Jaypee University of Information Technology
Waknaghat, Distt. Solan (H.P.)

Learning Resource Center

CLASS NUM: BOOK NUM.:

ACCESSION NO.: SP09092 / SP0913092

This book was issued is overdue due on the date stamped below. if the book is kept over due, a fine will be charged as per the library rules.

Due Date	Due Date	Due Date
LRC CHECKED 12 NOV 2014		

ASYNCHRONOUS DATA DISTRIBUTION USING LUBY TRANSFORM CODES

Project Report submitted in partial fulfilment of the requirement for the degree of

Bachelor of Technology

in

Electronics and Communication Engineering

Under the Supervision of

Dr. M. Usman

By

Kshitij Bathla (091122)

Abhishek Sati (091034)

Vikas Chauhan (091101)

to

Learning Resource Center JUIT Waknaghat HP

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled "**Asynchronous Data Distribution Using Luby Transform Codes**", submitted by **Kshitij Bathla, Abhishek Sati and Vikas Chauhan** in partial fulfillment for the award of degree of Bachelor of Technology in Electronics and Communication Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

[illegible]

Date: 31-05-2013

Dr. M. Usman

Assistant Professor

Department Of Electronics and communication

Acknowledgement

We take this opportunity to express our profound gratitude and deep regards to our guide Dr. M. Usman for his exemplary guidance, monitoring and constant encouragement throughout the course of this thesis. The blessing, help and guidance given by him time to time shall carry us a long way in the journey of life on which we are about to embark.

We will also take this opportunity to express a deep sense of gratitude to Prof. Dr. T.S. Lamba, Dr. Bhasker Gupta, Mr. Viranjay Mohan Shrivastava, Dr. Rajiv Kumar, Mr. Akhil Ranjan, Mr. Solomon Telluri, Ms. Shruti Jain, Mr. Bhupendra Kumar, and Mr. Vikas Hastir for making our minds work a step ahead every time we thought about implementing new ideas to complete the project.

We are obliged to the staff members of Jaypee University of Information Technology, for the valuable information provided by them in their respective fields. We are grateful for their cooperation during the period of our assignment.

Lastly, we thank almighty, our parents, brother, sisters and friends for their constant encouragement without which this assignment would not be possible.

Date: 30th May, 2013

[illegible] Kshitij Bathla

[illegible] Abhishek Sati

[illegible] Vikas Chauhan

TABLE OF CONTENTS

LIST OF FIGURES

Abstract

To reduce time complexity in data transfer which was a big drawback of conventional one to one method of data downloading, a new method was proposed which can help rapid data transfer without any retransmissions and zero redundancy. The Fountain codes help us in achieving the above mentioned goal. We have implemented a class of fountain codes known as the Luby Transform Codes. Fountain codes provide an efficient way to transfer information over erasure channels like Internet, where files are transmitted in multiple small packets, each of which is either received without error or not received at all.

Certain networks, such as ones used for cellular wireless broadcasting, do not have a feedback channel. Applications on these networks still require reliability. Fountain codes in general, and LT codes in particular, get around this problem by adopting an essentially one-way communication protocol.

Luby transform codes (LT codes) are the first class of practical fountain codes that are near-optimal erasure correcting codes. LT codes are *rateless* because the encoding algorithm can in principle produce an infinite number of message packets (i.e., the percentage of packets that must be received to decode the message can be arbitrarily small). LT codes are the first codes fully realizing the digital fountain concept. The key to make LT codes work well is the degree distribution used in the encoding procedure, which is sampled to determine the degree of each encoding symbol. These codes use a very simple encoding and decoding process with the help of Soliton Distribution thereby providing precise data to the downloader.

Our aim is to download a file from multiple sources at the same time so as to receive the entire file at a very fast speed. Since these codes have been implemented for wireless environments there would be noise in the transmission channel. Additive White Gaussian Noise and Rayleigh Fading are some examples of noise that may occur in our transmission channel. These have also been taken into consideration while data is being sent. Since no retransmission requests are made in fountain codes, a lot of bandwidth is saved. They are *erasure correcting codes* because they can be used to transmit digital data reliably on an erasure channel.

CHAPTER 1

INTRODUCTION

Reliable transmission of data over a communication channel has been a topic of much research. As the information passes through the communication channel it gets distorted by the effects of the channel. Reliability is achieved by either retransmitting the erroneous information or using appropriate coding techniques. Two major techniques that are used to deal with such problems are as mentioned below:-

1.1 Automatic Repeat Request (ARQ)

ARQ is one of the methods to deal with loss of information. There must be two channels, one forward and one backward between the sender and the receiver. The forward channel is used to transmit the information to the receiver. The backward channel is used by the receiver to send positive or negative acknowledgements to the sender. If any information symbols are lost during transmission or are received with error, the receiver sends a retransmission request to the sender and the symbol is retransmitted. A timer is also used by the sender so that if for some reason, the receiver fails to send any acknowledgement, then the sender automatically retransmits the symbol [6].

1.2 Forward Error Correction (FEC)

It is a technique used for controlling errors over unreliable or noisy communication channels. The main principle used is that the sender encodes the information in a redundant way using an error correcting code. The redundancy allows the receiver to detect a limited number of errors that may occur anywhere in the message, and often to correct these errors without retransmission. FEC gives the receiver the ability to correct errors without needing a reverse channel to request retransmission of data, but at the cost of a fixed, higher forward channel bandwidth. The code rate for any error correcting code depends on the channel characteristics. So, if there are multiple transmitters then implementing error correcting codes for each and every channel becomes a challenge [6].

1.3 Disadvantages of TCP in wireless networks

The Internet provides a platform for rapid and timely information exchange among a disparate array of clients and servers. Transmission Control Protocol (TCP) and Internet Protocol (IP) are separately designed and closely tied protocols that define the rules of communication between end hosts, and are the most commonly used protocol suite for data transfer in the Internet.

The combination of TCP/IP dominates today's communication in various networks from the wired backbone to the heterogeneous network due to its remarkable simplicity and reliability. TCP has become the de facto standard used in most applications ranging from interactive sessions such as Telnet and HTTP, to bulk data transfer like FTP. TCP was originally designed primarily for wired networks.

In a wired network, random bit error rate, a characteristic usually more pronounced in the wireless network, is negligible, and congestion is the main cause of packet loss. The emerging wireless applications, especially high-speed multimedia services and the advent of wireless IP communications carried by the Internet, call for calibration and sophisticated enhancement or modifications of this protocol suite for improved performance.

TCP is a layer four transport protocol that uses the basic IP services to provide applications with an end-to-end and connection oriented packet transport mechanism that ensures the reliable and ordered delivery of data. Each TCP packet is associated with a sequence number, and only successfully received in-order packets are acknowledged to the sender by the receiver, by sending corresponding packets (acknowledgments, ACKs) with sequence numbers of the next expected packets.

On the other hand, packet loss or reception of out of order packets indicates failures. To eradicate such failures, TCP implements flow control and congestion control algorithms. Unlike the optical fibre backbone and copper wired access networks, wireless links use the open air as the transmission medium and are subject to many uncontrollable quality-affecting factors such as weather conditions, urban obstacles, multipath interferences, large moving objects, and mobility of wireless end devices.

As a result, wireless links exhibit much higher BERs than wired links. Also, limitations of radio coverage and user mobility require frequent handoffs, thus resulting in temporal disconnections and reconnections between the communicating end hosts during a communication session. Note that a short disconnection event can actually stall the TCP transmission for a much longer period. This effect is demonstrated in Fig. 1, where the evolution of the congestion window reflects TCP transmission throughput, and the time axis is in units of RTTs. This result is taken from a single TCP connection between two hosts, of which the link capacity is 2 Mb/s and the RTT is 20 ms. During the disconnection, both data packets and ACKs are dropped, and each retransmission attempt leads to an unsuccessful retransmission.

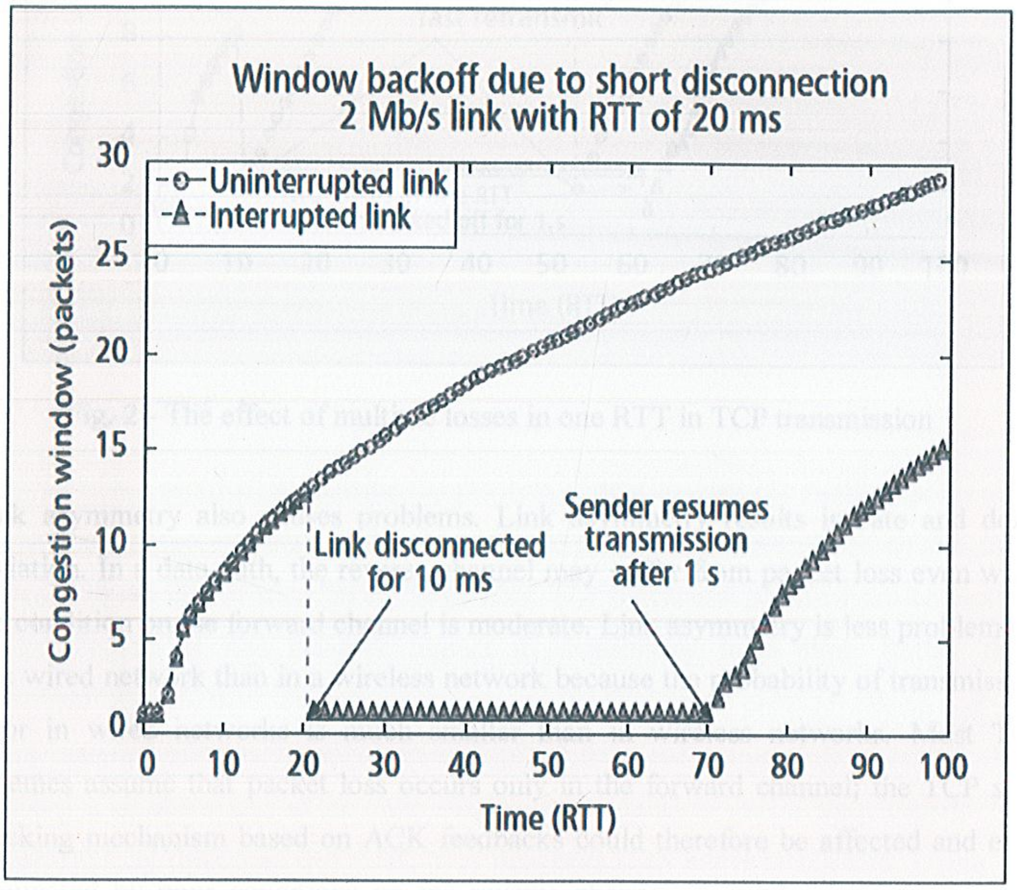


Fig. 1 - The effect of a short disconnection in TCP transmission

Noises and other factors in the wireless environment usually cause random bit errors to occur in short bursts, thus leading to a higher probability of multiple random packet losses within one RTT. Again, multiple unsuccessful retransmissions within one RTT would cause the retransmission timer to exponentially back off. Therefore, for instance, two random packet losses within an RTT would cause the TCP sender to stall its transmission for a period of about 1 s. This phenomenon is illustrated in Fig. 2

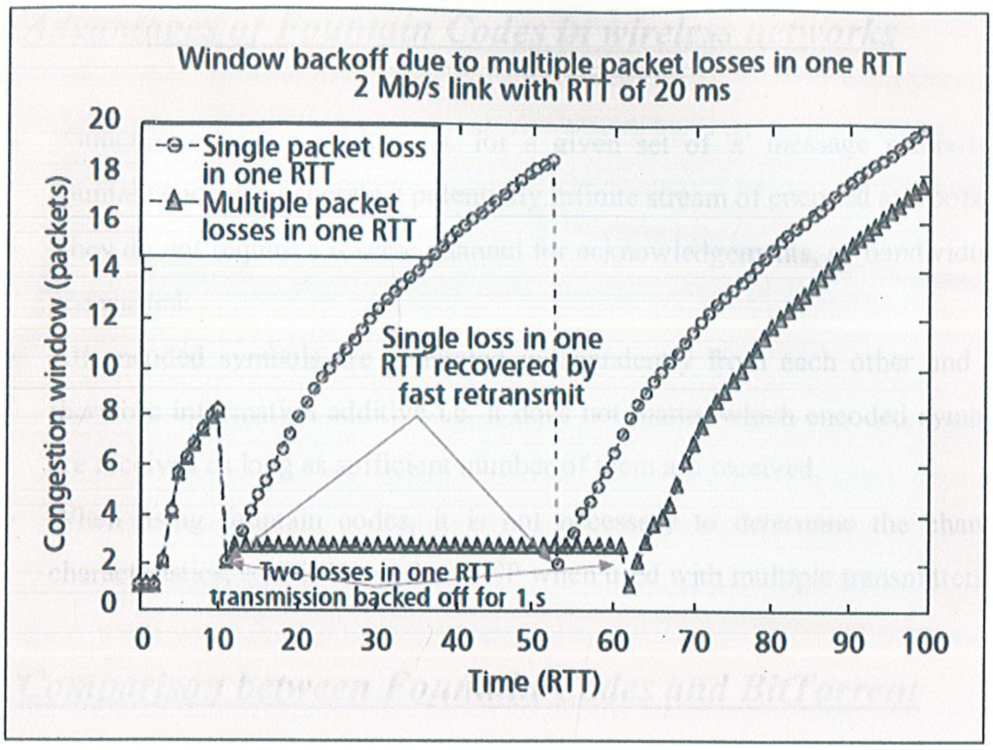


Fig. 2 - The effect of multiple losses in one RTT in TCP transmission

Link asymmetry also causes problems. Link asymmetry results in rate and delay variation. In a data path, the reverse channel may suffer from packet loss even when the condition on the forward channel is moderate. Link asymmetry is less problematic in a wired network than in a wireless network because the probability of transmission error in wired networks is much smaller than in wireless networks. Most TCP schemes assume that packet loss occurs only in the forward channel; the TCP self-clocking mechanism based on ACK feedbacks could therefore be affected and even dominated by poor conditions on the reverse channel. A sender could mistakenly interpret a bad reverse channel condition as congestion on the forward channel and unnecessarily reduce the sending rate, thus resulting in TCP performance degradation [1].

1.4 Advantages of Fountain Codes in wireless networks

- Fountain codes are rateless i.e. for a given set of 'k' message symbols; a fountain code can generate a potentially infinite stream of encoded symbols.
- They do not require a reverse channel for acknowledgements, so bandwidth is not wasted.
- All encoded symbols are generated independently from each other and are therefore information additive i.e. it does not matter which encoded symbols are received as long as sufficient number of them are received.
- When using fountain codes, it is not necessary to determine the channel characteristics, so it is better than TCP when used with multiple transmitters.

1.5 Comparison between Fountain Codes and BitTorrent

1.5.1 BitTorrent

BitTorrent is a protocol supporting the practice of peer-to-peer file sharing that is used to distribute large amounts of data over the Internet. BitTorrent is one of the most common protocols for transferring large files. The BitTorrent protocol can be used to reduce the server and network impact of distributing large files. Rather than downloading a file from a single source server, the BitTorrent protocol allows users to join a "swarm" of hosts to download and upload from each other simultaneously. The protocol is an alternative to the older single source, multiple mirror sources technique for distributing data, and can work over networks with lower bandwidth. Using the BitTorrent protocol, several basic computers, such as home computers, can replace large servers while efficiently distributing files to many recipients. This lower bandwidth usage also helps prevent large spikes in internet traffic in a given area, keeping internet speeds higher for all users in general, regardless of whether or not they use the BitTorrent protocol.

A user who wants to upload a file first creates a small torrent descriptor file that they distribute by conventional means (web, email, etc.). They then make the file itself available through a BitTorrent node acting as a seed. Those with the torrent descriptor file can give it to their own BitTorrent nodes which, acting as peers or leechers, download it by connecting to the seed and/or other peers. The file being distributed is divided into segments called pieces. As each peer receives a new piece of the file it becomes a source (of that piece) for other peers, relieving the original seed from having to send that piece to every computer or user wishing a copy. With BitTorrent, the task of distributing the file is shared by those who want it; it is entirely possible for the seed to send only a single copy of the file itself and eventually distribute to an unlimited number of peers.

Each piece is protected by a cryptographic hash contained in the torrent descriptor. This ensures that any modification of the piece can be reliably detected, and thus prevents both accidental and malicious modifications of any of the pieces received at other nodes. If a node starts with an authentic copy of the torrent descriptor, it can verify the authenticity of the entire file it receives.

Pieces are typically downloaded non-sequentially and are rearranged into the correct order by the BitTorrent Client, which monitors which pieces it needs, and which pieces it has and can upload to other peers. Pieces are of the same size throughout a single download (for example a 10 MB file may be transmitted as ten 1 MB Pieces or as forty 256 KB Pieces). Due to the nature of this approach, the download of any file can be halted at any time and be resumed at a later date, without the loss of previously downloaded information, which in turn makes BitTorrent particularly useful in the transfer of larger files. This also enables the client to seek out readily available pieces and download them immediately, rather than halting the download and waiting for the next (and possibly unavailable) piece in line, which typically reduces the overall length of the download.

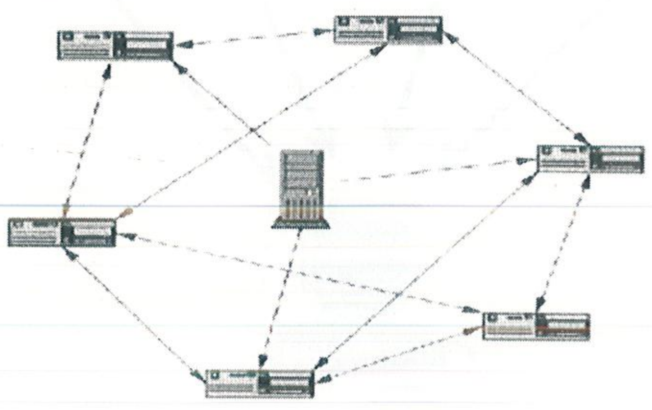


Fig. 3 BitTorrent Structure

When a peer completely downloads a file, it becomes an additional seed. This eventual shift from peers to seeders determines the overall "health" of the file (as determined by the number of times a file is available in its complete form). The distributed nature of BitTorrent can lead to a flood like spreading of a file throughout many peer computer nodes. As more peers join the swarm, the likelihood of a complete successful download by any particular node increases. Relative to traditional Internet distribution schemes, this permits a significant reduction in the original distributor's hardware and bandwidth resource costs. In general, BitTorrent's non-contiguous download methods have prevented it from supporting progressive download or "streaming playback".

1.6 Asynchronous Data Transfer

In the project scenario, there are multiple transmitters, all carrying the same information, and one receiver which is connected to all the transmitters. The transmitters are asynchronous in nature i.e. there is no coordination between them.

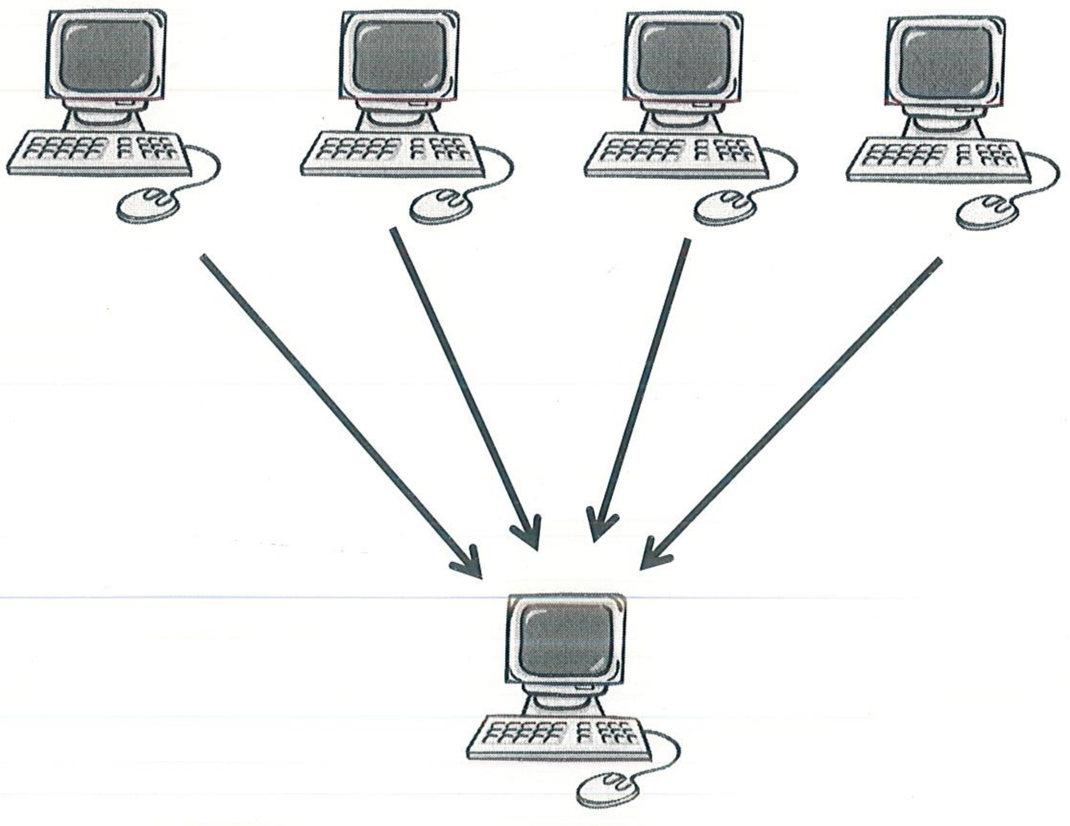


Fig. 4 Asynchronous Data Transfer

The channel characteristics of each transmitter-receiver pair are different. But it is not necessary to know the channel characteristics because of the nature of Fountain Codes. Each transmitter sends the information at a different rate to the receiver in the form of encoded symbols, and when the receiver gets enough encoded symbols to successfully decode the original information, it stops receiving the symbols. The fountain codes are information additive, meaning that it does not matter which encoded symbols are received as long as sufficient number of them are received. The encoded symbols are independent of each other.

Even if the receiver gets disconnected from any of the transmitters, it does not mean that the original information cannot be obtained. At least one transmitter should be present for successful decoding. The number of transmitters involved is directly proportional to the speed of data transfer.

Unlike BitTorrent protocol, Asynchronous Data Transfer is one sided, i.e. the receiver does not upload the same data back to the transmitters, simultaneously.

CHAPTER 2

FOUNTAIN CODES

Consider a case where a large file is disseminated to a large number of people who may want to access it at different times and have transmission links of different quality. Current networks use unicast-based protocols such as the transport control protocol (TCP), which requires a transmitter to continually send the same packet until acknowledged by the receiver.

While unicast protocols successfully use receiver initiated requests for retransmission of lost data to provide reliability, it is widely known that implementing it for multicasts is uncontrollable. For example, consider a scenario where software has to be disseminated over the Internet to a large number of users. As clients lose packets, their retransmission requests can overburden the server in a phenomenon known as feedback implosion. Even in the event that the server can handle the requests, the retransmitted packets are often of use only to a small subset of clients. Thus, a conclusion can be drawn that not only are adaptive retransmission based solutions for multicasts and broadcasts unscalable and inefficient in wired networks, they are also unworkable on wireless networks where the back channel has high latency and limited or no capacity, if it is available at all [2].

Forward error correction (FEC) based on erasure channels (discussed later) in one approach that to tackle this multicast related problem. The basic principle behind the use of erasure codes is that the original source data, in the form of a sequence of k packets, along with additional redundant packets, are transmitted by the sender, and the redundant data can be used to recover lost source data at the receivers. A receiver can reconstruct the original source data once it receives a sufficiently large subset of the packets. The main benefit of this approach is that different receivers can recover from different lost packets using the same redundant data. In principle, this idea can greatly reduce the number of retransmissions, as a single transmission of redundant data can potentially benefit many receivers simultaneously.

The work of Nonnenmacher, Biersack and Towsley defines a hybrid approach to reliable multicast, coupling requests for retransmission with transmission of redundant code words, and quantifies the benefits of this approach in practice. Their work, and the work of many other authors, focuses on erasure codes based on Reed-Solomon codes. The limitation of these codes is that encoding and decoding times are slow, effectively limiting k to small values for practical applications. Hence, their solution involves breaking the source data into smaller blocks of packets and encoding over these blocks. Receivers which fail to receive a packet from a given block request retransmission of an additional code word from that block. They demonstrate that this approach is effective for dramatically reducing the number of retransmissions when packet loss rates are low (they typically consider 1% loss rates).

It can easily be seen that this architecture does not scale well when many users access a server concurrently and is extremely inefficient when the information transmitted is always the same. In effect, TCP and other unicast protocols place strong importance on the ordering of packets to simplify coding at the expense of increased traffic. This approach also does not enable receivers to join the session dynamically.

To eliminate the need for retransmission and to allow receivers to access data asynchronously, the use of a data carousel or broadcast disk approach can ensure full reliability. In a data carousel approach, the source repeatedly loops through transmission of all data packets. Receivers may join the stream at any time, and then listen until they receive all distinct packets comprising the transmission. Clearly, the reception overhead at a receiver, measured in terms of unnecessary receptions, can be extremely high using this approach. Adding redundant code words to the carousel can dramatically reduce reception overhead. Using Reed-Solomon codes, a fixed amount of redundancy is added to blocks of the transmission. The source then repeatedly loops through the set of blocks, transmitting one data or redundant packet about each block in turn until all packets are exhausted, and then repeats the process. This interleaved approach enables the receiver to reconstruct the source data once it receives sufficiently many packets from each block. The limitation of using this approach over lossy networks is that the receiver may still receive many unnecessary transmissions, especially while waiting for the last packets from the last few blocks it needs to reconstruct [2].

We will be discussing an alternate approach, where packets are not ordered and the recovery of some subset of packets will allow for successful decoding. This class of codes, called fountain codes, was pioneered by a start-up called Digital Fountain and has greatly influenced the design of codes for binary erasure channels (BECs), a well-established model for the Internet.

A digital fountain is conceptually simpler, more efficient, and applicable to a broader class of networks than previous approaches. A digital fountain injects a stream of distinct encoding packets into the network, from which a receiver can reconstruct the source data. The key property of a digital fountain is that the source data can efficiently be reconstructed intact from any subset of the encoding packets equal in total length to the source data. Our approach is to construct better approximations of a digital fountain than currently exist as a basis for protocols that perform reliable distribution of bulk data.

Fountain codes, also known as rate-less erasure codes, are a class of erasure codes with the property that a potentially limitless sequence of encoding symbols can be generated from a given set of source symbols such that the original source symbols can ideally be recovered from any subset of the encoding symbols of size equal to or only slightly larger than the number of source symbols. The term *fountain* or *rate-less* refers to the fact that these codes do not exhibit a fixed code rate. A fountain code is optimal if the original k source symbols can be recovered from any k encoding symbols. Fountain codes are known to have efficient encoding and decoding algorithms and that allow the recovery of the original k source symbols from any n of the encoding symbols with high probability, where n is just slightly larger than k.

An erasure code is a forward error correction (FEC) code for the binary erasure channel, which transforms a message of k symbols into a longer message (code word) with n symbols such that the original message can be recovered from a subset of the n symbols. The fraction $r=k/n$ is called the code rate which in the case of fountain codes, is variable.

2.1 The random linear fountain

Consider the following encoder for a file of size K packets $s_1, s_2, \ldots, s_K$. A 'packet' here is the elementary unit that is either transmitted intact or erased by the erasure channel. We will assume that a packet is composed of a whole number of bits. At each clock cycle, labelled by n, the encoder generates K random bits $\{G_{nk}\}$, and the transmitted packet t_n is set to the bitwise sum, modulo 2, of the source packets for which G_{nk} is 1.

$$t_n = \sum_{k=1}^{K} s_k G_{kn} \qquad (1)$$

This sum can be done by successively exclusive-or-ing the packets together. You can think of each set of K random bits as defining a new column in an ever growing binary generator matrix, as shown at the top of Fig. 5.

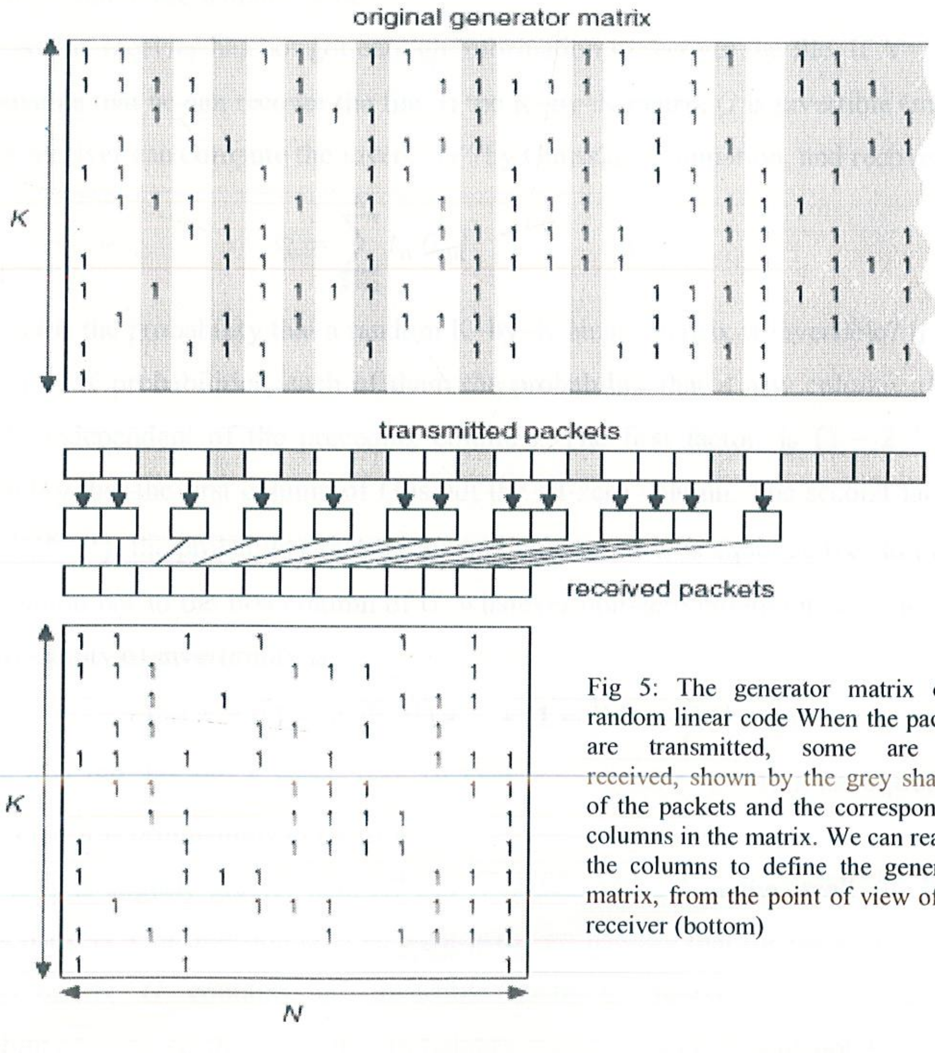


Fig 5: The generator matrix of a random linear code When the packets are transmitted, some are not received, shown by the grey shading of the packets and the corresponding columns in the matrix. We can realign the columns to define the generator matrix, from the point of view of the receiver (bottom)

Now, the channel erases a bunch of the packets; a receiver, holding out his bucket, collects *N* packets. What is the chance that the receiver will be able to recover the entire source file without error? Let us assume that he knows the fragment of the generator matrix *G* associated with his packets, for example, maybe *G* was generated by a deterministic random-number generator, and the receiver has an identical generator that is synchronised to the encoder's. Alternatively, the sender could pick a random key, k_n, given which the *K* bits $\{G_{nk}\}_{k=1}^{K}$ are determined by a pseudo-random process, and send that key in the header of the packet. As long as the packet size *l* is much bigger than the key size (which need only be 32 bits or so), this key introduces only a small overhead cost. In some applications, every packet will already have a header for other purposes, which the fountain code can use as its key. For brevity, let's call the K–by–N matrix fragment '*G*' from now on.

Now, as we were saying, what is the chance that the receiver will be able to recover the entire source file without error?

If *N*<*K*, the receiver has not got enough information to recover the file. If *N*=*K*, it is conceivable that he can recover the file. If the K–by–K matrix *G* is invertible (modulo 2), the receiver can compute the inverse G^{-1} by Gaussian elimination, and recover

$$s_k = \sum_{k=1}^{K} t_n \, G_{nk}^{-1} \qquad (2)$$

So, what is the probability that a random K–by–K binary matrix is invertible? It is the product of *K* probabilities, each of them the probability that a new column of *G* is linearly independent of the preceding columns. The first factor, is $(1-2^{-K})$, the probability that the first column of *G* is not the all-zero column. The second factor is $\left(1-2^{-(K-1)}\right)$, the probability that the second column of *G* is equal neither to the all-zero column nor to the first column of *G*, whatever non-zero column it was. Iterating, the probability of invertibility is

$$(1-2^{-K})\left(1-2^{-(K-1)}\right) * \dots * (1-\tfrac{1}{8})(1-\tfrac{1}{4})(1-\tfrac{1}{2}),$$

Which is 0.289, for any *K* larger than 10.That is not great (we would have preferred 0.999!) but it is promisingly close to 1.

What if *N* is slightly greater than *K*? Let *N*=*K*+*E*, where *E* is the small number of excess packets. Our question now is, what is the probability that the random K–by–N binary matrix *G* contains an invertible K–by–K matrix? Let us call this probability $1-\delta$, so that δ is the probability that the receiver will not be able to decode the file when E excess packets have been received. This failure probability δ is plotted against E for the case $K=100$ in Fig. 6 (it looks identical for all $K>10$). For any K, the probability of failure is bounded above by

$$\delta(E) \leq 2^{-E}$$

This bound is shown by the thin dotted line in Fig. 6.

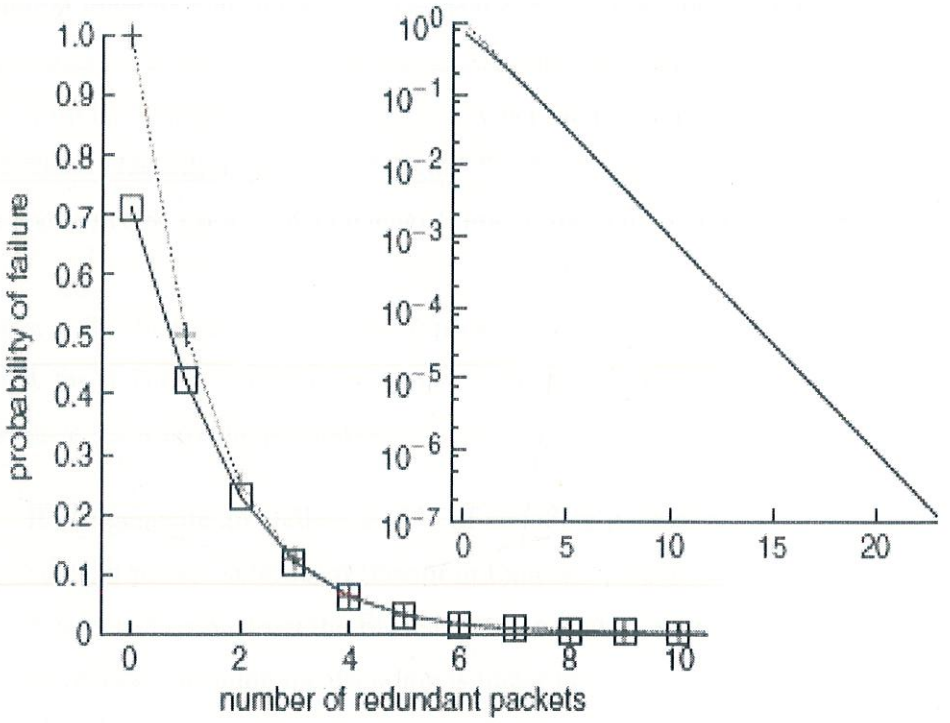


Fig 6: Performance of the random linear fountain

In summary, the number of packets required to have probability $(1-\delta)$ of success is $\simeq K + log_2\frac{1}{\delta}$. The expected encoding cost per packet is $K/2$ packet operations, since on average half of the packets must be added up (a packet operation is the exclusive-or of two packets of size l bits). The expected decoding cost is the sum of the cost of the matrix inversion, which is about K^3 binary operations, and the cost of applying the inverse to the received packets, which is about $K^2/2$ packet operations. While a random code is not in the technical sense a 'perfect' code for the erasure channel (it has only a chance of 0.289 of recovering the file when K packets have arrived), it is almost perfect. An excess of E packets increases the probability of success to at least $(1 - \delta)$, where $\delta = 2^{-E}$. Thus, as the file size K increases, random linear fountain codes can get arbitrarily close to the Shannon limit. The only bad news is that their encoding and decoding costs are quadratic and cubic in the number of packets encoded. This scaling is not important if K is small (less than one thousand, say); but we would prefer a solution with lower computational cost [3].

2.2 ***Digital Fountain Ideal***

The digital fountain was devised as the ideal protocol for transmission of a single file to many users who may have different access times and channel fidelity. The name is drawn from an analogy to water fountains, where many can fill their cups with water at any time. The output packets of digital fountains must be universal like drops of water and hence be useful independent of time or the state of a user's channel.

Consider a file that can be split into k packets or information symbols and must be encoded for a Binary Error Correction. A digital fountain that transmits this file should have the following properties:

1. It can generate an endless supply of encoding packets with constant encoding cost per packet in terms of time or arithmetic operations.
2. A user can reconstruct the file using any k packets with constant decoding cost per packet, meaning the decoding is linear in k.
3. The space needed to store any data during encoding and decoding is linear in k.

These properties show digital fountains are as reliable and efficient as TCP systems, but also universal and tolerant, properties desired in networks.

Reed-Solomon (RS) codes are the first example of fountain-like codes because a message of k symbols can be recovered from any k encoding symbols. However, RS codes require quadratic decoding time and are limited to a smaller block length n.

Low-density parity-check (LDPC) codes reduce the decoding complexity by use of the sum-product algorithm and iterative decoding techniques. They come closer to the fountain code ideal. However early LDPC codes are restricted to fixed-degree regular graphs due to which significantly more than k encoding symbols are needed to successfully decode the transmitted signal [4].

2.3 Binary Erasure Channel

The erasure channel is a memory less channel where symbols are either transmitted perfectly or erased. Hence, the output alphabet is simply the input alphabet and the erasure symbol '?' For an erasure probability p, the conditional probability of the channel is

$$p(y|x) = \begin{Bmatrix} 1-p, & y = x; \\ p & y = ?; \\ 0 & otherwise \end{Bmatrix}$$

As mentioned, this is a commonly-accepted model for packet transmission on the Internet. A binary erasure channel (BEC) corresponds to when the input can only take values 0 and 1. In this case, the channel capacity is well-known to be $C=1-P$ [4].

Channels with erasures are of great importance. For example, files sent over the internet are chopped into packets, and each packet is either received without error or not received. Noisy channels to which good error-correcting codes have been applied also behave like erasure channels: much of the time, the error-correcting code performs perfectly; occasionally, the decoder fails, and reports that it has failed, so the receiver knows the whole packet has been lost. A simple channel model describing this situation is a q-ary erasure channel (Fig. 7), which has (for all inputs in the input alphabet $(0, 1, 2, \ldots\ldots, q-1)$ a probability 1-f of transmitting the input without error, and probability f of delivering the output '?'. The alphabet size q is 2^l, where l is the number of bits in a packet.

Common methods for communicating over such channels employ a feedback channel from receiver to sender that is used to control the retransmission of erased packets. For example, the receiver might send back messages that identify the missing packets, which are then retransmitted. Alternatively, the receiver might send back messages that acknowledge each received packet; the sender keeps track of which packets have been acknowledged and retransmits the others until all packets have been acknowledged.

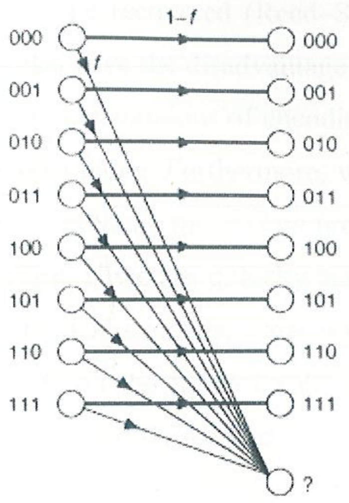


Fig 7: An erasure channel – the 8-ary erasure channel The eight possible inputs {0, 1, 2, . . . , 7} are here shown by the binary packets 000, 001, 010, . . . ,111

These simple retransmission protocols have the advantage that they will work regardless of the erasure probability *f*, but purists who have learned their Shannon theory will feel that these protocols are wasteful. If the erasure probability f is large, the number of feedback messages sent by the first protocol will be large. Under the second protocol, it is likely that the receiver will end up receiving multiple redundant copies of some packets, and heavy use is made of the feedback channel. According to Shannon, there is no need for the feedback channel: the capacity of the forward channel is *(1-f)l* bits, whether or not we have feedback. Reliable communication should be possible at this rate, with the help of an appropriate forward error correcting code.

The wastefulness of the simple retransmission protocols is especially evident in the case of a broadcast channel with erasures; channels where one sender broadcasts to many receivers, and each receiver receives a random fraction *(1-f)* of the packets. If every packet that is missed by one or more receivers has to be retransmitted, those retransmissions will be terribly redundant. Every receiver will have already received most of the retransmitted packets.

So, we would like to make erasure-correcting codes that require no feedback or almost no feedback. The classic block codes for erasure correction are called Reed–Solomon codes. An (*N, K*) Reed–Solomon code (over an alphabet of size $q=2^l$) has the ideal property that if any *K* of the *N* transmitted symbols are received then the original K source symbols can be recovered (Reed–Solomon codes exist for $N<q$). However, Reed–Solomon codes have the disadvantage that they are practical only for small K, N, and q: standard implementations of encoding and decoding have a cost of order $K(N-K)\ log_2 N$ packet operations. Furthermore, with a Reed–Solomon code, as with any block code, one must estimate the erasure probability f and choose the code rate $R=K/N$ before transmission. If we are unlucky and f is larger than expected and the receiver receives fewer than K symbols, what are we to do? We would like a simple way to extend the code on the fly to create a lower-rate (N', K) code. For Reed–Solomon codes, no such on-the-fly method exists.

2.4 *Types of fountain codes*

- LT Codes
- Raptor Codes
- Online Codes

2.5 *Tornado Codes*

We begin our study of fountain codes by studying Tornado codes. These codes, were the first steps towards achieving the digital fountain ideal described earlier. Tornado codes are block codes and hence not rateless. However, they do approach Shannon capacity with linear decoding complexity.

We consider a system model in which a single transmitter performs bulk data transfer to a larger number of users on an erasure channel. Our objective is to achieve complete file transfer with the minimum number of encoding symbols and low decoding complexity. For k information symbols, RS codes can achieve this with $k\ logk$ encoding and quadratic decoding times. The reason for the longer decoding time is that in RS codes, every redundant symbol depends on all information symbols. By contrast, every redundant symbol depends only on a small number of information symbols in Tornado codes. Thus they achieve linear encoding and decoding complexity, with the cost that the user requires slightly more than k packets to successfully decode the transmitted symbols. Moreover, Tornado codes can achieve a rate just below the capacity $1-p$ of the BEC. Thus they are capacity-approaching codes.

Tornado codes are closely related to Gallager's LDPC codes, where codes are based on sparse bipartite graphs with a fixed degree d_λ for left nodes (information symbols) and ρ for right nodes (encoding symbols). Unlike regular LDPC codes, Tornado codes use a cascade of irregular bipartite graphs. The main contribution is the design and analysis of optimal degree distributions for the bipartite graph such that the receiver is able to recover all missing bits by a simple erasure decoding algorithm. The innovation of Tornado code has also inspired work on irregular LDPC codes [4].

CHAPTER 3
LUBY TRANSFORM CODES

3.1 Introduction

LT codes are the first realization of a class of erasure codes that we call universal erasure codes. The symbol length for the codes can be arbitrary, from one-bit binary symbols to general *l*-bit symbols. We analyse the run time of the encoder and decoder in terms of symbol operations, where a symbol operation is either an exclusive-or of one symbol into another or a copy of one symbol to another. If the original data consists of k input symbols then each encoding symbol can be generated, independently of all other encoding symbols, on average by $O(\ln(k/\delta))$ symbol operations, and the k original input symbols can be recovered from an $k + O(\sqrt{k} * ln^2(k/\delta))$ of the encoding symbols with probability $1-\delta$ by on average $O(\mathrm{k} * \ln(k/\delta))$ symbol operations.

LT codes are rateless, i.e., the number of encoding symbols that can be generated from the data is potentially limitless. Furthermore, encoding symbols can be generated on the fly, as few or as many as needed. Also, the decoder can recover an exact copy of the data from any set of the generated encoding symbols that in aggregate are only slightly longer in length than the data. Thus, no matter what the loss model is on the erasure channel, encoding symbols can be generated as needed and sent over the erasure channel until a sufficient number have arrived at the decoder in order to recover the data. Since the decoder can recover the data from nearly the minimal number of encoding symbols possible, this implies that LT codes are near optimal with respect to any erasure channel. Furthermore, the encoding and decoding times are asymptotically very efficient as a function of the data length. Thus, LT codes are universal in the sense that they are simultaneously near optimal for every erasure channel and they are very efficient as the data length grows.

The key to the design and analysis of the LT codes is the introduction and analysis of the LT process. This process and its analysis is a novel generalization of the classical process and analysis of throwing balls randomly into bins and it may be of independent interest. We provide a full analysis of the behavior of the LT process using first principles of probability theory, which precisely captures the behavior of the data recovery process.

3.2 Comparison to traditional erasure codes

Traditional erasure codes are typically block codes with a fixed rate, i.e., k input symbols are used to generate $n - k$ redundant symbols for a total of n encoding symbols, and the rate of the code is k/n. For example, research in networking has suggested using implementations of both Reed-Solomon and Tornado codes for reliable data distribution applications, and in these cases both k and n either are limited to fairly small values due to practical considerations or are fixed before the encoding process begins. Ideally, as is the case with a Reed- Solomon code, any k of the n encoding symbols is sufficient to recover the original k input symbols. Sometimes, as is the case with a Tornado code, slightly more than k of the n encoding symbols are needed to recover the original k input symbols.

Reed-Solomon codes in practice are only efficient for relative small settings of k and n. The reason is that for a standard implementation of Reed-Solomon codes, $k(n - k)A/2$ symbol operations are needed to produce the n encoding symbols, where A is the size of the finite field used. Decoding times are similar. Although there are theoretical algorithms for implementing Reed-Solomon codes that are asymptotically faster for both encoding and decoding, i.e., k times polylogarithmic in n, the quadratic time implementations are faster in practice for values of k and n of interest.

Tornado codes are block erasure codes that have linear in n encoding and decoding times. LT codes are somewhat similar to Tornado codes. For example, they use a similar rule to recover the data, and the degree distributions used for Tornado codes is superficially similar to the degree distributions used for LT codes. However, the actual degree distribution used for Tornado codes turns out to be inapplicable to LT codes. For Tornado codes the degree distribution on input symbols is similar to the Soliton distribution (described later) and the degree distribution on the first layer of redundant symbols is close to the Poisson distribution. The Soliton distribution on input symbols is inapplicable for LT codes, as this distribution cannot be the resulting degree distribution on input symbols when encoding symbols are generated independently, no matter what the distribution on neighbors of encoding symbols is chosen to be. One could imagine flipping the distributions on Tornado codes so that, like the LT codes, the Poisson distribution is on the input symbols and the Soliton distribution is on the redundant symbols. However, the distribution induced on the redundant symbols by the missing input symbols in the Tornado code graph is far from the Soliton distribution, and applying the recovery rule to the induced distributions clearly won't lead to a reasonable reception overhead. Beyond the differences between LT codes and Tornado codes mentioned previously, LT codes have significant application advantages over Tornado codes. Let $c = n/k$ be the constant stretch factor of the Tornado code design. Once k and n have been fixed the Tornado encoder produces n encoding symbols, and cannot produce further encoding symbols on the fly if the demand arises. In contrast, the encoder can generate as few or as many encoding symbols as needed on demand.

Tornado codes use a cascading sequence of bipartite graphs between several layers of symbols, where the input symbols are at the first layer and redundant symbols are at each subsequent layer. In practice, this requires either prior construction of the exact same graph structure at both the encoder and decoder, or prior construction of the graph structure at the encoder that is then communicated to the decoder. This pre-processing is quite cumbersome in either case, and in particular the graph structure size is proportional to $n = ck$. Although the amortized overhead is lessened if the same length data is to be encoded and decoded repeatedly, a different graph structure is required for each data length and thus the pre-processing is quite a large drawback if each data is of a different length. In contrast, the degree distributions used by LT codes are easily computed based on the data length, and this is the only pre-processing needed prior to invoking either the encoder or the decoder. For the Tornado codes the decoding depends on having a sufficient number of distinct encoding symbols from the n encoding symbols. In some transport applications, the total number of encoding symbols transmitted from a sender is several times the number of encoding symbols than can be reasonably generated by the Tornado encoder. For these applications, in some cases a good policy is to randomly select an encoding symbol each time a symbol is to be sent. In this case, it is not hard to see that the average number of encoding symbols that a receiver needs to receive to obtain k distinct encoding symbols among the $n=ck$ encoding symbols is $ck\ ln(c/(c - 1))$.

(Recall that at least k distinct encoding symbols are required to recover the data). Thus, for example, if $c = 2$ then the client needs to receive $1.386k$ encoding symbols on average in order to receive k distinct encoding symbols. To make this more acceptable, e.g., reception of $1.05k$ encoding symbols, requires $c > 10$. These values for c makes the Tornado codes unattractive for these types of applications because the encoder and decoder memory usage and the decoding time is proportional to c times the data length. In contrast, for LT codes the encoder and decoder memory usage is proportional to the data length and the decoding time depends only on the data length, independent of how many encoding symbols are generated and sent by the encoder.
Both Reed-Solomon and Tornado codes are systematic codes, i.e. all the input symbols that constitute the data are directly included among the encoding symbols, whereas LT codes are not systematic.

3.3 LT Codes Design

The length l of the encoding symbols can be chosen as desired. The overall encoding and decoding is more efficient in practice for larger values of l because of overheads with bookkeeping operations, but the value of l has no bearing on the theory. In transport applications l is sometimes chosen to be close to the length of the packet payload. The encoder works as follows. The data of length N is partitioned into $k = N/l$ input symbols, i.e., each input symbol is of length l. Encoding symbols are then generated. Given an ensemble of encoding symbols and some representation of their associated degrees and sets of neighbors, the decoder repeatedly recovers input symbols using the following rule as long as it applies.
If there is at least one encoding symbol that has exactly one neighbor then the neighbor can be recovered immediately since it is a copy of the encoding symbol. The value of the recovered input symbol is exclusive-ored into any remaining encoding symbols that also have that input symbol as a neighbor, the recovered input symbol is removed as a neighbor from each of these encoding symbols and the degree of each such encoding symbol is decreased by one to reflect this removal.

3.4 The LT process

We introduce the LT process to help describe the design and analysis of a good degree distribution for LT codes. Let K be the total number of encoding symbols to be considered in the analysis (typically K is slightly larger than k). The LT process is a novel generalization of the classical process of throwing balls randomly into bins. A well-known analysis of the classical process shows that $K = k * ln(k/\delta)$ balls are necessary on average to ensure that each of the k bins is covered by at least one ball with probability at least $1 - \delta$. In the analysis of the LT process, encoding symbols are analogous to balls and input symbols are analogous to bins, and the process succeeds if at the end all input symbols are covered.

All input symbols are initially uncovered. At the first step all encoding symbols with one neighbor are released to cover their unique neighbor. The set of covered input symbols that have not yet been processed is called the ripple, and thus at this point all covered input symbols are in the ripple. At each subsequent step one input symbol in the ripple is processed: it is removed as a neighbor from all encoding symbols which have it as a neighbor and all such encoding symbols that subsequently have exactly one remaining neighbor are released to cover their remaining neighbor. Some of these neighbors may have been previously uncovered, causing the ripple to grow, while others of these neighbors may have already been in the ripple, causing no growth in the ripple. The process ends when the ripple is empty at the end of some step. The process fails if there is at least one uncovered input symbol at the end. The process succeeds if all input symbols are covered by the end.

In the classical balls and bins process, all the balls are thrown at once. Because of the high probability of collisions, i.e., multiple balls covering the same bin, many more balls must be thrown to cover all bins than there are bins. In contrast, the proper design of the degree distribution ensures that the LT process releases encoding symbols incrementally to cover the input symbols. Initially only a small fraction of the encoding symbols are of degree one and thus they cover only a small fraction of the input symbols. Covered input symbols are processed one at a time, and each one processed releases potentially other encoding symbols to randomly cover unprocessed input symbols. The goal of the degree distribution design is to slowly release encoding symbols as the process evolves to keep the ripple small so as to prevent redundant coverage of input symbols in the ripple by multiple encoding symbols, but at the same time to release the encoding symbols fast enough so that the ripple does not disappear before the process ends.

Because the neighbors of an encoding symbol are chosen at random independently of all other encoding symbols, it is easy to verify that the step at which a particular encoding symbol is released is independent of all the other encoding symbols. Furthermore, once an encoding symbol is released it covers a uniformly chosen input symbol among the unprocessed input symbols, independent of all other encoding symbols. These properties make the LT process especially amenable to the design of good degree distributions and analysis of the process with respect to these distributions.

It is not hard to verify that there is a one to one correspondence between the LT process and the decoder, i.e., an encoding symbol covers an input symbol if and only if the encoding symbol can recover that input symbol. Thus, the LT process succeeds if and only if the decoder successfully recovers all symbols of the data. The total number of encoding symbols needed to cover all the input symbols corresponds exactly to the total number of encoding symbols needed to recover the data. The sum of the degrees of these encoding symbols corresponds exactly to the total number of symbol operations needed to recover the data [5].

3.4.1 The Encoder

The process of generating an encoding symbol is conceptually very easy to describe:

- Randomly choose the degree *d* of the encoding symbol from a degree distribution (discussed later). The design and analysis of a good degree distribution is a primary focus of the remainder of this paper.
- Choose uniformly at random *d* distinct input symbols as neighbors of the encoding symbol.
- The value of the encoding symbol is the exclusive-or of the *d* neighbors.

When using the encoding symbols to recover the original input symbols of the data, the decoder needs to know the degree and set of neighbors of each encoding symbol. There are many different ways of communicating this information to the decoder, depending on the application. For example, the degree and a list of neighbor indices may be given explicitly to the decoder for each encoding symbol. As another example, the degree and neighboring indices of each encoding symbol may be computed by the decoder implicitly based for example on the timing of the reception of the encoding symbol or the position of the encoding symbol relative to the positions of other encoding symbols. As another example, a key may be associated with each encoding symbol and then both the encoder and decoder apply the same function to the key to produce the degree and set of neighbors of the encoding symbol. In this case, the encoder may randomly choose each key it uses to generate an encoding symbol and keys may be passed to the decoder along with the encoding symbols. Each key may instead be produced for example by a deterministic process, e.g., each key may be one larger than the previous key. The encoder and decoder may have access to the same set of random bits, and the key may be used as the seed to a pseudo-random generator that uses these random bits to produce the degree and the neighbors of the encoding symbol.

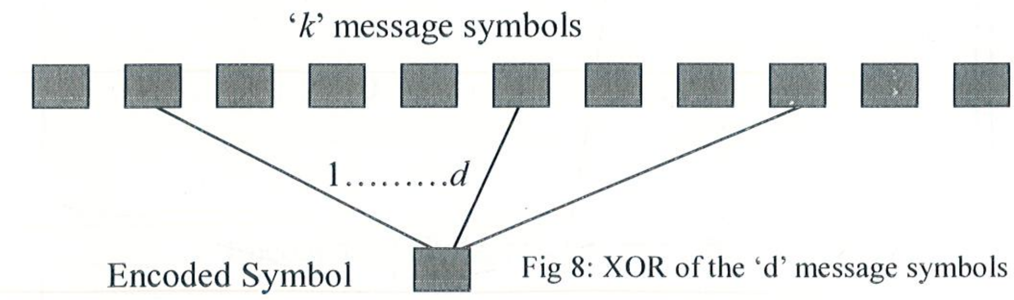
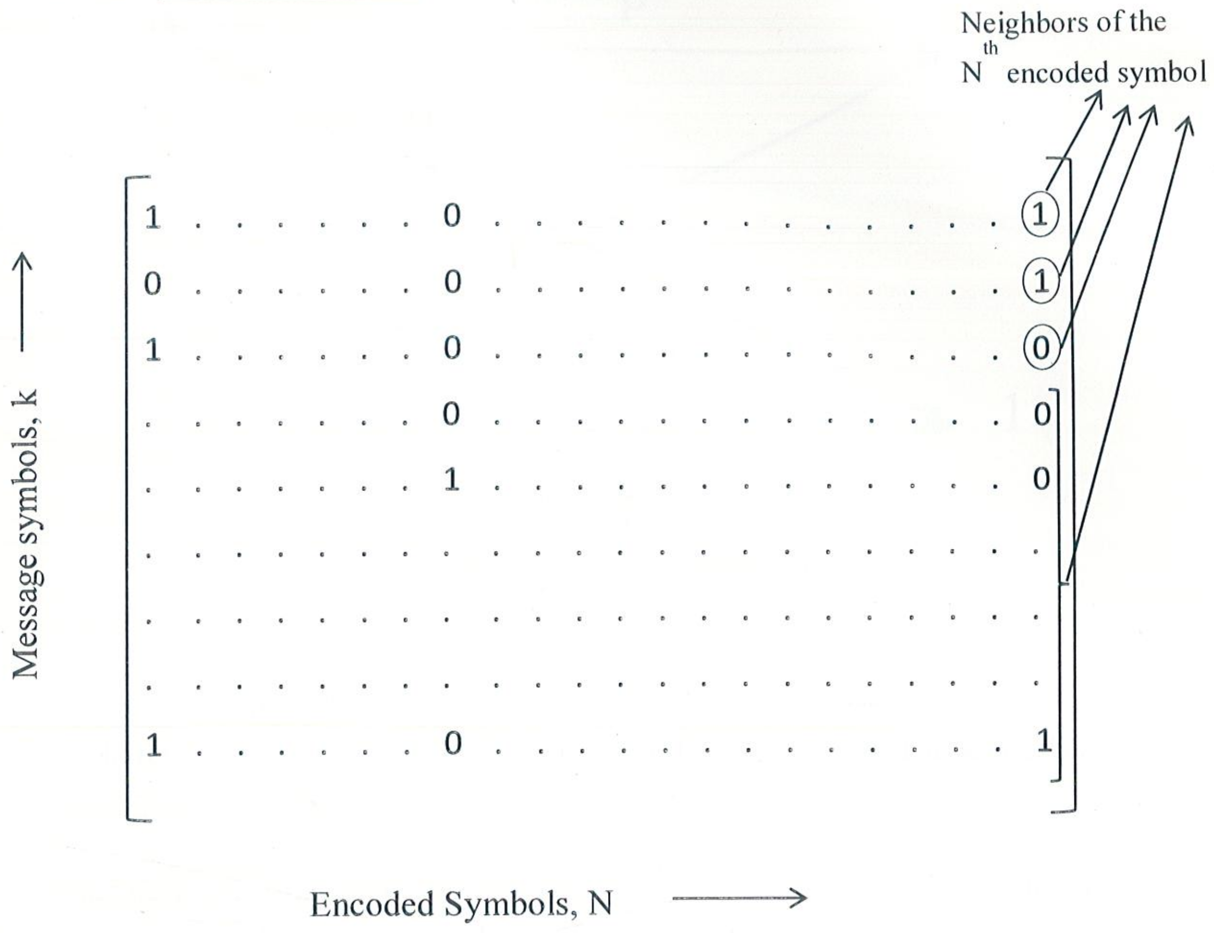


Fig 8: XOR of the 'd' message symbols

3.4.2 Sending the Connectivity Pattern (Generator Matrix)

The decoder needs to know the degree of each packet that is received, and which source packets it is connected to in the graph. This information can be communicated in various ways. For example, if the sender and receiver have synchronized clocks, they could use identical pseudo-random number generators, seeded by the clock, to choose each random degree and each set of connections. Alternatively, the sender could pick a random key, κ_n, given which the degree and the connections are determined by a pseudo-random process, and send that key in the header of the packet. As long as the packet size l is much bigger than the key size (which need only be 32 bits or so), this key introduces only a small overhead cost.

3.4.2.1 *Generator Matrix*



3.4.3 *CHANNEL*

This is the communication channel over which the encoded symbols are transmitted. In this project the Binary Symmetric channel and AWGN channels have been considered.

3.4.3.1 *NOISE*

Binary Symmetric Channel

A binary symmetric channel (or BSC) is a common communications channel model used in coding theory and information theory. In this model, a transmitter wishes to send a bit (a zero or a one), and the receiver receives a bit. It is assumed that the bit is usually transmitted correctly, but that it will be "flipped" with a small probability (the "crossover probability"). This channel is used frequently in information theory because it is one of the simplest channels to analyze.

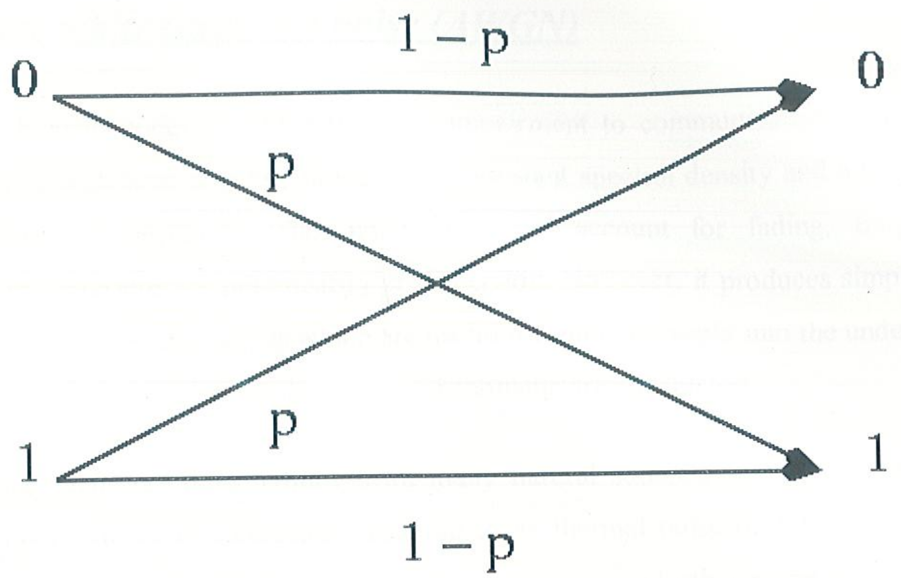


Fig. 9 Condition for binary symmetric channel

Binary Symmetric Channel is a binary channel; that is, it can transmit only one of two symbols (usually called 0 and 1). The transmission is not perfect, and occasionally the receiver gets the wrong bit.

This channel is often used by theorists because it is one of the simplest noisy channels to analyze. Many problems in communication theory can be reduced to a BSC. Conversely, being able to transmit effectively over the BSC can give rise to solutions for more complicated channels.

A binary symmetric channel with crossover probability *p* is a channel with binary input and binary output and probability of error *p*; that is, if X is the transmitted random variable and Y the received variable, then the channel is characterized by the conditional probabilities

$$\Pr(Y=0 \mid X=0) = 1-p$$
$$\Pr(Y=0 \mid X=1) = p$$
$$\Pr(Y=1 \mid X=0) = p$$
$$\Pr(Y=1 \mid X=1) = 1-p$$

It is assumed that $0 \le p \le 1/2$. If $p > 1/2$, then the receiver can swap the output (interpret 1 when it sees 0, and vice versa) and obtain an equivalent channel with crossover probability $1-p \le 1/2$ [1].

Additive white Gaussian noise (AWGN)

It is a channel model in which the only impairment to communication is a linear addition of wideband or white noise with a constant spectral density and a Gaussian distribution of amplitude. The model does not account for fading, frequency selectivity, interference, nonlinearity or dispersion. However, it produces simple and tractable mathematical models which are useful for gaining insight into the underlying behaviour of a system before these other phenomena are considered.

Wideband Gaussian noise comes from many natural sources, such as the thermal vibrations of atoms in conductors (referred to as thermal noise or Johnson-Nyquist noise), shot noise, black body radiation from the earth and other warm objects, and from celestial sources such as the Sun.

The AWGN channel is a good model for many satellite and deep space communication links. It is not a good model for most terrestrial links because of multipath, terrain blocking, interference, etc. However, for terrestrial path modelling, AWGN is commonly used to simulate background noise of the channel under study, in addition to multipath, terrain blocking, interference, ground clutter and self-interference that modern radio systems encounter in terrestrial operation.

Gaussian Distribution

Gaussian distribution is a continuous probability distribution defined by the formula,

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

The parameter μ in this formula is the mean of the distribution (and also its median and mode). The parameter σ is its standard deviation; its variance is therefore σ^2. A random variable with a Gaussian distribution is said to be normally distributed. If $\mu = 0$ and $\sigma = 1$, the distribution is called the standard normal distribution or the unit normal distribution. The normal distribution is symmetric about its mean, and is non-zero over the entire real line.

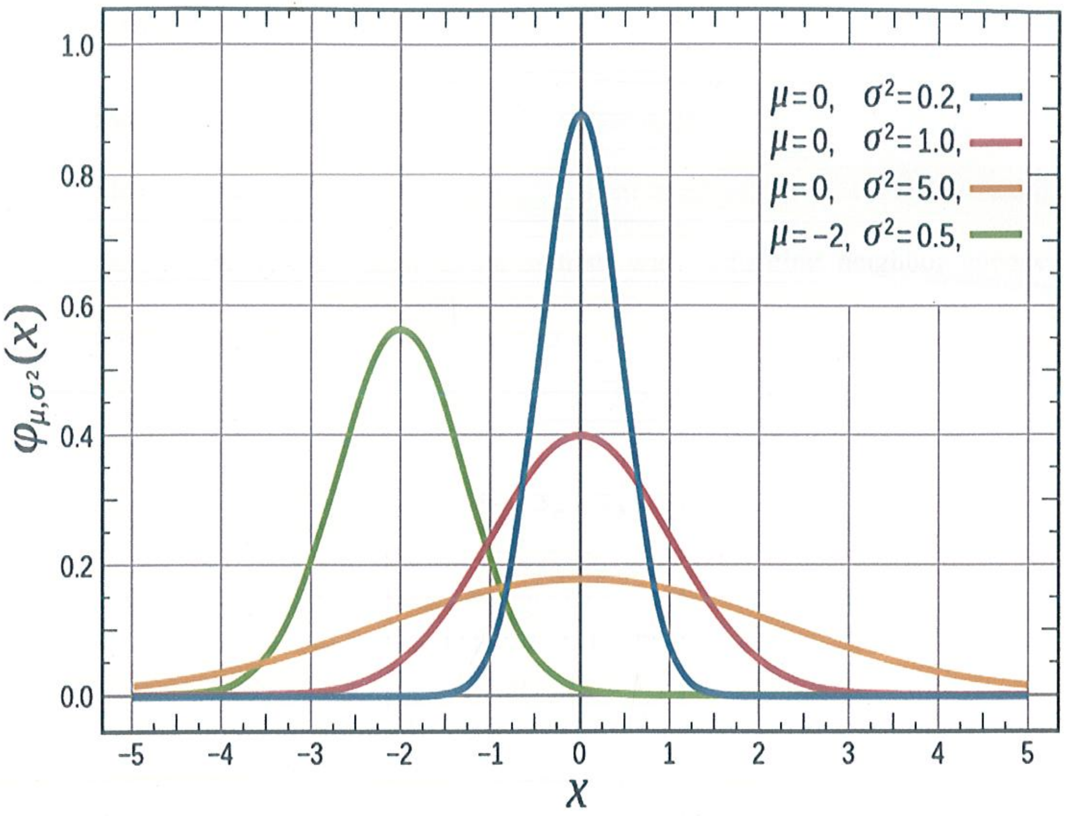


Fig 10: Gaussian Probability Density Function

3.4.4 The Decoder

- Decoder searches for encoded symbol with degree 1.
- This encoding symbol is an exact copy of its neighboring information symbol.
- Decoder XORs this information symbol with remaining neighbor encoded symbols and decreases their degree by 1.
- Repeat the process until all the message symbols have been obtained.

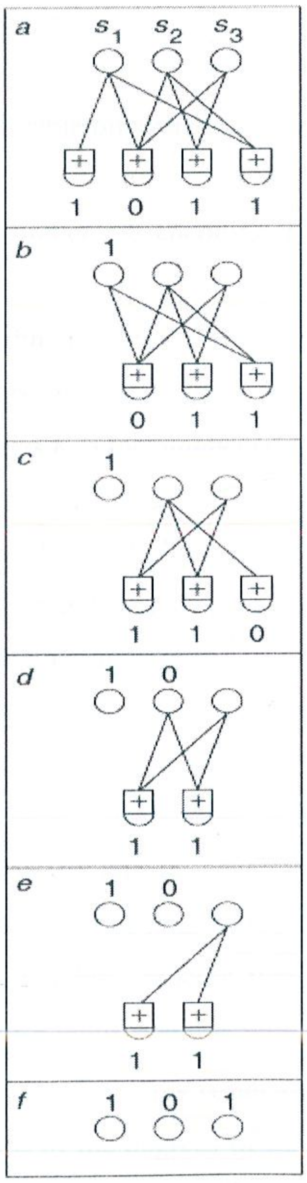


Fig 11: Example decoding for a fountain code with K=3 source bits and N=4 encoded bits

Learning Resource Center
SP0913092
Acc. No. SP09092
JUIT, Waknaghat, Solan (H.P.)

3.5 LT Degree Distributions

Recall that each encoding symbol has a degree chosen independently from a degree distribution.

Degree Distribution: For all d, $\rho(d)$ is the probability that an encoding symbol has degree d. As we now develop, the random behavior of the LT process is completely determined by the degree distribution $\rho(\cdot)$, the number of encoding symbols K, and the number of input symbols k. Our objective is to design degree distributions that meet the following two design goals.

As few encoding symbols as possible on average are required to ensure success of the LT process. Recall that the number of encoding symbols that ensure success of the LT process corresponds to the number of encoding symbols that ensure complete recovery of the data.

The average degree of the encoding symbols is as low as possible. The average degree is the number of symbol operations on average it takes to generate an encoding symbol. The average degree times K is the number of symbol operations on average it takes to recover the entire data.

The classical process of throwing balls into bins can be viewed as a special case of the LT process where all encoding symbols have degree one and thus are all released and thrown initially, i.e. the following distribution.

All-At-Once distribution: $\rho(1) = 1$.

In terms of LT codes, this corresponds to generating each encoding symbol by selecting a random input symbol and copying its value to the encoding symbol. The analysis of the classical balls and bins process implies that $k * ln(k/\delta)$ encoding symbols are needed to cover all k input symbols with probability at least $1 - \delta$ with respect to the All-At-Once distribution. This result can be easily modified to show that for any distribution the sum of the degrees of the encoding symbols must be at least $k * ln(k/\delta)$ to cover all input symbols at least once. Thus, although the total number of symbol operations with respect to the All-At-Once distribution is minimal, the number of encoding symbols required to cover the k input symbols is an unacceptable $ln(k/\delta)$ times larger than the minimum possible. Below we develop the Soliton distribution that ensures that just over k encoding symbols with the sum of their degrees being $O(k * ln(k/\delta))$ suffice to cover all k input symbols. Thus, both the number of encoding symbols and the total number of symbol operations are near minimal with respect to the Soliton distribution.

3.5.1 <u>The Ideal Soliton distribution</u>

The basic property required of a good degree distribution is that input symbols are added to the ripple at the same rate as they are processed. This property is the inspiration for the name Soliton distribution, as a soliton wave is one where dispersion balances refraction perfectly.

A desired effect is that as few released encoding symbols as possible cover a input symbol that is already in the ripple. This is because released encoding symbols that cover input symbols already in the ripple are redundant. Since released encoding symbols cover a random input symbol from among the unprocessed input symbols, this implies that the ripple size should be kept small at all times. On the other hand, if the ripple vanishes before all k input symbols are covered then the overall process ends in failure. The ripple size should be kept large enough to ensure that the ripple does not disappear prematurely. The desired behavior is that the ripple size never gets too small or too large. The Ideal Soliton distribution displays ideal behavior in terms of the expected number of encoding symbols needed to recover the data. Unfortunately, like most ideal things, this distribution is quite fragile, in fact so much so that it is useless in practice. However, its description and analysis captures many of the crucial elements of the robust distributions described later.

<u>Ideal Soliton distribution</u>: The Ideal Soliton distribution is $\rho(1), \ldots, \rho(k)$, where

- $\rho(1) = \frac{1}{k}$
- $for\ all\ i = 2, \ldots k,\ \rho(i) = 1/(i-1)$

Note that $\sum_i \rho(i) = 1$ *as required of a probability distribution.*

Suppose the expected behavior of the LT process is its actual behavior. Then, the Ideal Soliton distribution works perfectly, i.e. exactly k encoding symbols are sufficient to cover each of the k input symbols exactly once. If the expected behavior actually happens, then there is always exactly one input symbol in the ripple, and when this symbol is processed the released encoding symbol covers an input symbol among the unprocessed input symbols to re-establish this condition, etc.

However, this analysis makes the completely unrealistic assumption that the expected behavior is the actual behavior, and this is far from the truth. In fact, the Ideal Soliton distribution works very poorly in practice because the expected size of the ripple is one, and even the smallest variance causes the ripple to vanish and thus the overall process fails to cover and process all input symbols.

Note that for the Ideal Soliton distribution the sum of the degrees of k encoding symbols is on average $k * ln(k)$. Recall that for the All-At-Once distribution it takes on average $k * ln(k)$ encoding symbols to cover all input symbols with constant probability. It is interesting that the number of symbol operations for the Ideal Soliton distribution and for the All-At-Once distribution coincide, although the number of encoding symbols is quite different. The intuition for this is that for any distribution the sum of the degrees of all the encoding symbols needs to be around $k * ln(k)$ in order to cover all the input symbols, but the Ideal Soliton distribution compresses this into the fewest number of encoding symbols possible. Thus, the total amount of computation is the same in both cases (since it is proportional to the sum of the degrees of all the encoding symbols). However, the total amount of information that needs to be received (which is proportional to the number of encoding symbols) is compressed to the minimum possible with the Ideal Soliton distribution.

Following are some examples of plots of ideal soliton distribution

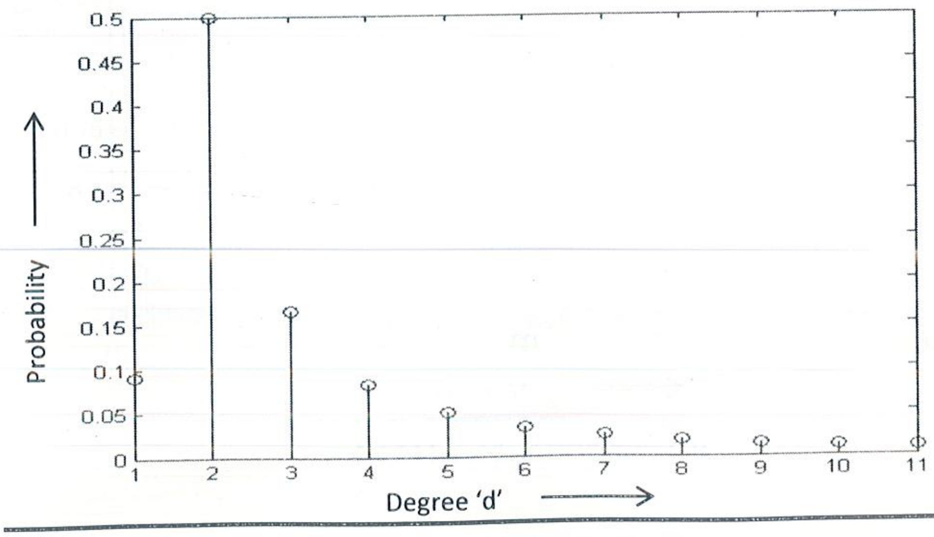


Fig. 12 Ideal Soliton with k=11

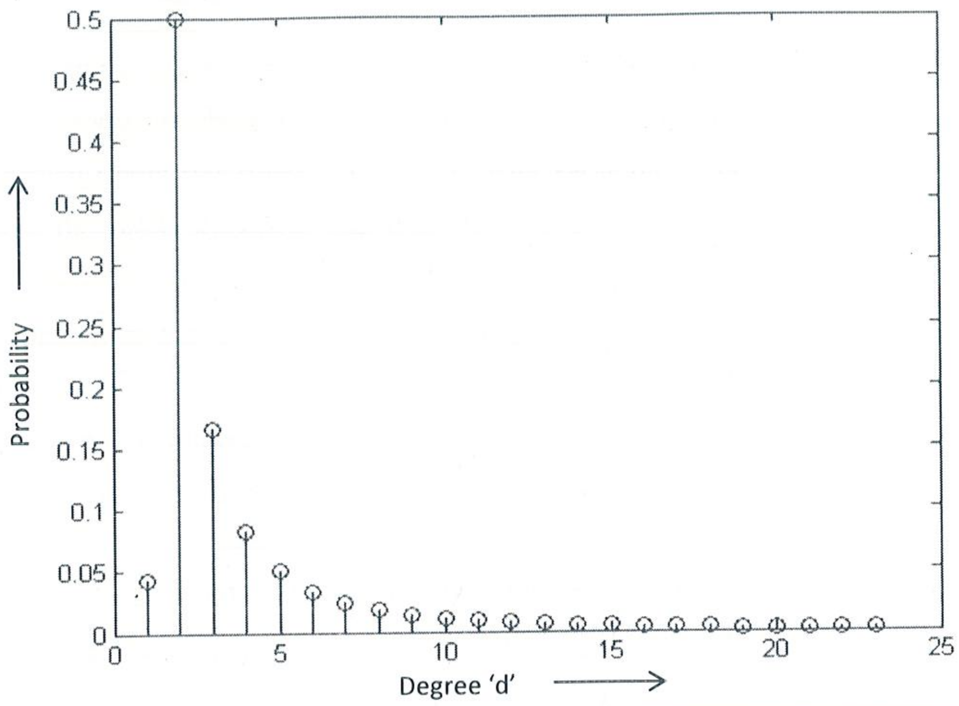


Fig. 13 Ideal Soliton with k=23

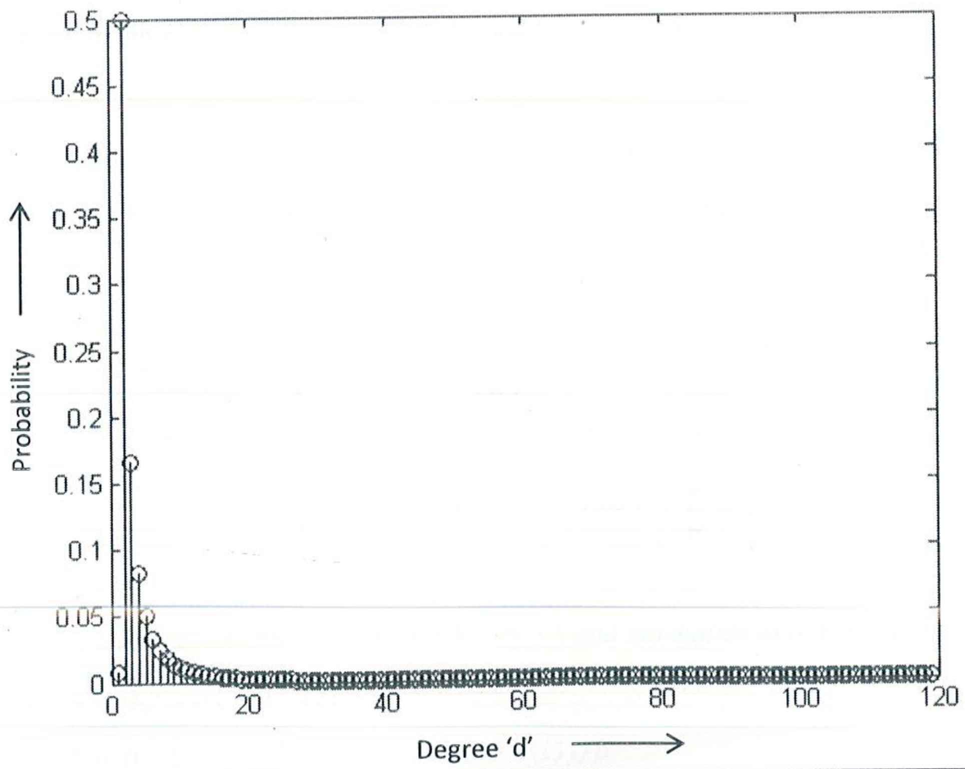


Fig 14 Ideal Soliton with k=119

3.5.2 The Robust Soliton distribution

Although the Ideal Soliton distribution works poorly in practice, it does give insight into a robust distribution. The problem with the Ideal Soliton distribution is that the expected ripple size (one) is too small. Any variation in the ripple size is likely to make the ripple disappear and then the overall process fails. The Robust Soliton distribution ensures that the expected size of the ripple is large enough at each point in the process so that it never disappears completely with high probability. On the other hand, in order to minimize the overall number of encoding symbols used, it is important to minimize the expected ripple size so that not too many released encoding symbols redundantly cover input symbols already in the ripple.

Let δ be the allowable failure probability of the decoder to recover the data for a given number K of encoding symbols. The idea here is to design the distribution so that the expected ripple size is about $ln(k/\delta)\sqrt{k}$ throughout the process.
The intuition is that the probability a random walk of length k deviates from its mean by more than $ln(k/\delta)\sqrt{k}$ is at most δ. As we describe below, it turns out this can be achieved using $K = k + O(ln^2\left(\frac{k}{\delta}\right)\sqrt{k})$ encoding symbols.

Robust Soliton distribution: The Robust Soliton distribution is $\mu(\cdot)$ defined as follows. Let $R = c * ln(k/\delta)\sqrt{k}$ for some suitable constant $c > 0$. Define

$$\tau(i) = \begin{cases} \dfrac{R}{ik} & for\ i = 1, \dots \dfrac{k}{R} \\ \dfrac{Rln\left(\frac{R}{\delta}\right)}{k} & for\ i = \dfrac{k}{R} \\ 0 & for\ i = \dfrac{k}{R} + 1, \dots k \end{cases}$$

Add the Ideal Soliton distribution $\rho(\cdot)$ to $\tau(\cdot)$ and normalize to obtain $\mu(\cdot)$:

- $\beta = \sum_{i=1}^{k} \rho\,(i) + \tau(i)$
- $for\ all\ i = 1, \dots k,\ \mu(i) = (\rho(i) + \tau(i))/\beta$

The intuition for the supplement $\tau(\cdot)$ is the following. At the beginning, $\tau(1)$ ensures that the ripple starts off at a reasonable size. Consider the process in the middle. Suppose that an input symbol is processed and L input symbols remain unprocessed. Since the ripple size decreases by one each time an input symbol is processed, on average the ripple should be increased by one to make up for this decrease. If the ripple size is R then the chance that a released encoding symbol adds to the ripple is only $(L - R)/L$. This implies that at this point it requires $L/(L-R)$ released encoding symbols on average to add one to the ripple. Thus, if the ripple size is to be maintained at approximately R, then the density of encoding symbols with degree $i = k/L$ should be proportional to

$$\frac{L}{i(i-1)*(L-R)} = \frac{k}{i(i-1)*(k-iR)}$$

$$= \frac{1}{i(i-1)} + \frac{R}{(i-1)*(k-iR)} \approx \rho(i) + \tau(i),$$

for $i = 2, \ldots, k/R-1$. The final spike $\tau(k/R)$ ensures that all the input symbols unprocessed when $L=R$ are all covered. This is similar to simultaneously releasing $R\ln(R/\delta)$ encoding symbols when R input symbols remain unprocessed to cover them all at once. Thus, the wastage caused by releasing enough encoding symbols to cover each of these input symbols at least once is only a small fraction of the total number k of input symbols.

We set the number of encoding symbols to $K = k\beta$. This implies that$k * (\rho(i) + \tau(i))$ is the expected number of encoding symbols of degree I [5].

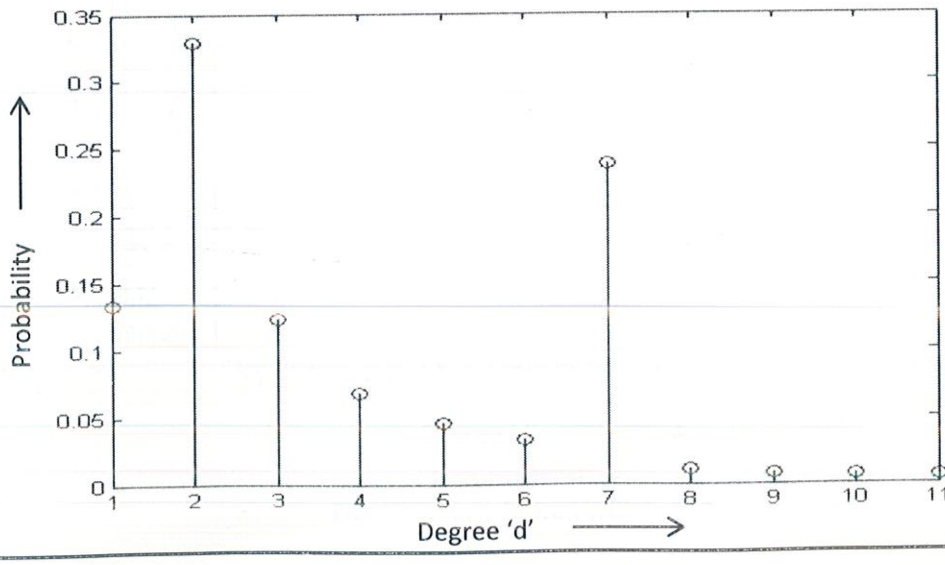


Fig. 15 Robust Soliton with k=11

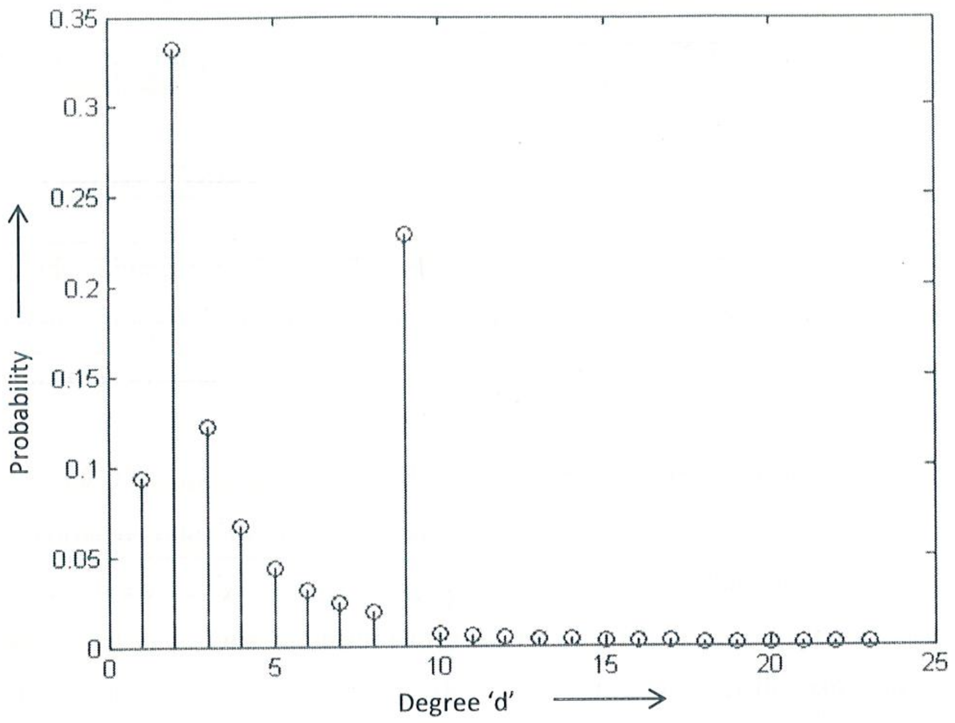


Fig. 16 Robust Soliton with k=23

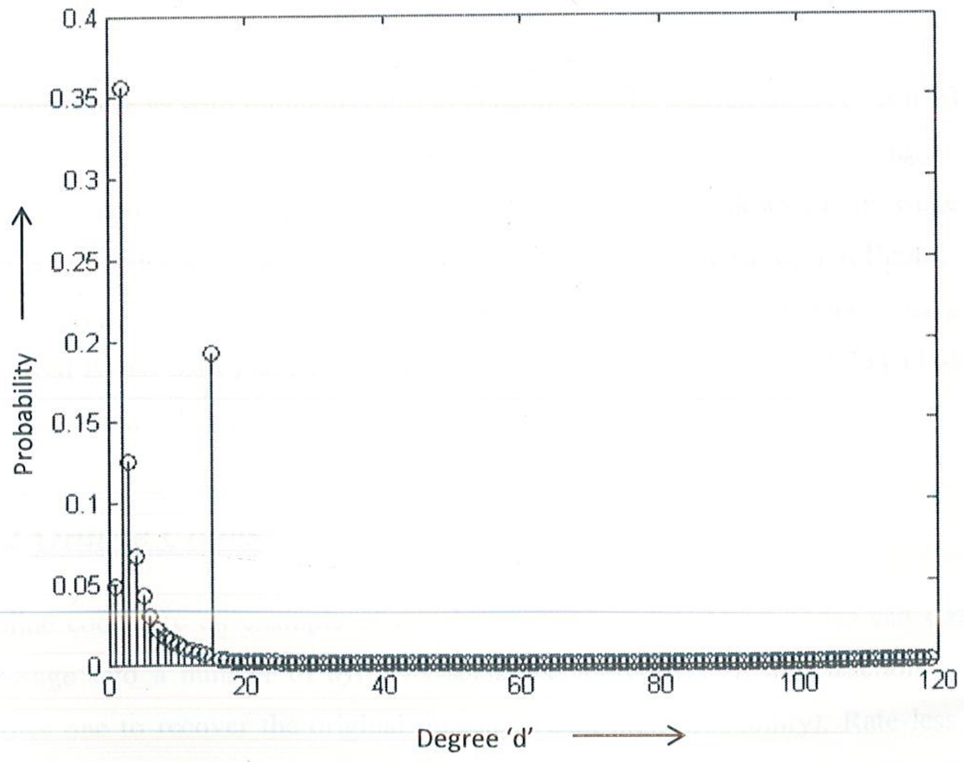


Fig. 17 Robust Soliton with k=119

CHAPTER 4

OTHER TYPES OF FOUNTAIN CODES

4.1 Raptor Codes

LT codes illuminated the benefits of considering rate less codes in realizing a digital fountain. However, they require the decoding cost to be *O(ln(k))* in order for every information symbol to be recovered and decoding to be successful. Raptor (rapid Tornado) codes were developed and patented in 2001 as a way to reduce decoding cost to *O(1)* by pre-processing the LT code with a standard erasure block code (such as a Tornado code). In fact, if designed properly, a Raptor code can achieve constant per-symbol encoding and decoding cost with overhead close to zero and a space proportional to k. This has been shown to be the closest code to the ideal universal digital fountain. A similar vein of work was proposed in under the name online codes. We have already seen two extreme cases of Raptor codes. When there is no pre-code, then we have the LT code. On the other hand, we can consider a block erasure code such as the Tornado code as an example of a pre-code only (PCO) Raptor code for the BEC.

Raptor codes, as with fountain codes in general, encode a given message consisting of a number of symbols, k, into a potentially limitless sequence of encoding symbols such that knowledge of any k or more encoding symbols allows the message to be recovered with some non-zero probability. With the latest generation of Raptor codes, the RaptorQ codes, the chance of decoding failure when k symbols have been received is less than 1%, and the chance of decoding failure when $k+2$ symbols have been received is less than one in a million.

4.2 Online Codes

Online codes are an example of rate-less erasure codes. These codes can encode a message into a number of symbols such that knowledge of any fraction of them allows one to recover the original message (with high probability). Rate-less codes produce an arbitrarily large number of symbols which can be broadcast until the receivers have enough symbols.

Unlike LT codes, online codes have a stronger recoverability guarantee. They fail to recover the encoded message with probability that goes to 0 as the message size n grows. Furthermore, online codes only take constant time to generate an encoding block.

CHAPTER 5

APPLICATIONS OF LT CODES

LT codes are an excellent solution in a wide variety of situations, two of which are mentioned over here.

5.1 Storage

You wish to make a back-up of a large file, but you are aware that your magnetic tapes and hard drives are all unreliable: catastrophic failures, in which some stored packets are permanently lost within one device, occur at a rate of something like 10^{-3}per day. How should you store your file?

A LT code can be used to spray encoded packets all over the place, on every storage device available. To recover the file, whose size was K packets, one simply needs to find $K' \approx K$ packets from anywhere. Corrupted packets do not matter; we simply skip over them and find more packets elsewhere.

This method of storage also has advantages in terms of speed of file recovery. In a hard drive, it is standard practice to store a file in successive sectors of a hard drive, to allow rapid reading of the file; but if, as occasionally happens, a packet is lost (owing to the reading head being off track for a moment, giving a burst of errors that cannot be corrected by the packet's error-correcting code), a whole revolution of the drive must be performed to bring back the packet to the head for a second read. The time taken for one revolution produces an undesirable delay in the file system. If files were instead stored using the fountain principle, with the digital drops stored in one or more consecutive sectors on the drive, then one would never need to endure the delay of rereading a packet; packet loss would become less important, and the hard drive could consequently be operated faster, with higher noise level, and with fewer resources devoted to noisy-channel coding.

5.2 Broadcast

Imagine that ten thousand subscribers in an area wish to receive a digital movie from a broadcaster. The broadcaster can send the movie in packets over a broadcast network, for example, by a wide-bandwidth phone line, or by satellite. Imagine that *f=0.1%* of the packets are lost at each house. In a standard approach in which the file is transmitted as a plain sequence of packets with no encoding, each house would have to notify the broadcaster of the *fK* missing packets, and request that they be retransmitted. And with ten thousand subscribers all requesting such retransmissions, there would be a retransmission request for almost every packet. Thus the broadcaster would have to repeat the entire broadcast twice in order to ensure that most subscribers have received the whole movie, and most users would have to wait roughly twice as long as the ideal time before the download was complete.

If the broadcaster uses a LT code to encode the movie, each subscriber can recover the movie from any $K' \approx K$ packets. So the broadcast needs to last for only, say, *1.1K* packets, and every house is very likely to have successfully recovered the whole file.

Another application is broadcasting data to cars. Imagine that we want to send updates to in-car navigation databases by satellite. There are hundreds of thousands of vehicles, and they can only receive data when they are out on the open road; there are no feedback channels. A standard method for sending the data is to put it in a carousel, broadcasting the packets in a fixed periodic sequence. 'Yes, a car may go through a tunnel, and miss out on a few hundred packets, but it will be able to collect those missed packets an hour later when the carousel has gone through a full revolution; or may be the following day'. If instead the satellite uses a fountain code, each car needs to receive only an amount of data equal to the original file size (plus 5%).

5.3 Asynchronous Data Transfer

This project is itself an application of LT codes in which there are multiple transmitters, all of which are carrying the same information, which can be a movie or any other file. One receiver is connected to all these transmitters and trying to obtain that file. The transmitters are asynchronous in nature i.e. there is no coordination between them. The receiver downloads the data simultaneously from all the transmitters using LT code methodology in a speedy way. This process also conserves bandwidth as there is no retransmission path required.

CHAPTER 6

METHODOLOGY

- Initially, we did some research on fountain codes by reading various research papers and patents and gathering information about what they are, what was the need to develop such codes and finally how to implement them.

- After understanding the process, the first task was to define the ideal and robust Soliton distributions for generation of degree of each encoded symbol.

- The next task was to generate at random the degree for each encoded symbol using the Soliton distributions. This was achieved with the help of a cumulative probability distribution which was generated from the Robust Soliton Distribution.

- After the successful generation and allotment of degrees to each encoded symbol we started the encoding of message symbols into the encoded symbols. This was done by randomly selecting d number of unique message symbols which act as neighbours of the encoded symbol and then XOR-ing them to form the encoded symbol.

- The encoded symbols and their neighbours were stored in a generator matrix of size (k,n), where k is the no. of message symbols and n is the no. of encoded symbols.

- This generator matrix is used by the decoder for decoding the message symbols thereby recovering the exact data which was sent.

- Next we add Binary symmetric channel and then follow all the steps that were mentioned above. Prior to decoding the receiver compares the original encoded symbols and the new ones and removes the erroneous encoded symbols and their respective columns from the generator matrix.

- We also added an AWGN channel and then followed the same steps as mentioned above.

- Finally we made three separate functions for three transmitters all of them having the same information.

- We then implemented three separate noise channels, one for each transmitter. The three channels have different SNR.

- The decoder at the receiver removes the erroneous symbols received from each of the transmitters. Then, some encoded symbols are chosen from each of the transmitters and are concatenated together to form a new encoded symbol matrix. Similar procedure is followed with the generator matrix to give a final generator matrix.

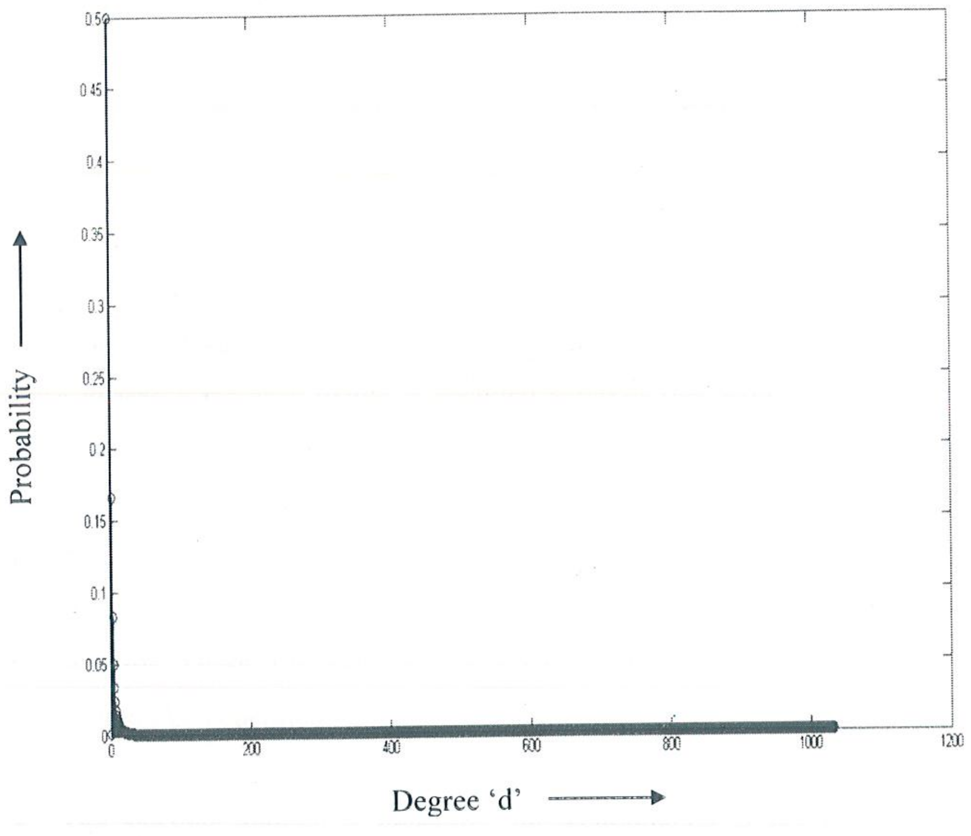
- The receiver uses the updated generator matrix to decode the received encoded symbols, thereby obtaining the complete data.

CHAPTER 7

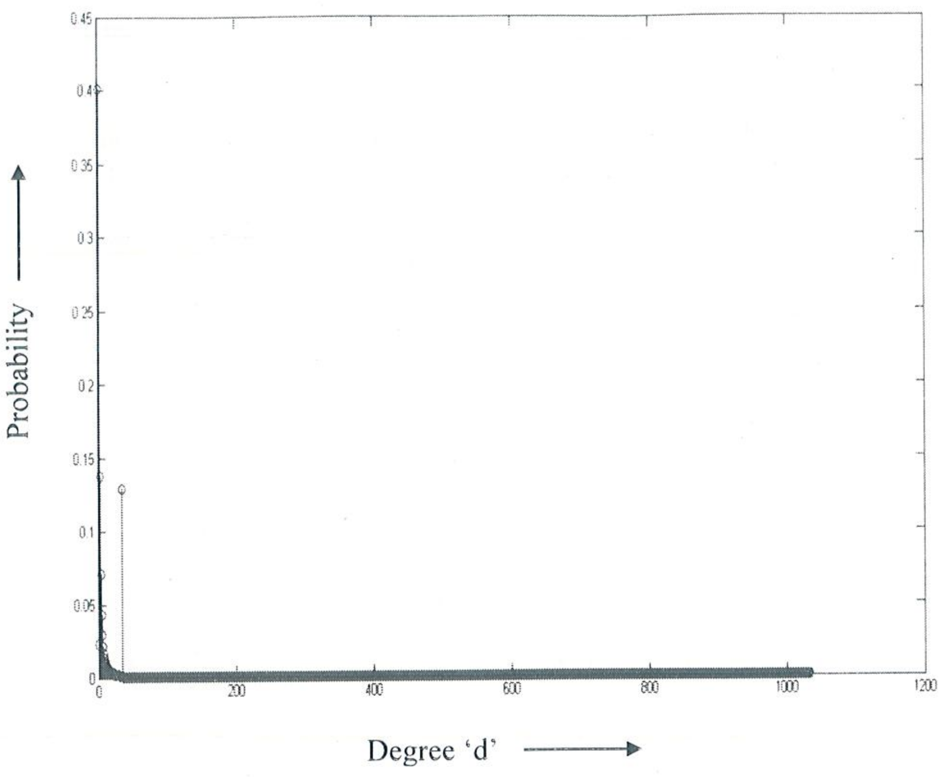
RESULTS

We are working with three asynchronous transmitters and one common receiver. First let us consider the case when the channel is ideal i.e. there is no noise in the channel.

- The first step was to generate a message vector, named **msg** of length *k*. We have taken *k* to be 1033. The same message symbols are used in all the three transmitters.
- Next we successfully generated the Ideal Soliton Distribution.



- Then we successfully generated the Robust Soliton Distribution using the Ideal Soliton Distribution that we generated above.



- Using the results from the Robust Soliton Distribution, we randomly generated a degree *d* for each of the *N* encoded symbols that would be generated. We have taken *N* to be twice of *k*.

- We stored the generated degrees in a vector of length *N*. Then using these degrees we chose uniformly at random *d* distinct symbols out of the *k* message symbols. These *d* message symbols act as neighbours to the encoded symbol that will be generated.

- The encoded symbol is generated by XOR-ing the *d* neighbours with one another.

- The generated encoded symbols and their neighbour information is stored in the form of a generator matrix of size (*k*,*N*). The rows of the generator matrix represent the message symbols while the columns represent the encoded symbols. The neighbours of an encoded symbol are denoted by a 1 in the matrix.

- As the original generator matrix for 1033 message symbols would have been too large to show, we have shown a similar generator matrix for k = 13 message symbols.

```
generator1 =

  0  0  0  0  0  0  0  0  0  1  0  1  0  0  0  1  0  1  1  0  0  0  0  0  0
  0  0  1  0  1  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  1  0  0
  1  0  0  0  1  0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  1  0  0  0
  0  0  0  0  0  0  0  0  1  0  0  1  0  0  0  1  0  0  0  0  0  0  0  0  0
  1  0  0  1  1  0  1  0  1  1  1  0  0  0  0  0  0  1  0  0  1  0  1  0  1
  0  0  0  0  1  0  0  0  0  1  0  1  0  1  1  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  1  0  0  1  0  1  0  0  0  1  1  1  0  0  0  0  1  1  0  0
  0  0  0  0  1  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  1  0  0  1  0
  0  0  1  0  1  0  0  1  0  1  0  1  1  0  1  0  0  1  0  1  0  0  0  1  1
  0  0  0  1  0  0  0  0  1  1  0  0  0  0  1  1  0  1  1  0  0  0  1  1  0
  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  1  1  0  1  1  0  0  1  0  0  1  0  0  1  0  0  0  0  0  0  0  1  0  0
  0  0  0  0  0  0  1  1  1  0  0  0  0  0  1  0  1  0  0  0  0  0  1  0  0
```

- Assumption is made that the generator matrix is provided to the decoder through some suitable means.

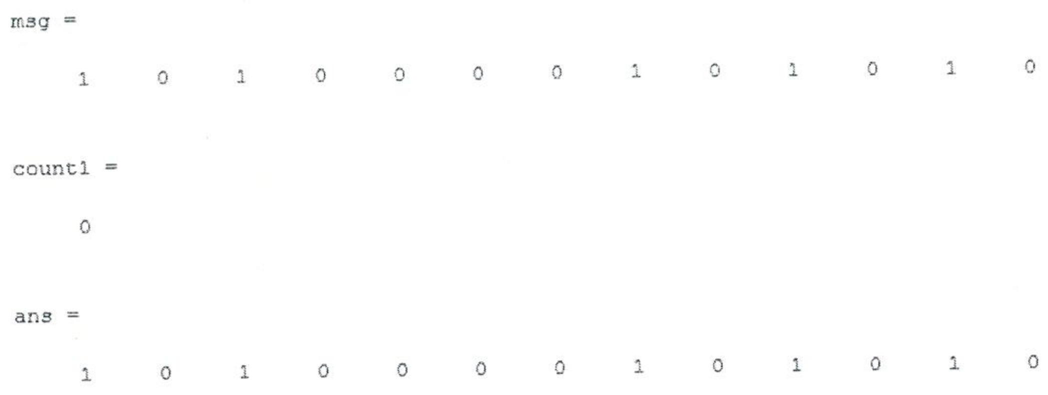
- The generator matrices from all the three encoders are made available to the decoder. Portions of these matrices are concatenated together to form a new generator matrix which has the same size as that of the original matrices.

- The above procedure is followed on the encoded symbol vectors from the three encoders to give a final encoded symbol vector.

- The decoder then successfully decodes the new generator matrix to give the message symbols which are stored in a vector.

- We compared the decoded symbols vector with the original symbols vector **msg** by using a counter **count1** which counts the number of decoding errors. If **count1** is 0 then the decoding would be considered successful.

- We have shown an example for $k = 13$ message symbols.



```
msg =

     1     0     1     0     0     0     0     1     0     1     0     1     0


count1 =

     0


ans =

     1     0     1     0     0     0     0     1     0     1     0     1     0
```

Next we add Binary Symmetric Channel to the system.

- We generate the encoded symbols and the generator matrix in the same manner as discussed for the ideal channel case.

- Then we corrupt the encoded symbols of each of the three transmitters using three different binary symmetric channels. The crossover probability used in each of the three channels is different.

- In the screenshot below, ans is the corrupted encoded symbols vector. We have produced this screenshot for $k = 13$.

```
enc1 =

  Columns 1 through 25

     0     1     1     1     0     0     0     1     1     0     1     1     1     1     0     0     1     1     1     1     1     0     0     1     0

  Column 26

     1


ans =

  Columns 1 through 25

     0     1     1     1     0     0     0     1     1     0     0     1     1     1     0     0     1     1     0     1     1     0     1     1     0

  Column 26

     1
```

- Next we compare the corrupted encoded symbols with the original encoded symbols and remove the erroneous encoded symbols from the three corrupted vectors. We also remove the columns corresponding to the erroneous encoded symbols from the three generator matrices.

- The process mentioned above is a way of implementing erasure correction in the system.

- Below we have shown two screenshots of the generator matrix of the first transmitter before BSC is added and after removal of columns corresponding to erroneous encoded symbols.

```
generator1 =

  Columns 1 through 14

     1     1     1     0     0     0     0     0     0     0     0     0     1     1
     1     0     0     0     1     0     0     1     0     0     1     1     0     0
     1     0     0     0     1     0     0     1     0     0     0     0     0     0
     1     0     0     0     0     1     0     1     0     0     0     0     0     1
     1     1     0     0     1     0     0     1     0     0     0     0     1     0
     1     1     0     0     1     0     0     0     0     0     0     0     1     1
     1     1     0     0     0     0     0     0     0     0     0     0     0     1
     1     0     0     0     0     0     0     0     0     0     0     0     0     0
     1     0     0     1     1     1     1     1     0     0     0     0     1     1
     1     0     0     0     0     0     0     1     1     0     1     0     1     0
     1     1     0     0     0     0     1     1     0     1     0     0     0     1
     1     1     1     0     1     0     0     0     0     0     0     0     0     1
     1     1     0     0     1     1     0     0     0     1     0     0     1     0

  Columns 15 through 26

     0     0     0     0     0     1     1     1     0     1     0     0
     0     0     1     1     1     1     1     1     0     0     0     0
     0     1     0     0     0     1     0     1     0     0     0     1
     0     0     1     0     1     0     0     1     1     0     1     1
     0     0     1     1     1     0     1     1     0     1     1     0
     0     0     0     0     0     0     1     1     0     1     1     0
     0     0     1     1     1     1     0     1     0     0     1     0
     1     0     0     0     0     1     0     1     0     0     0     0
     0     0     0     1     1     1     1     1     0     1     0     0
     0     0     1     0     0     0     1     1     0     1     0     1
     1     0     0     1     1     0     1     1     0     0     1     1
     0     1     1     0     0     0     0     1     1     0     1     0
     0     0     1     1     1     1     0     1     0     0     1     0
```

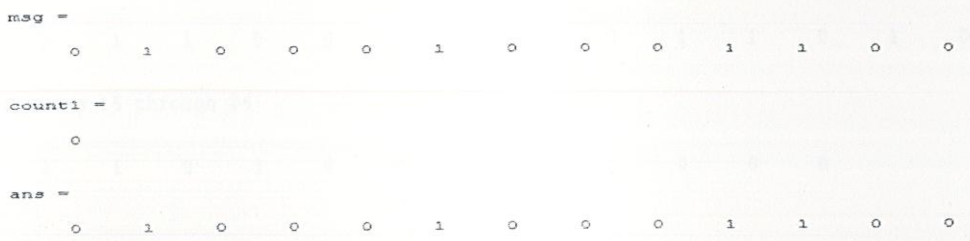
```
generator1 =

     1     1     1     0     0     0
     1     0     0     1     0     0
     1     0     0     1     0     0
     1     0     0     0     1     0
     1     1     0     1     0     0
     1     1     0     1     0     0
     1     1     0     0     0     0
     1     0     0     0     0     0
     1     0     0     1     1     1
     1     0     0     0     0     0
     1     1     0     0     0     1
     1     1     1     1     0     0
     1     1     0     1     1     0
```

- The three new generator matrices are concatenated together to give a new generator matrix. The same is done with the three new encoded symbol vectors to give a new error-free encoded symbol vector.

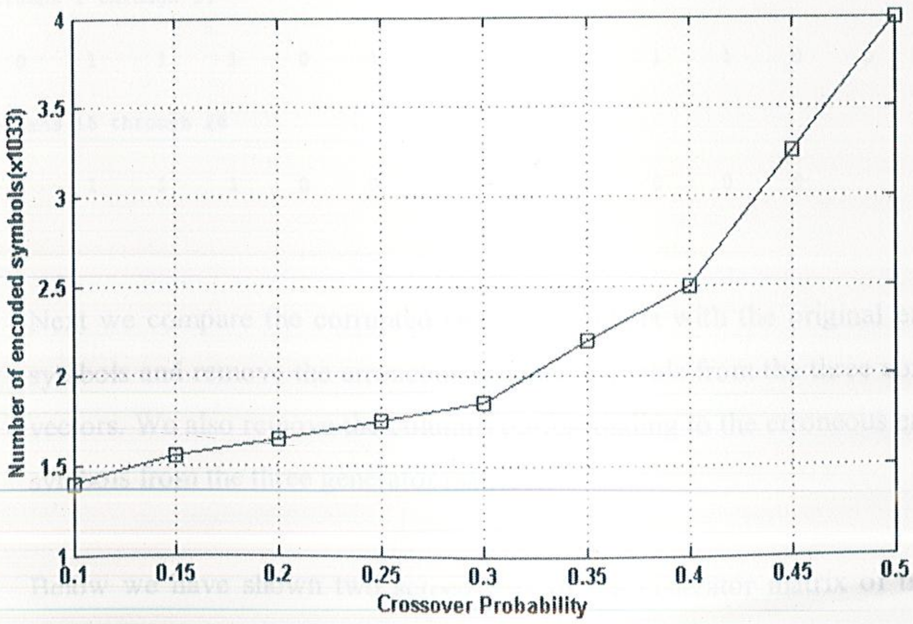
- The decoder then decodes the generator matrix to give the message symbols.

- We compare the decoded message symbols with the original ones by using **count1** MATLAB function.

- Below is a screenshot in which we have compared the decoded message symbols(**sol**) with the original ones(**msg**).



```
msg =

     0     1     0     0     0     1     0     0     0     1     1     0     0

count1 =

     0

ans =

     0     1     0     0     0     1     0     0     0     1     1     0     0
```



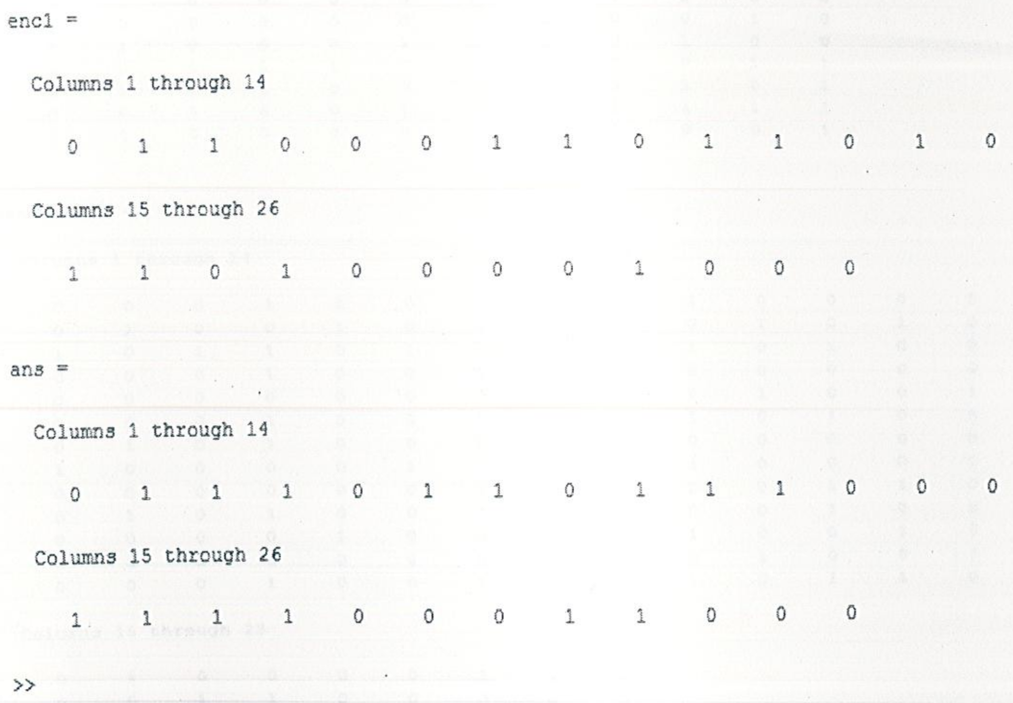
As we increase the crossover probability of the Binary Symmetric Channel, the overheads increase thereby increasing the decoding time of the decoder.

Now we add Additive White Gaussian Noise (AWGN) to the system.

- We generate the encoded symbols and the generator matrix in the same manner as discussed for the ideal channel case.

- Then we corrupt the encoded symbols of each of the three transmitters using three AWGN channels. The Signal to Noise Ratio (SNR) used in each of the three channels is different.

- In the screenshot below, ans is the corrupted encoded symbols vector. We have produced this screenshot for $k = 13$.



```
enc1 =

  Columns 1 through 14

     0     1     1     0     0     0     1     1     0     1     1     0     1     0

  Columns 15 through 26

     1     1     0     1     0     0     0     0     1     0     0     0


ans =

  Columns 1 through 14

     0     1     1     1     0     1     1     0     1     1     1     0     0     0

  Columns 15 through 26

     1     1     1     1     0     0     0     1     1     0     0     0

>>
```

- Next we compare the corrupted encoded symbols with the original encoded symbols and remove the erroneous encoded symbols from the three corrupted vectors. We also remove the columns corresponding to the erroneous encoded symbols from the three generator matrices.

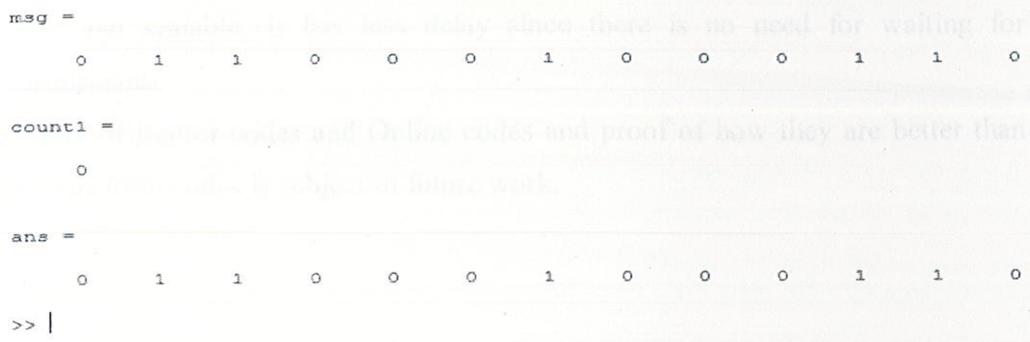
- Below we have shown two screenshots of the generator matrix of the first transmitter before BSC is added and after removal of columns corresponding to erroneous encoded symbols.

```
generator1 =

  Columns 1 through 14

     0     0     0     1     0     0     0     0     0     0     0     0     1     0
     0     1     0     0     0     1     0     0     0     0     0     0     0     1
     1     0     1     1     0     0     1     1     0     0     1     0     1     0
     0     0     0     1     0     0     0     0     0     1     1     1     0     0
     0     0     0     0     0     0     0     0     0     0     1     0     1     1
     0     0     0     1     0     0     0     0     0     0     0     0     1     0
     0     1     0     1     0     0     0     0     0     0     0     0     0     0
     1     0     0     0     0     0     1     0     0     0     0     0     1     0
     0     0     0     0     0     0     0     0     1     0     1     0     0     0
     0     1     0     1     0     0     0     0     0     0     1     0     0     0
     0     0     0     0     1     1     0     0     0     0     1     1     1     0
     0     0     0     0     0     0     0     0     0     0     1     0     0     1
     0     0     0     1     0     0     0     1     0     0     0     0     1     0

  Columns 15 through 26

     0     0     0     0     1     0     0     0     0     1     1     0
     0     1     0     0     0     1     1     0     0     0     0     0
     1     0     0     0     0     0     1     1     0     0     1     0
     0     0     0     0     0     0     1     1     0     1     0     1
     0     0     1     1     1     1     0     1     0     0     1     1
     1     0     0     0     1     0     0     0     0     1     1     1
     0     0     0     0     0     0     0     0     0     1     0     0
     0     0     0     0     0     0     1     1     0     0     1     0
     1     1     0     0     0     1     1     1     0     1     0     0
     1     0     0     0     1     0     1     0     0     0     1     1
     0     1     0     1     0     0     1     0     1     1     0     1
     0     0     1     0     0     1     0     1     1     1     1     1
     1     1     0     0     0     0     0     1     0     0     0     1


generator1 =

  Columns 1 through 14

     0     0     0     1     0     0     0     0     0     1     0     0     0     0
     0     1     0     0     1     0     0     0     0     0     1     0     1     0
     1     0     1     1     0     1     1     1     0     1     0     1     0     0
     0     0     0     1     0     0     0     1     1     0     0     0     0     0
     0     0     0     0     0     0     0     1     0     1     1     0     0     1
     0     0     0     1     0     0     0     0     0     1     0     1     0     0
     0     1     0     1     0     0     0     0     0     0     0     0     0     0
     1     0     0     0     0     1     0     0     0     1     0     0     0     0
     0     0     0     0     0     0     0     1     0     0     0     1     1     0
     0     1     0     1     0     0     0     1     0     0     0     1     0     0
     0     0     0     0     1     0     0     1     1     1     0     0     1     0
     0     0     0     0     0     0     0     1     0     0     1     0     0     1
     0     0     0     1     0     0     1     0     0     1     0     1     1     0

  Columns 15 through 23

     0     1     0     0     0     0     1     1     0
     0     0     1     1     0     0     0     0     0
     0     0     0     1     1     0     0     1     0
     0     0     0     1     1     0     1     0     1
     1     1     1     0     1     0     0     1     1
     0     1     0     0     0     0     1     1     1
     0     0     0     0     0     0     1     0     0
     0     0     0     1     1     0     0     1     0
     0     0     1     1     1     0     1     0     0
     0     1     0     1     0     0     0     1     1
     1     0     0     1     0     1     1     0     1
     0     0     1     0     1     1     1     1     1
     0     0     0     0     1     0     0     0     1
```

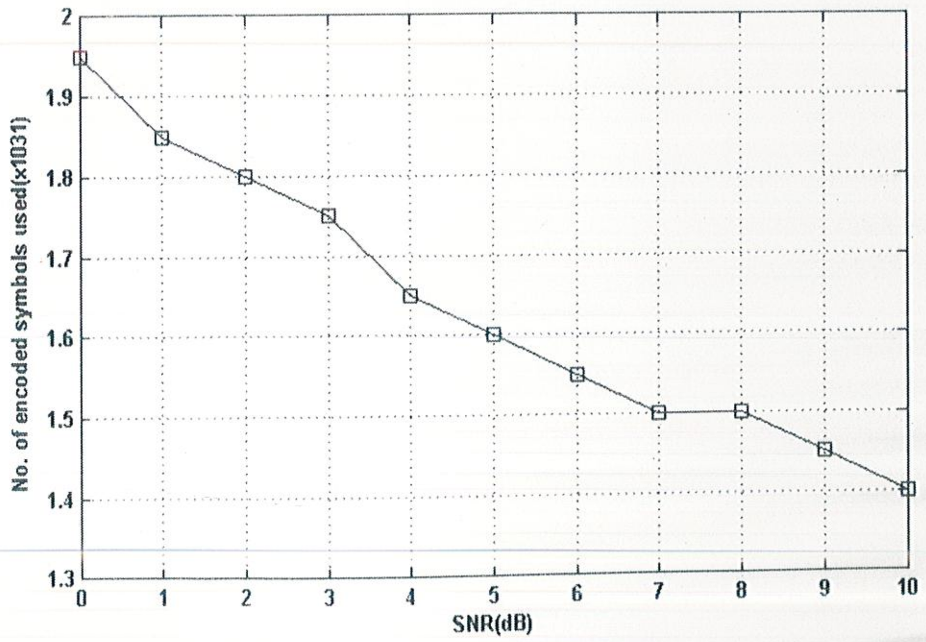
- The three new generator matrices are concatenated together to give a new generator matrix. The same is done with the three new encoded symbol vectors to give a new error-free encoded symbol vector.

- The decoder then decodes the generator matrix to give the message symbols.
- We compare the decoded message symbols with the original ones by using **count1** MATLAB function.

- Below is a screenshot in which we have compared the decoded message symbols(**sol**) with the original ones(**msg**).



```
msg =

     0     1     1     0     0     0     1     0     0     0     1     1     0

count1 =

     0

ans =

     0     1     1     0     0     0     1     0     0     0     1     1     0

>> |
```



As we can see from the plot, on increasing the input SNR(dB), the overheads decreases thereby reducing the decoding time.

CONCLUSION

The overall objective of this project is to transfer data asynchronously to a receiver who is trying to download a file from several sources. The initial results over the binary symmetric channel and the AWGN channel show that although the codes generated using the Robust Soliton distribution are decodable with smaller overhead, the overheads will be high when the channel noise is high. Using LT codes data can be downloaded from multiple sources without any coordination between the transmitters. Further, there is no need for retransmission and hence, no need for a reverse channel to send retransmission requests. This makes the technique bandwidth-efficient and scalable. It has less delay since there is no need for waiting for retransmissions.

The study of Raptor codes and Online codes and proof of how they are better than Luby Transform codes is subject of future work.

BIBLIOGRAPHY

- Mobile Radio Communications - Raymond Steele and Lajos Hanzo
- Wireless Communication: Principles and Practice - Theodore S. Rappaport
- Communication Networks: Fundamental concepts and Key Architecture -Leon Garcia and Widjaja
- Error Correction Coding: Mathematical methods and algorithms -Todd K. Moon

REFERENCES

1. Ye Tian, Kai Xu, Nirwan Ansari (March 2005). TCP in Wireless Environments.

2. A digital fountain approach to reliable distribution of bulk data- Byers, Luby, Mitzenmacher, Rege

3. Capacity approaching codes design and implementation, Special Section, Fountain Codes, D.J.C. MacKay

4. 6.972 Principles Of Digital Communication II, Fountain Codes, Gauri Joshi, Joong Bum Rhim, John Sun, Da Wang

5. LT Codes, Michael Luby, Digital Fountain, Inc.

6. www.wikipaedia.org

7. Shokrollahi (2006), "Raptor Codes", Transactions on Information Theory (IEEE) 52 (6): 2551-2567.

8. P. Maymounkov (November 2002). "Online Codes". (Technical Report).

9. Ravi Palanki, Jonathan S Yedidia, "Rateless codes on Noisy channels", IEEE International Symposium on Information Theory (ISIT), June 2004.