Topic 1: A Better UDP

Carson Clanton
Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, Texas 76019
clanton@uta.edu

Kevin Hayes
Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, Texas 76019
kevin.hayes@mavs.uta.edu

William Hunter
Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, Texas 76019
whunter@mavs.uta.edu

Richard Meth
Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, Texas 76019
richard.meth@mavs.uta.edu

Ryan Nordeen
Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, Texas 76019
ryan.nordeen@mavs.uta.edu

***Abstract*—This article describes an approach to improving the reliability of UDP. Our approach uses sequence numbers for packets to force in-order delivery to the Network Application layer and Error-Correcting Codes to help reproduce lost packets without requesting retransmission of these missing packets. The level of reproducibility will be largely dependent on the Error-Correcting Code chosen and the selection of its associcated redundancy parameters.**

I. Introduction

When a software engineer begins designing a network application, he or she has to make a decision between two transport layer options, TCP or UDP [2]. There are advantages and disadvantages to both. With UDP comes nimbleness and low overhead, along with dropped and possibly out-of-sequence packets. With TCP comes reliable stream-based communication at the expense of throughput. There isn't much middle ground between the two. The approach discussed in this paper attempts to change that.

We don't attempt or intend to redesign either protocol. Instead, we seek to enhance the reliability of UDP, so that it is a little more like TCP, while remaining connectionless. To do this, we will require two added features to be built "on top" of UDP: Sequence Numbers and Error-Correcting Codes.

The introduction of sequence numbers should be somewhat obvious; on the receiving end of the UDP pathway there should be someway to put packets back in order. The subtlety, however, of requiring sequence numbers is that in order for our error-correcting codes approach to work, for the code we experimented with, at least 75% of the packet payload data must be available and properly sequenced. So, the inclusion of sequence numbers is twofold.

The second feature to be added, Error-Correcting Codes or ECC, is a little more complicated and requires more of an introduction, if for no other reason than to get acclimated to the terminology in the literature.

Given the no-guaranty and low-reliability characteristics of the networking layer and the little added value of UDP, a virtual communication link through a UDP socket represents an unreliable, and potentially noisy, channel, making it a good candidate for a ECC solution.

A. Error-Correcting Codes

Below in Figure 1 is a schematic of the general communication link for sending messages [1]. Here the message is first sent by the *source* to the *encoder* where the message is assigned a *codeword*, i.e. a string of characters from some chosen alphabet. The encoded message is sent through a *channel*, which is certain to have some level of noise that may possibly alter the codeword. This possibly-altered codeword is then received by the *decoder* where it is matched with its most likely candidate codeword which is in turn translated into a message resembling the original message and passed on to the *receiver*. The degree of resemblance to the original message depends on how appropriate the code is in relation to the channel.

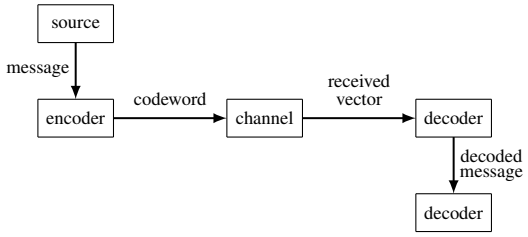


Fig. 1: Schematic of a general communication link

Example I.1. Suppose we are using an alphabet containing only two symbols, say 0 and 1 and we wish to send a message to a friend of either "YES" or "NO". We first might consider encoding 1 for "YES" and 0 for "NO"; however, if the codeword 0 is sent and then altered by the channel to be received as 1, the decoder will incorrectly pass the message "YES" to the receiver. One way of improving the situation is to add redundancy by repeating the symbol. For instance, we might encode "YES" as the codeword 11 and "NO" as 00. Now if "NO" is sent, two errors would have to occur before the decoder returns an incorrect message. If one error occurs then the received vector will either be 10 or 01, neither of which is a codeword. At this point, the receiver might request a retransmission. This is an example of a code in which one error may be detected. To strengthen the code to be error-correcting we might increase the redundancy by repeating the message five times. Thus we encode our message "NO" as 00000. Perhaps the channels interferes to cause the decoder to receive the vector 00110. Using the method of nearest neighbor decoding the decoder assesses the message

and decides that of the two possible codewords (i.e. 00000 and 11111) 00000 is more likely the one originally sent and hence is correctly decoded as "NO".

Definition I.2. Let $\mathbf{F}$ be a finite set, or **alphabet**, of q elements. A q-**ary code** $\mathbf{C}$ is a set of finite sequences of symbols of $\mathbf{F}$, called **codewords** and written $x_1x_2\cdots x_n$, or $(x_1, x_2, \ldots, x_n)$, where $x_i \in \mathbf{F}$ for $i = 1, \ldots, n$. If all the sequences have the same length n, then $\mathbf{C}$ is a **block code** of **block length** n.

Given an alphabet $\mathbf{F}$, it is consistent with terminology for vector spaces when $\mathbf{F}$ is a field to denote the set of all sequences of length n of elements of $\mathbf{F}$ by $\mathbf{F}^n$ and to call these sequences vectors. The member of $\mathbf{F}$ at the ith position of a vector is known as the coordinate at i.

Definition I.3. Let $v = (v_1, v_2, \ldots, v_n)$ and $w = (w_1, w_2, \ldots, w_n)$ be two vectors in $\mathbf{F}^n$. The **Hamming distance**, $d(v, w)$, between v and w is the number of coordinate places in which they differ:

$$d(v, w) = \{i | v_i \neq w_i\}$$

We will usually refer to the Hamming distance as simply the **distance** between two vectors.

Nearest-neighbor decoding is a method in which the received vector is translated as the codeword of smallest distance, whenever it is uniquely determined. This method maximizes the likelihood of correcting errors provide the two assumptions mentioned above hold (i.e. probability of an error $< 1/2$ and each symbol is equally likely to be transmitted).

Definition I.4. The **minimum distance** $d(\mathbf{C})$ of a code $\mathbf{C}$ is the smallest of the distances between distinct codewords; i.e.

$$d(\mathbf{C}) = \min\{d(v, w) | v, w \in \mathbf{C}, v \neq w\}$$

Example I.5. The error-correcting code in Example I.1 is a binary code of block length 5. The distance between the codeword 00000 and the received vector 00110 is 2. The distance between the same received vector and the codeword 11111 is 3. Thus, 00000 is the nearest neighboring codeword. The minimum distance of this code is 5.

Definition I.6. A code $\mathbf{C}$ over $\mathbf{F}$ of length n is **linear** if $\mathbf{C}$ is a subspace of the vector space $V = \mathbf{F}^n$. If $\dim(\mathbf{C}) = k$ and $d(\mathbf{C}) = d$, then we write $[n, k, d]$ for the q-ary code $\mathbf{C}$; if the minimum distance is not specified we simply write $[n, k]$. The **information rate** is k/n and the **redundancy** is $n - k$.

Since $\mathbf{C}$ is a subspace, it follows that the **all-zero** vector must be a codeword in $\mathbf{C}$ also the sum of any two codewords must also be a codeword in $\mathbf{C}$. We will also see after the next definition that the distance d is also easy to calculate for linear codes.

Definition I.7. Let $V = \mathbf{F}^n$. For any vector $v = (v_1, v_2, \ldots, v_n) \in V$ set

$$S = \{i | v_i \neq 0\}$$

Then S is called the support of v and the weight of v is $|S|$. The **minimum weight** of a code is the minimum of the weights of the nonzero codewords.

Example I.8. Listed below are all 16 codewords in the linear $[7, 4, 3]$ binary code. They are listed in order of increasing weight.

0000000	0111000	1000111	0101101
1100100	0001110	1101010	0110110
1010010	0010101	1110001	0011011
1001001	0100011	1011100	1111111

Notice that the minimum weight is 3. By Theorem 1.1 it follows that $d(\mathbf{C}) = 3$. It can be shown that these 16 codewords form a subspace of $\mathbf{F}^7$ with basis $\{(1, 0, 0, 0, 1, 1, 1), (0, 1, 0, 0, 0, 1, 1), (0, 0, 1, 0, 1, 0, 1), (0, 0, 0, 1, 1, 1, 0)\}$. Thus $\dim(\mathbf{C}) = 4$ as conveyed in the notation and the redundancy is 3 since $n - k = 7 - 4 = 3$.

Theorem I.9. *Let $\mathbf{C}$ be a code of minimum distance d. If $d \geq 2t + 1$, then $\mathbf{C}$ can be used to correct up to t errors.*

II. Design

A Better UDP is "wedged in" as another IP stack layer sitting between the transport layer and the application layer, see Figure 2. We did not modify any kernel source and A Better UDP is not a standard yet, so to use it an application developer or software engineer will have to include this functionality as a library. Instead of using a socket library to interface with the transport layer, the engineer will link with A Better UDP library to interface with the protocol stack below the application layer. Henceforth we will refer to A Better UDP as BUDP.

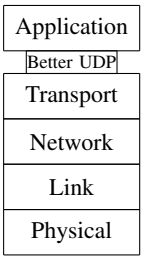


Fig. 2: Five-layer Internet Protocol Stack with a Better UDP Wedged in

Messages, from the application layer at the source, are passed to BUDP, where, currently, four at a time are buffered, assigned sequence numbers, encoded to produce an ECC, and then sent as datagrams via standard UDP to their destination address. At the destination, the datagrams are collected and stored in an accumulation buffer, where they are placed in their proper sequence, see Figure 3. The purpose of buffering is to support the process of encoding the ECC.

One of three conditions, prompts their delivery to the application layer at the destination:

1) All four datagrams are received intact.
2) Three datagrams are received as well as the ECC, intact.
3) A countdown timer reaches zero.

The first of these conditions is obvious, while the other two require further explanation. It is easier to explain 3) first, so we will do that now. As stated previously in our introduction, we don't intend to rewrite TCP and we would like to keep our UDP implementation connectionless, so instead of requesting retransmission of missing datagrams, for example, when the one of the first two conditions above are not met, our implementation will simply give what it has to the application layer at the destination. Our justification for doing this is that it is what a standard UDP implementation would do. Actually, a standard implementation of UDP would do less, because our implementation will at least give the application layer in-order messages.

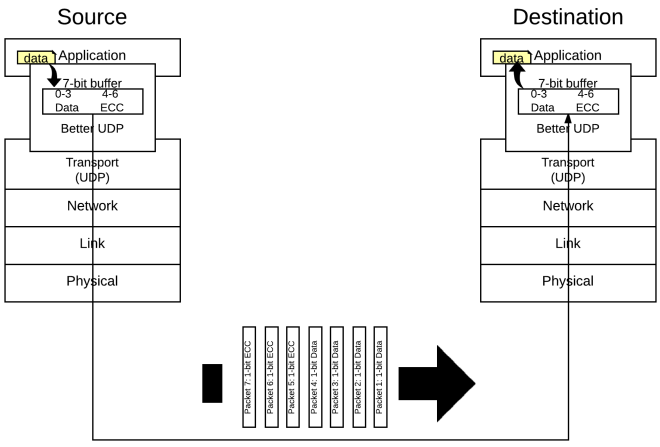


Fig. 3: A Better UDP Overview Schematic

The second condition which can prompt message delivery to the application layer (i.e. condition 2) above) requires the use of Error-Correcting Codes. For the purposes of prototyping our design, we used the single error-correcting Hamming code introduced above. In general, a more robust and scalable ECC, like Reed-Solomon which is discussed in RFC5510 [3], should be used, as the code we used is quite limited and so we had to keep our message size unrealistically small, 1-bit. For our design the [7,4,3] Hamming code serves as a placeholder until Reed-Solomon can be integrated. Nonetheless, the use of the Hamming code will allow us to benchmark our design against both standard TCP and standard UDP.

Encoding the message stream, using four messages at a time, produces a 3-bit ECC, which is essentially a concatenation onto the four data bits coming from the source application layer. These ECC bits are sent in three follow-on datagrams to the destination. These datagrams also contain sequence numbers, which are a continuation of the sequence numbers assigned to data/message datagrams. To use the [7,4,3] Hamming code, there are two very important matrices required: the *generator matrix* G

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

and the *parity-check matrix* H

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

At the source, to encode a message m, which for purposes of the math is represented as a row vector in $\mathbf{F}_2^4$, multiply it on the right by G to get mG, which is a row vector in $\mathbf{F}_2^7$, where the first four coordinates are m and the last 3 coordinates are the parity-check bits, or (as we have been referring to them) the "ECC".

At the destination, to decode the received vector $r \in \mathbf{F}_2^7$, mutliply it on the right by H^{T} to get rH^{T}, which is a row vector in $\mathbf{F}_2^3$ known as the syndrome. Next, syndrome decoding is performed whereby a *coset leader* e is matched with the syndrome rH. This coset leader represents the error vector between the received vector r and the closet codeword c to it. To determine c simply add r and e modulo 2. That is,

$$c = r + e \mod 2$$

Figure 4 shows a simplified diagram of the ECC encoding scheme we implemented, where 'M' stands for message and the number is the datagram sequence number. There are actually two levels of ECCs that we envisioned using, but due to the limited time we had to spend on this project we were not able to get the full implementation finished. The level we implemented, which is show in Figure, is the so-called outer or stream level ECC encoding and requires having the payload data in buffer which is treated as a vector in $\mathbf{F}^7$ in the mathematics of the encoding process. The second level is the so-called inner or message level ECC encoding. This inner ECC would allow for checking and correcting datagram integrity. That is, if a data were received at the destination for found to be corrupted, it could be "fixed". This is something that would be much more possible with a Reed-Solomon encoding scheme. Figure 5 shows how the two level ECC scheme would look. Although, to make a scheme like this work the inner ECC has to have almost as bits as there are data bits. The reason for this is because in the case of a lost

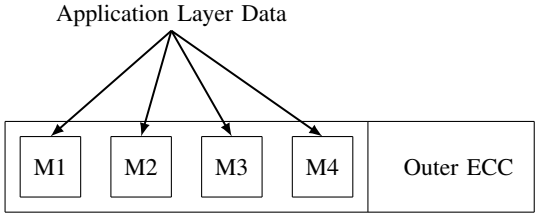


Fig. 4: Simplified ECC Encoding Scheme

packet the outer ECC will have to regenerate the missing inner ECC. Once this inner ECC is regenerated, it will then be used to regenerate the missing data payload (from scratch). The amount of redundancy in the ECC will be roughly equal the size of the data. Our initial thought on this is that it will force a fixed-size payload, so that the size of the inner ECC can be pre-determined, but once this two-level scheme is prototyped we may find this isn't the case.

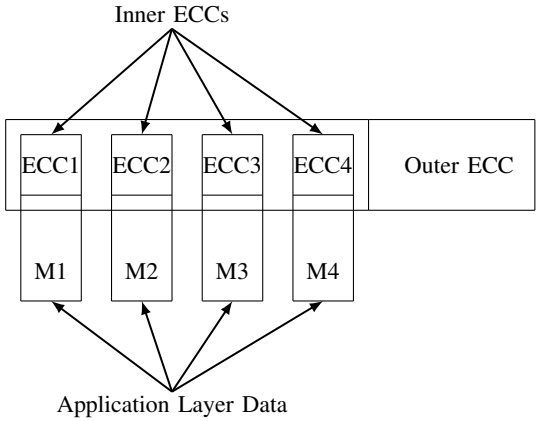
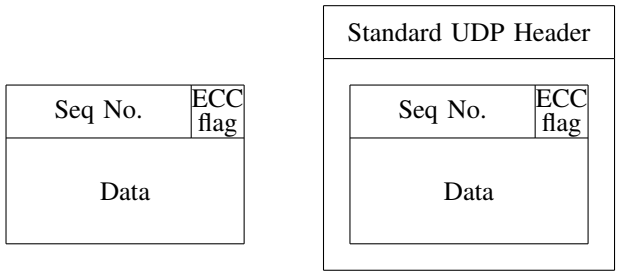


Fig. 5: Two-level ECC Encoding Scheme

Our BUDP datagram will be encapsulated in an ordinary UDP datagram. More specifically, BUDP will contain a header including a field for the sequence number and a field for an ECC flag, indicating whether the payload holds an ECC or data. This BUDP datagram (BUDP header + BUDP payload), will form the payload of a regular UDP datagram. See Figure 6 below.



(a) BUDP Datagram (b) BUDP encapsulated in UDP

Fig. 6: BUDP Encapsulation

III. Results

We expected our our implementation to perform at about half the throughput of stock UDP, due to the fact that there nearly as many redundancy bits as there are data bits. What we were curious to see is how well it fared compared to standard TCP.

To run these benchmarks, we forced packets to be drop via....

IV. Conclusion

We have presented a design for a better UDP implementation where datagrams arrive in-order to the application layer at the destination and when a datagram is lost it can be reconstructed, provided 75% of the datagrams in its ECC grouping and the ECC arrive intact. If these latter two conditions are not met, then BUDP simply hands over what it does have to the application layer after a countdown clock expires.

Our results indicate that....

In the future, we would like to integrate a more robust Error-Correcting Code such as Reed-Solomon as documented in RFC5510. Futhermore, we would like to implement the two-level ECC encoding scheme to build in more redundancy.

Our view is in order to deal with the unreliability inherent in the Internet Protocol, a designer has to either build in redundancy to help clean the communication channel or (s)he has to build in a retransmission infastructure, as is the case with TCP. We prefer the error-correcting redundancy.... or we did the redundancy since it isn't available otherwise, etc. etc.

Acknowledgment

The Error-Correct Codes subsection of the Introduction was taken, in part, from a presentation one of the co-authors presented at a Mathematical Association of America Texas Section conference in April 2005 [1] focusing on its connection to Finite Geometries. It was included here to help demystify some of the math related to the [7,4,3] Hamming code used in our prototype. All other sections and subsections are original. This is the first time any of these authors have used Error-Correcting Codes to improve a transport protocol.

References

[1] Carson B Clanton. Use of finite geometries in error-correcting codes. Mathematical Association of America Texas Section, 2005.

[2] James F Kurose and Keith W Ross. *Computer networking*. Pearson Education, 2012.

[3] Jérôme Lacan, Vincent Roca, Jani Peltotalo, and Sami Peltotalo. Reed-solomon forward error correction (fec) schemes. Institute of Electrical and Electronics Engineers, 2009.