**Master’s Project: Fountain Code Chat Server**

**Austin Wolcott**

Montclair State University

wolcotta1@montclair.edu

**Abstract**

Many existing communication networks such as mobile terrestrial networks are susceptible to packet loss. For Internet-based real-time applications such as video streaming that rely bon UDP, packet loss is also common. TCP addresses packet loss by retransmitting packets, but packet retransmission creates communication inefficiencies that reduce bandwidth. When used on a network prone to dropping packets, common protocols such as TCP and UDP may lead to data corruption or failure to transmit the data properly. Similarly, on networks which enforce online censorship, data may be intentionally dropped during transmission, to prevent a message from reaching their intended destination. Such limitations can be avoided with the implementation of fountain codes. Fountain codes have a unique property that provides the ability to reconstruct lost packets without packet retransmissions. This paper presents a proof of concept for a real-time chat room application, where each client could send messages to other clients by using a fountain code . The chat room works for messages of varying lengths.

**Introduction**

TCP suffers in lossy environments as it does not distinguish between losses of packets that are due to congestion in the network and other types of losses. In case of congestion, TCP backs off its rate in order to avoid link overload and promote fair sharing of the network resources (Hourt et al., 2013).

Wireless and cellular communication may sometimes be faulty. They are more prone to lossy communication than wired networks are. This can lead to data corruption when using that communication channel. For example, sharing a file over a lossy network may lead to the file being corrupted during transmission, due to certain packets of information being dropped by the network. Typically, when this occurs, the data is retransmitted over the network, but this puts more stress on a network. If possible, sending redundant information over a network should be avoided. Whether benign messages sent between two friends or critical data between revolutionaries or government officials, a lossy network could be detrimental.

For networks with unreliable data transmission, fountain codes may be beneficial. Fountain code encoding gives the receiver the ability to reconstruct portions of plaintext which may have been lost during transmission by XOR’ing chunks of plaintext into tokens which are sent over the network. The receiver can XOR certain tokens to reconstruct pieces of the plaintext which may have been lost in transmission, minimizing the need to retransmit the same data over the network. Therefore, fountain codes may be successfully implemented for use with lossy communication channels. If properly implemented, fountain codes could limit the amount of redundant transmissions which occur over a given network. Similar to lossy networks, censored networks also prevent information from reaching its final destination.

Censorship is when certain information, fitting specific requirements, is intentionally withheld from reaching its final destination. Whether its destination is a wide audience, a specific person, or a single server, that message could be considered censored if it never reaches that destination due to someone or something restricting the information contained within it. In China, for example, their government censors a large portion of the internet due to the fact they don’t approve of the information their citizens could come across. In some countries, the governing bodies require that organizations register with them to allow the decryption of data sent over their network. In countries where this is the case, encryption may not be an effective solution for secure network communication.

In the context of this paper, the method of censorship we consider is dropping packets. If traffic deemed malicious is detected, the network could simply drop the packet to halt the spread of that information. This can manifest in an email which is never received, or a web page which returns an error code. This project’s implementation of Fountain Codes could be expanded to potentially be used to circumvent censorship.

**Literature Review**

Azab et al. (2022) discussed the current state of network traffic classification. In particular, he writes about port-based identification and deep packet inspection, as well as their drawbacks. He also mentions machine learning technology in regards to network traffic classification.

Tian et al. (2020) discussed covert channels, in relation to constructing them. They write that there are two levels for which they could create covert channels: the communication content channel and the transmission network level. They also introduce a few new types of network covert channels. These are based on blockchain, IPv6 and streaming media. They also discuss attacks against covert channels, particularly elimination, detection, and limitation.

Marciniszyn (2022) discusses placing covert channels within a fountain code encoded message. They claim that fountain codes afford the communication redundancy, as it can combat packet loss. They then discuss possible ways to increase the likelihood of successful message reconstruction.

MacKay (2005) suggests that Fountain Codes could be successfully and effectively implemented in use cases involving long-term data storage and data broadcasting. If a long-term storage solution, such as magnetic tapes, were to fail, certain data could be lost forever. MacKay suggests that spraying encoded data all over a storage device could allow for data recovery, if only a portion of the encoded data could be recovered. Similarly, data could be lost in data broadcasting, such as video streaming. If the data is encoded using fountain codes, whenever a packet is lost, the receiver could simply reconstruct that piece of data, allowing for a seamless end user experience.

There are multiple types of Fountain Codes. Qureshi et al. (2014) wrote about Reed-Solomon (RS) erasure codes, which can be used to reconstruct data which was lost. Qureshi also discusses Raptor Codes and Luby Transform (LT) codes, which are less computationally demanding than RS codes. This is why we decided to implement LT codes in our application.

Brar and Sandhu (2016) found that using RS codes before using raptor codes allowed for ratelessness and forward error correction. They found that this implementation is useful for noisy communication channels and would reduce redundant transmission.

MacKay (2005) reviews random linear fountain codes, as well as Raptor Codes and LT codes.

Du (2012) discusses LT codes in relation to soliton distributions, and a degree distribution with only one parameter. This is a more efficient implementation of LT codes.

Prior research has also demonstrated the computational efficiencies of LT codes compared to other erasure codes such as Reed-Solomon (Chang et al., 2008).

**Overview of Project**

For this research project a real-time chat application was developed to allow users to send messages by using a Lubey transform fountain code. Encoding is performed on the client as a and transmitted the chat server. The front end on the client consists of a command line application that allows the user to send messages through standard TCP traffic to a central chat server. The chat server broadcasts the encoded message to all other clients connected to the same chatroom. Lastly the clients decode the encoded message received from the chat server. The client initiating the message will see the plaintext however, the message typed by the user on the client is not the same as the data being transmitted. . The message is encoded via fountain codes, and the corresponding data for each fountain code token is sent to the server, broadcasted, and decoded by each individual client. Files created to aid in the data transfer are promptly deleted after a message is decoded successfully, such that the only information remaining is the plaintext sent by the user, and the decoded plaintext decoded by the clients.

**Encoding and Decoding Process**

Fountain code encoding leverages XOR Lubey transform to create tokens which get sent over the network, instead of sending plaintext.

Plaintext is broken apart into various chunks, which each contain a small portion of the complete plaintext. Using a soliton distribution and a random seed, random sets of chunks are XOR’d together, resulting in a new token. The amount of tokens generated may vary depending on the number of chunks created from the plaintext. The distribution of the number of chunks XOR’d together to create a token is also determined by the soliton distribution. It is necessary for some chunks to be sent as the plaintext, so a small percentage of tokens are equivalent to a chunk of the plaintext. This is required to decode the message.

These tokens are sent over a network instead of the plaintext chunks themselves. Upon receipt of just some of the tokens, the receiver should start trying to decode the original plaintext. Decoding is done by using XOR on both tokens and chunks of plaintext to reverse engineer the original message. Because the content of the tokens rely on XOR’ing various chunks of plaintext, the receiver could deduce pieces of the plaintext by XOR’ing them the same way that the tokens were calculated. XOR is a symmetric function, which allows the receiver to do this. This allows for a waterfall effect to occur, where XOR’ing two values may reveal a chunk of plaintext, and using that new value, the receiver can decode another chunk of plaintext, and this could repeat until the entire plaintext is recovered.

Fountain Codes are very effective at transmitting data over networks where packets are likely to be dropped or lost, because the receiver could theoretically reconstruct the original data, as long as enough tokens make it to the receiver.

Unfortunately, enough tokens simply making it to the receiver isn’t always enough. The correct tokens which allow for the complete reconstruction of the data would need to be received, which isn’t guaranteed. In this situation, however, the sender could simply retransmit tokens until the complete message is reconstructed on the receiving end. This means that Fountain Codes operate best as a stream of data, constantly sending tokens over the network, until the receiver has enough information to reconstruct the original data.

Our implementation of fountain codes sends all tokens at once, and we assume that no data gets blocked or dropped.

The chat server implementation involves basic TCP traffic. A simple chatroom was developed using python and allows up to 5 clients to connect to a central server. For each message sent to the server, the server reflects that data and broadcasts it to each connected client. In effect, the clients are communicating with each other, as the server doesn’t save any data corresponding to any messages. The clients also aren’t aware of who is receiving their messages. Therefore, if a client connects to the server illegitimately or maliciously, minimal information can be collected of the other clients they are sharing messages with. This does allow for a malicious client to receive files illegitimately, which will be discussed later.

As an example, assume two clients, A and B, had connected to the server. If client A attempts to send the string, “Hello World”, it would be encoded on client A’s machine before being transmitted. The process of encoding the message on the client is presented in Figure 1.

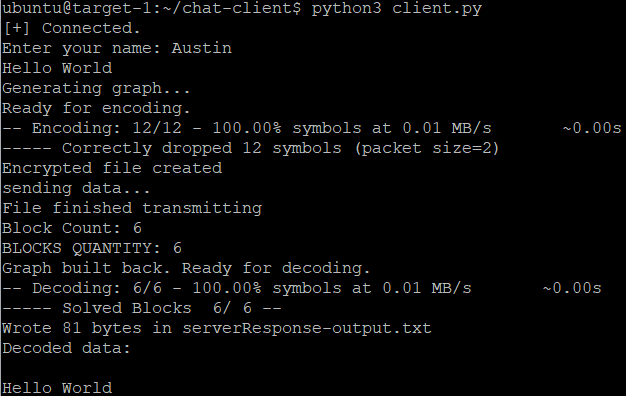


Figure 1: An example output for a client that sends the phrase “Hello World”

The encoding is done on client A’s machine, because it will send the data. Once a message is input by the user, a text file is created on client A’s machine containing the inputted string. In this case, the text file would read, “Hello World”. An encoding script is run on that file, which creates an encoded file. The encoded data is represented as a text file, where each line of data corresponds to a token that client A will send over the network.

Figure 2 shows a theoretical breakdown of how the string “Hello World” may be encoded. The tokens at the top would be calculated by XOR’ing certain chunks, designated by m*X*, where *X* is the index of the message chunk. The arrows show which tokens can be used to recalculate the original plaintext chunk. For example, token 0 was calculated by XOR’ing m2 and m4. It can also be seen that t10 is only connected to m10. This designates that token 10 is a piece of plaintext which would be transmitted as is, which is required to begin the decoding process.

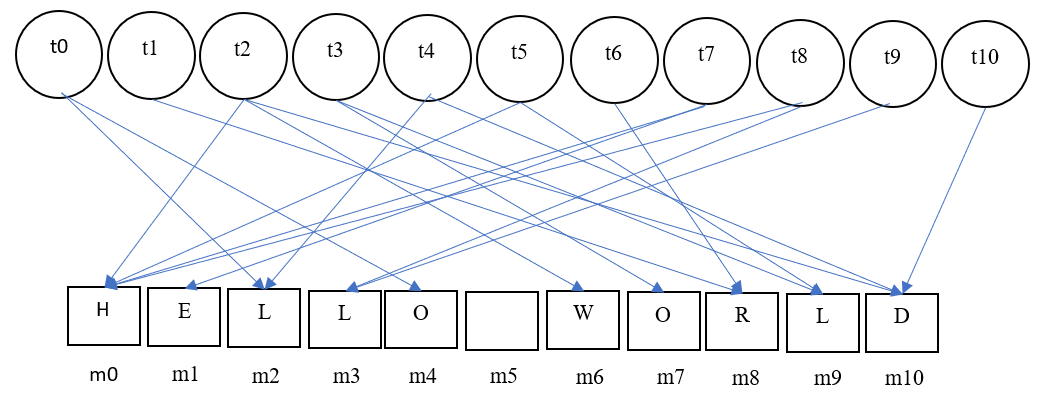


Figure 2: A simplified example of how the string “Hello World” may be encoded on client A.

Figure 3 shows a matrix used to represent the potential encoded data. The number of messages to XOR in each row is determined by the soliton distribution. In this case, c4 is calculated by XOR’ing m3 and m5. Furthermore, c1 is just plaintext. Specifically, c1 is equivalent to m3.

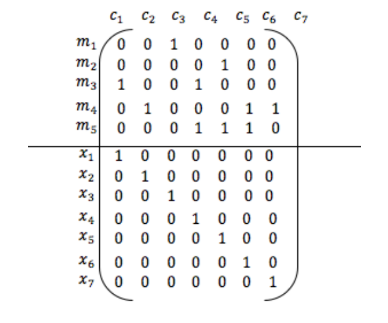


Figure 3: An example of matrices representing possible encoding (Brar and Sandhu, 2016)

Figure 4 is an actual instance illustrating the encoded data. Each column represents a token’s index, degree, and the data saved in the token, respectively. The other values are the number of chunks in the original plaintext and how many bytes comprise the original message, respectively.

In our application, we use a chunk size of two, so that when sending the string “Hello World”, the chunks of plaintext used for decoding are “He”, “ll”, etc. This is why the number of chunks and the number of bytes is not equal.

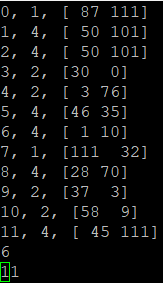


Figure 4: An example of how the tokens are represented in our chat room application.

This encoded message, in Figure 4, is then sent to the server as a text file, where the server reflects this message to all clients connected to it, including the client which sent the message. In this case, both client A and client B would receive the encoded data reflected by the server. The server does not save any data related to the contents of the message.

Each individual client, upon receiving a text file representing the encoded tokens, will run the decoding script on that data. This script creates a new output text file on each client containing the decoded plaintext. The output file and input file should be the same. In this case, they should both read “Hello World”. On client A, there would be both an input and an output file, but on client B, there will only be the output file.

**Decoding**

To decode chunks of plaintext, first the script reads the data from any token without dependencies. In those cases, there is no XOR’ing required. Once the script knows a chunk of original plaintext, it can use that to decode other tokens or other plaintext chunks. For every set of dependencies, once there is only one unknown variable, the script will XOR the rest to infer the value of the unknown. For example, if plaintext chunk X can be created by XOR’ing tokens Y and Z, but both X and Y are known, that leaves Z as the unknown. Because XOR is a symmetric function, XOR’ing X and Y will calculate Z. Once we know the value of Z, we can look for another set of dependencies with only one unknown variable, and repeat until we have recovered the full plaintext.

The decoded plaintext is output to each client which decoded it, allowing the users to see the plaintext originally sent by another client. This allows client B to output the data originally sent by client A, as if it was simply transmitted in plaintext. The intermediate encoded files are removed from both clients once the message is decoded, so that the encoded tokens are not saved for each message. To the end user, the message is transmitted over the network seamlessly, without leaving remnants of the use of Fountain Codes. The end user is only aware of the message they send/receive, which would both be “Hello World”.

Client A could send another message, or Client B can respond with their own message. Regardless, the next message sent over the network would be encoded, transmitted and decoded in the same manner. The encoding and decoding process is depicted in Figure 5.

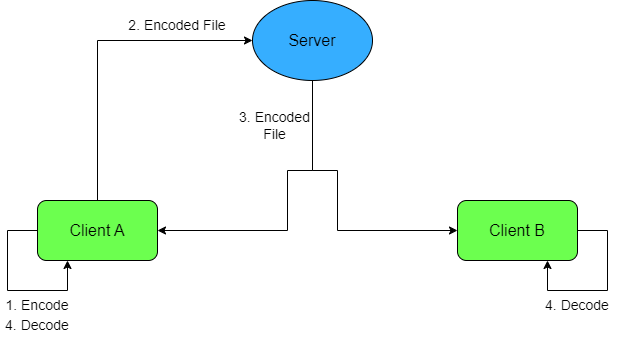


Figure 4: A flowchart showing the transmission process with Fountain Code encoding.

**Limitations**

The current stage of the proof of concept has several limitations. First the application does not ensure the integrity of the transmitted file. If the file is properly encoded and properly decoded, it’s expected that the input and output files would share the same hash. Currently, the application does not check any hashes, and therefore, if a message is improperly decoded, the receiver would not be aware.

In cases where the character length of a message is divisible by the packet size used to break down the plaintext, the decoding process may fail to correctly decode the message. The last packet size’s worth of characters are omitted from the result. This is why we believe that our implementation, although effective in a chatroom, would be better used for file sharing. Short messages, common in chat rooms, require the use of small packet sizes, so that they can be broken down and XOR’d into sufficiently many tokens. File sharing, however, could allow for larger packet sizes, and therefore, it would be less likely for the file to be truncated upon transmission.

In these cases, as well, sometimes, the file cannot be decoded. The pseudorandom values which determine the distribution of dependencies may not allow the decoder to fully decode the message. If the distribution relies on the final piece of plaintext being received and decoded, if the encoding or decoding algorithm neglects it, the message will not be decoded. This behavior is seen in the application, when the message is of a particular length.

Because the clients aren’t given information about what other clients they are sending messages to, it would be possible for a user to maliciously listen in on conversations among legitimate users. All the attacker would have to do is gain access to the server, and then wait for information to be broadcasted.

**Future Research**

Future work should implement hash checking as an important addition to the application. Verifying the integrity of data after fountain code encoding is necessary for any implementation outside of a lab setting. It would also be useful in ensuring that no output is truncated.

Fixing the bug involving truncated output would also be necessary for any implementation outside of a lab setting. If the output would be truncated,

If the ciphertext could not be fully decoded, I suggest re-encoding the plaintext and retransmitting the encoded file, as a new random number generation seed while encoding may allow the ciphertext to be decoded.

I also suggest giving the user the option to send data via Fountain Codes vs sending plaintext to investigate the benefits of Fountain Codes in regards to network censorship. I believe that our current implementation would allow data to be transmitted via Fountain Codes which would otherwise be dropped if detected in plaintext. We were not able to test this, however. This should be treated as a new area of research.

Similarly, giving the user an option to transmit data via Covert Channels would allow for greater versatility of the application. Perhaps the user would opt to use Covert Channels if they would like to send a short message, as they have a slow transmission rate, whereas the user would use Fountain Codes for large amounts of data. If it were to be implemented this way, the user would be able to communicate securely using this chatroom for a variety of purposes. They could share large files with a few peers, or have discussions among themselves, without running the risk of their message being truncated simply due to the length of the message.

It would be great if an Intrusion Prevention/Detection System could be implemented, such as Snort, to display when certain messages are sent over the network in plaintext. Perhaps it would drop that packet, or it would log the event, so that we could see that a message was sent which should have been censored. If this were implemented, it would be very easy to show that when sent using Fountain Code encoding, the network censorship gets circumvented. This gives legitimacy to the idea that fountain codes could be very useful in cases where networks are monitored.

**Conclusion**

Lossy networks are more common in wireless and cellular communication, and therefore, those types of networks are more susceptible to corrupting data and redundant transmission. Data corruption can be detrimental, and redundant transmission places unnecessary stress on a network. These cases should be avoided, if possible. We have implemented a chat server which uses Fountain Code encoding to prevent data corruption or unnecessary retransmission due to a lossy network. Our work could be expanded to demonstrate a potential method to circumvent censorship, as well. We were able to successfully implement Fountain Code encoding and decoding on multiple clients, which communicate with each other, once connected to a central server. We’ve found that if all Fountain Code tokens are sent over the network at once, messages can be properly decoded and displayed to the user.

**Bibliography**

Azab, A., Khasawneh, M., Alrabaee, S., Choo, K.-K. R., & Sarsour, M. (2022). Network traffic classification: Techniques, datasets, and challenges. Digital Communications and Networks. [https://doi.org/10.1016/j.dcan.2022.09.009](about:blank)

Brar, A. S., & Sandhu, A. S. (2016). Design and Implementation of Enhanced Model of Fountain Codes for Unicast Transmission. Indian Journal of Science and Technology.

Du, Qingxiao. “Implementation of lt code with A novel degree distribution.” Open Journal of Applied Sciences, vol. 02, no. 04, 2012, pp. 203–207, [https://doi.org/10.4236/ojapps.2012.24b046](about:blank).

Chang, S. K., Yang, K. C., & Wang, J. S. (2008, May). Unequal-protected LT code for layered video streaming. In *2008 IEEE International Conference on Communications* (pp. 500-504). IEEE.Hourt, N., Kar, K., & Ganguly, B. (2013, November). Performance of loss-tolerant TCP (LT-TCP) in the presence of correlated losses. In *MILCOM 2013-2013 IEEE Military Communications Conference* (pp. 1341-1346). IEEE.

Hourt, N., Kar, K., & Ganguly, B. (2013, November). Performance of loss-tolerant TCP (LT-TCP) in the presence of correlated losses. In *MILCOM 2013-2013 IEEE Military Communications Conference* (pp. 1341-1346). IEEE.

MacKay, D. J. (2005). Fountain codes. IEE Proceedings-Communications, 152(6), 1062-1068.

MacKay, D. J. C. (2005). Capacity Approaching Codes Design and Implementation. Fountain Codes, IEE Proc-Commun, 152(6).

Marciniszyn, E. (2022). Fountain codes and covert channels (Doctoral dissertation, Master’s thesis, FernUniversität in Hagen, Germany).

Qureshi, J., Heng Foh, C., & Cai, J. (2014). Primer and recent developments on Fountain Codes. Recent Advances in Communications and Networking Technology, 2(1), 2–11. [https://doi.org/10.2174/22117407112019990001](about:blank)

Singh Brar, Anmol, and Amandeep Singh Sandhu. “Design and implementation of enhanced model of fountain codes for unicast transmission.” Indian Journal of Science and Technology, vol. 9, no. 19, 31 May 2016, [https://doi.org/10.17485/ijst/2016/v9i19/92876](about:blank).

Spriteware. (n.d.). Spriteware/Lt-Codes-Python: Fountain code: Efficient python implementation of lt codes. GitHub. [https://github.com/Spriteware/lt-codes-python](about:blank)

Tian, J., Xiong, G., Li, Z., & Gou, G. (2020). A survey of key technologies for Constructing Network Covert Channel. Security and Communication Networks, 2020, 1–20. [https://doi.org/10.1155/2020/8892896](about:blank)