Fault Tolerant Network Coding

A Project Report

submitted in partial fulfillment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

by

Ravishankar Krishnaswamy

under the guidance of

Prof. C. Pandu Rangan

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

May 2007

Certificate

This is to certify that this project report titled **Fault Tolerant Network Coding** submitted by **Ravishankar Krishnaswamy** in partial fulfillment for the award of the degree of **Bachelor of Technology** in Computer Science and Engineering, is a bona-fide record of work carried out by him under my guidance and supervision in the Department of Computer Science and Engineering, Indian Institute of Technology, Madras.

Place : Chennai   Prof. C. Pandu Rangan

Acknowledgments

Firstly, I would like to offer my heartfelt thanks to Prof. C. Pandu Rangan, my project advisor. Had I not listened to his wonderful lectures on the design and analysis of algorithms, I may not even have developed a passion for research in the field of theoretical computer science. All his courses have been extremely enjoyable and valuable. Not only did they deeply imbibe the fundamentals of theory, but they were also highly inspirational and motivational. He has introduced me to several interesting "hot" areas in this field, the focus of this thesis being one of them. He has also made sure that the lab environment is perfect for research and has whatever one would need - books on almost all areas in theory and state of the art computers. All in all, he has made working in the theory lab a delightful and truly memorable experience.

I would also like to thank Dr. N.S. Narayanaswamy, my faculty advisor and the professor who took my first class in IIT Madras. He has always been a constant source of encouragement and support. I have thoroughly enjoyed and benefited greatly from all my discussions with him, academic or otherwise. I would like to convey my gratitude to Dr. B. Ravindran, Dr. Shankar Balachandran, and Dr. C. Chandrasekar, whose courses were really enjoyable and very informative. Dr. Shankar, with his friendly approach towards the students, and his unique ways of making class very interesting, has truly been a source of inspiration. Many thanks to all the other members of the faculty too. I have completely enjoyed my undergraduate life in the CS Department. Outside the department, I extend my sincere thanks to Professors P. Veeramani of the Department of Mathematics and G. Srinivasan of the Department of Management Studies. Their courses have been among the best that I have attended. I have gained a lot through my interactions with them.

I would also like to express my gratitude to my internship mentor Prof. Ravi Sundaram. Working with him was a delight and I truly experienced the joy of doing research. I will eternally remember the brainstorming sessions that we had held for hours trying to find a solution to the problem we were working on. I realized that pursuing research takes grit, determination, perspiration and inspiration.

I would like to thank my friends Aravindan (Captain) and "RG" Karthekeyan. They have made my stay in the theory lab very special and extremely memorable. I have learnt a lot about how to approach problems and think out of the box from Aravindan, especially during my internship and B.Tech project, when we were working on the same problems. He has always been a source of motivation right from my school days. RG has also been a great friend and lab companion for the last two years. With him around, there has never been a dull moment in the lab. We have had a lot of fun during our lab "night-outs". My thanks also to the other lab mates KP, Suri, Meena, Ashish, Arpita, Amjed, Harini, Madhu, Masila Mani, Sharmila, all of whom have been very friendly and made this lab a fun place. I would also like to thank my immediate seniors Rajsekar, Ranjith, Ravichandra, and Kabi for all the help they extended whenever I needed some.

I am also indebted to my CS mug group - Captain, RG, Keki and KL. We really had a wonderful time mugging for the exams and doing the assignments. I am also grateful to my batch mates Laddoo, Gandhe, Atuls, Tigre, RK (Pogo), Karra, Harsha, Buchi and the rest of the computer science batch of 2007 (also the duals of the 2008 batch). I have had many unforgettable moments with them in class, in the KV cricket sessions, and in the DCF. I wish all of them the very best of luck in their future endeavors. I would also like to thank the digtams gumbal - wing mates Raghu, Range, Tooms, Keki, Zooper Buh, Doc, Aziz, Kuh, Veezhay, and KL, Bubs and Captain. They have made my hostel life in Mandak extremely memorable. Mandak 326 has truly been my home away from home. Memories from the 5^{th} wing shall never be cached out. I would also like to express my thanks to k7, iMani, nathan, and many more friends from my VM school days. I have had a great time in their company.

Lastly, I would like to thank my parents and close relatives for constantly supporting me and encouraging me throughout my life. Appa and Amma have always been there with me, during the times of pain and the moments of joy.

Abstract

Consider a communication network in which a source node wishes to multicast information to some sink nodes on the network. In the traditional setting, every node can only pass on the data it receives from incoming links to the links leaving it. We consider an extended setting which gives the intermediate nodes more "power". In our model, each node may send any *linear combination* of the received data on the outgoing links. Such protocols, known as *Linear Network Coding* schemes, have been proved to guarantee optimal multicast throughput (that is, the maximum number of different data packets the source can transmit to all the sinks in one execution of the protocol matches the theoretical bound). In this work, we address the problem of fault tolerance in network coding. More specifically, we are interested in obtaining reliable coding schemes which guarantee optimal (in a sense we shall explain later) throughput even when some edges stop functioning, or are corrupt by noise.

We modify an existing algorithm to provide a centralized scheme for finding such codes that tolerate any failure pattern from a polynomially bounded set of possible failure patterns in a network. When an edge fails, it is assumed to transmit a zero message. A failure pattern is a set of edges that fail. Our algorithm is rate optimal in the asymptotic limit with respect to the message packet size, in the sense that it sustains a flow rate equal to the minimum of the *max-flows* from the source to the sinks with the failed edges being deleted. Such codes could be utilized in networks where certain edges are prone to failure, and yet optimal throughput is expected regardless of the edges failing. We also present simple extensions which optimally tolerate corruption of edges by white noise (they transmit a purely random message).

Finally, we present some interesting optimization problems (*min-cost*, *min-latency*, and associated variants) that are of a high degree of relevance to the network coding setting. We explain why *min-cost* can be solved efficiently and why *min-latency* can not.

Table of Contents

CHAPTER 1

Introduction

1.1 Motivation for Network Coding

Consider the *single source multicast transmission* problem. That is, given a graph, a source node, and a set of sink nodes, we wish to maximize the multicast throughput (number of message packets the source can send to all the sinks). Each edge is assumed to be able to transmit a single message packet. The *information theoretically maximum possible throughput* that can be achieved in such networks is trivially the minimum of the *max-flows* from the source node to each of the sink nodes. Whether this rate can be achieved at all, and if it can be, how it can be achieved efficiently in practical scenarios are two of the most important throughput related questions in the field of networking.

In existing computer networks, information is transmitted from the source node to each destination node through a chain of intermediate nodes by a method known as *store-and-forward*. In this method, message packets received from an input link of an intermediate node are stored and a copy is forwarded to the next node via an output link. It is easy to see that the multicast of each message packet forms a steiner tree rooted at the source, containing the sinks. Thus, the problem of deciding whether k messages could be multicast from the source to the sinks is equivalent to the problem of deciding whether it is possible to pack k different mutually *edge-disjoint* steiner trees rooted at the source and containing the sinks. The condition of edge-disjointedness is required as each link can transmit only a unique message packet. This problem, however, is known to be computationally intractable (NP-Hard) [1]. In fact, it is hard to even obtaining a constant factor approximation algorithm for the optimization version of the above problem. Further, even the optimal solution for the steiner packing problem would not provide us with the information theoretically maximum achievable throughput in the network. This has led to research in other approaches that differ from traditional routing protocols even in the most fundamental assumptions.

1.2 Network Coding - An Introduction

Network Coding is a new research area that has several interesting applications in the design of practical networking systems. In network coding, the intermediate nodes may send out packets that are a function of the previously received information packets, as opposed to just *storing and forwarding* the incoming packets. In other words, intermediate nodes *"process"* the incoming message packets before transmitting them on the outgoing links. Network coding essentially capitalizes on the fact that information flows differ from commodity flows in that they can be both processed and duplicated at intermediate nodes.

1.3 Advantages

One may wonder in what way all this really helps. There are two main benefits of this approach.

1. Gain in throughput
2. High degree of robustness

1.3.1 Throughput Gain

With regard to the throughput, it is easy to see that network coding can not do worse than today's routing algorithms as it subsumes these algorithms, i.e, the store-and-forward mechanism too is essentially a case where the outgoing packet is a function of the incoming packets. In addition, there are classes of examples where network coding clearly outperforms store-and-forward algorithms. The famous *butterfly network example* demonstrates one such case where network coding fares better. Moreover, there exist efficient network coding protocols which obtain optimal multicast throughput.

In the butterfly network (Figure 1.1), there are two sources (at the top of the picture), each having knowledge of some value A and B. There are two destination nodes (at the bottom), and each wants to know both A and B. Each edge can carry only a single value (we can think

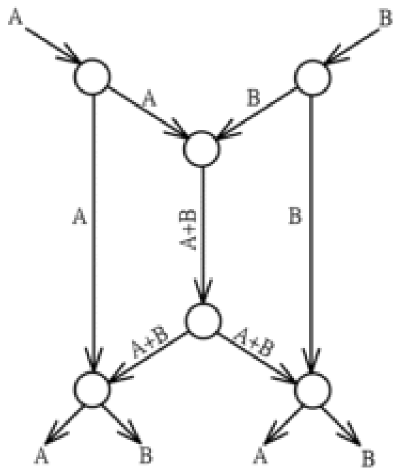

Figure 1.1: The Butterfly Example

of an edge transmitting a bit in each time slot). If we only used routing, then the central edge would be able to carry A or B, but not both. Suppose we send A through the center; then the left destination would receive A twice and not know B at all. Sending B poses a similar problem for the right destination. We say that routing is insufficient because no routing scheme can transmit both A and B simultaneously to both destinations. Using a simple code, we get both A and B to both destinations simultaneously by sending the sum of the symbols through the center (in other words, we encode A and B using the formula "A+B"). The left destination receives A and A+B, and can find B by subtracting the two values.

1.3.2 Inherent Robustness

The other advantage of network coding is robustness. Since the intermediate edges do not directly transmit the actual message packets that the source wishes to broadcast, and only carry linear combinations of them, it may not be possible for a passive adversary to figure out even parts of the original messages. This provides a sense of security against passive adversaries. For instance, consider the same butterfly network, and let the adversary have "snooping" powers on the central edge. Without network coding, he would be able to obtain complete information about either A or B. However, when network coding is employed, though he gains the same amount of total information, he gets it in the form of A+B. He therefore gathers not much useful information about A and B. Such an advantage translates into resilience and provide

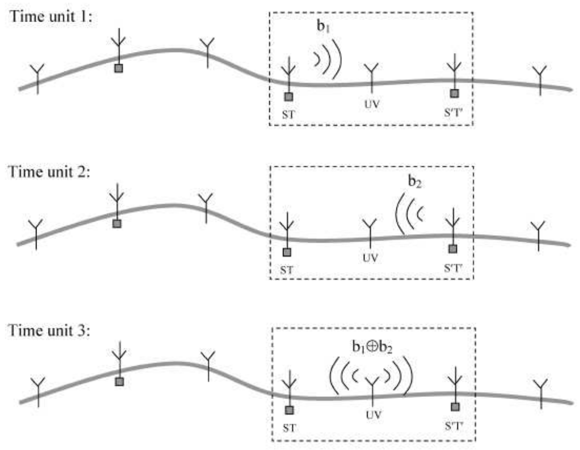

Figure 1.2: Network Coding in a Wireless Setting

more secure multicast schemes.

1.4 Applications

Because of its throughput and robustness advantages, network coding has an increasing application base including but not limited to:

- Alternative schemes to forward error correction and ARQ in traditional networks.
- Networks with robustness and resilience towards attacks like snooping, eavesdropping or replay attacks.
- Digital file distribution and P2P file sharing.
- Bidirectional low energy transmission in wireless sensor networks.
- Many-to-many broadcast network capacity augmentation.

Figure 1.2 (given in [2]) demonstrates how network coding could be utilized to minimize the communication cost (energy) in a wireless broadcast setting. The transceiver in the middle, on receiving b_1 and b_2 from the extreme nodes, broadcasts $b_1 \oplus b_2$. Each receiver can therefore decode both b_1 and b_2 upon receiving $b_1 \oplus b_2$. Without network coding, the central transceiver would have to broadcast b_1 and b_2 in two separate time-slots.

In the case of P2P file sharing protocols, network coding presents the advantage that a user waiting for a file to be downloaded completely needs to wait only until users who can provide requisite number of linearly independent combinations join the network. In traditional systems, a person without some fragments of the file needed to wait for users who had that particular fragment. Since the field in which the operations are done is large, it is very probable to get newer linear combinations very soon.

1.5 Related Work

The concept of network coding was first introduced in the landmark paper [3] by Ahlswede et al. They proved bounds on the capacity of network coding in a famous result analogous to the *Max flow Min cut Theorem.* They showed that with network coding, it is possible to obtain a multicast throughput equal to the minimum of the *max-flows* separating the source from the sinks. In traditional routing, achieving this throughput level is not possible (as demonstrated in the butterfly network shown in figure 1.1). Further, achieving the maximum possible throughput that traditional routing allows equates to solving the steiner tree problem which is NP-Hard. It is hard to even be approximated to within a constant factor [1]. This is what sparked off a flurry of research in this area. Following the results of [3], Cai and Yeung present in two papers [4, 5], results analogous to those in coding theory in the network coding framework. They prove bounds on the maximum achievable broadcast rates under various settings like corruption by adversaries (analogous to the hamming bounds etc.). They also present network coding schemes that match these bounds. However, the schemes they provided were useful only to demonstrate the existence of network codes which can optimally correct these corruptions. They were not efficient, and therefore not practically useful.

Then came another seminal paper by Cai and Yeung [6] which showed that *Linear Network Coding* was sufficient to achieve the optimal throughput in a single source multicast setting. This meant that even when nodes were restricted to sending just linear combinations of incoming packets, maximum throughput schemes could be designed. Their result, however, was applicable only for the single source setting. In [7], Jaggi et al presented an efficient polynomial scheme for network code construction for a single source multicast setting. This meant that in spite of the fact that network coding outperforms traditional routing in terms of throughput, finding such codes can be done efficiently! A similar scheme is also given in [8]. In [9], a new algebraic framework of network coding is presented, and analogous results are proved in this model.

In [10], the authors analyze the concept of *Randomized Network Coding*, where each internal node makes random linear combinations of its incoming packets and transmits these on the outgoing edges. Random network coding is a highly practical model that would work with a high degree of efficiency without imposing much computational burden on the intermediate nodes also. Moreover, it is a completely localized scheme where each node requires no more information than that about its neighbours. The key concept is that nodes take random linear combinations of incoming packets and send them on outgoing links. When the sink receives the packets, it checks if it has received the required number of linearly independent packets, and solves the equations for the original messages. However, if it has not received sufficient packets to solve for the original messages, the protocol is said to have “failed ” and needs to be repeated. It is simple to observe that the success probability grows higher as the field size of operation increases. This is what makes this approach highly practical and attractive. Another paper [11], also analyzes and proves the competitive advantage that randomized network coding offers when compared to traditional protocols.

In [12], the authors introduce the first distributed polynomial-time rate-optimal network codes that work in the presence of *Byzantine nodes*. Byzantine adversaries on nodes are those which arbitrarily corrupt a packet along the outgoing edges of the node. The authors prove that when the adversary can eavesdrop on all links and jam Z links (introduce Z new arbitrary corrupt packets), a throughput rate of $C-2Z$ can be achieved, where C is the network capacity (minimum of the *max-flows* from source to each sink). This result is analogous to that in error correcting codes that tolerate arbitrary corruption in k components.

Another interesting notion of secure network coding was introduced in [13]. In this paper, the authors consider the setting where there is a passive adversary who “snoops” on a certain number of edges (say k) and the objective is to make the network *weakly secure*. They provide a suitable network coding scheme where the adversary gets no “useful” information on the intended message packets which the source wishes to multicast. That is, whatever set of k

edges he taps and obtains information from, he would not be able to solve for any of the original messages using any subset of these equations (remember that each packet is a linear combination of the original message packets). In other words, he *obtains* information about all the messages put together (about all of them put together), but not any that is *useful.* Such codes are known as *weakly secure network codes.*

There has been a lot of research on various other aspects of network coding ranging from efficient centralized and later decentralized algorithms for network coding to construction of network codes for the wireless scenario. All traditional networking problems are being solved in a network coding model. Focus has also been on more practical issues pertaining to network coding like maximum likelihood decoding in noisy networks, and efficient network coding in ad-hoc mobile environments. The network coding homepage (located at http://www.ifp.uiuc.edu/ koetter/NWC/index.html) is a very good resource for the latest happenings related to this field.

1.6 Focus

In this work, we primarily focus on the robustness (tolerance to the failure/corruption of certain edges) of multicast network coding. Essentially, we target the problem of finding efficient network codes which guarantee optimal throughput for a single source multicast setting when some edges can fail. More specifically, we are interested in efficiently designing coding schemes that can tolerate one among polynomially many failure patterns. A failure pattern is a subset of the edges indicating which edges have failed and thus transmit a *null message.* A similar setting is also considered by [7] and [9], but they assume that the sinks have knowledge about the edges that failed. Such an assumption, however, may not be valid in real-life networks. To the best of our knowledge, this is the first centralized algorithm for network coding when edges fail, without any assumptions about the knowledge of the failures. We also consider the case where packets along some edges are corrupted by white noise. Lastly, we introduce the problem of *min-latency* network coding - an optimization problem similar in flavor but distinctly different in nature to the famous *min-cost* variation of the network flow problem.

1.7 Organisation of this thesis

In Chapter 2, we introduce the preliminaries of *Network Coding* and describe the formal models we shall be using. We define our representation of the networks, and formalize linear network coding. We also explain our packet structure, and characterize failure patterns and network faults. In Chapter 3, we focus on the problem of fault tolerant network coding. We first provide a bound on the throughput that can be achieved when some edges can fail, and follow it by presenting an algorithm which finds the "optimal" network coding scheme that tolerates edge failures. We finally introduce the problem of *min-latency* network coding and identify future areas of work in this direction in chapters 4 and 5.

CHAPTER 2

Network Coding - A Formal Model

In this chapter, we formally introduce the preliminaries of *Network Coding*, and the model which we shall be using throughout the rest of this thesis.

2.1 The Network Model

Graph Representation:

We represent the network as a graph $G = (V, E)$. V is the set of nodes (vertices) and E is the set of links (edges) in the network. An edge $e \in E$ is of the form (v_1, v_2), where $v_1 \in V$ and $v_2 \in V$. Like we had mentioned in Chapter 1, we focus on the *single source multicast problem* where a source wishes to transmit some messages to a set of designated sinks. Let $s \in V$ be the source node. Let $T = \{t_1, t_2, ..., t_{|T|}\} \subseteq V \setminus \{s\}$ be the set of sinks. We assume that s is not a sink without any loss of generality, although that would not in any way affect the working of the algorithm. All the networks that we will be considering in this work are assumed to be **DAGs**. A **DAG** or a *directed acyclic graph* is a directed graph without any cycles. This assumption immediately implies that a *topological sorting* on the nodes of the network can be performed.

Topological Ordering:

A topological sort is an ordering of vertices in a **DAG** such that, if there is path from node u to node v, then v appears after u in the ordering (Therefore, a cyclic graph cannot have a topological order). A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go in one direction. A partial ordering of the edges also exists. We can place the outgoing edges of a particular node in one group, and order the groups in the topological ordering of the nodes. Within a group, we arbitrarily order the edges. With

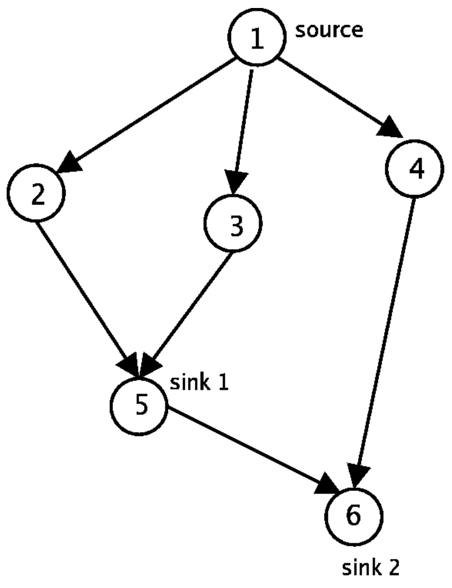

Figure 2.1: Example of a Directed Network Graph

such a partial ordering of the edges, we notice that the position of an incoming edge f to an edge e is before that of e. Figure 2.1 illustrates a sample topological ordering with an example network graph. In this figure, the nodes are numbered in the topologically sorted ordering. We can see that for any two nodes that occur at positions i and j respectively in the ordering with $i > j$, there can never be a path from i to j. These facts shall be used later in this work.

Unit Capacity Edges:

We also make an assumption that each edge $e \in E$ can transmit only a *single packet* of information. Though this may seem to be restricting the generality of the model, it actually does not. If an edge were to have a higher capacity say k, we could replace it with k links of unit capacity between the same end-vertices. Further, most of the current day networks use fixed sized packets (which could be imagined as the capacity).

Multicast Flow:

Let n be the minimum of the *max-flows* from s to each of the sinks in T. That is, the *max-flow* from s to any sink t, $t \in T$ is at least n. We notice that n is an information theoretic upper bound on the number of messages that can be simultaneously multicast to all the sinks.

2.2 Linear Network Coding

Let us suppose that the source s wishes to transmit h symbols $x_1, x_2, \ldots, x_h$ to each of the sinks in T. Unlike standard routing algorithms where the intermediate nodes just forward incoming packets, network coding is a technique which allows them to perform algebraic operations on the packets. In linear network coding, the message transmitted on any edge e is a linear combination of the messages in incoming edges. A linear network coding scheme can be described by assigning to each node, an $in * out$ matrix where *in* and *out* are the *in-degree* and *out-degree* of the node. The message(s) transmitted on the *out* outgoing edge(s), can be obtained by post-multiplying this matrix with the $1 * in$ row vector of the *in* incoming message(s). Note that each outgoing message is a linear combination of the incoming messages. Also, to make this a general model, we assume that there are h imaginary links leading to the source carrying the symbols $x_1, x_2, \ldots, x_h$ respectively. This way, the messages on the links leading out of the source (which are linear combinations of $x_1, x_2, \ldots, x_h$) can also be expressed by a matrix as defined above.

Linear network coding possesses another interesting and important property. Because the intermediate nodes transmit linear combinations of the incoming messages, we can inductively work our way to the source, and prove that each link essentially transmits a fixed linear combination of the source messages $x_1, x_2, \ldots, x_h$. Each link, can therefore be associated with a *global encoding vector* $\mathbf{b(e)}$ such that the message transmitted on the link is an inner product of $\mathbf{b(e)}$ with the vector $(x_1, x_2, \ldots, x_h)$.

2.3 Message Packet Structure

In our framework, we consider a packet p that is transmitted on an edge to be an $h+1$ tuple $(m, b_1, b_2, \ldots, b_h)$ where $m \in \mathbb{F}_{q^l}$ and $b_j \in \mathbb{F}_q$. That is, apart from transmitting the message m, we also transmit the coefficients of the linear combination of the source messages it represents. That is,

$$m = \sum_{i=1}^{h} b_i x_i \tag{2.1}$$

Also, notice that the global encoding vector for edge e is just,

$$\mathbf{b(e)} = (\mathbf{b_1}, \mathbf{b_2} \ldots, \mathbf{b_h}) \tag{2.2}$$

The overhead that the b_i's present is minimal when compared to the message content size (m is from $\mathbb{F}_{q^l}$ whereas $b_i \in \mathbb{F}_q$), and becomes negligible in the asymptotic limit $l \to \infty$.

2.4 Network Faults

We formally introduce the notion of faults (failures) in the network links. Throughout this thesis, an edge that fails is assumed to transmit a $\mathbf{0}$ message (or a null message). A receiving node can at once determine whether a particular link leading to it has behaved faultily or not. To represent a failure pattern, let us assume that the edges are first ordered as $e_1, e_2, \ldots, e_{|E|}$ (we use the partial ordering scheme mentioned in section 2.1). Now, an edge failure pattern can be represented as an $|E|$ dimensional binary vector, with the understanding that if the i^{th} dimension assumes the value 1, then the edge e_i has failed. For example, the pattern $\mathbf{p} = (\mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{0}, \ldots, \mathbf{0}, \mathbf{1})$ would mean that the edges e_1, e_2, and $e_{|E|}$ have failed in this particular multicast transmission session. We use the symbol $\mathbf{p}$ to represent a *failure pattern*. We shall also use $\mathcal{FP}$ to denote a set of failure patterns.

A network coding scheme is said to tolerate polynomially many failure patterns if it can ensure an optimal throughput under the occurrence of any failure pattern $\mathbf{p} \in \mathcal{FP}$, when $|\mathcal{FP}|$ is bounded by a polynomial function of the size of the graph ($|V|$).

2.5 Protocol Time Slots

We finally wish to note that this work deals with network coding protocols that operate in one time window. That is, once a node receives and sends data, it remains idle. This is the reason why the messages that the source wishes to send are represented as x_i and not by $x_i(t)$. Therefore, all the results that we present are for protocols that do not use such time dependencies (like taking a linear combination of $x_i(t)$ and $x_k(t-1)$. There have been, however, recent developments in constructing such time dependent network codes as well.

CHAPTER 3

Tolerance to Edge Failures

This chapter deals with efficiently constructing network codes that can tolerate what are commonly known as *fail-stop adversaries* (or) adversaries who do not corrupt the network, but rather arbitrarily determine whether the edges carry useful information or not. In other words, edges can fail arbitrarily (we assume that the failures are not intermittent). We first demonstrate how and why network coding outperforms traditional routing in the case when edges can fail, and then present a rate optimal (in the asymptotic limit) centralized algorithm for network code construction which can tolerate failure in one among polynomially many failure patterns.

We shall first describe the problem that is being solved in this section:

Problem Statement:

Given any graph $\mathbf{G}$ *with edges of unit capacity, can we construct a centralized network coding scheme that guarantees a multicast throughput of* $\mathbf{h}$ *message packets, where the minimum of the* ***max-flow*** *from the source to each sink is at least* $\mathbf{h}$ *even after deleting the edges belonging to any failure pattern* $\mathbf{p} \in \mathcal{FP}$*. This is optimal in the asymptotic limit w.r.t* $l \to \infty$*. Here, a message packet is of the form* $(m, b_1, b_2, \ldots, b_h)$*, where* $m \in \mathbb{F}_{q^l}$ *and* $b_j \in \mathbb{F}_q$*. We also assume that* $\mathcal{FP}$ *is a set of polynomially many failure patterns.*

By optimal, we mean the following: If each of the failure patterns is such that the minimum of the *max-flows* from source to each sink after deleting the edges the pattern represents is at least h, then the network coding scheme should be able to guarantee a throughput of h. Notice that even in the case some patterns actually have a higher value of the minimum of the *max-flows*, we claim "optimality" if the algorithm guarantees only a throughput of h.

3.1 Upper Bound

We now give a class of networks where we provide an upper bound on the communication rate when edges fail. Consider a network (shown in Figure 3.1) with one source and two sinks and n intermediate nodes. The source is connected to the n intermediate nodes, each of which has an outgoing arc to both the sinks. We classify the edges connecting the source and the intermediate nodes to be in *level 1*, and the edges connecting the intermediate nodes to the sinks to be in *level 2*. Clearly, the minimum of the *max-flows* from the source to each of the sinks in this setting is n. When we wish to tolerate an edge that can fail, we notice that this value drops to $n-1$. If the network nodes do not employ network coding, we see that a rate of $n-1$ cannot be achieved. This is because, if the source were to send $n-1$ packets of information over to the intermediate nodes, it could repeat only one of the packets in the remaining *level 1* edge. If $n > 1$ and a *level 1* edge not carrying a packet that is repeated fails, there is no way by which that information could be retrieved. On the other hand, with network coding, the source could send the $n-1$ information packets $x_1, x_2, \ldots, x_{n-1}$ along the first $n-1$ *level 1* edges to the $n-1$ intermediate nodes and the packet $x_1 + x_2 + \ldots + x_{n-1}$ to the last intermediate node. As an example, the packet along the left most *level 1* edge would be of the form $(x_1, 1, 0, 0, \ldots, 0)$. Note that any $n-1$ out of the n vectors that are transmitted along *level 1* edges are *linearly independent*. Therefore, even when any one edge fails, the sink will definitely receive $n-1$ linearly independent equations in the variables $x_1, x_2, \ldots, x_{n-1}$. It can therefore solve for these and obtain the original messages. Thus, this network code can tolerate failure of a single edge.

When up to k edges can fail in the network, we notice that without network coding, the maximum possible throughput is $\lfloor n/(k+1) \rfloor$. This is because each message needs to be transmitted on at least $k+1$ different *level 1* edges to the intermediate nodes. If a particular message is transmitted on only up to k *level 1* edges, these can be cut off from the intermediate nodes there by preventing them (and the receivers, eventually) from getting it. Such a circumstance can be avoided in network coding by sending out various different linear combinations (which satisfy the property that any $n-k$ of them are linearly independent) along these edges. Thus, even if k of them fail, the receiver would be able to get the $n-k$ message packets by solving the $n-k$ linearly independent equations that he would receive from the non-failed edges. This

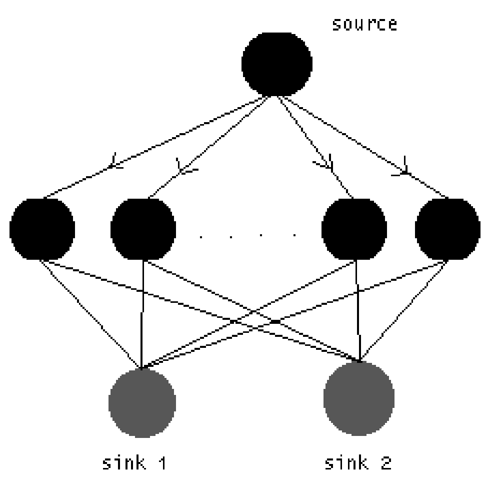

Figure 3.1: One Source Two Sink Network

is the gist of our approach in obtaining network codes that tolerate failure in edges. The next section formally describes our coding scheme that can tolerate edge failure in networks.

3.2 Optimal Fault Tolerant Network Coding Scheme

In this section, we present an efficient optimal centralized algorithm which when given a graph, produces a network coding scheme that tolerates failure in any one of polynomially many failure patterns. Our algorithm is similar in spirit to that presented in [7] which gives an efficient algorithm for optimal multicast network coding.

For convenience, we first consider the case where at most *one edge* can fail, and later extend it to more general cases. Let us reiterate the fact that n is the minimum of *max-flows* from the source to the sinks of the graph G. It is quite simple to observe that on removal of an edge, this value cannot decrease by more than 1. There are also examples where this value does indeed reduce by 1 (the one in Figure 3.1 serves as one such example). This justifies our notion of *optimality* in that our schemes guarantees a throughput of at least $n - 1$ under all circumstances.

3.2.1 Preliminaries

We assume that there are n hypothetical edges $g_1, g_2, \ldots, g_n$ leading in to the source node s.

As mentioned in Section 2.1, there exists an ordering of the nodes of the graph, and therefore an ordering of the edges as well (since we consider only directed acyclic graphs). Let us denote the ordered edges as $e_1, e_2, \ldots, e_{|E|}$.

For an edge e, let *head*(e) denote the vertex from which e originates, and *tail*(e) be the vertex that e ends up in. For $e = (u, v)$, *head*$(e) = u$ and *tail*$(e) = v$. Also note that each edge is of *unit capacity* like we had mentioned in 2.1. That is, it can transmit a single message packet.

3.2.1.1 Flow Decomposition

Just like the algorithm in [7], we first compute an n flow from the source node s to each sink $t_i \in T$. This exists because n is the minimum of the *max-flows* from s to the sinks. Further, this can be done efficiently because the problem of computing *max-flows* is solvable in polynomial time. Here is where we identify the crucial difference between network coding and traditional routing algorithms. Whilst the latter imposes a requirement that the flows be edge disjoint, the former has no such constraints as the overlapping flows can readily be mixed (in the name of performing "operations" on incoming packets). This flow can also be decomposed into n edge disjoint paths (the celebrated algorithm by Ford and Fulkerson given in [14] is one such means of achieving this). Let f_t^i be the i^{th} such path from s to sink t, and let $\mathcal{F}_t = \{f_t^i | i = 1, 2, \ldots, n\}$ be the set of the n paths connecting s with t. We address the flow represented by the paths in $\mathcal{F}_t$ as the n flow from s to t.

We use the notation *sinks*(e) to denote the set of sinks e serves. That is,

$$sinks(e) = \{t | \exists \mathbf{i} \text{ such that } \mathbf{e} \in \mathbf{f_t^i}\} \tag{3.1}$$

Also, let *prev*(e, t) be the edge f such that e and f belong to some path f_t^i. Note that this is unique for each e, t pair as there can exist only one flow path along e designated to sink t. In other words, f is the preceding edge to e in the flow from s to t. Again, it is important to remember that each edge is of *unit capacity*, and that the n flow from s to each sink t has been decomposed into n edge disjoint paths from s to t.

3.2.1.2 Global Encoding Vectors and their Sets

In a fashion similar to [7], we maintain dynamically updated sets B_t^i - the set of *global encoding vectors* (these have been introduced and defined in Section 2.1 - $\mathbf{b(e)}$ is the global encoding vector corresponding to an edge e). At each stage of the algorithm, B_t^i contains the global encoding vector for one edge from each of the n edge disjoint paths from s to t chosen by the flow decomposition procedure mentioned in section 3.2.1.1, under the assumption that edge e_i has failed throughout the working of the protocol. We also maintain the collection of sets C_t - the set of edges (one from each path from the source s to sink t) currently being considered by the algorithm. We use the notation $carried(e, j)$ to denote the global encoding vector that would be applied on edge e when the failure occurs on edge e_j. $carried(e, 0)$ would be the global encoding vector for e when no faults occur in the graph. Therefore,

$$\text{if } e \in C_t, \text{ then } carried(e, j) \in B_t^j \text{ for } 1 \leq j \leq |E|. \tag{3.2}$$

Also our algorithm ensures that,

$$carried(e, j) = \sum_{f|tail(f)=head(e)} \alpha_f \; carried(f, j) \; \forall \; j = 0, 1, 2, \ldots, |E| \tag{3.3}$$

The above equation just reiterates the fact that the outgoing message is a linear combination of the incoming messages under all circumstance.

3.2.2 Pseudo Inverse for Linear Independence

Like in [7], we also make use of the following property.

Lemma 3.2.1 *Consider a basis B of ${\mathbb{F}_q}^{n-1}$ and vectors $b \in B$, $a \in {\mathbb{F}_q}^{n-1}$ such that $\forall \; b' \in B \setminus \{b\} \; b'.a = 0$, and $b.a = 1$. Then any vector $x \in {\mathbb{F}_q}^{n-1}$ is linearly dependent on $B \setminus \{b\}$ iff $x.a = 0$.*

Proof: By the virtue of the fact that B is a basis set, any vector has a unique representation as a linear combination of elements of B. Let x be uniquely represented as $\sum_{b' \in B} x_{b'} b'$. Therefore

$$x.a = \sum_{b' \in B} x_{b'}(b'.a) \tag{3.4}$$

Therefore, $x.a = 0$ *iff* $x_b = 0$ meaning x is linearly independent of b *iff* $x.a = 0$.

We denote such an a as the *pseudo-inverse* of b with respect to B.

Corollary 3.2.2 *When B is the identity vector set $\{(1,0,0,\ldots,0),(0,1,0,\ldots,0),\ldots,(0,0,\ldots,1)\}$, the pseudo-inverse vector of any $b \in B$ is b itself. This is because $b.b = 1$ and $b'.b = 0$ when $b' \in B \setminus \{b\}$.*

3.2.3 Replacement Vector

Lemma 3.2.3 *Consider vectors x_i, y_i for $1 \leq i \leq s$, x_i and $y_i \in {\mathbb{F}_q}^{n-1}$ such that $x_i.y_i \neq 0$. Then, provided $|\mathbb{F}_q| > s$, there exists a vector $x = \alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_s x_s$ such that $x.y_i \neq 0$ for $1 \leq i \leq s$.*

The proof, which also immediately leads to an efficient algorithm for finding such an x, is presented in [7]. The total complexity of finding such an x is shown to be $\mathcal{O}(s^2 n)$.

3.2.3.1 Key Idea of the Algorithm

We initialize $C_t = \{g_1, g_2, \ldots, g_n\}$ for all sinks $t \in T$. The centralized algorithm then traverses the edges in the topological ordering (we have shown that such an ordering exists in section 2.1 of chapter 2), and assigns global encoding functions to each edge under the various failure scenarios (in this case, one failure pattern represents failure on a single edge). During real-time, the node identifies the global encoding functions of each incoming edge, and then chooses the outgoing global encoding vector. When the algorithm traverses a particular edge e_i, it handles the scenario that e_i is the failed edge, and sets global encoding vectors for all the edges of G. As and when we traverse an edge e, we update the sets B_t^k and C_t so as to maintain the following invariant.

Invariant $\mathcal{I}$ for the Algorithm:

The invariant we maintain for each family of $\mathcal{B}^i = \{B_t^i | t \in T\}$ is:

As the centralized algorithm traverses the edges in the topological ordering, any $n-1$ out of the n vectors in set B_t^i will be linearly independent before edge e_i is encountered, and there will exist a set of $n-1$ linearly independent vectors in B_t^i after e_i has been traversed. We keep updating the sets B_t^i during our traversal. If we obtain a network code such that this invariant is satisfied, we show that each sink $t \in T$ receives at least $n-1$ linearly independent equations in $x_1, x_2, \ldots, x_{n-1}$ thereby enabling it to solve for the variables and obtain these values deterministically.

3.2.3.2 Updating the pseudo-inverse Vectors

Given a set B, $b \in B$, let a be the pseudo-inverse of b with respect to B, and b' be such that $b'.a \neq 0$. We now describe a simple way of updating the pseudo-inverse vectors when b is replaced by b' in B. For this section alone, we denote by $a_{old}(c)$ and $a_{new}(c)$, the values of the pseudo-inverse vector of a vector c (with respect to B being implicitly assumed) before and after replacing b with b' in B. That is, $a_old(c)$ is the pseudo-inverse vector of c with respect to B, and $a_new(c)$ is the pseudo-inverse vector of c with respect to $B \setminus \{b\} \cup \{b'\}$.

1. $a_{new}(b') = (b'.a_{old}(b))^{-1} a_{old}(b)$.
2. $a_{new}(c) = a_{old}(c) - (b'.a_{old}(c)) a_{new}(b') \ \forall\ c \in B \setminus \{b\}$.

We now prove that $a_{new}(b')$ is the pseudo-inverse vector of b' with respect to $\{b'\} \cup B \setminus \{b\}$.To see this, notice that,

Proof:

$$\begin{aligned} b'.a_{new}(b') &= b'.((b'.a_{old}(b))^{-1} a_{old}(b)) & (3.5)\\ &= (b'.a_{old}(b))^{-1} (b'.a_{old}(b)) & (3.6)\\ &= 1 & (3.7)\end{aligned}$$

and,

$$\begin{aligned} c.a_{new}(b') &= c.((b'.a_{old}(b))^{-1}a_{old}(b)) \ \forall c \in B \setminus \{b\} && (3.8)\\ &= (b'.a_{old}(b))^{-1}(c.a_{old}(b)) && (3.9)\\ &= 0 \text{ as } c.a_{old}(b) = 0. && (3.10) \end{aligned}$$

Likewise, we prove that $a_{new}(c)$ is the pseudo-inverse vector of c with respect to $\{b'\} \cup B \setminus \{b\}$.

$$\begin{aligned} c.a_{new}(c) &= c.a_{old}(c) - (b'.a_{old}(c))(c.a_{new}(b')) && (3.11)\\ &= 1 - 0 && (3.12)\\ &= 1 && (3.13) \end{aligned}$$

and,

$$\begin{aligned} c'.a_{new}(c) &= c'.a_{old}(c) - (b'.a_{old}(c))(c'.a_{new}(b')) \ \forall c' \in B \setminus \{c\} \setminus \{b\} && (3.14)\\ &= 0 && (3.15) \end{aligned}$$

Thus, when we have a basis set B and a set of its pseudo-inverse vectors also, re-computing the pseudo-inverse vectors when a single element $b \in B$ is replaced by b' is done using the above explained procedure. This shall be utilized later in this section.

3.2.3.3 The $b - a$ Table

We now introduce the $b - a$ table which we maintain for each sink t. The table is partitioned into $|E|$ groups, each corresponding to the failure of an edge. Before the edge e_k is traversed, the k^{th} group of this table contains the values of the pseudo-inverse vectors (a) for $b \in B_t^k$ with respect to each subset of B_t^k of size $n-1$ containing b (such a set would form a basis for $\mathbb{F}_q{}^{n-1}$ due to the invariant maintained). For example, the a value of $(0, 0, \ldots, 1)$ with respect to the set $\{(1, 0, 0, \ldots, 0), (0, 1, 0, \ldots, 0), \ldots, (0, 0, \ldots, 1)\}$ would be $(0, 0, \ldots, 1)$ itself (see corollary 3.2.2). Once e_k is traversed, this group would contain the values of the pseudo-inverse vectors (a) for $b \in B_t^k$ with respect one particular linearly independent subset of B_t^k of size $n-1$ containing b. This is because, after e_k is traversed, in the event that the k^{th} edge had failed, there is a guarantee that only $n-1$ of the n flows to the sinks that e_k serves do not get corrupted.

Initializing the b-a table:

As we shall explain in algorithm *TolerateSingleEdge* given below, we will be starting with B_t^k =$\{(1,0,0,\ldots,0),(0,1,0,\ldots,0),\ldots,(0,0,\ldots,1),(1,1,\ldots,1)\}$ for all t and k. We initialize each group of the b-a table in the following fashion. First we add entries of the form $(\mathbf{i}_r, \mathbf{i}_r)$ where $\mathbf{i}_r$ is the r^{th} identity vector (it is present in B_t^k). This is true as a result of corollary 3.2.2. Then, we run the pseudo-inverse vector updating algorithm (section 3.2.3.2) assuming that the vector $(1,1,\ldots,1)$ replaces each identity vector one by one. We all all the resulting pairs of (b,a) to the table.

3.2.4 Centralized Algorithm for Fault Tolerant Network Coding

We now present the algorithm that efficiently identifies the network coding such that the above mentioned invariant $(\mathcal{I})$ is maintained.

Algorithm 3.1: *TolerateSingleEdge*

Input: *A network graph G with one source and several sinks such that the max-flow from the source to each sink is at least n.*

Output: *A linear network code for G which can tolerate failure on any edge of G, and provides a multicast throughput of n-1.*

1. The flow decomposition procedure (section 3.2.1.1) is executed to obtain the n edge disjoint paths from s to each sink t. The imaginary edge g_i is made to precede the i^{th} such path.
2. For all k, Set $B_t^k = \mathcal{B}$ where $\mathcal{B} = \{(1,0,0,\ldots,0),(0,1,0,\ldots,0),\ldots,(0,0,\ldots,1),(1,1,\ldots,1)\}$. Let each vector be the global encoding vector on each of the n imaginary edges leading to s. That is, for all $k = 1,2,\ldots,|E|$, we set *carried*$(g_r,k) = (0,0,\ldots,1,0,\ldots,0)$ - the r^{th} identity basis vector in $\mathbb{F}_q{}^{n-1}$ for $1 \le r < n$, and *carried*$(g_n,k) = (1,1,\ldots,1)$. We also initialize $C_t = \{g_1, g_2, \ldots, g_n\}$. The $b-a$ table is then populated following the procedure given in section 3.2.3.3.

```
3.  Traverse the edges in the topological order $e_1$ to $e_{|E|}$. It is to be noted that when traversing
    $e_i$, all incoming edges to $head(e_i)$ have already been traversed. Let $e_j$ be the edge that
    is currently traversed. Perform steps 4 to 21
4.      For $k = 0$ to $j - 1$
5.          Initialize the list $U = \phi$.
6.          For $t \in sinks(e_j)$
7.              Obtain from the $b - a$ table for sink $t$, all values of $(b_i, a_j)$ such that
                $b_i = carried(prev(e_j, t), k)$ and add the $(b_i, a_j)$ tuple to the list $U$.
8.          End For
9.          Remove all redundant entries from $U$.
10.         Using the replacement vector algorithm in 3.2.3, find a vector $b$ such that
            $b.a_j \neq 0$ for all $a_j$ such that $(b_i, a_j) \in U$.
11.         Set $carried(e_j, k) = b$.
12.         For $t \in sinks(e_j)$
13.             Mark $B_t^k$ for updating to $B_t^k \setminus \{carried(prev(e_j, t), k)\} \cup \{b\}$. Also run the
                procedure in section 3.2.3.2 to obtain the changes to be done to the $b - a$
                table. Mark these changes as well.
14.         End For
15.     End For
16.     Set $carried(e_j, j) = 0$.
17.     For $t \in sinks(e_j)$
18.         Set $B_t^k = B_t^k \setminus \{carried(prev(e_j, t), k)\} \cup \{0\}$.
19.         Update the $j^{th}$ group of the $b - a$ table such that only the $b, a$ values
            corresponding to the non-zero $b$ vectors in $B_t^k$ remain (there will be
            $n - 1$ such vectors).
20.     End For
```

```
21.        Update all the changes in the sets $C_t, B_t^k$ and the $b - a$ table that were marked for
           updating in step 13.
22. End Traversal
```

End of Algorithm 3.1

We now show that the above algorithm does indeed maintain the invariant $\mathcal{I}$ through till its completion. We also show that the messages transmitted on each edge are linear combinations of the incoming messages. That the throughput would be $n-1$ in all circumstances follows from the fact that when edge e_k fails, there exists a subset of size $n-1$ of the set B_t^k which remains linearly independent. The sinks can solve for the original packets from these linearly independent packets.

Proof: We first notice that the way we assign the global encoding vectors to the imaginary edges g_i in step 2 of the algorithm, invariant $\mathcal{I}$ is trivially satisfied (any $n-1$ out of n vectors in $\mathcal{B}$ are linearly independent). We also notice that the global encoding vector which the algorithm chooses for an edge is a function of the incoming packets' global encoding vectors alone and not the actual failed edge. This is very important as no knowledge about the failed edge is assumed for the working of the algorithm. This is so because, for two different sets of incoming global encoding vectors $\{b_1, b_2, \ldots, b_i\}$ and $\{b'_1, b'_2, \ldots, b'_i\}$, where i is the in-degree of the current node we are considering, the list U that is chosen would be the same if $b_j = b'_j$. Therefore, the outgoing global encoding vector would also be the same, regardless of which edge has failed. This is why the table(s) and the sets are updated only at the end of the complete iteration, and not then and there. Also, lemma 3.2.3 ensures that the outgoing vector is only a linear combination of the global encoding vectors along the incoming edges. The fact that the above procedure maintains the invariant $\mathcal{I}$ is because of step 10 in the algorithm. That is, the replacement vector algorithm in section 3.2.3 is given all possible (b, a) pairs so that whatever needs to be maintained linearly independent, is maintained so. Therefore, in the event that an edge e_k fails, the set of vectors that are incoming into sink t would contain among them, at least the set B_t^k. The sink t uses this set of linearly independent packets to solve for the original broadcast messages.

Complexity:

Steps 1 and 2 are executed only once, and their complexities are $\mathcal{O}(|V||E|^2)$ and $\mathcal{O}(n^2|V|)$ respectively. The main step (complexity-wise) in the iteration from steps 4 to 21 is finding the global encoding vector on an edge using the replacement algorithm (section 3.2.3) in step 10 of the algorithm. Its complexity (which is $\mathcal{O}(s^2n)$ as explained in section 3.2.3) here is $\mathcal{O}(n^5|E|^2)$, as $s = \mathcal{O}(n^2|E|)$ in the table. Therefore the overall complexity of the algorithm is $\mathcal{O}(n^5|E|^3)$, as the iteration runs $|E|$ times.

3.2.5 Polynomially Many Edge Failure Patterns

We now extend the above technique to one which can tolerate failure in one among polynomially many failure patterns. We represent the set of failure patterns by $\mathcal{FP}$ (see chapter 2). Let $\mathbf{p} \in \mathcal{FP}$ be a particular failure pattern. On deleting the edges that fail in $\mathbf{p}$, let the minimum of the *max-flows* from the source to the sinks be $k_{\mathbf{p}}$. Also let k be $min(k_{\mathbf{p}}|\mathbf{p} \in \mathcal{FP})$. In other words, k is a lower bound on the minimum of the *max-flows* from source to the sinks after deletion of any single failure pattern. The extension we present would guarantee a throughput of k. It can be separated into two phases - the initialization stage and the core stage. However, the initialization phase (described below) is only probabilistically correct, although it can be completed efficiently. Once the initialization is done, the centralized algorithm is deterministic and has a polynomial running time. As mentioned above, we assume throughout this section, that failure in any of the edge patterns would still maintain a flow of k to each of the sinks.

3.2.5.1 Phase 1: Initialization

We first create a set of n vectors in ${\mathbb{F}_q}^k$, such that any k of them are linearly independent. To do this, we randomly choose n vectors $r_1, r_2, \ldots, r_n$. We show that, if the field is appropriately large, any k of the n vectors are linearly independent with significantly high probability. This is why our initialization stage, as it currently stands, is a probabilistic procedure.

Lemma 3.2.4 *If $q > \lambda n(C^n_{k-1})$, the probability that any k out of n randomly chosen vectors from ${\mathbb{F}_q}^k$ are linearly independent is at least $1 - 1/\lambda$, where $\lambda > 1$.*

Proof: Let B_i be the (good) event that the required property is satisfied - that is any k out of $r_1, r_2, \ldots, r_i$ are linearly independent, when r_i is chosen. We need to bound $P(B_n)$.

$$P(B_1) \;=\; 1 - 1/q^k \tag{3.16}$$

This is because the first vector we choose would result in a good event as long as we don't choose the zero vector.

$$P(B_2) \;=\; (1 - 1/q^k)(1 - q/q^k) \tag{3.17}$$

This is so, because when we choose r_2, a good event would arise iff B_1 had happened, and r_2 is not a linear combination of r_1. This argument can be iteratively progressed till choosing r_k. When choosing r_k, a good event would arise iff B_{k-1} had happened, and r_k is not a linear combination of $r_1, r_2, \ldots, r_{k-1}$. Since there are q^{k-1} such linear combinations, the probability that r_k is not dependent on them is $1 - q^{k-1}/q^k$. Therefore,

$$P(B_k) \;=\; (1 - 1/q^k)(1 - q/q^k)(1 - q^2/q^k)\ldots(1 - q^{k-1}/q^k) \tag{3.18}$$

Now, after choosing the first k elements, a good event would happen when choosing r_{k+1} if it is not linearly dependent on any $k-1$ out of the first k chosen elements. Likewise, B_{k+2} happens if B_{k+1} happens and r_{k+2} is linearly independent of any subset of $k-1$ elements out of the first $k+1$ elements - $r_1, r_2, \ldots, r_{k+1}$. Similar arguments hold for the following inequalities as well.

$$P(B_{k+1}) \;\geq\; P(B_k)((1 - (C^k_{k-1})q^{k-1}/q^k)) \tag{3.19}$$

$$P(B_{k+2}) \;\geq\; P(B_{k+1})((1 - (C^{k+1}_{k-1})q^{k-1}/q^k)) \tag{3.20}$$

and so on till n,

$$P(B_n) \;\geq\; P(B_{n-1})((1 - (C^n_{k-1})q^{k-1}/q^k)) \tag{3.21}$$

Therefore,

$$P(B_n) \;\geq\; P(B_k)((1 - (C^n_{k-1})/q))^{n-k} \tag{3.22}$$

$$P(B_n) \;\geq\; (1 - 1/q)^k(1 - (C^n_{k-1})/q)^{n-k} \tag{3.23}$$

Finally,

$$P(B_n) \;\geq\; (1 - (C^n_{k-1})/q)^n \tag{3.24}$$

Also, since $q > n(C^n_{k-1})$, we have

$$(1 - (C^n_{k-1})/q)^n \quad \geq \quad (1 - (C^n_{k-1})n/q) \tag{3.25}$$

as, on expanding the binomial terms, every pair of the remaining terms is positive. Therefore, the probability $P(B_n)$ is at least $1 - 1/\lambda$ when $q > \lambda n(C^n_{k-1})$.
Hence the result.

3.2.5.2 Phase 2:

Since k is a lower bound on the *max-flows* to the sinks under failure of any pattern $\mathbf{p} \in \mathcal{FP}$, for each failure pattern, there would be at least k flow paths which have no faulty edges on them that enter each sink. We note down this set of k flows for each failure pattern. There would only polynomially many such sets (as we provide a scheme that only tolerates polynomially many edge failure patterns). We then run the modified centralized algorithm *TolerateSingleEdge* with the invariant that at any stage, these *noted sets* remain linearly independent. This invariant is similar in spirit to the one (any $n-1$ out of the n vectors are linearly independent) maintained in the centralized algorithm for tolerating a single edge failure. Therefore, at the termination of the algorithm, under any failure pattern, the k flow of the corresponding *noted set* would carry a linearly independent set of packets. This is used to solve for the original packets.

3.2.6 Noisy Edges

In this section, we again extend the original algorithm to handle edges where failures are modeled as white noise being transmitted on the edges instead of $\mathbf{0}$. Incorporating tolerance to white noise is quite simple. One easy approach would be adding redundancy at certain positions in the packet (more specifically, a specific pattern that can help distinguish "valid" packets from those that have been corrupted by noise). In our system, we introduce added redundancy in the following fashion. We have the transmitted packet as an $n+2$ tuple $(m, b_1, b_1, b_2, \ldots, b_n)$ instead of an $n+1$ tuple $(m, b_1, b_2, \ldots, b_n)$. Note that the coefficient b_1 has been repeated once. We could detect white noise on edges with probability $1/q$ by performing the simple check that the 2^{nd} and 3^{rd} elements of the received packet are the same. This also causes only negligible increase in the overhead on the message packet size, as the b_is are from a significantly smaller finite field when compared to m. Once we identify the noise-corrupted packets, we then treat those edges as *failed* (in the old sense), and perform the algorithm given in the previous section assuming a $\mathbf{0}$ for such noise-detected edges. This protocol works with significant probability ($1 - 1/q$ even when there is only one redundant field element). If we want a scheme that works with greater probability, all we do is add more redundant information (like repeating b_1 twice).

3.3 Summary

In this chapter, we looked at the problem of finding fault tolerant network codes. We first proved that there is a gap between the maximum possible throughput rates achievable in the network coding case and the non network coding case. We then obtained a centralized deterministic algorithm which finds an optimal throughput network coding scheme that can tolerate failure on up to one edge. This was then extended to one that can tolerate failure in one among a set of polynomially many *failure patterns*. Finally, we looked at the simple extension which can detect and tolerate white noise.

CHAPTER 4

Some Optimization Problems in Network Coding

In this chapter, we introduce some combinatorial problems that are of importance to network coding. One such problem is *min-cost* network coding. We also introduce the problem of *min-latency* network coding, and highlight its similarities and differences from the *min-cost* variant.

4.1 Min-Cost Network Coding

Problem Statement:
Given a graph $\mathbf{G}$ *with each edge having unit capacity and a cost, a source* $\mathbf{s}$ *and some sinks, identify the edges along which the* $\mathbf{h}$ *flow from the source to each sink must be routed, so as to minimize the total cost of all the edges used. We are given that the* ***max-flow*** *from the source to each of the sinks is at least* $\mathbf{h}$. *Each edge can carry a flow to several sinks in spite of the fact that it has unit capacity, since we are working in the network coding domain where flows can be combined.*

Lun et al. in [15] prove that the linear programming formulation given in figure 4.1 indeed solves the problem of finding the minimum cost network coding solution. In their formulation, the notations used are slightly different from the ones used in this work. They consider nodes to be represented by $i \in \mathcal{N}$, and edges $(i, j) \in A$. Their version also considers each edge (i, j) to have a capacity of c_{ij}. However, that does not alter the problem significantly. They assume that the flow requirement from the source to each sink is R. Once we identify the flows from s to each sink (not bothering about whether or not they are disjoint), we can then run the centralized algorithm mentioned in [7] with input as the identified flows. we can therefore separate the

$$
\begin{array}{lll}
\text{minimize} & \sum_{(i,j)\in A} a_{ij} z_{ij} & \\
\text{subject to} & z_{ij} \geq x_{ij}^{(t)}, & \forall\ (i,j) \in A,\ t \in T, \\
 & \sum_{\{j|(i,j)\in A\}} x_{ij}^{(t)} - \sum_{\{j|(j,i)\in A\}} x_{ji}^{(t)} = \begin{cases} R & \text{if } i = s, \\ -R & \text{if } i = t, \\ 0 & \text{otherwise}, \end{cases} & \forall\ i \in N,\ t \in T, \\
 & c_{ij} \geq x_{ij}^{(t)} \geq 0, & \forall\ (i,j) \in A,\ t \in T.
\end{array}
$$

Figure 4.1: LP Formulation for Min-Cost Network Coding[15]

whole problem into two parts - identifying the flows and then running Jaggi's algorithm given in [7]. Identifying the *min-cost* flows is precisely what this linear program solves.

Thus we observe that the above solution immediately gives us an efficient method of finding the *min-cost* network codes for a given network. We now shift our focus on the *min-latency* network coding problem and also present a few of its variants.

4.2 Min-Latency Network Coding

As we have mentioned in earlier chapters, the basic principle of network coding is that any outgoing packet is a linear combination of all incoming packets. This automatically means that for sending out a packet on a link, the node would have to wait till all its incoming packets arrive. This is the crucial difference between optimization problems in the network coding model and those in the non network coding setting. Thus, if each edge (link) on the network has a *delay* or a *latency*, a problem worth focusing on would be to minimize the time by which all the sinks have received enough information to obtain the original packets. This is what the *min-latency* network coding problem is.

Problem Statement:

Given a graph **G** *with each edge having unit capacity and a latency, a source* **s** *and some sinks, identify the edges along which the* **h** *flow from the source to each sink must be routed, so as to* *minimize the time by which all the sinks receive their respective flows. We are given that the* ***max-flow*** *from the source to each of the sinks is at least* **h***. Each edge can carry a flow to several sinks in spite of the fact that it has unit capacity, since we are working in the network coding domain where flows can be combined. Also, before sending flows along outgoing links, each node must wait for all its incoming flows.*

It is interesting to note that this problem becomes quite standard in the non network coding case. It reduces to one where each final time is the maximum path length of flows reaching that sink. This is because the intermediate nodes do not have to wait for all the incoming edges. This problem is known to be strongly NP-Hard ([16]). Which is why its status in the network coding setting assumes significance. We believe it is hard in the network coding setting too, but do not have a reduction as yet. We now look at a few variations of the latency problem, with the same model but slightly different objectives.

4.2.1 Variants of the Latency Problem

Variant 1:

Given a graph **G** *with each edge having unit capacity and a latency, a source* **s** *and some sinks, identify the edges along which the* **h** *flow from the source to each sink must be routed, so that the time difference between the sink that first receives all its flows and the sink that last receives all its flows is bounded by* **T** *and the time at which the protocol completes is bounded by* **C***. We are given that the* ***max-flow*** *from the source to each of the sinks is at least* **h***. Each edge can carry a flow to several sinks in spite of the fact that it has unit capacity, since we are working in the network coding domain where flows can be combined. Also, before sending flows along outgoing links, each node must wait for all its incoming flows.*

Variant 2:

Given a graph **G** *with each edge having unit capacity and a latency, a source* **s** *and some sinks, identify the edges along which the* **h** *flow from the source to each sink must be routed, so that the total buffering cost at all the nodes is bounded by* **T** *and the time at which the protocol completes is bounded by* **C***. Buffering cost for a node is the sum of the times for which the*

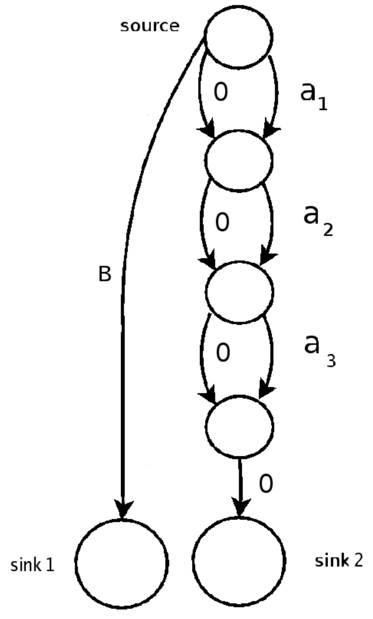

Figure 4.2: Basic Structure of Reduction

incoming packets are buffered at the node. We are given that the ***max-flow*** *from the source to each of the sinks is at least* **h**. *Each edge can carry a flow to several sinks in spite of the fact that it has unit capacity, since we are working in the network coding domain where flows can be combined. Also, before sending flows along outgoing links, each node must wait for all its incoming flows.*

Both these problems are NP-Hard and a similar reduction from the subset sum problem works. This is diagrammatically represented in figure 4.2.

4.2.1.1 The Subset Sum Problem

Problem Statement:

Given a positive integer bound **B** *and a set of* **n** *positive integers, find a subset such that the difference between the sum of the integers in the subset and* **B** *is bounded above by* **C**, *and the sum of the integers in the subset is not more that* **B**.

We now present the hardness reduction proof from the *subset sum* problem to *variant 1* of the *min-latency* problem. The reduction for the second variant too is almost identical.

Proof: The *subset sum* problem is NP-Hard (see [16]). Let the n given integers be $a_1, a_2, \ldots, a_n$. We construct a graph like the one shown in figure 4.2, having one source and two sinks. We also have n intermediate nodes (one for each a_i) placed in between the source and sink 2. On the left side is a single directed edge of latency B connecting the source and sink 1. On the right hand side, the source, the intermediate nodes, and sink 2 are connected in series. Each of these intermediate nodes has two incoming edges, with delays 0 and a_i respectively. Sink 2 has an incoming edge from the last of the intermediate nodes, with latency 0. All edges have unit capacity. It is easy to note that the minimum of the *max-flow* from the source to a sink is only 1. We therefore ask for a *unit flow* to each sink that tries to minimize the difference in latency (at the sink) between the individual flows. It now becomes evident that there is a flow of latency difference at most C with the both the sinks receiving their flow by a time B *iff* there is a subset, with the sum of its elements not exceeding B but less than B by at most C. For the forward proof, given such a subset, we can create a flow that passes through the edges corresponding to the chosen elements in the subset, and flows through the 0 edges going out of the other intermediate nodes. The flow to sink 1 passes through the edge with latency C. For the reverse proof, our subset comprises of those integers that the right side flow chose to pass through (note that it has the option of either choosing the 0 delay edge or the edge with latency a_i). Hence the reduction.
The reduction for the second variant is along similar lines.

CHAPTER 5

Conclusion

5.1 In this work

In this work, we have primarily focused on the problem of *Fault Tolerant Network Coding*, where the objective is to find network coding schemes that provide optimal throughput while being able to tolerate *failure* on some edges. The kind of tolerance we considered is intermediate between a completely *fool-proof* system and one which assumes faultless networks. That is, while we allow for certain edges to fail, we assume that when they fail, they transmit a null message and not some arbitrary message. The *fool-proof* code construction problem would ask of the coding scheme to tolerate *Byzantine* adversaries on the links. The throughput we were able to obtain is also understandably in between that which can be achieved in the faultless case and the *fool-proof* case.

We have provided an algorithm which is in essence an extension of the centralized scheme for polynomial time multicast code construction in [7]. Our scheme too is a centralized(it requires the whole graph as input) efficient algorithm that generates linear network codes which can tolerate failures in one among polynomially many failure patterns. It is also rate optimal in that the throughput it guarantees can be tightly matched by the maximum possible throughput in certain classes of examples. As an example, for the single edge failure case, our scheme designs codes whose multicast throughput is $n-1$(where n is the minimum of the *max-flows* from the source to the sinks). Our scheme can also be extended very easily and efficiently(with negligible overhead) to one that tolerates some edges that are corrupted by white noise by adding some deterministic redundancy to every packet - a property that a perfectly random packet will not exhibit in most cases.

We have finally discussed about some optimization problems related to network coding.

These problems are separated from the actual network code generation phase due to the nature of the objective functions. We have first explained the *min-cost* network coding problem, and then introduced the *min-latency* network coding problem. We have proved the NP-Hardness of variants of the *min-latency* problem, and also highlighted the differences between the network coding version and the non network coding version of these..

5.2 Future Directions

An interesting open problem that would make the fault tolerant network coding scheme described in this work more complete is to come up with an algorithm that can tolerate failure in any k edges. This is not addressed in this work as the total number of failure patterns in this case becomes exponential in the size of the graph. Another open problem is to come up with a deterministic algorithm that can tolerate byzantine adversaries on some edges. Further, the throughput achieved by our scheme is only tightly matched by a theoretical upper bound in certain classes of networks. Therefore, finding a scheme that is always tight would also pose an interesting challenge. With respect to the *min-latency* problem, the status of the decision problem itself remains open, though it is in all likelihood, NP-Complete. Also, getting efficient approximation algorithms for the latency variants would have a lot of practical application. Finally, not much research work has been done for the case where there are multiple sources in the graph. The multi-source version is harder to solve for most of the problems in network coding.

References

[**1**] K. Jain, M. Mahdian, and M.R. Salavatipour. Packing steiner trees. In *14th ACM-SIAM Symposium on Discrete Algorithms(SODA)*, pages 266–274, 2003.

[**2**] R.W. Yeung, S-Y.R. Li, N. Cai, and Z. Zhang. *Network Coding Theory.* Now Publishers, June 2006.

[**3**] R. Ahlswede, N. Cai, S-Y.R. Li, and R.W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.

[**4**] N. Cai and R. W. Yeung. Network error correction, part i: basic concepts and upper bounds. *Communications in Information and Systems*, 6(1):19–36, 2006.

[**5**] N. Cai and R. W. Yeung. Network error correction, part ii: Lower bounds. *Communications in Information and Systems*, 6(1):37–54, 2006.

[**6**] S-Y.R. Li, N. Cai, and R.W. Yeung. Linear network coding. *IEEE Transactions on Information Theory*, 49(2):371–381, 2003.

[**7**] S. Jaggi, P. Sanders, P.A. Chou, M. Effros, S. Egner, K. Jain, and L. Tolhuizen. Polynomial time algorithms for multicast network code construction. *IEEE Transactions on Information Theory*, 51(6):1973–1982, 2005.

[**8**] P. Sanders, S. Egner, and L. Tolhuizen. Polynomial time algorithms for network information flow. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 286–294, New York, NY, USA, 2003. ACM Press.

[**9**] R. Koetter and M. Medard. Beyond routing: An algebraic approach to network coding. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies(INFOCOM)*, pages 122–130, 2002.

[**10**] T.C. Ho, R. Koetter, M. Medard, D. R. Karger, and M. Effros. Benefits of coding over routing in a randomized setting. In *Proceedings of the 2003 IEEE International Symposium*

on Information Theory(ISIT), page 422, 2003.

[**11**] P.A. Chou, Y. Wu, and K. Jain. Practical network coding. In *Proceedings of 41st Annual Allerton Conference on Communication, Control and Computing*, 2003.

[**12**] S. Jaggi, M. Langberg, S. Katti, T. Ho, D. Katabi, and M. Medard. Resilient network coding in the presence of byzantine adversaries. Accepted to the 26th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), 2007.

[**13**] K. Bhattad and K.R. Narayanan. Weakly secure network coding. NetCod 2005 - Workshop on Network Coding, Theory, and Applications, 2005.

[**14**] L.R. Ford Jr. and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[**15**] D. Lun, M. Medard, T. Ho, and R. Koetter. Network coding with a cost criterion. MIT LIDS TECHNICAL REPORT P-2584, 2004.

[**16**] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Freeman Press, 1979.