which two packets get through, the receiver can reconstruct the original data. The generalization and exploitation of this simple principle results in large, practically achievable gains in performance, as we shall see.

NC remained in the realm of pure information theory until the advent of Random Linear Network Coding (RLNC) [1]. It has evolved from simple modulo additions of two packets to creating linear combinations of multiple packets in a finite field and communicating the digital result as doing so enables significant improvement in the bandwidth efficiency of networks. RLNC inherently generates robustness and adaptability in dynamic environments [2]. RLNC has proven suitable for distributed, dynamic environments such as wireless networks. Future networks, such as small cell environments featuring Device-to-Device (D2D) communication and cooperation between devices, will have to ensure that every user in the network is fairly provided with services, where distinct metrics concerning the quality of service, such as throughput, delay, and latency, all have to be met. In cooperative environments, RLNC can achieve the upper bound of efficiency in multicasting. RLNC has been established to be a practical solution in these settings that can provide higher throughput and reliability with lower latency over unreliable network infrastructure. A key feature of RLNC is that it can be readily integrated into legacy systems, thanks to its compatibility with existing protocols and its ability to be implemented both in software as well as hardware. While RLNC seeks to achieve optimum efficiency in terms of bandwidth usage by sending coded packets over different channels, it also naturally provides erasure correction and imparts resistance to man-in-the-middle attacks. While the use of NC by itself provides only weak protections security [3], it has been extensively studied how, with some augmentation, it is possible to harness more benefits such as increased security and privacy protection using RLNC in the challenging, quantum computing times ahead.

In a world with increasing demands on multiple fronts, it becomes a necessity to re-envision communication protocols and RLNC has already been proven to be a way ahead. Different industrial adopters are already benefiting from coding principles and use RLNC in a variety of applications. It has found its way to standards and industrial deployments [4, 5] in line with the significant theoretical research that has been happening around the topic in the last couple of decades. There have been great resources, not only the large number of papers published in reputed conferences and journals but also great books introducing different dimensions of RLNC in the literature. However, a textbook that assists the adaptation of NC from its excellent results in the literature to engineering solutions has not materialized. This book tries to bridge the gap between the academic advancements in the area of NC to its practical realizations, from a complete engineering perspective.

1.1 Vision and Outline

This book walks the reader through the nuances of practical NC and its implementation in real-world applications with curated, directly relevant mathematical explanations. Excellent resources for theoretical explanation of the concepts are listed, but mainly as additional readings, as the book itself is self-contained in the material it presents. It concentrates on how these concepts can be used to formulate engineering solutions in the complex communication scenarios that are expected as part of current and future networks. This book begins with the basics of NC and explores more advanced concepts step-by-step with implementation details and possible use cases. Further, it provides a look ahead on how NC can be used in varied scenarios such as post-quantum cryptography and heterogeneous wireless networks. This textbook is written in a way that it can be a stand-alone read for engineers or go along with a lab-based semester or quarter course on NC.

It expects minimal prior knowledge of communication and information theory and presents the concepts from the introductory level.

As a structured and gradual learning experience, the book lays out the concepts of NC from basics to its more complex adaptations in a simple but rigorous manner. This textbook is designed to suit the requirements of the course material for a lab-based semester-long course or a one-quarter course, while it also allows communication engineers to use it for self-study and explore RLNC-based implementations. For a one-quarter (9-week) course, Chapters 1, 2, 4, and 5 will form the basis of a course with sufficient substance and detail to serve as a stand-alone introduction. For such a course, it would be recommended to skip some of the optional material (marked with **) in Chapters 2 and 4. Portions marked with ** indicate that the material in them provides interesting theoretical background, but they can be omitted. The sections with ** will be of interest to students who are curious about some of the mathematical underpinnings of NC, but they are not required for implementing RLNC algorithms or for developing an understanding of how they can benefit the operation.

For a full one-term (12-week) course, a more software engineering slanted curriculum will incorporate the quarter course material discussed above with all of Chapter 2, and, additionally, all of Chapter 3, most of Chapter 5, and all of Chapter 6. A one-term course oriented more towards network optimization and architecture will benefit from covering all of Chapter 2, as well as all of Chapter 4 and additionally Chapters 5 and 6. A one-term course oriented towards a more theoretical audience will benefit from covering all of Chapter 2, as well as all of Chapters 4, 7, and 8.

The clear mapping between the book and class organization allows instructors or engineers following a self-paced program, a clear and ready way to use the book. While the primary focus is on the application of NC, different network models such as multipath and mesh networks as well as their design principles are covered.

The book can be used in modular way to meet the needs of different engineering goals. Engineers who intend to understand how to construct a detailed, low-level coding language version of NC modules may benefit from reading portions marked with a *. These provide extensive algorithmic development of implementing finite field operations in a context that is useful to NC. Those sections can be skipped by readers whose goal is to understand NC and incorporate it into systems, but who intend to use commercially available packages, say KODO library, to manage the mechanics of NC.

The organization of the book is as follows. This first chapter focuses on introducing NC as the use equations that result from combining data rather than the original data in networks. The data allows flexibility which is not only convenient, it also is the root of the robustness, design flexibility, and theoretical optimality of NC in many settings. The chapter also introduces the reader to a Python-based package that will readily manage erasures. This package will provide readers with a tool that will suffice to verify the majority of the engineering uses of NC.

Chapter 2 introduces the reader to finite representations. The approach is rigorous, but entirely practical in its philosophy. The finite field results focus on explaining the arithmetic that is critical to NC implementations. This discussion will make sure the engineers get a clear picture of NC implementations and should be able to choose different designs according to their application scenario.

Chapter 3 provides the reader with a detailed guide to implementing NC. Together, Chapters 2 and 3 provide a detailed description of the finite field representations of the data required for the inner mechanics of NC. The chapter also includes detailed pseudo-codes and delves into computational efficiency aspects.

Chapter 4 considers the use of NC for managing losses in networks. We shall see that, whether there is a single receiver or several, the use of NC allows us to move away from considering uncoded packets of data, and to consider instead the transmission of equations, or degrees of freedom. Matters such as acknowledging received information, and adapting the transmission of data according to such acknowledgement, are covered in detail. A primer on queuing in this chapter suffices to exhibit the main performance advantages of NC in networks with losses, and the attendant benefits in terms of delay and throughput.

Chapter 5 allows the reader to compare practical NC to other state-of-the-art communication protocols. Different NC protocol constructions are introduced. Building on Chapter 4, we show how protocols can be designed to realize the principles of NC. We provide means of analyzing the benefits of NC in protocols and schema for reasoning about and for modeling the operation of NC-based protocols.

In Chapter 6, we delve in detail in to how to incorporate NC into current transport protocols, and realize the principles and gains discussed in Chapters 4 and 5. We consider two of the main transport protocols, Transmission Control Protocol (TCP) and QUIC, and provide a full guide on how to integrate NC into these widely deployed protocols.

Chapter 7 moves towards treating more complex networks. First, the chapter extends the concept of representing data as equations to the network setting to show that successive linear transformations of data in different nodes of a network can be viewed as a single linear transformation in an end-to-end fashion, regardless of the network's topology. Leveraging concepts from Chapters 2 and 4, this chapter sets a simple mathematical framework for NC in arbitrary topologies.

Chapter 8 focuses on the interplay between security and NC. This chapter considers how data integrity, secrecy, and reliability can be ensured with NC. Topics such quantum-safe encryption, which is based on coding data together, detection and defeating pollution attacks, are covered after a rapid primer on the essentials of information-theoretic secrecy.

Some of the material in Chapters 2, 4, 5, 7, and 8 overlaps with MIT's 6.120A, 6.7410/1, 6.263, 6.441 and 6.989 as taught by Médard and with Northeastern University's EECE 7332 as taught by Duffy.

We should note that NC has significant associated intellectual property. While it is not our role to detail it in this book, we provide a representative list in the appendix, as it may be of interest to the engineer who seeks to incorporate NC into commercial systems.

1.2 Coding

In information theory, coding is a technique used to improve efficiency and decrease error rates in data communication over noisy channels in order to reach channel capacity. There are two main types of coding traditionally: source coding and channel coding. Source coding involves optimizing the representation of data in outgoing symbols to enable exact recovery of the original message (lossless source coding) or recovery with some distortion (lossy source coding). Channel coding involves adding redundancy to the outgoing symbols so that the receiver can decode the original message over a noisy channel. Both of these coding processes are typically applied to source-generated messages before they are sent over the noisy channel in practical systems. However, traditional coding schemes only involve the source and sink nodes. Network coding, as shown in Fig. 1.1, emerged as an approach to improve both reliability and efficiency together, but from a network perspective.

In multi-hop communication systems, there are often intermediate nodes, also known as routers, between the source and sink. These intermediate nodes

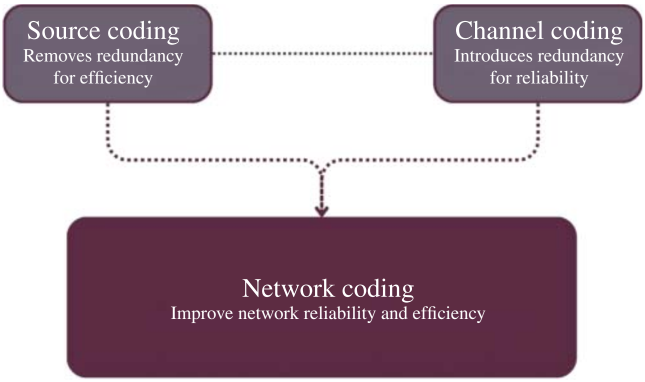

Source coding
Removes redundancy for efficiency
Channel coding
Introduces redundancy for reliability
Network coding
Improve network reliability and efficiency

Figure 1.1 The introduction of network coding has created a new area within coding theory that focuses on enhancing the reliability and efficiency of the network.

typically only forward the information they receive to the next node, a method known as the store-and-forward approach. However, the concept of **network coding**, introduced by Ahlswede et al. [6], allows intermediate nodes to also code the inputs they receive before sending them out, rather than just replicating the inputs to the outgoing links. This expands the possibilities for coding and can significantly improve the design of switching systems and network architecture. Throughout this book, unless specified, the term “coding” refers to NC and can happen at any node, not just the source. “Decoding” refers to the process of re-extracting the original data from network-coded packets. Additionally, the concept of “recoding” will be discussed as intermediate nodes in the network can also code their inputs to create new combinations to send on their outgoing links.

Why Coding?
There are multiple reasons for using coding in a network. In particular, coding can be used to compensate for lost portions of data.

- Packets can be lost in transit due to congestion.
- Links can have outages, dropping packets.
- Networks can be lossless but have bottlenecks.
- Data can be severely corrupted by noise and interference to the extent that it should be erased.

If sources were not allowed to algebraically manipulate data, i.e. if data were **immutable**, the solutions to resolve these issues are resource allocation, acknowledgments (ACKs) and negative acknowledgments (NACKs), retransmission protocols, and so forth. The essential idea of **coding** is that data is not immutable, but can readily be manipulated algebraically and that doing so effectively results in a relaxation of constraints, from which it is easy to see that gains can follow.

1.3 Data as Mutable – A Relaxation

In order to develop intuition and motivate the developments that follow, we will allow ourselves a number of acceptable simplifying assumptions, e.g. that all packets have the same number of bits, and temporarily suspend our belief in one significant way:

- We will imagine that we can readily represent real numbers to perfect precision in the digital domain.

Doing this will motivate us to work through the applied algebra that informs the algorithms and their implementation that follows.

1.3.1 Immutable Data

For notation, let $X_{(i)}$, $i = 1, 2, \dots$ denote a sequence of packets where each packet $X_{(i)} \in \{0, 1\}^n$ is a binary string of length n bits. Equivalently, we can think of $X_{(i)}$ as representing an integer in $\{0, 1, \dots, 2^n - 1\}$ or even as defining a polynomial of degree $n - 1$ whose coefficients are the data's binary entries.

A sender wishes to communicate to a receiver $(X_{(1)}, \dots, X_{(K)})$ over a network that experiences erasures. The receiver has to obtain all K integers in order to be able to reconstruct the original binary data. This sensibility is illustrated in Fig. 1.2. If data were immutable and drops can happen, the sender would have to resort to either:

1. doing repetition coding, where we send $(X_{(1)}, \dots, X_{(K)})$ more than once so that if a particular $X_{(i)}$ gets erased by the network, the receiver has other opportunities to receive it;
2. or, alternatively, if we allow feedback from the receiver to the sender, perhaps they could let the sender know which ones they have received or which ones they are missing to inform retransmissions.

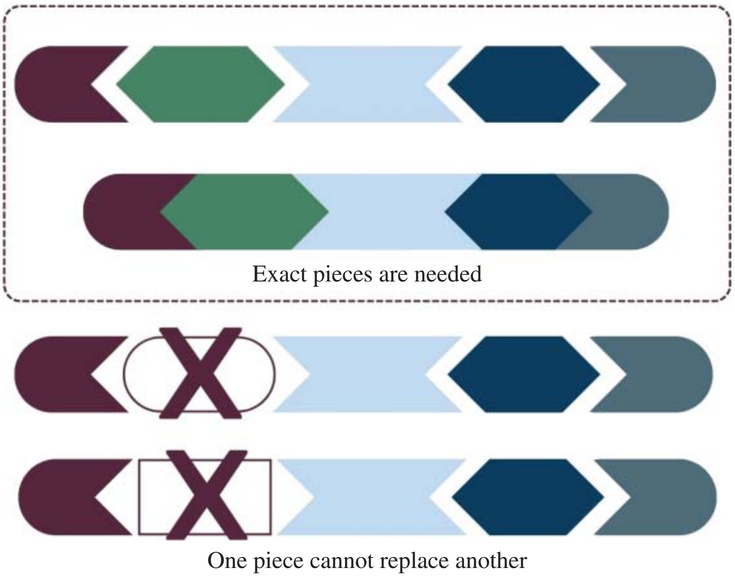


Figure 1.2 Traditional view of data as immutable. This figure is inspired by joint work with Michelle Effros.

1.3.2 Mutable Data

Coding offers an alternative way forward, as illustrated in Fig. 1.3, where we functionally combine the values in packets before sending them. For reasons that will soon become apparent, we will always take linear combinations.

The following difficulties must be tackled to realize NC.

- Manage the finite nature of the representation of digital data.
- Establish efficient means to do encoding, the creation of linear combinations, and decoding, for which it will turn out that Gauss–Jordan elimination is the answer.
- Consider how the coding coefficients, the multipliers in the linear equations, should be created.
- Exercise our imagination in where this essential **relaxation** of network problems can be exploited.

With the first two elements, the good news, to be covered in Chapter 2, is that we can create different types of finite field constructs (prime, extension fields, and polynomials over fields) for manipulating representations of binary data and compute in any of them. In all of these addition, subtraction, and multiplication will be easy. The challenge will be to manage division. The **magic** will be that once we figure out what we need for divisors, which we can do by brute force if needs be, all of our real-valued algorithms, including Gauss–Jordan elimination, will carry over unchanged. Readers interested in detailed implementation will learn those

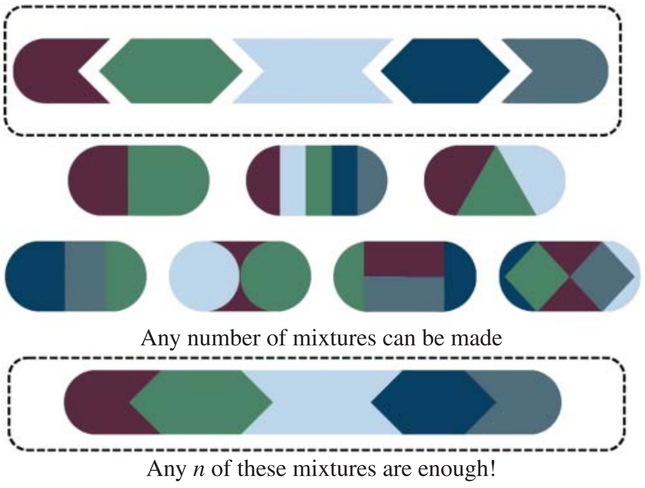

Any number of mixtures can be made
Any n of these mixtures are enough!

Figure 1.3 Coding view of data as being manipulable. This figure is inspired by joint work with Michelle Effros.

skills in Chapter 3. The last element will be the topic of the chapters from 4 to 8. Before going to deeper discussions on these concepts, let us look into a few of such applications from a high-level perspective.

1.4 Network Coding – Data as Equations

The initial discussions that can be associated with the concepts of NC can be traced back to work of Celebiler and Stette in 1978 [7], but it got a major boost with the seminal paper of Ahlswede et al. in 2000 [6]. From then on, the advantages of NC have been broadly investigated. The basic idea is to allow intermediate nodes to perform linear operations on the packets they receive in their incoming edges and send these encoded packets to their outgoing edges instead of simply forwarding the received packets. While it would also be possible to use nonlinear operations to combine packets, most practical systems use linear coding operations as they suffice to realize gains in all but the most artificial of setups and lend themselves to a straightforward procedure for decoding. The use of NC reduces the number of packets to be sent over a channel and provides a mechanism to achieve the maximum efficiency promised by the max-flow min-cut theorem [8].

There are different formal definitions of NC. One of the most common ones, which can be adopted to the context of this book, is coding at a node in a packet network. A packet network is a type of network in which data is transmitted into relatively small units called packets. Each packet contains no more than a set amount of data (e.g. in Ethernet, a packet can contain no more than 1500 bytes of data) and the necessary routing information to enable it to reach its destination. NC allows nodes to combine or "code" the data contained in multiple packets together before forwarding them, in contrast to traditional store-and-forward networks. Most of the modern communication systems can be considered as a packet network and we may use this notion of packet networks and the definition of NC in packet networks commonly in this book. A more extensive discussion on formal definitions can be found here [9].

1.5 Use Cases and Examples

NC is useful for ensuring reliability and/or increasing throughput in a communication network. If a receiver knows coding coefficients and can perform Gauss–Jordan elimination, an obvious benefit of using NC is that a network node is no longer required to gather all data packets one-by-one, instead it only has to receive enough linearly independent encoded packets. Here, we showcase some of the benefits that result from this observation in different application scenarios.

We assume that raw data can be expressed in binary even if we interpret it in different ways.

1.5.1 Addressing Bottlenecks

Consider the butterfly topology depicted in Fig. 1.4. Assume that time is slotted, each directed link can forward one packet per unit time, the source node is marked as e, and the two receivers are R_1 and R_2. The source wishes to efficiently use the network to communicate packets to the receivers. If data were immutable, the middle link would have to be time-shared, and so the total maximum throughput rate of the network would be 3 packets per unit time, one packet down each edge and one packet that is sent down one edge, copied, and then relayed across the bridge in the middle to the other receiver.

Treating data as mutable, however, we can consider two packets, A and B consisting of binary strings, at the source. What could the network do if on the middle link the XOR of the bits representing A and B were sent at each time-step?

- The receiver R_1 would get A as well as the XOR of A and B so that, by XOR, they could recover both A and B.
- The receiver R_2 would get B as well as the XOR of A and B so that, again by XOR, they could recover both A and B.

Thus, four data units can reach the destinations (two each) within the same time taken for three unit transmissions in the previous case and the network has a maximum throughput of 4 rather than 3. If we considered A and B as integers and replaced XOR with addition, the network would also have a throughput of 4 as long as we assume that there's no problem that $A + B$ might be too big an integer

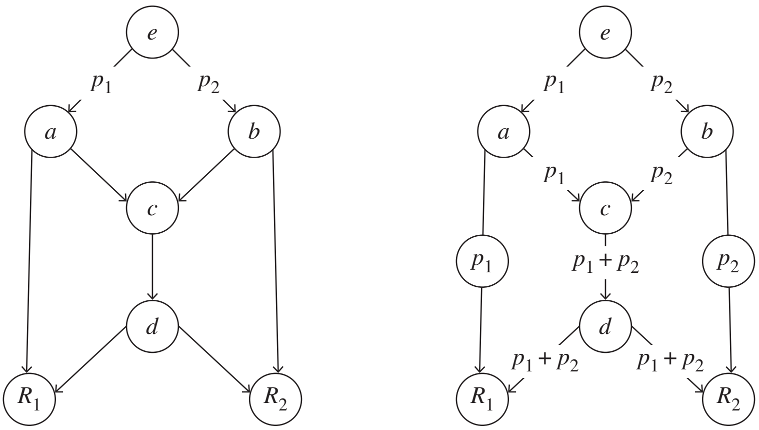


Figure 1.4 Butterfly example. This figure is inspired from the work of Ahlswede et al. [6].

to be represented by a packet of size n bits and so could "wraparound." We have temporarily suspended our concern about the latter.

The more general version of what the receivers are doing in this example is **Gaussian elimination** where, from the equations A and $A + B$ (or B and $A + B$), both A and B are extracted. Essentially all erasure correcting codes rely on Gaussian elimination to resolve linear combinations of data. It lies at the heart of the operation of **Random Linear Network Codes**, which will be at the core of our discussions in this book.

1.5.2 Addressing Packet Drops in a Point-to-Point Communication

Packet drops happen for a wide variety of reasons and are one of the primary transport congestion control protocols, such as the Transmission Control Protocol/Internet Protocol (TCP/IP). The basic operation of TCP/IP is for a sender with a lot of data to discover the capacity of a network by keeping a number of unacknowledged packets in flight whose size develops according to Additive Increase and Multiplicative Decrease (AIMD), as shown in Fig. 1.5. With each packet successfully acknowledged, the window of unacknowledged packets in flight expands linearly. With each packet that has gone unacknowledged, the window experiences a multiplicative decrease. With a large amount of data to communicate, packet drops are inevitable.

To manage erasures, whether they arise from packet drops, link outages, or data that is too corrupt to recover, we will consider collections of data as packets of information. Imagine we make an $N \geq K$ by K matrix A of real values, each randomly selected from any continuous distribution so that the probability two entries are equal is zero,

$$A = \begin{pmatrix} \alpha_{(1,1)} & \alpha_{(1,2)} & \cdots & \alpha_{(1,K-1)} & \alpha_{(1,K)} \\ \alpha_{(2,1)} & \alpha_{(2,2)} & \cdots & \alpha_{(2,K-1)} & \alpha_{(2,K)} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ \alpha_{(N,1)} & \alpha_{(N,2)} & \cdots & \alpha_{(N,K-1)} & \alpha_{(N,K)} \end{pmatrix}.$$

Instead of sending $(X_{(1)}, \dots, X_{(K)})$, what if the sender transmitted N linear combinations of packets imagined as integers, which we call coded-packets, in order:

$$\begin{pmatrix} \alpha_{(1,1)} & \alpha_{(1,2)} & \cdots & \alpha_{(1,K-1)} & \alpha_{(1,K)} \\ \alpha_{(2,1)} & \alpha_{(2,2)} & \cdots & \alpha_{(2,K-1)} & \alpha_{(2,K)} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ \alpha_{(N,1)} & \alpha_{(N,2)} & \cdots & \alpha_{(N,K-1)} & \alpha_{(N,K)} \end{pmatrix} \begin{pmatrix} X_{(1)} \\ \vdots \\ X_{(K)} \end{pmatrix} = \begin{pmatrix} X_{(1)}{}' \\ \vdots \\ X_{(N)}{}' \end{pmatrix}.$$

We have a problem, of course: the $X_{(j)}$ are capable of being represented in binary with n bits whereas $X_{(i)}{}' = \sum_{j=1}^{K} \alpha_{(i,j)} X_{(j)}$ is real valued as the α are, but let us suspend our disbelief for now. To recover the K pieces of information, it suffices to receive *any* K coded packets and then use **Gaussian elimination**.

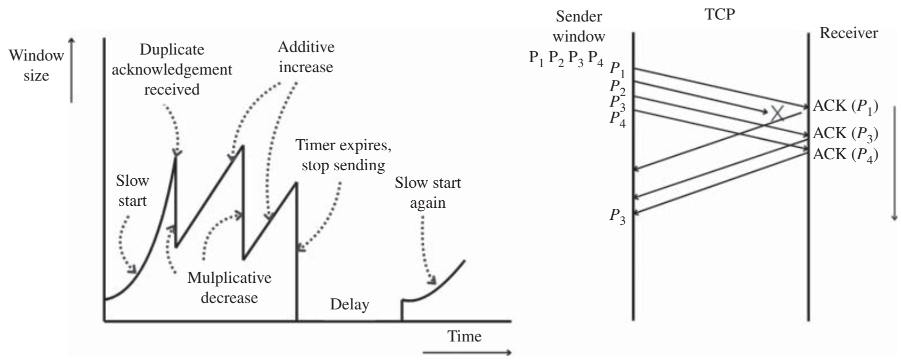


Figure 1.5 TCP/IP. Left image is inspired from one by Brad Karp.

In the traditional view, all $(X_{(1)}, \dots, X_{(K)})$ must be received and if, say, one single packet at random gets lost, it must identified and retransmitted. In the new view, what does the receiver need in order to be able to extract the information? What if one packet is lost at random and $N = K + 1$ so that the receiver gets:

$$\begin{pmatrix} \alpha_{(1,1)} & \alpha_{(1,2)} & \cdots & \alpha_{(1,K-1)} & \alpha_{(1,K)} \\ \alpha_{(2,1)} & \alpha_{(2,2)} & \cdots & \alpha_{(2,K-1)} & \alpha_{(2,K)} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ \backslash & \backslash & \backslash & \backslash & \backslash \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ \alpha_{(N,1)} & \alpha_{(N,2)} & \cdots & \alpha_{(N,K-1)} & \alpha_{(N,K)} \end{pmatrix} \begin{pmatrix} X_{(1)} \\ \vdots \\ X_{(K)} \end{pmatrix},$$

where the \ indicates erased, can the receiver recover the $(X_{(1)}, \dots, X_{(K)})$?

So long as the receiver knows the matrix of coding coefficients and gets **any** K coded packets of the $K + 1$ transmissions, they are capable of recovering the K pieces of original data by **Gaussian elimination** as they have K equations in K unknowns.

Exercise 1.1 Gauss–Jordan elimination in the reals

Write your own code (rather than using any built-in function within an environment such as MATLAB, Python, and R) that takes any (invertible) K times K matrix with *real-valued* entries, A, and performs Gauss–Jordan elimination to determine A^{-1} such that $AA^{-1} = A^{-1}A$ is the $K \times K$ identity matrix. To test the correctness of the code, for $K = 8$ create matrices with random, independent entries, $a_{i,j}$, from some continuous distribution (e.g. Gaussian or uniform in an interval)

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,K} \\ \vdots & \vdots & \vdots \\ a_{K,1} & \cdots & a_{K,K} \end{pmatrix},$$

and random vectors, $x = (x_1, \dots, x_k)$, of length K of integer values in $\{0, 1, \dots, q-1\}$ for some integer q, evaluate

$$y^T = Ax^T.$$

Using your Gaussian elimination solver, determine an estimate of x, $\hat{x}^T = A^{-1}y^T$, from A and y and confirm whether it is correct to within a reasonable tolerance due to machine precision when dealing with reals (i.e. $|x - \hat{x}| <$ tol, where $|\cdot|$ can be any distance in $\mathbb{R}^K$ that you prefer) for 10^5 random matrices A and inputs x. What is your empirical estimate of the probability of $\hat{x} \neq x$?

Another approach is to repair packet losses retroactively which typically requires some feedback from the receiver about the lost packets. Of course, the sender could simply retransmit the original packets without any coding. NC helps in reducing the necessary feedback from the receiver, because it does not have to communicate which packets were lost, just how many. The sender can simply generate and send as many coded packets as the number of packets lost on the receiver.

Let's put some values into this RLNC approach. Suppose we have a set of numbers, $\{1, 2, 3, 4, 5\}$, that we want to transmit over a communication network (our $K = 5$). We can use RLNC to encode these numbers into linear combinations of packets, which can then be transmitted over the network. A random encoding matrix can be used to create linear combinations of these five numbers. Let's consider a 10×5 matrix of positive integers,

$$A = \begin{pmatrix} 1 & 4 & 3 & 2 & 9 \\ 2 & 7 & 1 & 6 & 5 \\ 8 & 6 & 2 & 3 & 1 \\ 1 & 4 & 7 & 9 & 2 \\ 4 & 5 & 1 & 2 & 8 \\ 7 & 9 & 3 & 5 & 1 \\ 2 & 4 & 6 & 8 & 5 \\ 7 & 2 & 8 & 1 & 3 \\ 1 & 7 & 3 & 5 & 2 \\ 4 & 5 & 7 & 3 & 9 \end{pmatrix}.$$

This matrix can be used to create up to 10 coded combinations of the original messages ($N = 10$). Consider the original message matrix as

$$X = (1, 2, 3, 4, 5).$$

Now, $Y^T = A \times X^T$ gives

$$Y = (74, 70, 45, 74, 59, 76, 94, 53, 49, 114).$$

Considering the above example, any of the 5 messages out of 10 possible ones, along with their coefficients, is enough to decode the source packets by Gaussian elimination and so recover the original data.

1.5.3 Multiple Receivers – Reliable Broadcast

What is the impact of this type of coding in a broadcast setting? Imagine you have a transmitter that wishes to send one set of K pieces of data to R distinct receivers. The receivers have independent channels that have probability ϵ of packet erasure. The likelihood that a given packet is erased for at least one receiver, and so needs to be retransmitted, is

$$1 - (1 - \epsilon)^R \approx \epsilon R,$$

if ϵ is small, by Taylor expansion. The overall channel looks R times worse than the individual ones. A back-of-the-envelope calculation says we need to send $N = K/(1 - \epsilon R)$ packets to receive K on average.

With coded packets, however, each receiver only needs to get *any* K packets, and so the overall channel still looks like it has erasure rate ϵ and our back of the envelope says we need only $N = K/(1 - \epsilon)$.

For example, let's consider a case with three receivers trying to get the message X (same X, A, and Y as in the previous example). After five transmissions, the first one receives (74, 70, 74, 59), the second one receives (74, 45, 74, 59), and the third one receives (70, 45, 74, 59). That is, each node misses one packet, but the lost packets are in different time-slots. In the uncoded setting, the sender would have to retransmit three different packets to compensate for each lost packet. With coded packets, however, as soon as they all receive 76 at the sixth slot correctly, each receiving node can complete the decoding successfully. This reduced the number of retransmissions required from three to one.

Sometimes reliable data distribution is not an application requirement. For example, when video streaming to a large number of receivers on a wireless network, where feedback would be impractical. In this case, NC can help to maximize the value of each retransmission to the receivers. Ideally, a number of redundancy packets should be transmitted for to provide a sufficient degree of reliability for the majority of receivers. This redundancy ratio (also known as "overshoot") can be adjusted according to the estimated packet erasure probability to ensure it is appropriately dimensioned.

1.5.4 Recoding and Multi-hop Networks

Because coding generates equations, the way in which equations are created is not important, it is only the final equations that are relevant. In particular, we can take packets that have already been coded and recode them, taking note of new coefficients.

Imagine you wish to communicate data through a two-hop relay where the first one independently drops packets with erasure probability ϵ_{12} and the second with probability ϵ_{23}:

$$X_{(i)} \overbrace{\longrightarrow}^{\epsilon_{12}} X_{(i)} \text{ or erasure } \overbrace{\longrightarrow}^{\epsilon_{23}} X_{(i)} \text{ or erasure,}$$

which, from the receiver's point of view, is equivalent to a single hop with error probability

$$X_{(i)} \quad \overbrace{\longrightarrow}^{1-(1-\epsilon_{12})(1-\epsilon_{23})} \quad X_{(i)} \text{ or erasure.}$$

As a result, the fraction of packets that are not erased, the throughput, is $(1 - \epsilon_{12})(1 - \epsilon_{23})$.

With coded packets, however, the rate is higher. If two stages of erasure coding with decoding and re-encoding at the second node, then information can be communicated between nodes 1 and 2 at a rate of $1 - \epsilon_{12}$ packets per unit time and between nodes 2 and 3 at a rate of $1 - \epsilon_{23}$ packets per unit time. Thus, information can be communicated between nodes 1 and 3 at a rate of $\min\left(1 - \epsilon_{12}, 1 - \epsilon_{23}\right)$, which is in necessarily greater than we could achieve without coding, $(1 - \epsilon_{12})(1 - \epsilon_{23})$.

In a multi-hop network, where nodes have limited knowledge of other devices, especially those that are several hops away, the sender will have to wait for an acknowledgment or send arbitrary packets in advance to compensate for potential losses in the hops. Using coding operations can greatly increase the number of transmissions with new information rather than simply repeating an already sent packet. Coded packets have a high likelihood of containing new information, and if recoding is also enabled, new recoded packets can be generated before all the original packets from the generation are received. RLNC is beneficial in minimizing communication between devices by providing implicit coordination through random combinations.

1.5.5 Distributed Storage – Incast

If we had K pieces of data that we want to safely store for recovery on R different devices, none of which can individually store all K pieces and each of which could be subject to failure, what's the best strategy for distributing our data?

With data being mutable and coded packets, clearly we should just store independent linear combinations everywhere. Therefore, to recover the data, we need only collect K linear equations of the data of interest from any set of servers and use Gaussian elimination to retrieve the original data.

Note that as a linear combination of linear combinations is just a linear combination itself, we just need to know the new coefficients. This will prove to be a feature that RLNC exploits as it allows the recoding of coded data without having to decode it first.

1.5.6 Security

The goal of cryptography is to map data through a highly nonlinear function such that, in the absence of side-knowledge (e.g. a key), the output is unrelated to the input. Encrypting and decrypting data is usually a computationally costly (hence energy intensive and slow) process, which explains the ongoing interest in physical layer security, for example, which we will return to later.

If you have K variables and fewer than K linear equations, can you solve the equations? If not, how much uncertainty do you have in attempts to recover the original data? One approach to ensuring that one has fewer than K linear equations is to encrypt only a subset of the data, for example one of the equations. Unless that encrypted equation can be decrypted, all the data is secure.

The examples above are simple indicators of what we will see in Chapters 4 to 8. However, there is an additional important step before we delve into them in detail: A toolbox that enables you to explore and exploit the mutability of data. We don't intend to provide a full-blown experimental setup but provide enough resources to empirically understand the core concepts of NC.

1.6 A Toolbox for Implementing Network Coding

When exploring the topic of erasure-correcting algorithms, implementation is the method by which we can bring abstract concepts to life. While a solid theoretical foundation is undoubtedly central to comprehending the intricacies of these algorithms, it is equally important to acknowledge the value of hands-on experience in their actual implementation.

In this section, we will introduce you to the tools that will accompany you throughout the book. These tools are carefully chosen to facilitate your learning journey and provide a practical understanding of how the algorithms presented work in real-world scenarios.

Throughout the book, the tools we will use are:

- **Python**: Python offers ease of use, cross-platform support, and abundant resources for documentation and libraries, making it a user-friendly and versatile programming language for algorithmic implementations. If you have no prior experience with Python, we would recommend the following resources for getting started [10, 11].
- **PyErasure**: PyErasure is a Python library that provides flexible erasure coding algorithms.

In the following subsections 1.6.1 and 1.6.3, we will introduce the abovementioned tools in more detail, providing you with the essential information needed to get started regardless of your current starting point.

1.6.1 Tool 1: The Python Programming Language

The purpose of this section is not to teach you the details of the Python programming language but rather to provide the resources needed to get started if you have no prior Python experience. If you are already an experienced Python programmer, you can skip this section and proceed to the section 1.6.3. However, if you are new to Python, we have curated a list of valuable resources to help you kickstart your Python journey.

- **Python Documentation**: The official Python documentation is a comprehensive resource that covers the language syntax, standard libraries, and best practices. It serves as a reliable reference for understanding Python's features and capabilities. You can access the documentation online at https://python.org/doc.
- **Online Tutorials**: Numerous online tutorials offer step-by-step guidance on learning Python from scratch. Websites like Codecademy, W3Schools, and Real Python provide interactive tutorials that introduce the language's fundamentals and guide you through practical examples. These tutorials often include exercises and quizzes to reinforce your understanding.
- **Python Learning Paths**: Platforms like Coursera, Udemy, and edX offer structured learning paths for Python programming. These courses range from beginner-friendly introductions to more advanced topics, providing a structured curriculum and hands-on assignments to enhance your skills.
- **Python Books**: There are several highly regarded books that cater to Python beginners, such as *Python Crash Course* by Eric Matthes [12] and *Automate the Boring Stuff with Python* by Al Sweigart [13]. These books offer a hands-on approach, walking you through Python concepts with practical examples and exercises.
- **Online Communities and Forums**: Engaging with the Python community can be immensely helpful when starting out. Platforms like Stack Overflow, Reddit's *r/learnpython*, and the official Python Discord server are great places to ask questions, seek guidance, and learn from experienced Python programmers.

1.6.2 Getting Python (Step-by-Step)*

To download Python, you can visit the official Python website at www.python.org. Here are the steps to download Python:

1. Open your web browser and go to https://www.python.org/downloads/.
2. On the downloads page, you will find the latest version of Python for your operating system. The website automatically detects your operating system, but you can also choose a different version or release candidate if desired.
3. Select the appropriate installer for your operating system. Python is available for Windows, macOS, and various Linux distributions. Note that both macOS and Linux typically come with Python preinstalled. Choose the installer that matches your system specifications (32-bit or 64-bit).

4. Once you have selected the installer, click on the download link to initiate the download.
5. After the download is complete, locate the downloaded installer file and run it.
6. Follow the installation wizard instructions to install Python on your computer. During the installation process, you may have the option to customize the installation, such as choosing the installation location or adding Python to the system PATH.
7. Once the installation is complete, you can verify the installation by opening a command prompt or terminal and typing "python" or "python3" (depending on your system configuration). If Python is installed correctly, you will see the Python interpreter prompt.

Congratulations! You have successfully downloaded and installed Python on your computer. You can now start writing and executing Python code.

Remember, mastering a programming language takes practice and hands-on experience. As you progress through the book, you will gain a deeper understanding of Python's application in algorithm implementation. So, even if you are new to Python, don't be discouraged. With the resources provided, you have everything you need to embark on this exciting journey of algorithmic exploration.

1.6.3 Tool 2: PyErasure – Erasure Correcting Algorithms in Python

In this book, we aim to equip you with the necessary background knowledge to build your own erasure correcting algorithms. While we encourage you to embark on that journey, we have chosen to utilize an existing erasure coding library, PyErasure, for the majority of the examples and lab checkouts. This approach offers readers, who are primarily interested in algorithm utilization, a well-defined and functional starting point.

PyErasure has been specifically designed to prioritize simplicity and ease of understanding, extension, and modification over raw speed. It provides support for fundamental erasure coding algorithms, and with minimal effort, it can be extended to incorporate more sophisticated algorithms.

By leveraging PyErasure, you will be able to focus on exploring the utilization and application of erasure coding algorithms without getting bogged down in intricate implementation details. This approach allows for a smoother learning experience, enabling you to grasp the core concepts and principles before diving deeper into algorithmic customization and enhancement.

Throughout the book, we will explore various use cases and scenarios where PyErasure can be effectively employed. You will have the opportunity to analyze, experiment with, and adapt existing erasure coding algorithms to suit specific requirements. This practical approach will enhance your understanding of how erasure coding works in real-world scenarios, while also providing a solid foundation for future algorithmic implementations.

PyErasure is freely available. However, its use does require a registration/license. To get a license, please visit: https://www.steinwurf.com/research-license-request. Once you receive email confirmation that your license has been processed, we can proceed to installation. We may use the Python Package Index (pypi.org) using the following command to install the library:

```
python3 -m pip install git+ssh://git@github.com/
steinwurf/pyerasure
```

With PyErasure installed, we can implement our first encoding/decoding example. In the following, we will utilize PyErasure to implement a straightforward encoding and decoding setup. While this setup omits certain details, such as emulating packet loss and latency, our focus will solely be on the essential Application Programming Interface (API) required for generating encoded symbols at the encoder and consuming them at the decoder.

Listing 1.1 pyerasure01.py

```python
# introduction/pyerasure01.py.

import pyerasure
import pyerasure.generator
import os

symbols = 64
field = pyerasure.finite_field.Binary8
symbol_bytes = 1024

encoder = pyerasure.Encoder(field=field, symbols=symbols, symbol_bytes=
        symbol_bytes)
decoder = pyerasure.Decoder(field=field, symbols=symbols, symbol_bytes=
        symbol_bytes)

# Coefficient generator
generator = pyerasure.generator.RandomUniform(field=field, symbols=symbols)

# Generate random data
data_in = bytearray(os.urandom(encoder.block_bytes))

# Set the data on the encoder
encoder.set_symbols(data_in)

# While the decoder is not complete (i.e. we don’t have all the data)
# Generate a new encoded symbols and pass it to the decoder.
while not decoder.is_complete():
    # Generate the encoded symbol
    coefficients = generator.generate()
    symbol = encoder.encode_symbol(bytearray(coefficients))

    # Decode the symbol
    decoder.decode_symbol(symbol, bytearray(coefficients))

print("Decoding complete!")
```

This code demonstrates the usage of the PyErasure library to perform erasure coding operations. Here's a breakdown of what the code does:

1. The code imports the necessary modules: `pyerasure`, `pyerasure.generator`, and `os`.
2. It defines the parameters for the erasure coding operation: `symbols` (number of symbols), `field` (finite field type, in this case, Binary8), and `symbol_bytes` (number of bytes per symbol).
3. It creates an `Encoder` object and a `Decoder` object using the specified parameters.
4. It creates a `RandomUniform` generator object to generate coefficients for encoding.
5. Random data is generated using `os.urandom` and stored in the `data_in` variable.
6. The `set_symbols` method of the encoder is used to set the data for encoding.
7. The code enters a loop that continues until the decoder has reconstructed all the data.
8. Inside the loop, a new encoded symbol is generated using the generator and the `encode_symbol` method of the encoder.
9. The generated symbol and coefficients are passed to the `decode_symbol` method of the decoder for decoding.
10. The loop continues until the `is_complete` method of the decoder returns `True`, indicating that the decoding process is complete.
11. Finally, a message is printed indicating that the decoding is complete.

In summary, this code demonstrates the encoding and decoding process using the PyErasure library. It generates random data, encodes the data using erasure coding, and then decodes the encoded symbols until all the original data is recovered.

1.7 Summary

The first chapter of the book introduces the fundamental concepts of NC, emphasizing its simplicity and potential for improving network performance. It explains the shift from traditional packet forwarding to coding packets at network nodes, highlighting the benefits of sending combinations of packets. The chapter provides a historical overview of NC, defines key terms, and explores the advantages of treating data as mutable. Additionally, practical use cases of NC are discussed to illustrate its real-world applications. The chapter also outlines the vision and structure of the book to make it easier for both teaching courses and for individuals interested in self-study. By the end of this chapter, we hope you are able to:

(A) Understand the idea of "data as mutable."
(B) Know some use cases for NC.
(C) Familiarize yourself with some useful tools for implementation.

Additional Reading Materials

Previous textbooks on NC [9, 14, 15] provide extensive coverage of the mathematical concepts of NC and some of the use cases of NC. However, a tutorial to practical implementations of NC in the new era of heterogeneous and more open networks is the gap that restricts it from being largely applied in the real world. We are trying to cover this in the following chapters.

References

1 T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong, "A random linear network coding approach to multicast," *IEEE Transactions on Information Theory*, vol. 52, no. 10, pp. 4413–4430, 2006.

2 C. Fragouli, J.-Y. Le Boudec, and J. Widmer, "Network coding: an instant primer," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 1, pp. 63–68, 2006.

3 K. Bhattad and K. R. Narayanan, "Weakly secure network coding," *First Workshop on Network Coding, Theory, and Applications*, pp. 8–20, 2005.

4 Steinwurf ApS, "Barracuda Networks optimizes SD-WAN traffic with patented erasure correction technology from Steinwurf," 2020. [Online]. Available: https://www.steinwurf.com/blog/barracuda-networks.

5 Steinwurf smart networks. [Online]. Available: https://www.steinwurf.com/.

6 R. Ahlswede, N. Cai, S.-Y. Li, and R. W. Yeung, “Network information flow,” *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, 2000.

7 M. Celebiler and G. Stette, “On increasing the down-link capacity of a regenerative satellite repeater in point-to-point communications,” *Proceedings of the IEEE*, vol. 66, no. 1, pp. 98–100, 1978.

8 S.-Y. Li, R. W. Yeung, and N. Cai, “Linear network coding,” *IEEE Transactions on Information Theory*, vol. 49, no. 2, pp. 371–381, 2003.

9 T. Ho and D. Lun, *Network Coding: An Introduction*. Cambridge University Press, 2008.

10 Python Software Foundation, “Python.org - Beginner’s guide to python,” 2023. [Online]. Available: https://www.python.org/about/gettingstarted/.

11 Codecademy, “Codecademy - learn python 3,” 2023. [Online]. Available: https://www.codecademy.com/learn/learn-python-3.

12 E. Matthes, *Python Crash Course: A Hands-on, project-based Introduction to Programming*. No Starch Press, 2023.

13 A. Sweigart, *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press, 2019.

14 R. W. Yeung, *Information Theory and Network Coding*. Springer Science & Business Media, 2008.

15 M. Médard and A. Sprintson, *Network Coding: Fundamentals and Applications*. Academic Press, 2011.

2

Finite Field Arithmetic for Network Coding

After the first chapter, if we could perform all our data operations in the reals, we could go directly to implementations and algorithms. As we're restricted in our representation to a fixed number of bits, we have to consider arithmetic in the setting of finite fields, which are also known as Galois fields. Finite fields are used heavily in cryptography as well as coding, so the introduction here would be informative for the study of that topic too.

A field $\mathbb{F}$ is a set of at least two elements where two operations called addition (denoted $+$) and multiplication (denoted $\cdot$) that take elements of $\mathbb{F} \times \mathbb{F}$ and return elements of $\mathbb{F}$ are defined such that the following are satisfied.

- **Closure**: If $a, b \in \mathbb{F}$, then $a + b \in \mathbb{F}$ and $a \cdot b \in \mathbb{F}$.
- **Associativity**: $a + (b + c) = (a + b) + c$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in \mathbb{F}$.
- **Commutativity**: $a + b = b + a$ and $a \cdot b = b \cdot a$ for all $a, b \in \mathbb{F}$.
- **Additive and multiplicative identity**: There exist two distinct elements 0 and 1 in $\mathbb{F}$ such that $a + 0 = a$ and $a \cdot 1 = a$.
- **Additive inverses**: For every $a \in \mathbb{F}$, there exists an element in $\mathbb{F}$, denoted $-a$, called the additive inverse of a, such that $a + (-a) = 0$.
- **Multiplicative inverses**: For every $a \neq 0 \in \mathbb{F}$, there exists an element in $\mathbb{F}$, denoted by a^{-1} or $1/a$, called the multiplicative inverse of a, such that $a \cdot a^{-1} = 1$.
- **Distributivity of multiplication over addition**: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

It can be readily verified that the elements of familiar number systems, e.g. the reals $\mathbb{R}$, the rationals $\mathbb{Q}$, and the complex numbers $\mathbb{C}$, form fields. If the set $\mathbb{F}$ contains a finite number of elements, the field is said to be **finite**. You are already familiar with at least one finite field, $\mathbb{F}_2$.

Example 2.1 $\mathbb{F}_2 = \{0, 1\}$ *is a finite field where addition and multiplication are XOR and logical AND; see Table* 2.1.

The finite field $\mathbb{F}_2$ can, alternatively, be thought of as the integers with their usual addition and times operators, but modulo 2.

Table 2.1 The finite field $\mathbb{F}_2$ consists of elements 0 and 1, which satisfy the addition (+) and multiplication (·) tables.

$+$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

Example 2.2 $\mathbb{F}_{12} = \{0, 1, \dots, 11\}$ *is* ***not*** *a finite field when equipped with the usual addition and multiplication modulo* 12. *For example, the entry* 2 *has no multiplicative inverse (i.e. there does not exist* $a \in \mathbb{F}_{12}$ *such that* $2 \cdot a \bmod 12 = 1$). $\mathbb{F}_{12}$ *also has other issues. For example*, $3 \cdot 4 \bmod 12 = 0$, *so if we did have a multiplicative inverse for* 3 *or for* 4, *we would have* $4 = (3^{-1} \cdot 3) \cdot 4 \bmod 12 = 3^{-1} \cdot (3 \cdot 4) \bmod 12 = 0$. *We have broken mathematics.*

Example 2.3 $\mathbb{F}_5 = \{0, 1, \dots, 4\}$ ***is*** *a finite field when equipped with the usual addition and multiplication modulo* 5. *The multiplicative inverse of* 1 *is* 1, *for* 2 *it is* 3 *as* $2 \cdot 3 \bmod 5 = 6 \bmod 5 = 1$, *for* 3 *it is* 2, *and for* 4 *it is* 4 *as* $16 \bmod 5 = 1$. *So, we have addition, subtraction, multiplication, and multiplicative inverses for all elements apart from* 0.

It can be seen from the examples that a finite field handles all four common algebraic operations – addition, subtraction, multiplication, and division – with a modulo operation. Addition and multiplication are similar to the real numbers while subtraction and division are performed with their additive and multiplicative inverses. Division is the most involved operation and we will cover it in more detail later in this chapter. However, it is already clear that we can have fields with a finite number of elements in which we can treat these operations in a similar fashion to how they are performed with real numbers. The number of elements of a field is called the field's **order**. We will show that we may construct a finite field $\mathbb{F}_q$ of order q if q is a prime power, $q = p^m$, where p is a prime number and $m \geq 1$ is a positive integer.

- If $m = 1$, we call it a **prime field**, integer representations suffice, and field operations can be done with modulo p integer arithmetic.
- If $m > 1$, it's called an **extension field** and we will use polynomial representations with field operations done modulo a primitive polynomial, which serves the same function as a prime.

These two types of finite fields underlie data applications. As 2 is prime, we can create extension fields in common computer storage sizes. As prime or extension fields may be more efficient depending on hardware and software considerations, one may use either. To understand finite fields, it's best to start with prime fields.

For prime value q, one can create the finite field $\mathbb{F}_q = \{0, \dots, q-1\}$ using the usual integer addition and multiplication, but modulo q. If $q = 11$, we have $\mathbb{F}_q = \{0, 1, 2, \dots, 10\}$ and, for example,

- **Addition**: $5 + 9 \bmod 11 = 3$, since $14 \bmod 11 = 3$.
- **Subtraction**: $4 - 10 \bmod 11 = 5$, since $-6 \bmod 11 = 5$.
- **Multiplication**: $6 \cdot 7 \bmod 11 = 9$, since $42 \bmod 11 = 9$.
- **Inversion**: $4^{-1} \bmod 11 = 3$, since $4 \cdot 3 \bmod 11 = 1$.
- **Division**: $5/4 \bmod 11 = 4$, since $5 \cdot 3 \bmod 11 = 4$.

Essentially, addition, subtraction, and multiplication are "easy," while division is the challenging operation. Division by a number x is the multiplication by the multiplicative inverse x^{-1} of that number, i.e. dividing by x is multiplying by x^{-1}. Establishing that multiplicative inverses exist is key to confirming we have a finite field. Having mechanisms to determine the value of a multiplicative inverse is key to efficient computation.

2.1 (Not So) Brute Force Determination of Inverses

For a given multiplication rule and number of elements, note that checking if inverses exist is actually straightforward. Take each element $a \in \{1, 2, \dots, q-1\}$ and multiply it by each $b \in \{1, 2, \dots, q-1\}$ until you find b such that $a \cdot b = 1$ and record b as a^{-1} in a lookup table. When one has exhausted the a, check that every element of $\{1, 2, \dots, q-1\}$ is found only once in the lookup table. If so, we have our inverses.

If we do have a finite field, an upper bound on the complexity of building this lookup table would occur if the identification of the inverse was always the last element in the list. If that were to happen, then, as identification of an inverse removes at least one thing from our list of length $q-1$, at most we'd do

$$\sum_{i=1}^{q-1}(q-i) = \sum_{i=1}^{q-1} i = \frac{(q-1)q}{2}$$

multiplications, which is of order q^2. For the sort of q that will be useful for Network Coding (NC), building a lookup table with this method is readily doable and a once-off computational expense. In some computational environments, however, lookup tables with their associated memory storage can be inefficient and so the development of efficient on-the-fly algorithms is important. Key to these will be understanding the GCDs and remainders.

Exercise 2.1

For each prime value $p \in (16, 255)$, by brute force build a lookup table by empirically identifying a^{-1} in $\bmod p$ arithmetic (i.e. such that $a \cdot a^{-1} \bmod p = 1$) and show that each inverse is unique (i.e. there are no repeated entries in the lookup table).

2.2 Division in the Integers and Greatest Common Divisors

Let us first explore the nature of division for integers, $\mathbb{Z}$, further as that will form the core of our understanding of how we algebraically manipulate binary data and identify multiplicative inverses.

Definition 2.1 "a divides b," written $a \mid b$, if and only if there exists an integer k such that $ak = b$.

Some common divisibility facts:

- If $a \mid b$ and $b \mid c$ then $a \mid c$, as if $ax = b$ and $by = c$ then $a(xy) = c$.
- If $a \mid b$ then $a \mid bc$, as if $ax = b$ then $a(xc) = bc$.
- If $a \mid b$ and $a \mid c$ then $a \mid b + c$.
- If $a \mid b$ and $a \mid c$ then $a \mid sb + tc$ for all $s, t \in \mathbb{Z}$.

For integers $s, t, sb + tc$ is known as an **integer linear combination (i.l.c)** of b and c, where the integers can be negative.

Definition 2.2 The Greatest Common Divisor (GCD) of the integers a and b, given at least one is not 0, is denoted $\gcd(a, b)$ and is the largest positive integer that divides both a and b. For $a \neq 0$, we define $\gcd(a, 0) = \gcd(0, a) = a$.

As examples $\gcd(4, 10) = 2$, $\gcd(3, 7) = 1$, and $\gcd(5, 25) = 5$. Let us start with a simple fact about gcd that will help us understand an algorithm that computes it.

Lemma 2.1 For any pair of integers a and b, $\gcd(a, b) = \gcd(a, b - a)$.

Proof: Let $g = \gcd(a, b)$. Then $g \mid a$ and $g \mid b$, and g is the largest integer for which this is true. As $g \mid a$ and $g \mid (b - a)$, g must be less than or equal to $\gcd(a, b - a)$, hence $\gcd(a, b) \leq \gcd(a, b - a)$.

Let $h = \gcd(a, b - a)$. Then $h \mid a$ and $h \mid b - a$. Thus $h \mid a$ and $h \mid ((b - a) + a)$, hence $h \mid bh$ is less than or equal to the largest number that divides both a and b, namely $\gcd(a, b)$. Hence $\gcd(a, b - a) \leq \gcd(a, b)$.

Applying Lemma 2.1 to the previously considered examples: $\gcd(4, 10) = \gcd(4, 6) = 2$, $\gcd(3, 7) = \gcd(3, 4) = 1$, and $\gcd(5, 20) = 5$.

We shall use the following theorem, without providing a proof, as we need it for the definition of remainder. Division with a remainder is called **Euclidean Division** and you may have come across it in school. Remainders are closely related to modulo arithmetic and they also play a key role when binary data is considered through the lens of their polynomial representation.

Theorem 2.1 ***Division Theorem*** For all pairs of integers a, b with $b > 0$, there exists a unique pair of integers q, r where $a = qb + r$ and $0 \leq r < b$. The number $q = a \operatorname{div} b$ is the quotient, and $r = a \operatorname{rem} b$ is the remainder.

We have the following result, whose proof follows by iteration from Lemma 2.1.

Lemma 2.2 ***Iterating remainders*** For any pair of integers a and b, $\gcd(a, b) = \gcd(a, b \operatorname{rem} a)$.

Proof: $\gcd(a, b) = \gcd(a, b - a) = \gcd(a, b - 2a) = \cdots = \gcd(a, b - qa)$, where q is the quotient. Applying Lemma 2.2 to the examples: $\gcd(4, 10) = \gcd(4, 2) = 2$, $\gcd(3, 7) = \gcd(3, 1) = 1$, and $\gcd(5, 25) = \gcd(5, 0) = 5$.

If $\gcd(a, b) = 1$, a and b are called **relatively prime**. That is important as if $\gcd(a, b) = 1$ we shall see that we will be able to identify a multiplicative inverse for a taken mod b. A simple approach to identifying $\gcd(a, b)$ is to use **Euclid's algorithm** as described in Euclid's Elements circa 300 BC. With a little additional bookkeeping to keep track of a linear combination of a and b along the way, we get the **Extended Euclidean Algorithm**, also known as the **Pulverizer**. The Pulverizer gives us a minimal i.l.c for the gcd, providing (not necessarily non-negative) integers s, t such that $as + bt = \gcd(a, b)$.

Take $a \geq b \geq 0$, where we do not have both a and b equal to 0. A finite-state machine for Euclid's algorithm is as follows:

- Start at state $(x, y) = (a, b)$.
- From each state (x, y), evaluate the quotient q and remainder r, where $x = qy + r$ and $0 \leq r < y$, and transition to $(y, x \operatorname{rem} y)$ as long as $y > 0$.
- The final state (x, y) has $y = 0$ and the output is $\gcd(a, b) = x$.

It remains to understand why Euclid's algorithm identifies $\gcd(a, b)$, but let's first see that it works in an example where the additional bookkeeping in braces is the Pulverizer in operation – it solely keeps track of the sequence of remainders in terms of the original a and b. Consider $a = 69$ and $b = 15$. We have

- $(69, 15)$, where $69 = 4(15) + 9$ (and $9 = 69 - 4(15)$).
- $(15, 9)$, where $15 = 1(9) + 6$ (and $6 = 15 - 9 = 15 - (69 - 4(15)) = -69 + 5(15)$).
- $(9, 6)$, where $9 = 1(6) + 3$ (and $3 = 9 - 6 = 2(69) - 9(15)$).
- $(6, 3)$, where $6 = 2(3) + 0$ (and $0 = 6 - 2(3) = -5(69) + 23(15)$).
- $(3, 0)$, where $3 = 1(3) + 0$.

Hence $\gcd(69, 15) = 3 = 2(69) - 9(15)$, so the algorithm appears to work and by keeping track of the representation of the remainder in terms of a and b, we get a representation of the gcd as an i.l.c with $s = 2$ and $t = -9$.

Here is a finite-state machine for the **Pulverizer**:

- Start at state $(a, b, 1, 0, 0, 1)$.
- From each state (x, y, s, t, s', t'), transition to $(y, x \operatorname{rem} y, s', t', s + s'(x \operatorname{div} y), t - t'(x \operatorname{div} y))$ as long as $y > 0$.
- The final state $(x_f, y_f, s_f, t_f, s'_f, t'_f)$ has $y_f = 0$ and the output is $\gcd(a, b) = x_f$, and we have that s_f and t_f are such that $s_f a + t_f b = \gcd(a, b)$.

Let's reconsider our earlier example with $a = 69$ and $b = 15$. Following the finite-state machine, we will get the same answer as before.

- $(69, 15, 1, 0, 0, 1)$, where $69 = 4(15) + 9$ so that $x \operatorname{div} y = 4$ and $x \operatorname{rem} y = 9$.
- $(15, 9, 0, 1, 1, -4)$, where $15 = 1(9) + 6$ so that $x \operatorname{div} y = 1$ and $x \operatorname{rem} y = 6$.
- $(9, 6, 1, -4, 1, 5)$, where $9 = 1(6) + 3$ so that $x \operatorname{div} y = 1$ and $x \operatorname{rem} y = 3$.
- $(6, 3, 1, 5, 2, -9)$, where $6 = 2(3) + 0$ so that $x \operatorname{div} y = 2$ and $x \operatorname{rem} y = 0$.
- $(3, 0, 2, -9, 5, 23)$, where $3 = 1(3) + 0$.

Quick check: $2(69) + (-9)(15) = 3$. We have **pulverized!**

Let us execute the Pulverizer again with another example, this time with $a = 1001$ and $b = 777$.

$$\begin{aligned}
1001 &= 1(777) + 224 & 224 &= (1001) - 1(777)\\
&& &= (a) - 1(b) = a - b\\
777 &= 3(224) + 105 & 105 &= (777) - 3(224)\\
&& &= (b) - 3(a - b) = -3a + 4b\\
224 &= 2(105) + 14 & 14 &= (224) - 2(105)\\
&& &= (a - b) - 2(-3a + 4b) = 7a - 9b\\
105 &= 7(14) + 7 & 7 &= (105) - 7(14)\\
&& &= (-3a + 4b) - 7(7a - 9b) = -52a + 67b\\
14 &= 7(2), &&
\end{aligned}$$

and so $\gcd(a, b) = 7 = -52(a) + 67(b) = -52(1001) + 67(777)$.

Using Lemma 2.2, the consistency of the propositions in Euclid's algorithm is clear. To verify the correctness of the procedure, the only concern is termination. However, $x + y$ strictly decreases as $x \operatorname{rem} y < x$ when $y \leq x$ and thus the number of steps to reach $\gcd(a, b) = \gcd(x, 0) = x$ is at most $a + b$.

In the **Pulverizer**, each number in Euclid's algorithm is expressed as an i.l.c. of a and b. We can thus find integers s, t such that $as + bt = \gcd(a, b)$. To convince ourselves of the veracity, we must show that such a representation is always possible.

Theorem 2.2 ***Bézout's Identity*** The GCD of a and b, $\gcd(a, b)$, can be written as an i.l.c of a and b. In other words, there exist s, t such that $\gcd(a, b) = as + bt$.

Proof: If we were not restricted to integers, the statement of Bézout's Identity would be immediate. As we are limited to integers, note that $\gcd(a, b)$ divides every i.l.c. $sa + tb$. Hence $\gcd(a, b)$ must be less than or equal to every i.l.c $sa + tb$. Define the set

$$S := \{sa + tb | s, t \in \mathbb{Z} \text{ and } sa + tb > 0\},$$

which is a subset of $\mathbb{Z}$ and is non-empty. Let m be the smallest element of S. Note that $\gcd(a, b)|m$. So, $\gcd(a, b) \leq m$. Now we shall now show $m|a$.

We can use remainder notation to write $a = qm + r$ for $0 \leq r < m$ for some q. The proof follows by contradiction. Assume that $r \neq 0$. Recall that $m = sa + tb$ and $s, t \in \mathbb{Z}$. Hence, $r = a - qm = a - q(sa + tb) = (1 - qs)a + (-qt)b$. Now, $(1 - qs), (-qt) \in \mathbb{Z}$. Therefore, $r \in S$. But this is a contradiction, because m is the smallest element of S. Therefore, $r = 0$ and $m|a$. The same argument holds to show that $m|b$. So, we have that m is a divisor of both a and b. Hence $\gcd(a, b) \geq m$. Since $\gcd(a, b) \leq m$ and $\gcd(a, b) \geq m$, it is the case that $\gcd(a, b) = m$.

In general, note that s and t need not be unique. Consider $a = b = 2$, then we could have $s = 1$ and $t = 0$ or vice versa. Moreover, the proof above did not tell us how to find an s and t pair, it only told us that they must exist. It is the **Pulverizer** that enabled us to identify them.

For our purposes, the importance of the Pulverizer stems from the following proposition.

Proposition 2.1 ***(Multiplicative inverse element from the Pulverizer)*** If $\gcd(a, b) = sa + tb = 1$, then $s \bmod b$ is the multiplicative inverse of $a \bmod b$ and vice versa.

Proof: From Bézout's Identity we have that $\gcd(a, b) = as + bt$ and so

$$(as + bt) \bmod b = as \bmod b = 1,$$

hence $s \bmod b = a^{-1}$. We have found our first method beyond brute force to identify a multiplicative inverse! We can Pulverize!

Let us execute the Pulverizer again with another example, this time with $a = 9$ and $b = 7$.

$$\begin{aligned}
9 &= 1(7) + 2 & 2 &= (9) - 1(7) & &= (a) - 1(b) & &= a - b \\
7 &= 3(2) + 1 & 1 &= (7) - 3(2) & &= (b) - 3(a - b) & &= -3a + 4b \\
2 &= 2(1) + 0 \\
1 &= 1(1) + 0
\end{aligned}$$

and so $\gcd(a, b) = 1 = -3(9) + 4(7)$ and $b^{-1} = 4 \bmod 9 = 4$. Check $7 \cdot 4 \bmod 9 = 1$. Similarly, $a^{-1} = -3 \bmod 7 = 4$, which can be checked with $4 \cdot 7 \bmod 9 = 1$.

Exercise 2.2

Consider $a = 7$, $b = 3$ and $a = 12$, $b = 7$, Pulverize and confirm the resulting i.l.c. identifies inverses.

2.3 Division with Modulo in the Integers – Why Primes

We are all used to doing modulo arithmetic in our daily life. One of the simplest examples is a clock. If we use a 12 hour clock, then we know that when we say "it is 3 o'clock," it means that 12 hours from now, or any multiple of 12 hours from now, it will be 3 again. Thirteen hours from now, or 1 hour plus any multiple of 12 from now, it will be 4, and so on. We are working modulo 12. If we use a 24 hour clock, then we are working modulo 24.

Another example is the days of the week. If it is Monday today, then any number of weeks from now, so any multiple of 7 days from now, it will be Monday again. For the days of the week, we are working modulo 7. While we are used to working with modulo arithmetic, it will turn out that only modulo a prime ensures we can consistently divide.

Definition 2.3 We say a is congruent to b modulo n (represented as $a \equiv b \mod n$) if and only if $n \mid a - b$.

For mod n, a number k is always congruent to its remainder: if $k = nq + r$, then $n \mid nq = k - r$, so $k \bmod n = r \bmod n$.

Theorem 2.3 For any pair of integers a and b, $a \equiv b \mod n$ if and only if $(a \operatorname{rem} n) = (b \operatorname{rem} n)$.

Proof: If $(a \operatorname{rem} n) = (b \operatorname{rem} n) = r$, then $a = nq + r$ and $b = nq' + r$ for some q, q'. So $a - b = n(q - q')$ which is a multiple of n, so $a \bmod n = b \bmod n$ and $a \equiv b \mod n$.

For the converse, consider $a \bmod n = b \bmod n$, so $a - b = nk$ for some k. Then $b = qn + r$ where $0 \le r \le n-1$, so $r = (b \text{ rem } n)$. Therefore, $a \bmod n = b + nk = (k+q)n + r$. Since $0 \le r \le n-1$, $k+q$ and r are the unique values guaranteed by the Division Theorem, i.e. r also equals a rem n.

We can now start doing arithmetic with modulo.

Theorem 2.4 If $a \bmod n = a' \bmod n$, then for any b, $a + b \bmod n = a' + b \bmod n$ and $ab \bmod n = a'b \bmod n$.

Proof: $a' = a + sn$. Then $(a' + b) - (a + b) = sn$, a multiple of n. Likewise, $a'b - ab = (a + sn)b - ab = bsn$, another multiple of n.

When adding, subtracting, or multiplying, we can replace a by any number to which it is congruent to $a \bmod n$, without changing the result mod n. For exam-
ple, $a \cdot b \cdot c \cdot d + e \cdot f \bmod n = ((a \cdot b \bmod n)(c \cdot d \bmod n) + (e \cdot f \bmod n)) \bmod n$. E.g. $(101 + 203) \bmod 10 = 4$ and $(55 \cdot 95) \bmod 10 = (55 \bmod 10)(95 \bmod 10) \bmod 10 = 25 \bmod 10 = 5$.

We can also use modulo for the base of exponents.

Theorem 2.5 If $a \bmod n = a' \bmod n$, then for any $k \ge 0$, $a^k \bmod n = (a')^k \bmod n$.

Proof: This is just repeated multiplication, so apply the previous lemma k times: $a \cdot a \cdots a = a' \cdot a' \cdots a' \bmod n$.

We can place intermediate calculations with their remainders, which helps us work with smaller numbers. For example, as $42 \bmod 11 = 9 \bmod 11 = 9$, we can readily evaluate $42^2 \bmod 11 = 9^2 \bmod 11 = 81 \bmod 11 = 4$.

Note that we cannot extend the mod arithmetic properties to the exponent k. You can see this with a counterexample. Consider the evaluation of

$$x = 35^{11111}(6 + 99^{5000}) \bmod 100.$$

It is not correct to just reduce the exponents mod 100. For the right exponent, $99^2 = 1 \bmod 100$, so $99^{5000} = 1 \bmod 100$. For the left term, look for a pattern:

$$\begin{aligned}
35^1 &= 35 \bmod 100\\
35^2 &= 25 \bmod 100\\
35^3 &= 25 \cdot 35 = 75 \bmod 100\\
35^4 &= 75 \cdot 35 = 25 \bmod 100.
\end{aligned}$$

Will continue bouncing between 25 and 75. So $35^{11111} = 75 \bmod 100$. We find $x = 75 \cdot (6+1) \bmod 100 = 25 \bmod 100$, so this must be the remainder.

Thus, addition, subtraction, and multiplication modulo an integer are easy. Recall that for integers the tricky operation was division. This is going to be true again when we deal with integers using modulo operations. Let us consider for example that $3 \cdot 2 = 3 \cdot 4 \bmod 6$. Can we "divide both sides by 3" and conclude that $2 = 4 \bmod 6$? No.

Subtraction is the inverse of addition. Can we find the inverse of multiplication? Let us think on reals, where there is no multiplicative inverse of 0. Maybe we can do multiplicative inverses, but we do not expect 0 to have an inverse.

A **multiplicative inverse** of x is a number you can multiply x by to get 1. In the real, $\mathbb{R}$, the multiplicative inverse of 3 is $1/3$, because $3 \cdot 1/3 = 1$. If "$1/3$" made sense $\bmod 6$, then we could multiply both sides of the last example by $1/3$ to conclude that $2 = 4 \bmod 6$, so 3 doesn't have a multiplicative inverse mod 6.

There are two parts for multiplicative inverses. First, do they exist? Second, if they do exist, how do we find them? We shall tackle these two questions in the discussion below.

2.3.1 Existence of Multiplicative Inverses**

When do $\bmod n$ inverses exist for a number a?

Theorem 2.6 An integer $a \neq 0$ has a multiplicative inverse $\bmod n$ if and only if $\gcd(a, n) = 1$.

Proof: ** a has an inverse $\bmod n$ if and only if there exists b such that $ab = 1 \bmod n$ if and only if exists b and q such that $ab - 1 = nq$, i.e. $ab - nq = 1$ if and only if 1 is a linear combination of a and n. By Bézout's Identity, this last part happens if and only if $\gcd(a, n) = 1$.

Having a multiplicative inverse means we "can cancel from both sides" or "divide" by that amount. E.g. 7 and 13 are inverses of each other mod 30. If we know $7x = 14 \bmod 30$ can we conclude that $x = 2 \bmod 30$? Instead of dividing, let's multiply both sides by 13:

$$\begin{aligned} 7x &= 14 && \bmod 30 \\ 13 \cdot 7x &= 13 \cdot 14 && \bmod 30 \\ (13 \cdot 7)x &= (13 \cdot 7) \cdot 2 && \bmod 30 \\ (1)x &= (1) \cdot 2 && \bmod 30. \end{aligned}$$

So yes, since 7 has a multiplicative inverse, we can "cancel it from both sides."

The gcd approach works whether we are doing mod a prime or not, but we may not have multiplicative inverses for all values and so may not have a field. The following corollary establishes that if p is a prime, then using modulo p arithmetic gives us a field (a prime field).

Corollary 2.1 If p is prime and $a \bmod p \neq 0 \bmod p$, then a has a multiplicative inverse mod p.

Proof: $\gcd(a, p)$ must be p (if $p \mid a$) or 1 (only other factor of p). Now apply previous result.

We have established that Prime Fields exist with modulo arithmetic. Let us reassure ourselves that Prime Fields are useful by considering the solution of some linear equations. Consider $\mathbb{F}_7 = \{0, 1, 2, 3, 4, 5, 6\}$. We have the following

a	1	2	3	4	5	6
a^{-1}	1	4	5	2	3	6

which we can confirm by calculating $a \cdot a^{-1} \bmod 7$. Consider the linear equation in normal, real arithmetic:

$$2a = 3.$$

To identify a you would divide both sides by 2 (i.e. multiply by 2^{-1}) to get $a = 1.5$. Consider the same linear equation in modulo arithmetic

$$2 \cdot a \bmod 7 = 3,$$

What is a? Following the same process, we have that

$$a = 2^{-1} \cdot 2 \cdot a \bmod 7 = 2^{-1} \cdot 3 \bmod 7 = 4 \cdot 3 \bmod 7 = 5,$$

and so $a = 5$. To confirm that is correct, we can evaluate that indeed $2 \cdot 5 \bmod 7 = 3$.

Consider another example:

$$c = (4 \cdot a + 2 \cdot b) \bmod 7 = 5$$
$$d = (a + b) \bmod 7 = 1.$$

If you were told c and d and knew the multipliers, how would you evaluate a and b? You might start with

$$c - 2d = (4 \cdot a + 2 \cdot b) - 2 \cdot (a + b) \bmod 7 = 2 \cdot a \bmod 7 = (5 - 2) \bmod 7 = 3,$$

and we know from the previous example that $a = 5$. Using the second equation, we could then deduce that $5 + b \bmod 7 = 1$ so that $b = 3$. To confirm that finding, we can evaluate $4(5) + 2(3) \bmod 7 = 5$ and $5 + 3 \bmod 7 = 1$.

In order to identify multiplicative inverses for $a \in \{1, 2, \dots, p - 1\}$ in the prime field with modulo-p arithmetic, so far we have brute force and the Pulverizer. Fermat's Little Theorem will provide us with one more algorithm.

Theorem 2.7 ***Fermat's Little Theorem*** If p is prime and $a \neq 0 \bmod p$, then $a^{p-1} = 1 \bmod p$.

Proof: The idea is to look at numbers $a, 2a, 3a, \ldots, (p-1)a$ and realize that this is the same as $1, 2, 3, \ldots, (p-1)$ mod p, possibly in jumbled order. For example, with $p = 7$ and $a = 3$, $3, 6, 9, 12, 15, 18 \bmod 7 = 3, 6, 2, 5, 1, 4 \bmod 7$. None of these are $0 \bmod p$, so there are only $p - 1$ possible remainders. As a result, it is sufficient to show there are no duplicates. Note that $ai = aj \bmod p$ implies $i = j \bmod p$, because a has a multiplicative inverse mod p no two of the numbers $1, 2, \ldots, p-1$ are equivalent mod p, and hence, indeed, there are no duplicates. Now, since both sets are same mod p, their product is congruent mod p:

$$(p-1)! \cdot a^{p-1} \bmod p = (p-1)! \bmod p.$$

Since $\gcd((p-1)!, p) = 1$, $(p-1)!$ has an inverse mod p, hence we can cancel it: $a^{p-1} \bmod p = 1 \bmod p$.

As a direct consequence of Fermat's Little Theorem, we have the following alternative algorithm for identifying multiplicative inverses in a prime field.

Proposition 2.2 ***Multiplicative inverses via Fermat*** Let p be a prime and let $a \in \{1, 2, \ldots, p-1\}$, then $a^{p-1} = a \cdot a^{p-2} = 1 \bmod p$ and hence the multiplicative inverse of $a \bmod p$ is $a^{p-2} \bmod p$.

Modular arithmetic provides tools for problems beyond our needs, but here is one fun example.

Theorem 2.8 A number is divisible by 9 if and only if its sum of digits is divisible by 9.

Proof: Say $n = \sum_{i=0}^{k} d_i 10^i$. Note that

$$10^i \bmod 9 = 1^i \bmod 9 = 1 \bmod 9, \text{so} \sum_{i=0}^{k} d_i 10^i \bmod 9 = \sum_{i=0}^{k} d_i \cdot 1 \bmod 9.$$

We just need to check whether both sides are 0.

Exercise 2.3

For all prime values $q < 15$, empirically establish that for any $a \in \{1, \ldots, q-1\}$

$$a^{-1} = a^{q-2} \bmod q \tag{2.1}$$

and that, for a given q, each a^{-1} is distinctly associated to a single a. We deduce that divisors do exist and can be determined by Eq. (2.1) for values in this range.

Exercise 2.4

Using either the formula in Eq. (2.1) or the brute force approach, for each of $q \in \{4, 6, 8, 9, 10, 12, 14, 15\}$ establish that divisors are not unique (i.e. there exist at least one pair $a \neq b \in S$ such that $a^{-1} = b^{-1}$ and/or at least one element has no inverse).

2.4 Prime Fields

At this stage, we know we can now define arithmetic modulo a prime to create a prime field and we can evaluate integer division using the **Pulverizer** or **Fermat's Little Theorem** or by building a **lookup table of inverses by brute force**.

Armed with these, we can proceed to use much of our knowledge from real-valued systems by replacing division with multiplication by an inverse. In particular, to implement Gaussian or Gauss–Jordan elimination, that's what one should do.

Note that while the operation of the finite field calculus mimics the more familiar one in, say, the reals, the numerical outcomes are quite different. For example, consider again $\mathbb{F}_7 = \{0, 1, 2, 3, 4, 5, 6\}$ where we have the following collection of inverses

a	1	2	3	4	5	6
a^{-1}	1	4	5	2	3	6

Given the matrix

$$A = \begin{pmatrix} 2 & 2 \\ 5 & 1 \end{pmatrix},$$

in the reals we would identify its inverse as

$$A^{-1} = \begin{pmatrix} -1/8 & 1/4 \\ 5/8 & -1/4 \end{pmatrix} \text{ as } AA^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

In $\mathbb{F}_7$, however, we would identify its inverse as

$$A^{-1} = \begin{pmatrix} 6 & 2 \\ 5 & 5 \end{pmatrix},$$

which can be checked by confirming $AA^{-1} \bmod 7$ is the 2×2 identity. Clearly, the entries in A^{-1} in the finite field $\mathbb{F}_7$ bear no relation to the entries in the inverse matrix in the reals. In terms of properties, once considered with the correct $+$ and $\cdot$ operations, however, they both function the same.

Let us temporarily return to the ramifications for our coding application in a toy example. Imagine we had an encoding matrix in $\mathbb{F}_7$ that took two inputs and returns three outputs, where the first two outputs correspond to the linear combinations we have just considered:

$$B = \begin{pmatrix} 2 & 2 \\ 5 & 1 \\ 6 & 2 \end{pmatrix}.$$

Then two pieces of data $X_{(1)} \in \mathbb{F}_7$ and $X_{(2)} \in \mathbb{F}_7$ can be coded into three by

$$B \begin{pmatrix} X_{(1)} \\ X_{(2)} \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 5 & 1 \\ 6 & 2 \end{pmatrix} \begin{pmatrix} X_{(1)} \\ X_{(2)} \end{pmatrix} = \begin{pmatrix} 2 \cdot X_{(1)} + 2 \cdot X_{(2)} \bmod 7 \\ 5 \cdot X_{(1)} + 1 \cdot X_{(2)} \bmod 7 \\ 6 \cdot X_{(1)} + 2 \cdot X_{(2)} \bmod 7 \end{pmatrix}.$$

If the first two packets are received correctly, they would equal $A(X_{(1)}, X_{(2)})^T$, where A is given above, but

$$A^{-1}A \begin{pmatrix} X_{(1)} \\ X_{(2)} \end{pmatrix} = \begin{pmatrix} X_{(1)} \\ X_{(2)} \end{pmatrix},$$

so to recover the transmitted data, the receiver would merely have to multiply by the inverse matrix to the coding matrix (i.e. perform **Gaussian Elimination**). That this works can be established by checking that all 2×2 sub-matrices of B are invertible. As a result, one can deduce that receiving any two linear combinations would enable the receiver to decode and determine the original data, where we have the following matrix and inverse pairs

$$A = \begin{pmatrix} 2 & 2 \\ 6 & 2 \end{pmatrix}, \qquad A^{-1} = \begin{pmatrix} 5 & 2 \\ 6 & 5 \end{pmatrix}$$
$$\text{and}\, A = \begin{pmatrix} 5 & 1 \\ 6 & 2 \end{pmatrix}, \qquad A^{-1} = \begin{pmatrix} 4 & 5 \\ 2 & 3 \end{pmatrix}.$$

Note that while data at the source would most likely be binary, and the greatest n such that $2^n < 7$ is 2, when the data are coded together they become elements of $\mathbb{F}_7$ which is why it is worth noting everything follows through for $X_{(i)} \in \mathbb{F}_7$ and not just $X_{(1)} \in \{0, 1, 2, 3\}$.

2.5 Mathematical Aside: Beyond Linear Equations**

While we will only need linear equations for the coding we will consider, note that we can do all the arithmetic we want in a finite field. To help cement our belief in this fact, consider quadratic equations. Start by noting that if we build a table of squares in $\mathbb{F}_7$

a	1	2	3	4	5	6
a^2	1	4	2	2	4	1

then we can deduce a table of square roots

a	1	2	4
$a^{1/2}$	1, 6	3, 4	2, 5

In the reals, we're used to numbers having two square roots, one positive and one negative, so while it might be odd that we have two positive square roots for some numbers, it shouldn't seem exceptional. Nor should it seem exceptional that some numbers, here 5 and 6, have no square root. After all, all negative numbers in the reals have no square root in the reals.

Imagine we wish to solve a quadratic equation

$$a \cdot x^2 + b \cdot x + c = 0.$$

We know the following formula, which we normally think of as only being appropriate for real numbers:

$$x = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}. \tag{2.2}$$

For example, if $a = 2$, $b = 1$ and $c = 4$, then the equation

$$2 \cdot x^2 + x + 4 = 0 \tag{2.3}$$

has no solution in the reals, as $b^2 - 4 \cdot a \cdot c < 0$.

What if we consider quadratics in $\mathbb{F}_7$? Let's reconsider Eq. (2.2) in the finite field, where everything is being done in the mod 7 arithmetic, making note of the following terms:

$$\begin{aligned} -b &= -1 = 6 \\ b^2 &= 1^2 = 1 \\ 4 \cdot a \cdot c &= 4 \cdot 2 \cdot 4 = 32 \bmod 7 = 4 \\ 2 \cdot a &= 2 \cdot 2 = 4 \\ (2 \cdot a)^{-1} &= 4^{-1} = 2. \end{aligned}$$

We have to consider

$$\sqrt{b^2 - 4 \cdot a \cdot c} = \sqrt{1 - 4} = \sqrt{-3} = \sqrt{4} = \{2, 5\}.$$

Hence, the solutions to the quadratic equation described in Eq. (2.3) when considered in $\mathbb{F}_7$ are

$$(6 \pm 2) \cdot 2 \bmod 7 = \{8, 16\} \bmod 7 = \{1, 2\}$$
$$(6 \pm 5) \cdot 2 \bmod 7 = \{22, 2\} \bmod 7 = \{1, 2\}.$$

To confirm that these are correct, we can evaluate

$$\begin{aligned} & 2 \cdot x^2 + 1 \cdot x + 4 \\ \text{when } x = 1, \quad & 2 + 1 + 4 \bmod 7 = 0 \\ \text{and when } x = 2, \quad & 2 \cdot 2^2 + 1 \cdot 2 + 4 \bmod 7 = 14 \bmod 7 = 0. \end{aligned}$$

If one places $x = 3, 4, 5$ or 6 into the quadratic equation (2.3) considered in $\mathbb{F}_7$, one finds it will give an nonzero evaluation, and so there are no other solutions.

That is, not only do quadratic equations make sense in a prime field, but we can also identify their solution by following the usual procedure from the reals, simply taking care to ensure that addition, multiplication, subtraction, and division are performed according to the rules of the prime field.

2.6 Extension Fields

As we have discussed, when p is a prime number, the integers modulo p constitute a prime field with the elements $\{0, 1, \ldots, p-1\}$. Recall that the tricky part was division, i.e. the finding of a multiplicative inverse, and we managed that already.

Let us return to the example of the clock. If we work with modulo 12 as in the clock, a difficulty arises. As Example 2.2 points out, $\mathbb{F}_{12}$ is not a finite field and we cannot do the arithmetic there in a way that makes sense. The problem arose from the fact that we could factor 12 into 3 and 4. If we work modulo a prime, as we do with the days of the week, then that issue does not arise. For example, if we work with the days of the week, then we have a prime field, because 7 is prime, where the first day of the week is 0.

A natural question may be: if our numbers are represented as bits, so using a binary representation, will there be a problem representing the data in a prime field? It turns out that the loss from changing the arithmetic field can be made small, as explored in the following exercise.

Exercise 2.5

a) Assuming divisors exist for any prime value p so that modulo p arithmetic works, for a prime value $p > 2^n$, what is the smallest integer $n'(p)$ that can be used to represent the values $S = \{0, 1, \ldots, p-1\}$ in binary?

b) For n from 6 to 12, plot the fractional loss of representation (i.e. fraction of binary strings that will not be used to represent an element in S), $(2^{n'(p)} - p)/2^{n'(q)}$ for each $2^6 < p < 2^{12}$.
c) As a function of $n \in \{6, 7, \ldots, 12\}$, plot the fractional loss for the largest p that satisfies $2^n < p < 2^{n'(p)}$.

2.6.1 Suitability for Binary Data

A finite field extends the definition of a field by specifying that the operations addition and multiplication, including their inverse subtraction and division, performed on the field elements must result in an element within the field. For integers, this property is achieved using modulo arithmetic. In general, any integer i can be mapped onto a chosen field as $i \bmod p$, when p is a prime number.

The smallest possible finite field, $\mathbb{F}_p$, is the prime field with $p = 2$ also called the **binary field**. In this case, the order and characteristic, the smallest positive number of copies of the multiplicative identity 1 that sum to the additive identity 0, of the finite field are equal. However, as we shall see, if using extension fields this is not always the case. The binary field consists of two elements $\{0, 1\}$ and is of particular interest since the binary operations are easily implemented and represented in software and hardware. In the case of the binary fields, addition and multiplication are performed mod 2.

We can do Boolean logic as arithmetic in $\mathbb{F}_2$. In Table 2.1 it can be seen that addition can be implemented using the bitwise XOR ($\wedge$) operator and multiplication by the bitwise AND (&) operators found in most programming languages. If an implementation only required operations on 1-bit field elements, a similar subtraction and division table could be constructed, where it would quickly be seen that addition and subtraction are identical over $\mathbb{F}_2$. The division table is identical to the multiplication table, except that division by zero would not be allowed. Performing calculations in the 1-bit binary field would however not yield very efficient implementations, as most hardware does not naively support a 1 bit data type, i.e. most computer hardware is optimized for operating on 8, 16, 32, or 64 bit data types. Applications working with $\mathbb{F}_2$ therefore typically perform computations using vectors of 1 bit field elements to make better use of the underlying hardware. For a range of reasons that depend on the application, it may be desirable to increase the field size and a common way of achieving this is by extending the field.

While $\mathbb{F}_2$ is easy to work in, it does not give us a lot of choices. Recall that our goal in NC is to create systems of linear equations that can be solved with Gauss–Jordan elimination. If we are operating in $\mathbb{F}_2$, then we do not have many choices. For example, if we want to make equations involving two variables x and y, then our **only** choice is $x + y$, where the coefficient for each variable is 1. The lack of options quickly becomes a problem. Indeed, many of the issues with traditional coding techniques center around the fact that they operate over binary fields and are therefore severely limited in their constructions. They end up having to create complicated schemes because they remain stubbornly binary.

How can we move away from having almost no choices? We need to grow the number of elements in a field, the size, or cardinality, also called order of the field. When working with higher order fields, many advantages naturally present themselves. For example, we will see that in a finite field of order q, the fraction of all matrices that are invertible, and therefore have uniquely solvable equations, is approximately $1 - 1/q$. For binary, this means about half of all systems of linear equations are not solvable. As the field's order increases, this fraction goes to 0, which is a fundamental property underlying the utility of Random Linear Network Coding (RLNC) when deployed with fields operating on data frames of bytes or bigger. When working in an extension field on bytes, which consist of 8 bits, the approximation says that $100(1 - 2^{-8})\% > 99.6\%$ of randomly selected matrices are invertible.

We have two ways of growing the order of our fields. The first is to stay with prime fields, which we have developed, and make the prime larger. The second is extension fields, where we are going to start with a prime, commonly just 2, and create from it a larger order field through the use of polynomial representations and arithmetic. Polynomials can have degrees that have a natural, direct engineering interpretation as delay.

Exercise 2.6

a) Write code that takes binary strings of length n and maps them to an integer representation in $\{0, 1, \dots, 2^n - 1\}$.
b) Repeatedly randomly generate sets of $K = 3$ binary strings, each of length $n = 4$ bits, and map them to integers, $(X_{(1)}, X_{(2)}, X_{(3)})$. For the largest prime $q > 2^n$ that minimizes the fractional loss, create two random $K \times K$ encoding matrices in S by selecting the entries, $\alpha_{(i,j)}$ and $\beta_{(i,j)}$ for $i, j \in \{1, \dots, K\}$ uniformly at random in S. Using multiplication mod q, empirically establish:

$$\begin{pmatrix} \beta_{(1,1)} & \cdots & \beta_{(1,K)} \\ \vdots & \vdots & \vdots \\ \beta_{(K,1)} & \cdots & \beta_{(K,K)} \end{pmatrix} \left(\begin{pmatrix} \alpha_{(1,1)} & \cdots & \alpha_{(1,K)} \\ \vdots & \vdots & \vdots \\ \alpha_{(K,1)} & \cdots & \alpha_{(K,K)} \end{pmatrix} \begin{pmatrix} X_{(1)} \\ \vdots \\ X_{(K)} \end{pmatrix} \right)$$
$$= \left(\begin{pmatrix} \beta_{(1,1)} & \cdots & \beta_{(1,K)} \\ \vdots & \vdots & \vdots \\ \beta_{(K,1)} & \cdots & \beta_{(K,K)} \end{pmatrix} \begin{pmatrix} \alpha_{(1,1)} & \cdots & \alpha_{(1,K)} \\ \vdots & \vdots & \vdots \\ \alpha_{(K,1)} & \cdots & \alpha_{(K,K)} \end{pmatrix} \right) \begin{pmatrix} X_{(1)} \\ \vdots \\ X_{(K)} \end{pmatrix}.$$

In the erasure coding context, this means we can recode without having to decode.

c) If α and β entries are chosen uniformly at random, what is the empirical frequency of the entries of the β times α matrix? Do each of the values in $S = \{0, 1, \dots, q-1\}$ seem to appear equally often or are some values more prevalent.

2.6.2 Basics of Extension Fields

When implementing finite field operations the number of elements in a field can be increased by using polynomials to represent the field elements. A nonzero polynomial $f(x)$ of degree m over a field $\mathbb{F}_{q^m}$, where q is a prime and $m > 1$, has the form

$$f(x) = f_{m-1}x^{m-1} + \cdots + f_2x^2 + f_1x + f_0, \tag{2.4}$$

where the coefficients $f_i \in \mathbb{F}_q$ for $0 \leq i \leq m$. Polynomials essentially allow the same arithmetic operations as integers; however, when polynomials are used, the operations are performed mod-$p(x)$, where $p(x)$ is an irreducible polynomial. As with a prime number, an irreducible polynomial is one that cannot be factored into products of two polynomials. It can be shown that we may find irreducible polynomials for any p and m [1, p. 28], which we will assume without proof. For example $x^2 + 6x + 9$ is not an irreducible polynomial since $(x + 3) \cdot (x + 3) = x^2 + 6x + 9$. However, $x + 1$ is an irreducible polynomial since it cannot be factorized.

Ordinary polynomial addition, i.e. without being in a finite field, is performed component-wise, e.g. for two polynomials with a maximum degree of k

$$f(x) = h(x) + g(x). \tag{2.5}$$

$$f(x) = \sum_{i=0}^{k} (h_i + g_i)x^i. \tag{2.6}$$

For ordinary polynomial multiplication, the coefficients of $f(x) = h(x)g(x)$ are determined by convolution, the resulting polynomial $f(x)$ is of degree equal to the sum of the degrees of h and g:

$$f_i = \sum_{j=0}^{i} h_j g_{i-j}, \tag{2.7}$$

which is called the convolution of the coefficients of $h(x)$ and $g(x)$. For readers used to work in the transform domain, which we shall allude to again in Section 2.9, the connections between convolution and multiplication may be familiar (multiplication in the original domain of representation generally maps to convolution in the transform domain and vice versa). There is no need, however, to be in any way familiar with transforms to follow any of this presentation.

When working in an extension field, we calculate $f(x) + g(x)$ as $f(x) + g(x) \bmod p(x)$, where $p(x)$ is a polynomial of degree less than m that we will return to soon. This uses the usual component-wise addition as given in Eq. (2.6), the only difference is that each individual coefficient sum is modulo q, where q is the prime used to establish the extension field rather than the polynomial $p(x)$, i.e. $h_i + g_i \bmod q$.

The product $h(x)g(x)$ in $\mathbb{F}_{q^m}$ can be found by first multiplying $h(x)$ and $g(x)$ using ordinary polynomial multiplication. Then we ensure that the resulting polynomial $f(x)$ has degree $< m$ by reducing it modulo $p(x)$. The modulo operation can be implemented as polynomial long division, the natural extension of Euclidian division, and then taking the remainder, which parallels the development for integer modulo arithmetic, and we will expand on that aspect soon. As for polynomial addition, we must also ensure that all resulting coefficients are elements in $\mathbb{F}_q$ by reducing them modulo q.

As all primes other than 2 are odd, and a positive integer power of an odd number is odd, only binary extension fields give us something evidently distinct from the prime fields we have already considered.

2.6.3 Binary Extension Field

Of particular interest are binary extension fields, $\mathbb{F}_{2^m}$. These fields are common as the binary representation allows efficient implementations in either software or hardware. In the binary extension field, all field elements may be represented as binary polynomials of degree at most $m - 1$:

$$\mathbb{F}_{2^m} = \left\{ f_{m-1}x^{m-1} + \ldots + f_2x^2 + f_1x + f_0 \ : f_i \in \{0, 1\} \right\} .$$

In the binary extension field, all polynomial elements can be represented as m bit binary numbers. It is important to notice the correspondence between the binary and polynomial representation. The bits from right to left are the coefficients

Table 2.2 Example mapping between polynomial and binary representation for a polynomial in $\mathbb{F}_{2^8}$.

1 1 0 1 0 0 1 1
$1x^7 + 1x^6 + 0x^5 + 1x^4 + 0x^3 + 0x^2 + 1x + 1x^0$
$x^7 + x^6 + x^4 + x + 1$

Table 2.3 Field elements of the Galois field GF(2^3) in polynomial and binary representation.

$\mathbb{F}_{2^3} = \mathrm{GF}(2^3)$		
Polynomial	**Binary**	**Decimal**
0	000	0
1	001	1
x	010	2
$x+1$	011	3
x^2	100	4
x^2+1	101	5
x^2+x	110	6
x^2+x+1	111	7

of the powers of x in increasing order. Table 2.2 shows the direct mapping for a polynomial in $\mathbb{F}_{2^8}$.

As an example consider the field given by $\mathbb{F}_{2^3}$ in Table 2.3; in this case the field will consist of $2^3 = 8$ polynomial elements of degree $< m$.

The following table provides some examples of representations of elements of $\mathbb{F}_{2^8}$ for you to fill in, with solutions in Section 2.10.

Polynomial	Binary	Integer
$x^7+x^6+x^4+x+1$		
	11001001	
		133
x^4+x^2+x		
	00011001	
		8

Understanding the connection between the different representations will be important to understand how finite fields are implemented in software. In general, we can construct $\mathbb{F}_2$ extension fields containing any 2^m number of elements where $m \geq 1$. In an implementation, the field size can be chosen so that the polynomial representation will fit into an available data type, e.g. $\mathbb{F}_{2^8}$ can typically be represented in an unsigned char. For this reason, fields of $2^8, 2^{16}, 2^{32}$ elements are common choices.

For polynomials defined on an extension field $\mathbb{F}_{p^m}$, Eq. (2.6) tells us what to do: for each x^k we simply add the corresponding coefficients mod p. In a binary extension field $p = 2$ and so we are reduced to doing binary evaluations. Some examples to try:

$a(x)$	$b(x)$	$(a+b)(x)$
$x^7+x^6+x^2$	$x^7+x^5+x^3+x^2$	
x^5+x	x^3+x^2	
x^7+x^3	x^7+x+1	
x^3+x^2+x+1	$x+1$	
$x^7+x^6+x^5+x^4+x^3+x^2+1$	x^4+x^2+1	

In the binary field, addition and subtraction are identical. Consequently, the result of $(a-b)(x)$ is the same as the result of $(a+b)(x)$.

2.7 Polynomial Multiplication/Division

For ordinary polynomial multiplication without modulo the coefficients of $f(x) = h(x)g(x)$ are determined by convolution and the resulting polynomial $f(x)$ is of degree $\deg(h) + \deg(g)$. Some examples to try, where the last one is the most interesting:

$a(x)$	$b(x)$	$a(x)b(x)$
x^5+x	x^3+x^2	
x^7+x^3	x^7+x+1	
x^3+x^2+x+1	$x+1$	

2.7.1 Binary Extension Field Multiplication

Polynomial multiplication in an extension field can be thought of as the multiplication of the polynomials in the finite field restriction, followed by reduction to a remainder with an irreducible polynomial. In the following example, it is shown how to perform arithmetic operations on two polynomials from Table 2.3. The irreducible polynomial $p(x) = x^3 + x + 1$ is used to ensure that all calculations are within $\mathbb{F}_{2^3}$. Note, that coefficient operations are still performed in $\mathbb{F}_2$ and we can therefore use the addition table in Table 2.1 when adding coefficients and subsequently Eq. (2.5).

As mentioned, the modulo operation may be implemented using polynomial long division, taking the remainder. As an example, the following computes the polynomial long division of $(x^3 + x) \mod (x^3 + x + 1)$ shown in the multiplication example from above:

$$\begin{array}{r|lll} & & & 1 \quad \triangleleft \text{ quotient} \\ \cline{2-4} x^3+x\ +1 & x^3 & +x & \\ & x^3 & +x & +1 \\ \cline{2-4} & & & 1 \quad \triangleleft \text{ remainder} \\ \cline{4-4} \end{array}$$

Walking through the above long division follows the natural extension of the Euclidian division. We subtract $x^3 + x + 1$ from $x^3 + x$. Since the higher order degree is the same in both polynomials, we do not need to multiply $x^3 + x + 1$ by a power of x before subtracting it. That is, the long division gives us the representation $(x^3 + x) = p(x) + 1 = (x^3 + x + 1) + 1$.

If the highest degree of the polynomial we were subtracting (the dividing polynomial) was less than the degree of the polynomial from which we are subtracting, then we would have needed to multiply the dividing polynomial by a power of x, say x^k that made the two polynomials of the same power. That x^k would be recorded above the horizontal line. The remainder would be another polynomial, and we would iterate long division on that new polynomial.

Armed with reduction mod $p(x)$, we have some examples where $p(x) = x^3 + x + 1$.

- **Addition**: $(x^2 + x + 1) + (x^2 + 1) = x$, since $x \bmod (x^3 + x + 1) = x$.
- **Subtraction**: $(x) - (x^2 + x) = x^2$, since $x^2 \bmod (x^3 + x + 1) = x^2$.
- **Multiplication**: $(x^2 + x) \cdot (x + 1) = 1$, since $(x^3 + x) \bmod (x^3 + x + 1) = 1$.
- **Inversion**: $x^{-1} = x^2 + 1$, since $x \cdot (x^2 + 1) \bmod (x^3 + x + 1) = 1$.
- **Division**: $x^2/x = x$, since $x^2 \cdot (x^2 + 1) \bmod (x^3 + x + 1) = x$.

More examples. Consider the irreducible polynomial $p(x) = x^5 + x^2 + 1$ when working in $\mathbb{F}_{2^5}$. We have the polynomial multiplication

$$\begin{aligned} f(x) &= (x^3 + x + 1)(x^2 + 1) \\ &= x^5 + (1 + 1 \bmod 2)x^3 + x^2 + x + 1 \\ &= x^5 + x^2 + x + 1, \end{aligned}$$

and so

$$f(x) \bmod p(x) = (x^5 + x^2 + x + 1) \bmod (x^5 + x^2 + 1) = x.$$

As another example in $\mathbb{F}_{2^5}$ with $p(x) = x^5 + x^2 + 1$,

$$\begin{aligned} f(x) &= (x^4 + x)(x^3 + x^2) \\ &= x^7 + x^6 + x^4 + x^3, \end{aligned}$$

and so

$$f(x) \bmod p(x) = (x^7 + x^6 + x^4 + x^3) \bmod (x^5 + x^2 + 1) = x^2 + x,$$

where when one writes out the long-division one finds a quotient of $x^2 + x$ so that $f(x) = (x^2 + x)p(x) + x^2 + x$. One final example in $\mathbb{F}_{2^5}$ with $p(x) = x^5 + x^2 + 1$,

$$\begin{aligned} f(x) &= (x^3)(x^2 + x + 1) \\ &= x^5 + x^4 + x^3, \end{aligned}$$

and

$$f(x) \bmod p(x) = (x^5 + x^4 + x^3) \bmod (x^5 + x^2 + 1) = x^4 + x^3 + x^2 + 1,$$

and the quotient was 1.

2.7.2 Binary Extension Field Division

As with prime fields, division in an extension field is regarded as multiplication by an inverse. For example, in $\mathbb{F}_{2^5}$ where we are using $p(x) = x^5 + x^2 + 1$, if we wish to compute $(x^4 + x)/(x^3 + x^2) \bmod p(x)$ we think of it as $(x^4 + x) \cdot (x^3 + x^2)^{-1} \bmod p(x)$. One can first verify that $(x^3 + x^2)^{-1} = (x^2 + x + 1)$ by multiplication

$$(x^3 + x^2) \cdot (x^2 + x + 1) \bmod p(x) = x^5 + x^2 \bmod (x^5 + x^2 + 1) = 1.$$

and then evaluate

$$\begin{aligned} (x^4 + x) \cdot (x^3 + x^2)^{-1} \bmod p(x) &= (x^4 + x) \cdot (x^2 + x + 1) \bmod p(x) \\ &= x^6 + x^5 + x^4 + x^3 + x^2 + x \bmod (x^5 + x^2 + 1) \\ &= x^4 + 1. \end{aligned}$$

Exercise 2.7

By hand, execute Euclidian division for a polynomial where the divisor polynomial has a degree strictly smaller than the dividing polynomial for two examples, where the arithmetic is in different finite prime fields for the two examples.

2.8 Primitive Polynomials

In addition to the polynomial, binary, and decimal representation of the binary field elements shown in Table 2.3, in certain cases we may be able to use an alternative representation utilizing two useful properties of binary extension fields, namely primitive polynomials and primitive elements.

For an irreducible polynomial $p(x)$ defined in $\mathbb{F}_{2^m}$ we investigate the result of $x^j \bmod p(x)$, where $j \in \{0, 1, \dots, \infty\}$. We will find that the modulo operation will only produce a finite number of elements, e.g. for $p(x) = x^3 + x + 1$ we obtain:

$$x^0 \bmod p(x) = 1. \tag{2.8}$$

$$x^1 \bmod p(x) = x. \tag{2.9}$$

$$x^2 \bmod p(x) = x^2. \tag{2.10}$$

$$x^3 \bmod p(x) = x + 1. \tag{2.11}$$

$$x^4 \bmod p(x) = x^2 + x. \tag{2.12}$$

$$x^5 \bmod p(x) = x^2 + x + 1. \tag{2.13}$$

$$x^6 \bmod p(x) = x^2 + 1. \tag{2.14}$$

$$x^7 \bmod p(x) = 1. \tag{2.15}$$

$$x^8 \bmod p(x) = x. \tag{2.16}$$

$$\vdots$$

As shown in Eq. (2.8)–(2.16) we obtain only a limited number of remainders from the modulo operation. The number of unique field elements generated by a polynomial is denoted by the polynomial's **period**. In general, we say that the period is the smallest nonzero number P such that:

$$x^P \bmod p(x) = 1 \bmod p(x),$$

where the maximum period that may be obtained for a polynomial of degree m is

$$P_{max} = 2^m - 1.$$

If a polynomial has a period P_{max} we denote it as a **primitive polynomial**.

Correspondingly, any element α for which increasing powers generates all nonzero elements of a given field is called a **primitive element**. Note, that for all primitive polynomials, x is a primitive element. All primitive polynomials are irreducible. However, not all irreducible polynomials are primitive. This means that by choosing an arbitrary irreducible polynomial we may risk that we cannot generate all field elements as the element x is raised to some power j. An example of an irreducible polynomial which is not primitive is $p(x) = x^4 + x^3 + x^2 + x + 1$, where $P_{max} = 2^4 - 1 = 15$:

$$x^0 \bmod p(x) = 1. \tag{2.17}$$

$$x^1 \bmod p(x) = x. \tag{2.18}$$

$$x^2 \bmod p(x) = x^2. \tag{2.19}$$

$$x^3 \bmod p(x) = x^3. \tag{2.20}$$

$$x^4 \bmod p(x) = x^3 + x^2 + x + 1. \tag{2.21}$$

$$x^5 \bmod p(x) = 1. \tag{2.22}$$

$$x^6 \bmod p(x) = x. \tag{2.23}$$

$$\vdots$$

In this case, $p(x)$ does not generate the P_{max} unique elements and is therefore not primitive. For more information on this topic see [2, pp. 25–54].

We will refer to the x^j as the **power representation of the corresponding polynomial element**. The advantage of primitive polynomials and elements is that using the power representation, the product or division of two field elements is reduced to summing or subtracting the exponent values modulo $2^m - 1$, where m is the degree of the primitive polynomial. This may in many cases be computed faster than a direct multiplication or division of two binary polynomials. This is shown for multiplication in the following example, using $p(x) = x^3 + x + 1$ from above. Note that the degree here is $m = 3$, so $2^m - 1 = 7$. Consider

$$a(x) = (x^2 + 1) \cdot (x^2 + x + 1) \bmod p(x).$$

Using the mapping from Eq. (2.14) and (2.13):

$$a(x) = x^6 \cdot x^5 \bmod p(x)$$
$$a(x) = x^{(6+5 \bmod 7)} \bmod p(x)$$
$$a(x) = x^4 \bmod p(x).$$

Mapping back to polynomial representation using Eq. (2.12):

$$a(x) = x^2 + x.$$

Algorithms using finite field arithmetic can, in certain cases, increase performance by utilizing lookup tables with the mapping between the polynomial representation and the power representation.

2.9 Polynomials in Delay – Data Streams

We have seen polynomials used in extension fields. There is another way to use polynomials. With the finite fields we have presented, whether it be a prime field or an extension field, we were manipulating scalars. The only tricky part was to define the operations of those scalars in a way that made sense.

In NC, we may want to code together streams of data by **combining across time**. For example, we may want to sum together data in a stream with data in the same stream but with a delay of 1, 3, or 5 time units. The coding operations will themselves take place over a finite field, say prime field or extension field, such as we defined before. We see that this is a way also of extending the number of possibilities for our operations. The cardinality is now unbounded – we can have a system that will, theoretically, reach infinitely far into the past (we generally consider causal systems, so we do not combine with the future).

In order to indicate that we delay by k time-steps, we shall use the notation D^k, where D can be thought of as delay. This notation is borrowed from circuit design, where a single shift of a shift register is denoted by a multiplication by D. Thus, a single shift (delay) in time is a multiplication in D. Delaying by two is applying a delay twice, so multiplying by D twice, hence is multiplying by D^2, and so on.

The data will itself be represented as a polynomial, with the coefficient of degree k representing the data k time steps ago.

So, we would represent a string of data (where time is moving backwards) $\underline{x} = (x_0, x_1, x_2, \ldots)$ as $x(D) = x_0 + x_1 \cdot D + x_3 \cdot D^2 + \cdots$. This notion may be new or may have appeared to readers with different backgrounds under different guises. Electrical engineers who have done circuit design may have seen this delay notation. In signal processing, the delay notion is inherently important and may have been seen as the z-transform. For people with a background in probability, the z-transform may have been seen as the moment generating function for a discrete probability mass function. Depending on how you may have been introduced to the z-transform, you may have seen the role of D as being that of z or z^{-1}.

We can create now what is called a **polynomial ring** that we denote $\mathbb{F}_p(D)$, where we assume a prime field, but we could expand this to an extension field. If you have a background in coding, you may recognize this as the structure of a convolutional code (there the D becomes an x). We can multiply together polynomials in the usual way, and the arithmetic used to compute the coefficients of the polynomial is done in $\mathbb{F}_p$. Note that this is different from an extension field as we are not taking modulo a polynomial.

For example, suppose that, in binary, we sum data with a delay of 1, 3, or 5 time units, as mentioned at the start of this section. We would then have that the result of applying that code to the data stream $x(D)$ would be:

$$x(D) \cdot (D + D^3 + D^5).$$

In this case, we consider the ring $\mathbb{F}_2(D)$.

2.10 Solutions

The completion of the table in Section 2.6.3 is as follows.

Polynomial	Binary	Integer
$x^7 + x^6 + x^4 + x + 1$	11010011	211
$x^7 + x^6 + x^3 + 1$	11001001	201
$x^7 + x^2 + x$	10000110	133
$x^4 + x^2 + x$	00010110	22
$x^4 + x^3 + 1$	00011001	25
x^3	00001000	8

Example additions/subtractions:

$$\begin{aligned}
(x^7 + x^6 + x^2) + (x^7 + x^5 + x^3 + x^2) &= (1+1)x^7 + x^6 + x^5 \\
&\quad + x^3 + (1+1)x^2 + (1+1)x^2 \\
&= x^6 + x^5 + x^3. \\
(x^5 + x) + (x^3 + x^2) &= x^5 + x^3 + x^2 + x^1. \\
(x^7 + x^3) + (x^7 + x + 1) &= (1+1)x^7 + x^3 + x + 1 \\
&= x^3 + x + 1. \\
(x^3 + x^2 + x + 1) + (x + 1) &= x^3 + x^2 + (1+1)x + (1+1)1 \\
&= x^3 + x^2. \\
(x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + 1) + (x^4 + x^2 + 1) &= x^7 + x^6 + x^5 \\
&\quad + (1+1)x^4 + x^3 + (1+1)x^2 + (1+1)1 \\
&= x^7 + x^6 + x^5 + x^3.
\end{aligned}$$

Example multiplications

$$\begin{aligned}
(x^5 + x)(x^3 + x^2) &= x^8 + x^7 + x^4 + x^3. \\
(x^7 + x^3)(x^7 + x + 1) &= x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3. \\
(x^3 + x^2 + x + 1)(x+1)x^4 + (1+1)x^3 + (1+1)x^2 + (1+1)x + 1 &= x^4 + 1.
\end{aligned}$$

2.11 Summary

Finite field arithmetic underlies network coding. As a practical guide to network coding, we provide an essential, engineering-centric study on this topic in Chapter 2. Focusing on division in finite fields and the concept of multiplicative inverse, we strive to maintain a balance between the mathematics underpinning them and how they are used in the context of network coding. We cover prime fields, extension fields, and binary extension fields to make this learning process easy to follow. If some of the concepts in this chapter are familiar to you, we expect them to serve as a refresher for everyone and provide insights into their application to network coding. By now, you should be able to:

(A) Perform arithmetic operations in finite fields.
(B) Find multiplicative inverses and perform division.
(C) Understand the use of primitive polynomials.

Additional Reading Materials

For Linear algebra, we recommend [3]. For the connection between finite fields and error correction, [4] is a great resource.

References

1 D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.
2 S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*. Pearson/Prentice Hall, 2004.
3 M. Artin, *Algebra*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
4 E. R. Berlekamp, *Algebraic Coding Theory (Revised Edition)*. World Scientific, 2015.

3

Finite Field Implementations for Network Coding*

Creating efficient finite field implementations has been an active research topic for several decades. Many applications in areas such as cryptography, signal processing, erasure coding, and now also network coding depend on this research to deliver satisfactory performance. The efficient implementation of the arithmetic operations in software are complicated for a number of reasons, for example:

- As shown in Chapter 2, all arithmetic operations must be performed modulo some prime p or irreducible polynomial $p(x)$. For integers and the polynomial coefficients, not in the binary field, the modulo operation is typically performed using an integer division which has a particular high latency on most Central Processing Units (CPUs). Furthermore for polynomials, in all fields, the modulo operation must be performed after multiplication and division (where division is typically implemented using multiplication with the inverse element) to ensure that the resulting polynomials degree is within the field.
- For prime fields and extension fields where the binary representation of the field elements does not fit the typically available data types, e.g. char, short, int, etc., fully utilizing the underlying hardware is typically hard, as most CPUs are designed for $8, 16, 32$, or 64 bit data. Therefore, in practice, operating on data which is not byte-aligned requires bit-masking and bit-shifting, making operations slower.

In this chapter, we present a selected number of methods and algorithms that have been proposed for creating efficient finite field implementations. Besides the ones covered here, a large amount of additional algorithms exist in current literature. Many of these are optimized for specific use cases, e.g. within the field of cryptography and thus requiring extremely large field elements (of the order of hundreds of bits per element), and are therefore not directly useful in Network Coding (NC) applications, where lower field sizes typically are sufficient [1]. Therefore, this chapter attempts to cover the ones most widely proposed and used for NC applications. Armed with the description of the algorithms given here, it is possible to leverage public libraries, e.g. [2], to implement them.

As we will see finite fields may be implemented in a number of different ways, and in general one cannot point out a single superior approach covering all possible use cases. Different applications often have different requirements, and for NC some of the factors typically impacting the choice of implementation are:

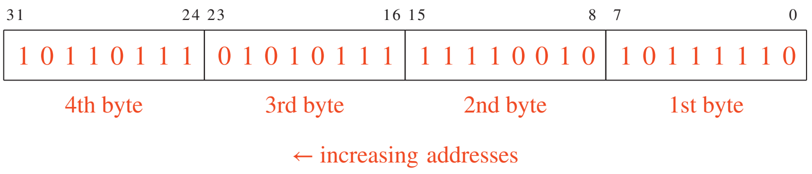
Field size: Depending on the network topology we may require a finite field of a certain size, see [1] and [3] for more information.

Memory consumption: On memory-constrained devices, algorithms with a limited memory consumption may be a requirement.

Speed: We may wish to optimize the algorithms for speed, in which case we simply want an as fast as possible implementation.

In addition to specific algorithmic optimizations several researchers also deal with platform-specific optimizations such as parallelism on multicore architectures or utilizing the computational capabilities of Graphics Processing Units (GPUs). An overview of this type of optimizations with a focus on NC applications is given in [4].

The algorithms covered here fall in the three categories depicted in Fig. 3.1. To avoid misunderstandings and make the descriptions clearer, we will use the following terms and definitions: We assume that an implementation platform has a W-bit architecture, where W is a multiple of 8 (today, typically architectures are 8, 16, 32, and 64 bit). The bits of a W-bit word U are numbered from 0 to $W - 1$, with the rightmost bit of U denoted bit 0. We will assume Little-endian byte order when addressing multi-byte data, i.e. least significant byte first, as shown in the following example for $W = 32$:



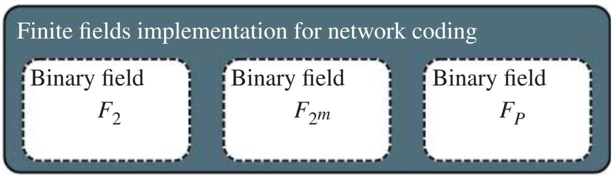


Figure 3.1 Finite fields commonly used in network coding implementations.

The following symbols will be used to denote different operations on words U and V:

$U \wedge V$	bitwise exclusive or (XOR)
$U \,\&\, V$	bitwise and (AND)
$U \gg i$	Right shift U by i positions, with the i high-order bits set to 0
$U \ll i$	Left shift U by i positions, with the i low-order bits set to 0
$U \,\%\, V$	Compute the modulo between U and V

3.1 Binary Field Implementations

The binary field $\mathbb{F}_2$, consisting of the two elements $\{0, 1\}$, has been widely used, not only in network coding implementations but also in various other fountain code-based systems [5, 6]. The use of the binary field is in many cases advantageous since it allows computationally efficient implementations, though there are other drawbacks. When working in the binary field we may utilize the underlying hardware perfectly by operating on vectors of field elements at a time. As an example, we may consider an 8 bit data type (e.g. an unsigned char) as a vector of eight field elements, i.e. with each bit representing an element with value 0 or 1. Pairwise addition of two vectors of eight elements can then be implemented using a single XOR operation of two unsigned char variables. Recall from Section 2.4 that all operations in $\mathbb{F}_2$ may be implemented either using the bitwise XOR or AND operators. These operations may typically be performed using a single fast CPU instruction yielding fast implementations. Network coding algorithms using this field can be found in [7].

Listing 3.1 Addition and subtraction in $\mathbb{F}_2$

```python
def binary_add_subtract(a,b):
    """
    Add or subtracts two vectors of 1 bit field
    elements packed in two W-bit words.

    a, the first field element
    b, the second field element
    """
    return a ^ b
```

Note that we did not show an algorithm for division here. The reason is that division is only defined for nonzero elements in any field, i.e. division by zero is not allowed. Since we are working in the binary field, the only nonzero element is 1, where $1/1 = 1$.

Listing 3.2 Multiplication in $\mathbb{F}_2$

```
def binary_multiply(a,b):
    """

    Multiplies two vectors of 1 bit field
    elements packed in two W-bit words.

    a, the first field element
    b, the second field element
    """
    return a & b
```

As mentioned, to better match the supported operations of the underlying hardware when working in $\mathbb{F}_2$, we typically do calculations on vectors of multiple 1 bit elements. As an example, if using a $W = 32$ bit architecture using a 32 bit data type would allow adding, subtraction, or multiplying up to 32 field elements at a time using a single $\oplus$ or & operation. The following examples show these operations for a number of $W = 8$ bit vectors:

- **Addition**: $11011001 \oplus 01011011 = 10000010$.
- **Subtraction**: $01001101 \oplus 00011001 = 01010100$.
- **Multiplication**: $11001011 \,\&\, 01001010 = 01001010$.

We have omitted division for the reasons explained in Section 3.1.

3.2 Binary Extension Field Implementations

In NC the binary extension field is one of the most commonly used. This is mainly driven by the fact that the density of invertible sets of linear equations increases with field size and, indeed, higher field sizes are required to get the full benefit of NC in some network topologies. Higher order fields, particularly those of modest order, which are sufficient for practical purposes, still allow efficient implementation.

As we will see in what follows, working in an extension field requires more complex algorithms for multiplication and division operations, when compared to the basic binary field. This added complexity is mainly introduced by the polynomial representation used to create the extension field. To combat that, the common strategy is to use lookup tables to fetch precomputed results of the division or multiplication operations. While in most cases that approach can outperform algorithms calculating the results "on the fly," its performance becomes largely platform dependent, e.g. on CPU cache sizes, memory prefetching, etc. [8]. Furthermore on some resource-constrained devices, such as sensor boards, using several kilo Bytes (kB) for a lookup table may be an unacceptable requirement.

In the following we will first provide algorithms for directly computing the result of the different arithmetic operations, we will refer to these as “runtime” algorithms. Following this, we will present a number of methods for implementing multiplication and division using precomputed lookup tables, we will refer to these as “off-the-shelf” algorithms.

3.2.1 Runtime Addition and Subtraction

As previously shown in Section 2.6.2, addition and subtraction using the polynomial representation is done element-wise for the coefficients of the polynomials, and since the coefficients in the binary extension field are members of the binary field $\mathbb{F}_2$, this means that addition and subtraction may be implemented using the XOR operation.

Listing 3.3 Addition and subtraction in $\mathbb{F}_{2^m}$

```
def binary_extension_add_subtract(a, b):
    """
    Add or subtracts two W-bit field
    elements.

    a, the first field element
    b, the second field element
    """
    return a ^ b
```

As the degree of the resulting polynomial $c(x)$ cannot exceed the degree of the chosen prime polynomial, no further computations are needed.

As an example, consider a $W = 8$ bit architecture with the two polynomials $a(x) = x^7 + x^6 + x^2$ and $b(x) = x^7 + x^5 + x^3 + x^2$ with the binary representation of 11000100 and 10101100, respectively.

- **Addition or subtraction:** $11000100 \oplus 10101100 = 01101000$.

The result may be confirmed by adding the two polynomials directly:

$$\begin{aligned} c(x) &= (x^7 + x^6 + x^2) + (x^7 + x^5 + x^3 + x^2) \\ &= (1 \oplus 1)x^7 + x^6 + x^5 + x^3 + (1 \oplus 1)x^2 \\ &= x^6 + x^5 + x^3, \end{aligned}$$

where $x^6 + x^5 + x^3$ has the anticipated binary representation 01101000.

3.2.2 Runtime Multiplication and Division

Compared to the straightforward implementation of addition and subtraction, multiplication and division require substantially more computation, as shown in the following algorithm. This also illustrates why using precomputed lookup tables can increase performance.

The central idea behind the following algorithm is to perform multiplication one factor at a time, while ensuring that the resulting polynomial in each step has a degree $< m$. As an example, given the two polynomials $a(x) = x^3 + x + 1$, $b(x) = x^2 + 1$ and an irreducible polynomial $p(x) = x^5 + x^2 + 1$, we want to compute the result $c(x) = a(x) \cdot b(x)$:

$$\begin{aligned} c(x) &= (x^3 + x + 1) \cdot (x^2 + 1) \\ &= x^5 + (1 \oplus 1)x^3 + x^2 + x + 1 \\ &= x^5 + x^2 + x + 1. \end{aligned} \tag{3.1}$$

As the degree of $c(x)$ equals m, we perform the modulo operation using the irreducible polynomial $p(x)$

$$\begin{aligned} c(x) &= x^5 + x^2 + x + 1 \bmod p(x) \\ c(x) &= x \bmod p(x). \end{aligned} \tag{3.2}$$

To implement this in practice we need the following two operations.

Multiplication with *x*: We perform the multiplication of the two polynomials using three iterations of the algorithm equivalent to the following:

$$c(x) = \underbrace{(x^3 + x + 1) \cdot (x^2 \cdot 1)}_{iteration 3} + \underbrace{(x^3 + x + 1) \cdot (x^1 \cdot 0)}_{iteration 2} + \underbrace{(x^3 + x + 1) \cdot (x^0 \cdot 1)}_{iteration 1}.$$

A key part of understanding the algorithm is to notice that in each iteration we multiply $a(x)$ with an increasing power of x from $b(x)$. However, as shown in "iteration 2" we only use the result if the corresponding coefficient $b(x)$ is nonzero. Using this approach we perform the multiplication by iterating through the binary representation of $b(x)$, while in every step multiplying $a(x)$ with x and if necessary reducing the result modulo $p(x)$. Where the multiplication with x is equivalent to a left bit-shift. For example, taking $a(x)$ from the example we see that:

$$\underbrace{(x^3 + x + 1)}_{00001011} \rightarrow \underbrace{(x^3 + x + 1) \cdot x}_{00001011 \ll 1} \rightarrow \underbrace{x^4 + x^2 + x}_{00010110}.$$

Multiplying $a(x)$ with x^2 is equivalent to performing two bit-shifts and so forth.

Modulo reduction with *p*(*x*): During each iteration of the algorithm we have to ensure that the degree of the resulting polynomial stays below m. In the cases where when the degree of the result polynomial reaches m, we must perform the modulo reduction with the irreducible polynomial $p(x)$. In the example, this occurs after "iteration 3." From the example, we have the desired result:

$$c(x) = x^5 + x^2 + x + 1 \bmod x^5 + x^2 + 1 = x.$$

As both polynomials in this case have degree m, the modulo operation may be performed using a single subtraction. We know from Section 3.2.1 that the subtraction in the binary extension field may be implemented using an XOR between the binary representation of the two polynomials:

$$00100111 \oplus 00100101 = 00000010.$$

The above result may also be verified using polynomial long division, and is equivalent to:

$$\begin{array}{r|rrrrl}
 & & & & 1 & \triangleleft \text{ quotient} \\ \cline{2-5}
x^5 + x^2 + 1 & x^5 & +x^2 & +x & +1 & \\
 & x^5 & +x^2 & & +1 & \\ \cline{2-5}
 & & & x & & \triangleleft \text{ remainder} \\ \cline{4-5}
\end{array}$$

Where we can verify that the remainder x indeed has the binary representation 00000010.

The following example[1] shows steps of the algorithm for the input $a = x^2 + x + 1$ and $b = x + 1$:

Listing 3.4 Runtime multiplication algorithm for $\mathbb{F}_{2^m}$

```
def online_simple_multiply(a, b, p, m):
    """
    Performs the simple online multiply algorithm in the
    2^m extension field

    a, the first polynomial
    b, the second polynomial
    p, the irreducible polynomial
    m, the degree of the irreducible polynomial
    """

    if a == 0 or b == 0:
```

1 This implementation may not be suitable for use in certain cryptographic schemes as we stop the for loop as soon as possible, which may make the code vulnerable to timing attacks.

```
        return 0

    # Mask to check if the degree is about to reach m
    mask = 1 << (m−1)
    highbit = 0

    # The resulting polynomial
    c = 0

    for i in range(m):

        if b == 0:
            break

        if b & 1:
            c = c ˆ a

        highbit = a & mask

        a = a << 1
        b = b << 1

        if highbit:
            a = a ˆ p

    return c
```

The following shows the iterations of the algorithm using the example given in Eq. (3.1), where $a(x) = x^3 + x + 1$, $b(x) = x^2 + 1$, and $p(x) = x^5 + x^2 + 1$. The input to the algorithm is the decimal representation of the polynomials as shown in Table 2.3.

Initialization:

$$a = 11 \qquad b = 5 \qquad c = 0 \qquad \text{highbit} = 0$$
$$p = 37 \qquad m = 5 \qquad \text{mask} = 16$$

Iteration 1:

$$a = 22 \qquad b = 2 \qquad c = 11 \qquad \text{highbit} = 0$$
$$p = 37 \qquad m = 5 \qquad \text{mask} = 16$$

Iteration 2:

$$a = 9 \qquad b = 1 \qquad c = 11 \qquad \text{highbit} = 1$$
$$p = 37 \qquad m = 5 \qquad \text{mask} = 16$$

Iteration 3:

$$\begin{array}{cccc} a = 18 & b = 0 & c = 2 & \text{highbit} = 0 \\ p = 37 & m = 5 & \text{mask} = 16 & \end{array}$$

In "iteration 3" we hit the terminating condition b equal 0. We see that the resulting polynomial is $c = 2$, which corresponds to 00000010 in binary or $c(x) = x$ as a polynomial and matches our calculations in Eq. (3.2).

3.2.3 Runtime Division Algorithm

We now define the algorithm used for division, to do this we have to recall the discussion presented in Section 2.3. It points to the fact that division may be implemented in terms of multiplication with the inverse element, i.e.

$$c(x) = \frac{a(x)}{b(x)}$$

will be calculated as:

$$c(x) = a(x) \cdot b(x)^{-1}.$$

As we already have defined an algorithm for multiplication, we only need to find an algorithm to determine the inverse of an element in the binary extension field. A common way to implement this is to use the Pulverizer, that is the extended Euclidean algorithm, but with polynomials. The Pulverizer calculates the Greatest Common Divisor (GCD) of two polynomials, but also finds two additional polynomials $u(x)$ and $v(x)$ such that:

$$\gcd(a(x), b(x)) = a(x) \cdot u(x) + b(x) \cdot v(x).$$

Using this equation, we can find the GCD of two arbitrary polynomials. However, if we choose the field's irreducible polynomial as one of the input polynomials to the algorithm, we get the following result:

$$\begin{aligned} \gcd &= a(x) \cdot u(x) + p(x) \cdot v(x) = 1 \bmod p(x) \\ \gcd &= a(x) \cdot u(x) = 1 \bmod p(x). \end{aligned}$$

As $p(x)$ is an irreducible polynomial, the GCD will always be 1. Furthermore, as we are working $\bmod p(x)$ any multiple of $p(x)$ will be zero. We therefore see that $u(x)$ will represent the inverse of $a(x)$.

The following algorithm calculates the inverse polynomial by utilizing the fact that the GCD between two polynomials does not change when subtracting one from the other. This algorithm, and the extended Euclidean algorithm applied to polynomials from which it is derived, is described in more detail in Section 3.3.

Listing 3.5 Runtime inverse algorithm for $\mathbb{F}_{2^m}$

```
def online_simple_inverse(a, p):
    """
    Finds the inverse of a polynomial a(x) in
    the 2^m extension field

    a, the polynomial whos inverse we want
    p, the irreducible polynomial
    """
    if a == 1:
        return 1

    r_large = p
    r_small = a

    y_large = 0
    y_small = 1

    j = 0
    while(r_large != 1):

        j = find_degree(r_large) – find_degree(r_small)

        if j < 0:
            r_large, r_small = r_small, r_large
            y_large, y_small = y_small, y_large

            j = abs(j)

        r_large = r_large ^ (r_small << j)
        y_large = y_large ^ (y_small << j)

    return y_large
```

Note that the algorithm requires the degree of the resulting polynomials to be known. A suitable function for that purpose may be derived by iterating over the binary representation of each polynomial. Code for such a function may be found in Section 3.3.1.

The division algorithm may now be implemented in terms of the multiplication and inverse algorithms as shown in the following:

Listing 3.6 Runtime divide algorithm for $\mathbb{F}_{2^m}$

```
def online_simple_divide(a, b, p, m):
    """
    Divides the two input polynomials and find the resulting
    polynomial in the 2^m extension field

    a, the numerator polynomial
    b, the denominator polynomial
    p, the prime polynomial
    m, the degree of the prime polynomial
    """
    inverse = online_simple_inverse(b,p)
    return online_simple_multiply(inverse, a, p, m)
```

As the division algorithm is simply composed using the multiplication and inverse algorithm, we will only show an example of the inverse algorithm here. In the following iterations, we find the inverse of $a(x) = x^3 + x + 1$ using the irreducible polynomial $p(x) = x^5 + x^2 + 1$. As with the multiplication, the input to the algorithm is specified using the decimal representation of the polynomials:

Initialization:

$$\begin{aligned} \text{r\_large} &= 37 & \text{y\_large} &= 0 & a &= 11 & p &= 37 \\ \text{r\_small} &= 11 & \text{y\_small} &= 1 & p &= 37 \end{aligned}$$

Iteration 1:

$$\begin{aligned} \text{r\_large} &= 9 & \text{y\_large} &= 4 & a &= 11 & p &= 37 \\ \text{r\_small} &= 11 & \text{y\_small} &= 1 & p &= 37 \end{aligned}$$

Iteration 2:

$$\begin{aligned} \text{r\_large} &= 2 & \text{y\_large} &= 5 & a &= 11 & p &= 37 \\ \text{r\_small} &= 11 & \text{y\_small} &= 1 & p &= 37 \end{aligned}$$

Iteration 3:

$$\begin{aligned} \text{r\_large} &= 3 & \text{y\_large} &= 21 & a &= 11 & p &= 37 \\ \text{r\_small} &= 2 & \text{y\_small} &= 5 & p &= 37 \end{aligned}$$

Iteration 4:

$$\begin{aligned} \text{r\_large} &= 1 & \text{y\_large} &= 16 & a &= 11 & p &= 37 \\ \text{r\_small} &= 2 & \text{y\_small} &= 5 & p &= 37 \end{aligned}$$

In "iteration 4" we hit the terminating condition r_large = 1, in this case, the algorithm will have calculated the inverse polynomial in the variable y_large, which in this case contains the decimal value 16 that represents the polynomial $u(x) = x^4$. We may now check that this indeed is the inverse of $a(x)$, such that

$$a(x) \cdot u(x) = 1 \bmod p(x).$$

We check by inserting the two polynomials,

$$(x^3 + x + 1) \cdot (x^4) = 1 \bmod p(x)$$

$$x^7 + x^5 + x^4 = 1 \bmod p(x) \tag{3.3}$$

$$1 = 1. \tag{3.4}$$

The steps given in Eqs. (3.3) and (3.4) are calculated by reducing the polynomial $x^7 + x^5 + x^4$ by the irreducible polynomial $p(x) = x^5 + x^2 + 1$. We may verify this using polynomial long division,

$$\begin{array}{r|cccccl}
 & & & & x^2 & +1 & \triangleleft \text{ quotient} \\
\cline{2-6}
x^5 + x^2 + 1 & x^7 & +x^5 & +x^4 & & & \\
 & x^7 & & +x^4 & +x^2 & & \\
\cline{2-5}
 & & x^5 & & +x^2 & & \\
 & & x^5 & & +x^2 & +1 & \\
\cline{3-6}
 & & & & & 1 & \triangleleft \text{ remainder} \\
\cline{6-6}
\end{array}$$

3.2.4 Full Multiplication Table

As seen in section 3.2.2, the multiplication of two elements may require a significant number of operations. To avoid this we may instead precompute the results and at runtime use a lookup table to find the results. In this way, we may reduce the number of operations needed per multiplication at the cost of higher memory consumption. The lookup table may be computed using the iterative multiplication and division algorithms given in the previous section.

Listing 3.7 Creates a full multiplication and division table for $\mathbb{F}_{2^m}$

```
def create_full_table(p,m):
    """
    Precomputes and creates two lookup table for multiplying
    and dividing elements in a given 2^m binary extension field.

    p, the irreducible polynomial used
    m, the degree of the irreducible polynomial
    """

    # The number of elements in the field
    order = 1 << m

    # Allocate the two tables
    multtable = [0]*order*order
    divitable = [0]*order*order

    for i in range(order):
        offset = i * order

        for j in range(order):
            multtable[offset+j] = online_simple_multiply(i, j, p, m)

            if j == 0: # Cannot divide by zero
                continue

            divitable[offset+j] = online_simple_divide(i, j, p, m)


    return (multtable, divitable)
```

Multiplying or dividing two field elements may now be evaluated using a lookup into either the multiplication or division table, as shown in Listing 3.8.

Listing 3.8 Example lookup function for the full multiplication or division tables created in $\mathbb{F}_{2^m}$

```
def multdiv_full_table(a,b,t,m):
    """
    Multiplies/divides two binary extension field elements using the
    precompute lookup table.
    For each element there are 2^m results, this is used to
    index into the table

    a, the first polynomial
    b, the second polynomial
    t, the precomputed lookup table
    m, the degree of the irreducible polynomial
    """
    return t[(a << m) + b]
```

Memory consumption: As seen, the full multiplication and division tables have a highly attractive complexity, requiring only a single bit-shift, addition, and memory lookup to perform either a multiplication or division. However, the low complexity comes with a higher memory consumption. With the field $\mathbb{F}_{2^m}$, the table will contain $2^m \cdot 2^m = 2^{2m}$ elements, where each element consumes m bits storage. Computing the total storage for both a multiplication and division table we get $2^{2m+1}m/8$ bytes.

3.2.5 Log and AntiLog Table

One way to reduce the memory footprint of the full table-based multiplication and division is to construct the lookup table utilizing the power representation of the finite field elements introduced in Section 2.8. This type of implementation is called the Log/AntiLog method in literature. In the following example, we will produce such a lookup table for primitive polynomial $p(x) = x^3 + x + 1$.

From Section 2.8 we have the following possible representations for the field elements:

This basic idea is to create a mapping from the binary or integer representation to the exponent in the power representation and from the exponent in the power representation back. In this example $\mathbb{F}_{2^3}$, these are shown in Table 3.2. Using Table 3.2, we see that the Log table maps between the decimal value of an element and its corresponding exponent in the power representation, such that

$$\text{Log}[5] = 6,$$

which we can verify using Table 3.1 which shows that the decimal value 5 has the power representation x^6. Correspondingly, the AntiLog table maps from the exponent value in the power representation to the decimal representation,

$$\text{AntiLog}[6] = 5.$$

This may now be availed of by using the values in Table 3.2, such that multiplication and division becomes:

$$\begin{aligned} 5 * 3 &= \text{AntiLog}[\text{Log}[5] + \text{Log}[3] \bmod 7] \\ &= \text{AntiLog}[6 + 3 \bmod 7] \\ &= \text{AntiLog}[2] = 4 \end{aligned}$$

Table 3.1 Different representations of the elements of the finite field $\mathbb{F}_{2^3}$.

Power	Polynomial	Binary	Decimal
0	0	000	0
x^0	1	001	1
x^1	x	010	2
x^2	x^2	100	4
x^3	$x+1$	011	3
x^4	x^2+x	110	6
x^5	x^2+x+1	111	7
x^6	x^2+1	101	5

Table 3.2 Logarithmic tables for the finite field $\mathbb{F}_{2^3}$.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Log	—	0	1	3	2	6	4	5
AntiLog	1	2	4	3	6	7	5	—

$$\begin{aligned}
2 * 4 &= \text{AntiLog}[\text{Log}[2] + \text{Log}[4] \bmod 7] \\
&= \text{AntiLog}[1 + 2 \bmod 7] \\
&= \text{AntiLog}[3] = 3 \\
6/3 &= \text{AntiLog}[\text{Log}[6] - \text{Log}[3] \bmod 7] \\
&= \text{AntiLog}[4 - 3 \bmod 7] \\
&= \text{AntiLog}[1] = 2 \\
2/7 &= \text{AntiLog}[\text{Log}[2] - \text{Log}[7] \bmod 7] \\
&= \text{AntiLog}[1 - 5 \bmod 7] \\
&= \text{AntiLog}[3] = 3.
\end{aligned}$$

As shown, we can find the power representation for the field elements by calculating the j^{th} power of the primitive element x. In the following code listing, we construct the Log and AntiLog tables using the runtime multiplication algorithm to repeatedly multiply x by itself and find the corresponding field elements.

Listing 3.9 Creates the Log and AntiLog tables for $\mathbb{F}_{2^m}$

```
def create_log_table(p,m):
    """
    Creates the Log and AntiLog tables mapping between power
    representation and decimal/binary representation in a
    given 2^m binary extension field.

    p, the primitive polynomial used
    m, the degree of the primitive polynomial
    """

    # The number of elements in the field
    order = 1 << m

    # Allocate the two tables
    log = [None]*order
    antilog = [None]*order

    # Initial value corresponds to x^0
    power = 1

    for i in range(order−1):

        log[power] = i
        antilog[i] = power

        # 2 is the decimal representation of the polynomial
        # element 'x'
        power = online_simple_multiply(2, power, p, m)

    return (log, antilog)
```

For multiplication we can use the Log and AntiLog tables as follows:

Listing 3.10 Multiplication using the Log and AntiLog tables for $\mathbb{F}_{2^m}$

```
def multiply_log_table(a,b,log,antilog,m):
    """
    Multiplies two field elements using the
    precompute lookup table.

    a, the first field element
    b, the second field element
    log, the precomputed Log lookup table
    antilog, the precomputed AntiLog lookup table
    m, the degree of the irreducible polynomial
    """
    if a == 0 or b == 0:
        return 0

    # Calculate the exponent sum modulo the field
    # order − 1
    exp_sum = (log[a] + log[b]) % (2^m − 1)

    return antilog[exp_sum]
```

We have also seen that division using the Log and AntiLog tables is similar to multiplication, but uses subtraction of exponent values instead of addition:

Listing 3.11 Division using the Log and AntiLog tables for $\mathbb{F}_{2^m}$

```
def divide_log_table(a,b,log,antilog,m):
    """
    Divides two field elements using the
    precompute lookup table.

    a, the numerator field element
    b, the denomiator field element
    log, the precomputed Log lookup table
    antilog, the precomputed AntiLog lookup table
    m, the degree of the irreducible polynomial
    """
    # No division by zero
    assert(b > 0)

    if a == 0:
        return 0

    # Calculate the exponent sum modulo the field
    # order - 1
    exp_sum = (log[a] - log[b]) % (2^m - 1)

    return antilog[exp_sum]
```

Removing modulo operator: A performance bottleneck in the presented algorithm is the use of the modulo operator to reduce the exponent sum. Since the modulo operator typically requires significantly more CPU cycles to perform than addition and subtraction, performance can be improved by avoiding its use. Here we assume that the implementation uses a finite precision-type system with C/C++ compatible unsigned integer overflow rules. We first note that if the exponent sum overflows, we can detect it as follows. The exponent sum of two W-bit unsigned integers is bound by $s = a + b < 2^{W+1}$, in case of an overflow we have $2^W \leq a + b < 2^{W+1}$ in which case the leading bit in the $W + 1$ bit representation of the sum equals 1, since this high bit cannot be represented in the W bit representation IT, will be discarded. Discarding the high means that $s < a$ and $s < b$, which can easily be checked.

After detecting the overflow we have to ensure that the modulo reduction is performed correctly. We do this by noticing that if an overflow occurs, this is equivalent to subtracting 2^W from the sum. However, since we work $\mod (2^W - 1)$, we have to compensate by adding 1 to the resulting sum. If no overflow occurs, we do not have to perform any operations.

Memory consumption: Both the Log and AntiLog tables will contain a single entry for each element in the used field. In general a $\mathbb{F}_{2^m}$ field has 2^m elements of m bits. Thus the total memory consumption for both the Log and AntiLog tables becomes $2^{m+1}m/8$ bytes.

3.2.6 Extended Log and AntiLog Table

In the previous section, we showed how to construct the Log and AntiLog tables and also discussed how the modulo operator could be avoided to increase performance. A different approach to handle overflows in the exponent sum is to extend the tables to handle this situation. The following optimization is based on the fact that the maximum exponent is given as $e_{max} = 2^m - 2$ and the minimum exponent is given as $e_{min} = 0$. Thus, we have the maximum exponent sum as:

$$sum_{max} = (2^m - 2) + (2^m - 2) = 2^{m+1} - 4.$$

Likewise the minimum exponent sum can be calculated as:

$$sum_{min} = 0 - (2^m - 2) = 2 - 2^m.$$

Accepting increased memory consumption, we may extend the AntiLog table to handle all possible sums. The following AntiLog table extends the example given in Table 3.2 for $p(x) = x^3 + x + 1$:

To understand the mapping recall that we are computing the sum modulo $2^m - 1$. In the example this yields:

$$-6 \bmod 7 = 1 \tag{3.5}$$

$$-5 \bmod 7 = 2 \tag{3.6}$$

$$\vdots \tag{3.7}$$

$$11 \bmod 7 = 4 \tag{3.8}$$

$$12 \bmod 7 = 5. \tag{3.9}$$

From Eq. (3.5) we see that index −6 should map to the same element as index 1, Eq. (3.6) shows index −5 maps to the same element as index 2, and so forth. We can verify these results in Table 3.3 by checking that e.g. index −6 and 1 both map to the same element. We may now rewrite the examples from Section 3.2.5 without the mod operator:

$$\begin{aligned} 5 * 3 &= \text{AntiLog}[\text{Log}[5] + \text{Log}[3]] \\ &= \text{AntiLog}[6 + 3] \\ &= \text{AntiLog}[9] = 4. \end{aligned}$$

$$\begin{aligned} 2 * 4 &= \text{AntiLog}[\text{Log}[2] + \text{Log}[4]] \\ &= \text{AntiLog}[1 + 2] \\ &= \text{AntiLog}[3] = 3. \end{aligned}$$

Table 3.3 Extended AntiLog tables for the finite field $\mathbb{F}_{2^3}$.

Index	−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6	7	8	9	10	11	12
AntiLog	2	4	3	6	7	5	1	2	4	3	6	7	5	1	2	4	3	6	7

$$\begin{aligned}6/3 &= \text{AntiLog}[\text{Log}[6] - \text{Log}[3]]\\ &= \text{AntiLog}[4 - 3]\\ &= \text{AntiLog}[1] = 2.\end{aligned}$$

$$\begin{aligned}2/7 &= \text{AntiLog}[\text{Log}[2] - \text{Log}[7]]\\ &= \text{AntiLog}[1 - 5]\\ &= \text{AntiLog}[-4] = 3.\end{aligned}$$

The following function shows how the extended Log and AntiLog tables may be created:

Listing 3.12 Creates the extended Log and AntiLog tables for $\mathbb{F}_{2^m}$

```
def create_extended_log_table(p,m):
    """
    Creates the extended Log and AntiLog tables mapping between power
    representation and decimal/binary representation in a
    given 2^m binary extension field.

    p, the primitive polynomial used
    m, the degree of the primitive polynomial
    """

    # The number of elements in the field
    order = 1 << m

    # Allocate the two tables
    log = [None] * order
    antilog = [None] * (3 * order - 5)

    # Initial value corresponds to x^0
    power = 1

    # Array offset
    low_offset = 0
    mid_offset = low_offset + order - 2
    high_offset = mid_offset + order - 1

    for i in range(order-1):

        log[power] = i
        antilog[mid_offset + i] = power

        # 2 is the decimal representation of the polynomial
        # element 'x'
        power = online_simple_multiply(2, power, p, m)

    # Fill the extended table based on the previously computed
    # results
    for i in range(order−2):
        antilog[low_offset + i] = antilog[mid_offset + i + 1]
        antilog[high_offset + i] = antilog[mid_offset + i]

    return (log, antilog)
```

Memory consumption: The Log table size has not changed, i.e. it contains 2^m elements of m bits. However, extending the AntiLog table to avoid the modulo operation has increased the number of elements to:

$$\begin{aligned}\text{AntiLog}_{elements} &= \underbrace{(2^m - 2)}_{negative\ sums} + \underbrace{(2^{m+1} - 4)}_{positive\ sums} + \underbrace{(1)}_{zero\ sum} \\ &= (3 \cdot 2^m) - 5.\end{aligned}$$

Thus the total memory consumption for both the extended Log and AntiLog tables becomes:

$$(2^m + 3 \cdot 2^m - 5) \cdot \frac{m}{8} = (2^{m+2} - 5) \cdot \frac{m}{8} \text{ bytes},$$

which is still a significant reduction in memory consumption when compared to the full lookup tables presented in Section 3.2.4.

Exercise 3.1 Performance Comparison of Basic Arithmetic Operations Using Runtime Algorithms and Lookup Tables.

In this exercise, you will compare the performance of addition, subtraction, multiplication, and division operations using runtime algorithms and lookup tables. The goal is to measure the speed of each operation with random input values and calculate the throughput in MB/s for each operation/algorithm pair.

1) **Measure execution time:**
   - Implement timing mechanisms to measure the execution time of each arithmetic operation using both runtime algorithms and lookup tables.
   - Execute the test cases multiple times to obtain reliable average execution times.
2) **Calculate throughput:**
   - Determine the size of the input values (e.g. number of bits) for each arithmetic operation.

- Calculate the throughput (operations per second) for each operation–algorithm pair.
- Convert the throughput to MB/s for easier comparison.

3) **Compare performance:**
   - Compare the throughput of addition, subtraction, multiplication, and division operations using runtime algorithms and lookup tables.
   - Analyze the results to identify the most efficient combination of operation and algorithm for each arithmetic operation.

Optional:

- Investigate the use of SIMD (Single Instruction, Multiple Data) instructions or GPU acceleration to further optimize performance.

Exercise 3.2 Measuring Encoding and Decoding Speed

In this exercise, you will utilize PyErasure to evaluate the time difference between encoding and decoding operations. The goal is to measure the speed of these processes using different finite field sizes and explore the impact of varying the number of symbols in the encoder and decoder.

1) Implement a test using PyErasure to measure the time difference between encoding and decoding for a fixed number of symbols (e.g. 32). Calculate the speed difference in MB/s.
2) Extend the test by varying the finite field sizes and observe how the performance changes. Document the results and analyze the relationship between finite field size and encoding/decoding speed.
3) Further extend the test to vary the number of symbols in both the encoder and decoder. Plot how the performance depends on the number of symbols.
4) Come back after Chapter 4 to experiment with different scenarios, such as systematic encoding, and assess their impact on encoding and decoding speed.

Optional:

- Study ways to optimize encoding and decoding speed, considering parameters like CPU utilization, parallelization, and memory usage.

3.3 Extended Euclidean Algorithm**

In this section, we describe the extended Euclidean algorithm that we introduced in Section 2.6.2, which can be used to find the inverse of integers and polynomials defined over some field $\mathbb{F}_{p^m}$. As explained earlier in Chapter 2, the extended Euclidean algorithm, i.e. the Pulveriser, is based on the classical Euclidean algorithm for finding the GCD of two numbers. The classical Euclidean algorithm works by using the principle that given two numbers a and b, where $a \leq b$ subtracting a from b does not change the GCD. This yields the following identity:

$$\gcd(a, b) = \gcd(b - qa, a), \tag{3.10}$$

which, as explained in Section 2.2, gives

$$\gcd(a, b) = \gcd(b \bmod a, a). \tag{3.11}$$

This leads to the following interactive algorithm where we find the quotient q for positive integers a and b where $a \leq b$. Here b is divided by a to obtain a quotient q and a remainder r satisfying $r = b - qa$ and $0 \leq r \leq a$. From Eq. (3.10) we know that $\gcd(a, b) = \gcd(r, a)$, the problem of finding the $\gcd(a, b)$ has now been reduced to computing $\gcd(r, a)$ where $(r, a) < (a, b)$; this process now continues until the remainder is 0, which immediately yields the result $\gcd(0, d)$, where d is the GCD. The following example shows the process:

$$\begin{aligned}\gcd(4, 27) &= \gcd(27 - 6 \cdot 4, 4) = \gcd(3, 4)\\ &= \gcd(4 - 1 \cdot 3, 3) = \gcd(1, 3)\\ &= \gcd(3 - 3 \cdot 1, 1) = \gcd(0, 1)\\ &= 1.\end{aligned}$$

The final nonzero remainder is the gcd in this case 1.

The Euclidean algorithm outlined may now be extended to find the integers x, y such that $ax + by = d$, where $d = \gcd(a, b)$. The extended Euclidean algorithm, i.e. the Pulverizer, can be implemented in a number of different ways. In the following, we will outline an algorithm based on the Table Based Method (also known as a “magic box”). The algorithm is based on the observation that the remainders computed in the standard GCD algorithm can be expressed as linear combinations of the original integers a and b:

$$r_i = ax_i + by_i. \tag{3.12}$$

So, in general, we may say:

$$r_{i-1} = ax_{i-1} + by_{i-1}. \tag{3.13}$$
$$r_{i-2} = ax_{i-2} + by_{i-2}. \tag{3.14}$$
$$\vdots \tag{3.15}$$

From Eq. (3.11) we see that the remainder r_i can be calculated as:

$$r_i = r_{i-2} \bmod r_{i-1} = r_{i-2} - \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor \cdot r_{i-1}. \tag{3.16}$$

Substituting Eqs. (3.13) and (3.14) into Eq. (3.16) yields:

$$r_i = ax_{i-2} + by_{i-2} - \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor \cdot (ax_{i-1} + by_{i-1}),$$

which may be rewritten as:

$$\begin{aligned} r_i &= ax_{i-2} + by_{i-2} - ax_{i-1} \cdot \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor - by_{i-1} \cdot \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor \\ &= a\left(x_{i-2} - x_{i-1} \cdot \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor\right) + b\left(y_{i-2} - y_{i-1} \cdot \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor\right). \end{aligned} \tag{3.17}$$

Comparing Eq. (3.18) to Eq. (3.12) we see that:

$$x_i = x_{i-2} - x_{i-1} \cdot \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor \tag{3.18}$$

$$y_i = y_{i-2} - y_{i-1} \cdot \left\lfloor \frac{r_{i-2}}{r_{i-1}} \right\rfloor. \tag{3.19}$$

After finding these identities we can determine the coefficients x_i and y_i for any given remainder r_i. We may also note that for every remainder we must know the results of the previous two steps, i.e. to compute r_i for a given input a and b we need to first determine $\{x_{i-2}, x_{i-1}, y_{i-1}, y_{i-2}, r_{i-1}, r_{i-2}\}$. We first initialize the algorithm by calculating r_0 and r_1 by first setting $x_0 = 0, x_1 = 1, y_0 = 1, y_1 = 0$ for these values and solve for r_i using Eq. (3.12).

$$r_0 = ax_0 + by_0 = a \cdot 0 + b \cdot 1 = b.$$

$$r_1 = ax_1 + by_1 = a \cdot 1 + b \cdot 0 = a.$$

We continue calculating $r_2 = ax_2 + by_2$ using the formulas given in Eqs. (3.18) and (3.19). This process continues until the remainder $r_i = 0$ in which case we have found:

$$r_{i-1} = ax_{i-1} + by_{i-1} = \gcd(a, b).$$

Algorithm 3.1 outlines the pseudo-code for the extended Euclidean algorithm for integers. This algorithm uses the extended Euclidean procedure which we have outlined earlier, starting with Eq. (3.12).

The following iterations show the algorithm for the integers $a = 3$ and $b = 11$.

- Initialization

$$\begin{array}{lllll} a = 3 & b = 11 & x = 0 & y = 0 & q = 0 \\ r = 0 & x_0 = 1 & x_1 = 0 & y_0 = 0 & y_1 = 1 \end{array}$$

Algorithm 3.1 Extended Euclidean Algorithm for integers in $\mathbb{Z}$, where $a \leq b$.

```
1: procedure EXTENDED EUCLIDEAN (a,b)
2:     $x_0 = 0, x_1 = 1$
3:     $y_0 = 1, y_1 = 0$
4:     $r_0 = b, r_1 = a$
5:     while $r_1 \neq 0$ do
6:         $q = \left\lfloor \frac{r_0}{r_1} \right\rfloor$
7:         $r = r_0 - r_1 \cdot q$
8:         $x = x_0 - x_1 \cdot q$
9:         $y = y_0 - y_1 \cdot q$
10:        $x_1, x_0 \Leftrightarrow x, x_1$
11:        $y_1, y_0 \Leftrightarrow y, y_1$
12:        $r_1, r_0 \Leftrightarrow r, r_1$
13:    end while
14: return $r_0, x_0$ and $y_0$
15: end procedure
```

- Iteration 1

$$a = 2 \quad b = 3 \quad x = -3 \quad y = 1 \quad q = 3$$
$$r = 2 \quad x_0 = -3 \quad x_1 = 1 \quad y_0 = 1 \quad y_1 = 0$$

- Iteration 2

$$a = 1 \quad b = 2 \quad x = 4 \quad y = -1 \quad q = 1$$
$$r = 1 \quad x_0 = 4 \quad x_1 = -3 \quad y_0 = -1 \quad y_1 = 1$$

- Iteration 3

$$a = 0 \quad b = 1 \quad x = -11 \quad y = 3 \quad q = 2$$
$$r = 0 \quad x_0 = -11 \quad x_1 = 4 \quad y_0 = 3 \quad y_1 = -1$$

After terminating the algorithm we may verify the result, i.e. $ax + by = d = \gcd(a, b)$, where we have $a = 3$, $b = 11$, $x = 4$, $y = -1$, and $\gcd(a, b) = d = 1 = (3 \cdot 4 + 11 \cdot -1)$.

In the following, we will use this result to find the inverse of a number in a prime field $\mathbb{F}_p$. To see how this may be achieved we use the following observations. First note that if p is prime, then $\gcd(a, p) = 1$. This means that for the extended Euclidean algorithm, we have $ax + py = 1$. However, as we are in a finite field we should reduce the coefficients modulo p. This yields $ax + yp = 1 \bmod p$, which is $ax = 1 \bmod p$ since the modulo of $y \cdot p \bmod p$ always will be zero. From this, we see that x must be the inverse of a, as the definition of inverse states that $a \cdot a^{-1} = 1$. Based on these results we see that the following optimizations may be made to Algorithm 3.1, so that we may use it to find the inverse of a.

Algorithm 3.2 Finding inverse integer in $\mathbb{F}_p$.

```
1:  procedure INVERSE (a)
2:      $x_0 = 0, x_1 = 1$
3:      $r_0 = p, r_1 = a$
4:      while $r_0 \neq 1$ do
5:          $q = \left\lfloor \frac{r_0}{r_1} \right\rfloor$
6:          $r = r_0 - r_1 \cdot q$
7:          $x = x_0 - x_1 \cdot q$
8:          $x_1, x_0 \Leftrightarrow x, x_1$
9:          $r_1, r_0 \Leftrightarrow r, r_1$
10:     end while
11: return $x_0 \bmod p$
12: end procedure
```

- We only need to keep track of the x coefficients as the equation we are solving for is essentially $a \cdot x = 1$.
- We can terminate the loop when the remainder equals 1. We may use this as the terminating condition as we know $\gcd(a, p)$ will always equal 1.

Implementing these optimizations, we obtain the following algorithm 3.2 for finding the inverse of an integer in $\mathbb{F}_p$.

The following iterations show the algorithm for the integers $a = 4$ and $p = 7$.

- Initialization

$$a = 4 \qquad b = 7 \qquad x = 0 \qquad q = 0$$
$$r = 0 \qquad x_0 = 1 \qquad x_1 = 0$$

- Iteration 1

$$a = 3 \qquad b = 4 \qquad x = -1 \qquad q = 1$$
$$r = 3 \qquad x_0 = -1 \qquad x_1 = 1$$

- Iteration 2

$$a = 1 \qquad b = 3 \qquad x = 2 \qquad q = 1$$
$$r = 1 \qquad x_0 = 2 \qquad x_1 = -1$$

After reaching the terminating condition, i.e. $a = 1$ we may check that the x found is indeed the inverse of a:

$$a \cdot x \bmod p = 1$$
$$4 \cdot 2 \bmod 7 = 1.$$

The final part of this section describes how the above results may be used in an extension field, i.e. a finite field $\mathbb{F}_{p^m}$ using polynomial representation for the field elements. It turns out that the same formulas apply for polynomials as for integers. Using polynomial long division to find the quotient and remainder in Algorithm 3.1 and otherwise performing the same routine would yield the correct answer.

That is, for two binary polynomials $a(x)$ and $b(x)$ we may find a GCD, for which the following holds

$$\gcd(a(x), b(x)) = \gcd(b(x) - c(x) \cdot a(x), a(x)), \tag{3.20}$$

for any binary polynomial $c(x)$. As with the integers, Eq. (3.20) states that subtracting a multiple of $a(x)$ from $b(x)$ does not change gcd. As for the integers, we may extend this result so that using the extended Euclidean algorithm we may find an $z(x)$ and $w(x)$ such that

$$a(x) \cdot z(x) + b(x) \cdot w(x) = d(x) = \gcd(a(x), b(x)).$$

Algorithms for both the Euclidean and extended Euclidean algorithms for polynomials are given in [9, p. 82–83]. However, in these algorithms, we must perform a polynomial long division to obtain the quotient and remainder (denoted q and r in the extended Euclidean algorithm for integers), which in practice is not easily implemented in software. Instead, we may use the alternative extended Euclidean algorithm presented here [10, p. 57–58]. The algorithm utilizes the fact that the division may be replaced with a number of subtractions. Recall that the $\gcd(a, b) = \gcd(b, b - ca)$. In this algorithm, only one step of each polynomial long division is performed for each iteration of the algorithm. Essentially this means that in each iteration of the algorithm, we perform the long division using a single multiplication followed by a subtraction to cancel the highest degree polynomial term. As an example, consider the two polynomials $g(x) = x^4 + x^2 + 1$ and $f(x) = x^3 + 1$. To cancel the term x^4 we first multiply $f(x)$ with x and subtract from $g(x)$, i.e.:

$$\begin{aligned} r(x) &= g(x) + f(x) \cdot x \\ &= (x^4 + x^2 + 1) + (x^3 + 1) \cdot x \\ &= (1 + 1) \cdot x^4 + x^2 + x + 1 \\ &= x^2 + x + 1. \end{aligned} \tag{3.21}$$

Note that in the binary field addition and subtraction are identical, hence the use of $+$. Furthermore, recall that in the binary field $1 + 1 = 0$ which cancels x^4 in Eq. (3.21). We may now continue *gcd* algorithm by reducing $f(x)$ by $r(x)$:

$$\begin{aligned} h(x) &= f(x) + r(x) \cdot x \\ &= (x^3 + 1) + (x^2 + x + 1) \cdot x \\ &= (1 + 1) \cdot x^3 + x^2 + x + 1 \\ &= x^2 + x + 1. \end{aligned}$$

$$\begin{aligned}
k(x) &= r(x) + h(x) \\
&= (x^2 + x + 1) + (x^2 + x + 1) \\
&= (1 + 1) \cdot x^2 + (1 + 1) \cdot x + (1 + 1) \\
&= 0.
\end{aligned}$$

As the last remainder is zero, the algorithm ends with $h(x) = x^2 + x + 1$ as the GCD of $g(x)$ and $f(x)$. We may backtrack the steps in this algorithm to find the coefficients of the extended Euclidean algorithm (starting from $h(x)$ which is our gcd).

$$\begin{aligned}
h(x) &= f(x) + r(x) \cdot x \\
&= f(x) + (g(x) + f(x) \cdot x) \cdot x \\
&= f(x) + g(x) \cdot x + f(x) \cdot x^2 \\
&= (1 + x^2) \cdot f(x) + x \cdot g(x).
\end{aligned}$$

Using the outlined gcd approach for polynomials, we may use the following algorithm 3.3:

Algorithm 3.3 Extended Euclidean Algorithm for polynomials in $\mathbb{F}_{2^p}$, where *deg*$(a) \leq$ *deg*(b).

```
1: procedure EXTENDED EUCLIDEAN (a,b)
2:     $g_0 = 1, g_1 = 0$
3:     $h_0 = 0, h_1 = 1$
4:     $u = a, v = b$
5:     while u $\neq 0$ do
6:         $j = \deg(u) - \deg(v)$
7:         if $j < 0$ then
8:             $u \leftrightarrow v$
9:             $g_0 \leftrightarrow g_1$
10:            $h_0 \leftrightarrow h_1$
11:            $j = -j$
12:        end if
13:        $u = u + v \cdot x^j$
14:        $g_0 = g_0 + g_1 \cdot x^j$
15:        $h_0 = h_0 + h_1 \cdot x^j$
16:    end while
17: return $v$, $g_1$ and $h_1$
18: end procedure
```

As with the integers, we may modify the extended Euclidean algorithm to find the inverse of any given polynomial $\bmod p(x)$ as shown in Algorithm 3.4.

Algorithm 3.4 Inverse Algorithm for polynomials in $\mathbb{F}_{2^p}$.

```
1:  procedure INVERSE (a)
2:      $g_0 = 1, g_1 = 0$
3:      $u = a, v = p$
4:      while u $\neq$ 1 do
5:          $j = \deg(u) - \deg(v)$
6:          if $j < 0$ then
7:              $a \leftrightarrow b$
8:              $g_0 \leftrightarrow g_1$
9:              $j = -j$
10:         end if
11:         $u = u + v \cdot x^j$
12:         $g_0 = g_0 + g_1 \cdot x^j$
13:     end while
14: return $g_0$
15: end procedure
```

3.3.1 Find Degree Function

Certain algorithms require that the degree of the polynomials be determined. The following code listing gives an example implementation of such a function.

Listing 3.13 Function to determine the degree of arbitrary polynomial in $\mathbb{F}_{2^m}$

```
def find_degree(a):
    """
    Finds returns the degree of the polynomial
    i.e. the number 5 = 101 = Xˆ2 + 1 has degree 2

    a, the polynomial whos degree we wish to find
    """
    degree = 0

    a = a >> 1

    while(a > 0):
        degree = degree + 1
        a = a >> 1

    return degree
```

3.4 Summary

In the third chapter, we present a number of different finite field implementations. The rationale is that which implementation has the best performance for a specific application can vary. The chapter aims not only to present the optimizations suitable for NC operations but also to provide a pool of algorithms for the implementations. Using those algorithms, the implementer has a wide variety of choices encompassing different field sizes, memory consumption, and computational complexity. The code listings presented in this chapter demonstrate the different algorithms implemented in the popular Python programming language. These algorithms form a solid starting point for further experimentation with finite fields. Highly efficient implementations of the algorithms in C++ can be accessed after obtaining a license from Steinwurf ApS. This chapter enables you to:

A) Understand algorithms for finite field implementations.
B) Perform finite field arithmetic using different techniques.
C) Familiarize yourself with different types of lookup tables and use them.

Additional Reading Materials

For information about modulo arithmetic implementations in GPUs and FPGAs, [11] provides a solid overview.

References

1 O. Geil, R. Matsumoto, and C. Thomsen, "On field size and success probability in network coding," in *Arithmetic of Finite Fields*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2008, vol. 5130, pp. 157–173.

2 M. Hostetter, "Galois: A performant NumPy extension for Galois fields," 2020. [Online]. Available: https://github.com/mhostetter/galois.

3 T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong, "A random linear network coding approach to multicast," *IEEE Transactions on Information Theory*, vol. 52, no. 10, pp. 4413–4430, 2006.

4 J. Heide, M. V. Pedersen, F. H. Fitzek, and T. Larsen, "Network coding in the real world," in *Network Coding: Fundamentals and Applications*. Elsevier, 2012, pp. 87–114.

5 M. Luby, "LT codes," in *Proceedings of the 43rd Symposium on Foundations of Computer Science*, 2002, pp. 271–280.

6 A. Shokrollahi, "Raptor codes," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

7 J. Heide, M. V. Pedersen, F. H. Fitzek, and T. Larsen, "Network coding for mobile devices-systematic binary random rateless codes," in *IEEE International Conference on Communications*, 2009.

8 K. M. Greenan, E. L. Miller, and T. J. E. Schwarz, "Optimizing Galois field arithmetic for diverse processor architectures and applications," in *IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, 2008, pp. 257–266.

9 A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1996.

10 D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

11 J.-P. Deschamps, J. L. Imana, and G. D. Sutter, *Hardware Implementation of Finite-field Arithmetic*. McGraw-Hill Education, 2009.

4

Coding for Erasures

Communication is often discussed in the context of data units, often called **packets**. Packets traversing a network can be lost for various factors, including physical impairments of the transmission medium, contention, and overflow caused by congestion. Receivers can generally detect these erasures reliably. This is often done by running a verification mechanism such as Cyclic Redundancy Check (CRC) codes, which use the polynomials in delay that we discussed in Section 2.9 but for a distinct purpose.

Networks have extensive mechanisms to ensure that packets are received uncorrupted, and in order, from senders to their intended receivers. We shall explore some such mechanisms, network protocols, in Chapter 5. In order to design protocols, it is essential to first understand how coding manages losses. If we do not code, there is a limited design space for such protocols, even though there is a rich and varied literature in the domain. When we consider coding, our design space becomes larger and far more powerful, but it also means that we must envisage new possibilities and, in particular, how to determine what codes are fit for purpose. Recall that solutions that do not use coding effectively use a highly degenerate form of coding.

By now, we have also understood that data is mutable and we can take advantage of that in a number of contexts. Let's revisit the basics: we can take data as binary, and represent chunks of length n bits and consider them as elements of a finite field where we have two algebraic choices to manage the finiteness:

- We can select the largest prime p such that $2^n < p < 2^{n+1}$, use an integer representation and field operations can be done with modulo p integer arithmetic, a **prime field**.
- We can work directly in an **extension field** by representing the data as polynomials and defining operations modulo an irreducible polynomial.

We shall see that the larger the field, the larger the density of matrices that are invertible, but for practical NC purposes fields of size $2^8 = 256$ or larger suffice.

Recall that we can use all of the algorithms that we are used to from working in $\mathbb{R}$, but replace division with multiplication by an inverse.

4.1 From Chunks to Packets

So far, we have assumed we have been given "chunks" of data $X_{(1)}, X_{(2)}, \dots$ that we wish to communicate where each $X_{(i)} \in \mathbb{F}_q$ for some q. When we consider data networks, we will largely work in the extension field $\mathbb{F}_{2^n}$ for some n. In general, we have in mind that $n = 8$ or 16 or 32 bits, i.e. 1 to 4 bytes, while, for example, in Ethernet, the maximum transmission unit is 1500 bytes.

We do not want to treat 1500 bytes as a single chunk, which would require a finite field of size at least $2^{8\times 1500}$, which is as unnecessary as it is unmanageable. Instead, each packet can be divided into a sequential series of chunks, with each chunk being considered an element of a smaller field and performing operations across packets on these small chunks. That is, a chunk is defined as a small portion of the packet that can be represented within the extension field or prime field that will be used for coding. The data in each packet is, therefore, composed of a series of chunks.

Thus when we code K packets together, we do so with a single encoding matrix on a chunk-by-chunk basis:

$$\begin{aligned}
\text{Packet 1} &= (X_{(1)}^{(1)}, X_{(1)}^{(2)}, \dots) \\
\text{Packet 2} &= (X_{(2)}^{(1)}, X_{(2)}^{(2)}, \dots) \\
&\ \ \vdots \qquad \vdots \\
\text{Packet K} &= (X_{(K)}^{(1)}, X_{(K)}^{(2)}, \dots),
\end{aligned}$$

and use only a single coding matrix to encode each chunk across the packets. Thus the coded packets consist of a series of coded chunks and the receiver need only be aware of a single coding matrix that's reused by all chunks. This mechanism for coding packets by considering them as a sequence of chunks is shown in Fig. 4.1.

As a toy example, consider the case where we have two packets, each consisting of 8 bits to send,

$$\begin{aligned}
\mathbf{p}_1 &= 11001011, \\
\mathbf{p}_2 &= 00011010,
\end{aligned}$$

that we plan to encode into the prime field with 7 elements, $\mathbb{F}_7$, using the encoding matrix

$$A = \begin{pmatrix} 2 & 2 \\ 5 & 1 \end{pmatrix}$$

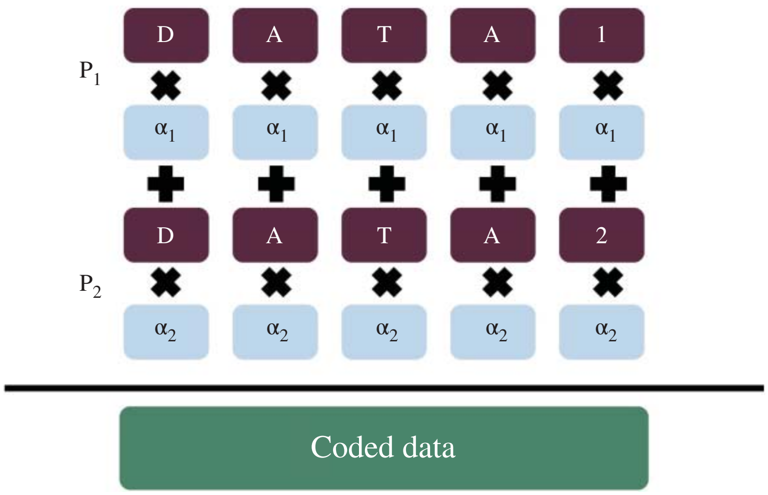


Figure 4.1 Mechanism for coding together packets. This figure is inspired by joint work with Daniel Lucani.

which has inverse

$$A^{-1} = \begin{pmatrix} 6 & 2 \\ 5 & 5 \end{pmatrix}.$$

To encode, we group the bits into pairs, convert them to integers, multiply each pair of integers by A, and convert back to binary where we now need 3 bits for each coded symbol rather than 2 as our coded data is in $\mathbb{F}_7$.

$$\text{encoding with } A \;\Bigg\downarrow\quad
\begin{array}{rcccc}
\mathbf{p}_1 = & \multicolumn{4}{c}{11001011} \\
\mathbf{p}_2 = & \multicolumn{4}{c}{00011010} \\
\hline
\mathbf{p}_1 = & 11 & 00 & 10 & 11 \\
\mathbf{p}_2 = & 00 & 01 & 10 & 10 \\
\hline
\mathbf{p}_1 = & 3 & 0 & 2 & 3 \\
\mathbf{p}_2 = & 0 & 1 & 2 & 2 \\
\hline
2\cdot\mathbf{p}_1 + 2\cdot\mathbf{p}_2 = & 6 & 2 & 1 & 3 \\
5\cdot\mathbf{p}_1 + 1\cdot\mathbf{p}_2 = & 1 & 1 & 5 & 3 \\
\hline
2\cdot\mathbf{p}_1 + 2\cdot\mathbf{p}_2 = & 110 & 010 & 001 & 011 \\
5\cdot\mathbf{p}_1 + 1\cdot\mathbf{p}_2 = & 001 & 001 & 101 & 011 \\
\hline
2\cdot\mathbf{p}_1 + 2\cdot\mathbf{p}_2 = & \multicolumn{4}{c}{110010001011} \\
5\cdot\mathbf{p}_1 + 1\cdot\mathbf{p}_2 = & \multicolumn{4}{c}{001001101011}
\end{array}
\quad\Bigg\uparrow\; \text{decoding with } A^{-1}$$

To decode, we follow the reverse process, using A^{-1} rather than A. Note that if an extension field was used, the coded packets would be the same size as the original packets.

In practice, if a prime field was used, a much larger prime than 7 would be employed. To encode each set of 7 bits into 8, one would use $p = 251$, the largest prime smaller than 2^8, while to encode each set of 15 bits into 16 one would use the prime $65,521$. This also means that depending on the number of bits and corresponding prime, different proportions of possibilities will not be used. For example, with 8 bits and $p = 251$, 5 out of 256 possibilities are not used while with 16 bit and $p = 65,521$, only 15 out 65, 536 are not used. Figure 4.2 plots the proportion of possibilities that would not be used in the prime field employed to represent chunks of n bits as $n + 1$ bits.

In summary, when we code together two source packets $\mathbf{p}_1$ and $\mathbf{p}_2$ over the finite field with q elements, $\mathbb{F}_q$, then we are creating a new coded packet $\alpha_1 \cdot \mathbf{p}_1 + \alpha_2 \cdot \mathbf{p}_2$ such that its k^{th} chunk is the sum in $\mathbb{F}_q$ of the product of α_1 by the k^{th} chunk of $\mathbf{p}_1$ in its representation in $\mathbb{F}_q$ and the product of α_2 by the k^{th} chunk of $\mathbf{p}_2$ in its representation in $\mathbb{F}_q$.

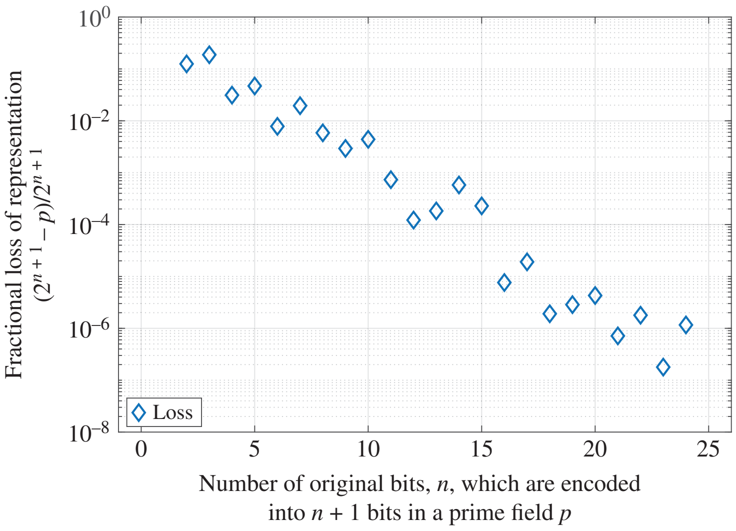


Figure 4.2 To encode n bits in a prime field, one needs to represent them in $n + 1$ bits. As implied by the Bertrand–Chebyshev Theorem, $2^n < p < 2^{n+1}$, some of the binary strings of length $n + 1$ go unused. Plotted is that fraction, $(2^{n+1} - p)/2^{n+1}$, for the largest prime p smaller than 2^{n+1}.

4.2 Erasure Resilience

To produce a code that is resilient to erasures, instead of sending packets consisting of uncoded chunks, $(X_{(1)}, \ldots, X_{(K)})$, the sender can transmit N linear combinations of source packets considered as integers or polynomials, which we call coded-packets:

$$A\begin{pmatrix} X_{(1)} \\ \vdots \\ X_{(K)} \end{pmatrix} = \begin{pmatrix} \alpha_{(1,1)} & \alpha_{(1,2)} & \cdots & \alpha_{(1,K)} \\ \alpha_{(2,1)} & \alpha_{(2,2)} & \cdots & \alpha_{(2,K)} \\ \vdots & \vdots & \cdots & \vdots \\ \alpha_{(N,1)} & \alpha_{(N,2)} & \cdots & \alpha_{(N,K)} \end{pmatrix} \begin{pmatrix} X_{(1)} \\ \vdots \\ X_{(K)} \end{pmatrix} = \begin{pmatrix} X_{(1)}' \\ \vdots \\ X_{(N)}' \end{pmatrix},$$

where the entries $\alpha_{(i,j)}$ of the matrix A are either elements of $\mathbb{F}_p$ or $\mathbb{F}_{2^n}$ depending on which finite field context the sender and receiver have chosen. If for K original packets we send N coded packets, we say that the rate of the code is $R = K/N$.

As long as at least K of the $\left(X_{(1)}', \ldots, X_{(N)}'\right)$ are received such that there exists a $K \times K$ matrix consisting of the corresponding rows of A that is invertible, the receiver can perform **Gaussian elimination**, i.e. multiplication by the inverse of the coding matrix, in the finite field to recover the original data $\left(X_{(1)}, \ldots, X_{(K)}\right)$. To that end, next we consider the determination of the inverse of the coding matrix.

4.3 Gauss–Jordan Elimination

Gauss–Jordan Elimination is an algorithm that solves systems of linear equations and can find the inverse of any invertible matrix. It relies upon three elementary row operations one can use on a matrix:

1) Swap the positions of two rows.
2) Multiply one of the rows by a nonzero value.
3) Add a multiple of one row to another row.

Working in a finite field is no different from the more familiar real numbers, but, of course, we perform the operations within the field. The purpose of Gauss–Jordan Elimination is to use the three elementary row operations to convert a matrix into **reduced-row echelon form (RREF)**. A matrix is in RREF if the following conditions are satisfied:

1) All rows with only zero entries are at the bottom of the matrix.
2) The first nonzero entry in a row, called the leading entry or the pivot, is to the right of the leading entry of the row above it.

3) The leading entry, also known as the pivot, in any nonzero row is 1.
4) All other entries in the column containing a leading 1 are 0.

For example, A and B are in RREF but C breaks the second and third conditions and D breaks the fourth condition.

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{pmatrix}, C = \begin{pmatrix} 0 & 3 & 3 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, D = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

An algorithmic procedure to perform Gauss–Jordan Elimination is as follows:

1) Swap the rows so that all rows with all zero entries are on the bottom.
2) Swap the rows so that the row with the largest, leftmost nonzero entry is on top.
3) Multiply the top row by a value such that top row's leading entry becomes 1.
4) Add multiples of the top row to the other rows so that all other entries in the column containing the top row's leading entry are all 0.
5) Repeat steps $2-4$ for the next leftmost nonzero entry until all the leading entries are 1.
6) Swap the rows so that the leading entry of each nonzero row is to the right of the leading entry of the row above it.

To obtain the inverse of a $K \times K$ matrix A that is invertible:

1) Create the partitioned matrix $(A|I)$, where I is the $K \times K$ identity matrix.
2) Perform Gauss–Jordan Elimination on the partitioned matrix with the objective of converting the first part of the matrix to reduced-row echelon form.
3) The resulting partitioned matrix will take the form $(I|A^{-1})$.

Let us consider an example of Gauss–Jordan elimination in the **reals** to determine an inverse matrix, which proceeds by taking the matrix to be inverted

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

and appending the identity matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

One performs linear operations on the matrix of interest to convert it to the identity and executes those operations on the appended matrix. When the initial matrix is finally the identity, the appended matrix is now its inverse.

We need not consider steps 1 through 3 as the leading entry of the first column is already 1, but consider the following example

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}\begin{bmatrix}1 & 0\\0 & 1[-3]01\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 2\\0 & -2\end{bmatrix}\begin{bmatrix}1 & 0\\-3 & 11\times -\frac{1}{2}\end{bmatrix}$$

$$\Rightarrow \begin{bmatrix}1 & 2\\0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\3/2 & -1/2[-2]10\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}-2 & 1\\3/2 & -1/2\end{bmatrix}$$

To check that the second matrix is, indeed, the inverse, compute

$$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}\begin{pmatrix}-2 & 1\\3/2 & -1/2\end{pmatrix} = \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}$$

and so

$$A^{-1} = \begin{pmatrix}-2 & 1\\3/2 & -1/2\end{pmatrix}$$

in the reals.

In a **finite field**, we follow the same algorithm, but with the new definition of addition and multiplication. For example, consider $\mathbb{F}_7 = \{0, 1, 2, 3, 4, 5, 6\}$ equipped with modulo 7 arithmetic so that we have the following table of inverses

a	1	2	3	4	5	6
a^{-1}	1	4	5	2	3	6

Let us proceed as before

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}\begin{bmatrix}1 & 0\\0 & 1[-3\cdot = 4\cdot]01\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 2\\0 & 5\end{bmatrix}\begin{bmatrix}1 & 0\\4 & 11\cdot 3\end{bmatrix}$$

$$\Rightarrow \begin{bmatrix}1 & 2\\0 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\5 & 3[-2\cdot = 5\cdot]10\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}5 & 1\\5 & 3\end{bmatrix}$$

Note that in $\mathbb{F}_7$

$$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}\begin{pmatrix}5 & 1\\5 & 3\end{pmatrix} = \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}$$

in the reals and so

$$A^{-1} = \begin{pmatrix}5 & 1\\5 & 3\end{pmatrix}.$$

As illustrated, the usual algorithm for Gauss–Jordan elimination in the reals works unaltered for finite fields.

4.4 Code Construction

When we consider the “network” aspect, we will focus on **Random Linear Network Codes** [1], where the $\alpha_{(i,j)}$ are selected uniformly at random in the appropriate finite field. In doing so, at each hop, we will essentially be including the possibility that our encoding is **any** arbitrary matrix in the finite field, which will be the **relaxation** that we take advantage of. While figuring out how to decode an RLNC, however, we have also figured out how we can decode **any** erasure correcting code, as they all just amount to different structures in the encoding matrix A.

4.4.1 Proportion of Non-invertible Matrices

For a given finite field, a natural question is what proportion of matrices are invertible. In RLNC, where encoding matrices are created uniformly at random, this proportion corresponds to the likelihood that a randomly chosen matrix is invertible and so provides sufficient information to recover the data without the provision of additional coded packets. For “large” fields with q elements, where it will turn out large is quite small, there will be a potentially surprising result: a good approximation is that a proportion $1 - 1/q$ of matrices are invertible irrespective of the value of the number of coded packets, i.e. without a strong K dependence.

To establish this potentially surprising and useful fact, we follow the line of reasoning found in [2]. Note that the first row of an invertible matrix can be any one of the $q^K - 1$ nonzero vectors, i.e. K-tuples. Once it is chosen, row 2 can then be any one of the $q^K - q$ vectors not a multiple (in the finite field) of the first row. Proceeding inductively we see that row $j + 1$ can be chosen to be any of the $q^K - q^j$ vectors not in the (j-dimensional) span of the first j rows. Thus the number of $K \times K$ matrices that are invertible is

$$|\mathrm{GL}(n,q)| = \left(q^K - 1\right) \left(q^K - q\right) \cdots \left(q^K - q^{K-1}\right).$$

The number of all possible matrices is q^{K^2}. Hence the proportion of all matrices that are invertible or, alternatively, the likelihood that a $K \times K$ matrix selected uniformly at random in the finite field with q elements is invertible is given by:

$$\begin{aligned} |\mathrm{GL}(n,q)| / q^{K^2} &= q^{-K}\left(q^K - 1\right) q^{-K}\left(q^K - q\right) \cdots q^{-K}\left(q^K - q^{K-1}\right) \\ &= \left(1 - q^{-K}\right) \left(1 - q^{-K-1}\right) \cdots \left(1 - q^{-1}\right). \end{aligned} \tag{4.1}$$

To a first-order approximation, this is about $1 - 1/q$.

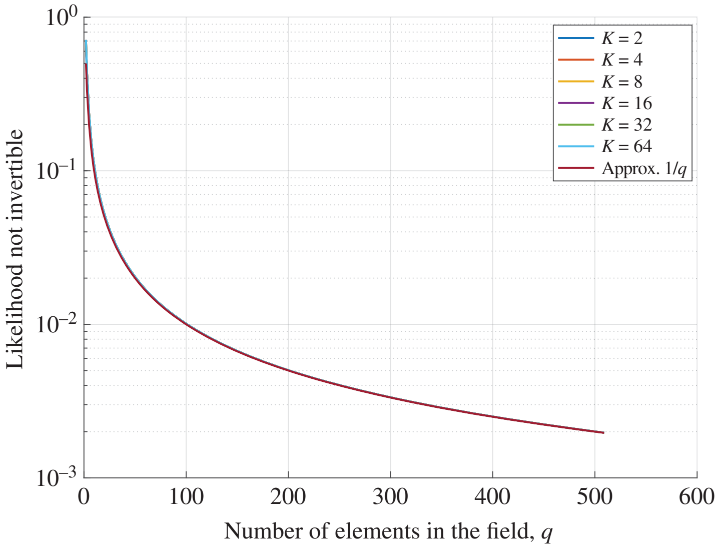


Figure 4.3 Likelihood that a randomly chosen $K \times K$ matrix in a prime-q field is not invertible, along with the approximation $1/q$.

Figure 4.3 plots the results from the formula in Eq. (4.1) for the likelihood that a matrix is **not** invertible as well as the approximation $1/q$. The most remarkable observation is that the likelihood that a randomly selected matrix in the finite field is not invertible is essentially independent of the matrix dimensions and solely depends on the field size.

Note that, the density of non-invertible matrices in a binary extension field is decreasing exponentially in m, the number of bits in the representation, as $q = 2^m$, e.g. Fig. 4.4. With $m = 8$ (i.e. coding on bytes), there is a 4 in $1,000$ chance that a randomly chosen matrix won't be invertible, while with $m = 16$ (i.e. coding on pairs of bytes), this drops to 2 in $100,000$.

Exercise 4.1 Decoding Probability with PyErasure

Use PyErasure to simulate decoding probability across different finite field sizes (e.g. $\mathbb{F}_2$ and $\mathbb{F}_{2^8}$) for a generation size of 32.

- Analyze how the decoding probability changes with varying field sizes, emphasizing the improvement with larger fields.
- Calculate the number of linear dependent packets for each field size and determine the overhead incurred.
- Extend the analysis to generation sizes of 64, 96, 128, and 256 using PyErasure.

(Continued)

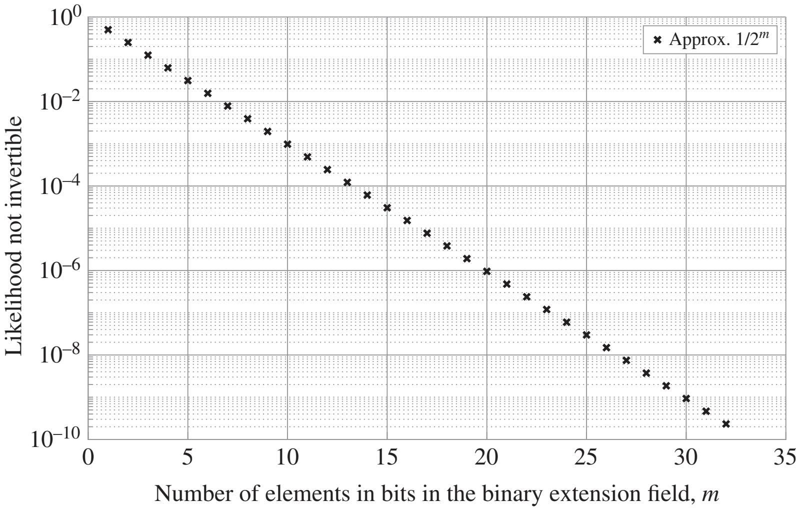


Figure 4.4 For a binary extension field, the likelihood that a randomly chosen $K \times K$ matrix is not invertible decays exponentially in the number of bits.

Exercise 4.1 (Continued)

- Investigate how the overhead varies with different generation sizes, considering the impact of larger generation sizes on decoding probability and overhead.

4.4.2 Non-RLNC Erasure Codes

Non-RLNC erasure codes are typically built in the prime field $\mathbb{F}_2$ rather than in an extension field. As a result, the likelihood that a randomly created encoding matrix is not invertible is no more than $1/2$ or, more accurately, for large K is it approximately 0.29 by Eq. (4.1). Thus codes have to be made in a highly structured way to ensure they are well constructed. By going to a higher field, we can convert this challenging construction problem into a random construction where the likelihood of non-invertibility, approximately $1/q$, is merely determined by the field size.

4.4.3 Simple Scenario to Evaluate Potential Gain

Consider the following situation. A sender wishes to communicate K packets of source information, $(X_{(1)}, X_{(2)}, \dots, X_{(K)})$, to a receiver over a channel where they know each packet will get lost with probability $\epsilon \in (0, 1)$. If the sender wishes to ensure that the receiver gets all the packets with high likelihood, how many copies should they send if they don't code and if they do code?

We can do more sophisticated calculations, but let's consider something simple to get an idea. Cantelli's inequality [3], also known as one-sided Chebyshev, tells us that for any random variable D with finite mean, $E(D)$, and finite variance, $\mathrm{Var}(D)$, then for any $\lambda > 0$ we have that

$$P(D \geq E(D) + \lambda) \leq \frac{\mathrm{Var}(D)}{\mathrm{Var}(D) + \lambda^2}.$$

The main advantage of this inequality is that it only relies on knowing the mean and variance, while the main disadvantage being that it can be a little loose as a result. Regardless, if we wish to find a $\lambda > 0$ such that $E(D) + \lambda$ would ensure $P(D \geq E(D) + \lambda) \leq \tau$ it suffices to have

$$\lambda \geq \sqrt{\mathrm{Var}(D)\frac{(1-\tau)}{\tau}}.$$

The geometric distribution gives the probability that the first occurrence of success requires a independent uses of the channel, each with success probability ϵ,

$$P(D = a) = \epsilon^{a-1}(1-\epsilon),$$

for $a = 1, 2, 3, 4, \ldots$. As D is a geometric distribution, we have

$$E(D) = \frac{1}{1-\epsilon} \text{ and } \mathrm{Var}(D) = \frac{\epsilon}{(1-\epsilon)^2}.$$

Thus if we wish to ensure that by repeatedly transmitting the first packet, we have done so enough times to ensure that with a probability that it hasn't been received is less than τ, we set the number of transmissions to be

$$E(D) + \lambda = \frac{1}{1-\epsilon}\left(1 + \sqrt{\epsilon\frac{(1-\tau)}{\tau}}\right).$$

To send K such packets and wanted to ensure that the likelihood that all K packets go through was greater than $1 - \tau_K$, we would require

$$(1-\tau)^K \geq 1 - \tau_K.$$

If τ is small, then $(1-\tau)^K \approx 1 - K\tau$ and so, under these approximations, we would require that $\tau \leq \tau_K/K$ and the total number of transmissions would need to be

$$\frac{K}{1-\epsilon} + \frac{K}{1-\epsilon}\sqrt{\epsilon\frac{(K-\tau_K)}{\tau_K}},$$

which is super-linear in K because we need to ensure each and every packet gets through and manage the fact that the more trials we do (i.e. packets we send) the more of a margin of error we have to give for each one.

Let us see what an approximation gives if we could send an indefinite number of coded combinations of the K source packets. Note that the number of transmissions until any K coded packets are received is the sum of K independent geometric distributions

$$C = D_1 + \cdots + D_K$$

and so the mean and variance of C are just K times the mean and variance of D_1

$$E(C) = \frac{K}{(1-\epsilon)} \quad \text{and variance} \, \mathrm{Var}(C) = \frac{K\epsilon}{(1-\epsilon)^2}.$$

Cantelli's inequality suggests that we can ensure that we get at least K coded packets with a probability greater than $1 - \tau_K$ if

$$E(C) + \lambda = \frac{K}{(1-\epsilon)} + \frac{\sqrt{K}}{1-\epsilon}\sqrt{\epsilon\frac{(1-\tau_K)}{\tau_K}}$$

coded packets are sent, which has a $\sqrt{K}$ overhead rather than a K one, where the reduction is coming from the opportunism of coded packets. This $\sqrt{K}$ is a central limit theorem effect.

Essentially, if we need every uncoded packet to get through, we need a significant overhead of repetitions for each and every one to ensure all are received with good likelihood. If we can use coded packets, we can make use of the reduction in randomness that comes from repeated trials to have only $\sqrt{K}$ additional transmissions. While a more sophisticated analysis is possible, this sketch provides reason to believe significant gains should be possible by using coded packets.

4.5 Innovative Packet and Acknowledgment

4.5.1 Innovative Packet

An **innovative packet** is a linear combination whose coefficient vector is outside the span of previously received packets' coefficient vectors. When we use RLNC, with high probability, every transmitted packet will have the innovation property, i.e. it will bring new information to every receiver, except in the case when the receiver already knows as much as the sender. Thus, every successful reception will bring a unit of new information [4]. However, the problem of decoding delay remains. Imagine a scheme that would segment the stream into blocks, also called **generations**, and processes one block at a time. If playback can begin only after receiving a full block, then high throughput would require a large delay.

This raises the interesting question – can we use received coded packets even before the full block is received? In general, source packets are not usable until the point up to which all the packets have been recovered in order, which we call the **front of contiguous knowledge**. Delay depends on not just the number of recovered packets, but also the order in which they are recovered [5].

Ultimately, we have two notions of delay:

1) **Completion delay**: The delay in decoding all packets in a generation, which is related to throughput.
2) **In-order delivery delay**: the time between a packet is first transmitted as a coded combination and when it is decoded.

In an ideal world, in a system with feedback and delays, we would like to jointly minimize both. We shall see that is not possible, only trade-offs.

A natural idea is to use a **systematic** code. With I being the $K \times K$ identity matrix we have

$$A = \begin{pmatrix} I \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ \alpha_{(K+1,1)} & \alpha_{(K+1,2)} & \cdots & \alpha_{(K+1,K-1)} & \alpha_{(K+1,K)} \\ \alpha_{(K+2,1)} & \alpha_{(K+2,2)} & \cdots & \alpha_{(K+2,K-1)} & \alpha_{(K+2,K)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \alpha_{(N,1)} & \alpha_{(N,2)} & \cdots & \alpha_{(N,K-1)} & \alpha_{(N,K)} \end{pmatrix}.$$

A code whose matrix includes I, generally at the start, is termed a systematic code.

Why would we use a systematic code? It is still the case that, so long as the $\alpha_{(i,j)}$ are not badly selected, we can resolve the Gaussian elimination with any K equations. The advantage is that the first K equations are uncoded, which means that we would receive all the source packets in order without the need to wait for the whole block, up until the first packet loss occurs. Moreover, the Gaussian elimination would be computationally simpler as all of the uncoded data that were received are in the clear.

Exercise 4.2

(a) Write code that takes binary strings of length n and maps them to an integer representation in $\{0, 1, \dots, 2^n - 1\}$.

(b) Repeatedly randomly generate sets of $K = 3$ binary strings, each of length $n = 4$ bits, and map them to integers, $(X_{(1)}, X_{(2)}, X_{(3)})$. For the largest prime $q > 2^n$ that minimizes the fractional loss, create two random $K \times K$ encoding matrices in S by selecting the entries, $\alpha_{(i,j)}$ and $\beta_{(i,j)}$ for $i, j \in \{1, \dots, K\}$ uniformly at random in S. Using multiplication mod q,

(Continued)

Exercise 4.2 (Continued)

empirically establish:

$$\begin{pmatrix} \beta_{(1,1)} & \cdots & \beta_{(1,K)} \\ \vdots & \vdots & \vdots \\ \beta_{(K,1)} & \cdots & \beta_{(K,K)} \end{pmatrix} \left(\begin{pmatrix} \alpha_{(1,1)} & \cdots & \alpha_{(1,K)} \\ \vdots & \vdots & \vdots \\ \alpha_{(K,1)} & \cdots & \alpha_{(K,K)} \end{pmatrix} \begin{pmatrix} X_{(1)} \\ \vdots \\ X_{(K)} \end{pmatrix} \right)$$
$$= \left(\begin{pmatrix} \beta_{(1,1)} & \cdots & \beta_{(1,K)} \\ \vdots & \vdots & \vdots \\ \beta_{(K,1)} & \cdots & \beta_{(K,K)} \end{pmatrix} \begin{pmatrix} \alpha_{(1,1)} & \cdots & \alpha_{(1,K)} \\ \vdots & \vdots & \vdots \\ \alpha_{(K,1)} & \cdots & \alpha_{(K,K)} \end{pmatrix} \right) \begin{pmatrix} X_{(1)} \\ \vdots \\ X_{(K)} \end{pmatrix}.$$

In the erasure coding context, this means we can recode without having to decode.

Bonus point: If α and β entries are chosen uniformly at random, what is the empirical frequency of the entries of the α times β matrix? Do each of the values in $S = \{0, 1, \dots, q-1\}$ seem to appear equally often, or are some values more prevalent?

4.5.2 Role of Acknowledgments

In many networking systems, we rely on **feedback** from the sender to the receiver in order to manage losses of packets. Suppose we have full feedback, then reliable communication over a packet erasure channel can be achieved using **Automatic Repeat reQuest (ARQ)**. Simply put, if a packet is not received, it is transmitted again until the sender hears from the receiver that a successful reception of the packet has indeed taken place.

If we represent ARQ as a coding matrix, then every transmission has only one non-zero coefficient reflecting the packet to be transmitted at that time slot, as shown in Fig. 4.5. In the following representation of coding matrices, we can think of the y-axis as representing time and the x-axis as indicating which packets are to be combined. Each row contains an entry indicating the packet sent during that time slot, and a crossed-out X in a row means that the row has been erased. Note that the pattern of lost transmissions is the same in all examples that follow.

This simple scheme achieves 100% throughput and, if there is a single point-to-point link and no delay in the feedback, the lowest possible in-order packet delivery delay. If there is a feedback delay, that is no longer the case, and coding can provide a benefit over ARQ. The example in Fig. 4.5 has a delay of two time slots. The first packet $\mathbf{p}_1$ is sent at time 1, it is received at time 3, an acknowledgment of it is immediately transmitted at time slot 3 (transmission not shown) and received two time slots later. If a packet is not received (say packet 3), a negative acknowledgment, abbreviated to **NACK**, is sent.

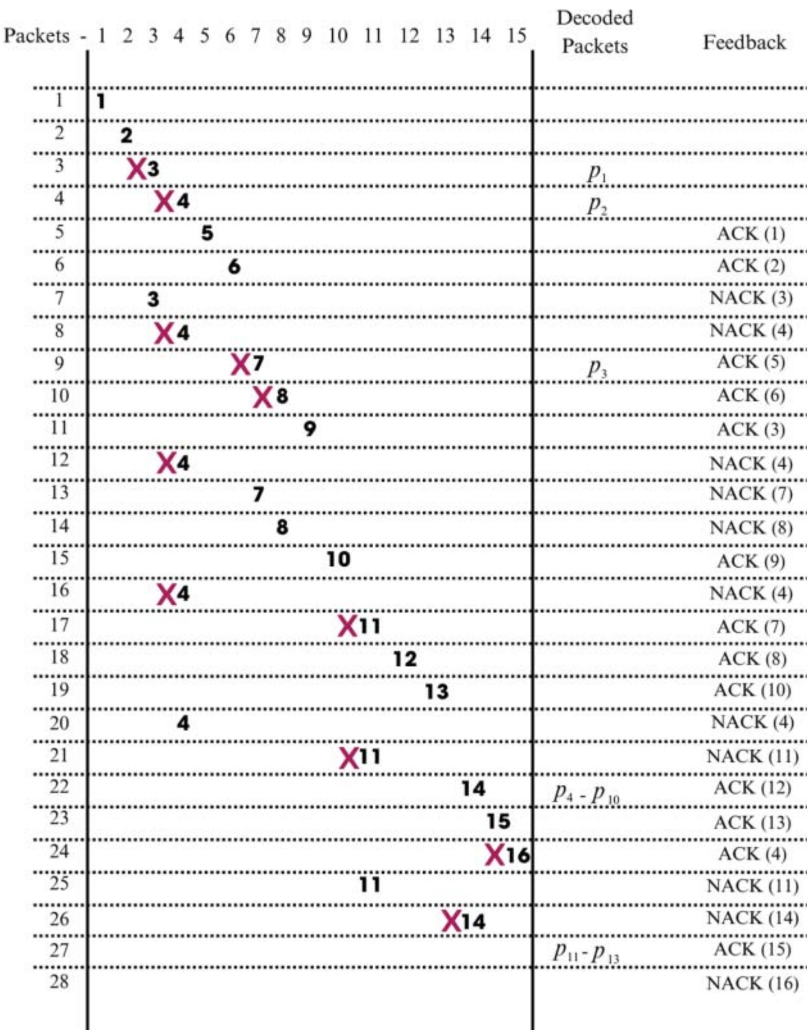


Figure 4.5 Representation of ARQ. Source: This figure is inspired by joint work with Jason Cloud.

Exercise 4.3
Show or convince yourself that, if an ACK or NACK is received immediately after the packet arrives or fails to arrive, respectively, so that a retransmission immediately follows a failed transmission of a packet, the delay between a packet transmission and its successful reception is minimized.

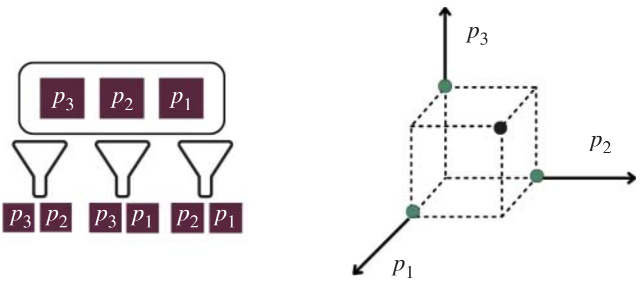
4.6 Network Coding for Losses

It is directly observable that ARQ does not work well with broadcast-mode links because retransmitting a packet that some receivers did not get is wasteful for the others that already have it. In contrast, network coding readily extends to broadcast-mode links.

The following example, inspired by Eryilmaz et al. [6], Fig. 4.6 illustrates the broadcast effect in wireless networks where a sender's transmissions are heard by all receivers, with possible losses. The sender broadcasts three packets, denoted by $\mathbf{p}_1$ through $\mathbf{p}_3$, sent at different consecutive time slots, to three receivers, who all wish to receive all three packets. The pattern of losses at the receivers from the sender is such that at each of the first time slots, a single different receiver misses the packet transmitted at that time slot.

If one were to have the receiver retransmit the packets, then at the fourth time slot, one of the three packets would need to be selected for retransmission, for the benefit of one of the receivers. All three packets would need to be retransmitted consecutively, requiring at minimum six time slots, assuming no further losses occur after the first three packet transmissions.

With the benefit of the knowledge we have gathered about coding, we see that we can be more efficient. Each receiver wishes to recover three packets, each of which is a vector of bits, which we can represent as a vector in a finite field, using the techniques we have developed. For the time being, we shall not worry about the specific



$t = 1$	$t = 2$	$t = 3$	$t = 4$
X	p_2	p_3	$p_1 + p_2 + p_3$
p_1	X	p_3	$p_1 + p_2 + p_3$
p_1	p_2	X	$p_1 + p_2 + p_3$

Figure 4.6 A broadcast channel with erasures. This figure is inspired by joint work with Atilla Eryilmaz.

structure of other aspects of the packet, such as identifiers, or about the fact that packets might be of different lengths. Such matters are better treated in the contact of specific protocols and will be addressed.

Each receiver begins with three unknowns to be recovered. At time 1, receivers 2 and 3 and now have one of those unknowns, $\mathbf{p}_1$, resolved. At time 2, receivers 1 and 3 have another unknown, $\mathbf{p}_2$, resolved. Finally, at time 3, receivers 1 and 2 have the third unknown $\mathbf{p}_3$.

We see that at time slot 3, each receiver can represent its knowledge as a matrix that is not yet invertible. A receiver needs to have a 3×3 invertible matrix but instead has a 2×3 matrix. Sending a coded packet that is a single equation, if received by all three receivers, will allow all receivers to complete their individual matrices to being 3×3 matrices that are readily amenable to Gaussian elimination:

$$\text{Rec. 1}\begin{pmatrix}0 & 1 & 0\\0 & 0 & 1\\1 & 1 & 1\end{pmatrix} \qquad \text{Rec. 2}\begin{pmatrix}1 & 0 & 0\\0 & 0 & 1\\1 & 1 & 1\end{pmatrix} \qquad \text{Rec. 3}\begin{pmatrix}1 & 0 & 0\\0 & 1 & 0\\1 & 1 & 1\end{pmatrix}.$$

If we work over an extension field, the number of bits for the new coded packet that is transmitted is the same as that of any packet. Figure 4.6 shows that a simple equation suffices to complete the matrix at each receiver. We represent in Fig. 4.6 that third equation as a point in a three-dimensional space, where the equation that is transmitted is point $(1,1,1)$ along the axes represented by the three $\mathbf{p}_i$s.

At this point, we know that the specific coefficients assigned in the third transmission do not matter much. What matters is that the resulting matrices at the receivers are invertible. So, different choices of coding coefficients, other than 1 for every packet, would have been appropriate, if the finite field allows for more choices beyond 0 and 1. The exercise below shows that a simple binary is generally not enough for reducing the number of transmissions. We can consider, rather than the specific equations, the notion of receiving **degrees of freedom (DoFs)**, i.e. innovative linear combinations. **Degree of freedom** refers to a new dimension in the appropriate vector space [4, 5]. It is on these DoFs that our management of erasures will rely.

Exercise 4.4

If data is being considered in $\mathbb{F}_2$, by coding we have gone from three points $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$ in the left-hand side of Fig. 4.7 to seven possibilities (excluding sending nothing) on the right-hand side. By coding over higher field sizes, we can extend to having more possibilities, which may be useful.

(Continued)

Exercise 4.4 (Continued)

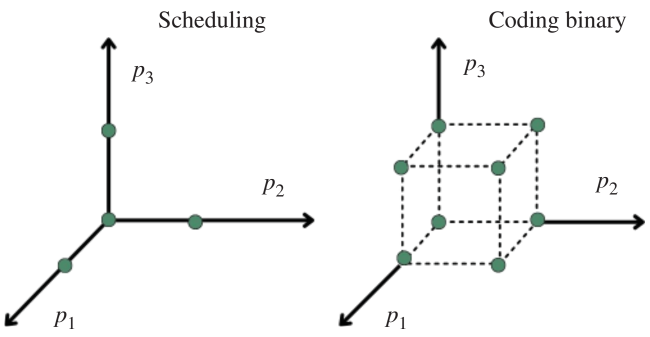


Figure 4.7 Coding versus not coding for three packets over a binary field. This figure is inspired by joint work with Atilla Eryilmaz.

For the three-receiver example of Fig. 4.6, come up with an example of erasure patterns where coding over $\mathbb{F}_2$ will not minimize the number of transmissions needed for all the receivers to receive all of the three packets.

4.6.1 Coding Until Completion

The simplest approach to considering coding until completion comes from rateless codes. In that case, feedback only arrives when **all the packets have been decoded**. There is no concern for managing packets online. Often the matrices are still systematic codes, but no other effort to manage delay is made. For an illustration of a rateless code, see Fig. 4.8.

In this example, the first six source packets are coded together. Two additional combinations of these six packets are also sent a priori to compensate for the losses. In such a case, the receiver can wait until the reception of the eighth coded packet to send feedback. That is, it can decode as soon as it gets six combinations, but, in case of losses, it need not send feedback immediately after the expected reception time of the sixth packet. In this example, there are more than two packets lost in the first eight slots. Thus the receiver is not yet able to decode the packets. As we expect a delay of two time slots in forward and feedback transmissions, the feedback after the eighth slot transmission reaches the sender only by the 12th slot. Then an additional combination is sent in the next slot. This provides the missing degree of freedom and allows for the decoding of all six packets in this first **generation** of packets. This happens at slot 15, and the feedback reaches the sender at slot 17. Slot 18 sees the next generation of source packets being coded together and sent.

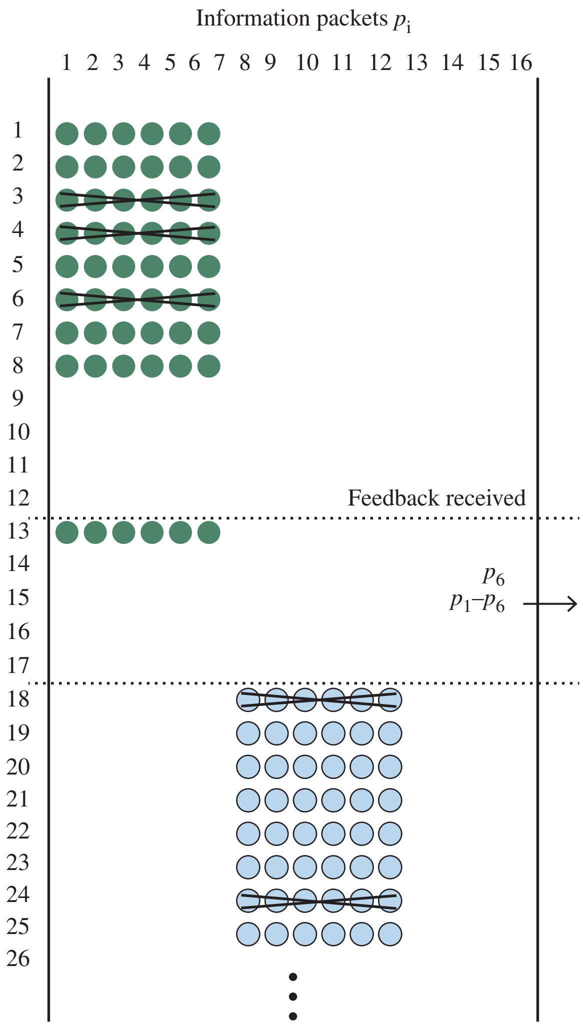


Figure 4.8 Representation of coding until completion. With a delay of 2 on each transmission, there is some overhead in the receiver telling the sender that a generation of *K* packets has been decoded. This figure is inspired by joint work with Jason Cloud.

At the end of a generation, i.e. once *K* coded packets have been received, the matrix of encoding vectors looks like

$$\begin{pmatrix} \alpha_{(1,1)} & \alpha_{(1,2)} & \cdots & \alpha_{(1,K)} \\ \alpha_{(2,1)} & \alpha_{(2,2)} & \cdots & \alpha_{(2,K)} \\ \vdots & \vdots & \cdots & \vdots \\ \alpha_{(K,1)} & \alpha_{(K,2)} & \cdots & \alpha_{(K,K)} \end{pmatrix}$$

which can only be decoded at the last step. In the absence of feedback delay, this approach minimizes the delay in decoding a generation. In the presence of feedback delay, it only does so as $K \to \infty$.

Exercise 4.5 Implementing Multicast/Broadcast Simulation

Using PyErasure, implement a program that simulates a multicast/ broadcast scenario using two different transmission methods: uncoded and coded. Follow the instructions below:

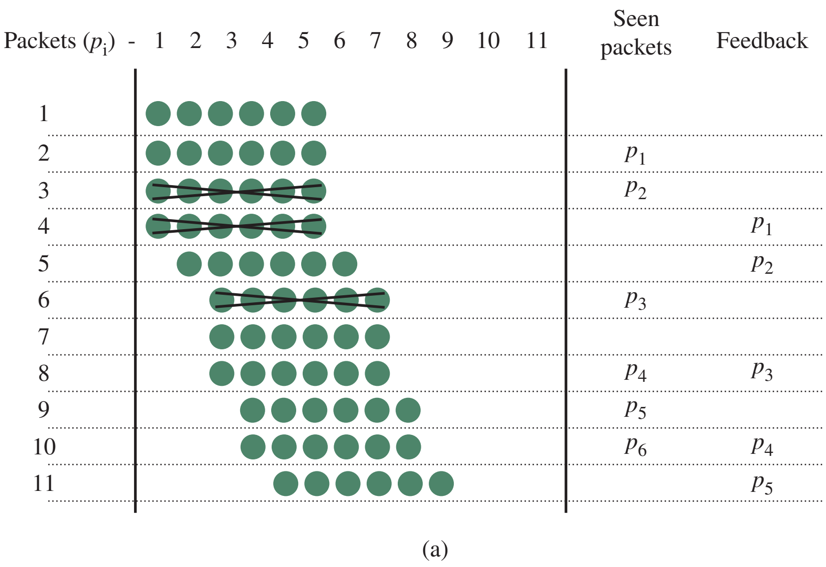
1) Initialize parameters:
   - $N = 100$ (number of receivers)
   - Packet drop rate $= 5\%$
   - $K = 200$ (number of packets to transmit)
2) Uncoded System Simulation:
   - Replicate packet i until it has been received by all receivers.
   - Track the number of transmissions needed by the sender to transmit all K packets to all receivers.
3) Coded System Simulation (e.g. Random Linear Network Code):
   - Encode and send a new coded packet in each iteration.
   - Track the number of transmissions needed by the sender to achieve decoding at all receivers.
4) Output:
   - Print the number of transmissions needed for both the uncoded and coded systems.

Bonus: try to vary the number of receivers N and plot the change in transmissions needed.

4.6.2 Coding Until Completion – Sliding Window

An alternative approach is to use a sliding window approach where the last **seen** packet is fed back. Consider the same example as before, but now the feedback is sent based on the number of packets that have been received, even if they have not been decoded. This notion of seen packets is a powerful tool in analyzing and understanding network coding. It will be explained in greater detail later in Section 4.7.2, but now let us form some initial considerations.

We have already discussed the idea of degrees of freedom as the new knowledge each coded packet brings to the receiver. Seen packets also reflect the number of degrees of freedom received by the destination node. However, to use this as feedback information and drop seen packets from the coding window may seem confusing. The concept here is that a seen packet is a packet in a combination where the rest of the combination can be represented as yet another linear combination of future packets. Thus future packets need not include any information about this packet or previous packets for it to be decoded.



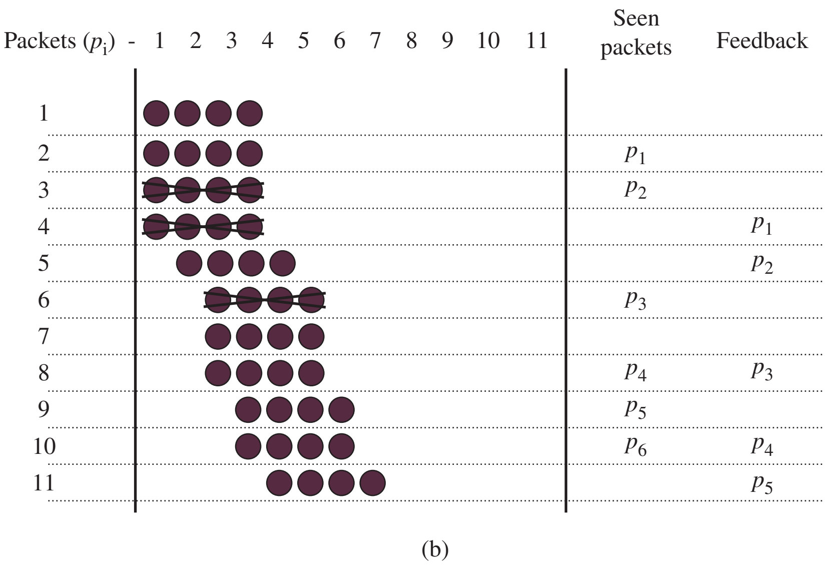


Figure 4.9 Representation of different scenarios of sliding window coding with a delay in feedback. (a) With a window size of six packets and feedback delay of four slots and (b) with a window size of four packets and feedback delay of four slots. This figure is inspired by joint work with Jason Cloud.

With this idea about seen packets, reconsider the previous example. In this case, the first six packets are coded together. As soon as the receiver gets the first coded combination, it can acknowledge it has seen the first packet. When the sender gets this acknowledgment at slot 4, it can remove the first packet from the queue at the source and include the next packet. This forms a sliding window based on the packets seen. If a new packet is not seen, the window does not slide. This is shown in Fig. 4.9a.

However, we have a lot of packets in the window every time. Do we really need that much? Figure 4.9b shows that it can still work with four packets included in the window. If the maximum window size is less than the round trip delay, then there may be instances where no packets are sent before feedback about a new "seen" packet is received and the window is kept stagnant. To avoid this, it is better to have the maximum window size larger than the delay if possible. Figure 4.10 shows a case when there is no delay and a window size of only 2. With a **window-size** of 2, the final received matrix would look as follows at the receiver

$$\begin{pmatrix} \alpha_{(1,1)} & \alpha_{(1,2)} & 0 & \cdots 0 & 0 & 0 \\ 0 & \alpha_{(2,2)} & \alpha_{(2,3)} & \cdots 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots 0 & \alpha_{(K-1,K-1)} & \alpha_{(K-1,K)} \\ 0 & 0 & 0 & \cdots 0 & 0 & \alpha_{(K,K)} \end{pmatrix}.$$

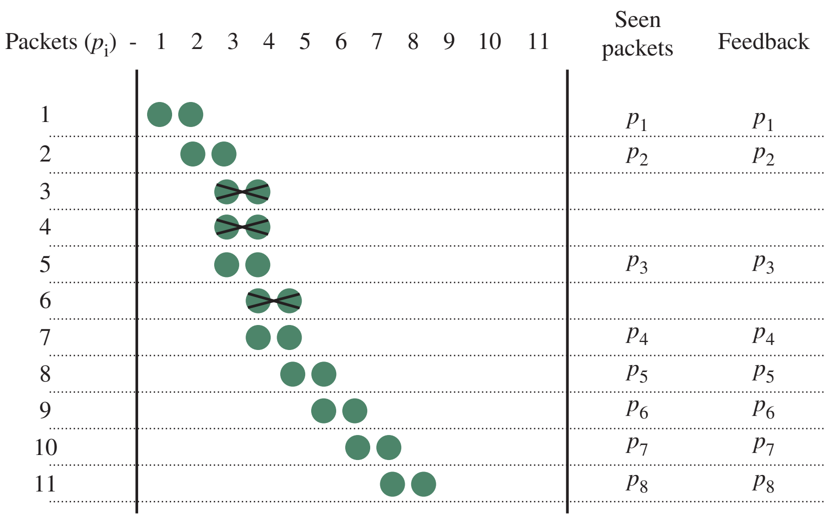


Figure 4.10 Representation of a sliding window with no delay in feedback and a maximum window size of 2.

Again, this can only be decoded at the last step when the final equation enables the rest of the Gaussian elimination.

4.6.3 Semi-systematic Code with Feedback

One approach to reducing in-order delivery delay is shown in Fig. 4.11. Working through the example, it is systematic code in the sense that the original packets

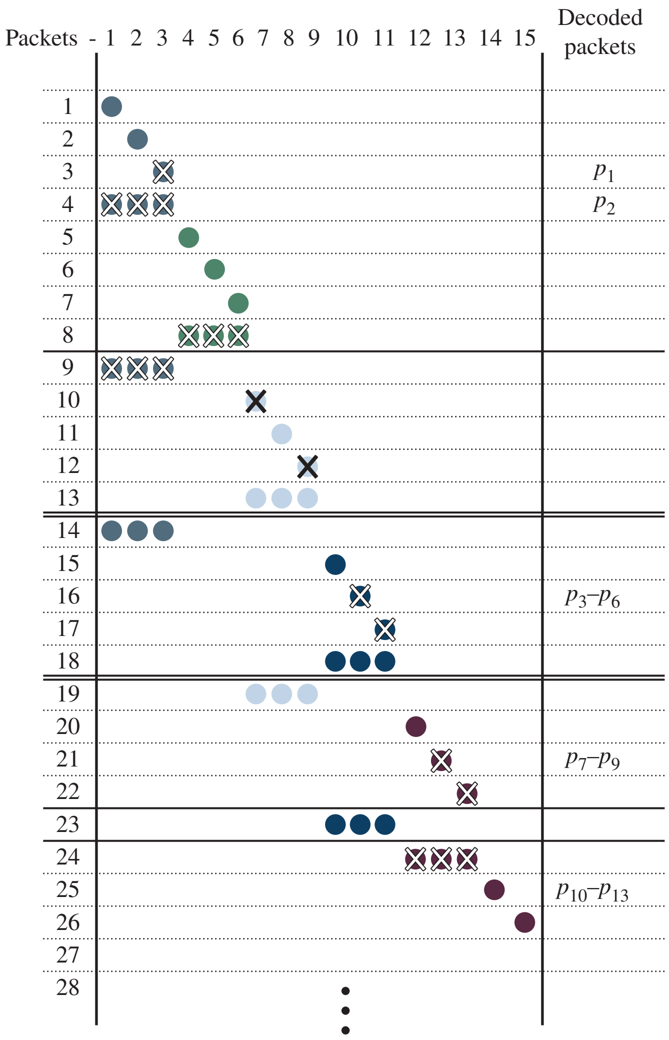


Figure 4.11 Representation of systematic coding with feedback with a delay of 2 on each transmission. This figure is inspired by joint work with Jason Cloud.

are sent as it is. However, there is an additional coding packet sent after every third packet, which is a combination of all the previous three packets. You can see that at slot 4 and slot 8 in the example. However, as can be seen from the error pattern, packets at slots 3 and 4 are erased. Thus even after an extra coded packet is sent, $\mathbf{p}_3$ couldn't be decoded. With the Round Trip Time (RTT) of four slots, this information reaches the sender at slot 8 and an additional coded packet for the first generation (first three packets) is sent at slot 9. However, this packet is lost again. This information will be reached at the source, as feedback, only at slot 13, and we can see another coded packet for the first generation sent at slot 14. This time it passes through. By now, all packets from $\mathbf{p}_1$ to $\mathbf{p}_6$ have reached the receiver. However $\mathbf{p}_7$ is missed and there were more errors in that generation, so it will require an additional transmission, which we see in slot 19. This is an adaptive method of coding where a priori messages are sent but still consider the feedback to send additional packets when required.

Exercise 4.6 Sparse Coefficient Generator for PyErasure

Implement a custom "sparse" coefficient generator for PyErasure to reduce the number of non-zero values in the coefficient matrix, thereby minimizing the number of operations required for encoding and decoding.

1) Design Sparse Coefficient Generator:
   - Design and implement a custom coefficient generator for PyErasure that produces sparse coefficient matrices with a reduced number of non-zero values.
   - Integrate the sparse coefficient generator into PyErasure, following the structure and conventions of existing coefficient generators like Random
     Uniform.
2) Testing and Evaluation:
   - Test the sparse coefficient generator with PyErasure to ensure its functionality and correctness.
   - Evaluate the impact of the sparse coefficient matrices on encoding and decoding operations in terms of computational complexity and efficiency.
3) Performance Comparison:
   - Compare the performance of encoding and decoding operations using the sparse coefficient generator against existing coefficient generators in PyErasure.
   - Analyze the reduction in the number of operations and the resulting improvements in computational efficiency.

4.7 Queueing and Network Coding

A natural question is whether it is possible to obtain benefits from network coding and ARQ by **acknowledging degrees of freedom instead of original packets**. This new framework allows the code to incorporate receivers' states of knowledge and thereby enables the sender to control the evolution of the front of contiguous knowledge. The scheme may thus be viewed as a first step in feedback-based control of the tradeoff between throughput and decoding delay. This new kind of feedback is useful in queue management as it can be used to decide which source packets must be retained in the sender's queue. The ultimate goal is to use feedback to minimize both queue size and decoding delay, but here we will focus on queue size.

4.7.1 Managing Queue Size at the Sender

Consider a packet broadcast channel subject to erasures. We shall call a collection of K source packets a **generation** of packets. If feedback is used only to signal completion of a generation, then the sender will have to store the entire generation till it is decoded. If instead, receivers ACK each source packet upon decoding, the sender can drop the packets that all receivers have decoded.

However, even storing all undecoded packets may be sub-optimal. Consider a situation where the sender has K packets and all receivers have received $(K-1)$ linear combinations: $(\mathbf{p}_1+\mathbf{p}_2), (\mathbf{p}_2+\mathbf{p}_3), \ldots, (\mathbf{p}_{K-1}+\mathbf{p}_K)$. No packet can be decoded by any receiver, so the sender cannot drop any packet. However, the backlog in degrees of freedom is just 1. It would be enough if the sender stores any single one of the $\mathbf{p}_i$. The degree of freedom backlog is also called the **virtual queue**.

Ideally, we want the number of packets stored in the physical queue to track the number of packets in the virtual queue. The **drop-when-decoded** scheme will not always achieve this goal as even if the receivers get degrees of freedom at the specified rate, there can be a delay in decoding the original packets. We can, however, achieve this goal if we allow **ACKs on DoF**.

4.7.2 Feedback for Network Coding and Its Implications

An online coding and queue update algorithm can use ACKs on DoFs to guarantee that the physical queue size at the sender will track the backlog in DoFs. We formally introduce the notion of a node **seeing** a message packet, which is defined as follows. Recall that we treat packets as vectors over a finite field.

A receiver is said to have **seen** a packet $\mathbf{p}$ if it has enough information to compute a linear combination of the form $(\mathbf{p}+\mathbf{q})$, where $\mathbf{q}$ is itself a linear combination

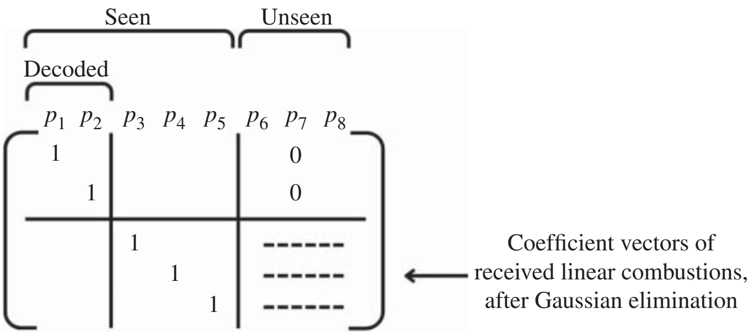

Seen
Unseen
Decoded
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8
Coefficient vectors of received linear combustions, after Gaussian elimination

Figure 4.12 Decoded, seen, and unseen packets. This figure is inspired by joint work with Jay Kumar Sundararajan [7].

involving only packets that arrived after **p** at the sender. Note that decoding implies seeing, as we can pick $\mathbf{q} = \mathbf{0}$.

Figure 4.12 illustrates the meaning of a seen packet. A decoded packet $\mathbf{p}_i$ is such that there is a row in the received coding matrix that has, after performing necessary Gaussian elimination, a 1 in the ith column and 0s everywhere else. A seen packet is such that, after manipulation of the matrix, there is a row in the received coding matrix which has a 1 in the ith column and 0s in all of the preceding columns. This means that the linear contribution, due to coding, of previous packets, $\mathbf{p}_j$ where $j < i$ can be eliminated. For a decoded packet, the linear contribution of all packets $\mathbf{p}_j$ where $j \neq i$ can be eliminated.

The scheme is called the **drop-when-seen** algorithm [8, 9] because **a packet is dropped if all receivers have seen it**. The intuition is that if all receivers have seen **p**, it is enough for the sender's transmissions to involve only packets beyond **p**. After decoding these packets, the receivers can compute **q** and hence obtain **p** as well. Therefore, even if the receivers have not decoded **p**, no information is lost by dropping it.

Whereas ARQ ACKs a packet upon decoding, this scheme ACKs a packet when it is seen. We present a deterministic coding scheme that guarantees that the coded packet, if received successfully, would **simultaneously cause each receiver to see its next unseen packet**. Now, seeing a new packet translates to receiving a new degree of freedom (proved later). This means, the innovation guarantee property is satisfied and 100% throughput can be achieved.

Table 4.1 gives an example of the proposed idea in a packet erasure broadcast channel with two receivers A and B. The sender's queue is shown after the arrival point and before the transmission point of a slot. In each slot, the sender picks the oldest unseen packet for A and B. If they are the same packet, then that packet is sent. If not, their XOR is sent. This rule will cause both receivers to see their

Table 4.1 An example of the drop-when-seen algorithm.

				A		B	
Time	**Sender's queue**	**Transmitted packet**	**Channel state**	**Decoded**	**Seen but not decoded**	**Decoded**	**Seen but not decoded**
1	$\mathbf{p}_1$	$\mathbf{p}_1$	$\rightarrow$ A, $\nrightarrow$ B	$\mathbf{p}_1$	—	—	—
2	$\mathbf{p}_1, \mathbf{p}_2$	$\mathbf{p}_1 \oplus \mathbf{p}_2$	$\rightarrow$ A, $\rightarrow$ B	$\mathbf{p}_1, \mathbf{p}_2$	—	—	$\mathbf{p}_1$
3	$\mathbf{p}_2, \mathbf{p}_3$	$\mathbf{p}_2 \oplus \mathbf{p}_3$	$\nrightarrow$ A, $\rightarrow$ B	$\mathbf{p}_1, \mathbf{p}_2$	—	—	$\mathbf{p}_1, \mathbf{p}_2$
4	$\mathbf{p}_3$	$\mathbf{p}_3$	$\nrightarrow$ A, $\rightarrow$ B	$\mathbf{p}_1, \mathbf{p}_2$	—	$\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$	—
5	$\mathbf{p}_3, \mathbf{p}_4$	$\mathbf{p}_3 \oplus \mathbf{p}_4$	$\rightarrow$ A, $\nrightarrow$ B	$\mathbf{p}_1, \mathbf{p}_2$	$\mathbf{p}_3$	$\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$	—
6	$\mathbf{p}_4$	$\mathbf{p}_4$	$\rightarrow$ A, $\rightarrow$ B	$\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4$	—	$\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4$	—

oldest unseen packet. In slot 1, $\mathbf{p}_1$ reaches A but not B. In slot 2, $(\mathbf{p}_1 \oplus \mathbf{p}_2)$ reaches A and B. Since A knows $\mathbf{p}_1$, it can also decode $\mathbf{p}_2$. As for B, it has now seen (but not decoded) $\mathbf{p}_1$. At this point, since A and B have seen $\mathbf{p}_1$, the sender drops it. This is fine because, B will eventually decode $\mathbf{p}_2$ (this happens in slot 4), at which time it can obtain $\mathbf{p}_1$. Similarly, $\mathbf{p}_2$ and $\mathbf{p}_3$ are also dropped once they are seen. Finally, in slot 6, both receivers receive $\mathbf{p}_4$ and decode all 4 packets, thus catching up with the sender. This example shows clearly that it is fine to drop packets before they are decoded. However, the drop-when-decoded policy will drop $\mathbf{p}_1$ and $\mathbf{p}_2$ in slot 4, and $\mathbf{p}_3$ and $\mathbf{p}_4$ in slot 6. Thus, our new strategy clearly keeps the queue shorter.

Exercise 4.7

The example above uses only binary coding over $\mathbb{F}_2$. Convince yourself that it is sufficient because there are only two receivers. Compare this with exercise 4.4.

4.7.3 A Quick Primer on Queueing Theory

Armed with a mechanism to encode packets into a finite field and decode them on receipt with Gaussian elimination, it is simple and informative to create a stochastic simulation to assess the performance of the various packet coding rules, including the management of the queue at the sender.

It is instructive, however, to engage in a simple queueing theory analysis to assess how the average queue-length at the sender scales as packet arrival rates increase. As our goal is to gain intuition without the distraction of excessive mathematical detail, we do this in the simplest of settings:

- We assume a single sender, receiver pair.
- We assume perfect feedback, zero-delay confirming receipt or erasure.
- We shall assume that the arrival times of packets to the sender, $\{\tau_i\}$, are independent and exponentially distributed with mean inter-arrival time λ^{-1}.
- We shall assume that the times between which successful, non-erased transmissions are made (should there be a packet present in the queue), $\{\xi_i\}$, are independent and exponentially distributed with inter-departure time μ^{-1}. If no packet is present, the transmission opportunity is lost.
- We shall assume that $\lambda^{-1} > \mu^{-1}$, i.e. $\lambda < \mu$, so that arrivals, on average, occur at greater intervals than successful transmissions so that the system is stable.
- We shall define the load on the system as $\rho = \lambda/\mu$, notating $\rho < 1$ by the previous assumption.
- We shall assume that the sender has an unbounded packet storage buffer.

If a packet was dropped from the queue on successful transmission, in queueing parlance (i.e. **Kendall notation**) the packet arrival and departure process would form an **M/M/1 queue**, where M stands for memoryless, 1 indicates how many servers are present, and the lack of another number indicates there is no buffer constraint.

In queueing theory, two fundamental quantities are often studied:

- The **stationary waiting time distribution**, **W**: if the queueing system has been running for a long time, the distribution of the time that an arriving packet will have to wait until its service is complete and it departs the queue.
- The **stationary queue-length distribution**, **Q**: if the queueing system has been running for a long time, the distribution of the number of packets that it will find ahead of it in the first-come first-served queue.

Understanding statistics of these distributions, such as the mean $E(Q)$ and variance $\mathrm{Var}(Q)$ of the queue-length distribution, informs about storage needs.

We shall build on this basic model to consider what happens to the queue length at the sender when they send coded packets. To do so, we will also add:

- At each successful transmission, the sender transmits a **coded packet** that has a non-zero linear combination of everything in its buffer.
- We shall assume that the finite field is sufficiently large that every coded packet is **innovative**.
- We shall consider the **drop-when-seen** and **drop-when decoded** queue management rules.

The core idea in this setting is that each successful transmission results in a new seen packet and so, under **drop-when-seen**, the sender can reduce the number of packets it is storing; under **drop-when decoded**, the receiver can only decode when it has received as many packets as are in the queue, whereupon the sender can empty the queue.

For example, consider the case where the sender gets two packets before it can successfully relay one, it then gets a third packet before two successful transmissions in a row without receiving any new packets. In **drop-when-seen**, the coding matrix would be

$$\begin{pmatrix} \alpha_{(1,1)} & \alpha_{(1,2)} & 0 \\ 0 & \alpha_{(2,2)} & \alpha_{(2,3)} \\ 0 & 0 & \alpha_{(3,3)} \end{pmatrix}$$

but in **drop-when decoded** it would be

$$\begin{pmatrix} \alpha_{(1,1)} & \alpha_{(1,2)} & 0 \\ \alpha_{(2,1)} & \alpha_{(2,2)} & \alpha_{(2,3)} \\ \alpha_{(3,1)} & \alpha_{(3,1)} & \alpha_{(3,3)} \end{pmatrix}.$$

Neither receiver can decode early, but **drop-when-seen** results in a sparser matrix and the sender has a smaller collection of buffered packets. What we wish to understand is how much shorter the sender's queue will be. In a system where the arrival rate is such that 90% of the time there are packets in the sender's queue, Fig. 4.13 provides simulated sample-paths of the queue-length at the sender under the **drop-when-seen** and **drop-when-decoded** strategies as a function of time. Clearly the former is far superior.

We shall use results that come from the memoryless property, which result in the system being **Markovian** (what happens next only depends on randomness

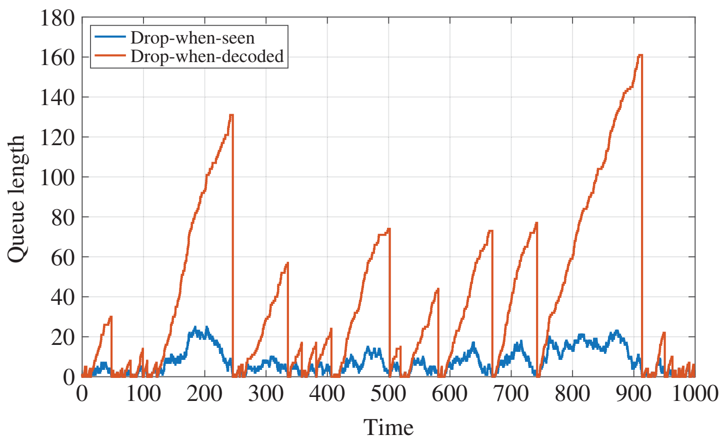


Figure 4.13 Simulation of queue-length evolution of the sender for **drop-when-seen** and **drop-when-decoded** within the mathematical model described in this section.

and the current state of the system, not the history of the system beyond that) and an incredibly robust property of queueing systems called **Little's law**.

4.7.4 The Magic of Independent Exponentials

Before we consider the coded packet setup, let's start with the basic M/M/1 queue. A key feature of the system is the memorylessness, which arises due to a key feature of independent exponential times to events.

Consider two independent, exponentially distributed random variables, τ and ξ, with rate λ and τ, respectively. That is,

$$F_\tau(x) = P(\tau \le x) = 1 - e^{-\lambda x}, \qquad f_\tau(x) = \frac{dP(\tau \le x)}{dx} = \lambda e^{-\lambda x},$$
$$F_\xi(y) = P(\xi \le x) = 1 - e^{-\mu x}, \qquad f_\xi(x) = \frac{dP(\xi \le x)}{dx} = \mu e^{-\mu x},$$

and hence

$$\mathbb{E}(\tau) = \int_0^\infty (1 - F_\tau(x))dx = \int_0^\infty e^{-\lambda x}dx = \frac{1}{\lambda},$$
$$\text{and } \mathbb{E}(\xi) = \int_0^\infty (1 - F_\xi(x))dx = \frac{1}{\mu}.$$

Consider τ and ξ as representing random times for different events. We are interested in when the first of them will occur, and which one it will be. Using independence,

$$P(\min(\tau, \xi) > x) = P(\tau > x, \xi > x) = P(\tau > x)P(\xi > x) = e^{-\lambda x}e^{-\mu x}$$
$$= e^{-(\lambda+\mu)x},$$

from which we deduce that $\min(\tau, \xi)$ is itself an exponentially distributed random variable with rate $\lambda + \mu$.

Let us consider the probability that τ occurs first,

$$P(\tau = \min(\tau, \xi)) = P(\tau < \xi) = \int_0^\infty \lambda(1 - F_\tau(x))f_\xi(x)dx$$
$$= \int_0^\infty \lambda e^{-\lambda x}e^{-\mu x}dx = \lambda \int_0^\infty e^{-(\lambda+\mu)x}dx$$
$$= \frac{\lambda}{\lambda + \mu},$$

and, similarly,

$$P(\xi = \min(\tau, \xi)) = \frac{\mu}{\lambda + \mu}.$$

Note that $P(\tau = \xi) = 0$, so we can break ties however we wish without impacting probabilities. Moreover, we could condition on the value of $\min(\tau, \xi)$ and nothing would change. The time at which the first of τ and ξ occurs is exponentially distributed, and the likelihood that one or the other is first is the same for all times.

Assuming that ξ occurs first at some time x, let us consider the distribution of the residual time until τ occurs:

$$P(\tau > x + t | \tau > x) = \frac{P(\tau > x + t, \tau > x)}{P(\tau > x)} = \frac{P(\tau > x + t)}{P(\tau > x)} = \frac{e^{-\lambda(x+t)}}{e^{-\lambda x}}$$
$$= e^{-\lambda t}.$$

That is, the residual time is itself exponentially distributed with the same parameter, and $E(\tau > x + t | \tau > x) = 1/\lambda$.

From these facts, one deduces that

- The time until the next arrival or successful transmission to the queue is exponentially distributed with mean $1/(\lambda + \mu)$.
- Given an arrival or successful transmission occurred at a given time, the likelihood it is an arrival is $\lambda/(\lambda + \mu)$ and the likelihood that it is a departure is $\mu/(\lambda + \mu)$.

Note that we have assumed $\lambda < \mu$, so we are assuming the likelihood an event is an arrival is smaller than the likelihood that it is a successful transmission and so, as mentioned earlier, $\rho = \lambda/\mu < 1$. From these deductions, we have a simple stochastic description of the evolution of the queue-length, which only develops at the instances of arrivals or departures

$$Q_{n+1} = \max\left(0, Q_n + 1_{\tau_{n+1} < \xi_{n+1}} - 1_{\tau_{n+1} \geq \xi_{n+1}}\right) = \max\left(0, Q_n + X_{n+1}\right),$$

where the $\{X_n\}$ process is independent and identically distributed with

$$P(X = x) = \begin{cases} \dfrac{\lambda}{\lambda + \mu} & \text{if } x = +1 \\ \dfrac{\mu}{\lambda + \mu} & \text{if } x = -1. \end{cases}$$

If one considers the embedded time process that describes the queue-length, which only changes at a packet arrival or departure, it can be represented as

$$\begin{tikzcd}
\text{Queue length } 0 \arrow[r, shift left, "\lambda/(\lambda+\mu)"] & 1 \arrow[l, shift left, "\mu/(\lambda+\mu)"] \arrow[r, shift left, "\lambda/(\lambda+\mu)"] & 2 \arrow[l, shift left, "\mu/(\lambda+\mu)"] \arrow[r, shift left, "\lambda/(\lambda+\mu)"] & 3 \arrow[l, shift left, "\mu/(\lambda+\mu)"] \arrow[r, shift left, "\lambda/(\lambda+\mu)"] & 4 \cdots \arrow[l, shift left, "\mu/(\lambda+\mu)"]
\end{tikzcd}$$

where the arrows indicate the direction of change and their likelihood.

This evolution results in the queue-length being a Markov chain and to determine the steady-state distribution, Q, it suffices to consider **detailed balance** equations. Consider the probability flow between two queue-lengths q and $q + 1$. Probability flows down from the queue-length $q + 1$ if there is a successful transmission and up from q if there is an arrival, so that

$$P(Q = q + 1)\frac{\mu}{\lambda + \mu} = P(Q = q)\frac{\lambda}{\lambda + \mu},$$

and hence

$$P(Q = q + 1) = P(Q = q)\frac{\lambda}{\mu} = P(Q = q)\rho.$$

As we have $P(Q = q + 1) = P(Q = q)\rho$ for $q \geq 0$ and

$$1 = \sum_{q=0}^{\infty} P(Q = q) = P(Q = 0)\sum_{q=0}^{\infty} \rho^q = P(Q = 0)\frac{1}{1 - \rho}.$$

As $P(Q > 0) = 1 - P(Q = 0) = \rho$, this explains our definition of ρ as the load of the system: it is the proportion of the time that the queue is non-empty. Using

$$P(Q = q) = (1 - \rho)\rho^q \quad \text{for } q \geq 0, \tag{4.2}$$

we can evaluate, e.g., the average queue-length

$$E(Q) = \sum_{q=0}^{\infty} qP(Q = q) = \sum_{q=0}^{\infty} q\rho^q(1 - \rho) = \frac{\rho}{1 - \rho}. \tag{4.3}$$

Figure 4.14 provides a plot of $E(Q)$ as a function of the load ρ, which only behaves poorly as the load becomes large, i.e. $\rho \to 1$.

In the coded system without feedback delays, where the sender codes everything in its queue, the **drop-when-seen** approach ensures that each successful transmission results in the receiver getting a newly seen in-order packet. Thus, each successful transmission allows the sender to drop one packet from its queue. As a result, the M/M/1 queue exactly describes the queue-length evolution in this

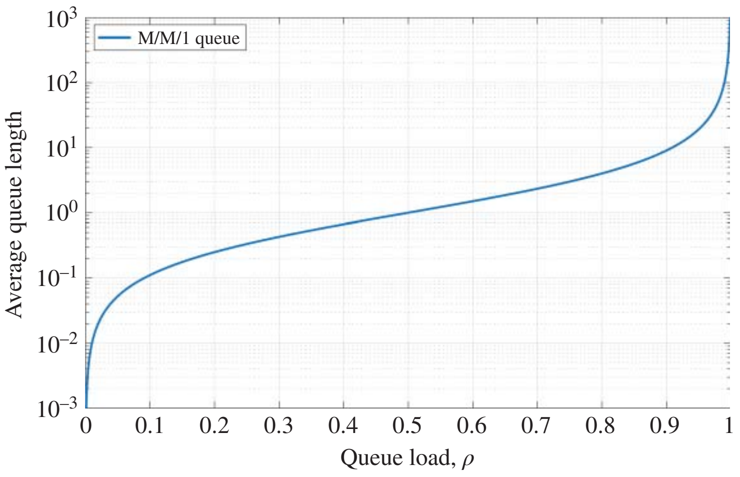


Figure 4.14 Average queue length (i.e. number of packets waiting) in an M/M/1 queue as a function of load, ρ.

system, and Eq. (4.3) tells us the average queue-length. Even when not describing the **drop-when-seen** queue, the evolution tells us how many more **degrees of freedom** the sender has than the receiver.

To relate this queue-length to the average time a packet spends in the system from its arrival to being dropped from the queue, we will use a powerful, generic result of queueing theory called Little's law.

4.7.5 Relating Average Queue Length to Average Waiting Time

Little's law is a simple, yet surprisingly powerful and general way to relate the number of packets in a system to the arrival rate λ into a system and the time that the packets spend in the system. It states that the average number of packets in a system is equal to the arrival rate multiplied by the average time a packet spends in the system:

$$E(Q) = \lambda E(W), \tag{4.4}$$

where $E(Q)$ is the average number of packets in the system, λ is the arrival rate of packets to the system, and $E(W)$ is the average time a packet spends in the system. That is, the average number of packets in the system is the average arrival rate of packets to the system times the average time they spend in the system. As a result $E(Q)$ and $E(W)$ are linearly related by a time-scale determined by the arrival rate $1/\lambda$. It holds for a much larger class of systems than we are considering.

We provide only a sketch proof of Little's law with geometric arguments. Consider Fig. 4.15. The number of arrivals is the top line of the staircase and the number of departures is the bottom line. An arrival is presented by a step up of

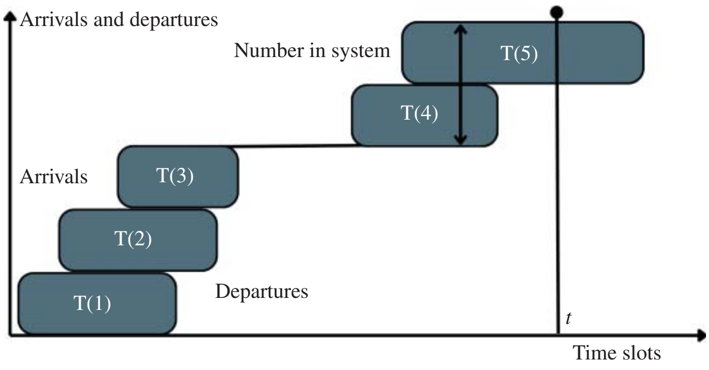


Figure 4.15 Arrivals, departures, and number in the system. Source: This figure is inspired by [10, Fig. 3.8].

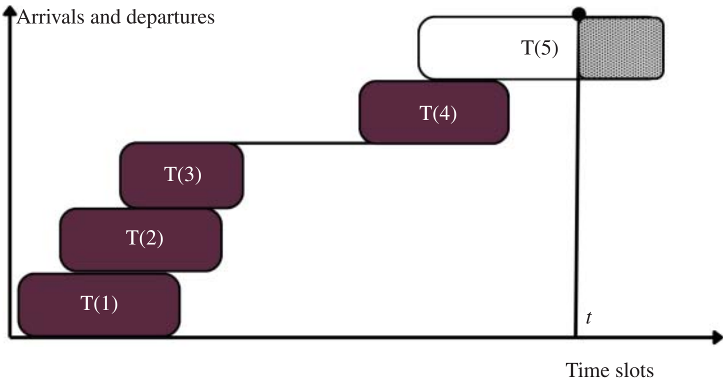


Figure 4.16 Relating arrivals, departures, number in the system and time in the system. Source: This figure is inspired by [10, Fig. 3.8].

size 1 on the top staircase and a departure by a step up of size 1 on the bottom staircase. As notation, we have

- Let $a(t)$ denote the number of arrivals to the system until time t.
- Let $d(t)$ denote the number of departures from the system until time t.
- Let $T(m)$ denote the time that the m-th packet spends in the system.

The number in the system at a time slot is the difference between the total arrivals, $a(t)$, and the total departures, $d(t)$, from the system by that time, t. When the total number of arrivals and departures is equal, there are no packets in the system.

Without rigorously defining stability, for the system not to grow in an unbounded fashion, packets that arrive to the system must at some point depart from it. If the system does not blow up, in the long run, the rate of departures must equal the rate of arrivals

$$\lambda = \lim_{t\to\infty} \frac{a(t)}{t} = \lim_{t\to\infty} \frac{d(t)}{t}.$$

Also on that basis, we also assume that there is a mean time spent in the system and it is finite,

$$E(W) = \lim_{t\to\infty} \frac{1}{a(t)} \sum_{m=1}^{a(t)} T(m) = \lim_{t\to\infty} \frac{1}{d(t)} \sum_{m=1}^{d(t)} T(m).$$

To relate this to the expected queue-length, instead we first consider the integral of the queue-length. The space between the two staircases can be seen a superposition of blocks of height 1 and length equal to $Q(t) = a(t) - b(t)$, the number of packets in the system at time t. If we integrate the queue-length for all arrivals $a(t)$ up to a time t we have the dark gray blocks shown in Fig. 4.15. If we integrate the queue-length for all departures $d(t)$ up to a time t we have the dark blocks shown in Fig. 4.16. So, if were to estimate the integral of the number of packets in the system by summing up the areas of the blocks corresponding to the time in the system of the packets that have arrived or left the system, we would overestimate by the amount shown in dotted area or underestimate by the amount shown in transparent part on the last block. As a result, we have that

$$\sum_{i=1}^{b(t)} T(i) \le \int_{s=0}^{t} Q(s)ds \le \sum_{i=1}^{a(t)} T(i).$$

When averaged over time, the discrepancy as time advances should be negligible relative to the total area between the two staircases. As a result, assuming that the time-average and ensemble average in the system are the same, we have that the average number of things in the system

$$E(Q) = \lim_{t\to\infty} \frac{1}{t} \sum_{i=1}^{a(t)} T(i) = \lim_{t\to\infty} \frac{1}{t} \sum_{i=1}^{d(t)} T(i) = \lim_{t\to\infty} \frac{a(t)}{t} \frac{1}{a(t)} \sum_{m=1}^{a(t)} T(m) = \lambda E(W),$$

which is an instance of **Little's law**.

4.7.6 Drop When Seen

For the M/M/1 queue, from Eqs. (4.3) and (4.4) we would deduce that the expected waiting time satisfies

$$E(W_{\mathrm{dws}}) = \frac{1}{\lambda} E(Q) = \frac{1}{\lambda} \frac{\rho}{1-\rho}. \tag{4.5}$$

As a result, with the **drop-when-seen** protocol, the average time that a packet spends in the sender's queue when there is no feedback delay or loss and all packets in the queue are coded at each time is given by that formula.

4.7.7 Drop When Decoded

Using the **drop-when-decoded** rule, the receiver can only decode when the sender's queue is empty because at that stage the receiver has as many coded packets (i.e. equations) as there were packets in this **generation**. As a result, the M/M/1 queue no longer describes the evolution of the queue-length at the transmitter. However, it still does describe the difference in the number of degrees of freedom between the sender and the receiver at any one time, which is referred to as the **virtual queue**. Each time the virtual queue hits zero, the receiver has enough equations to decode that generation, let the sender know that, and the sender empties their queue.

We provide a heuristic analysis of this case. Given that the queue-length at the nth arrival or successful transmission is greater than 0, the average change in the queue-length at the next arrival or successful transmission is given by

$$E(Q_{n+1} - Q_n | Q_n > 0) = \frac{\lambda}{\lambda + \mu} - \frac{\mu}{\lambda + \mu}.$$

Now, the expected time between arrival or successful transmission events is $1/(\lambda + \mu)$; therefore, on average, the queue-length will drop by $(\lambda - \mu)/(\lambda + \mu)$ in a time period of $1/(\lambda + \mu)$ and so the average time for the queue to reduce by one packet is

$$\frac{1}{\lambda + \mu} \Big/ \left(-\frac{\lambda - \mu}{\lambda + \mu} \right) = \frac{1}{\mu - \lambda} = \frac{1}{\mu} \frac{1}{1 - \rho},$$

bearing in mind that $\mu > \lambda$ to ensure stability. Thus, if the queue-length is q at time t, on average we expect it to become empty at time

$$q \frac{1}{\mu} \frac{1}{1 - \rho}.$$

Given a new packet arrives to the M/M/1 queue, the distribution of the queue-length that it will experience is the stationary queue-length in Eq. (4.2). Thus, by the law of total expectation, the expected time that it will take for the queue to empty, which corresponds to the waiting time in the **drop-when-decoded** algorithm is

$$E(W_{\text{dwd}}) = \sum_{q=0}^{\infty} q \frac{1}{\mu} \frac{1}{1 - \rho} P(Q = q) = \frac{1}{\mu} \frac{1}{1 - \rho} \frac{\rho}{1 - \rho}. \tag{4.6}$$

Expressing this averaging waiting time in terms of the drop-when-seen algorithm, we have

$$E(W_{\text{dwd}}) = \frac{1}{\lambda} \frac{\rho}{(1 - \rho)^2} = \frac{1}{(1 - \rho)} E(W_{\text{dws}}).$$

Using Little's law, $\lambda E(W) = E(Q)$, we have the same relationship between the two strategies for the average queue-length

$$E(Q_{\text{dws}}) = \frac{\rho}{1 - \rho} \quad \text{and} \quad E(Q_{\text{dwd}}) = \frac{\rho}{(1 - \rho)^2}.$$

As $\rho \in (0, 1)$, $1/(1 - \rho) > 0$ and so the waiting time and queue-length in the drop-when-decoded system are necessarily degraded when compared to the drop-when-seen strategy.

Figure 4.17 plots the predicted average queue-length, and so storage needs, for the two strategies, where the degraded performance of **drop-when-decoded** over **drop-when-seen** is particularly evident at higher loads. Setting $\mu = 1$ so that $\lambda = \rho$, Fig. 4.18 plots the average waiting-time before successful receipt of a packet and there it can be seen that **drop-when-decoded** provides notably

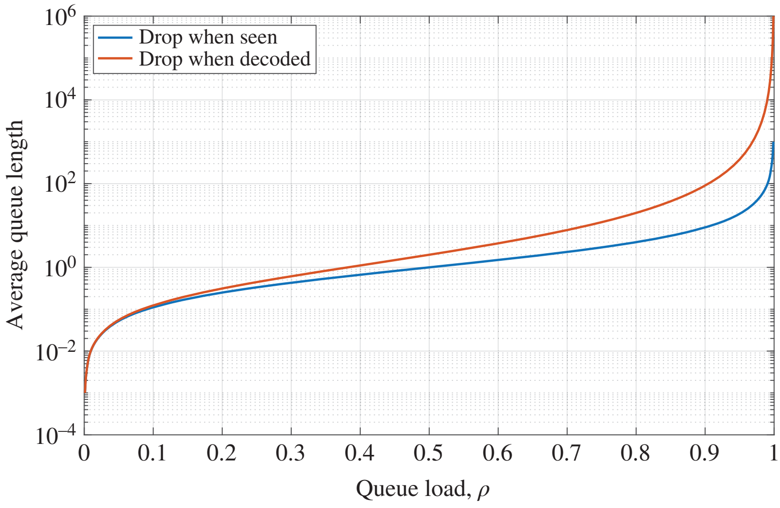


Figure 4.17 Average queue length as a function of load, ρ.

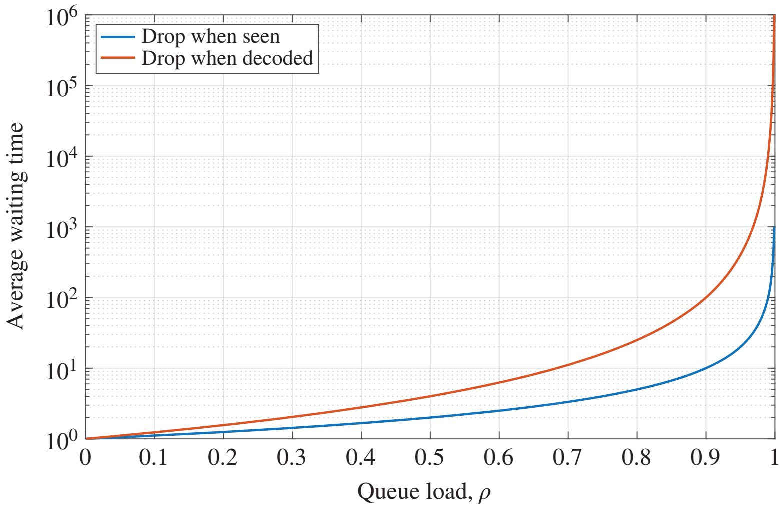


Figure 4.18 Average waiting time, given $\mu = 1$, as a function of load, ρ.

worse performance all loads. Note that the minimum average waiting time is 1 in that plot, corresponding to the average service time. It can be observed that **drop-when-seen** is the far superior strategy in this case, and that extends to more sophisticated arrangements including systems with more than one receiver.

4.8 Summary

Chapter four provides most of the crucial concepts related to NC. Starting with Gauss–Jordan elimination in finite fields, we delve into the realization of mutable data in practice and how the coding over multiple packets works in different settings. The decoding probability and gains of network coding in a lossy communication channel are explored. As is explained, the mixing of packets enables new information to be provided in every packet, leading to the concept of the degree of freedom and on to the innovative use of acknowledgments in a feedback-based communication scheme. Through considerations of queue behavior, this new approach provides significant gains in a lossy network. Major takeaways from this chapter can be considered as:

A) Concepts of degrees of freedom and role of acknowledgment in network coding.
B) Different coding strategies such as systematic coding, coding until completion, sliding window coding and coding with feedback.
C) Decoding probability and queue management in network coding.

Additional Reading Materials

Some additional reading on the likelihood of not being invertible can be found in [2]. More details on the gains of network coding in broadcast settings and networks with delay can be found in [6, 11, 12]. Discussions in [8, 9, 13] shed light on how the concept of degrees of freedom impacts queueing in coded networks, while [14–16] explain the concepts of queueing theory and Little's law in general.

References

1 T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong, "A random linear network coding approach to multicast," *IEEE Transactions on Information Theory*, vol. 52, no. 10, pp. 4413–4430, 2006.

2 W. C. Waterhouse, "How often do determinants over finite fields vanish?" *Discrete Mathematics*, vol. 65, no. 1, pp. 103–104, 1987.

3 P. Billingsley, *Probability and Measure*. John Wiley & Sons, 2017.

4 J. Cloud, D. Leith, and M. Médard, "A coded generalization of selective repeat ARQ," in *IEEE Conference on Computer Communications*, 2015, pp. 2155–2163.

5 M. Karzand, D. J. Leith, J. Cloud, and M. Médard, "Design of FEC for low delay in 5G," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 8, pp. 1783–1793, 2017.

6 A. Eryilmaz, A. Ozdaglar, M. Médard, and E. Ahmed, “On the delay and throughput gains of coding in unreliable networks,” *IEEE Transactions on Information Theory*, vol. 54, no. 12, pp. 5511–5524, 2008.
7 J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher, and J. Barros, “Network coding meets TCP: theory and implementation,” *Proceedings of the IEEE*, vol. 99, no. 3, pp. 490–512, 2011.
8 J. K. Sundararajan, D. Shah, M. Médard, and P. Sadeghi, “Feedback-based online network coding,” *IEEE Transactions on Information Theory*, vol. 63, no. 10, pp. 6628–6649, 2017.
9 J. K. Sundararajan, D. Shah, and M. Médard, “On queueing in coded networks-queue size follows degrees of freedom,” in *IEEE Information Theory Workshop on Information Theory for Wireless Networks*, 2007.
10 D. Bertsekas and R. Gallager, *Data Networks*. Prentice-Hall, Inc., 1987.
11 A. Eryilmaz, A. Ozdaglar, and M. Médard, “On delay performance gains from network coding,” in *Annual Conference on Information Sciences and Systems*, 2006, pp. 864–870.
12 E. Ahmed, A. Eryilmaz, M. Médard, and A. E. Ozdaglar, “On the scaling law of network coding gains in wireless networks,” in *IEEE Military Communications Conference*, 2007.
13 J. K. Sundararajan, “On the role of feedback in network coding,” Ph.D. dissertation, Massachusetts Institute of Technology, 2009.
14 J. D. Little, “A proof for the queuing formula: $L = \lambda W$,” *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.
15 S. Asmussen, *Applied Probability and Queues*. New York: Springer, 2003, vol. 2.
16 A. Leon-Garcia, *Probability and Random Processes for Electrical Engineering*. Pearson Education, 2008.

5

Designing of Protocols with Network Coding

So far, we have been considering concepts of coding and communication without regard for such questions as **timing, conveying coding coefficients, or other matters which in practice are important to ensure correct operation**. For communication between two different devices to happen successfully, both these parties need to share some understanding and rules about these concepts and what are the procedures followed at the other end.

Communication protocols are the rules and conventions that govern this information exchange. Protocols define how data is **represented, transmitted, and received** by different devices. There are many different protocols involved and required for an efficient communication system to exist. It is also important that they follow the same order or architecture on both ends.

One of the most commonly used set of protocols is from the **TCP/IP Protocol suite**. It defines the **Transport Control Protocol (TCP), the User Datagram Protocol (UDP), and the Internet Protocol (IP)**, the latter being in effect a packet definition, which we have heretofore assumed to be given. These are not the only communication protocols, but as part of the TCP/IP suite, these had a vast impact on the communication systems we use currently.

UDP is suitable for purposes where error checking and correction are either not necessary or are performed in the application. Time-sensitive applications can use UDP if dropping packets is preferable to waiting for those delayed due to retransmission, which may not be an option in a real-time system. There are recent protocols, like **Quick UDP Internet Connections (QUIC)**, that have been gaining traction from the engineering community, built atop UDP. Protocols, such as QUIC, can be substituted for TCP/IP and operate in a relatively similar fashion, albeit with significantly different implementations, since they sit atop UDP.

TCP provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network. It is connection-oriented and employs a congestion control algorithm that aims to make the most of shared communication resources without overloading them.

Network Coding for Engineers, First Edition. Muriel Médard, Vipindev Adat Vasudevan, Morten Videbæk Pedersen and Ken R. Duffy.
© 2025 The Institute of Electrical and Electronics Engineers, Inc. Published 2025 by John Wiley & Sons, Inc.

For each connection, TCP maintains a congestion window (cwnd) limiting the total number of unacknowledged packets that may be in transit end-to-end.

Congestion window mechanisms can be understood in the context of the development of TCP. The original deployments of TCP were primarily over wireline networks where packet transmission was reliable over the communication medium, but where system capacity, in terms of bandwidth and queue size, was limited. In such a setting, packet loss was generally inferred as being indicative of congestion, with packets being delayed excessively, and in some cases being lost to buffer overflows due to having the offered load exceed the transmit capabilities of the system. The buffer overflow losses, often termed droptail, will translate at the sender in having triple-duplicate ACKs, which results in a total of four ACKs, the original and three duplicates, as in Fig. 5.1.

TCP/IP in all of its variants was built without coding in mind. Protocols which were built without coding do not differentiate between seen packets and decoded packets. Effectively, they work with a degenerate type code, a repetition one, that makes **drop-when-seen** the same as **drop-when decoded** even though we have seen in Chapter 4 that those approaches should in general lead to quite different behavior. The core question we face is the following: **how can we leverage protocols that were built with acknowledgment mechanisms that assumed no coding in order to build protocols that are compatible with the original uncoded ones but provide the benefit of coding?**

As we saw in Chapter 4, congestion leads to growth of the queue of order $\rho/(1-\rho)$, which entails that, if ρ nears 1, it is important to reduce it in order to avoid uncontrolled growth of the packet queue at a buffer. Little's law tells us that the waiting time experiences a similar phenomenology, and so increasing delay can also be used to inform impending congestion. TCP seeks to manage that potentially catastrophic growth by stopping the window sliding (no new packets for a while) and by shrinking the window (reducing the number of packets that may become lost by being placed in a buffer overflow situation). This aspect is important to avoid overwhelming the network, so we wish to ensure that coding respects this original design principle.

On the other hand, modern networks have connections such as wireless links where loss may not be indicative of congestion. As a result, treating a lost or badly delayed packet as being indicative of congestion may result in a sub-optimal control response. Here we wish to ensure that we can leverage coding and its benefits. In some systems, the network seeks to assess whether losses arose from congestion or other causes. Such schemes, which use so-called explicit mechanisms, can be fairly readily modified to use coding. However, they remain relatively rare. Hence, we shall concentrate in this chapter and the next on introducing coding into protocols that have no means or assessing the root cause of a packet loss.

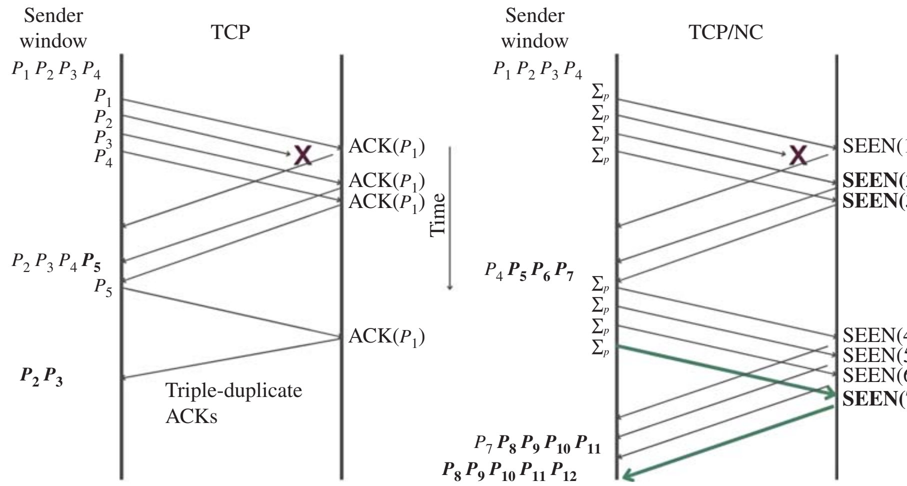


Figure 5.1 Example of TCP, coding, and ACKs. Source: This figure is inspired by joint work with MinJi Kim [1].

We explore how network coding goes with these communication protocols and if a network coding-specific protocol is necessary. We present the logical description of our Network Coding with TCP (TCP/NC) protocol, followed by a way to implement these ideas with as little disturbance as possible to the existing protocol stack. We start with a brief description of a couple of common TCP variants.

5.1 TCP Variants

TCP Reno is generally regarded as the standard TCP algorithm [2, 3]. TCP Reno increases the congestion window by one segment for every successful acknowledgment received and halves the congestion window when a packet loss is detected. This is the additive increase, multiplicative decrease AIMD scheme. TCP Reno uses fast retransmit and fast recovery to handle packet losses, which means that it retransmits the lost segment after receiving three duplicate ACKs, and reduces the congestion window to half of the previous value plus three segments.

TCP Vegas is an approach that does not rely on packet losses to infer congestion but rather focuses on delay [4]. The crux of the algorithm is to estimate the **base round-trip time**, the RTT in the **absence of congestion**, and use this information to find the discrepancy between the expected and actual transmission rate. It increases the congestion window by one unit for every RTT if the difference is less than a threshold and decreases the congestion window by one unit for every RTT if the difference is greater than a threshold. TCP Vegas also implements a slow start algorithm that increases the congestion window by one unit for every ACK received, but stops when the difference exceeds yet another threshold.

The basic design of TCP Vegas is to garner an estimate of congestion without waiting for it to happen first, as that would result in wasteful packet drops, but instead to infer congestion based on growing RTT. The **base RTT** is intended to be the time it would take to get a packet through the network in the absence of congestion. It is typically estimated by recording the **minimum RTT** observed so far. With cwnd being the congestion window (i.e. number of IP packets in flight that are not yet acknowledged), TCP Vegas aims to control cwnd to ensure that the difference between **expected output** and the **actual output** times the base RTT, the **extra data in the network**

$$D = \text{cwnd}\left(1 - \frac{\text{base RTT}}{\text{current RTT}}\right)$$

is kept within given bounds., d_1 and d_2, say, 1 and 3. At each update,

$$\text{cwnd} = \begin{cases} \text{cwnd} & \text{if } D \in [d_1, d_2] \\ \text{cwnd} + 1 & \text{if } D > d_1 \\ \text{cwnd} - 1 & \text{if } D < d_2 \end{cases}$$

5.2 Logical Description of TCP/NC

The main aim of this algorithm is to mask **non-congestion-induced** losses from TCP using random linear coding. There are some important modifications in order to incorporate coding. First, instead of the original packets, we transmit random linear combinations of packets in the congestion window. While such coding helps with erasure correction, it also leads to a problem in acknowledging data, given that the original ACK mechanism was designed for an uncoded scheme.

TCP operates with units of packets,[1] which have a well-defined ordering. Thus, the packet sequence number can be used to acknowledge the received data. The unit in our protocol is a **DoF**. However, when packets are coded together, there is no clear ordering of the DoFs that can be used for ACKs. The protocol we present here addresses and presents the solution to this problem. The notion of **seen packets** we have explored in Chapter 4 proves to be crucial, as it defines an ordering of the degrees of freedom that is consistent with the packet sequence numbers, and can therefore be used to acknowledge DoFs.

Upon receiving a linear combination, the receiver finds out which packet, if any, has been newly seen because of the new arrival and acknowledges that packet using the TCP/IP mechanism. The receiver thus pretends to have received the packet as though using uncoded TCP/IP even if it cannot be decoded yet. This is not a difficulty because if all the packets in a file have been seen, then they can all be decoded as well.

The idea of transmitting random linear combinations and acknowledging seen packets achieves our goal of masking **non-congestion** losses from TCP as follows. With a large field size, every random linear combination is highly likely to cause the next unseen packet to be seen. Hence, even if a transmitted linear combination is lost, the next successful reception of a (possibly) different random linear combination will cause the next unseen packet to be seen and acknowledged.

From the TCP sender's perspective, this appears as though the transmitted packet waits in a fictitious queue until the channel stops erasing packets and allows it through. Thus, there will never be any duplicate ACKs. Every ACK will cause the congestion window to advance. In short, **the lossiness of the link is presented to TCP as an additional queueing delay that leads to a larger effective round-trip time**.

The term **round-trip time** thus has a new interpretation. It is the effective time the network takes to **reliably** deliver a degree of freedom (including the delay

1 Actually, TCP operates in terms of bytes. For simplicity of presentation, the present section uses packets of fixed length as the basic unit. All the discussion in this section extends to the case of bytes as well.

for the coded redundancy, if necessary), followed by the return of the ACK. This is larger than the true network delay it takes for a lossless transmission and the return of the ACK. The more lossy the link is, the larger will be the effective RTT. Presenting TCP with a larger value for RTT may seem counter intuitive as TCP's rate is inversely related to RTT. However, if done correctly, it improves the rate by preventing loss-induced window closing, as it gives the network more time to deliver the data in spite of losses, before TCP times out. Therefore, **non-congestion** losses are effectively masked.

5.3 Seen Packets and Congestion Control

How does the coded packet approach affect congestion control? Since we mask losses from the congestion control algorithm, the TCP-Reno style approach to congestion control using packet loss as a congestion indicator is not immediately applicable to this situation. However, the congestion-related losses are made to appear as a longer RTT. Therefore, we can use an approach that infers congestion from an increase in RTT. The natural choice is TCP-Vegas. The discussion in this section is presented in terms of TCP-Vegas. The algorithm, however, can be extended to make it compatible with TCP-Reno as well.

In order to use TCP-Vegas correctly in this setting, we need to ensure that it uses the effective RTT of a DoF, including the fictitious queueing delay. In other words, the RTT should be measured from the point when a packet is first sent out from TCP, to the point when the ACK returns saying that this packet has been seen. This is indeed the case if we simply use the default RTT measurement mechanism of TCP-Vegas. The TCP sender notes down the transmission time of every packet. When an ACK arrives, it is matched to the corresponding transmit timestamp in order to compute the RTT. Thus, no modification is required.

Consider the example shown in Figure 5.1. The representation is typical of that used in the literature of protocols, Time advances from top to bottom. Transmissions are indicated by arrows, where the angle of the arrow is determined by the delay. A lower delay will lead to an arrow closer to horizontal, and a longer delay will lead the arrow to be closer to vertical. The transmitter is generally indicated as a vertical line at the left-hand side, representing time as seen at the sender. The receiver is represented as a vertical line on the right-hand side, indicating the time at the receiver.

On the left-hand side of Figure 5.1, we see the operation of a characteristic version of TCP. The packets are acknowledged when received. In reality, an acknowledgment of packet i is really instantiated as a request for packet $i + 1$. Transmission of packets is done over a window, which consists of the packets that are currently being considered for current transmission or retransmission. Generally, a window will increase as the number of packets in flight increases, since packets that are not yet acknowledged at the receiver will be in the window. Packets in flight will increase when the delay in either direction (sender to receiver for packets or receiver to sender for ACKs) increases, or when the rate of packets, say packets per second, goes up. The rate of packets transmitted is often, by abuse of nomenclature, named the bandwidth, as physical bandwidth, measured in Hertz, allows the rate to scale up roughly proportionally. Thus, the Bandwidth-Delay Product (BDP), which is the delay (actually the RTT) in seconds multiplied by the rate (in packets per second) is an important element of TCP window size selection. The operation of TCP allows the window to slide as acknowledgments are received. This sliding window style of operation means that packets are being continuously shed from the window and included in the window.

To visualize in a simple example the operation of TCP when network coding and the use of seen packets is introduced, consider the right-hand side of Fig. 5.1. Suppose the congestion window's length is 4. Assume TCP sends four packets to the network coding layer at $t = 0$. All four transmissions are linear combinations of these four packets. The first transmission causes the first packet to be seen. The second transmission is lost, and the third transmission causes the second packet to be seen (the discrepancy is because of losses). As far as the RTT estimation is concerned, transmissions 2 and 3 are treated as attempts to convey the second degree of freedom. The RTT for the second packet must include the final attempt that successfully delivers the second degree of freedom, namely the third transmission. In other words, the RTT is the time from $t = 0$ until the time of reception of the third ACK. The core idea is that the ACK mechanism of TCP is employed so that an ACK from TCP is actually an ACK for seeing that packet, in line with **decode-when-seen** in Chapter 4.

5.4 Mechanisms of Use of Seen Packets

Two mechanisms whereby network coding with a sliding window benefits TCP are already evident from this discussion. The first is that the window does not cease to slide as with triple-duplicate ACKs. If the losses are random, say from an occasional mishap in the transmission rather than a protracted congestion, then such an approach is quite beneficial. We should note that some TCP protocols, such as Cubic, do allow what is generally termed selective ARQ, which is more in line with the type of retransmissions that we overviewed in Chapter 4, where we proceed to repeating packets that were not received rather than taking the actions described above for TCP. However, such systems generally allow only a few packets to be lost by only having a handful of packets in the process of selective ACK. Thus, if the BDP is high, even modest losses of the order of a percent or so, which is a regime that is not atypical in many wireless settings, will exceed the ability of the protocol to manage selective ARQ.

The second mechanism through which network coding with a sliding window is helpful lies in the seen packets. The ability to manage losses through coding might at first appear to be manageable, rather than by a sliding window that advances through learning of seen packets, through a code such as a rateless code, where an ACK was produced after a set of original packets was sent. This coding-until-completion approach we considered in Chapter 4 and has the issues we discussed before.

For the rateless code, we would not be able to advance the window until enough packets to decode were received. We know already from Chapter 4 that, if instead of managing the queue in terms of DoFs as known at the sender from the receiver via the latter's acknowledgment of seen packet, we would have a queue that would grow as $\rho/(1-\rho)^2$. This queue growth would heighten the incidence of droptail events.

Another approach would be through a block code with a given number of repair, or redundant packets, where the redundancy was tailored to the expected rate of packet losses. However, in those cases we would not have seen packets. For the block code, we would not have a window progression mechanism except at the level of a block. If the entire block was successfully decoded, then the window could advance block by block; otherwise, it could not advance. In some cases, too much redundancy would have been transmitted, in a preemptive fashion, for example, if no losses occurred. In other cases, insufficient equations for the losses experienced by a block would lead to decoding failure and the need to retransmit a whole other block, which is wasteful of resources unless all of the packets in the previously undecoded block were lost. Thus, despite many attempts to combine rateless or block codes with TCP and related protocols, to our knowledge the only successful approaches necessitate a sliding window network coding technique.

Because a sliding window requires a code that is adapted to any window and pattern of losses, RLNC is the natural approach to code with a sliding window. A structured code that would require a different construction tailored to each of the myriad patterns of losses would be altogether unsuitable.

The implementation of the sliding window network coding ideas in the existing protocol stack needs to be done in as non-intrusive a manner as possible. We present a solution that embeds the network coding operations in a separate layer below TCP and above IP on the source and receiver side, as shown in Figure 5.2. The operation of these modules is described next.

The sender module accepts packets from the TCP source and buffers them into an encoding buffer which represents the coding window,[2] until they are ACKed by

2 Whenever a new packet enters the TCP congestion window, TCP transmits it to the network coding module, which then adds it to the coding window. Thus, the coding window is related to

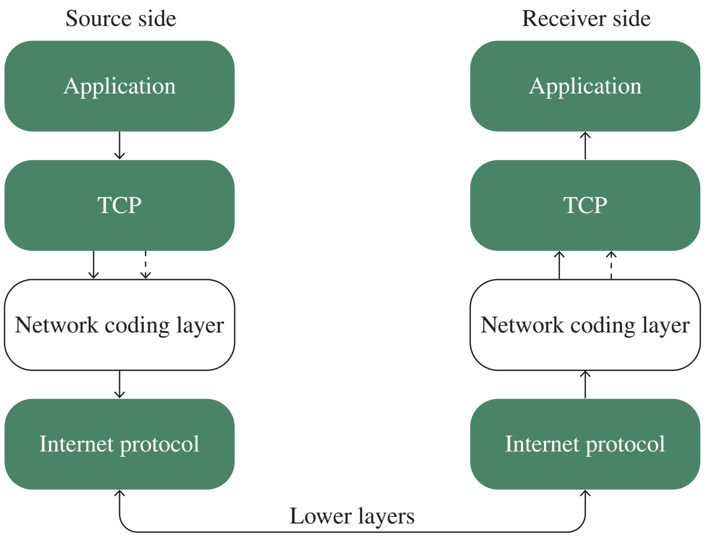


Figure 5.2 New network coding layer in the protocol stack. This figure is inspired by joint work with Jay Kumar Sundararajan [5].

the receiver. The sender then generates and sends random linear combinations of the packets in the coding window. The coefficients used in the linear combination are also conveyed in the header.

For every packet that arrives from TCP, R linear combinations are sent to the IP layer on average, where R is the redundancy, or repair, parameter. The average rate at which linear combinations are sent into the network is thus a constant factor more than the rate at which TCP's congestion window progresses. This is necessary in order to compensate for the loss rate of the channel and to match TCP's sending rate to the rate at which data is actually sent to the receiver. If there is too little redundancy, then the data rate reaching the receiver will not match the sending rate because of the losses. This leads to a situation where the losses are not effectively masked from the TCP layer. Hence, there are frequent timeouts, leading to a low throughput. At the other extreme, too much redundancy is also bad, since then the transmission rate becomes limited by the unused repair generated by the code itself. Sending too many linear combinations can congest the network. The ideal level of redundancy is to keep R of the order of the reciprocal of the probability of successful reception, with some extra margin to account for variability.

the TCP layer's congestion window but generally not identical to it. For example, the coding window will still hold packets that were transmitted earlier by TCP, but are no longer in the congestion window because of a reduction of the window size by TCP. Involving more packets in a linear combination will only increase its chances of being innovative.

Thus, in practice, the value of R should be dynamically adjusted by estimating the loss rate, possibly using the RTT estimates.

Upon receiving a linear combination, the receiver module first retrieves the coding coefficients from the header and appends them to the basis matrix of its knowledge space. Then, it performs a Gauss–Jordan elimination as we have seen in the last chapter, to find out which packet is newly seen so that this packet can be ACKed. The receiver module also maintains a buffer of linear combinations of packets that have not been decoded yet. Upon decoding the packets, the receiver module delivers them to the TCP sink.

The algorithm is specified below using pseudo-code. This specification assumes a one-way TCP flow.

5.4.1 Source Side

The source-side algorithm has to respond to two types of events – the arrival of a packet from the source TCP and the arrival of an ACK from the receiver via IP.

```
1) Set NUM to 0.
2) Wait state: If any of the following events occurs, respond as follows; else, wait.
3) Packet arrives from TCP sender:
   a) If the packet is a control packet used for connection management, deliver it
      to the IP layer and return to the wait state.
   b) If the packet is not already in the coding window, add it to the coding
      window.
   c) Set NUM = NUM + $R$. ($R$ = redundancy factor)
   d) Repeat the following $\lfloor \text{NUM} \rfloor$ times:
        i) Generate a random linear combination of the packets in the coding
           window.
       ii) Add the network coding header specifying the set of packets in the cod-
           ing window and the coefficients used for the random linear combination.
      iii) Deliver the packet to the IP layer.
   e) Set NUM = fractional part of NUM.
   f) Return to the wait state.
4) ACK arrives from receiver: Remove the ACKed packet from the coding
   buffer and hand over the ACK to the TCP sender.
```

5.4.2 Receiver Side

On the receiver side, the algorithm again has to respond to two types of events: the arrival of a packet from the source and the arrival of ACKs from the TCP sink.

```
1) Wait state: If any of the following events occurs, respond as follows; else, wait.
```

2) **ACK arrives from TCP sink:** If the ACK is a control packet for connection management, deliver it and return to the wait state; else, ignore the ACK.
3) **Packet arrives from source side:**
   a) Remove the network coding header and retrieve the coding vector.
   b) Add the coding vector as a new row to the existing coding coefficient matrix, and perform Gauss–Jordan elimination to update the set of seen packets.
   c) Add the payload to the decoding buffer. Perform the operations corresponding to the Gauss–Jordan elimination, on the buffer contents. If any packet gets decoded in the process, deliver it and remove it from the buffer.
   d) Generate a new TCP ACK with sequence number equal to that of the oldest unseen packet.

5.4.3 Soundness of the Protocol

We argue that this protocol guarantees reliable transfer of information. In other words, every packet in the packet stream generated by the application at the source will be delivered eventually to the application at the sink. We observe that the acknowledgment mechanism ensures that the coding module at the sender does not remove a packet from the coding window unless it has been ACKed, i.e. unless it has been seen by the sink. Thus, we only need to argue that if all packets have been seen, then the packets can be decoded.

Property 5.1 From a file of n packets, if every packet has been seen, then every packet can also be decoded.

If the sender knows a file of n packets, then the sender's knowledge space is of dimension n. Every seen packet corresponds to a new DoF. Hence, if all n packets have been seen, then the receiver's knowledge space is also of dimension n, in which case it must be the same as the sender's. All packets can be decoded in that case.

In other words, seeing n different packets corresponds to having n linearly independent equations, or DoFs, in n unknowns. Hence, the unknowns can be found by solving the system of equations. In practice, one does not have to necessarily wait until all packets are seen to decode all packets. Some of the unknowns can be found even along the way. In particular, whenever the number of equations received catches up with the number of unknowns involved, the unknowns can be found. Now, for every new equation received, the receiver sends an ACK. The congestion control algorithm uses the ACKs to control the injection of new unknowns into the coding window. Thus, the discrepancy between the number of equations and the number of unknowns does not tend to grow with time, and therefore will hit zero often based on the channel conditions. As a consequence, the decoding buffer will tend to be stable.

5.5 Queueing Analysis for an Idealized Case of TCP/NC

In this section, we focus on an idealized scenario in order to provide a first-order analysis of our new protocol. We aim to explain the key ideas of TCP/NC protocol with an emphasis on the interaction between the coding operation and the feedback. The model used in this section will also serve as a platform which we can build on to incorporate more practical situations.

We abstract out the congestion control aspect of the problem by assuming that the capacity of the system is fixed in time and known at the source, and hence the arrival rate is always maintained below the capacity. We also assume that nodes have infinite capacity buffers to store packets.

We focus on a topology that consists of a chain of erasure-prone links in tandem, with perfect end-to-end feedback from the sink directly to the source. In such a system, the behavior of the queue sizes at various nodes is investigated to show that the queues for all rates below capacity are stabilized.

5.5.1 System Model

The network we study in this section is a daisy chain of N nodes, each node being connected to the next one by a packet erasure channel, as shown in Figure 5.3. We assume a slotted time system. The source generates packets according to a Bernoulli process of rate λ packets per slot. The point of transmission is at the very beginning of a slot. Just after this point, every node transmits one random linear combination of the packets in its queue. The relation between the transmitted linear combination and the original packet stream is conveyed in the packet header. We ignore this overhead for the analysis in this section. We ignore propagation delay as well. Thus, the transmission, if not erased by the channel, reaches the next node in the chain almost immediately. However, the node may use the newly received packet only in the next slot's transmission. We assume perfect, delay-free feedback from the sink to the source. In every slot, the sink generates the feedback signal after the instant of reception of the previous node's transmission. The erasure event happens with a probability $(1 - \mu_i)$ on the channel connecting node i and $(i + 1)$, and is assumed to be independent across different channels and over time. Thus, the system has a capacity $\min_i \mu_i$ packets per slot. We assume that $\lambda < \min_i \mu_i$, and define the load factor $\rho_i = \lambda/\mu_i$.

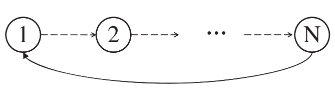


Figure 5.3 Topology: Daisy chain with perfect end-to-end feedback.

5.5.2 Queue Update Mechanism

Each node transmits a random linear combination of the current contents of its queue and hence, it is important to specify how the queue contents are updated at the different nodes. Queue updates at the source are relatively simple because, in every slot, we assume the end receiver sends an ACK directly to the source, containing the index of the oldest packet not yet seen by the sink. Upon receiving the ACK, the source simply drops all packets from its queue with an index lower than the sink's feedback.

Whenever an intermediate node receives an innovative packet, this causes the node to see a previously unseen packet. The node performs Gauss–Jordan elimination on the matrix with the new packet appended. The last row of the matrix after the Gauss–Jordan elimination we call the **witness**. This is essentially the new information from the received packet and will be used to create new coded packets at the intermediate node.

Each node computes the witness of the newly seen packet and adds this to the queue. Thus, intermediate nodes store the witnesses of the packets that they have seen. The idea behind the packet drop rule is similar to that at the source – an intermediate node may drop the witnesses of packets up to but excluding what it believes to be the receiver's, or sink's, first unseen packet with regards to the source transmitted packets, based on its knowledge of the sink's status at that point in time.

The intermediate nodes, in general, may only know an outdated version of the sink's status because we assume that the intermediate nodes do not have direct feedback from the sink (see Fig. 5.3). Instead, the source has to inform them about the sink's ACK through the same erasure channel used for the regular forward transmission. This feed-forward of the sink's status is modeled as follows.

Whenever the channel entering an intermediate node is in the ON state (i.e. no erasure), the node's version of the sink's status is updated to that of the previous node. In practice, the source need not transmit the sink's status explicitly. The intermediate nodes can infer it from the set of packets that have been involved in the linear combination – if a packet is no longer involved, that means the source must have dropped it, implying that the sink must have ACKed it already.

Remark 5.1 This model and the following analysis also work for the case when not all intermediate nodes are involved in the network coding. If some node simply forwards the incoming packets, then an erasure event on either the link entering this node or the link leaving this node will cause a packet erasure. These two links can be replaced by a single link whose probability of being ON is simply the product of the ON probabilities of the two links being replaced. Thus, all non-coding nodes can be removed from the model, which brings us back to the same situation as in the above model.

5.5.3 Queueing Analysis

We now analyze the size of the queues at the nodes under the queueing policy described above. The following theorem shows that if we allow coding at intermediate nodes, then it is possible to achieve the capacity of the network, namely $\min_k \mu_k$. In addition, it also shows that the expected queue size in the heavy-traffic limit ($\lambda \to \min_k \mu_k$) has an asymptotically optimal linear scaling in $1/(1-\rho_k)$.

Note that, if we only allow forwarding at some of the intermediate nodes, then we can still achieve the capacity of a new network derived by collapsing the links across the non-coding nodes, as described in Remark 5.1.

Property 5.2 As long as $\lambda < \mu_k$ for all $0 \le k < N$, the queues at all the nodes will be stable. The expected queue size in steady state at node $k (0 \le k < N)$ is given by:

$$\mathbb{E}[Q_k] = \sum_{i=k}^{N-1} \frac{\rho_i(1-\mu_i)}{(1-\rho_i)} + \sum_{i=1}^{k-1} \rho_i$$

An implication: Consider a case where all the ρ_i's are equal to some ρ. Then, the above relation implies that in the limit of heavy traffic, i.e. $\rho \to 1$, the queues are expected to be longer at nodes near the source than near the sink.

A useful lemma: The above property will be proved after the following one, which shows that the random linear coding scheme has the property that every time there is a successful reception at a node, the node sees the next unseen packet with high probability, provided the field is large enough. This fact will prove useful while analyzing the evolution of the queues.

Property 5.3 Let S_A and S_B be the set of packets seen by two nodes A and B, respectively. Assume $S_A \backslash S_B$ is non-empty. Suppose A sends a random linear combination of its witnesses of packets in S_A and B receives it successfully. The probability that this transmission causes B to see the oldest packet in $S_A \backslash S_B$ is $O\left(1 - \frac{1}{q}\right)$, where q is the field size.

Proof: Let M_A be the RREF basis matrix for A. Then, the coefficient vector of the linear combination sent by A is $\mathbf{t} = \mathbf{u}M_A$, where $\mathbf{u}$ is a vector of length $|S_A|$ whose entries are independent and uniformly distributed over the finite field $\mathbb{F}_q$. Let d^* denote the index of the oldest packet in $S_A \backslash S_B$.

Let M_B be the RREF basis matrix for B before the new reception. Suppose $\mathbf{t}$ is successfully received by B. Then, B will append $\mathbf{t}$ as a new row to M_B and perform Gaussian elimination. The first step involves subtracting from $\mathbf{t}$, suitably scaled versions of the pivot rows such that all entries of $\mathbf{t}$ corresponding to pivot columns of M_B become 0. We need to find the probability that after this step, the leading non-zero entry occurs in column d^*, which corresponds to the event that B sees packet d^*. Subsequent steps in the Gaussian elimination will not affect this event. Hence, we focus on the first step.

Let P_B denote the set of indices of pivot columns of M_B. In the first step, the entry in column d^* of $\mathbf{t}$ becomes

$$t'(d^*) = t(d^*) - \sum_{i \in P_B, i<d^*} t(i) \cdot M_B(r_B(i), d^*),$$

where $r_B(i)$ is the index of the pivot row corresponding to pivot column i in M_B. Now, due to the way RREF is defined, $t(d^*) = u(r_A(d^*))$, where $r_A(i)$ denotes the index of the pivot row corresponding to pivot column i in M_A. Thus, $t(d^*)$ is uniformly distributed. Also, for $i < d^*$, $t(i)$ is a function of only those $u(j)$'s such that $j < r_A(d^*)$. Hence, $t(d^*)$ is independent of $t(i)$ for $i < d^*$. From these observations and the above expression for $t'(d^*)$, it follows that for any given M_A and M_B, $t'(d^*)$ has a uniform distribution over $\mathbb{F}_q$, and the probability that it is not zero is therefore $\left(1 - \frac{1}{q}\right)$.

For the queueing analysis, we assume that a successful reception always causes the receiver to see its next unseen packet, as long as the transmitter has already seen it. The above property argues that this assumption becomes increasingly valid as the field size increases. In reality, some packets may be seen out of order, resulting in larger queue sizes. However, we believe that this effect is minor and can be neglected for a first-order analysis.

With this assumption in place, the queue update policy described earlier implies that the size of the physical queue at each node is simply the difference between the number of packets the node has seen and the number of packets it believes the sink has seen.

To study the queue size, we define a virtual queue at each node that keeps track of the DoF backlog between that node and the next one in the chain. The arrival and departure of the virtual queues are defined as follows. A packet is said to arrive at a node's virtual queue when the node sees the packet for the first time. A packet is said to depart from the virtual queue when the next node in the chain sees the packet for the first time. A consequence of the assumption stated above is that the set of packets seen by a node is always a contiguous set. This allows us to view the virtual queue maintained by a node as though it were a First In – First Out (FIFO) queue. The size of the virtual queue is simply the difference between the number of packets seen by the node and the number of packets seen by the next node downstream.

We are now ready to explain why the property 5.2 holds. For each intermediate node, we study the expected time spent by an arbitrary packet in the physical queue at that node, as this is related to the expected physical queue size at the node, by Little's law.

Consider the kth node, for $1 \leq k < N$. The time a packet spends in this node's queue has two parts:

(1) *Time until the packet is seen by the sink:*

The virtual queue at a node behaves like a queue as in [6]. Given that node k has just seen the packet in question, the additional time it takes for the next node to see that packet corresponds to the waiting time in the virtual queue at node k. For a load factor of ρ and a channel ON probability of μ, the expected waiting time was derived in [6] to be $\frac{(1-\mu)}{\mu(1-\rho)}$, using results from [7]. Now, the expected time until the sink sees the packet is the sum of $(N-k)$ such terms, which gives

$$\sum_{i=k}^{N-1} \frac{(1-\mu_i)}{\mu_i(1-\rho_i)}.$$

(2) *Time until sink's ACK reaches intermediate node:*

The ACK informs the source that the sink has seen the packet. This information needs to reach node k by the feed-forward mechanism. The expected time for this information to move from node i to node $i+1$ is the expected time until the next slot when the channel is ON, which is just $\frac{1}{\mu_i}$ (since the ith channel is ON with probability μ_i). Thus, the time it takes for the sink's ACK to reach node k is given by

$$\sum_{i=1}^{k-1} \frac{1}{\mu_i}.$$

The total expected time T_k a packet spends in the queue at the kth node ($1 \leq k < N$) can thus be computed by adding the above two terms. Now, assuming the system is stable (i.e. $\lambda < \min_i \mu_i$), we can use Little's law to derive the expected queue size at the kth node, by multiplying T_k by λ:

$$\mathbb{E}[Q_k] = \sum_{i=k}^{N-1} \frac{\rho_i(1-\mu_i)}{(1-\rho_i)} + \sum_{i=1}^{k-1} \rho_i.$$

Exercise 5.1

Consider a network consisting of three nodes in a daisy-chain. Describe the evolution of queues when we have feedback based on degrees of freedom, using random network coding, and we seek to send a finite length file from node 1 to node 3.

5.6 Performance Analysis

Now, we consider this TCP/NC protocol to do a performance analysis and understand how to do a similar analysis to other network-coding-inspired protocols. We consider a lossy network and extend the performance evaluation of a TCP model in [8], which provides a simple yet good model to predict the performance of TCP. We focus on the steady-state throughput behavior of both TCP and TCP/NC as a function of erasure rate, RTT, and maximum window size. We'll see what differences it makes to use a TCP/NC protocol and how some of these parameters and their implications change.

We measure users' quality of experience using the throughput perceived by the user or the application, i.e. **goodput**. We make a clear distinction between the terms *goodput* and *throughput*, where goodput is the number of *useful* bits over unit time received by the user and throughput is the number of bits transmitted by the base station per unit time. In essence, throughput is indicative of the bandwidth/resources provisioned by the service providers; while goodput is indicative of the user's quality of experience. For example, the base station, after accounting for the FEC overhead, may be transmitting bits at 10 Mbps, i.e. throughput is 10 Mbps. However, the user may only receive useful information at 5 Mbps, i.e. goodput is 5 Mbps. This disparity is so obvious and common when we consider lossy networks. Even a 1–3% packet loss rate can significantly degrade the performance of TCP by reducing its goodput. TCP/NC with its forward erasure correction capabilities provides better goodput in lossy environments. Furthermore, the loss patterns and RTT may impact the goodput if you aim for reliable communication, especially if there are more losses in the tail. Network coding addresses this as well with the forward erasure correction.

Let's look at the overheads that are inherent to network coding. For a receiver to decode a network coded packet, the packet needs to indicate the coding coefficients used to generate the linear combination of the original data packets. So these coefficients need to be known to the receiver. Furthermore, the encoder uses these coefficients to generate the linear combinations of the message, called codewords, and the receiver needs to inverse this process to find the original messages.

The communication overhead associated with the coefficients depends on the field size used for coding as well as the number of original packets combined. We already saw in Chapter 4 that we do not need a large field size to operate. A field size of 2^8 is good enough for most practical applications. Thus each coefficient symbol is of 8 bits or 1 byte. Therefore, even if we combine 50 original packets, the coding coefficients amount to 50 bytes overall. Note that a packet is typically around 1500 bytes. Therefore, the overhead associated with coding vector is not substantial. Furthermore, in advanced settings, a *seed* can be used at the source to create the random coefficients, using a coefficient generator algorithm, and share only this seed instead of the whole set coding coefficients for reducing the overhead. The receiver could use the same seed and the same coefficient generator algorithm to recreate the coding coefficient once received. This approach is much more efficient in terms of communication overhead but has limitations when it comes to recoding at intermediate nodes.

The second overhead associated with network coding is the encoding and decoding complexity, and the delay associated with the coding operations. Note that to affect TCP's performance, the decoding/encoding operations must take a substantial amount of time to affect the RTT estimate of the TCP sender and receiver. However, we note that the delay caused by the coding operations is negligible compared to the network RTT. For example, the network RTT is often in milliseconds (if not in hundreds of milliseconds), while encoding/decoding operations involve a matrix multiplication/inversion in $\mathbf{F}_{256}$ which can be performed in a few microseconds.

5.6.1 A Model for Congestion Control

First, we discuss the congestion avoidance mechanism employed by TCP, which entails adjusting the congestion control window size W based on received acknowledgments (ACKs). Specifically, the window size W increments by $1/W$ for each ACK, effectively increasing it to $W + 1$ when all packets within the congestion control window are acknowledged. On the other hand, upon detecting erasure or congestion, the window size W is reduced.

These operations are observed as in *rounds* as outlined in [8]. Here, W_i represents the congestion control window size at the onset of round i. At the start of each round, the sender transmits W_i packets from its congestion window. Subsequently, transmission halts until at least one ACK for these packets is received, marking the conclusion of the current round and the commencement of round $i + 1$.

For simplicity, we assume each round lasts a RTT, irrespective of W_i, under the assumption that packet transmission time is negligible compared to RTT. This simplification delineates the events within each round i: initial transmission of W_i packets, potential packet losses, reception of cumulative ACKs by the receiver (wherein missing packets are indicated by repeated ACKs, i.e. if the packets $1, 2, 3, 5, 6$ arrive at the receiver in sequence, then the receiver ACKs packets $1, 2, 3, 3, 3$. This signals that it has not yet received packet 4.). It is also possible that some ACKs are lost. Upon receiving ACKs, the sender adjusts its window size. Assuming a_i packets are acknowledged in round i, the update rule is represented as $W_{i+1} \leftarrow W_i + a_i/W_i$.

We see how the congestion window increments, but how does this help for congestion control and when is the congestion window reduced? There are two situations where TCP reduces its window size for congestion control.

1) **Triple-duplicate (TD):** When the sender receives four ACKs with the same sequence number, then $W_{i+1} \leftarrow \frac{1}{2}W_i$.
2) **Time-out (TO):** If the sender does not hear from the receiver for a predefined time period, called the "time-out" period (which is T_o rounds long), then the sender closes its transmission window, $W_{i+1} \leftarrow 1$. At this point, the sender updates its TO period to $2T_o$ rounds and transmits one packet. For any subsequent TO events, the sender transmits the one packet within its window, and doubles its TO period until $64T_o$ is reached, after which the TO period is fixed to $64T_o$. Once the sender receives an ACK from the receiver, it resets its TO period to T_o and increments its window according to the congestion avoidance mechanism. During time-out, the throughput of both TCP and TCP/NC is zero.

In practice, TCP may not send ACKs for every packet but sends a cumulative ACK for some number of packets, say 2. However, here we consider an ACK is sent for every packet for simplicity of analysis.

Typically, TCP operates within the confines of a maximum window size denoted as $W_{\max}$. Till this limit is reached, the TCP sender employs a congestion avoidance strategy to increment the window size. Upon reaching $W_{\max}$, the window size is held constant at this maximum value until either a TD or a TO event occurs.

We incorporate random packet erasures, characterized by a probability p, representing the likelihood of a packet being lost at any given time. For this analysis, we assume these losses occur independently, without correlation between successive losses.

Let us now analyze the performance of TCP and TCP/NC in terms of two metrics: the average throughput $\mathcal{T}$, and the expected window evolution $E[W]$, where $\mathcal{T}$ represents the total average throughput while window evolution $E[W]$ reflects the perceived throughput at a given time. We define $\mathcal{N}_{[t_1,t_2]}$ to be the number of packets received by the receiver during the interval $[t_1, t_2]$. The total average throughput is defined as:

$$\mathcal{T} = \lim_{\Delta\to\infty} \frac{\mathcal{N}_{[t,t+\Delta]}}{\Delta}. \tag{5.1}$$

We denote $\mathcal{T}_{\text{tcp}}$ and $\mathcal{T}_{\text{nc}}$ to be the average throughput for TCP and TCP/NC, respectively.

Exercise 5.2

Model TCP and TCP/NC in any of the tools that you are interested in. Analyze the window evolution for 5–6 different error probabilities between 0.01 and 0.5 using Monte Carlo simulations. How often does a TD happen? What about TO? Compare with the theoretical analysis. Vary the error pattern to model more bursty losses. Try to identify different situations of TD and TO.

5.6.2 Intuition

In traditional TCP, random erasures within the network can trigger triple-duplicate ACKs. For instance, in Fig. 5.4a, during round i, the sender transmits W_i packets, but only a_i of them reach the receiver. Consequently, the receiver

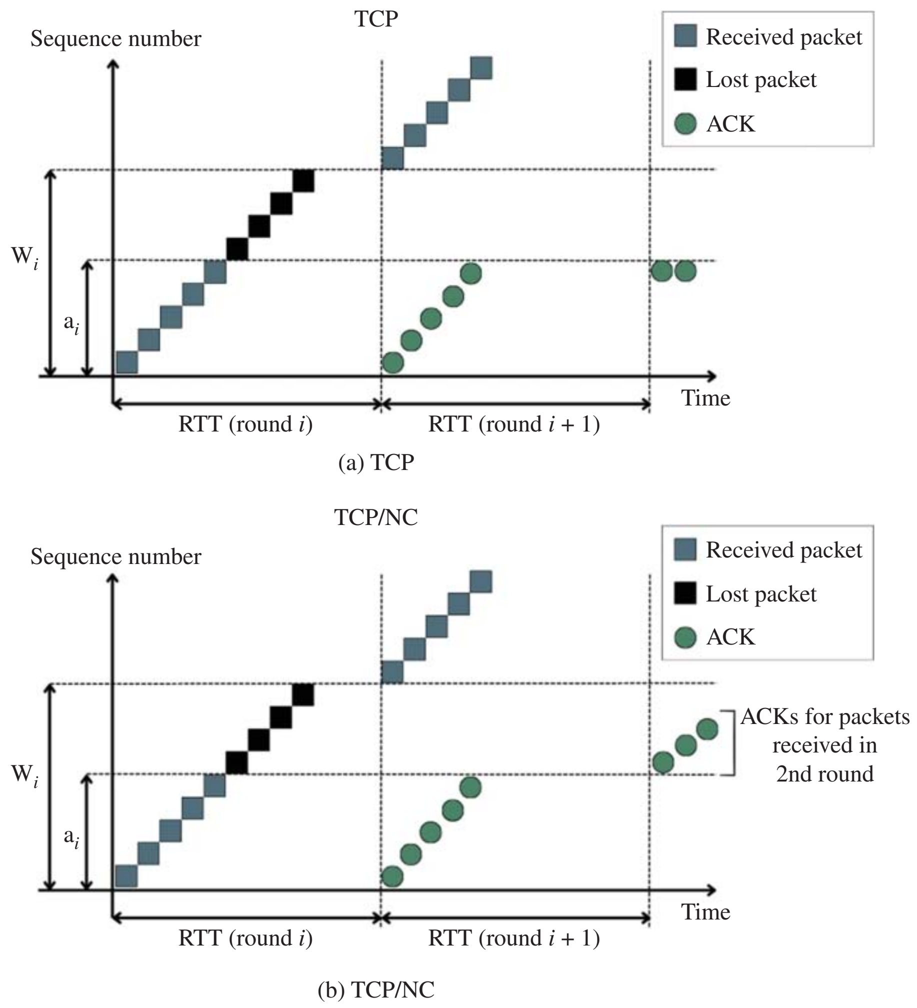


Figure 5.4 The effect of erasures: TCP experiences triple-duplicate ACKs, and results in $W_{i+2} \leftarrow W_{i+1}/2$. However, TCP/NC masks the erasures using network coding, which allows TCP to advance its window. This figure depicts the sender's perspective; therefore, it indicates the time at which the sender transmits the packet or receives the ACK. This figure is inspired by joint work with MinJi Kim [1] and by [8].

acknowledges the arrival of the a_i packets and awaits packet $a_i + 1$. Upon receiving these ACKs, round $i + 1$ commences at the sender. This new round has a window size ($W_{i+1} \leftarrow W_i + a_i/W_i$) and initiates transmission of new packets within the window. However, as the receiver is still expecting packet $a_i + 1$, any additional packets prompt the receiver to request packet $a_i + 1$ again. This scenario triggers a triple-duplicate ACKs event, leading the TCP sender to close its window (i.e. $W_{i+2} \leftarrow \frac{1}{2}W_{i+1} = \frac{1}{2}(W_i + a_i/W_i)$).

Now let's look at how TCP/NC react to a similar situation. With network coding, each linearly independent packet delivers fresh information. Consequently, any subsequent packet (in Fig. 5.4b, the first packet sent in round $i + 1$) can be interpreted as packet $a_i + 1$. Hence, the receiver can increment its ACK, enabling the sender to continue data transmission. Consequently, network coding conceals network losses from TCP and prevents premature window closure by misinterpreting link losses as congestion. It's noteworthy that TCP/NC operates differently, now looking at the **DOFs**, rather than the original packet itself. With this concept, network coding effectively extends the RTT duration, thereby adjusting the transmission rate to accommodate losses without abruptly closing the window.

5.6.3 Throughput Analysis for TCP*

Even though packet erasures may occur independently, the loss of a single packet can still disrupt the reception of subsequent packets. This stems from TCP's requirement for in-order packet reception. When a packet is lost within a transmission window, all subsequent packets within that window become out of order. Consequently, the TCP receiver discards these out-of-order packets. Consequently, the throughput behavior of standard TCP in the presence of independent losses resembles that described in [8], where losses exhibit correlation within a single round.

We examine the expected throughput between consecutive TD events, depicted in Fig. 5.5. Let's assume the TD events occur at time t_1 and $t_2 = t_1 + \Delta$, where $\Delta > 0$. Additionally, suppose that round j commences immediately after time t_1, and packet loss transpires in the r-th round, denoted as round $j + r - 1$.

First, let's calculate $E[\mathcal{N}_{[t_1,t_2]}]$, the expected number of packets between t_1 and t_2. During the interval $[t_1, t_2]$, there are no packet losses. Given that the probability of a packet loss is p, the expected number of consecutive packets successfully sent from sender to receiver is

$$E\left[\mathcal{N}_{[t_1,t_2]}\right] = \left(\sum_{k=1}^{\infty} k(1-p)^{k-1}p\right) - 1 = \frac{1-p}{p}. \tag{5.2}$$

Note that the out-of-order packets (in white in Fig. 5.5) are not considered as *received*. Packets sent after the lost packets (in black in Fig. 5.5) will not be accepted

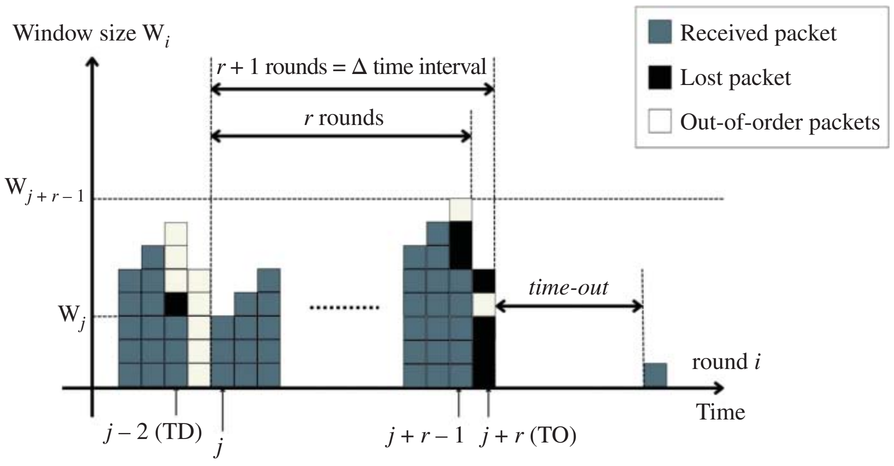


Figure 5.5 TCP's window size with a TD event and a TO event. In round $j - 2$, losses occur resulting in triple-duplicate ACKs. On the other hand, in round $j + r - 1$, losses occur; however, in the following round $j + r$ losses occur such that the TCP sender only receives two-duplicate ACKs. As a result, TCP experiences time-out. This figure is taken from [1], inspired from [8].

by the standard TCP receiver. Thus, Eq. (5.2) does not take into account the packets sent in round $j - 1$ or $j + r$.

First, we determine the anticipated duration between two TD events as $E[\Delta]$. Illustrated in Fig. 5.5, following packet losses in round j, an extra round is needed for the loss feedback from the receiver to reach the sender. Consequently, there exist $r + 1$ rounds within the timeframe $[t_1, t_2]$, and $\Delta = \text{RTT}(r + 1)$. Hence,

$$E[\Delta] = \text{RTT}(E[r] + 1). \tag{5.3}$$

To derive $E[r]$, note that $W_{j+r-1} = W_j + r - 1$ and

$$W_j = \frac{1}{2}W_{j-1} = \frac{1}{2}\left(W_{j-2} + \frac{a_{j-2}}{W_{j-2}}\right). \tag{5.4}$$

Equation (5.4) follows TCP's congestion control. TCP interprets the losses in round $j - 2$ as congestion, and as a result, halves its window. Assuming that, in the long run, $E[W_{j+r-1}] = E[W_{j-2}]$ and that a_{j-2} is uniformly distributed between $[0, W_{j-2}]$,

$$E[W_{j+r-1}] = 2\left(E[r] - \frac{3}{4}\right) \quad \text{and } E[W_j] = E[r] - \frac{1}{2}. \tag{5.5}$$

During these r rounds, we expect to successfully transmit $\frac{1-p}{p}$ packets as noted in Eq. (5.2). This results in:

$$\frac{1-p}{p} = \sum_{k=0}^{r-1} a_{j+k} = \left(\sum_{k=0}^{r-2} W_{j+k}\right) + a_{j+r-1} \tag{5.6}$$

$$= (r-1)W_j + \frac{(r-1)(r-2)}{2} + a_{j+r-1}. \tag{5.7}$$

Taking the expectation of Eq. (5.7) and using Eq. (5.5),

$$\frac{1-p}{p} = \frac{3}{2}(E[r]-1)^2 + E[a_{j+r-1}]. \tag{5.8}$$

Note that a_{j+r-1} is assumed to be uniformly distributed across $[0, W_{j+r-1}]$. Thus, $E[a_{j+r-1}] = E[W_{j+r-1}]/2 = E[r] - \frac{3}{4}$ by Eq. (5.5). Solving Eq. (5.8) for $E[r]$, we find:

$$E[r] = \frac{2}{3} + \sqrt{-\frac{1}{18} + \frac{2}{3}\frac{1-p}{p}}. \tag{5.9}$$

The steady state $E[W]$ is the average window size over two consecutive TD events. This provides an expression of steady-state average window size for TCP (using Eq. (5.5)):

$$E[W] = \frac{E[W_j] + E[W_{j+r-1}]}{2} = \frac{3}{2}E[r] - 1. \tag{5.10}$$

The average throughput can be expressed as

$$\mathcal{T}'_{\text{tcp}} = \frac{E[\mathcal{N}_{[t_1,t_2]}]}{E[\Delta]} = \frac{1-p}{p}\frac{1}{\text{RTT}(E[r]+1)}. \tag{5.11}$$

For small p, $\mathcal{T}'_{\text{tcp}} \approx \frac{1}{\text{RTT}}\sqrt{\frac{3}{2p}} + o(\frac{1}{\sqrt{p}})$; for large p, $\mathcal{T}'_{\text{tcp}} \approx \frac{1}{\text{RTT}}\frac{1-p}{p}$. If we only consider TD events, the long-term steady-state throughput is equal to that in Eq. (5.11).

This analysis assumes unbounded growth of the window size, which may not hold true in practical cases. To accommodate the maximum window size $W_{\max}$, we introduce the following approximation:

$$\mathcal{T}_{\text{tcp}} = \min\left(\frac{W_{\max}}{\text{RTT}}, \mathcal{T}'_{\text{tcp}}\right). \tag{5.12}$$

For small p, this result coincides with the results in [8].

The above analysis does not consider time-out events. If the losses are high between two consecutive rounds, TO events can occur in TCP, as illustrated in Fig. 5.5. A TO event occurs if two consecutive rounds have losses with two or fewer out-of-order packets transmitted in the latter round. Thus, $\mathbf{P}(\text{TO}|W)$, the probability of a TO event given a window size of W, is given by

$$\mathbf{P}(\text{TO}|W) = \begin{cases} 1 & \text{if} \quad W < 3; \\ \sum_{i=0}^{2} \binom{W}{i} p^{W-i}(1-p)^i & \text{if} \quad W \geq 3. \end{cases} \tag{5.13}$$

Please note that when the window size is small ($W < 3$), losses lead to TO events. For instance, consider the scenario where $W = 2$ with packets $\mathbf{p_1}$ and $\mathbf{p_2}$ within its window. If $\mathbf{p_2}$ is lost, the TCP sender may transmit another packet $\mathbf{p_3}$ in the subsequent round, as the acknowledgment for $\mathbf{p_1}$ allows it to send a new packet. However, this action would result in a single duplicate acknowledgment with no additional packets in the pipeline, prompting the TCP sender to wait for acknowledgments until it times out.

In Eq. (5.13), we approximate W with the expected window size $E[W]$ obtained from Eq. (5.10). The TO event length is contingent upon the duration of the loss events. Then, the expected duration of TO period (in RTTs) is given as

$$\begin{aligned} &E[\text{duration of TO period}] \\ &= (1-p)\left[T_o p + 3T_o p^2 + 7T_o p^3 + 15T_o p^4 + 31T_o p^5 \right. \\ &\qquad \left. + \sum_{i=0}^{\infty}(63 + i{\cdot}64)T_o p^{6+i}\right] \\ &= (1-p)\left[T_o p + 3T_o p^2 + 7T_o p^3 + 15T_o p^4 + 31T_o p^5 + 63T_o \frac{p^6}{1-p} \right. \\ &\qquad \left. + 64T_o \frac{p^7}{(1-p)^2}\right]. \end{aligned} \tag{5.14}$$

Now, from the results in Eqs. (5.12)–(5.14), we can find an expression for the average throughput of TCP as

$$\mathcal{T}_{\text{tcp}} = \min\left(\frac{W_{\max}}{\text{RTT}}, \frac{1-p}{p} \frac{1}{\text{RTT}\,(E[r] + \mathbf{P}(\text{TO}|E[W])E[\text{duration of TO period}])}\right). \tag{5.15}$$

5.6.4 Throughput Analysis for TCP/NC

Now, let's look at the expected throughput for TCP/NC following the same methodology. We already discussed the intuition that the erasures that lead to TD and/or TO events in normal TCP may not lead to a similar impact in TCP/NC in Section 5.6.2 since each linearly independent packet provides a new degree of freedom to the receiver. This is shown in Fig. 5.6. Packets sent after the lost packets are also acknowledged by the receiver allowing extension of the window.

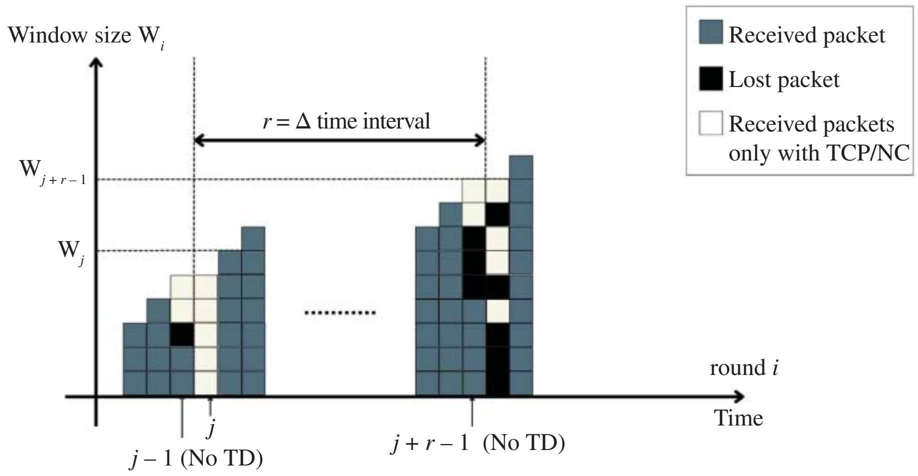


Figure 5.6 TCP/NC's window size with erasures that would lead to a triple-duplicate ACKs event when using standard TCP. Note that unlike TCP, the window size is non-decreasing. This figure is inspired by joint work with MinJi Kim [1] and by [8].

This also means that the TCP/NC will not experience window closing due to random losses as often as TCP.

TCP/NC window evolution

From Fig. 5.6, it's evident that TCP/NC can sustain its window size even in the presence of losses. This is attributed partly to TCP/NC's capability to accept packets that TCP would deem out of order. Consequently, TCP/NC's window progression deviates from that of TCP and can be defined by a straightforward recursive relationship as

$$E[W_i] = E[W_{i-1}] + \frac{E[a_{i-1}]}{E[W_{i-1}]} = E[W_{i-1}] + \min\{1, R(1-p)\}. \tag{5.16}$$

The recursive relationship encapsulates the notion that any packet linearly independent of previously received packets is deemed *innovative* and consequently acknowledged. Therefore, any arrival at the receiver is acknowledged with high probability. Thus, we can expect that $E[a_{i-1}]$ packets will be acknowledged, leading to an increment in the window size by $\frac{E[a_{i-1}]}{E[W_{i-1}]}$. It's noteworthy that $E[a_{i-1}] = (1-p) \cdot R \cdot E[W_{i-1}]$, as the encoder on average transmits R linear combinations for every original packet sent by the TCP sender.

Considering the maximum window size $W_{\max}$, the following expression can be formulated for TCP/NC's expected window size:

$$E[W_i] = \min(W_{\max}, E[W_1] + i \min\{1, R(1-p)\}), \tag{5.17}$$

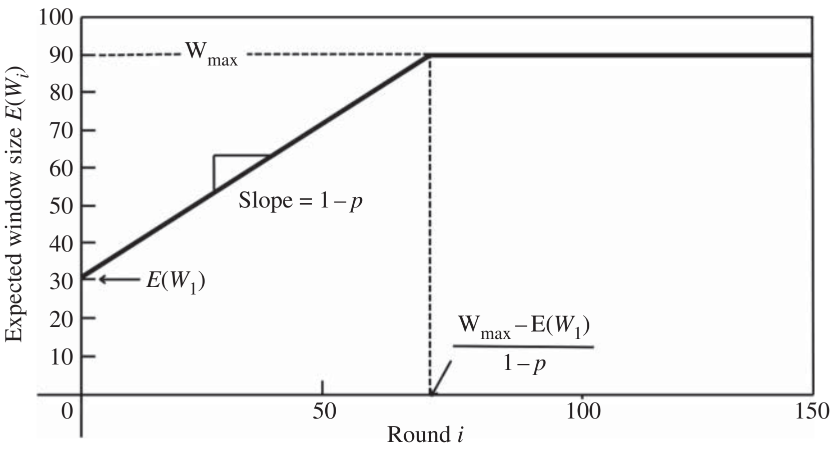


Figure 5.7 Expected window size for TCP/NC where $W_{max} = 90$, $E[W_1] = 30$. We usually assume $E[W_1] = 1$; here we use $E[W_1] = 30$ to exemplify the effect of $E[W_1]$. Source: This figure is by joint work with MinJi Kim [1].

where i is the round number. $E[W_1]$ is the initial window size, and we set $E[W_1] = 1$. Fig. 5.7 shows an example of the evolution of the TCP/NC window using Eq. (5.17).

Following the window evolution, now we can look for an expression for the TCP/NC throughput. The throughput of round i, $\mathcal{T}_\text{i}$, is directly proportional to the window size $E[W_i]$, i.e.

$$\mathcal{T}_\text{i} = \frac{E[W_i]}{\text{SRTT}} \min\{1, R(1-p)\} \text{ packets per second,} \tag{5.18}$$

where R is the redundancy factor of TCP/NC, and SRTT is the round-trip time estimate. The RTT and its estimate SRTT play an important role in TCP/NC. We note that $\mathcal{T}_\text{i} \propto (1-p)\cdot R\cdot E[W_i]$. At any given round i, TCP/NC sender transmits $R\cdot E[W_i]$ coded packets, and we expect $pR\cdot E[W_i]$ packets to be lost. Thus, the TCP/NC receiver only receives $(1-p)\cdot R\cdot E[W_i]$ degrees of freedom.

- Redundancy Factor (R): The redundancy factor $R \geq 1$ represents the ratio between the average rate at which linear combinations are transmitted to the receiver and the rate at which TCP's window progresses. For instance, if the TCP sender has 10 packets in its window, then the encoder transmits 10R linear combinations. If R is sufficiently large, the receiver will receive at least 10 linear combinations to decode the original 10 packets, unlike TCP, which would send just 10 packets. This redundancy is crucial (a) to compensate for losses within the network, and (b) to align TCP's sending rate with the rate at which data is received at the receiver. References [5, 9] introduce the redundancy factor in the context of TCP/NC, demonstrating that $R \geq \frac{1}{1-p}$ is necessary.

- Effective Round-Trip Time (SRTT): This is the estimated round-trip time that TCP maintains by observing the behavior of packets transmitted. It is calculated by averaging the time taken for a packet to be acknowledged after it is sent and often referred to as "smoothed" RTT. In Eq. (5.18), we utilize SRTT instead of RTT because SRTT represents the "effective" round-trip time experienced by TCP/NC. In lossy networks, TCP/NC's SRTT often exceeds RTT. This phenomenon is illustrated in Fig. 5.1. Initially, the first coded packet ($\mathbf{p_1} + \mathbf{p_2} + \mathbf{p_3}$) is received and acknowledged ($\mathbf{seen}(\mathbf{p_1})$), allowing the sender to accurately estimate the RTT, resulting in SRTT = RTT. However, if the second packet ($\mathbf{p_1} + \mathbf{2p_2} + \mathbf{p_3}$) is lost, the third packet ($\mathbf{p_1} + \mathbf{2p_2} + \mathbf{2p_3}$) is employed to acknowledge the second degree of freedom ($\mathbf{seen}(\mathbf{p_2})$). In our model, for the sake of simplicity, we assume that the transmission time of a packet is significantly shorter than the RTT, resulting in SRTT $\approx$ RTT despite losses. However, in practical scenarios, depending on the packet size, the transmission time might not be negligible.

5.6.5 TCP/NC Average Throughput

From Eq. (5.18), the average throughput over n rounds for TCP/NC can be found as

$$\begin{aligned}
\mathcal{T}_{\text{nc}} &= \frac{1}{n}\sum_{i=1}^{n}\frac{E[W_i]}{\text{SRTT}}\min\{1, R(1-p)\} \\
&= \frac{\sum_{i=1}^{n}\min(W_{\max}, E[W_1] + i\min\{1, R(1-p)\})}{n\cdot\text{SRTT}} \\
&= \frac{\sum_{i=1}^{n}\min(W_{\max}, E[W_1] + i)}{n\cdot\text{SRTT}} \quad \text{since}\, R \geq \frac{1}{1-p} \\
&= \frac{1}{n\cdot\text{SRTT}}\cdot f(n),
\end{aligned} \tag{5.19}$$

where

$$f(n) = \begin{cases} nE[W_1] + \frac{n(n+1)}{2} & \text{for} \quad n \leq r^* \\ nW_{\max} - r^*(W_{\max} - E[W_1]) + \frac{r^*(r^*-1)}{2} & \text{for} \quad n > r^* \end{cases}$$

$$r^* = W_{\max} - E[W_1].$$

Note that as $n \to \infty$, the average throughput $\mathcal{T}_{\text{nc}} \to \frac{W_{\max}}{\text{SRTT}}$.

An essential aspect of TCP is its congestion control mechanism. The analysis presented may raise concerns that network coding could potentially impede TCP's ability to respond to congestion. However, it's crucial to emphasize that the above analysis operates under the assumption of only random losses occurring with probability p, without considering correlated losses. Additionally, it's important to recognize that the erasure correction capability of network coding is constrained by the redundancy factor R. We will see further discussion on how to choose R for practical applications in the next chapter where we discuss the implementation strategies for the NC-enabled protocols.

Exercise 5.3

Extend the analysis in exercise 5.2 to find the average throughput via Monte Carlo simulations. Compare with theoretical results. Consider a maximum window size of 100.

Exercise 5.4

Optional project: Follow the process we explained in this chapter to propose a new protocol using NC by yourself and perform the analysis. You can try to adopt any protocol that is of interest.

5.7 Summary

Chapter 5 focuses on the design of NC protocols. By detailing the integration of NC with the Transmission Control Protocol (TCP), we aim to illustrate how to integrate NC into any other delivery protocol. The benefits of "seen" packets and how they enable better management of the congestion window are made evident in this chapter. Particularly, the innovative packets in each coded packet ensure that any received packet helps to keep the congestion window growing and avoid the so-called "triple-duplicate" condition. This integration leads to a higher throughput, as you can see from the analysis. We focus on goodput, taking the redundancy factor into account. This analysis can also be extended to other conditions that you may face in practical situations following the same approach. We expect that you have learned:

(A) How to conceptualize a network coding protocol.
(B) To perform an analysis of different parameters of a network coding protocol.
(C) Consider the overheads and analyze the complexity of the protocol.

Additional Reading Materials

Further details on the throughput analysis, its trade-offs, and congestion control of network coding with TCP can be found in [1, 10, 11]. The network-coding-inspired TCP works are primarily presented in [5, 12, 13].

References

1 M. Kim, M. Médard, and J. Barros, “Modeling network coded TCP throughput: a simple model and its validation,” *arXiv preprint arXiv:1008.0420*, 2010.

2 V. Jacobson, “Congestion avoidance and control,” *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, pp. 314–329, 1988.

3 V. Jacobson, “Modified TCP congestion avoidance algorithm,” *Technical report*, 1990.

4 L. S. Bramko, S. W. O’Malley, and L. L. Peterson, “TCP Vegas: New techniques for congestion detection and avoidance,” in *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, 1994, pp. 24–35.

5 J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher, and J. Barros, “Network coding meets TCP: theory and implementation,” *Proceedings of the IEEE*, vol. 99, no. 3, pp. 490–512, 2011.

6 J. K. Sundararajan, D. Shah, and M. Médard, “ARQ for network coding,” in *IEEE International Symposium on Information Theory*, 2008, pp.1651–1655.

7 J. J. Hunter, *Mathematical Techniques of Applied Probability, Vol. 2, Discrete Time Models: Techniques and Applications*. New York: Academic Press, 1983.

8 J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, “Modeling TCP throughput: a simple model and its empirical validation,” in *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, 1998, pp. 303–314.

9 J. K. Sundararajan, S. Jakubczak, M. Médard, M. Mitzenmacher, and J. Barros, “Interfacing network coding with TCP: an implementation,” *arXiv preprint arXiv:0908.1564*, 2009.

10 M. Kim, T. Klein, E. Soljanin, J. Barros, and M. Médard, “Trade-off between cost and goodput in wireless: replacing transmitters with coding,” in *Mobile Networks and Management*. Springer, 2013, pp. 1–14.

11 M. Kim, T. Klein, E. Soljanin, J. Barros, and M. Médard, “Modeling network coded TCP: analysis of throughput and energy cost,” *Mobile Networks and Applications*, vol. 19, no. 6, pp. 790–803, 2014.

12 J. K. Sundararajan, D. Shah, M. Médard, M. Mitzenmacher, and J. Barros, “Network coding meets TCP,” in *IEEE International Conference on Computer Communications*, 2009, pp. 280–288.

13 J. K. Sundararajan, “On the role of feedback in network coding,” Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

6

Implementation of Network Coding Protocols

6.1 A Real-World Implementation of TCP/NC

In this section, we discuss some of the practical issues that arise in designing an **implementation of the TCP/NC protocol** compatible with real TCP/IP stacks. These issues were not considered in the idealized setting discussed up to this point. We present a real-world implementation of TCP/NC and thereby show that it is possible to overcome these issues and implement a TCP-aware network-coding layer that has the property of a clean interface with TCP. Our real-world implementation demonstrates the compatibility of our protocol with TCP-Reno. The rest of this section pertains to TCP-Reno.

6.1.1 Sender-Side Module

The description of the protocol in Section 5.2 assumes a fixed packet length, which allows all coding and decoding operations to be performed symbol-wise on the whole packet. That is, an **entire packet serves as the basic unit of data** (i.e. as a single unknown), with the implicit understanding that the exact same operation is being performed on every symbol within the packet. The main advantage of this view is that the decoding matrix operations (i.e. Gauss–Jordan elimination) can be performed at the granularity of packets instead of individual symbols. Also, the acknowledgments (ACK)s are then able to be represented in terms of packet numbers. Finally, the coding vectors then have one coefficient for every packet, not every symbol. Note that the same protocol and analysis of Section 5.2 holds even if we fix the basic unit of data as a symbol instead of a packet. The problem is that the complexity will be very high as the size of the coding matrix will be related to the number of symbols in the coding buffer, which is much more than the number of packets (typically, a symbol is one byte long).

In practice, TCP is a byte stream-oriented protocol where ACKs are in terms of byte sequence numbers. If all packets are of fixed length, we can still apply the packet-level approach, since we have a clear and consistent map between packet sequence numbers and byte sequence numbers. In reality, however, TCP might generate segments of different sizes. The choice of how many bytes to group into a segment is usually made based on the maximum transmission unit (MTU) of the network, which could vary with time. A more common occurrence is that applications may use the PUSH flag option asking TCP to packetize the currently outstanding bytes into a segment, even if it does not form a segment of the maximum allowed size. In short, it is important to ensure that our protocol works correctly in spite of variable packet sizes.

A closely related problem is that of repacketization. **Repacketization**, as described in Chapter 21 of [1], refers to the situation where a set of bytes that were assigned to two different segments earlier by TCP may later be reassigned to the same segment during retransmission. As a result, the grouping of bytes into packets under TCP may not be fixed over time.

Both variable packet lengths and repacketization need to be addressed when implementing the coding protocol. To solve the first problem, if we have packets of different lengths, we could elongate the shorter packets by appending sufficiently many dummy zero symbols until all packets have the same length. This will work correctly as long as the receiver is somehow informed how many zeros were appended to each packet. While transmitting these extra dummy symbols will decrease the throughput, generally, this loss will not be significant, as packet lengths are usually consistent.

However, if we have repacketization, then we have another problem, namely it is no longer possible to view a packet as a single unknown. This is because we would not have a one-to-one mapping between packets sequence numbers and byte sequence numbers; the same bytes may now occur in more than one packet. Repacketization appears to destroy the convenience of performing coding and decoding at the packet level.

To counter these problems, we propose the following solution. The coding operation described in Section 5.2 involves the sender storing the packets generated by the TCP source in a **coding buffer**. We pre-process any incoming TCP segment before adding it to the coding buffer as follows:

1) First, any part of the incoming segment that is already in the buffer is removed from the segment.
2) Next, a separate TCP packet is created out of each remaining contiguous part of the segment.
3) The source and destination port information is removed. It will be added later in the network coding header.

4) The packets are appended with sufficiently many dummy zero bytes, to make them as long as the longest packet currently in the buffer.

Every resulting packet is then added to the buffer. This processing ensures that the packets in the buffer will correspond to disjoint and contiguous sets of bytes from the byte stream, thereby restoring the one-to-one correspondence between the packet numbers and the byte sequence numbers. The reason the port information is excluded from the coding is because port information is necessary for the receiver to identify which TCP connection a coded packet corresponds to. Hence, the port information should not be involved in the coding. We refer to the remaining part of the header as the TCP subheader.

Upon decoding the packet, the receiver can find out how many bytes are real and how many are dummy using the Start_i and End_i header fields in the network coding header (described below). With these fixes in place, we are ready to use the packet-level algorithm of Section 5.2. All operations are performed on the packets in the coding buffer. Figure 6.1 shows a typical state of the buffer after this preprocessing. The gaps at the end of the packets correspond to the appended zeros. It is important to note that the TCP control packets such as SYN packet and reset packet are allowed to bypass the coding buffer and are directly delivered to the receiver without any coding.

A coded packet is created by forming a random linear combination of a subset of the packets in the coding buffer. The coding operations are done over a field of size 2^8 in our implementation. In this case, a field symbol corresponds to one byte. The header of a coded packet should contain information that the receiver can use to identify what is the linear combination corresponding to the packet. We now discuss the header structure in more detail.

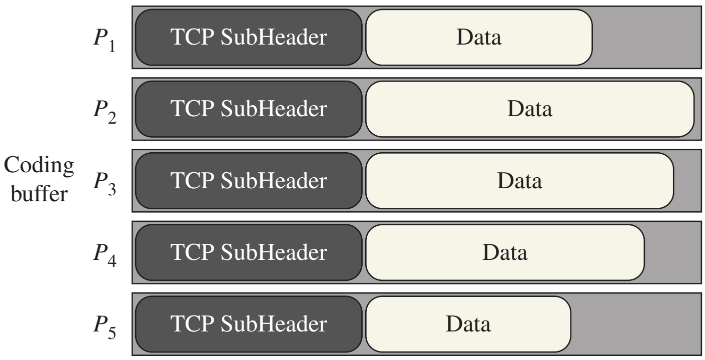


Figure 6.1 The coding buffer. This figure is inspired by joint work with Jay Kumar Sundararajan [2].

We assume that the network coding header has the structure shown in Fig. 6.2. The typical sizes (in bytes) of the various fields are written above them. The meaning of the various fields is described next:

- **Source and Destination Port:** The port information is needed for the receiver to identify the coded packet's session. It must not be included in the coding operation. It is taken out of the TCP header and included in the network coding header.
- **Base:** The TCP byte sequence number of the first byte that has not been ACKed. The field is used by intermediate nodes or the decoder to decide which packets can be safely dropped from their buffers without affecting reliability.
- n**:** The number of packets involved in the linear combination.
- Start_i**:** The starting byte of the ith packet involved in the linear combination.
- End_i**:** The last byte of the ith packet involved in the linear combination.
- α_i**:** The coefficient used for the ith packet involved in the linear combination.

The Start_i (except Start_1) and End_i are expressed relative to the previous packet's End and Start, respectively, to save header space. As shown in Fig. 6.2, this header format will add $5n + 7$ bytes of overhead for the network coding header in addition to the TCP header, where n is the number of packets involved in a linear combination. (Note that the port information is not counted in this overhead, since it has been removed from the TCP header.) We believe it is possible to reduce this overhead by further optimizing the header structure.

In theory, the sender transmits a random linear combination of all packets in the coding buffer. However, as noted above, the size of the header scales with the number of packets involved in the linear combination. Therefore, mixing all packets currently in the buffer will may lead to a large coding header.

To solve this problem, we propose mixing only a constant-sized subset of the packets chosen from within the coding buffer. We call this subset the *coding window*. The coding window evolves as follows. The algorithm uses a fixed parameter for the maximum coding window size W. The coding window contains the packet that arrived most recently from TCP (which could be a retransmission), and the $(W - 1)$ packets before it in sequence number, if possible. However, if some of the $(W - 1)$ preceding packets have already been dropped, then the window is allowed to extend beyond the most recently arrived packet until it includes W packets.

Note that this limit on the coding window implies that the code is now restricted in its power to correct erasures and to combat reordering-related issues. The choice of W will thus play an important role in the performance of the scheme. The correct value for W will depend on the length of burst errors that the channel is expected to produce. Other factors to be considered while choosing W are discussed in Section 6.1.3.

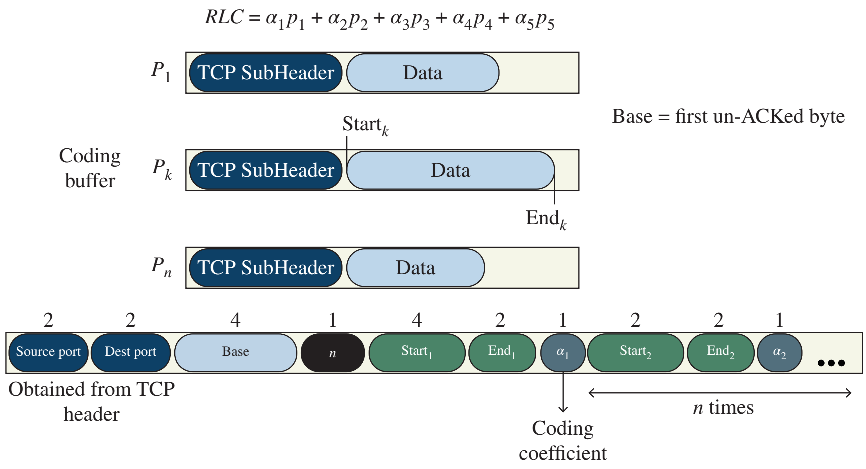


Figure 6.2 The network coding header. This figure is inspired by joint work with Jay Kumar Sundararajan [2].

In terms of buffer management, a packet is removed from the coding buffer if a TCP ACK has arrived requesting a byte beyond the last byte of that packet. If a new TCP segment arrives when the coding buffer is full, then the segment with the newest set of bytes must be dropped. This may not always be the newly arrived segment, for instance, in the case of a TCP retransmission of a previously dropped segment.

6.1.2 Receiver-Side Module

The decoder module's operations are outlined below. The main data structure involved is the decoding matrix, which stores the coefficient vectors corresponding to the linear combinations currently in the decoding buffer.

To manage acknowledgments, the receiver-side module stores the incoming linear combination in the decoding buffer. Then it unwraps the coding header and appends the new coefficient vector to the decoding matrix. Gauss–Jordan elimination is performed and the packet is dropped if it is not innovative (i.e. if it is not linearly independent of previously received linear combinations). After Gauss–Jordan elimination, the oldest unseen packet is identified. The decoder acknowledges the last seen packet by **requesting the byte sequence number of the first byte of the first unseen packet**, using a regular TCP ACK. Note that this could happen before the packet is decoded and delivered to the receiver TCP. The port and IP address information for sending this ACK may be obtained from the SYN packet at the beginning of the connection. Any ACKs generated by the receiver TCP are not sent to the sender. They are instead used to update the receive window field that is used in the TCP ACKs generated by the decoder (see subsection below). They are also used to keep track of which bytes have been delivered, for buffer management.

The Gauss–Jordan elimination operations are performed not only on the decoding coefficient matrix, but correspondingly also on the coded packets themselves. When a new packet is decoded, any dummy zero symbols that were added by the encoder are pruned using the coding header information. A new TCP packet is created with the newly decoded data and the appropriate TCP header fields, and this is then delivered to the receiver TCP.

For buffer management, the decoding buffer needs to store packets that have not yet been decoded and delivered to the TCP receiver. Delivery can be confirmed using the receiver TCP's ACKs. In addition, the buffer also needs to store those packets that have been delivered but have not yet been dropped by the encoder from the coding buffer. This is because such packets may still be involved in incoming linear combinations. The *Base* field in the coding header addresses this issue. *Base* is the oldest byte in the coding buffer. Therefore, the decoder can drop a

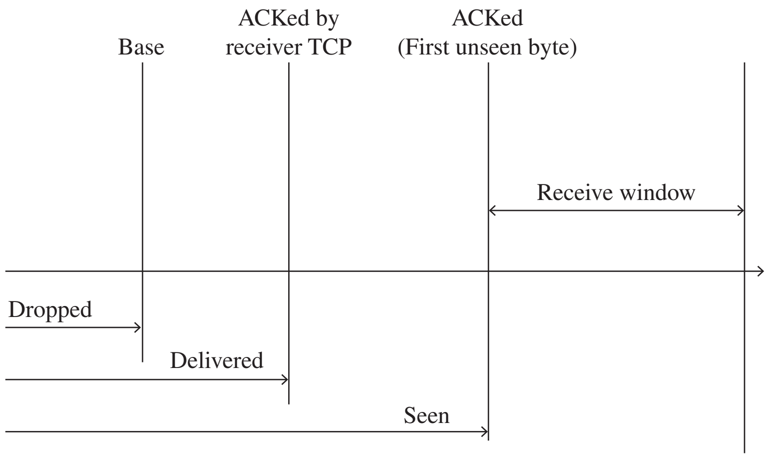


Figure 6.3 Receiver-side window management. This figure is inspired by joint work with Jay Kumar Sundararajan [2].

packet if its last byte is smaller than *Base*, and in addition, has been delivered to and ACKed by the receiver TCP. Whenever a new linear combination arrives, the value of *Base* is updated from the header, and any packets that can be dropped are dropped.

The buffer management can be understood using Fig. 6.3. It shows the receiver-side windows in a typical situation. In this case, *Base* is less than the last delivered byte. Hence, some delivered packets have not yet been dropped. There could also be a case where *Base* is beyond the last delivered byte, possibly because nothing has been decoded in a while.

The TCP receive window header field is used by the receiver to inform the sender how many bytes it can accept. Since the receiver TCP's ACKs are suppressed, the decoder must copy this information in the ACKs that it sends to the sender. However, to ensure correctness, we may have to modify the value of the TCP receive window based on the decoding buffer size. The last acceptable byte should thus be the minimum of the receiver TCP's last acceptable byte and the last byte that the decoding buffer can accommodate. Note that while calculating the space left in the decoding buffer, we can include the space occupied by data that has already been delivered to the receiver because such data will get dropped when *Base* is updated. If the window scaling option is used by TCP, this needs to be noted from the SYN packet, so that the modified value of the receive window can be correctly reported. Ideally, we would like to choose a large enough decoding buffer size so that the decoding buffer would not be the bottleneck, and this modification would never be needed.

Exercise 6.1 Finite Field Analysis for Encoding Vector Overhead

In this exercise, analyze the overhead incurred by encoding vectors in erasure coding schemes across different finite fields.

1) Finite Field Selection:
   - Explore and select various finite fields available for encoding in erasure coding schemes.
2) Encoding Vector Overhead Calculation:
   - Calculate the overhead introduced by encoding vectors in each finite field, considering factors such as field size and encoding matrix dimensions.
3) Comparative Analysis:
   - Compare the overhead of encoding vectors across different finite fields, highlighting any differences or patterns observed.

Optional:

- Consider ways of reducing the encoding vector overhead.

6.1.3 Discussion of Implementation Parameters

The choice of **redundancy factor** is based on the effective loss probability on the links. For a loss rate of p_e, with an infinite window W and using TCP-Vegas, the theoretically optimal value of R is $1/(1-p_e)$. The basic idea is that of the coded packets that are sent into the network, only a fraction $(1-p_e)$ of them are delivered on average. Hence, the value of R must be chosen so that in spite of these losses, the receiver is able to collect linear equations at the same rate as the rate at which the unknown packets are mixed in them by the encoder. As discussed below, in practice, the value of R may depend on the coding window size W. As W decreases, the erasure correction capability of the code goes down. Hence, we may need a larger R to compensate and ensure that the losses are still masked from TCP. Another factor that affects the choice of R is the use of TCP-Reno. The TCP-Reno mechanism causes the transmission rate to fluctuate around the link capacity, and this leads to some additional losses over and above the link losses. Therefore, the optimal choice of R may be higher than $1/(1-p_e)$.

There are several considerations to keep in mind while choosing W, **the coding window size**. The main idea behind coding is to mask the losses on the channel from TCP. In other words, we wish to correct losses without relying on the ACKs. Consider a case where W is just 1. Then, this is a simple repetition code. Every packet is repeated R times on average. Now, such a repetition would be useful only for recovering one packet, if it was lost. Instead, if W was say 3, then every linear combination would be useful to recover any of the three packets involved.

Ideally, the linear combinations generated should be able to correct the loss of any of the packets that have not yet been ACKed. For this, we need W to be large. This may be difficult, since a large W would lead to a large coding header.

The penalty of keeping W small, on the other hand, is that it reduces the error correction capability of the code. For a loss probability of 10%, the theoretical value of R is around 1.1. However, this assumes that all linear combinations are useful to correct any packet's loss. The restriction on W means that a coded packet can be used only for recovering those W packets that have been mixed to form that coded packet. In particular, if there is a contiguous burst of losses that result in a situation where the receiver has received no linear combination involving a particular original packet, then that packet will show up as a loss to TCP. This could happen even if the value of R is chosen according to the theoretical value. To compensate, we may have to choose a larger R.

The connection between W, R, and the losses that are visible to TCP can be visualized as follows. Imagine a process in which whenever the receiver receives an innovative linear combination, one imaginary token is generated, and whenever the sender slides the coding window forward by one packet, one token is used up. If the sender slides the coding window forward when there are no tokens left, then this leads to a packet loss that will be visible to TCP. The reason is, when this happens, the decoder will not be able to see the very next unseen packet in order. Instead, it will skip one packet in the sequence. This will make the decoder generate duplicate ACKs requesting that lost (i.e. unseen) packet, thereby causing the sender to notice the loss.

In this process, W corresponds to the initial number of tokens available at the sender. Thus, when the difference between the number of redundant packets (linear equations) received and the number of original packets (unknowns) involved in the coding up to that point is less than W, the losses will be masked from TCP. However, if this difference exceeds W, the losses will no longer be masked.

By adding enough redundancy, the coding operation essentially converts the lossiness of the channel into an extension of the round-trip time (RTT). This is why our initial discussion in Section 5.2 proposed the use of the idea with TCP-Vegas, since TCP-Vegas controls the congestion window in a smoother manner using RTT, compared to the more abrupt loss-based variations of TCP-Reno. However, the coding mechanism is also compatible with TCP-Reno. The choice of W plays an important role in ensuring this compatibility. The choice of W controls the power of the underlying code and hence determines when losses are visible to TCP. As explained above, losses will be masked from TCP as long as the number of received equations is no more than W short of the number of unknowns involved in them. For compatibility with Reno, we need to make sure that whenever the sending rate exceeds the link capacity, the resulting queue drops are visible to TCP as losses. A very large value of W is likely to mask even these congestion losses, thereby temporarily giving TCP a large estimate of capacity. This will eventually lead to a timeout and will affect throughput. The value of W should therefore be large enough to mask the link losses and small enough to allow TCP to see the queue drops due to congestion.

It is important to implement the encoding and decoding operations efficiently, since any time spent in these operations will affect the RTT perceived by TCP. The main computational overhead on the encoder side is the formation of the random linear combinations of the buffered packets. The management of the buffer also requires some computation, but this is small compared to the random linear coding, since the coding has to be done on every byte of the packets. Typically, packets have a length L of around 1500 bytes. For every linear combination that is created, the coding operation involves LW multiplications and $L(W-1)$ additions over $\mathbb{F}_{2^8}$, where W is the coding window size. Note that this has to be done R times on average for every packet generated by TCP. Since the coded packets are newly created, allocating memory for them could also take time.

On the decoder side, the main operation is the Gauss–Jordan elimination. Note that, to identify whether an incoming linear combination is innovative or not, we need to perform Gauss–Jordan elimination only on the decoding matrix, and not on the coded packet. If it is innovative, then we perform the row transformation operations of Gauss–Jordan elimination on the coded packet as well. This requires $O(LW)$ multiplications and additions to zero out the pivot columns in the newly added row. The complexity of the next step of zeroing out the newly formed pivot column in the existing rows of the decoding matrix varies depending on the current size and structure of the matrix. Upon decoding a new packet, it needs to be packaged as a TCP packet and delivered to the receiver. Since this requires allocating space for a new packet, this could also be expensive in terms of time.

As we shall see, the benefits brought by the erasure correction begin to outweigh the overhead of the computation and coding header for loss rates of about 3%. This could be improved further by more efficient implementation of the encoding and decoding operations.

Interface with TCP

An important point to note is that **the introduction of the new network coding layer does not require any change in the basic features of TCP**. This is consistent with our objective stated in the last chapter. As described above, the network coding layer accepts TCP packets from the sender TCP and in return delivers regular TCP ACKs back to the sender TCP. On the receiver side, the decoder delivers regular TCP packets to the receiver TCP and accepts regular TCP ACKs. Therefore, neither the TCP sender nor the TCP receiver sees any difference looking downward in the protocol stack. The main change introduced by the protocol is that the TCP packets from the sender are transformed by the encoder by the network coding process. This transformation is removed by the decoder, making it invisible to the TCP receiver. On the return path, the TCP receiver's ACKs are suppressed, and instead the decoder generates regular TCP ACKs that are delivered to the sender.

While the basic features of the TCP protocol see no change, other special features of TCP that make use of the ACKs in ways other than to report the next required byte sequence number, will need to be handled carefully. For instance, implementing the timestamp option in the presence of network coding across packets may require some thought. With TCP/NC, the receiver may send an ACK for a packet even before it is decoded. Thus, the receiver may not have access to the timestamp of the packet when it sends the ACK. Similarly, the TCP checksum field has to be dealt with carefully. Since a TCP packet is ACKed even before it is decoded, its checksum cannot be tested before ACKing. One solution is to implement a separate checksum at the network coding layer to detect errors. In the same way, the various other TCP options that are available have to be implemented with care to ensure that they are not affected by the premature ACKs.

We have seen that multi-hop networks benefit greatly from recoding. We have seen that coded packets can be created as each coded packet being a combination of multiple packets mixed together with randomly generated coding coefficients, and the receiving node can decode the packets once it has enough innovative coded packets. **Recoders can create their own coded packets from the innovative packets they have received**. This allows the source to stop transmitting once the next node has received enough innovative packets, without waiting for the sink to receive all of them. The intermediate node can regenerate additional coded combinations to compensate for losses in the forward channels. This process reduces the total number of transmissions and helps achieve the capacity of the network. The losses in a multi-hop network are not cumulative anymore, and each node has to compensate only for the losses in its outgoing links. Thus in a multi-hop network, network coding further improves the performance by reducing the need for retransmissions and improving end-to-end goodput and in-order delivery delay.

In case of a block RLNC approach, recoding is a very direct process. If we consider a generation of m packets, where each packet has n symbols, then each packet p_i in this generation can be considered as $\{p_{i,1}, p_{i,2} \cdots p_{i,n}\}$ and the generation will look like a $m \times n$ matrix P:

$$\begin{bmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & \ddots & \vdots \\ p_{i,1} & \cdots & p_{i,n} \\ \vdots & \ddots & \vdots \\ p_{m,1} & \cdots & p_{m,n}. \end{bmatrix} \tag{6.1}$$

In normal RLNC approach, this native generation will be multiplied by a matrix of randomly generated coefficients α which is an $m \times m$ matrix to create the encoded matrix P' as shown in Eq. (6.2), and the augmented generation will be this new matrix concatenated with the coefficient matrix as $[\alpha \, |P']$.

$$P' = \begin{bmatrix} \alpha_{1,1} & \cdots & \alpha_{1,m} \\ \vdots & \ddots & \vdots \\ \alpha_{i,1} & \cdots & \alpha_{i,m} \\ \vdots & \ddots & \vdots \\ \alpha_{m,1} & \cdots & \alpha_{m,m} \end{bmatrix} \times \begin{bmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & \ddots & \vdots \\ p_{i,1} & \cdots & p_{i,n} \\ \vdots & \ddots & \vdots \\ p_{m,1} & \cdots & p_{m,n}. \end{bmatrix} \tag{6.2}$$

This augmented generation will be transmitted over the communication channel as codewords (augmented packets). When an intermediate node receives enough linearly independent codewords to reproduce the generation, it can be recoded by multiplying the received generation with its locally generated coding coefficients. Let's consider R is the matrix received by the intermediate node. In a lossless environment, $R = [\alpha \, |P']$. If β is the $m \times m$ matrix of locally generated coefficients in the receiving node, then the re-coding happens as $\beta \times R$ and this will act as the new coded generation. It is not required to concatenate β to the resulting matrix since the coefficient part in the $\beta \times R$ already accounts for the operations with β. This shows that the communication overhead with recoding does not scale with the number of hops. At the sink node, the message will be finally decoded if there is at least m number of independent equations received, by Gauss–Jordan elimination.

However, the recoding in a Sliding Window Network Coding (SWNC) approach is not as direct as it is in block RLNC. If we just use sliding window encoding scheme only at the end nodes, such a scheme can have no significant improvement in performance other than ensuring the availability of innovative packets at the receiver. Further, such systems will require the source node to encode the packets at a rate that can compensate for the cumulative losses in the channel. However, an on-the-fly sliding window recoder can allow the intermediate node to recode over different window sizes and thus adapt to the different loss rates in the links. In this section, we look at the design of a practical sliding window recoder that can work efficiently over the links with varying loss rates and allow each intermediate node to code at a different rate required for their outgoing links. More details on a practical sliding window recoder are presented in [3].

Figure 6.4 shows the different network scenarios for transmitting information from a source to a receiver that we consider. The first case, scenario (a), depicts an SWNC-enabled network, but only the end nodes are capable of encoding/decoding and the second case, scenario (b), represents the case of SWNC with a source, a recoder node, and a receiver. In this case, the recoder node uses linear

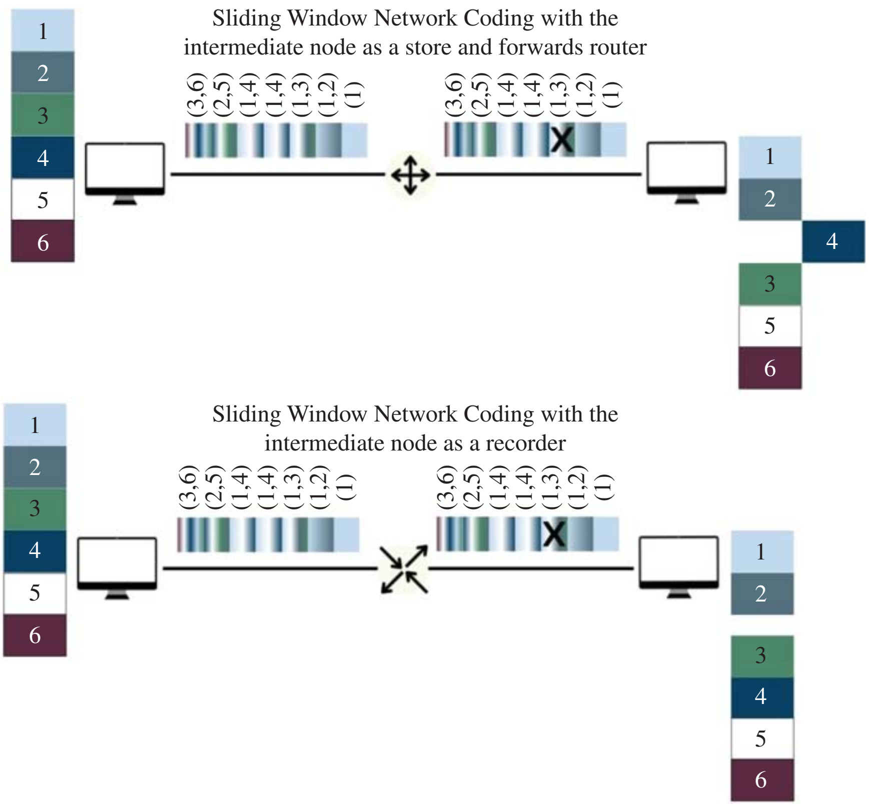

Sliding Window Network Coding with the intermediate node as a store and forwards router
1
2
3
4
5
6
(3,6)
(2,5)
(1,4)
(1,4)
(1,3)
(1,2)
(1)
X
Sliding Window Network Coding with the intermediate node as a recorder

Figure 6.4 Sliding window network coding – with and without recoder.

network coding to combine packets from the source and creates a new packet that is sent to the receiver. The recoder node has its own packet *buffer*, which is used to temporarily store packets received from the source node. It has the ability to recode with a new code rate, to compensate for a different loss rate in its forward channel. Even though we will focus on the single-path scenario with only one recoder, the recoding mechanism can be directly extended to multi-path and/or multi-hop scenarios as well.

The SWNC operates with a fixed code rate, which means that after a certain number of new packets are sent, a set number of Forward Erasure Correction (FEC) packets will be transmitted as well. The recoder allows each node to set its code rate to account for errors on its outgoing link. In exceptional cases of extreme error bursts where a priori repair packets are insufficient, feedback-based FEC (FB-FEC) can be utilized. This may reduce the code rate slightly but ensures all packets are transmitted reliably.

The sliding window at the sender and receiver ends is very similar to the base description in Section 5.4. However, at the intermediate node, it is important to consider that the packets sent out of it have innovative packets capable of overcoming the extra losses at its outgoing channel. The recoder should be "carefully" mixing the packets, and the original window sizes are maintained on the forward packets.

6.1.4 Recoding Process

The packets received from incoming links are stored in a buffer along with their respective window information. At the slot for sending a packet in the forward channel, the recoder selects the window size from available packets and combines them into a single packet. The recoder starts by selecting the first packet in the window, adding it to the recoding set, and using its opening window slot as the opening window slot of the outgoing packet. The recoder continues to add packets until the desired window size is reached. The closing window of the last included packet becomes the closing window value of the recoded packet. It is important to note that the window size at the recoder may not match the difference between the end values of the window associated with the recoded packet, as the window is maintained based on the windows of the incoming packets.

The next concern at the recoder after managing the window properly is to handle the coefficients. The outgoing packets should have coefficients that will allow the decoder to decode the packet. This is accomplished by adding the coefficients to the packet payload. The recoder performs the coding operation on both the packet payload and the incoming coefficients. Once the decoder has enough packets, it separates the received coefficients, calculates their inverse matrix, and decodes the message. This results in an increase in the payload size that consists of the actual payload length and the number of added coefficients, so there is a small reduction in the maximum payload size that can be used at the source node. However, it is important to note that the number of coefficients does not increase as the packet travels through the network and is bounded by the maximum window size at the encoder. The recoding algorithm is presented in Algorithm 6.1 and an example scenario with two hop communication is presented in Fig. 6.5.

6.1.5 Feedback Mechanism

Network-coding-based approaches do not rely on the arrival order or sequence number of packets. Instead, any coded packet can replace a lost packet within its coding window. The source node uses feedback from the destination to move the window forward, taking into account the number of innovative packets received at the decoder. This feedback is designed such that it includes both fully decoded

Algorithm 6.1 Recoder's process.

```
 1: Inputs: Incoming packet and coding headers
 2: Outputs: Outgoing packet and coding headers
 3: Identify min and max window values and coded payload
 4: if the incoming packet is innovative then
 5:     Add the incoming packet to buffer
 6: else
 7:     Discard the packet
 8: end if
 9: if slot to send a new packet then
10:     Add the next packet in buffer to the window
11: end if
12: Generate coefficients for the current window and create a recoded packet
13: Add the min window of first packet and the max window of last packet in the recod-
    ing window as the min and max of the outgoing packet's window
14: Send the outgoing packet with window limits
```

packets and partially decoded packets in the coding window that still need additional packets for complete decoding.

The window at each node is advanced based on the feedback received. The feedback value is compared to the window closing value of the packets, and any packet that does not contain an innovative packet (i.e. a packet with a value higher than the feedback value) is removed from the coding window. If there is a large burst of errors, the window will continue to grow until it reaches its maximum capacity. Different decision-making criteria can be used if the window reaches its capacity. For example, if every packet is critical, the window can be kept at that length and more repair packets can be sent. However, in many practical scenarios, the freshness of the packet is a crucial factor, such as in live video streaming where some lost packets may not be worth retransmitting. In such cases, the oldest packets may be dropped from the window to keep it sliding [4].

6.1.6 Implementation Strategies

The first step is to define the coding header and payload structures. As they are common in packet networks, headers often include additional information about the message such as source and destination IPs and flags. In network coding implementations, the header also includes information necessary for decoding the message. This section elaborates on the network-coding-related header values and flags that are essential for recoding.

In a sliding window mechanism, the size and opening point of the window are crucial factors and are therefore included in the coding header. The size of the window is represented by 1 byte, while the opening point of the window is represented

	Source		**Channel 1**	**Recorder**		**Channel 2**	**Receiver**
Slot No.	FB	Innovative packet	Packets in transit	Received packet	Coding window	Packets in transit	
1		P1	(1,1)ΣP				
2		P2	(1,2)ΣP	(1,1)ΣP = R1	(1,1)ΣR	(1,1)ΣP	
3		P3	(1,3)ΣP	(1,2)ΣP = R2	(1,2)ΣR	(1,2)ΣP	P1
4		P4	(1,4)ΣP	(1,3)ΣP = R3	(1,3)ΣR	(1,3)ΣP	P2
5			(1,4)ΣP FEC	(1,4)ΣP = R4	(1,3)ΣR FEC	(1,3)ΣP	Lost packet
6	1	P5	(2,5)ΣP	Already have 4 packets	(1,4)ΣR	(1,4)ΣP	P3
7	2	P6	(3,6)ΣP	(2,5)ΣP = R5	(2,5)ΣR	(1,5)ΣP	P4
8	3	P7	(4,7)ΣP	(3,6)ΣP = R6	(3,6)ΣR	(1,6)ΣP	P5
9	4	P8	(5,8)ΣP	Lost packet	(3,6)ΣR FEC	(1,6)ΣP	P6
10	4		(5,8)ΣP FEC	(5,8)ΣP = R7	(5,7)ΣR	(2,8)ΣP	P6
11	6	P9	(6,9)ΣP	(5,8)ΣP = R8	(5,8)ΣR	(2,8)ΣP	P6+1
12	6	P10	(7,10)ΣP	(6,9)ΣP = R9	(6,9)ΣR	(3,9)ΣP	P8
13		P11	(7,11)ΣP	(7,10)ΣP = R10	(7,9)ΣR FEC	(3,9)ΣP	P9
14	7	P12	(8,12)ΣP	(7,11)ΣP = R11	(7,10)ΣR	(5,10)ΣP	Lost packet
15	8		(9,12)ΣP FEC	(8,12)ΣP = R12	(7,11)ΣR	(5,11)ΣP	P10
16	9	P13	(10,13)ΣP	Already have 12 packets	(9,12)ΣR	(6,12)ΣP	P11
17	10	P14	(11,14)ΣP	(10,13)ΣP = R13	(10,12)ΣR FEC	(7,12)ΣP	P12
18	11	P15	(12,15)ΣP	(11,14)ΣP = R14	(11,13)ΣR	(7,13)ΣP	P12
19	12	P16	(13,16)ΣP	(12,15)ΣP = R15	(12,14)ΣR	(8,14)ΣP	Lost packet
20	12		(13,16)ΣP FEC	(13,16)ΣP = R16	(13,15)ΣR	(10,15)ΣP	P12+1
21	13	P17	(14,17)ΣP	Lost packet	(13,15)ΣR FEC	(10,15)ΣP	P12+2
22	14	P18	(15,18)ΣP	(14,17)ΣP = R17	(14,16)ΣR	(11,16)ΣP	P15
23	15	P19	(16,19)ΣP	(15,18)ΣP = R18	(15,17)ΣR	(12,17)ΣP	P16
24	16	P20	(17,20)ΣP	(16,19)ΣP = R19	(16,18)ΣR	(13,18)ΣP	Lost packet
25	16			(17,20)ΣP = R20	(17,18)ΣR	(14,18)ΣP	P16+1
26	17				(17,19)ΣR	(14,19)ΣP	P18
27	18				(17,20)ΣR	(14,20)ΣP	P19
28	19						P20

Figure 6.5 An example case of a 2-hop network with SWNC. The packets lost in transmission are highlighted with boxes in cl
columns.

by 2 bytes. The number of coefficients is limited by the window size, which is also included in the coding header (1 byte). To distinguish between repair packets and packets with innovative data compared to the previous packet, a single bit, 0 or 1, is added. In multi-hop systems, two flags are used to distinguish repair packets: the "source FEC flag," which is set to 1 by the source node when generating a repair packet, and the "last FEC flag," which is changed by each node and set to 1 when a node generates a repair packet. Both flags are set to 1 in the source node but may be different in other nodes. Thus the additional coding header for recoding is 5 bytes, with the last 6 bits kept null.

The essence of network coding is the possibility of creating new packets from a combination of original packets using random coefficients. These coefficients are necessary for decoding at the receiver; so they are either attached to the packet payload or header. The size of the coefficients is determined by the number of packets included in the coding window. In end-to-end coding schemes, where coding is performed only at the source and decoding occurs at the sink, the coefficients can be shared by sharing the seed value used to generate them, along with the window size, instead of sharing the coefficients themselves.

However, when recoding is performed, the coefficients must reflect the corresponding operations. Simply sharing the seed and recreating the original coefficients is not ideal for a network that wants to use network coding optimally by recoding at intermediate nodes. To ensure that recoding operations are reflected in the coding coefficients, we propose a simple change to the packet payload. The source node performs the encoding and attaches the coding coefficients to the payload, resulting in a payload size slightly larger than the original packet size. This simplifies the recoding process for nodes, as they can simply recode the packet payload using locally generated coefficients. The sink node can then disassemble the original payload and coefficients and decode correctly using the received coefficients. It's worth mentioning that the size of the payload does not increase as it travels through the network. The payload length is determined only by the maximum coding window and recoder nodes simply perform the recoding without adding their own coefficients to the payload.

The feedback packet is also defined slightly differently from a simple TCP-style packet, with a minor modification. Instead of sending the sequence number of the last correctly received packet, our approach requires the receiver to send two values: the number of correctly decoded packets and the number of partially decoded packets. Partially decoded packets refer to packets with innovative information that have not yet been fully decoded because there are not enough packets available compared to the window size of the arrived packet. These are the "seen" packets and can be completely decoded as soon as a repair packet arrives that completes the window for decoding. This is a critical aspect of network coding that reduces the delay in in-order delivery.

Exercise 6.2 Investigating Communication Rate in Relaying Systems

In this exercise, you will explore the rate of communication in a relay system consisting of a source, a relay, and a sink. Using the PyErasure library we can investigate the benefit of recoding. The communication channels between the source and the relay, as well as between the relay and the sink, are subject to erasure probabilities e_1 and e_2, respectively. Two types of relays should be implemented: a pure relay that forwards received packets, and a coded relay that recodes incoming packets.

1) Implement the relay system with two types of relays:
   - **Pure Relay:** Forwards packets it receives.
   - **Coded Relay:** Recodes incoming packets.
2) Simulate the communication process step by step:
   - At each step, allow the source and relay to generate a packet.
   - Note that the pure relay can only generate a packet if it has something in the queue, while the coded relay can generate via recoding as long as it has received more than one packet.
3) Count the number of steps needed to send 64 packets from the source to the sink.
4) Investigate the impact of varying erasure probabilities (e_1 and e_2) on the communication rate.
5) Analyze and compare the performance of the pure relay and the coded relay in terms of communication efficiency and reliability.
6) Document the implementation details, simulation results, and conclusions drawn from the experiment.

6.2 Adaptive Sliding Window

In this section, we look at an advanced sliding window approach, called the **Adaptive and Causal RLNC (AC-RLNC)** scheme. We know that the feedbacks are used to manage the window size, and it also provides the idea of how many packets are received at the sink. However, we made an assumption that the capacity of the channel and the error rate remained constant in the previous analysis. This may not be the case every time and there could be variations of losses in the channel. Fixing the code rate based on a prior estimation of the channel throughout the communication time is not the ideal solution in those cases. Although the network coding solutions we saw before can be reactive according to the feedback acknowledgments (i.e. causal as given for example in [5]), none of those solutions track the varying channel condition and the rate (i.e. not adaptive). In this section, we discuss the adaptive and causal version of RLNC (AC-RLNC) for a single link communication channel with delayed feedback. The proposed model can track the erasure pattern of the channel, and adaptively adjust its retransmission rates a priori and posteriori based on the channel quality (the erasure burst pattern) and the feedback acknowledgments. That is, while according to the actual rate of the channel, first, a priori algorithm sends an adaptive amount of retransmissions periodically. At each transmission by posteriori algorithm, the sender adaptively and causally decides if to send a retransmission or a coded packet that contains new data information.

The simulation results for the implementation of AC-RLNC presented in [6] demonstrate the robustness of the algorithm. They also show that in real wireless scenarios, in addition to the improvement in throughput, the gap between the mean in order delay and the maximum in order delay is very small unlike the one in Selective Repeat ARQ (SR-ARQ) where the growth rate of the gap is higher. Consistent with these, they validated the performance of data delivery of the algorithm via experimental simulations under the traces of Intel in [6].

We consider an adaptive, causal, real-time, slotted communication model with feedback (i.e. AC-RLNC) for low-latency constraints. Figure 6.6 shows the system model. We consider the case where erasures may occur over the forward channel. In each time slot t the sender transmits a coded packet c_t over the forward channel to the receiver. To simplify the technical aspects and focus on the key methods, we assume that the feedback channel is noiseless. The receiver acknowledges the sender for each coded packet transmitted over the feedback channel. Denoted by t_p is the maximum propagation delay over any channel, and by $t_d = |c_t|/r$ is the transmission delay of the packet, where $|c_t|$ is the size of each coded packet in bits and r denotes the rate of the channel in bits/second. Since the sender transmits one coded packet per time slot, t_d is also the duration of a time slot. We assume the size of the acknowledgments is negligible compared to the packet size. Hence, the RTT is

$$\text{RTT} = t_d + 2t_p. \tag{6.3}$$

For each tth coded packet it transmits, the sender receives a reliable ACK or negative ACK (NAK), which indicates that the packet was not received, after RTT. The goal of AC-RLNC scheme is to minimize the in-order delivery delay, D, and maximize the throughput, η.

6.2.1 Adaptive Coding Algorithm

Here we detail the AC-RLNC given in Algorithm 6.2. AC-RLNC differs from SR-ARQ in terms of the structure of the feedback and the retransmission criterion, which is mainly affected by the feedback, the window size, and the forward channel conditions. Namely, the sender tracks the channel rate and the DoF rate at the receiver via the feedback acknowledgments and selects whether to add a new

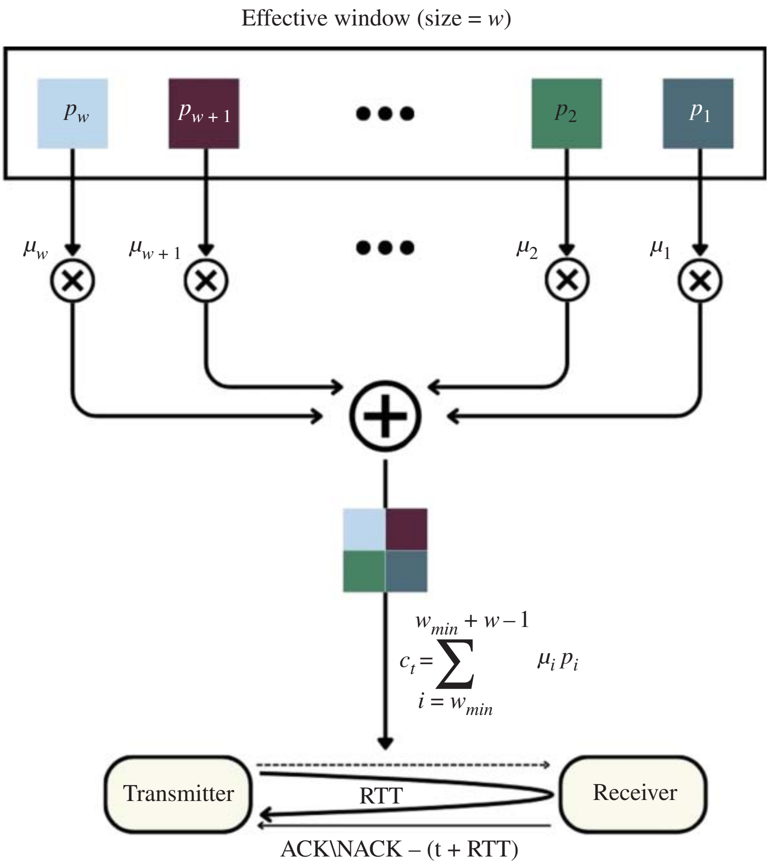


Figure 6.6 System model and encoding process of the coded RLNC combination. The adaptive causal encoding process and the effective window size w are detailed in Section 6.2.1. In this example, for simplicity of notation $w_{min} = 1$. This figure is inspired by joint work with Alejandro Cohen [6].

information packet to the next coded RLNC packet it sends. Figure 6.6 shows the system model and the adaptive causal encoding process of the AC-RLNC protocol with an effective window size w. The symbol definitions and an example realization of AC-RLNC are provided in Table 6.1 and Fig. 6.7, respectively. A detailed explanation of the algorithm and its performance analysis can be found in the paper [6]. Here we quickly go through the example realization to get the gist of it. This provides a detailed step-by-step description of the AC-RLNC algorithm realization in Fig. 6.7.

1) In Fig. 6.7, we show a coding matrix using AC-RLNC with feedback. In each time slot t, the sender transmits a coded packet c_t to the receiver.

Algorithm 6.2 Adaptive causal RLNC for packet scheduling.

```
1: while DoF($c_t$) > 0 do
2:     $t = t + 1$
3:     Update $d = \frac{m_d}{a_d}$ according to the known encoded packets
4:     if no feedback then
5:         if EW then
6:             Transmit the sameRLNC $m$ times: $a_d = a_d + m$
7:         else
8:             Add new $p_i$ packet to the RLNC and transmit
9:         end if
10:    else if feedback NACK then
11:        $e = e + 1$
12:        Update $m_d$ according to the known encoded packets
13:        if $r - d > th$ then
14:            if not EW then
15:                Add new $p_i$ packet to the RLNC and transmit
16:            else
17:                Transmit the same RLNC $m$ times: $a_d = a_d + m$
18:            end if
19:        else
20:            Transmit the same RLNC $a_d = a_d + 1$
21:            if EW then
22:                Transmit the same RLNC $m$ times: $a_d = a_d + m$
23:            end if
24:        end if
25:    else
26:        if EW then
27:            Transmit the same RLNC $m$ times: $a_d = a_d + m$
28:        end if
29:        if $r - d < th$ then
30:            Transmit the same RLNC $a_d = a_d + 1$
31:        else
32:            Add new $p_i$ packet to the RLNC and transmit
33:        end if
34:    end if
35:    Eliminate the seen packets from the RLNC
36:    if DoF($c_t$) > $\bar{o}$ then
37:        transmit the same RLNC until DoF($c_t$) = 0
38:    end if
39: end while
```

The receiver sends feedback (without any losses or delay) on each coded packet transmitted. For the tth coded packet transmitted, the sender receives a reliable acknowledgment (e.g. $ACK(t)$ or $NACK(t)$) after an RTT. For this particular example, RTT is 4 time slots.

Table 6.1 AC-RLNC algorithm: symbol definition.

Parameter	Definition
t	Time slot index
M	Number of information packets
p_i, $i \in [1, M]$	Information packets
c_t	RLNC to transmit at time slot t
$\mu_i \in \mathbb{F}_z$	Random coefficients
e	Total number of erasures in $[1, t]$
$D_{\text{mean}}, D_{\text{max}}$	The mean and maximum in order delivery delay of packet
p_e	Erasure probability
m_d	Number of DoFs needed by the receiver to decode c_t
a_d	DoF added to c_t
d	Rate of DoF (m_d/a_d)
r	Rate of the channel
th	Throughput-delay tradeoff parameter
$r - d > th$	Retransmission criterion
$\text{RTT} = k + 1$	Round-trip time
w_{min}	Index of the first information packet in c_t
EW	End window of k new packets
m	Number of FEC to add per window
$\bar{o}$	Maximum number of information packets allowed to overlap
$w \in \{1, \dots, \bar{o}\}$	Effective window size
E$\bar{o}$W	End overlap window of maximum new packets

2) In this example, at $t = 1$, the sender transmits information packet p_1 to the receiver. The effective window size is $k = 3$ (as $\text{RTT} = k + 1$). Since no feedback has been received from the receiver, and it is not the end of the effective window (EW), the sender includes packet p_1 into the effective window and generates a coded packet c_1 that includes a linear combination of the information packets included in the effective window (e.g. p_1), and transmits c_1 to the receiver. The sender may check to determine whether coded packet c_1

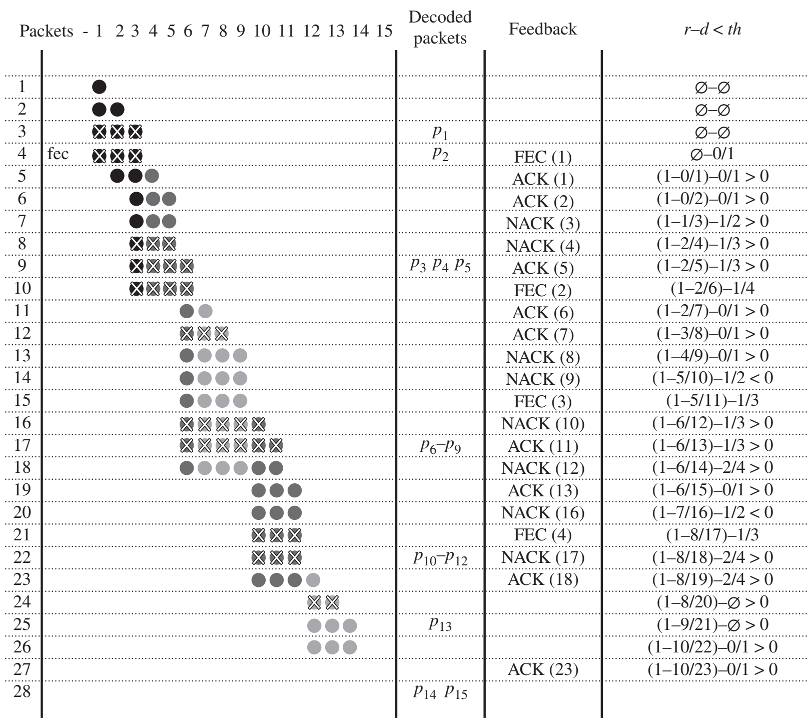

Packets - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Decoded packets
Feedback
r–$d < th$
1 ∅–∅
2 ∅–∅
3 p_1 ∅–∅
4 fec p_2 FEC (1) ∅–0/1
5 ACK (1) (1–0/1)–0/1 > 0
6 ACK (2) (1–0/2)–0/1 > 0
7 NACK (3) (1–1/3)–1/2 > 0
8 NACK (4) (1–2/4)–1/3 > 0
9 p_3 p_4 p_5 ACK (5) (1–2/5)–1/3 > 0
10 FEC (2) (1–2/6)–1/4
11 ACK (6) (1–2/7)–0/1 > 0
12 ACK (7) (1–3/8)–0/1 > 0
13 NACK (8) (1–4/9)–0/1 > 0
14 NACK (9) (1–5/10)–1/2 < 0
15 FEC (3) (1–5/11)–1/3
16 NACK (10) (1–6/12)–1/3 > 0
17 p_6–p_9 ACK (11) (1–6/13)–1/3 > 0
18 NACK (12) (1–6/14)–2/4 > 0
19 ACK (13) (1–6/15)–0/1 > 0
20 NACK (16) (1–7/16)–1/2 < 0
21 FEC (4) (1–8/17)–1/3
22 p_{10}–p_{12} NACK (17) (1–8/18)–2/4 > 0
23 ACK (18) (1–8/19)–2/4 > 0
24 (1–8/20)–∅ > 0
25 p_{13} (1–9/21)–∅ > 0
26 (1–10/22)–0/1 > 0
27 ACK (23) (1–10/23)–0/1 > 0
28 p_{14} p_{15}

Figure 6.7 The rows and columns of the coding matrix denote the time slot t and the information packet i indices, respectively. At a
time slot, a coded packet, which is a random linear combination of packets p_i, is transmitted. The lost packets are shown with crossed
The last column shows how the decision to send posterior repair packets is made. This figure is inspired by joint work with Alejandro
Cohen [6].

satisfies DoF(c_t) $\geq 2k$, and concludes that c_1 does not exceed the allowable DoF.

3) At $t = 2$, the sender wants to transmit information packet p_2 to the receiver. Since no feedback has been received from the receiver, and the effective window does not end with k information packets, the sender includes p_2 into the effective window and generates and transmits a coded packet c_2 that is a linear combination of the packets in the effective window (i.e. p_1 and p_2). The sender can then update the rate of DoF. It also checks whether DoF(c_2) $> 2k$.
4) At $t = 3$, the sender wants to transmit information packet p_3. Since no feedback has been received from the receiver, and the effective window does not end with k new information packets, the sender includes packet p_3 into the effective window and generates and sends a coded packet c_3 that is a linear combination of the information packets included in the effective window (i.e. p_1, p_2, and p_3). The sender then updates the rate of DoF and checks if DoF(c_3) $> 2k$.
5) At $t = 4$, the sender wants to transmit information packet p_4. However, notwithstanding that no feedback has been received from the receiver, the sender checks and determines that the effective window ends with k new information packets (i.e. p_1, p_2, and p_3). As a result, the sender generates and transmits a FEC packet to the receiver. For example, the FEC packet at $t = 4$ can be a new linear combination of p_1, p_2, and p_3. It is also possible that the sender may transmit the same FEC multiple times. We let the number of FECs be m, which can be specified based on the average erasure probability, or provided over the feedback channel. In other words, m is a tunable parameter. In Fig. 6.7, $m = 1$, which results in low throughput when the channel rate is high, and low in order delay when the rate is low. Hence, m can be adaptively adjusted exploiting the value of the average erasure rate in the channel to achieve a desired delay-throughput tradeoff.
6) At $t = 4$, transmission of FEC packet is noted by the designation “fec” in row 4. The sender can then update the DoF added to c_t and the rate of DoF. Note that the transmission of this FEC is initiated by the sender (e.g. the transmission of the FEC packet was not a result of feedback from the receiver). Furthermore, p_4 is not included in the effective window. The sender may check and determine that c_4 does not exceed the allowable DoF (DoF(c_4) $> 2k$).
7) At $t = 5$, the sender determines that it needs to transmit p_4. It also sees the acknowledgment of the receipt of c_1 (denoted by ACK(1) in the Feedback column at row $t = 5$). Hence, the sender can remove a DoF (e.g. p_1) from the effective window. The effective window slides to the right (equivalent to sliding window). In this case, the sender determines that the effective window does not end with k new information packets and checks if the rate r is higher than the DoF rate d, i.e. the threshold condition for retransmission

(e.g. transmission of a FEC) is $th = 0$. Hence, the sender can decide that the channel rate r is sufficiently higher than the DoF rate d (e.g. denoted by $(1 - 0/1) - 0/1 > 0$ in the right-most column at row $t = 5$). Note that $r = 1 - e/t$, where e is the number of erasure packets and t is the number of transmitted packets for which the sender has received acknowledgments. Having determined that $r - d \geq 0$, the sender includes p_4 into the effective window and generates a coded packet c_5 that includes a linear combination of the information packets in the effective window (i.e. p_2, p_3, and p_4), and transmits c_5 to the receiver. The sender then updates the rate of DoF. It may also check that c_4 does not exceed the allowable DoF (DoF(c_4) > $2k$).

8) At $t = 6$, the sender wants to transmit p_5. It also sees the acknowledgment of the receipt of c_2 (denoted by ACK(2) in the Feedback column at row $t = 6$). Hence, the sender can remove a DoF (e.g. p_2) from the effective window. In this case, the sender determines that the effective window does not end with k new information packets (but the effective window ends with new information packet p_4). The sender checks if $r - d \geq 0$. Having determined that $(1 - 0/2) - 0/1 > 0$ (in the right-most column at row $t = 6$), the sender includes p_5 into the effective window and generates and transmits a coded packet c_6 that includes a linear combination of the information packets included in the effective window (e.g. p_3, p_4, and p_5). The sender then updates the rate of DoF. It also checks that c_6 does not exceed the allowable DoF (DoF(c_6) > $2k$).
9) At $t = 7$, the sender wants to transmit p_6. The sender also sees a negative acknowledgment indicating the non-receipt of c_3 (denoted by NACK(3) in the Feedback column at row $t = 7$). The non-receipt of c_3 is denoted by the x'ed out dot in row 3. Upon the reception of NACK, the sender increments a count of the number of erasures (denoted as e). For example, since this is the first erasure, the count of e is incremented to the value of one. The sender then updates the missing DoF to decode c_7 according to the number of NACKs received. In this instance, the sender updates the missing DoF to decode c_7 to a value of one (1) since this is the first NACK. The sender then checks if the retransmission condition is satisfied. Since $(1 - 1/3) - 1/1 < 0$, it generates and transmits an FEC packet c_7 to the receiver. In this instance, the FEC packet c_7 is a new linear combination of p_3, p_4, and p_5. Note that the threshold condition $(1 - 1/3) - 1/2 < 0$ noted in the right-most column at row $t = 7$ indicates the state of the threshold condition subsequent to transmission of c_7. The transmission of this FEC is a result of feedback by the receiver (which is noted as "FB-FEC" in row $t = 7$). The sender then updates the DoF added to c_7. The sender determines that the effective window does not end with k new information packets (e.g. effective window ends with new information packets p_4 and p_5) and then updates the rate of DoF. It also checks that c_7 does not exceed the allowable DoF (DoF(c_7) > $2k$).

10) At $t = 8$, the sender determines that it still needs to send p_3 to the receiver. The sender also sees a negative acknowledgment indicating the non-receipt of c_4 (denoted by NACK(4) in the Feedback column at row $t = 8$). Upon the receipt of NACK, the sender increments a count of e. In this instance, since this is the second erasure, the count of e is incremented by one to a value of two (i.e. $e = 1 + 1$). The sender then updates the missing DoF to decode c_7. After it checks that the retransmission threshold is not satisfied, the sender generates and transmits an FEC packet c_8 to the receiver. The FEC packet c_8 is a new linear combination of p_3, p_4, and p_5. The transmission of this FEC is a result of feedback by the receiver (which is noted as "fb-fec" in row $t = 8$). The sender then updates the DoF added to c_8 and determines that the effective window does not end with k new information packets (effective window ends with new information packets p_4 and p_5). The sender then updates the rate of DoF. It may also check that c_8 does not exceed the allowable DoF (DoF(c_8) $> 2k$).
11) At $t = 9$, the sender determines that it still needs to send p_3 to the receiver. The sender also sees an acknowledgment indicating the receipt of c_5 (denoted by ACK(5) in the Feedback column at row $t = 9$). Since sender sees an acknowledgment, and determines that the effective window does not end with k new information packets (effective window ends with new information packets p_3, p_4, and p_5), the sender checks if $r - d \geq 0$. Having determined that $(1 - 2/5) - 1/3 > 0$ (as shown in the right-most column at row $t = 9$), the sender includes packet p_6 into the effective window, and generates and transmits a coded packet c_9 that includes a linear combination of the information packets included in the effective window (i.e. p_3, p_4, p_5, and p_6). The sender then updates the rate of DoF. It may also check that c_9 does not exceed the allowable DoF (DoF(c_9) $> 2k$).
12) At $t = 10$, the sender determines that it needs to send p_3 to the receiver. The sender also sees an acknowledgment indicating the receipt of c_6 (denoted by ACK(6)). Since the sender sees an acknowledgment, it determines that the effective window ends with information packets p_3, p_4, p_5, and p_6. As a result, the sender generates and transmits a FEC packet c_{10}. The sender then updates the added DoF (i.e. $ad = 3 + 1$). Now, since $(1 - 2/6) - 1/4 > 0$, the sender includes packet p_7 into the effective window. Then it generates a coded packet c_{11} to transmit at time slot 11 that includes a linear combination of the information packets in the effective window (i.e. p_3, p_4, p_5, p_6, and p_7). It may also check that c_{11} does not exceed the allowable DoF (DoF(c_{11}) $> 2k$).
13) At $t = 11$, the sender transmits c_{11}. However, according to the acknowledgment indicating the receipt of c_7 (denoted by ACK(7)), the sender removes DoF (e.g. information packets p_3, p_4, and p_5) from the effective window. The sender then updates the rate of DoF.

14) At $t = 12$, the sender sees an acknowledgment indicating the receipt of c_7 (denoted by ACK(7) in the Feedback column at row $t = 12$). Since the sender sees an acknowledgment and determines that the effective window does not end with k new information packets (effective window ends with new information packet p_7), the sender checks if $r - d \geq 0$. Having determined that $(1 - 3/8) - 0/1 > 0$ (as shown in the right-most column at row $t = 12$), the sender includes packet p_8 into the effective window. Then the sender generates and transmits a coded packet c_{12}. This coded packet includes a linear combination of the information packets available in the effective window (i.e. p_6, p_7, and p_8). It may also check that c_{12} does not exceed the allowable DoF ($\text{DoF}(c_{12}) > 2k$).
15) The rest of the example follows using similar steps. The sender continues sending information or coded packets during time slots $t \in [13, 28]$. Hence, the sender can adaptively adjust its transmission rate according to the process described above.

This adaptive algorithm also provides more flexibility to customize network coding solutions depending on the application requirements. We explore more of such cases with the QUIC protocol, which we briefly discussed in the beginning of the last chapter on protocol designs, Chapter 5.

6.3 Network Coding and QUIC

Originally introduced in 2012 by Jim Roskind at Google [7], QUIC aims to be almost equivalent to a TCP connection but with much reduced latency and uses UDP for this purpose. With specifications finalized in 2021, it is already supported by more than 30 M domains on the Internet and gaining a lot of attention. Two notable changes in QUIC rely on the understanding of typical Hyper Text Transfer Protocol (HTTP) traffic. (a) Many HTTP connections will demand transport layer security, so QUIC makes the exchange of setup keys and supported protocols part of the initial handshake process. Receiver response packet includes the data needed for future packets to use encryption. This eliminates the need to set up the TCP connection and then negotiate the security protocol via additional packets. (b) HTTP opens multiple parallel streams. Each QUIC stream is separately flow controlled, and lost data is retransmitted not based on TCP. Data is free to be processed while any individual multiplexed stream is repaired.

Some other tricks that make QUIC interesting are

- the packets are individually encrypted unlike in TCP;
- different connection ID sensibility to allow, e.g. mobile handover; etc.

while issues included (as ever) legacy interaction, systems dimensioned for TCP, systems that block UDP, etc.

Focusing on the reliability mechanism, QUIC provides a reliable bytestream abstraction using *streams*. Application data transiting through QUIC streams is carried inside `STREAM` frames. Upon detection of a loss, the application data carried by the lost `STREAM` frames is retransmitted in new `STREAM` frames sent in new QUIC packets. There are ongoing discussions within the Internet Engineering Task Force (IETF) [8] and the research community [9–11] to add forward erasure correction (FEC) capabilities to QUIC. It is especially useful for applications that cannot afford to wait for a retransmission, either due to strong delay requirements or connections suffering from long delays. Google experimented with a naive XOR-based FEC solution in early versions of QUIC that does not enable sending several repair symbols to protect a window of packets, preventing the solution from recovering loss bursts. QUIC–FEC [9] proposes several redundancy frameworks and codes (such as XOR, Reed–Solomon, and random linear codes (RLC)) for the QUIC protocol. QUIC–FEC showed that FEC with QUIC can be beneficial for small file transfers but is harmful for longer bulk transfers compared to SR-ARQ mechanisms. One of the main limitations of QUIC–FEC [9] is that the code rate is fixed during the connection, leading to the sending of unnecessary coded packets. **Pluginized QUIC (PQUIC)** [10] proposed a FEC plugin equivalent to the RLC part of QUIC–FEC.

Similarly, network coding solutions such as Coded TCP (CTCP) [12] and TCP/NC [13] adjust the level of redundancy according to the measured loss rate. While these approaches can show significant benefits compared to classical retransmission mechanisms, they still increase the overhead compared to the more efficient selective-repeat mechanisms in bulk download use cases. We advocate that sending redundancy packets in transport protocols **should be done in adequacy with the application needs in order to provide satisfactory results.** The AC-RLNC algorithm with pluginized QUIC can adapt to both the channel conditions as well as the application requirements, as in [14]. We leverage the idea of protocol plugins [10] and implement the reliability mechanism as a framework exposing two anchor points to applications. Applications can redefine these anchors according to their delay sensitivity and traffic pattern.

Here we move further from a one-size-fits-all solution to an application-tailored reliability mechanism. We explore three different use cases and show that adapting the reliability mechanism to the use case can drastically improve the quality of the transmission. The first use case is the bulk download scenario, discussed in Section 6.3.2. The second use case discussed in Section 6.3.3 is a scenario where the peer's receive window is small, resulting in the sender being regularly blocked by the flow control during loss events. The third use case discussed in Section 6.3.4 is a scenario where the application sends messages that must arrive before a specific deadline. Let us explore how this works!

6.3.1 Flexible Erasure Correction (FlEC)

In this section, we present the **flexible erasure correction (FlEC)** framework. FlEC starts from the previous theoretical work, AC-RLNC [6], and builds upon the PQUIC [10]. AC-RLNC provides a decision mechanism to schedule repair symbols depending on the network conditions and the feedback received from the receiver. In this approach, repair symbols are sent in reaction to two thresholds: the first is triggered as a function of the number of missing degrees of freedom by the receiver, and the second threshold sends repair symbols *once every RTT*. The original goal of the proposed algorithm [6] is to trade bandwidth for minimizing the in-order delivery delay of data packets.

We start from this idea of tracking the sent, seen, and received degrees of freedom as a first step to propose a redundancy scheduler for the transport layer. However, while this first idea provides a general behavior, this may be insufficient for real applications with tight constraints that cannot be expressed with AC-RLNC's parameters. For example, a video-conferencing application may prefer to maximize bandwidth over low-delay links and therefore rely on retransmissions only, while FEC is needed over high-delay links as such retransmissions cannot meet the application's delay constraints. Instead of proposing configurable constant thresholds to tune the algorithm, we make it dynamic by proposing two redefinable functions: *ds*() (for "*delay-sensitivity*") and *FECPattern*(). These two functions can be completely redefined to instantiate a reliability mechanism closely corresponding to the use case. This allows having completely different FEC behaviors for use cases with distinct needs such as HTTP versus video-conferencing. The *ds*() threshold represents the sensitivity of the application to the in-order delivery delay of the data sent. In AC-RLNC, the FEC scheduler sends redundancy once per RTT. In FlEC, the *FECPattern*() dynamic function allows triggering the sending of FEC at specific moments of the transfer depending on the use case. Sending FEC for every RTT may deteriorate the application performance, especially when the delay is low enough to rely on retransmissions only. Having a dynamic *FECPattern*() function avoids this problem. For instance, in a bulk download scenario, it can trigger FEC at the end of the download only and rely on retransmissions otherwise. For video transfer, it can trigger FEC after each video frame is sent. Figure 6.8 illustrates the idea of FlEC. The regular QUIC reliability mechanism is based on SR-ARQ. In FlEC, the SR-ARQ mechanism is a particular case among many other possibilities. Algorithm 6.3 shows our generic framework. We implement FlEC using PQUIC [10] and define *FECPattern*() and *ds*() as protocol operations. However, the

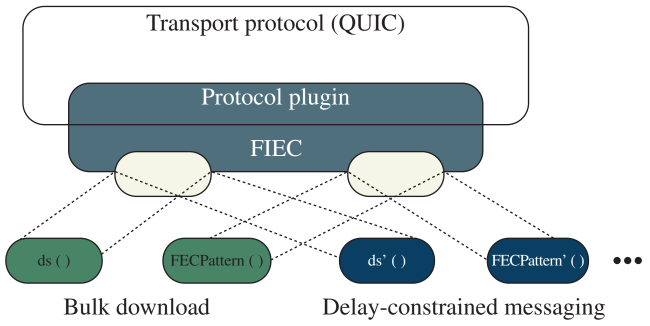


Figure 6.8 Design of the solution: a general framework with two pluggable anchors to redefine the reliability mechanism given the use case. This is inspired from [14].

Algorithm 6.3 Generic redundancy scheduler algorithm. The *ds*() and *FECPattern*() thresholds are redefined by the underlying application. The algorithm is called at each new available slot in the congestion window of the protocol. $\hat{l}$ and $\hat{r}$ represent the estimated uniform loss rate and the estimated receive rate, respectively.

```
Require: $\hat{l}$
Require: $feedback$, the most recent feedback received from the peer
Require: $W$, the current coding window
 1: $\hat{r} \leftarrow 1 - \hat{l}$
 2: $ad \leftarrow computeAd(W)$
 3: $md \leftarrow computeMd(W)$
 4: if $feedback = \varnothing$ then
 5:     if FECPattern() then
 6:         return $NewRepairSymbol$
 7:     else
 8:         return $NewData$
 9:     end if
10: else
11:     updateLossEstimations(feedback)
12:     if FECPattern() then
13:         return $NewRepairSymbol$
14:     else if $\hat{r} - \frac{md}{ad} < ds()$ then
15:         return $NewRepairSymbol$
16:     else
17:         return $NewData$
18:     end if
19: end if
```

same principles can be applied without PQUIC with the application redefining the operations natively thanks to the user-space nature of QUIC.

The *computeMd* function computes the number of missing degrees (*md*) of freedom (i.e. missing source symbols) in the current coding window. The *computeAd* function computes the number of added degrees (*ad*) of freedom (i.e. repair symbols) that protect at least one packet in the current coding window. Compared to AC-RLNC, we only consider in-flight repair symbols in *ad* to support retransmissions when repair symbols are lost. The higher the value returned by *ds*(), the more likely it is to send repair symbols prior to the detection of a lost source symbol, and the more robust is the delay between the sending of the source symbols and their arrival at the receiver. The extra cost is the bandwidth utilization. Sending repair symbols *a priori* occupies slots in the congestion window and is likely to increase the delay between the generation of data in the application and its actual transmission. Setting *ds*() to $-\hat{l}$ triggers the transmission of repair symbols only in reaction to a newly lost source symbol, thus implementing a behavior similar to regular QUIC retransmissions. In this work, retransmissions are done using repair symbols to illustrate that the approach is generic. However, regular uncoded retransmissions can be used for better performance without loss of generality. *FECPattern*() allows regulating the transmission of *a priori* repair symbols regardless of the channel state, in contrast with AC-RLNC [6] where this threshold is triggered once per RTT.

Table 6.2 describes how *ds*() and *FECPattern*() can be redefined to represent reliability mechanisms that fit the studied use cases. The first row of the table shows how to implement the classical Selective-Repeat ARQ mechanism used by default in QUIC. The second one implements the behavior of AC-RLNC [6]. *FECPattern*() is triggered once every RTT according to the EW parameter of AC-RLNC. The third one is tailored for the bulk use case: *ds*() is set to send Repair Symbols only when there are missing symbols at the receiver and *FECPattern*() sends Repair Symbols when there is no more data to send. The two other rows are explained in detail in sections 6.3.3 and 6.3.4 respectively. In this table, *c* is a non-negative user-defined constant. The higher *c* is, the more sensitive we are to a variance in the loss rate.

Table 6.2 Definition of *ds*() and *FECPattern*() for the considered use cases.

Use case	***ds*()**	***FECPattern*()**
Bulk transfer (SR-ARQ)	$-\hat{l}$	*false*
AC-RLNC [6]	$c \cdot \hat{l}$	*true* every RTT
Bulk transfer	$-\hat{l}$	*allStreamSent*()
Buffer-limited bulk	$c \cdot \hat{l}$	Algorithm 6.4
Messaging	$-\hat{l}$	Algorithm 6.6

6.3.2 Bulk File Transfers

Bulk file transfer is the simplest use case we consider. It consists of the download of a single file under the assumption that the receive buffer is large compared to the BDP of the connection. This is the classical use case for many transport protocols. Current open-source QUIC implementations use default receive window sizes that support such a use case. The receive window starts at 2 MB for locally initiated streams in `picoquic` [15]. The Chromium browser's implementation [16] starts with an initial receive window of 6 MB per stream and 16 MB for the whole connection. The metric that we minimize here is the total time to download the whole file. This includes REST API messages that often need to be completely transferred in order to be processed correctly by the application. A packet loss during the last round-trip can have a high relative impact on the download completion time. Protecting these tail packets can drastically improve the total transfer time at a cost significantly smaller than the cost of simply duplicating all these packets. On the other hand, protecting other packets than the tail ones with FEC can be harmful for the download completion time. The packet losses in the middle of the download can be recovered without FEC before any quiescence period provided that the receiver uses sufficiently large receive buffers. The expected behavior is therefore similar to SR-ARQ with tail loss protection. The Repair Symbols are always sent within what is allowed by the congestion window, meaning that *FlEC* does not induce any additional link pressure.

For a file transfer, we set the delay-sensitivity threshold to be equal to $-\hat{l}$.

$$\hat{r} - md/ad < -\hat{l} \rightarrow \mathit{sendRepairPacket}(). \qquad (6.4)$$

Substituting $\hat{l}$ by $(1 - \hat{r})$ in Eq. (6.4), we can rewrite it as

$$md/ad > 1 \rightarrow \mathit{sendRepairPacket}(),$$

so that we send Repair Symbols only when a packet is detected as lost and it has not been protected yet. The transmission of a Repair Symbol triggered by this threshold increases *ad* by 1 until *ad* becomes equal to *md*. Using the threshold defined in Eq. (6.4) ensures a reliable delivery of the data but does not improve the download completion time in the case of tail losses. *md* only increases after a packet is marked as lost by the QUIC loss detection mechanism. The *FECPattern*() operation controls the *a priori* transmission of Repair Symbols. In contrast with the previous solution [6], we redefine *FECPattern*() and set it to *true* only when all the application data has been sent instead of setting it to *true* once per RTT. This implies that only the last flight of packets will be protected. All the previous flights will be recovered through retransmissions. Indeed, given the fact that the receive window is large enough compared to the congestion window, there will be no silence period implied by any packet loss except for the last flight of packets. The total download completion time will thus not be impacted by any loss before the last flight of packets. Without using FEC, the loss of any packet in the last flight will cause a silence period between the sending of that lost packet and its retransmission. We track the loss conditions throughout the download and trigger the *FECPattern*() threshold according to the observed loss pattern. This loss-rate-adaptive approach is especially beneficial when enough packets are exchanged to accurately estimate the loss pattern. This occurs when the file is long or when loss information is shared among connections with the same peer. When a sufficient number of repair symbols are sent to protect the expected number of lost source symbols, the algorithm keeps slots in its congestion window to transmit new data.

6.3.3 Buffer-limited File Transfers

In numerous network configurations, the available memory on the end devices is a limiting factor. It is common to see delays longer than 500 ms in satellite communications, while their bandwidth is in the order of several dozens of megabits per second [17, 18]. Furthermore, with the arrival of 5G, some devices will have access to bandwidth up to 10 Gbps [19, 20]. While the edge latency of 5G infrastructures is intended to be in the order of a few milliseconds [21], the network toward the other host during an end-to-end transport connection may be significantly higher, partially due to the large buffers on the routers and the buffer-filling nature of currently deployed congestion control mechanisms. Packet losses occurring on those high BDP network configurations imply a significant memory pressure on reliable transport protocols running on the end devices. To ensure an in-order delivery, the transport protocol running on these devices needs to keep the data received out-of-order during at least one round-trip-time, requiring receive buffer sizes to grow to dozens of megabytes for each connection. Receive buffers that cannot bear the BDP of the network they are attached to are unable to fully utilize its capacity, even without losses. This typically occurs when the receive window (*rwin*) is relatively small compared to the sender's congestion window (*cwin*). Measurements show that TCP receivers frequently suffer from such limitations [22]. The problem gets even worse in case of packet losses as they prevent the receiver from delivering the data received out-of-order to the application. The data will remain in memory, reducing the amount of new data that can be sent until the lost data is correctly retransmitted and delivered to the application. Sacrificing a few bytes of the receiver memory in order to handle repair symbols and protect the receive window from being blocked upon packet losses can drastically improve the transfer time, even in a file transfer use case. In such cases, FEC can be sent periodically along with non-coded packets during the download and not just at the end of the transfer.

For this use case, *ds*() returns $\hat{l}$ to ensure that *ad* stays larger than *md*, according to the estimated loss rate. *FECPattern*() behaves as shown in Algorithm 6.4. We spread the Repair Symbols along the sent Source Symbols in order to periodically allow the receiver to unblock its receive window by recovering the lost Source Symbols and delivering the stream data in-order to the application. More precisely, the *FECPattern*() operation sends one Repair Symbol every $\frac{1}{\hat{l}}$ Source Symbols. The algorithm needs three loss statistics. The first is the estimated uniform loss rate $\hat{l}$. The two others are the $\hat{G}_p$ and $\hat{G}_r$ parameters of the Gilbert loss model. The Gilbert model [23] is a two-state Markov model representing the channel, allowing representing network configurations where losses occur in bursts. These loss patterns cannot be easily recovered by a simple XOR error correcting code as shown in the original QUIC article [22] but can be

Algorithm 6.4 FECPattern for buffer-limited use case.

```
Require: last, the ID of the last symbol present in the coding window when
    FECPattern() was triggered the last time
Require: nTriggered, the number of times FECPattern() has already been triggered
    since no new symbol was added to the window.
Require: maxTrigger, the maximum number of times we can trigger this threshold
    for the same window
Require: nRSInFlight, the number of Repair Symbols currently in flight
Require: W, the current coding window.
Require: FCBlocked(), telling us if we are currently blocked by flow control.
Require: $\hat{l}$, $\hat{G}_p$, $\hat{G}_r$
 1: if $nRSInFlight \geq 2 * \lceil |W| * \hat{l} \rceil$ then
 2:     return false                          ▷ Wait for feedback before sending new RS
 3: end if
 4: $nUnprotected \leftarrow W.last - last$
 5: $n \leftarrow min(\frac{1}{\hat{G}_p}, |W|)$
 6: $protect \leftarrow nUnprotected = 0 \lor nUnprotected \geq n \lor FCBlocked()$
 7: if $protect \land nUnprotected \neq 0$ then       ▷ Start Repair Symbols sequence
 8:     $nTriggered \leftarrow 1$
 9:     $last \leftarrow W.last$
10:     $maxTrigger \leftarrow \lceil max(\hat{l} * nUnprotected, \frac{1}{\hat{G}_r}) \rceil$
11: else if protect then
12:     if $FCBlocked() \lor nTriggered < maxTrigger$ then
13:         $nTriggered \leftarrow nTriggered + 1$      ▷ Continue sending symbols
14:     else
15:         $protect \leftarrow false$              ▷ Enough symbols have been sent
16:     end if
17: end if
18: return protect
```

recovered by the random linear codes used by *FlEC*. In the *GOOD* state of the Gilbert model, packets are received while the packets are dropped in the *BAD* state. $\hat{G}_p$ is the transition probability from the GOOD to the BAD state while $\hat{G}_r$ is the transition probability from the BAD to the GOOD state. In order to estimate the loss statistics $\hat{l}$, $\hat{G}_p$, and $\hat{G}_r$, we implement a *loss monitor* that estimates the loss rate and Gilbert model parameters over a QUIC connection.

When the sender is blocked by the QUIC stream flow control, *FECPattern*() sends more Repair Symbols to recover from the remaining potentially lost Source Symbols. While spreading the Repair Symbols along the coding window helps to recover the lost Source Symbols more rapidly compared to a block approach where all the repair symbols are sent at the end of the window, this also potentially consumes more bandwidth. Indeed, the Repair Symbols do not protect the entire window. This means that with an equal number of losses, some specific loss patterns will lead to Repair Symbols protecting a portion of the window with no loss and portions of the window requiring more Repair Symbols to be recovered.

6.3.4 Delay-constrained Messaging

Finally, we consider applications with real-time constraints such as video conferencing. Those applications send messages (e.g. video frames) that need to be successfully delivered within a short amount of time. The metric to optimize is the number of messages delivered on-time at the destination.

FEC can significantly improve the quality of such transfers by recovering from packet losses without retransmissions, at the expense of using more bandwidth. Researchers have already applied FEC to video applications [24, 25]. Some [24] take a redundancy rate as input and allocates the Repair Symbols given the importance of the video frame. Others [25] propose a congestion control scheme that reduces the impact of isolated losses on the sending rate. They then use this congestion control to gather knowledge from the transport layer to the application in order to adapt the transmission to the current congestion. Here the approach is its inverse: the application transfers its knowledge directly into the transport protocol to automatically adjust its stream scheduler and redundancy rate given the application's requirements. The goal is to protect whole messages instead of naively interleaving Repair and Source Symbols. Using application knowledge, *FlEC* protects as many frames as possible at once.

We consider an application sending variable-sized messages, each having its own delivery deadline. To convey these deadlines, we extend the transport API to an application-specific API. Furthermore, we replace the QUIC stream scheduler with an application-tailored stream scheduler to leverage application information.

This can be done easily since applications are bundled with their QUIC implementation and are able to easily extend it.

We propose the following API enabling an application to send deadline-constrained messages.

`send_fec_protected_msg(msg, deadline)`
The application submits its deadline-constrained messages. The QUIC protocol already supports the stream abstraction as an elastic message service. However, the stream priority mechanism proposed by QUIC, while being dynamic, is not sufficient to support message deadlines. The protocol operation attached to this function inserts the bytes submitted by the application in a new QUIC stream, closes the stream, and attaches the application-defined delivery deadline to it. The message must be delivered at the receiver within this amount of time to be considered useful. If the network conditions prevent an on-time delivery of the message, the message may be cancelled, possibly before being sent and the underlying stream be reset.

`next_message_arrival(arrival_time)`
This API call allows the application to indicate when it plans to submit the next message. While this API function is not useful for all kinds of unreliable messaging applications, applications having a constant message sending rate such as videoconferencing might benefit from providing such information.

The knowledge provided by the application to the transport layer is not only useful for the coded reliability mechanism. The information provided by the application-defined API calls is also valuable for the QUIC stream scheduler. Without this information, the QUIC scheduler schedules high-priority streams first and has two different ways to handle the scheduling of streams with the exact same priority: (a) round-robin or (b) First In – First Out (FIFO). We let the application define its own scheduler to schedule its streams more accurately. Algorithm 6.5 describes our QUIC stream scheduler for deadline-constrained messaging applications.

The *closestDeadlineStream*() function searches among all the available streams attached to a deadline to find the stream having the closest expiration deadline while still having chances to arrive on-time given the current one-way delay. The scheduler chooses the non-flow-control blocked stream that is the closest to expire while still having a chance to be delivered on-time to the destination. Our implementation estimates the one-way delay as $\frac{RTT}{2}$. Other methods exist [26, 27]. Recent versions of `picoquic` include a mechanism for estimating the one-way delay [28] when the hosts clocks are synchronized. In the absence of clock synchronization, the estimated one-way delay can only be interpreted

Algorithm 6.5 Application-tailored scheduler for delay-constrained messaging.

```
Require: $\mathcal{S}$, the set of available QUIC streams
Require: $\hat{OWD}$, the estimated one-way delay of the connection
Require: $now$, the timestamp representing the current time
Require: $FCBlocked(stream)$, telling if the specified stream is flow control-blocked.
Require: $closestDeadlineStream(\mathcal{S}, deadline)$, returning the non-expired stream with
    the closest delivery deadline to the specified deadline
 1: $scheduledStream \leftarrow \varnothing$
 2: $currentDeadline \leftarrow now + \hat{OWD}$                              ▷ Initialization
 3: while $scheduledStream = \varnothing$ do
 4:     $candidate \leftarrow closestDeadlineStream(\mathcal{S}, currentDeadline)$
 5:     if $candidate = \varnothing$ thenbreak
 6:     end if
 7:     if $\neg FCBlocked(candidate)$ then
 8:         $scheduledStream \leftarrow candidate$
 9:     else
10:         $\mathcal{S} \leftarrow \mathcal{S} \setminus \{candidate\}$
11:         $currentDeadline \leftarrow candidate.deadline$
12:     end if
13: end while
14: if $scheduledStream = \varnothing$ then
15:     return $defaultStreamScheduling(\mathcal{S})$                          ▷ Fallback
16: end if
```

relatively, which helps to estimate the one-way delay variation but not for decision thresholds such as the one used in Algorithm 6.5.

Let us examine *FECPattern*() and *ds*() for delay-constrained messaging. We now describe how our application redefines *FlEC*. Our application is sensitive to the delivery delay of entire messages more than the in-order delivery delay of individual packets. We thus set the *ds*() threshold to $-\hat{l}$ as it is useful to retransmit non-recovered lost packets that can still arrive on-time. *FECPattern*() is described in Algorithm 6.6. The algorithm triggers the sending of Repair Symbols to protect as many messages as possible according to the messages deadline and the next expected message timestamp if provided by the application. The rationale is the following. If the unprotected messages can wait for new messages to arrive before being protected, *FECPattern*() does not send Repair Symbols and waits for the arrival of new messages. Otherwise, Repair Symbols are sent to protect the entire window until it is considered fully protected. This idea of waiting for new messages before protecting comes from the fact that the messages can be small and sending Repair Symbols for each sent message can lead to a high overhead. By doing so, *FECPattern*() adapts the code rate according to the application needs.

Algorithm 6.6 *FECPattern*() for delay-constrained messaging.

```
Require: $\mathcal{S}$, the set of available QUIC streams
Require: $\hat{OWD}$, the estimated one-way delay of the connection
Require: $now$, the timestamp representing the current time
Require: $closestDL(\mathcal{S}, deadline)$, returning the message deadline that will expire the
    sooner from the specified deadline
Require: $last$, the last protected message.
Require: $nTriggered$, the number of times FECPattern has already been triggered
    since no symbol was added to the window.
Require: $maxTrigger$, the maximum number of times we can trigger this threshold
    for the same window
Require: $nextMsg$ (is $+\infty$ if the message API is not plugged), the maximum amount
    of time to wait before a new message arrives.
Require: $cwin$, $bif$, the congestion window and bytes in flight.
Require: $\theta$ space to save in cwin for directly upcoming messages.
Require: $\hat{l}$, $\hat{G}_p$, $\hat{G}_r$
 1: $nextDL \leftarrow closestDL(\mathcal{S}, max(now + \hat{OWD}, last.deadline))$
 2: $protect \leftarrow (nextDL = \varnothing \vee now + \hat{OWD} + nextMsg + \epsilon \geq nextDL)$
 3: $nUnprotected \leftarrow W.last - last$
 4: if $protect \wedge nUnprotected \neq 0$ then                ▷ Start Repair Symbols sequence
 5:     $nTriggered \leftarrow 1$
 6:     $last \leftarrow W.last$
 7:     $maxTrigger \leftarrow \lceil max(\hat{l} * nUnprotected, \frac{1}{\hat{G}_r})\rceil$
 8: else if $protect$ then
 9:     if $nTriggered < maxTrigger$ then
10:         $nTriggered \leftarrow nTriggered + 1$                 ▷ Continue sending
11:     else
12:         $protect \leftarrow false$                     ▷ Enough symbols have been sent
13:     end if
14: end if
15: return $appLimited() \wedge protect \wedge \frac{cwin}{bif} > \theta$
```

Exercise 6.3 Comparing Repair Generation Schemes

In this exercise, we will compare two repair generation schemes in a video streaming scenario where video frames are produced with varying numbers of symbols. The first scheme maintains a fixed-size block and generates repair only when a block is full, while the second scheme can generate repair at any point in time (e.g. after each video frame).

1) Implement Fixed-Size Block Scheme:

- Implement a simulation to calculate the amount of latency accumulated by the fixed-size block scheme, where repair is generated only when a block is full.
- Consider the delay induced by waiting for a block to be filled before generating repair.

2) Implement On-the-Fly Repair Generation Scheme:
   - Implement a simulation to compare the latency of the on-the-fly repair generation scheme, where repair is generated after each video frame.
   - Explore how this scheme reduces latency compared to the fixed-size block scheme.
3) Simulation with PyErasure:
   - Use PyErasure to implement the simulation with an imaginary video source producing a video frame every 33 ms.
   - Simulate video frames containing 18, 8, 4, and 2 symbols in a loop.
   - Set the PyErasure block size to 32 and analyze the performance of each repair generation scheme.
   - Experiment with different amounts of repair and packet loss.
4) Performance Evaluation:
   - Evaluate the performance of each repair generation scheme in terms of latency accumulation.
   - Compare the effectiveness of the fixed-size block scheme and the on-the-fly repair generation scheme in reducing latency.

6.4 Summary

Explicitly practical use cases of network coding protocols are explained in Chapter 6. Consisting of more algorithms and examples, this enables you to implement different coding techniques in your own setting and tailor them for different applications. The algorithms and implementations in this chapter also deal with realistic networks beyond the point-to-point setting in Chapter 2. The exercises in this chapter lead to a more system-level analysis of the algorithms, following the discussion of complexity and overheads associated with the real-world implementations in this chapter. With the discussion of more advanced protocols and different optimizations and tailoring based on applications, you should now be able to:

(A) Implement network coding in practical settings.
(B) Analyze your implementations based on different parameters and overheads to decide which one suits the application scenario.

(C) Adopt any of the existing algorithms, or build on top of them your own network coding implementations.

Additional Reading Materials

A detailed discussion on the use of acknowledgment in network coding and implementations of TCP/NC can be found in [2, 13, 29, 30]. The sliding window network coding over a single hop is initially discussed in [4] and the recoder for SWNC is discussed in [3]. AC-RLNC over a single path [6] and multi-path [31] provides further insights to designing novel network coding protocols. Michel et al. [14] is an extensive study on the adaptability of network coding with QUIC.

References

1 W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

2 J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher, and J. Barros, "Network coding meets TCP: theory and implementation," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 490–512, 2011.

3 V. A. Vasudevan, T. Soni, and M. Médard, "Practical sliding window recoder: design, analysis, and usecases," in *IEEE International Symposium on Local and Metropolitan Area Networks*, 2023.

4 J. Cloud and M. Médard, "Network coding over SATCOM: lessons learned," in *Wireless and Satellite Systems*. Springer, 2015, pp. 272–285.

5 J. Cloud, D. Leith, and M. Médard, "A coded generalization of selective repeat ARQ," in *IEEE Conference on Computer Communications*, 2015, pp. 2155–2163.

6 A. Cohen, D. Malak, V. B. Bracha, and M. Médard, "Adaptive causal network coding with feedback," *IEEE Transactions on Communications*, vol. 68, no. 7, pp. 4325–4341, 2020.

7 J. Roskind, "Quick UDP internet connections: multiplexed stream transport over UDP," 2012. [Online]. Available: https://www.ietf.org/proceedings/88/slides/slides-88-tsvarea-10.pdf.

8 IETF, "Coding for QUIC," Working Draft, IETF Secretariat, Internet-Draft draft-swett-nwcrg-coding-for-quic-03, July 2019. [Online]. Available: http://www.ietf.org/internet-drafts/draft-swett-nwcrg-coding-for-quic-03.txt.

9 F. Michel, Q. De Coninck, and O. Bonaventure, "QUIC–FEC: Bringing the benefits of Forward Erasure Correction to QUIC," *IFIP Networking*, 2019.

10 Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, "Pluginizing QUIC," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 59–74.

11 P. Garrido, I. Sánchez, S. Ferlin, R. Agüero, and O. Alay, “rQUIC: Integrating FEC with QUIC for robust wireless communications,” in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019.

12 M. Kim, J. Cloud, A. ParandehGheibi, L. Urbina, K. Fouli, D. Leith, and M. Médard, “Network coded TCP (CTCP),” *arXiv preprint arXiv:1212.2291*, 2012.

13 J. K. Sundararajan, D. Shah, M. Médard, M. Mitzenmacher, and J. Barros, “Network coding meets TCP,” in *IEEE International Conference on Computer Communications*, 2009, pp. 280–288.

14 F. Michel, A. Cohen, D. Malak, Q. De Coninck, M. Médard, and O. Bonaventure, “FlEC: Enhancing QUIC with application-tailored reliability mechanisms,” *IEEE/ACM Transactions on Networking*, vol. 31, no. 2, pp. 606–619, 2022.

15 C. Huitema, “Minimal implementation of the QUIC protocol,” https://github.com/private-octopus/picoquic/blob/master/picoquic/quicctx.c, 2021, Commit: 7f49f62ff7f3938eb1a0f49dfc551d7ed189454c.

16 “Quiche,” https://quiche.googlesource.com/quiche/+/refs/heads/main, file quic/tools/quic_client_base.cc, 2021, Commit: 98966fd9b7183bcdb42ce78e58be40bcf6d68493.

17 L. Thomas, E. Dubois, N. Kuhn, and E. Lochin, “Google QUIC performance over a public SATCOM access,” *International Journal of Satellite Communications and Networking*, vol. 37, no. 6, pp. 601–611, 2019.

18 N. Kuhn, G. Fairhurst, J. Border, and S. Emile, “QUIC for SATCOM,” Working Draft, IETF Secretariat, Internet-Draft draft-kuhn-quic-4-sat-03, January 2020. [Online]. Available: http://www.ietf.org/internet-drafts/draft-kuhn-quic-4-sat-03.txt.

19 T. S. Rappaport, S. Sun, R. Mayzus, H. Zhao, Y. Azar, K. Wang, G. N. Wong, J. K. Schulz, M. Samimi, and F. Gutierrez, “Millimeter wave mobile communications for 5G cellular: it will work!” *IEEE Access*, vol. 1, pp. 335–349, 2013.

20 M. Agiwal, A. Roy, and N. Saxena, “Next generation 5G wireless networks: a comprehensive survey,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 1617–1655, 2016.

21 3GPP, “Release description; Release 15,” 3rd Generation Partnership Project (3GPP), Technical Report (TR) 21.915, Oct. 2019, version 15.0.0. [Online]. Available: http://www.3gpp.org/DynaReport/21915.htm.

22 A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, “The QUIC transport protocol: design and internet-scale deployment,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 183–196.

23 E. O. Elliott, “Estimates of error rates for codes on burst-noise channels,” *The Bell System Technical Journal*, vol. 42, no. 5, pp. 1977–1997, 1963.

24 B. Cavusoglu, D. Schonfeld, and R. Ansari, "Real-time adaptive forward error correction for MPEG-2 video communications over RTP networks," in *IEEE International Conference on Multimedia and Expo.*, 2003, pp. 261–264.

25 R. Puri, K. Ramchandran, K.-W. Lee, and V. Bharghavan, "Forward error correction (FEC) codes based multiple description coding for internet video streaming and multicast," *Signal Processing: Image Communication*, vol. 16, no. 8, pp. 745–762, 2001.

26 A. Frömmgen, J. Heuschkel, and B. Koldehofe, "Multipath TCP scheduling for thin streams: active probing and one-way delay-awareness," in *IEEE International Conference on Communications*, 2018, pp. 1–7.

27 C. Huitema, "QUIC timestamps for measuring one-way delays," Internet Engineering Task Force, Internet-Draft draft-huitema-quic-ts-06, Sep. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-huitema-quic-ts-06.

28 C. Huitema, "Minimal implementation of the QUIC protocol," https://github.com/private-octopus/picoquic, 2020.

29 J. K. Sundararajan, D. Shah, and M. Médard, "ARQ for network coding," in *IEEE International Symposium on Information Theory*, 2008, pp. 1651–1655.

30 J. K. Sundararajan, "On the role of feedback in network coding," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

31 A. Cohen, G. Thiran, V. B. Bracha, and M. Médard, "Adaptive causal network coding with feedback for multipath multi-hop communications," *IEEE Transactions on Communications*, vol. 69, no. 2, pp. 766–785, 2020.

7

Network as a Matrix

7.1 Mathematical Model

A communication network is a collection of directed links connecting transmitters, switches, and receivers. Our first goal is to provide a succinct formulation of the network communication problem of interest here. A network may be represented by a directed graph $\mathcal{G} = (V, E)$ with a vertex set V and an edge set $E \subseteq V \times V$. Edges (links) are denoted by round brackets $(v_1, v_2) \in E$ and assumed to be directed. The **head** and **tail** of an edge $e = (v, v')$ are denoted by $v = \text{head}(e)$ and $v' = \text{tail}(e)$.

We define $\Gamma_I(v)$ as the set of edges that end at a vertex $v \in V$, the incoming edges, and $\Gamma_O(v)$ as the set of edges originating at v. Formally, we have $\Gamma_I(v) = \{e \in E : \text{head}(e) = v\}$, $\Gamma_O(v) = \{e \in E : \text{tail}(e) = v\}$. The **in-degree** $\delta_I(v)$ of v is defined as $\delta_I(v) = |\Gamma_I(v)|$ while the **out-degree** $\delta_O(v)$ is defined as $\delta_O(v) = |\Gamma_O(v)|$.

A network is called **cyclic** if it contains directed cycles, i.e. there exists a sequence of edges $(v_0, v_1), (v_1, v_2), \dots, (v_n, v_0)$ in $\mathcal{G}$. A network is called **acyclic** if it does not contain directed cycles. To each link $e \in E$ we associate a non-negative number $C(e)$, called the capacity of e.

Let $\mathcal{X}(v) = \{X(v, 1), X(v, 2), \dots, X(v, \mu(v))\}$ be a collection of $\mu(v)$ discrete data processes (which are often modeled as stochastic processes) that are observable at node v. We want to allow communication between selected nodes in the network, i.e. we want to replicate, by means of the network, a subset of the processes in $\mathcal{X}(v)$ at some different node v'.

We define a **connection** c as a triple $(v, v', \mathcal{X}(v, v')) \in V \times V \times \mathcal{P}_{\mathcal{X}(v)}$, where $\mathcal{P}_{\mathcal{X}(v)}$ denotes the power set of $\mathcal{X}(v)$. Given a connection $c = (v, v', \mathcal{X}(v, v'))$, we call v a **source** and v' a **sink** of c and write $v = \text{source}(c)$ and $v' = \text{sink}(c)$. For notational convenience, we will always assume that $\text{source}(c) \neq \text{sink}(c)$.

A node v can send information through a link $e = (v, u)$ originating at v at a rate of at most $C(e)$ bits per time unit. The process transmitted through link e is denoted by $Y(e)$. In addition to the processes in $\mathcal{X}(v)$, node v can observe processes $Y(e')$ for all

e' in $\Gamma_I(v)$. In general the process $Y(e)$ transmitted through link $e = (v, u) \in \Gamma_O(v)$ will be a function of both $\mathcal{X}(v)$ and $Y(e')$ if e' is in $\Gamma_I(v)$.

If v is the sink of any connection c, the collection of $\nu(v)$ processes $\mathcal{Z}(v) = \{Z(v,1),\ Z(v,2), \ldots, Z(v, \nu(v))\}$ denotes the output at $v = \text{sink}(c)$. A connection $c = (v, v', \mathcal{X}(v, v'))$ is **established successfully** if a (possibly delayed) copy of $\mathcal{X}(v, v')$ is a subset of $\mathcal{Z}(v')$.

Let a network $\mathcal{G}$ be given together with a set C of desired connections. One of the fundamental questions of **network information theory** is under which conditions a given communication scenario is admissible given **data is mutable**. We make some simplifying assumptions:

1) **The capacity of any link in $\mathcal{G}$ is a constant, e.g. one bit per time unit.** This is an assumption that can be satisfied to an arbitrary degree of accuracy. If the capacity exceeds one bit per time unit, we model this as parallel edges with unit capacity. Fractional capacities can be well approximated by choosing the time unit large enough.
2) **Each link in the communication network has the same delay.** We will allow for the case of zero delay in which case we call the network **delay-free**. We will always assume that delay-free networks are acyclic in order to avoid stability problems.
3) **When considered as random processes, the $\mathbf{X(v, l), l \in \{1, 2, \ldots, \mu(v)\}}$ are independent and have a constant and integral rate of, e.g. one bit per unit time.** The unit time is chosen to equal the time unit in the definition of link capacity. This implies that the rate $R(c)$ of any connection $c = (v, v', \mathcal{X}(v, v'))$ is an integer equal to $|\mathcal{X}(v, v')|$. This assumption can be satisfied with arbitrary accuracy by letting the time basis be large enough.
4) **When considered as random processes, the $\mathbf{X(v, l)}$ are independent for different** v. This assumption reflects the nature of a communication network. In particular, information that is injected into the network at different locations is assumed independent.

In addition to the above constraints, we assume that communication in the network is performed by transmission of vectors (symbols) of bits. The length of the vectors is equal in all transmissions, and we assume that all links are synchronized with respect to the symbol timing.

7.2 Routing

We begin by recalling the famous max-flow min-cut theorem for routing, which dates back to 1962. Let $\mathcal{G} = (V, E)$ be a communication network as described above. A **cut** between a node v and v' is a partition of the vertex set of $\mathcal{G}$ into two classes

S and $S^{\mathrm{C}} = V - S$ of vertices such that S contains v and S^{C} contains v'. The value $V(S)$ of the cut is defined as

$$V(S) = \sum_{\{e\,:\,\mathrm{head}(e)\in S,\ \mathrm{tail}(e)\in S^{\mathrm{C}}\}} C(e).$$

Theorem 7.1 ***(Max-flow, min-cut):*** Let a network with a single source and a single sink be given, i.e. the only desired connection is $c = (v, v', \mathcal{X}(v, v'))$. The network problem is solvable if and only if the rate of the connection $R(c)$ is less than or equal to the minimum value of all cuts between v and v'.

Some examples are:

1) Line network.
2) Butterfly.

The Ford–Fulkerson labeling algorithm (1962) [1] gives a more efficient way than brute-force search to find a routing solution for P2P connections provided a network is solvable. No coding occurs, so that processes, or flows, remain uncombined and require therefore no decoding. While it provides an elegant solution for P2P connections, the technique is not powerful enough to handle more involved communication scenarios.

In the remainder of this chapter, we develop some theory and notation necessary for more complex setups.

7.3 Algebraic Network Coding

Recall that any binary vector of length m can be interpreted as an element in $\mathbb{F}_{2^m}$, the finite field with 2^m elements. The processes $X(v,l)$, $Y(e)$, and $Z(v,l)$ can hence be modeled as discrete processes $X(v,l) = \{X_0(v,l), X_1(v,l), \dots\}$, $Y(e) = \{Y_0(e), Y_1(e), \dots\}$, and $Z(v,l) = \{Z_0(v,l),\ Z_1(v,l), \dots\}$, that consist of a sequence of symbols from $\mathbb{F}_{2^m}$.

7.3.1 Formulation

A $\mathcal{G} = (V, E)$ **delay-free** (and hence by assumption acyclic) communication network is said to be a $\mathbb{F}_{2^m}$-**linear** communication network if for all links, the process $Y(e)$ on a link $e = (v, u) \in E$ satisfies

$$Y(e) = \sum_{l=1}^{\mu(v)} \alpha_{e,l} X(v,l) + \sum_{e' : \mathrm{head}(e')=\mathrm{tail}(e)} \beta_{e',e} Y(e'),$$

where the $\alpha_{e,l}$ and $\beta_{e',e}$ are elements of $\mathbb{F}_{2^m}$.

This definition is concerned with the formation of processes that are transmitted on the links of the network. It is possible to consider time-varying coefficients $\alpha_{e,l}$ and $\beta_{e',e}$, and we call the network **time-invariant** or **time varying** depending on this choice. The output $Z(v, l)$ at any node v is formed from the random processes $Y(e)$ for $e \in \Gamma_I(v)$. It will be sufficient for our purposes to restrict ourselves to the case where the $Z(v, l)$ are also linear combinations of the $Y(e)$, i.e.

$$Z(v,j) = \sum_{e' : \text{head}(e')=v} \varepsilon_{e',j} Y(e').$$

where the coefficients $\varepsilon_{e',j}$ are elements of $\mathbb{F}_{2^m}$. Indeed, we will prove that it suffices to consider the formation of the $Z(v,j)$ by linear functions of $Y(e)$ for $e \in \Gamma_I(v)$. We can freely choose m and the field $\mathbb{F}_{2^m}$ containing the constants $\alpha_{e,l}$, $\beta_{e',e}$, and $\varepsilon_{e',j}$.

For a given network $\mathcal{G}$ and a given set of connections $\mathcal{C}$, we formally define a **network coding problem** as a pair $(\mathcal{G}, \mathcal{C})$. The problem is to give algebraic conditions under which a set of desired connections is feasible. This is equivalent to finding elements $\alpha_{e,l}$, $\beta_{e',e}$, and $\varepsilon_{e',j}$ in a suitably chosen field $\mathbb{F}_{2^m}$ such that all desired connections can be accommodated by the network. Such a set of numbers $\alpha_{e,l}$, $\beta_{e',e}$, and $\varepsilon_{e',j}$ will be called a **solution** to the network coding problem $(\mathcal{G}, \mathcal{C})$. If a solution exists the network coding problem will be called **solvable**. The solution is time-invariant (time-varying) if the $\alpha_{e,l}$, $\beta_{e',e}$, and $\varepsilon_{e',j}$ are independent (dependent) of the time.

For example, the Ford–Fulkerson labeling algorithm finds a routing (rather than coding) solution and so is restricted such that all parameters $\alpha_{e,l}$ and $\beta_{e',e}$ in are either zero or one.

In order to analyze the network framework, we will develop some algebraic concepts that enable formal manipulation and better understanding of the operation of the system, following the discussions in [2].

We first consider a P2P setup, as mentioned before, max-flow min-cut makes clear that coding is not needed in this case. We also start by considering a **delay-free** network.

Node v is the only source in the network and let:

- $\underline{X} = (X(v, 1), X(v, 2), \dots , X(v, \mu(v)))$ denote the vector of input processes observed at v;
- v' be the only sink node in the network;
- $\underline{Z} = (Z(v', 1), Z(v', 2), \dots , Z(v', \nu(v')))$ be the vector of output processes.

The most important consequence of considering an $\mathbb{F}_{2^m}$ **linear delay-free** network is that we can give a **transfer matrix** describing the relationship between an input vector $\underline{X}$ and an output vector $\underline{Z}$. Let M be the system transfer matrix of a network with input $\underline{X}$ and output $\underline{Z}$, i.e.

$$\underline{X}M = \underline{Z}, \tag{7.1}$$

where M is a matrix whose coefficients are elements in the field $\mathbb{F}_{2^m}$.

We begin with a simple example of a P2P connection in the communication network given in Fig. 7.1. We have the following set of equations governing the parameters $\alpha_{e,l}$, $\beta_{e',e}$, and $\varepsilon_{e,j}$ and the random processes in the network

$$\begin{aligned}
Y(e_1) &= \alpha_{e_1,1}X(v,1) + \alpha_{e_1,2}X(v,2) + \alpha_{e_1,3}X(v,3)\\
Y(e_2) &= \alpha_{e_2,1}X(v,1) + \alpha_{e_2,2}X(v,2) + \alpha_{e_2,3}X(v,3)\\
Y(e_3) &= \alpha_{e_3,1}X(v,1) + \alpha_{e_3,2}X(v,2) + \alpha_{e_3,3}X(v,3)\\
Y(e_4) &= \beta_{e_1,e_4}Y(e_1) + \beta_{e_2,e_4}Y(e_2)\\
Y(e_5) &= \beta_{e_1,e_5}Y(e_1) + \beta_{e_2,e_5}Y(e_2)\\
Y(e_6) &= \beta_{e_3,e_6}Y(e_3) + \beta_{e_4,e_6}Y(e_4)\\
Y(e_7) &= \beta_{e_3,e_7}Y(e_3) + \beta_{e_4,e_7}Y(e_4)\\
Z(v',1) &= \varepsilon_{e_5,1}Y(e_5) + \varepsilon_{e_6,1}Y(e_6) + \varepsilon_{e_7,1}Y(e_7)\\
Z(v',2) &= \varepsilon_{e_5,2}Y(e_5) + \varepsilon_{e_6,2}Y(e_6) + \varepsilon_{e_7,2}Y(e_7)\\
Z(v',3) &= \varepsilon_{e_5,3}Y(e_5) + \varepsilon_{e_6,3}Y(e_6) + \varepsilon_{e_7,3}Y(e_7).
\end{aligned}$$

It is straightforward to compute the transfer matrix describing the relationship between $\underline{X}$ and $\underline{Z}$. In particular, let matrices A and B be defined as

$$A = \begin{pmatrix} \alpha_{e_1,1} & \alpha_{e_2,1} & \alpha_{e_3,1}\\ \alpha_{e_1,2} & \alpha_{e_2,2} & \alpha_{e_3,2}\\ \alpha_{e_1,3} & \alpha_{e_2,3} & \alpha_{e_3,3} \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} \varepsilon_{e_5,1} & \varepsilon_{e_5,2} & \varepsilon_{e_5,3}\\ \varepsilon_{e_6,1} & \varepsilon_{e_6,2} & \varepsilon_{e_6,3}\\ \varepsilon_{e_7,1} & \varepsilon_{e_7,2} & \varepsilon_{e_7,3} \end{pmatrix}.$$

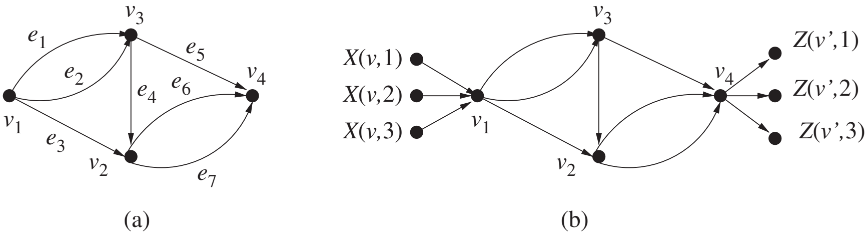


Figure 7.1 (a) A P2P connection in a simple network; (b) the same network with nodes representing the processes to be transmitted in the network. This figure is inspired by joint work with Ralf Kötter.

We can interpret the matrix A as the coding at the source, or precoding, of the data, which can take any form. The matrix B at the receiver is the decoding matrix, so it too can have any form.

The system matrix M is constrained by the graph and has been found to equal

$$M = A \begin{pmatrix} \beta_{e_1,e_5} & \beta_{e_1,e_4}\beta_{e_4,e_6} & \beta_{e_1,e_4}\beta_{e_4,e_7} \\ \beta_{e_2,e_5} & \beta_{e_2,e_4}\beta_{e_4,e_6} & \beta_{e_2,e_4}\beta_{e_4,e_7} \\ 0 & \beta_{e_3,e_6} & \beta_{e_3,e_6} \end{pmatrix} B^T.$$

If we examine the matrix M, we see that the elements correspond to different routes through the small illustrative network depicted in Fig. 7.1.

The determinant of matrix M equals

$$\det(M) = \det(A)(\beta_{e_1,e_5}\beta_{e_2,e_4} - \beta_{e_2,e_5}\beta_{e_1,e_5})(\beta_{e_4,e_6}\beta_{e_3,e_7} - \beta_{e_4,e_7}\beta_{e_3,e_6})\det(B).$$

We can choose parameters in an extension field $\mathbb{F}_{2^m}$ so that the determinant of M is nonzero over $\mathbb{F}_{2^m}$. Hence we can choose A as the identity matrix and B so that the overall matrix M is also an identity matrix, which is what decoding by the matrix B achieves. For example, the solution found by the Ford–Fulkerson algorithm would be equivalent to $\beta_{e_1,e_5} = \beta_{e_2,e_4} = \beta_{e_4,e_6} = \beta_{e3,e_7} = 1$, while all other parameters of type $\beta_{e',e}$ are chosen to equal zero.

Clearly, a P2P communication between v and v' is possible at a rate of three bits per unit time. We note that, if we grow m, there exists an infinite number of solutions to the posed networking problem, namely, all assignments to parameters $\beta_{e',e}$ which render a nonzero determinant of the transfer matrix M.

We see that the crucial property of the network is that the equation $(\beta_{e_1,e_5}\beta_{e_2,e_4} - \beta_{e_2,e_5}\beta_{e_1,e_5})(\beta_{e_4,e_6}\beta_{e_3,e_7} - \beta_{e_4,e_7}\beta_{e_3,e_6})$ admitted a choice of variables so that the polynomial did **not** evaluate to zero. Consider the set of polynomials over an infinite field $\mathcal{F}$ in variables $X_1, X_2, \ldots, X_n$. For any nonzero polynomial in that set there exists an infinite set of n-tuples $(x_1, x_2, \ldots, x_n)$ such that $f(x_1, x_2, \ldots, x_n) \neq 0$, by induction over the number of variables and the fact that $\mathbb{F}_{2^m}$ can be arbitrarily large.

The following theorem makes the connection between the network transfer matrix M (an algebraic quantity), and the min-cut max-flow Theorem (a graph-theoretic tool):

Theorem 7.2 Let a linear network be given. The following statements are equivalent:

1) A point-to-point connection $c = (v, v', \mathcal{X}(v, v'))$ is possible.
2) The min-cut max-flow bound (Theorem 7.1) is satisfied for a rate $R(c)$.
3) The determinant of the $R(c) \times R(c)$ transfer matrix M is nonzero over $[\ldots, \alpha_{e,l}, \ldots, \beta_{e',e}, \ldots, \varepsilon_{e',j}, \ldots]$.

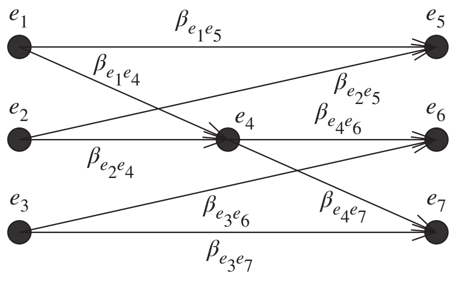


Figure 7.2 The directed labeled line graph $\mathfrak{G}$ corresponding to the network depicted in Fig. 7.1. This figure is inspired by joint work with Ralf Kötter.

Proof: We see (1) and (2) are equivalent by the min-cut max-flow theorem.

We show the equivalence of (1) and (3): The Ford–Fulkerson algorithm implies that a solution to the linear network coding problem exists. Choosing this solution for the parameters of the linear network coding problem yields a solution such that M is the identity matrix, and hence the determinant of M over $[\dots, \alpha_{e,l}, \dots, \beta_{e',e}, \dots, \varepsilon_{e',j}, \dots]$ does not vanish identically.

Conversely, if the determinant of M is nonzero over $[\dots, \alpha_{e,l}, \dots, \beta_{e',e}, \dots, \varepsilon_{e',j}, \dots]$ we can invert matrix M by choosing parameters $\varepsilon_{e',l}$ accordingly. From the observation that polynomials will admit nonzero solutions in a large enough finite field, we know that we can choose the parameters as to make this determinant nonzero. Hence (3) implies (1) and the equivalences are shown.

We conclude that studying the feasibility of connections in a linear network scenario is equivalent to studying the properties of solutions to polynomial equations over a field.

7.3.2 The Network as a Matrix

We know that the action of the linear network is a linear transformation of the input data, leading to the output, Eq. (7.1). A core question is: is there an easy way to determine the matrix M in terms of properties of the input, output, and graph?

We present a representation of network action using transfer matrices. In a linear communication network, any node v_i transmits, on an outgoing edge, a linear combination of the symbols observed on the incoming edges. This relationship between edges in a linear communication network is the incidence structure in which we are most interested.

We say that any edge $e = (u, v)$ **feeds into** edge $e' = (v, u')$ if head(e) is equal to tail(e'). We define the "directed labeled line graph" of $\mathcal{G} = (V, E)$ as $\mathfrak{G}(\mathcal{V}, \mathcal{E})$ with vertex set $\mathcal{V} = E$ and edge set $\mathcal{E} = \{(e, e') \in E^2 \, : \, \text{head}(e) = \text{tail}(e')\}$. Any edge $\mathfrak{e} = (e, e') \in \mathcal{E}$ is labeled with the corresponding label $\beta_{e',e}$. Figure 7.2 shows the directed labeled line graph of the network in Fig. 7.1.

We define the adjacency matrix F of the graph $\mathfrak{G}$ with elements $F_{i,j}$ given as

$$F_{i,j} = \begin{cases} \beta_{e_i,e_j} & \text{head}(e_i) = \text{tail}(e_j) \\ 0 & \text{otherwise.} \end{cases}$$

In order to consider the case that a network contains multiple sources and sinks, we consider

$$\begin{aligned} \underline{X} &= (\underline{X}_1, \underline{X}_2, \dots, \underline{X}_\mu) \\ &= (X(v_1,1), X(v_1,2), \dots, X(v_1,\mu(v_1)), X(v_2,1), \dots, X(v_{|V|}, \mu(v_{|V|}))) \end{aligned}$$

as the vector of input processes on all vertices in V. If a vertex v in a network is not a source node, we set $\mu(v)$ to zero and so $\underline{X} = (x_1, x_2, \dots, x_\mu)$ is a vector of length $\mu = \sum_i \mu(v_i)$. Let the entries of a $\mu \times |E|$ matrix A be defined as

$$A_{i,j} = \begin{cases} \alpha_{e_j,l} & x_i = X(\text{tail}(e_j), l) \;\text{ for some }\; l \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, let

$$\begin{aligned} \underline{Z} &= (\underline{Z}_1, \underline{Z}_2, \dots, \underline{Z}_\mu) \\ &= (Z(v_1,1), Z(v_1,2), \dots, Z(v_1,\nu(v_1)), Z(v_2,1), \dots, Z(v_{|V|}, \nu(v_{|V|}))), \end{aligned}$$

be the vector of output processes. If v_j is not a sink node of any connection we let $\nu(v_j)$ be equal to zero and so $\underline{Z}$ is a vector of length $\nu = \sum_i \nu(v_i)$. Let the entries of a $\nu \times |E|$ matrix B be defined as

$$B_{i,j} = \begin{cases} \varepsilon_{e_j,l} & z_i = Z(\text{head}(e_j), l) \;\text{ for some }\; l \\ 0 & \text{otherwise.} \end{cases}$$

By way of example, consider the network with two source nodes and two sink nodes shown in Fig. 7.3 along with its corresponding labeled line graph.

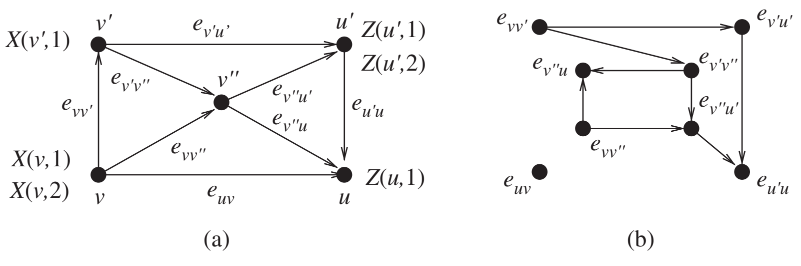


Figure 7.3 (a) A network with two source and two sink nodes. (b) The corresponding labeled line graph; labels in (b) are omitted for clarity. The edge e_{uv} does not feed into any other edge and no edge feeds into e_{uv}, which renders an isolated vertex in the labeled line graph. This figure is inspired by joint work with Ralf Kötter.

We assume that the network is supposed to accommodate two connections $c_1 = (v, u', \{X(v,1), X(v,2)\})$ and $c_2 = (v', u, \{X(v',1)\})$. We fix an ordering of edges as $e_{v,v'}, e_{v,v''}, e_{vu}, e_{v',v''}, e_{v',u'}, e_{v'',u}, e_{v'',u'}, e_{u',u}$. From the ordering of vertices (v, v', v'', u, u') the corresponding orderings of the edges in $\mathcal{G}$ are determined, i.e. $e_{v,v'} \prec_O e_{v,v''} \prec_O e_{vu} \prec_O e_{v',v''} \prec_O e_{v',u'} \prec_O e_{v'',u} \prec_O e_{v'',u'} \prec_O e_{u',u}$ and $e_{v,v''} \prec_I e_{v',v''} \prec_I e_{v,u} \prec_I e_{v'',u} \prec_I e_{u',u} \prec_I e_{v',u'} \prec_I e_{v'',u} \prec_I e_{u'',u}$. For this ordering, the adjacency matrix F of the labeled line graph $\mathfrak{G}$ is found to equal

$$F = \begin{pmatrix} 0 & \beta_{e_{v,v'},e_{v',v''}} & 0 & 0 & \beta_{e_{v,v'},e_{v',u'}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \beta_{e_{v,v''},e_{v'',u}} & \beta_{e_{v,v''},e_{v'',u'}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \beta_{e_{v',v''},e_{v'',u}} & \beta_{e_{v',v''},e_{v'',u'}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \beta_{e_{v',u'},e_{u',u}} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \beta_{e_{v'',u'},e_{u',u}} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Also, matrices A and B are found to equal:

$$A = \begin{pmatrix} \alpha_{e_{v,v'},1} & \alpha_{e_{v,v''},1} & \alpha_{e_{v,u},1} & 0 & 0 & 0 & 0 \\ \alpha_{e_{v,v'},2} & \alpha_{e_{v,v''},2} & \alpha_{e_{v,u},2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha_{v',v'',1} & \alpha_{v',u',1} & 0 & 0 \end{pmatrix},$$

and

$$B = \begin{pmatrix} 0 & 0 & \varepsilon_{e_{v,u},1} & 0 & 0 & \varepsilon_{e_{v'',u},1} & 0 & \varepsilon_{e_{u',u},1} \\ 0 & 0 & 0 & 0 & \varepsilon_{e_{v',u'},1} & 0 & \varepsilon_{e_{v'',u'},1} & 0 \\ 0 & 0 & 0 & 0 & \varepsilon_{e_{v',u'},2} & 0 & \varepsilon_{e_{v'',u'},2} & 0 \end{pmatrix}.$$

From the matrices F, A, and B, we can find the transfer matrix of the overall network. The main concept here is to verify that the path between nodes in the network is accounted for in the series $I + F + F^2 + F^3 + \cdots$, each multiplication by the matrix F corresponding to a single step through the network. Not going anywhere corresponds to multiplying by the identity matrix I, or F^0. Any two-step progression through the networks is F^2 and so on.

Eventually, we exceed the number of hops we can take on the network. We say that the matrix F is **nilpotent**, which means that eventually there will be a finite N_0 such that F_0^N is the all zero matrix which we denote by $\underline{0}$. Note that automatically $F^N = \underline{0}$ for all $N \geq N_0$.

Recall that if $x \in (-1, 1)$, then its geometric series satisfies

$$1 + x + x^2 + x^3 + \cdots = (1 - x)^{-1}.$$

If F is a matrix in the reals then something similar holds so long as the absolute values of all its eigenvalues are less than 1. At this stage, it will be no surprise that a similar result also holds in a finite field:

$$I + F + F^2 + F^3 + \cdots = (1 - F)^{-1}.$$

As an example, consider $\mathbb{F}_7$, the prime field with seven elements, and the matrix

$$F = \begin{pmatrix} 0 & 6 & 2 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix} \text{ we have that } F^2 = F \cdot F = \begin{pmatrix} 0 & 0 & 4 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

and F^L for $L > 2$ is the all zero matrix. As a result, we have that

$$M = I + F + F^2 + F^3 + \cdots = I + F = \begin{pmatrix} 1 & 6 & 6 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{pmatrix}.$$

Now

$$I - F = \begin{pmatrix} 1 & 1 & 5 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{pmatrix},$$

and using Gauss–Jordan in a finite field we identify its inverse as

$$(I - F)^{-1} = \begin{pmatrix} 1 & 6 & 6 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{pmatrix},$$

so we see that, indeed, $M = (I - F)^{-1}$.

Exercise 7.1

Verify for two different values of m that F given by Fig. 7.3 is nilpotent over $\mathbb{F}_{2^m}$. Verify that for the depth d of the network (maximum length route that can be taken staring at the source) $F^d = \underline{0}$ Create your own acyclic network and incidence graph, then repeat.

Theorem 7.3 Let a network be given with matrices A, B, and F. The transfer matrix of the network is

$$M = A(I - F)^{-1}B^T,$$

where I is the $|E| \times |E|$ identity matrix.

Proof: Matrices A and B do not substantially contribute to the overall transfer matrix as they only perform a linear mixing of the input and output processes. In order to find the "impulse response" of the link between an input random process $X(v, i)$ and an output $Z(v', j)$ we have to add all gains along all paths that the random process $X(v, i)$ can take in order to contribute to $Z(v', j)$. Since F is nilpotent, we can write $(I - F)^{-1} = (I + F + F^2 + F^3 + \cdots)$. To verify this fact, suppose that the smallest n for which $F^N = \underline{0}$ is N_0. We then have that

$$(I - F)^{-1} = (I + F + F^2 + F^3 + \cdots) = I + F + F^2 + F^3 + \cdots + F^{N_0 - 1}.$$

The theorem follows.

7.3.3 Multicast

In its simplest form, the multicast problem consists of the distribution of the information generated at a single source node v to a set of sink nodes $u_1, u_2, \dots, u_M$ such that **all** sink nodes get **all** source bits. In other words, the set of desired connections is given by $\{(v, u_1, \mathcal{X}(v)), (v, u_2, \mathcal{X}(v)), \dots, (v, u_M, \mathcal{X}(v))\}$. Clearly, each connection $(v, u_i, \mathcal{X}(v))$ must satisfy the cut-set bound between v and u_i.

Theorem 7.4 Let a delay-free network $\mathcal{G}$ and a set of desired connections $\mathbb{C} = \{(v, u_1, \mathcal{X}(v)), (v, u_2, \mathcal{X}(v)), \dots, (v, u_N, \mathcal{X}(v))\}$ be given. The network problem $(\mathcal{G}, \mathcal{C})$ is solvable if and only if the min-cut max-flow bound is satisfied for all connections in $\mathcal{C}$.

Proof: We have a single source in the network and, hence, the system matrix M is a matrix with dimension $|\mathcal{X}(v)| \times K|\mathcal{X}(v)|$. Moreover, by assumption and Theorem 7.2, each $|\mathcal{X}(v)| \times |\mathcal{X}(v)|$ submatrix corresponding to one connection has nonzero determinant over $[\underline{\xi}]$. We consider the product of the N determinants of the $|\mathcal{X}(v)| \times |\mathcal{X}(v)|$ submatrices. This product is a nonzero polynomial in $\underline{\xi}$. We can find an assignment for $\underline{\xi}$ such that all K determinants are nonzero and hence that all K submatrices are invertible. By choosing matrix B accordingly, we can guarantee that M is the K-fold repetition of the $|\mathcal{X}(v)| \times |\mathcal{X}(v)|$ unit matrix, proving the theorem.

The most important ingredient of the above theorem is the fact that **all sink nodes get the same information**. This implies that all **interference** between the connections can be resolved constructively. In other words, provided that the sink nodes know the part of the system matrix that describes their connection, the very notion of interference is moot. Another interesting aspect of this setup is that **the sink nodes do not have to be aware of the topology of the network**. Knowledge about the overall effects of all coding occurring in the network is sufficient to resolve their connection.

The construction of special codes for the multicast network coding problem is rather easy. We are given a polynomial in $\underline{\xi}$ (the product of the N determinants) and we must find a **nonroot** of that polynomial. From what we have seen of finite fields in Chapter 2, we know that it will suffice for us to select coding coefficients randomly from a large enough field, say by growing m in our extension field.

We may visualize the problem of solving a single source multicast over an arbitrary network as in Fig. 7.4. The matrix $(I - F)^{-1}$ is abstracted to being the central matrix. The A matrix, precoding, has zero entries except in the square part at the start, which would generate any precoding. The decoding matrix B is block diagonal. This shape reflects the fact that the different receivers cannot cooperate in their decoding. Each block in the block diagonal B matrix corresponds to the decoding matrix at the corresponding sink, shown in a color-coded fashion. The total response matrix consists of the concatenation of three square submatrices, which we show as color-coded (you can also differentiate between them by looking at the sink nodes in the figure and the different shades in the matrix if you are using a grayscale version of the book). Let us denote each of them by M_i. Each square submatrix M_i is the overall response, which includes the precoding, the network represented by F, and the decoding, for the corresponding sink i. We see that the ith square submatrix will, because of the matrix multiplication, see the decoding effect of only the ith block in the block diagonal B matrix. Our color coding reflects that phenomenon.

When we were considering a single P2P connection in Section 7.3.2, we found, in effect, a solution over $\mathbb{F}_2$. The graph-theoretic, apparently noncoding solution, can be interpreted as a solution over $\mathbb{F}_2$ where a flow from an edge is either forwarded or not along an incident edge, with the extra constraint that only one flow can be

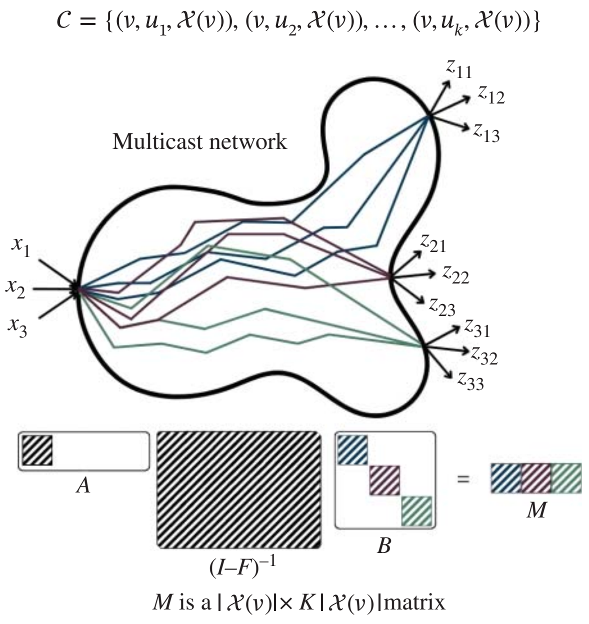

$C = \{(v, u_1, \mathcal{X}(v)), (v, u_2, \mathcal{X}(v)), \ldots, (v, u_k, \mathcal{X}(v))\}$
Multicast network
x_1
x_2
x_3
z_{11}
z_{12}
z_{13}
z_{21}
z_{22}
z_{23}
z_{31}
z_{32}
z_{33}
A
$(I-F)^{-1}$
B
=
M
M is a $|\mathcal{X}(v)| \times K|\mathcal{X}(v)|$ matrix

Figure 7.4 A representation of the multicast problem for a general network with three sinks. Each submatrix color corresponds to a sink node.

forwarded onto an edge. The decision of whether to forward or not is a Boolean one – either the flow from one edge is sent on an adjacent edge, or it is not. Recall from Section 2.4 on Prime Fields that we can express all Boolean expressions as $\mathbb{F}_2$ expressions, with AND being multiplication and XOR being addition.

With multiple sinks, it is still the case that each sink, i, needs to observe an invertible matrix, M_i. The issue is that Ford–Fulkerson only applies to the case where we have a single source and a single sink. Revisiting Theorem 7.2, it states that the min-cut max-flow condition from the source to any one of the sinks, say sink i, is a necessary and sufficient condition to establish the connection. It also states that it means that we are able to select coefficients in the network that lead to an invertible matrix M_i. Equivalently, for sink i, we must be able to find coefficients that are nonroots of the determinant of the matrix M_i the polynomial in the coefficients.

We need to make sure that the coefficients in the network coding problem are **simultaneously nonroots of multiple matrices**. Another way of stating this fact is that we now need the coefficients to be nonroots of the polynomial given by the product $\prod_{i=1}^{K} \det \left(M_i\right)$. Fortunately, we know that in order to find such coefficients, it suffices to grow the field, for example by taking an extension field with large enough m and selecting coefficients randomly, in order to obtain coefficients that, with high probability, will satisfy the required nonzero product of determinants.

Exercise 7.2

Consider a directed acyclic network with a single source node with three independent uniformly distributed Bernoulli inputs. Find conditions and an algorithm to provide one sink node some linear function of the three inputs (with uniformly selected coefficients), another sink node a linear function of the first two inputs (with all uniformly selected coefficients), and the other sink node with all of the inputs.

7.3.4 Multiple Sources and Multiple Sinks**

Returning to our own good use of network coding, let us expand our multicast example with the case of multiple sources and multiple sinks in a network coding problem where **all sources want to communicate all their information to all sinks**. In other words, the set of desired connections between N sources and K sinks is given as $C = \{(v_i, u_j, \mathcal{X}(v_i)) \,:\, i = 0, 1, \dots N, j = 1, 2, \dots K\}$. One characterization of this setup is again that it is interference-free since all sinks are supposed to receive *all* the information. We say that the min-cut max-flow bound is satisfied between a set of sink nodes $\{u_1, u_2, \dots, u_K\}$ and a set of source nodes $\{v_1, v_2, \dots, v_N\}$ if any cut separating a set U of nodes, $U \subset \{u_1, u_2, \dots, u_K\}$ from a set W of source nodes, $W \subset \{v_1, v_2, \dots, v_N\}$ has value at least $\sum_{v \in W} |H(\mathcal{X}(v))|$. We note that this condition can be efficiently checked by computing the determinant of the corresponding submatrix of the transfer matrix. Indeed, the submatrix corresponding to a set of source nodes and sink nodes is nonsingular if and only if the corresponding min-cut max-flow bound is satisfied between the sets of vertices.

Theorem 7.5 Let a linear, acyclic, delay-free network $\mathcal{G}$ be given with a set of desired connections $\mathcal{C} = \{(v_i, u_j, \mathcal{X}(v_i)) : i = 0, 1, \dots N, j = 1, 2, \dots K\}$. The network problem $(\mathcal{G}, \mathcal{C})$ is solvable if and only if the min-cut max-flow bound is satisfied for any cut between all source nodes $\{v_i : i = 0, 1, \dots N\}$ and any sink node u_j.

Proof: We consider the transfer matrices between the N source nodes and any of the K sink nodes individually. Each matrix, considered as a matrix over $\mathbb{F}_2[\underline{\xi}]$, is nonsingular by assumption. Hence we can find an assignment of numbers to the variables $\underline{\xi}$ such that the matrix evaluated at these points is nonsingular over $\overline{\mathcal{F}}$. This holds for each relevant $\sum_{i=1}^{N} \mu(v_i)$ by $\sum_{i=1}^{N} \mu(v_i)$ matrix. The sink nodes can obtain the desired information by applying matrix B to the corresponding matrices.

We note that Theorem 7.5 is considerably more general than Theorem 7.4 which it contains as a special case for $N = 1$. Nevertheless, the situations are relatively similar in spirit and Theorem 7.5 can be reduced to Theorem 7.4 by introducing a super node having access to the entire information and which feeds the corresponding information to the nodes v_i.

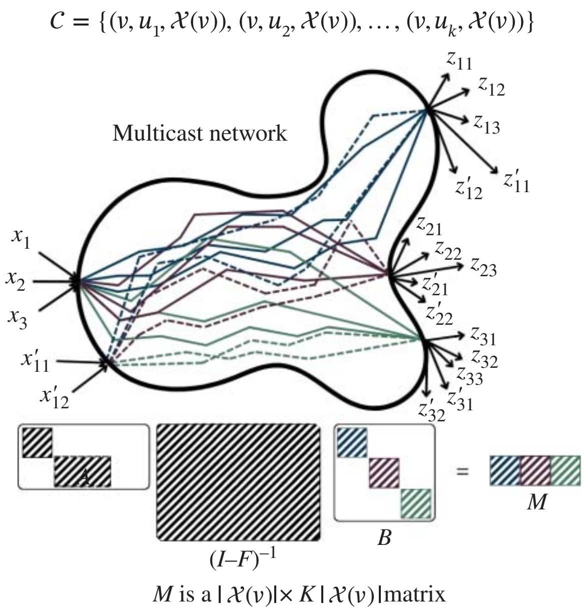


Figure 7.5 A representation of the multi-source multicast problem for a general network with three sinks. Each submatrix color corresponds to a sink node.

It is useful to visualize this multi-source multicast in Fig. 7.5 with color coding (again, follow the sink nodes in the figure and the different shades in the matrix if you are using a grayscale version of the book. The additional source has dashed lines to the sink nodes.) akin to Fig. 7.4. As for the single source multicast, the matrix $(I - F)^{-1}$ is the central matrix. The main difference lies in the A matrix, which now has zero entries except in the blocks corresponding to the source nodes, which would generate any precoding. The sources do not cooperate in their encoding; hence, the blocks corresponding to their encodings do not overlap in the matrix A. The decoding matrix B is block diagonal, reflecting, again, the fact that the different receivers do not cooperate in decoding. As before, each block in the block diagonal B matrix corresponds to the decoding matrix at the corresponding sink, shown in a color-coded fashion. The total response matrix consists of the concatenation of three square submatrices, the color-coded M_i. Each square submatrix M_i is the overall response, which includes the precoding, the network represented by F, and the decoding, for the corresponding sink i. We see that the ith square submatrix will, because of the matrix multiplication, see the decoding effect of only the ith block in the block diagonal B matrix.

A surprising fact in solving a given set of connections in the setup of Theorem 7.5 is that there is no encoding necessary at the source nodes. This is also clear from the observation that this case is "interference-free," which is crucial (the general linear network coding problem discussed below will expand on this notion of interference). However, allowing for proper encoding at the source node is crucial for the general networking problem.

Exercise 7.3

Consider a simple acyclic network with two correlated sources that are created from linear combinations of independent uniformly distributed Bernoulli sources. Describe conditions, along with a coding and decoding algorithm, that allow multicasting to a set of sinks in this setting.

Exercise 7.4

Construct an example, for nonmulticast settings, where nonscalar linear network codes may perform better than scalar ones.

The General Network Coding Problem

The situation is much changed if we consider the general network coding problem, i.e. we are given a network $\mathcal{G}$ and an arbitrary set of connections $\mathcal{C}$. To accommodate the desired connections, we have to ensure that (a) the min-cut max-flow bound is satisfied for every single connection and (b) there is no disturbing interference from other connections.

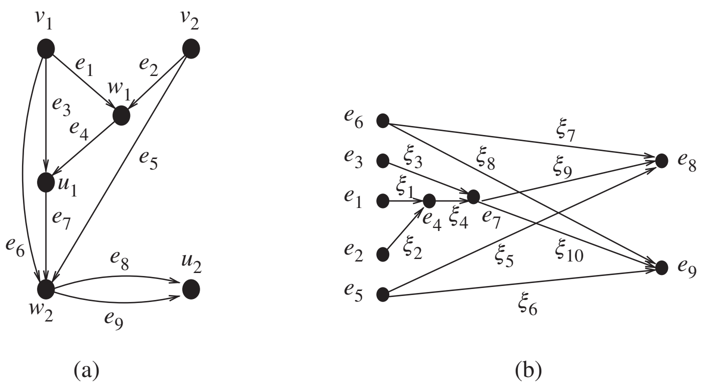


Figure 7.6 (a) A network with two sources and two sink nodes. (b) The corresponding labeled line graph. This figure is inspired by joint work with Ralf Kötter.

The following example outlines the basic requirements for the general case: Let the network $\mathcal{G}$ be given as depicted in Fig. 7.6.a. The corresponding labelled line graph is given in Fig. 7.6.b. We assume that we want to accommodate two connections in the network, i.e. $C = \{(v_1, u_1, \{X(v_1, 1), X(v_1, 2)\}), (v_2, u_2, \{X(v_2, 1), X(v_2, 2)\})\}$. Vectors $\underline{x}$ and $\underline{z}$ are given as $\underline{x} = (X(v_1, 1), X(v_1, 2), X(v_2, 1), X(v_2, 2))$ and $\underline{z} = (Z(u_1, 1), Z(u_1, 2), Z(u_2, 1), Z(u_2, 2))$. It is straightforward to check that the system matrix M such that $\underline{z} = \underline{x}M$ holds, is given as

$$\begin{pmatrix} \xi_{11} & \xi_{12} & \xi_{13} & 0 & 0 \\ \xi_{14} & \xi_{15} & \xi_{16} & 0 & 0 \\ 0 & 0 & 0 & \xi_{17} & \xi_{18} \\ 0 & 0 & 0 & \xi_{19} & \xi_{20} \end{pmatrix} \begin{pmatrix} 0 & \xi_1 & \xi_1\xi_4\xi_9 & \xi_1\xi_4\xi_{10} \\ 1 & 0 & \xi_3\xi_9 & \xi_3\xi_{10} \\ 0 & 0 & \xi_7 & \xi_8 \\ 0 & \xi_2 & \xi_2\xi_4\xi_9 & \xi_2\xi_4\xi_{10} \\ 0 & 0 & \xi_5 & \xi_6 \end{pmatrix} \begin{pmatrix} \xi_{21} & \xi_{22} & 0 & 0 \\ \xi_{23} & \xi_{24} & 0 & 0 \\ 0 & 0 & \xi_{25} & \xi_{26} \\ 0 & 0 & \xi_{27} & \xi_{28} \end{pmatrix}.$$

We can write M as a block matrix

$$M = \begin{bmatrix} M_{1,1} & M_{1,2} \\ M_{2,1} & M_{2,2} \end{bmatrix},$$

where $M_{1,1}$ denotes the transfer matrix between $(X(v_1, 1), X(v_1, 2))$ and $(Z(u_1, 1), Z(u_1, 2))$, $M_{1,2}$ denotes the transfer matrix between $(X(v_1, 1), X(v_1, 2))$ and $(Z(u_2, 1), Z(u_2, 2))$, etc.

It is easy to see that the network problem $(\mathcal{G}, C)$ is solvable if and only if the determinants of $M_{1,1}$ and $M_{2,2}$ are unequal to zero, while the matrices $M_{1,2}$ and $M_{2,1}$ are all zero matrices. Note that the determinant of $M_{1,1}$ and $M_{2,2}$ is nonzero over a large enough $\mathbb{F}_{2^m}$ if and only if the min-cut max-flow bound is satisfied.

Indeed, we have

$$\det(M_{1,1}) = (\xi_{11}\xi_{15} - \xi_{12}\xi_{14})\xi_1(\xi_{21}\xi_{24} - \xi_{22}\xi_{23}),$$

and

$$\det(M_{2,2}) = \xi_2\xi_4(\xi_{17}\xi_{20} - \xi_{18}\xi_{19})(\xi_9\xi_6 - \xi_5\xi_{10})(\xi_{25}\xi_{28} - \xi_{26}\xi_{27}).$$

It is interesting to note that the min-cut max-flow condition is satisfied for each connection individually but also for the cut between both sources and both sinks. This condition is guaranteed by edge e_6. If edge e_6 is removed the determinant of the transfer matrix would vanish identically, which indicates a violation of the min-cut max-flow condition applied to cuts separating v_1 and v_2 from u_1 and u_2.

It now becomes clear why the multicast and related problems were easy and why the general network coding problem is difficult. Indeed, it is so difficult that even for linear network coding its complexity remains unknown at the time of writing of this book. The multicast problem was that of avoiding roots of an equation, or finding *nonsolutions*. Finding nonsolutions is easy – just pick randomly and see whether a nonsolution arises. Imagine that a teacher assigned to students the homework that consists of finding x that does not satisfy a set of equations. Students are unlikely to take the trouble to solve the equations, then select a number that is different from the solution(s) they found. Instead, they would pick a number at random and just check that indeed it was not a solution, as there are few solutions and an infinite, or at least very large set in the finite field case, of possible nonsolutions. That is exactly what RLNC does.

However, in the general network coding case, we need to find solutions to equations, not only nonsolutions. Like most homeworks, guessing and checking is unlikely to yield a good answer. In order to satisfy $M_{2,1} = \underline{0}$ we have to choose $\xi_2 = 0$ which implies that $\det(M_{2,2})$ equals zero.

Hence, we cannot satisfy the requirements that $\det(M_{2,2}) \neq 0$ and $M_{2,1} = \underline{0}$ simultaneously and, hence, the network problem $(\mathcal{G}, \mathcal{C})$ is not solvable. It is worthwhile pointing out that this non solvability of the network coding problem is pertinent for *any* coding strategy and is not a shortcoming of linear network coding.

As before, let $\underline{x}$ denote the vector of input processes and let $\underline{z}$ denote a vector of output processes. We consider the transfer matrix in a block form as $M = \{M_{i,j}\}$ such that $M_{i,j}$ is the submatrix of M that describes the transfer matrix between the input processes at v_i and the output processes at v_j. The following theorem states a succinct condition under which a network problem $(\mathcal{G}, \mathcal{C})$ is solvable.

Theorem 7.6 Let an acyclic, delay-free linear network problem $(\mathcal{G}, \mathcal{C})$ be given and let $M = \{M_{i,j}\}$ be the corresponding transfer matrix relating the set of input nodes to the set of output nodes. The network problem is solvable if and only if there exists an assignment of numbers to $\underline{\xi}$ such that

1) $M_{i,j} = 0$ for all pairs (v_i, v_j) of vertices such that $(v_i, v_j, \mathcal{X}(v_i, v_j)) \notin \mathcal{C}$.
2) If $\mathcal{C}$ contains the connections $(v_{i_1}, v_j, \mathcal{X}(v_{i_1}, v_j))$, $(v_{i_2}, v_j, \mathcal{X}(v_{i_2}, v_j))$, $\dots, (v_{i_\ell}, v_j, \mathcal{X}(v_{i_\ell}, v_j))$ the submatrix $\left[M^T_{i_1,j}, M^T_{i_2,j}, \dots, M^T_{i_\ell,j}\right]$ is a nonsingular $v(v_j) \times v(v_j)$ matrix.

Proof: Assume the conditions of the theorem are met and assume the network operates with the corresponding assignment of numbers to $\underline{\xi}$. Condition (1) ensures that there is no disturbing interference at the sink nodes. Also, any sink node v_j can invert the transfer matrix $\left[M^T_{i_1,j}, M^T_{i_2,j}, \dots, M^T_{i_\ell,j}\right]$ and hence recover the sent information.

Conversely, assume that either of the conditions is not satisfied. If condition (1) is not satisfied, then the collection of random processes observed on the incoming edges of v_j is a superposition of desired information and interference. Moreover, the sink node v_j has no possibility of distinguishing interference from desired information, and, hence, the desired processes cannot be reliably reproduced at v_j. Condition (2) is equivalent to a min-cut max-flow condition, which clearly has to be satisfied if the network problem is solvable.

Theorem 7.6 gives a succinct condition for the satisfiability of a network problem. However, checking the two conditions is a tedious task as we have to find a solution, i.e. an assignment to number $\underline{\xi}$ that exhibits the desired properties. The theory of Gröbner bases provides a structured approach to this problem, but it is highly technical, and we leave it to interested readers to explore further.

7.4 Erasures

The view of the network as a matrix can at this point be merged and reconciled with the approach we have seen in earlier chapters where we have considered network coding as a matrix. The core of network coding is to consider the effect of the network, be it losses or coding at intermediate nodes, as expressible as a matrix seen by each receiver. Recovering the information, say packets, at each receiver is done by inverting that matrix.

Previously the transfer matrix of the network was

$$M = A(I - F)^{-1}B^T,$$

where I is the $|E| \times |E|$ identity matrix.

If we limit to only n uses of the network, instead of $(I - F))^{-1}$ we should consider $(I + F + F^2 + F^3 + \cdots + F^{n-1})$.

Let us consider a network that has erasures, or losses. We can express erasures on a link in a network through the absence of that link during the time when the loss occurs. Specifically, if the original transfer matrix of the full network is F, let us consider the transfer matrix $F_{\mathcal{D}}$ of the network with the set of edges $\mathcal{D} \subseteq \mathcal{E}$ removed. If we consider n consecutive uses of the network, then at each i from 1 to n, we have a $F_{\mathcal{D}_i}$ that represents the state of the network, with any missing edges, at the ith use of the network.

The response of the network now becomes

$$M = A\left(I + \sum_{i=1}^{n}\prod_{j=1}^{i} F_{\mathcal{D}_j}\right)B^T.$$

The same theorems we mentioned before hold. The feasibility of a system depends on the invertibility of M. If we have that the erasures themselves have a distribution, then M will be a random variable.

If the system is a multicast system, then the random selection of coefficients will suffice with high probability for a large enough field size for M. Again, from what we have seen of finite fields in Chapter 2, we know that it will suffice for us to select coefficients randomly from a large enough field, say by growing m in our extension field.

Suppose that we have independent losses as each of the n uses of the network, then we can consider the expected throughput to be the expected degree of M, which is the expected degree of $\left(I + \sum_{i=1}^{n}\prod_{j=1}^{i} F_{\mathcal{D}_j}\right)$.

7.5 An Aside on Misapplication

In general, a binary field may not suffice in terms of the number of choices to pick the coefficients. Routing, which is a subcase of binary, is even more restrictive and, while it is enough in the case of a single source and sink, it will not suffice for multicast in general. Note that sometimes people worry that coding may reduce performance. This is not possible. **Coding is a relaxation, and a relaxation cannot induce a loss in performance**. In certain cases, like Ford–Fulkerson, the relaxation is not needed. It can never be detrimental. That does not mean that coding should not be applied intelligently.

As an aside, examples of poor network coding are not uncommon in the literature. Unfortunately, approaches to coding often start from the fact that network coding alleviates congestion and misapplies the fact. For example, several of them attempt to **create** congestion in order to lead to coding opportunities. Imagine, for instance, burdening artificially a node D in order to allow combinations of many packets. This is akin to causing injury to illustrate the benefits of a new remedy. Network coding can help alleviate congestion, but that is no reason for causing it in the first place.

Another example is transmitting more packets from a node, sometimes trying to mimic wireless channels through many extraneous transmissions intended to provide overhearing. Again, network coding makes use of overhearing on a broadcast channel, but that does not mean that we should seek to create broadcast channels when they are not there naturally.

A third example of poor understanding leading to unfortunate coding choice is to negate the gains of (NC) in incast distributed storage by first selecting the preferred nodes from which to download and then coding over those nodes only. In fact, the gain that network coding provides, as discussed earlier, is that we avoid the coupon collector problem by ensuring that the requisite degrees of freedom are, with high probability, available at any minimum number of nodes. Obviating the need to seek specific pieces of content is the core of the benefit that NC brings to such systems. Applying coding after selecting the peers or storage nodes fails to make use of that benefit.

In all cases, the reader should be aware that coding cannot be highly beneficial and can never be detrimental, although bad engineering can make it such.

7.6 Summary

Chapter 7 focused on the mathematical modeling of network coding, providing additional insights that can guide design. Following the concepts from an algebraic model of network coding, we consider how a network can be modeled as a matrix and then use it to analyze the benefits of network coding. This assists in extending the discussion to more complex networks with multiple sources and destinations. It also discusses additional core concepts such as the min-cut max-flow theorem and explains network coding from a matrix-oriented approach. Further, it aims to discuss different application settings such as multicast and a general network coding problem. Focusing more on the mathematical background and concepts, this chapter enables you to:

(A) Understand the mathematical basis of network coding in detail.
(B) Extend your analysis to more generic and complex problems.
(C) Perform theoretical analysis on the gains of network coding and provide explanations and proofs.

Additional Reading Materials

Most of the discussion in this chapter is inspired by the initial works on algebraic network coding with Ralf Kötter and others [2, 3].

References

1 L. Ford and D. Fulkerson, *Flows in Networks*. Princeton, NJ, USA: Princeton University Press, vol. 276, p. 22, 1962.

2 R. Koetter and M. Médard, "An algebraic approach to network coding," *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 782–795, 2003.

3 D. S. Lun, M. Médard, R. Koetter, and M. Effros, "On coding for reliable communication over packet networks," *Physical Communication*, vol. 1, no. 1, pp. 3–20, 2008.

8

Security and Network Coding

There are many aspects to securing communications and, in the same way that network coding (NC) changes communications, it provides new approaches to security. In this chapter, we shall overview two main areas of security where network coding has a considerable impact. The first is in terms of **cryptography**, literally the hiding (crypto) of information (graphy). The second is in **verification**, the ability to determine that information is correct, or that a user is legitimate.

In order to consider security, we need to introduce some notions from probability and information theory with a little more formalism than we have so far.

- Random variable (r.v.): X.
- Sample value of a random variable: x.
- Set of possible sample values x of the r.v. X, its **sample space**, is: $\mathcal{X}$.
- Probability mass function (PMF) of a discrete r.v. X: $p_X(x)$.
- Joint PMF of a pair of discrete r.v.s X and Y: $p_{X,Y}(x,y)$.
- Conditional PMF of Y given X: $p_{Y|X}(y|x) = p_{X,Y}(x,y)/p_X(x)$.

A simple, central feature that will be relied upon is the finite field fact.

Theorem 8.1 Let M be a random variable taking values in a finite field $\mathbb{F}_q$. Regardless of how M is distributed, if K is selected independently of M and uniformly at random in $\mathbb{F}_q$, then $M+K$ is distributed uniformly at random in $\mathbb{F}_q$, i.e.

$$P(M+K=x) = 1/q \quad \text{for all} \quad x \in \mathbb{F}_q.$$

Proof:

$$\begin{aligned} P(M+K=x) &= \sum_{k\in\mathbb{F}_q} P(M=x-k, K=k) \\ &= \sum_{k\in\mathbb{F}_q} P(K=k)P(M=x-k) \end{aligned}$$

$$= \frac{1}{q} \sum_{k \in \mathbb{F}_q} P(M = x - k)$$
$$= \frac{1}{q} \sum_{m \in \mathbb{F}_q} P(M = m) = \frac{1}{q}.$$

We will also use the fact that if K_1 and K_2 are independent random variables taking discrete values in any field, then if

$$\begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix} = A \begin{pmatrix} K_1 \\ K_2 \end{pmatrix}$$

and A is invertible with inverse A^{-1}, then Y_1 and Y_2 are independent and for

$$\begin{pmatrix} k_1 \\ k_2 \end{pmatrix} = A^{-1} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

then

$$\begin{aligned} P\left((Y_1, Y_2) = (y_1, y_2)\right) &= P\left((K_1, K_2) = (A^{-1}(y_1, y_2)^T)^T\right) \\ &= P\left((K_1, K_2) = (k_1, k_2)\right) \\ &= P(K_1 = k_1)P(K_2 = k_2). \end{aligned}$$

8.1 Information Theoretic Security – Quick Primer

Entropy is a measure of the average uncertainty of a random variable. For a discrete random variable it is defined to be

$$H(X) = -\sum_{x \in \mathcal{X}} p_X(x) \log_2 \left(p_X(x)\right).$$

It can be readily established that entropy is always non-negative.

The **joint entropy** of two discrete r.v.s X and Y with joint PMF $p_{X,Y}(x, y)$ is

$$H(X, Y) = -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} p_{X,Y}(x, y) \log_2 \left(p_{X,Y}(x, y)\right).$$

The **conditional entropy** is the expected value of entropies calculated according to conditional distributions

$$\begin{aligned} H(Y|X) &= \sum_{x \in \mathcal{X}} p_X(x) H(Y|X = x) \\ &= \sum_{x \in \mathcal{X}} p_X(x) \left(-\sum_{y \in \mathcal{Y}} p_{Y|X}(y|x) \log_2 \left(p_{Y|X}(y|x)\right) \right) \\ &= -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} p_{X,Y}(x, y) \log_2 \left(p_{Y|X}(y|x)\right), \end{aligned}$$

This is the average of the entropy of Y given X over all possible values of X.

Comparing conditional entropy with joint entropy, we observe that:

$$\begin{aligned} H(X,Y) &= -\sum_{x\in\mathcal{X},y\in\mathcal{Y}} p_{X,Y}(x,y)\log_2(p_{X,Y}(x,y)) \\ &= -\sum_{x\in\mathcal{X},y\in\mathcal{Y}} p_{X,Y}(x,y)\log_2(p_{Y|X}(y|x)p_X(x)) \\ &= -\sum_{x\in\mathcal{X},y\in\mathcal{Y}} p_{X,Y}(x,y)\log_2(p_{Y|X}(y|x)) - \sum_{x\in\mathcal{X},y\in\mathcal{Y}} p_{X,Y}(x,y)\log_2(p_X(x)) \\ &= H(Y|X) + H(X). \end{aligned}$$

Iterating this leads to the **Chain Rule for Entropy**:

$$H(X_1,\dots,X_n) = \sum_{i=1}^{n} H(X_i|X_1\dots X_{i-1}).$$

Exercise 8.1

Is $H(Y|X) = H(X|Y)$?

To deduce essential properties of entropy, we first introduce some properties of concave functions. A function $f : \mathbb{R} \mapsto \mathbb{R}$ is **concave** if for all $0 \le a \le 1$

$$f(ax + (1-a)y) \ge af(x) + (1-a)f(y).$$

If f is twice differentiable, this is equivalent to its second derivative being non-positive.

The function f is **strictly concave** if

$$f(ax + (1-a)y) > af(x) + (1-a)f(y) \quad \text{for all} \quad a \in (0,1), x \ne y.$$

If f is twice differentiable, this is equivalent to its second derivative being negative everywhere.

If f is concave and X is a r.v., then

$$f(E_X(X)) \ge E_X[f(X)].$$

if f is strictly concave and $E_X[f(X)] = f(E_X(X))$, then we deduce that $X = E(X)$.

We establish that entropy is **concave**, in an appropriate sense. Consider the function $f(x) = -x\log_2(x)$. We have that

$$f'(x) = -x\log_2(e)\frac{1}{x} - \log_2(x) = -\log_2(x) - \log_2(e)$$

and

$$f''(x) = -\log_2(e)\frac{1}{x} < 0 \quad \text{for all} \quad x > 0.$$

Recalling the definition of entropy of a random variable, $H(X) = \sum_{x\in\mathcal{X}} f(p_X(x))$, we deduce that the entropy of X is concave in the value of $p_X(x)$ for every x.

Consider two random variables X_1 and X_2 with common sample space $\mathcal{X}$. Then the random variable X defined over the same $\mathcal{X}$ such that

$$p_X(x) = \lambda p_{X_1}(x) + (1-\lambda)p_{X_2}(x) \quad \text{for some} \quad \lambda \in (0,1)$$

satisfies

$$H(X) \geq \lambda H(X_1) + (1-\lambda)H(X_2). \tag{8.1}$$

Consider any random variable X_1^1 on $\mathcal{X}$. For simplicity, consider $\mathcal{X} = \{1, \dots, |\mathcal{X}|\}$ (we just want to use the elements of $\mathcal{X}$ as indices). Now consider X_2^1 a random variable such that $p_{X_2^1}(x) = p_{X_1^1}(\text{shift}(x))$ where shift denotes the cyclic shift on $(1, \dots, |\mathcal{X}|)$. By definition, $H(X_1^1) = H(X_2^1)$.

For what follows, we consider some properties of entropy and, in particular, the **maximum entropy** possible for any r.v. Consider X_1^2 defined over the same sample space $\mathcal{X}$ such that

$$p_{X_1^2}(x) = \lambda p_{X_1^1}(x) + (1-\lambda)p_{X_2^1}(x)$$

then by Eq. (8.1) $H(X_1^2) \geq H(X_1^1)$. With the obvious extension of notation, we can show recursively that

$$H(X_1^n) \geq H(X_1^m) \quad \text{for all} \quad n > m \geq 1.$$

However, note that

$$\lim_{n\to\infty} p_{X_1^n}(x) = \frac{1}{|\mathcal{X}|} \quad \text{for all} \quad x \in \mathcal{X}$$

and hence the uniform distribution maximizes entropy and $H(X) \leq \log_2(|\mathcal{X}|)$.

Relative entropy, also known as **Kullback Leibler divergence**, is a measure of the "distance" (it is not a true distance metric as it is not symmetric) between two PMFs $p_X(x)$ and $p_Y(y)$:

$$D(p_X || p_Y) = \sum_{x\in\mathcal{X}} p_X(x) \log_2\left(\frac{p_X(x)}{p_Y(x)}\right).$$

We are effectively considering the $\log_2$ to be a r.v. of which we take the mean (note that we use the standard mathematical convention of defining $0\log_2(0/p) = 0$ and $p\log_2(p/0) = \infty$).

We can now consider **mutual information**: let X, Y be r.v.s with joint PMF $p_{X,Y}(x,y)$ and marginal PMFs $p_X(x)$ and $p_Y(y)$, then

$$\begin{aligned} I(X;Y) &= \sum_{x\in\mathcal{X}, y\in\mathcal{Y}} p_{X,Y}(x,y) \log_2\left(\frac{p_{X,Y}(x,y)}{p_X(x)p_Y(y)}\right) \\ &= D\left(p_{X,Y}(x,y) || p_X(x)p_Y(y)\right). \end{aligned}$$

Mutual information is comparing the true joint PMF of X and Y to the PMF that would hold if the two r.v.s were independent. Intuitively, mutual information is a measure of how dependent the r.v.s are.

A useful expression for mutual information:

$$\begin{aligned} I(X;Y) &= H(X) + H(Y) - H(X,Y) \\ &= H(Y) - H(Y|X) \\ &= H(X) - H(X|Y) \\ &= I(Y;X). \end{aligned}$$

Note that while Kullback Leibler is not symmetric, mutual information is.

Exercise 8.2

What is $I(X;X)$?

Exercise 8.3

Show that $I(X;Y) \geq 0$ for any X and Y. From exercise 8.2, what is the maximum value of $I(X;Y)$?

We are now ready to consider perfect secrecy, or the **one-time pad**, sometimes named the **Shannon one-time pad**.

The canonical setup for hiding information, also known as cryptography, is as follows. **Alice (A)**, the legitimate sender, sends to **Bob (B)**, the legitimate receiver. The eavesdropper, **Eve (E)**, seeks to find out what Alice says to Bob. In order to hide the information, Alice uses cryptography, allowing Bob to decrypt because he has access to some secret information, often termed the key, to which Eve does not have access.

- Suppose that both Alice and Bob know a secret random variable, termed the key, K that is uniformly distributed in a finite field $\mathbb{F}_q$.
- Eve does not know anything about K other than that it exists.
- Alice and Bob share a message M, which is a random variable independent of K, generally termed the plaintext, in the following way: Alice computes $X = M + K$ in $\mathbb{F}_q$; both Bob and Eve observe X; Bob knows the realization of K and is always able to compute $X - K = M$ by operations in $\mathbb{F}_q$.
- There is no information that Eve obtains from X about M without knowledge of K.

Exercise 8.4

Show that $I(X;M) = 0$ so Eve obtains no information about M from observing X.

8.2 Data Hiding

8.2.1 Hiding Equations

Let us expand on the theme of the one-time pad with a coding example.

Alice first partitions her message into symbols in the way that we have described. As a result, her message becomes a block of symbols in her choice of finite field, say $\mathbb{F}_{11}$. To transmit a single message symbol $M \in \mathbb{F}_{11}$ securely to Bob, Alice first generates two symbols $K_1, K_2 \in \mathbb{F}_{11}$ uniformly at random, and independently from her message and each other, which serve as two one-time pads. Alice then generates three encoded symbols $X_1, X_2, X_3 \in \mathbb{F}_{11}$ using her message M and the two random symbols K_1 and K_2, given by

$$\begin{aligned} X_1 &= M + K_1 + K_2 \\ X_2 &= M + 2K_1 + 4K_2 \\ X_3 &= M + 3K_1 + 9K_2. \end{aligned} \tag{8.2}$$

Each encoded symbol X_i is transmitted to Bob who receives the three encoded symbols and is able to decode the message symbol M by means of a simple linear transform that inverts Eq. (8.2):

$$\begin{pmatrix} M \\ K_1 \\ K_2 \end{pmatrix} = \begin{pmatrix} 3 & 8 & 1 \\ 3 & 4 & 4 \\ 6 & 10 & 6 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix}. \tag{8.3}$$

If Eve can observe at most two encoded symbols from the set $\{X_1, X_2, X_3\}$ then, regardless of which two encoded symbols Eve detects, she cannot determine M.

For instance, if Eve receives X_1 and X_2, then the mutual information between her observations and the message symbol M can be computed from the entropy as:

$$\begin{aligned} I(M; X_1, X_2) &= H(X_1, X_2) - H(X_1, X_2 | M) \\ &= H(X_1, X_2) - H(K_1 + K_2, 2K_1 + 4K_2) \\ &= H(X_1, X_2) - 2\log(|\mathbb{F}_{11}|) \\ &\leq H(X_1) + H(X_2) - 2\log(11) \\ &= 0. \end{aligned} \tag{8.4}$$

This result follows directly from the definition of mutual information and the fact that, conditioned on the messages, the only uncertainty about X_1 and X_2 is in the random variables $K_1 + K_2$ and $2K_1 + 4K_2$, which are independent and uniform. Thus, because there is zero mutual information between Eve's observation and Alice's message, Eve learns nothing about M.

Alice can replace the random symbols with additional messages, M_2 and M_3, **if those messages are uniformly distributed and mutually independent**, and then perform the same encoding as in (8.2), with the random symbols replaced by the additional messages. Alice can thus obtain an optimum secure communication efficiency of 1, regardless of the number of channels. That is, Alice replaces the random symbols, K_1 and K_2, with message symbols M_2 and M_3, and then transmits the three encoded symbols X_1, X_2, X_3 as in Eq. (8.2) with M, K_1, K_2 replaced by M_1, M_2, M_3,

$$\begin{aligned} X_1 &= M_1 + M_2 + M_3 \\ X_2 &= M_1 + 2M_2 + 4M_3 \\ X_3 &= M_1 + 3M_2 + 9M_3. \end{aligned} \tag{8.5}$$

As the $\{M_i\}$ are uniformly and independently, identically distributed over $\mathbb{F}_{11}$, then the $\{X_i\}$ are also uniformly and independently, identically distributed over $\mathbb{F}_{11}$ As before, Bob can decode all three message symbols through Gaussian elimination.

This scheme guarantees zero mutual information with any subset of message symbols, yet may potentially allow Eve to obtain information about linear combinations of the message symbols. In order to implement this approach, Alice must ensure that the message symbols M_1, M_2, M_3 are uniformly distributed. The reason for this, intuitively, is that the message symbols themselves are performing the role of the random symbols K_1 and K_2. We note that there are known techniques described in the literature which can be used to enforce this uniformity condition, so this requirement is not a significant impediment. Thus, although $I(M_1, M_2, M_3; X_1, X_2)$ may not be zero, it is nevertheless possible for Alice to guarantee that the mutual information between any individual message and any two transmitted symbols is zero. That is, for any distinct $i, j, \ell \in \{1, 2, 3\}$, it follows that

$$\begin{aligned} I(M_i; X_1, X_2) &= H(X_1, X_2) - H(X_1, X_2 | M_i) \\ &= H(X_1, X_2) - H(M_j + M_\ell, 2M_j + 4M_\ell) \\ &= H(X_1, X_2) - 2\log(|\mathbb{F}_{11}|) \\ &\le H(X_1) + H(X_2) - 2\log(11) \\ &= 0. \end{aligned}$$

We emphasize that the information that Eve **can** obtain in this situation involves only linear combinations of Alice's messages is largely trivial, and cannot in general be used to decode or decipher any meaning.

Suppose for example that we have a distributed storage system where three different storage systems each store one different equation. Then if Eve has access to only one or two of the storage systems, she is unable to reconstruct the message M. If Bob is able to have access to all three, then he can reconstruct the message. This allows **trusted storage over untrusted systems**.

In some cases, it may not be able to prevent physical access to Eve to some of the equations. In that case, we may decide to use conventional encryption on one of the equations. In conventional encryption, an encryption function can be inverted (decrypted) by anyone holding a key. Unlike the key in the one-time pad, that key is generally reused for some period of time rather than being entirely different for each message. It is also much smaller in terms of bits than the message that is being hidden, unlike the one-time pad where the number of bits in the key and the message are the same. The key is exchanged between Alice and Bob, using one of the plethora of techniques for key exchange.

Thus, consider that in our example Eve sends

$$\begin{aligned} X_1 &= M_1 + M_2 + M_3, \\ X_2 &= M_1 + 2M_2 + 4M_3, \\ \text{Encrypt}\left(X_3\right) &= \text{Encrypt}\left(M_1 + 3M_2 + 9M_3\right). \end{aligned}$$

Only Bob can decrypt the third equation, that is to say invert the encryption of X_3, because he has access to the relevant key, whereas Eve does not. Thus, to obtain enough information to solve the equations that will provide messages M_1, M_2, M_3, it is necessary for Eve to be able to break the encryption used in the third equation. That encryption can be as sophisticated and expensive as we wish. Classical encryption schemes, such as the commonly used Advanced Encryption Standard (AES), have some delay in encrypting and decrypting. AES performs bit-wise scrambling over 128 bits. The encryption delay and the decryption delay of AES are essentially the same and are often of the order of other delays in the network system, so not necessarily dominant. In cases where the encryption/decryption delay becomes significant relative to other processes, for example when there are very high rates, then one can use parallelization.

Recently, as computing has become increasingly more powerful, traditional encryption methods that rely on the difficulty for Eve of solving certain problems are considered to be less secure. One of the key drivers of that re-examination is the advent of quantum computing. Even if one may not be worried about Eve's access to a quantum computer in the near term, data needs to be often stored securely in a way that is meant to last for some number of years. Post-quantum secure schemes, such as McEliece, work differently. They are not based only on computational complexity arguments but rather also on code-centric, information-theoretic arguments. For those schemes, typically the key grows with the data. While the key is not the same number of bits as the data as in the one-time pad, it does grow, as a roughly constant fraction of the data.

Such post-quantum schemes are not currently widely known because they are known to be onerous in two main ways. The first is that they require the transmission of a faction of bits that is not negligible, of the order of 30%. The second is that the complexity of the encryption is not excessive, but the complexity and attendant delay of decryption is extremely high, meaning that it severely limits achievable rates because of the delay associated with decrypting post-quantum cryptosystems.

If Alice encrypts one or more equations with a post-quantum secure cryptosystem, then Bob can first decrypt those equations, then perform matrix inversion to decode the messages. Because, as we have seen, the decoding has low complexity, it means that we can achieve **low complexity decryption** while being as secure as the encryption scheme used to encrypt one or more equations. Thus, we can reduce the overall complexity of decryption by requiring post-quantum encryption/decryption of only a few equations, but making use of coding to use that encryption to protect all of the encoded data. This type of scheme is called a **Hybrid Universal Network Coding Cryptosystem (HUNCC)** [1].

8.2.2 Hiding Coefficients

Another approach to cryptography can come from hiding the code itself. We have seen that in packets, the coding coefficients are a small overhead. However, the number of choices of coding packets is itself of the size of the key for a system such as AES. As mentioned before, AES will have, say 128 bits scrambling, so that the key size is itself around 128 bits (it is a little less because certain choices of scrambling, such as leaving the plaintext unchanged, are not practically viable).

The coefficients for coding together, say p packets over a prime field $\mathbb{F}_q$ will be $p\log_2(q)$. For a moderate number p and q, the number of bits needed to describe the coefficients is of the size of the key for a classical cryptographic system such as AES. For example, if AES needs 128 bits, then with $q = 2^8$ one can describe the coding coefficients for up to $p = 16$ packets and vice versa.

Consider again our example with three equations, so $p = 3$.

$$X_1 = \alpha_{1,1}M_1 + \alpha_{1,2}M_2 + \alpha_{1,3}M_3$$

$$X_2 = \alpha_{2,1}M_1 + \alpha_{2,2}M_2 + \alpha_{2,3}M_3$$

$$X_3 = \alpha_{3,1}M_1 + \alpha_{3,2}M_2 + \alpha_{3,3}M_3$$

Recall that each of the $\{M_i\}$ can be a vector over $\mathbb{F}_q$, so that the coding coefficients are amortized over its length. Assume that Bob knows the coefficients $\{\alpha_{i,j}\}$ but Eve does not. If the elements of each M_i are uniformly, independently, identically distributed over $\mathbb{F}_q$, then the elements of each X_is are also uniformly, independently, identically distributed over $\mathbb{F}_q$ as long as the $\{\alpha_{i,j}\}$s are chosen in a way where the coding matrix is invertible. Thus, it is not possible for Eve to glean information about, let alone reconstruct, the original messages, the $\{M_i\}$, by observing the $\{X_i\}$.

The coding coefficients thus become the secret key! That key may be shared separately between Alice and Bob, or the coefficients can be transmitted from Alice to Bob, for instance in the header of a coded packet, but hidden by another cryptographic scheme. As before, the security of the coding scheme is then effectively and efficiently outsourced to another cryptographic scheme that meets the requirements of the application at hand.

8.3 Pollution Attacks

Network coding provides some inherent security as we saw in the previous section 8.2. However, it also introduces new vulnerabilities in the system, especially if an adversarial user intrudes on the network. When network coding provides better security than legacy routing schemes to passive attacks, the effect of active attacks like Byzantine modification can be very dangerous with NC-based networks. The NC schemes allow the intermediate nodes to recode and mix the packets on the fly which makes it important to identify if any activity by a node is legitimate or not. In traditional routing networks, any modification to the packets on the fly can be considered inherently as a malicious activity, whereas for NC the modification of packets on the fly makes the essence of the scheme. Thus active attacks that directly disrupt the network operation or packets being transmitted become dangerous in NC environments [2].

Byzantine modification or pollution attacks are the most popular and dangerous attack among the different network coding security challenges. Also, it is one of the most studied security challenges specific to NC environment. In pollution attacks, the malicious insider nodes will perform incorrect coding operations and send invalid packets over the downstream links. This will lead to the decoding of incorrect packets at the sink and negate whatever throughput efficiency being achieved by NC. However, the intermediate nodes cannot be prevented from coding the packets it received, thus preventing pollution attacks is difficult. Pollution attacks are epidemic if unchecked at the earliest possible node. Once a polluted packet is introduced in the network, it could spread over all the paths it travels and multiplies the degradation of the throughput efficiency. Fig. 8.1 shows the conditions of Random Linear Network Coding (RLNC) without a malicious node in the network, and Fig. 8.2 shows an RLNC network including a malicious node.

8.3.1 Detection of Pollution After Decoding

When coding data, one must take care that the equations not be corrupted. Changing a single equation can affect the decoding of all of the messages involved in that equation, with a potential domino effect.

Exercise 8.5

For (8.5), choose a set of M_is and compute the corresponding X_is. Change one of the X_is. Compute the M_is corresponding to the new set of X_is. Note that all of the M_is are affected.

Thus, it is important to detect what is generally known as a pollution attack, where an equation has been changed. In this section, we shall consider two different ways of thwarting such attacks.

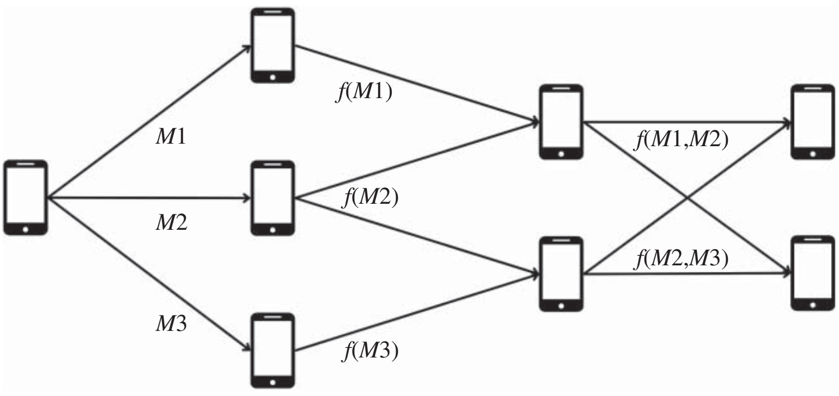


Figure 8.1 Benign RLNC scenario.

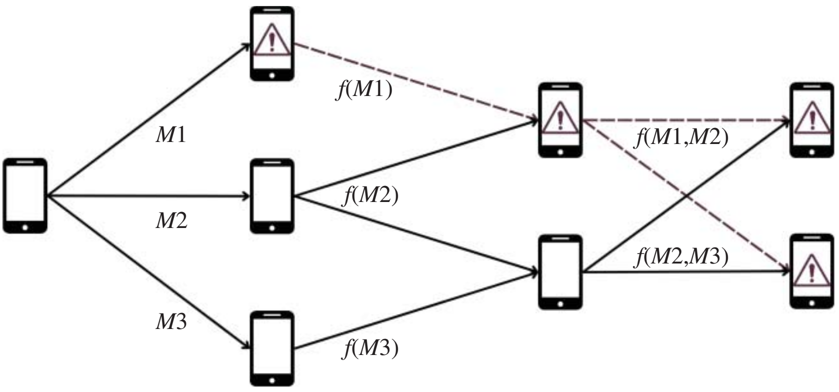


Figure 8.2 Malicious scenario showing the spread of a polluted packet.

One of the most immediate countermeasures to a pollution attack is to detect that it has occurred. In Exercise 8.5, the M_is recovered from the modification of a single X_i are consistent with the equations seen. There is no immediate way to detect that one of the X_is was incorrect.

One approach is to add an extra check, or hash. For example, if we know that only one of the equations can be polluted, then adding one more equation will show that there is a problem. This approach will not flag which equation was polluted, but it will show that pollution indeed did take place.

One can readily see how to extend that technique beyond error detection. Suppose that still only one variable was polluted. If instead of adding one more equation, we add two equations, then we can check for a set of $p + 1$ consistent equations, where we use a set of p equations to decode, one equation to check that the X corresponding to that equation is the one obtained from the M_is of the set of p equations. This is a simple form of error correction, where we detect an error, turn it into an erasure, and use erasure correction by ignoring the polluted X.

We can also use non-linear checks. Suppose that we have vectors $\underline{M_i} = (m_i[1], \ldots m_i[n])$.

We now send a non-linear check, for example $\sum_{i=1}^{p} \sum_{j=1}^{n} m_i[j]^2$, for instance by appending it or prepending it to at least one of the X_i. Unless an attacker is able to modify *all* of the $\underline{X_i}$s, they cannot reliably modify any subset of X_is to make them consistent with high probability with the hash. This is the case even if the attacker is able to modify the hash itself.

Exercise 8.6

Provide a hash for a given set of non-zero M_is (8.5) extended to the vector case when $n = 8$ with the non-linear hash $\sum_{i=1}^{p} \sum_{j=1}^{n} m_i[j]^2$. Verify that the check will work if one or two Xs are polluted. Consider another non-linear function and check that you observe a similar behavior.

The choice of the non-linear function turns out not to be very important. The main phenomenon we are using is that linear and non-linear functions act differently over finite fields. The overhead here is small, with only one element of $\mathbb{F}_q$ per $p \times n$, which is less than one equation per p. The problem is that it is not clear where the pollution took place. In general, it would be necessary to ignore all of the equations that shared variables with a polluted one. This approach will ensure that decodings are correct, but it could potentially allow an attacker to affect many packets by polluting only a small number of them.

8.3.2 Detection of Pollution Without Decoding**

Being able to determine on a single packet whether the equation is correct may seem at first hopeless. Checking that an equation is indeed the equation it pretends to be when we have not decoded and figured out the messages from which the equation was constructed may appear to violate the understanding of network coding that we have developed.

We shall use what we know of finite fields and some techniques of standard cryptography. First, let us explicitly represent every packet with the coefficients prepended. Thus, the p original packets or messages $\underline{M_i} = (m_i[1], \ldots m_i[n])$ in $\mathbb{F}_q^n$ will have a prepended vector of length p which is all 0s except for a 1 in the i^{th} position. The new packet or message, let us call $\underline{M_i'}$, is in $\mathbb{F}_q^{n+p}$. The $\underline{X_i}$s are obtained as

$$\underline{X_i'} = \sum_{j=1}^{p} \alpha_{i,j} \underline{M_j'}.$$

Thus, each $\underline{X}'_i$ has prepended to it the coefficients that were used to create it.

For our scheme, we shall need a prime r such that q divides $r - 1$. We are not claiming that this is easy, but it is something that schemes such as the celebrated Diffie–Hellman scheme in cryptography routinely provide. We shall also need g which is a generator of the group G of order q in $\mathbb{F}_r$, which means that all q elements of G are given by powers of g. Again, we are not considering the difficulty of finding such a g, or why it is possible, but making use of the fact that some conventional cryptosystems already make it possible to find such a g.

Alice will compute a vector $\underline{u}$ that is perpendicular to all of the $\underline{M}'_i$ in $\mathbb{F}_q^{n+p}$, where perpendicular means that the inner product of $\underline{u}$ and $\underline{M}'_i$, which is $< \underline{M}'_i, \underline{u} > := \sum_{j=1}^{n} m'_i[j]u[j]$, computed over $\mathbb{F}_q$ for every i between 1 and p, is 0.

Exercise 8.7

Check that $< \underline{X}'_i, \underline{u} >$ is 0.

We shall generate a private key by Alice, who selects uniformly at random $p + n$ elements $(a[1], \dots, a[p+n]) := \underline{a}$ from $\mathbb{F}_q \backslash \{0\}$. Note that the number of bits required to describe that key is $(p+n)\log(q)$, which is generally quite large, so this is rather secure key. Alice then uses a public key cryptography scheme, such as RSA, to publish a public key, which is seen by Eve and Bob. That public key is $\underline{h} := \left(g^{a[1]}, \dots, g^{a[p+n]}\right)$. The difficulty of finding $\underline{a}$ by observing $\underline{h}$ is the *discrete-log* problem. The q is not known to Eve, so the g is not either.

Alice computes $\left(u[1].a[1]^{-1}, \dots u[p+n].a[p+n]^{-1}\right) := \underline{s}$, signs it with an existing standard signature scheme and publishes it. This will be the means for Bob to check that received packets $\underline{X}'_i$ are correct.

Bob will verify that a packet $\underline{X}'_i$ is correct by the following checking operation in $\mathbb{F}_r$

$$\begin{aligned}
0 &\overset{?}{=} \prod_{j=1}^{p+n} h[j]^{s[j].X'_i[j]} \\
&= \prod_{j=1}^{p+n} g^{a[j]^{s[j].X'_i[j]}} \\
&= \prod_{j=1}^{p+n} g^{a[j].s[j].X'_i[j]} \\
&= \prod_{j=1}^{p+n} g^{a[j].u[j].a[j]^{-1}.X'_i[j]} \\
&= \prod_{j=1}^{p+n} g^{u[j].X'_i[j]} \\
&= g^{<\underline{X}'_i, \underline{u}>}.
\end{aligned} \tag{8.6}$$

Through (8.6) we see that Bob is able to verify on a per-packet basis that the packet is indeed a correct encoding of the original messages *even though he does not know yet the result of decoding*. We have used here the fact that we are able to do a form of linear homomorphic encryption for the finite fields in which we are operating. In particular, the cryptographic scheme allows verification not only of the original $\underline{M}_i'$s but also of linear operations in $\mathbb{F}_q$ over those original $\underline{M}_i'$s.

8.4 Summary

This chapter focuses on security aspects related to NC implementations. We provide a background on the concepts of information theory and security and then build on top of those to explain how NC can be used to enable secure communication. By combining NC with concepts from cryptography, you can achieve energy-efficient approaches while meeting practical security requirements. The approaches of partial encryption after encoding or hiding coefficients can prove to be beneficial in many practical cases. We consider potential adversarial caveats to the security of the network coding, such as pollution attacks, but also discuss some of the potential solutions for such security challenges. Having completed this chapter, you should have learned about:

(A) Basic concepts of information theory and security and how it pans out in the network coding regime.
(B) Network coding-inspired cryptosystems.
(C) Pollution attacks and potential solutions to detect pollution.

Additional Reading Materials

Discussions on hiding equations are originally based on [3], and implications on security are discussed in further detail in [4–8]. A survey on pollution attacks is present in [9] and [10–12], which discuss some integrity schemes for the detection of pollution attacks.

References

1 A. Cohen, R. G. L. D'Oliveira, S. Salamatian, and M. Médard, "Network coding-based post-quantum cryptography," *IEEE Journal on Selected Areas in Information Theory*, vol. 2, no. 1, pp. 49–64, 2021.

2 V. N. Talooki, R. Bassoli, D. E. Lucani, J. Rodriguez, F. H. Fitzek, H. Marques, and R. Tafazolli, “Security concerns and countermeasures in network coding based communication systems: a survey,” *Computer Networks*, vol. 83, pp. 422–445, 2015.

3 L. Lima, M. Médard, and J. Barros, “Random linear network coding: a free cipher?” in *IEEE International Symposium on Information Theory*, 2007, pp. 546–550.

4 L. Lima, J. P. Vilela, J. Barros, and M. Médard, “An information-theoretic cryptanalysis of network coding-is protecting the code enough?” in *IEEE International Symposium on Information Theory and Its Applications*, 2008, pp. 1–6.

5 L. Lima, S. Gheorghiu, J. Barros, M. Médard, and A. L. Toledo, “Secure network coding for multi-resolution wireless video streaming,” *IEEE Journal on Selected Areas in Communications*, vol. 28, no. 3, pp. 377–388, 2010.

6 K. Han, T. Ho, R. Koetter, M. Médard, and F. Zhao, “On network coding for security,” in *IEEE Military Communications Conference*, 2007, pp. 1–6.

7 P. F. Oliveira, L. Lima, T. T. V. Vinhoza, J. Barros, and M. Médard, “Coding for trusted storage in untrusted networks,” *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 6, pp. 1890–1899, 2012.

8 N. Cai and R. W. Yeung, “Secure network coding on a wiretap network,” *IEEE Transactions on Information Theory*, vol. 57, no. 1, pp. 424–435, 2011.

9 V. A. Vasudevan, C. Tselios, and I. Politis, “On security against pollution attacks in network coding enabled 5G networks,” *IEEE Access*, vol. 8, pp. 38416–38437, 2020.

10 S. Jaggi, M. Langberg, S. Katti, T. Ho, D. Katabi, and M. Médard, “Resilient network coding in the presence of Byzantine adversaries,” in *IEEE International Conference on Computer Communications*, 2007, pp. 616–624.

11 F. Zhao, T. Kalker, M. Médard, and K. J. Han, “Signatures for content distribution with network coding,” in *IEEE International Symposium on Information Theory*, 2007, pp. 556–560.

12 T. Ho, B. Leong, R. Koetter, M. Médard, M. Effros, and D. R. Karger, “Byzantine modification detection in multicast networks with random network coding,” *IEEE Transactions on Information Theory*, vol. 54, no. 6, pp. 2798–2803, 2008.

Concluding Remarks

We hope that this book serves as a handy reference for readers even after they have completed their class or course of self-study. We should note that, while the goal of the book was of course to instruct from an academic perspective, it also aims to guide and aid in implementation. In this respect, even techniques that have become extremely well known and are covered in this book often have underlying intellectual property associated to them, and it behooves the careful practitioner to be aware of this background, contacting the relevant assignees for any implementation. Such an approach is generally recommended, and it is of particular importance in this field. We provide a representative but not necessarily exhaustive list in the appendix.

Network coding continues to evolve as a technology *per se*, and also as a tool incorporated into different projects. In general, as issues of network variability, reliability, latency, and security rise in importance, so does the impetus for using the techniques that this book teaches. We hope that you enjoy, as much as we have over the years, exploring these capabilities and that network coding becomes a trusted element in your engineering toolbox. We look forward to seeing your work in the domain, your take on the field, the innovative ways in which you incorporate the approaches and to welcome you to the community of network coding practitioners.

Appendix

Sample List of Patents

- D. S. Lun, M. Médard, T. Ho, R. Koetter, and N. Ratnakar, “Minimum Cost Routing with Network Coding,” US Patent 7,414,978.
- T. Ho, R. Koetter, M. Médard, D. Karger, and M. Effros, “Randomized Distributed Network Coding,” US Patent 7,706,365.
- S. Deb, M. Médard, and R. Koetter, “A Random Linear Coding Approach to Distributed Data Storage,” US Patent 8,046,426.
- J.-K. Sundararajan, D. Shah, M. Médard, and P. Sadeghi, “Feedback-based Online Network Coding,” US Patent 8,068,426.
- S. Deb and M. Médard, “A Network Coding Approach to Rapid Information Dissemination,” US Patent 8,102,837.
- J.-K. Sundararajan, D. Shah, M. Médard, M. Mitzenmacher, J. Barros, and S. Jakubczak, “Method and Apparatus Providing Network Coding Based Flow Control,” US Patent 8,130,776.
- D. Lucani, M. Médard, and M. Stojanovic, “Random Linear Network Coding for Time Division Duplexing,” US Patent 8,279,781.
- S. Deb, M. Médard, and R. Koetter, “A Random Linear Coding Approach to Distributed Data Storage,” US Patent 8,375,102.
- G. Angelopoulos, M. Médard, and A. Chandrakasan, “Partial Packet Recovery in Wireless Networks,” US Patent 8386892.
- D. Lucani, M. Médard, and M. Stojanovic, “Random Linear Network Coding for Time Division Duplexing,” US Patent 8,451,756.
- D. Lucani, M. Kim, F. Zhang, X. Shi, M. Médard, and M.-J. Montpetit, “Network Coding for Multi-resolution Multicast,” US Patent 8,473,998.
- J.-K. Sundararajan, D. Shah, M. Médard, M. Mitzenmacher, J. Barros, and S. Jakubczak, “Method and Apparatus Providing Coding Based Flow Control,” US Patent 8,526,451.
- L. Lima, S. Gheorghiu, J. Barros, M. Médard, A. Toledo, and J. Vilela, “Secure Network Coding for Multi-resolution Wireless Video Streaming,” US Patent 8,571,214 B2, European Patent EP1173559.0.

- M. Kim, M. Médard, and A. ParandehGheibi, “Coding Approach for a Robust and Flexible Communications Protocol,” US Patent 8,780,693.
- A. ParandehGheibi, M. Kim, and M. Médard, “Advertisements as Keys for Streaming Protected Content,” US Patent 8,918,902.
- U. Ferner and M. Médard, “Method and Apparatus to Reduce Access Time in a Data Storage Device Using Coded Seeking,” US Patent 9,019,643.
- L. M. Zeger, M. Médard, and A. Peters, “Method and Apparatus for Efficient Transmission of Information to Multiple Nodes,” US Patent 9,025,607.
- L. Lima, S. Gheorghiu, J. Barros, M. Médard, A. L. Toledo, and J. Vilela, “Secure Network Coding for Multi-resolution Wireless Transmission,” US Patents 8,571,214 9,923,714, 9,137,492.
- L. M. Zeger, M. Médard, and A. Rezaee, “Traffic Backfilling via Network Coding in a Multi-packet Reception Network,” US Patent 9,143,274.
- M. Médard, A. Eryilmaz, and A. Ozdaglar, “Method for Coding-based, Delay-efficient Data Transmission,” US Patent 9,160,440.
- M. Médard, X. Shi, M. Montpetit, S. Teerapittayanon, and K. Fouli, “Wireless Reliability Architecture and Methods Using Network Coding,” US Patent 9,185,529.
- G. Angelopoulos, M. Médard, and A. P. Chandrakasan, “Partial Packet Recovery in Wireless Networks,” US Patent 9,203,441.
- M. Médard, X. Shi, M.-J. Montpetit, S. Teerapittayanon, and K. Fouli, “Wireless Reliability Architecture and Methods Using Network-coding,” US Patent 9,271,123.
- M. Médard, X. Shi, M.-J. Montpetit, S. Teerapittayanon, and K. Fouli, “Wireless Reliability Architecture and Methods Using Network-coding,” US Patent 9,253,608.
- U. Ferner and M. Médard, “Coded Seeking Apparatus and Techniques for Data Retrieval,” US Patent 9,361,936.
- L. M. Zeger, M. Médard, and A. Rezaee, “Method and Apparatus for Reducing Feedback and Enhancing Message Dissemination Efficiency in a Multicast Network,” US Patent 9,369,255.
- F. du Pin Calmon, W. Zeng, and M. Médard, “Method and Apparatus for Implementing Distributed Content Caching in a Content Delivery Network,” US Patent 9,369,541, also Chinese Patent, notice of allowance received, also European Application 14776562.2 (notice of allowance received), also Japanese Application No. 2016-501519 (notice of allowance received), also Korean Application.
- F. du Pin Calmon, J. Cloud, M. Médard, and W. Zeng, “Multi-path Data Transfer Using Network Coding,” US Patent 9,537,759, also European Application.
- L. Zeger, J. Cloud, and M. Médard, “Joint Use of Multi-packet Reception and Network Coding for Performance Improvement,” US Patent 9,544,126.

- L. Zeger, M. Médard, and A. Rezaee, “Traffic Backfilling via Network Coding in a Multi-packet Reception Network,” US Patent 9,559,831.
- M. Médard, U. Ferner, and T. Wang, “Network Coded Storage with Multi-resolution Codes,” US Patent 9,607,003, European 2974297, also Japanese, Chinese Patent Applications, (Korean Application 10-2015-7029104 notice of allowance received).
- S. Deb, M. Médard, and R. Koetter, “A Random Linear Coding Approach to Distributed Data Storage,” US Patent 9,680,928.
- M. Kim, D. Lucani, M. Médard, M.-J. Montpetit, X. Shi, and F. Zhao, “Network Coding for Multi-Resolution Multicast,” US Patent 9,762,957.
- M. Kim, M. Médard, and A. ParandehGheibi, “Coding Approach for a Robust and Flexible Communications Protocol,” US Patent 9,877,265.
- L. Lima, S. Gheorghiu, J. Barros, M. Médard, A. L. Toledo, and J. Vilela, “Secure Network Coding for Multi-resolution Wireless Transmission,” US Patent 9,923,714.
- B. Haeupler and M. Médard, “Method and Apparatus for Performing Finite Memory Network Coding in an Arbitrary Network,” US Patent 9,998,406.
- F. du Pin Calmon, J. Cloud, W. Zeng, and M. Médard, “Multipath Data Transfer Using Network Coding,” US Patent 10,009,259; European Patent Application 13742964.3 (notice of allowance received).
- M. Médard, A. Eryilmaz, and A. Ozdaglar, “A Method of Coding-based, Delay-efficient Data Transmission,” US Patent 10,027,399.
- A. G. Saavedra, M. Karzand, D. Leith, and M. Médard, “Low-delay Packet Erasure Coding,” US Patent 10,243,692, European Patent 3,117,546, Chinese Patent Application 201580024057.8.
- F. du Pin Calmon, M. Médard, L. Zeger, M. Christiansen, and K. Duffy, “Method and Apparatus for Secure Communication,” US Patent 10,311,243.
- M. Médard, U. Ferner, and T. Wang, “Network Coded Storage with Multi-resolution Codes,” US Patent 10,452,621.
- X. Shi and M. Médard, “Secure Network Coding for Multi-description Wireless Transmission,” US Patent 10,530,574.
- K. Konwar, P. N. Moorthy, N. Lynch, and M. Médard, “Layered Distributed Storage System and Techniques for Edge Computing Systems,” US Patent 10,735,515, Chinese Patent Application 201880033298.2, European Patent 3631641, Indian Patent Application 201947051525.
- A. G. Saavedra, M. Karzand, D. Leith, and M. Médard, “Low-delay Packet Erasure Coding,” US Patent 10,756,843.
- M. Médard, K. Konwar, P. N. Moorthy, N. Lynch, E. Kantor, and A. A. Schwarzmann, “Storage-optimized Data-atomic Systems and Techniques for Handling Erasures and Errors in Distributed Storage Systems,” US Patent 10,872,072.

- M. Médard, P. N. Moorthy, and V. Abdrashitov, “System for De-duplicating Network Coded Distributed Storage and Related Techniques,” US Patent 11,025,600.
- M. Médard, A. Cohen, R. D’Oliveira, and S. Salamatian, “Network Coding-based Post-quantum Cryptography,” US 2022/0069987 A1.
- K. Fouli and M. Médard, “Linear Network Coding with Pre-determined Coefficient Generation Through Parameter Initialization and Reuse,” US Patent 11,108,705.
- M. Médard, U. J. Ferner, and W. Tong, “Network Coded Storage with Multi-resolution Codes,” US Patent 11,126,595.
- M. Médard, L. V. Jánoky, and P. J. Braun, “System and Technique for Generating, Transmitting and Receiving Network Coded (NC) Quick UDP Internet Connections (QUIC) Packets,” US Patent 11381339-B2.
- K. Fouli and M. Médard, “Multipath Coding Apparatus and Related Techniques,” US Patent 11418449-B2.
- K. Fouli, F. Gabriel, M. Médard, S. Pandi, and S. Wunderlich, “System and Technique for Sliding Window Network Coding-based Packet Generation,” US Patent US-11424861-B2.
- F. du Pin Calmon, M. Médard, and K. Fouli, “Method and Apparatus for Coded Multipath Networking Using Path Management and Protocol Translation,” US Patent 11463372-B2.
- M. Médard and C. H. U. Hellge, “Apparatus and Method for Multi-code Distributed Storage,” US Patent 11463113-B2.
- F. du Pin Calmon, M. Médard, and K. Fouli, “Method and Apparatus for Coded Multipath Network Communication,” US Patent 11489761-B2.
- M. Médard, D. Malak, and A. Cohen, “Adaptive Causal Network Coding with Feedback,” US Patent 20230123204, April 2023.
- M. Médard, K. Fouli, and F. Du Pin Calmon, “Multipath Coding Apparatus and Related Techniques,” European Patent 3794755.
- K. Fouli and M. Médard, “Multipath Coding for Patent Latency Reduction,” European Patent 19729119.8.
- M. Médard, R. D’Oliveira, A. Cohen, and S. Salamatian, “Network Coding-based Secure Communication,” European Patent 4205345 A1.

Index