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 datagrams to force in-order delivery to the destination network application layer and error-correcting codes to help reproduce lost datagrams without requesting retransmission of these missing datagrams. The level of data reconstruction will be largely dependent on the error-correcting code chosen and the selection of its associated 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 [1]. There are advantages and disadvantages to both. With UDP comes nimbleness and low overhead, along with dropped and possibly out-of-sequence datagrams. With TCP comes reliable connection-oriented stream-based communication at the expense of throughput [1]. 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 quick and 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 datagrams 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 datagram payload data must be available and properly sequenced. So, the inclusion of sequence numbers is really 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 solution based on error-correcting codes.

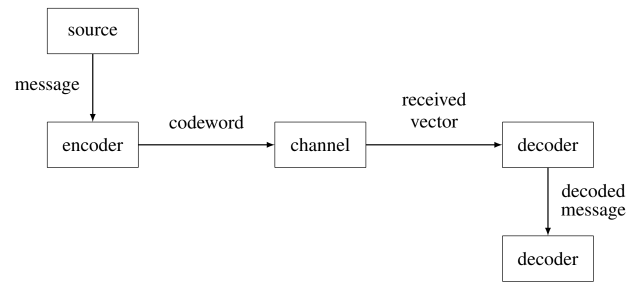


Fig. 1: Block diagram of a general communication link

A. Error-Correcting Codes

Above in Figure 1 is a block diagram of a general communication link for sending messages [2]. 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.

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 remainder of the protocol stack which sits below the application layer. Henceforth we will refer to A Better UDP as BUDP.

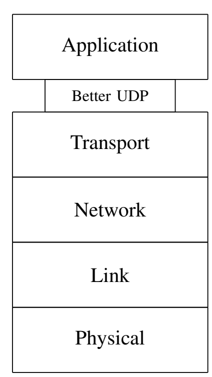


Fig. 2: Five-layer Internet Protocol Stack with a Better UDP Wedged in

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 at the source is to support the process of encoding the ECC. At the destination, it is a requisite part of the process since datagrams, presumably, arrive one-at-a-time and the decoding process cannot begin until a sufficient amount of the block data has been received. Here, we say block data, because as we will soon see datagrams have to be encoded in groups which we refer to, at various places in the source code, as blocks, for “blocks of datagrams”.

One of three conditions, prompts the delivery of messages 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 expires.

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 is 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.

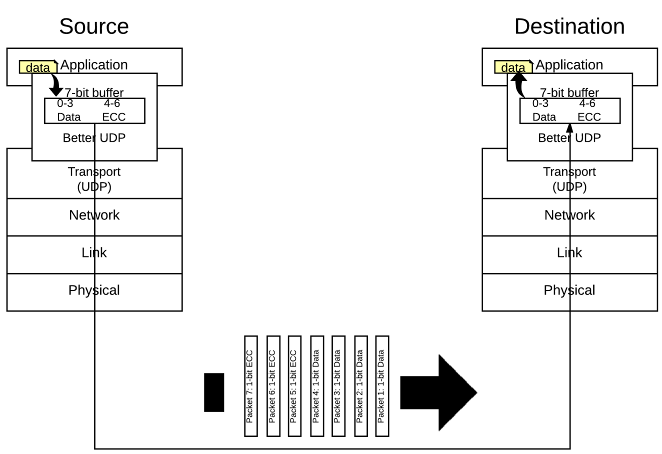


Fig. 3: A Better UDP Overview Diagram

The second condition which can prompt message delivery to the application layer requires the use of error-correcting codes. For the purposes of prototyping our design, we used the single error-correcting Hamming code introduced above which allowed us to benchmark our implementation against both standard TCP and standard UDP. Our purposes for using this code over the Reed-Solomon code suggested in RFC5510 [3] were four-fold:

1) It is simple to implement and suits our purposes very well.
2) One of our co-author was very familiar with it and its connection to Finite Geometry.
3) The [7,4,3] Hamming code is a perfect single error-correcting code, which happens to be the smallest perfect code, and is generated by a finite geometry known as the Fano Plane which is the smallest projective plane [2].
4) Looking long term, we perhaps want to move our implementation to the newer so-called “Low-Density Parity-Check Codes” which have been beating some of the more recent ECC frontrunners such as turbo codes in competitions [4]. Some of these codes can be generated from Finite Geometries, and those that are “can be decoded in various ways, ranging from low to high decoding complexity and from reasonably good to very good performance” [4].

A. Encoding and Decoding

Encoding the message stream, using four messages at a time, produces a 3 ECC messages which are sent seperately as follow-on datagrams. So, at an overly simplified level, the message stream, grouped in blocks of seven, flows like so:

$$\cdots ddddpppddddppp \cdots$$

where d is a data symbol from the source application, and p is a parity-check symbol associated with the four most recent d symbols prior to it. This represents an increase of 75% in the number of datagrams transmitted. These ECC or parity-check datagrams also contain sequence numbers, which are a continuation of the sequence numbers assigned to data messages. These sequence numbers are used in the identification of these datagrams as containing ECCs. They are also used to identify with which data-containing datagrams they are associated.

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

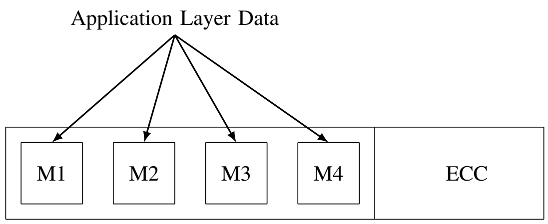


Fig. 4: Simplified ECC Encoding Scheme

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”. In RFC 5510 [3], these bits are called repair symbols.

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 which is the error between the received vector r and the closest neighboring codeword c to it. To determine c simply add r and e modulo 2. That is,

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

B. Encoding Scheme

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 considered using, but decided against this due to the unnecessary redundancy we felt it added. This two-level scheme is shown in Figure 5. The level we implemented in the one-level approach turns out to be an outer ECC in the two-level approach as shown in Figure 5. This two level scheme is very similar to the way CDs and DVDs are encoded with error-correcting redundancy [4].

The second level is the so-called inner or message level ECC encoding. This inner ECC would allow for checking and rectifying datagram integrity. That is, if a datagram were received at the destination and found to be corrupted, it could be “fixed”. We felt this redundancy would never fulfill its purpose since stock UDP already provides a checksumming mechanism that potentially discards the datagram in the case where data corruption is detected.

Besides, to make a scheme like this work, at least for the layer approach in the figure, the inner ECC would have to have almost as many parity bits as there are data bits in the application layer message. The reason for this is that in the case of a lost datagram the outer ECC will have to regenerate the missing inner ECC, which, in turn, would have to regenerate the application layer message 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.

With all the header overhead already inherent in the Internet Protocol plus adding a sequence number and stream-level error-correction codes, the actual application data might end up only accounting for 20-40% of the data transmitted, depending on how the application data is feed to the DUBP layer. Adding a second level of ECCs would only exacerbate this condition and provide little added benefit.

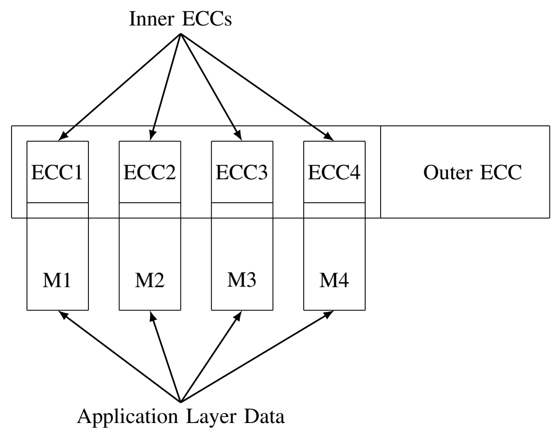
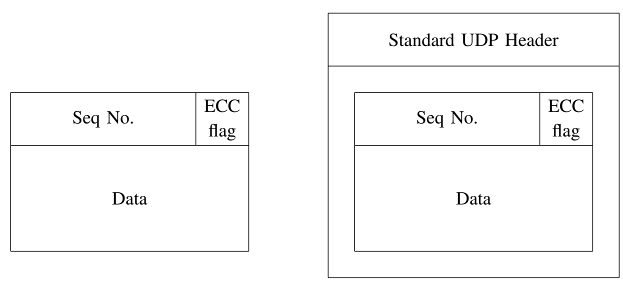


Fig. 5: Two-level ECC Encoding Scheme

C. Encapsulation

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 coded are design in standard C on a Linux Ubuntu 12.04 system. We implemented the receive side of our design as a state machine to deal with reordering of datagrams and the countdown timer. The error-correcting code detailed above was fully functional and verified to be correcting 1 byte erasure errors when that byte was the only byte missing in its datagram block.

We transmitted a text file over better UDP, which contained the text to the ebook "A Christmas Carol" by Charles Dickens as it seemed appropriate given the season.

Getting to full test environment where we would be able to record a data transfer session through a simulated WAN with non-zero packet loss probability presented more of a challenge. Our state machine performed well for the trivial case where loss probability=0. In fact, our UDP implementation reproduce the text perfectly.

When we attempted to increase the probability to 0.01, we entered a weird tracking state in our BUDP benchmark where it thought it already had unreceived data in the receive-side data buffer. It seems to be an indexing problem, that we never could fully debug.

We expected our our implementation to perform at $4/7$ the throughput of stock UDP, due to the fact that there were 75% more bits for redundancy. Said another way, we expected our implement to operate at this multiple of stock UDP, because $4/7$ is the information rate of the code we chose to implement. We attempted to verify this for the trivial case where there is zero packet loss by using the Linux `time` command. We aren't really sure what to make of the results as in terms of system BUDP operated at a 0.854 multiple of stock UDP. However, in terms of real time BUDP operated at a 0.181 multiple. Clearly, not much can be gleaned from these numbers. Interestingly, the average of these (i.e. 0.518) is closer to the number we were looking for. Surely, though, what we need is a better test environment. Performing a packet capture using Wireshark on the loopback address confirmed a multiple closer to the low end, with wireshark seeing a 0.21 multiple, which implies our implementation is 5 times slower that UDP.

What we were curious to see is how well it fared compared to standard TCP, but due to the tracking issue reported above the best we were able to do was show that with 10% packet loss, TCP was able to transmit the ebook without any data loss.

To run these benchmarks, we forced datagrams to be drop via the following Linux iptables command:

```
iptables -A INPUT -m statistic --mode random
      --probability 0.1 -j DROP
```

We confirm network packets were being dropped by pinging `localhost` and seeing about 1 on 10 ping requests drop out.

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. In this case, it essentially falls back to standard UDP implement plus the benefit of in-order datagrams.

Clearly, we need to do some more work on our implementation as the thoughput timings compared to stock UDP are less than half of what we were expecting. This is perhaps due to software computations for the correcting the missing datagrams. A hardware accelerated version is preferred.

In spite of the setback with our implementation not tracking according to theory, we feel we are on the right path with researching codes that are generated by finite geometries as codes of this nature are able to operate near the Shannon limit of the channel [4].

Also, we find it interesting that finite geometries keeps finding its way back into the discussion of optimal error-correcting codes [5] [4].

Our view is that in order to deal with the unreliability inherent in the Internet Protocol, a designer has to either build in redundancy to help clean up the communication channel or (s)he has to build in a retransmission infastructure, as is the case with TCP. We prefer the error-correcting approach as it would be the most elegant and mathematically interesting.

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

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

[3] J. Lacan, V. Roca, J. Peltotalo, and S. Peltotalo, "Reed-solomon forward error correction (fec) schemes." Institute of Electrical and Electronics Engineers, 2009.

[4] Y. Kou, S. Lin, and M. P. Fossorier, "Low-density parity-check codes based on finite geometries: a rediscovery and new results," *Information Theory, IEEE Transactions on*, vol. 47, no. 7, pp. 2711–2736, 2001.

[5] E. F. Assmus and J. D. Key, *Designs and their Codes*. Cambridge University Press, 1992, vol. 103.