ERMTP: An Eventually-Reliable Message Transmission Protocol for a Low-Bandwidth HF Environment

Huilong Huang, KD7YCO
Stephen Pink, KF1Y
Department of Computer Science
The University of Arizona

Introduction: Why Low-Bandwidth Reliability?

PSK31 [1] has developed into one of the most popular digital modes of the last few years with its relatively easy to use software combined with a very efficient use of amateur spectrum. Another reason for PSK31's popularity is that all an amateur needs to have is a general purpose computer running the PSK31 software; i.e., no special hardware is required. The limiting factor, however, in PSK31 communication is that this mode supports only unreliable communication, making keyboard-to-keyboard chat about the only feasible application. Other services besides chat such as file transfer, email, etc. need support from the underlying channel that corrects for all (or most) errors in the communication. It seems impossible to support reliable communication with the kind of coded, low-bandwidth, simplex channel provided in PSK31. For such reliable communication support, one has had to resort to modes such as Pactor II [2], which uses many times more bandwidth and needs specialized hardware to provide a reliable and reasonably efficient channel of communication.

The purpose of this paper is to present a new protocol, the Eventually-Reliable Message Transport Protocol (ERMTP), that can provide reliable communications over low-bandwidth noisy radio channels such as HF. We envision this protocol to support such applications as email and file transfer and their variations such as world wide web communication. ERMTP uses a half-duplex channel that delivers packets or cells of data to the application that has been checked for accuracy by the sender and receiver. Our current implementation of ERMTP is based on PSKCore [3] and, with some modification, can be used with current PSK31 user applications. Our intention is to create a reliable message service for the support of traditional data applications, yet retain the low-bandwidth quality of PSK31 as well as the reliance on PC sound-card-only hardware technology.

Eventually-Reliable Communications

ERMTP achieves reliability for message transmission in an "eventually-reliable" way. The basic idea is that the whole message transmission needs multiple rounds of interaction to be complete. During each round of interaction, the sender sends only the parts of the message that the sender believes the receiver has not correctly received. The receiver reports back to the sender which parts it has not received correctly. More rounds of transmission and acknowledgment ensue until finally the receiver reports that it has received all parts of the message.

The ERMTP Protocol

ERMTP is a connection-oriented protocol. Before sending a message, the sender must set up a connection with the receiver, as show in the figure below. In this stage, information such as *Message Length, Data Segment Length, Message ID*, etc, is exchanged between sender and receiver for initialization of the transmission. A timer is set at the sender after the *Request* packet is sent. The "Request" packet will be resent when the timer expires and no reply has been received. All packets carry a 3-byte *error detection code* to guarantee the correctness of the data and control information in the packet.

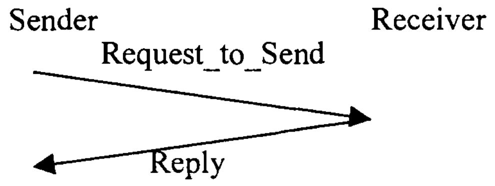


Figure 1. Connection Setup for Message Transmission

After the connection is set up, the sender starts sending the message. The message is split into multiple data *cells* i.e., packets with a fixed, equal length, except the last one, which may have less than the fixed number of bytes. Each cell carries a *sequence number* and error detection code to guarantee proper ordering and detect any errors. After all data cells of the message are sent, a *Data End* packet is sent to signal the end of data. A timer will be set when the Data End packet is sent. The Data End packet is re-sent when the timer expires and no response is received from the receiver.

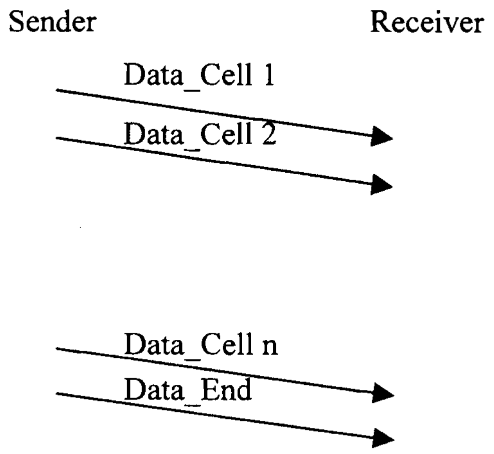


Figure 2. Split the message into multiple cells and send them separately

After receiving all the data cells and the Data End packet, the receiver side checks which cells are missing. Since the message length is known, the cells are of the same size, and each cell carries a sequence number, the missing cells are easily identified. The sequence numbers of the missing cells are then put into response cells and sent back to the sender. After all response cells are sent, a *Response End* packet is sent to signal the end of response. A timer is set after the Response End packet is sent. This packet is resent when the timer expires and no response packets have been received from the sender.

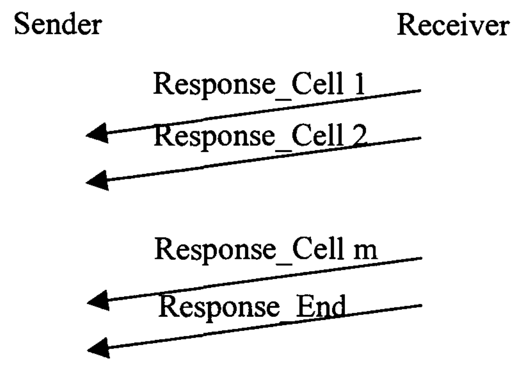


Figure 3. Receiver report the missing segments to the sender side

The interaction in Figure 2 and Figure 3 continues until the receiver receives all data *segments* correctly (one segment of the message is contained in every data cell.) The receiver then sends a *Transmission Success* packet back to the sender and the sender returns an *End Confirm* packet. The connection is then terminated and both sides return to idle state. The *Transmission Success* packet will be retransmitted when its timer expires and no *End Confirm* is received.

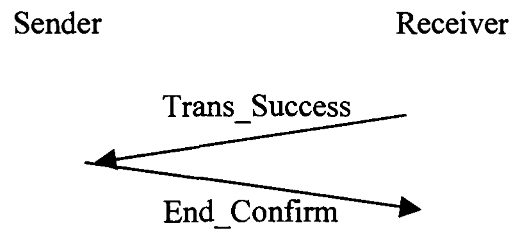


Figure 4. Successful termination of the connection

At any point, the transmission can be aborted by sending an *Abort* packet from either side. The other side then sends an *Abort Confirm* packet, as shown in the figure below:

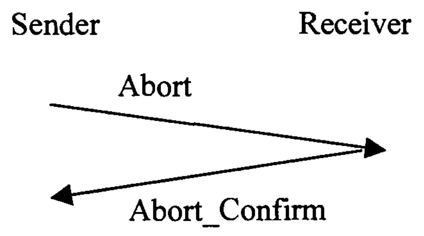


Figure 5. Abort operation

Brute Force Error Recovery Mechanism

There are two types of errors in radio transmission: random and bursty. Random errors in a digital signal are often single errors ocurring randomly in the transmission. Bursty errors are strings of bits in error followed by correct bits. Fragmenting a long message into multiple small-size cells is an advantage in correcting bursty errors; only the cells affected by the burst of errors need to be retransmitted. Assuming that the cell size is close to the average bursty error length, the overhead will not be high. However, for random errors, a single bit error within a data cell causes the retransmission of the whole data cell and this may lead to high overhead.

To decrease some of this overhead, we propose a method called *Brute Force Error Recovery* which, we believe, can gracefully handle random errors by trading communication overhead for computational overhead. The basic idea is that since random errors are independent, and in most cases sparsely distributed, (e.g. one error per thousand bits) the average interval between two adjacent random errors is usually large. If we choose the cell size correctly so that it is much less than the average interval, then if a cell contains a random error, the possibility that it contains only one such error is high. To correct this error, we can invert every bit in the cell one by one and recalculate the error detection code sum.

If we find exactly one case where changing the bit generates the correct error detection code sum, we have corrected that error. If no matching cases are found, there is more than one error in the cell. If multiple matching cases occur, our method does not give us, unfortunately, the ability to decide which is the correct one, and in both of these last two cases we cannot correct the error and need to ask for retransmission. Our Brute Force Error Recovery mechanism is illustrated in Figure 6. The mechanism does not require any communication overhead and the computational overhead is not very high. Normally, the complexity of calculating the error detection code we use is O(n), where n is the length of the packet. In our recovery method, we do the calculation n times, meaning that our mechanism has the computational complexity of O(n^2).

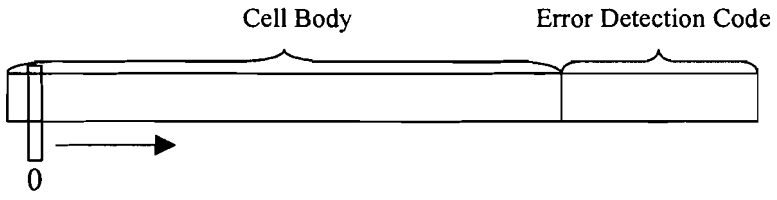


Figure 6. Invert every bit one by one to recover one bit error

However, in our implementation of the ERMTP protocol, the PSK31 DLL library [3] provides only a character (i.e. byte) interface. This is because *Vericode* (in BPSK mode) and *Conventional Code* (in QPSK mode) are used to represent the characters in an efficient way. Thus, it is hard with this PSK31-based system to recover the actual bit stream from the decoded characters. We implemented a variant of our Brute Force Recovery algorithm which works on a byte level. We still suppose there is only one byte in error in each cell, so we enumerate all 256 possible values for every byte and recalculate the error detection code sum. This byte-oriented variant of our algorithm adds more computational complexity. However, our experiment shows that compared to the low speed of the PSK31 link, the calculation is still fast enough for a data cell of 64 bytes, as the recovery time is only about 0.3 second on a Pentium 4 desktop.

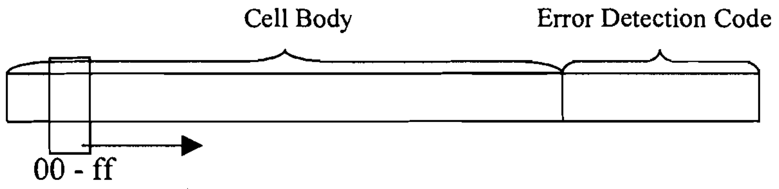


Figure 7. Try all 256 values for each byte to recover one byte error

The cases where we are missing one whole byte or where there is a redundant byte can be handled similarly with our Brute Force Recovery method. For the case of missing one whole byte, we can insert a byte into every possible place in the packet and enumerate all 256 possible values of that byte. The computational complexity is about the same as that of correcting one error byte. For the case of one redundant byte, we can delete one byte from the packet and verify the error detection code. In that case, every byte in the packet will be removed and the computational complexity will be much less than that of the case of correcting one byte that is in error.

The length of the error detection code and the coding algorithm are important to our brute force method. The error detection code should be strong enough to authenticate the message so that during recovery, the possibility of finding multiple matching cases is very low and the possibility of finding a wrong message that happens to match the error detection code sum is negligible. Too long a code length increases the overhead, but too short a code leads to collisions and makes the recovery trial fail. Assuming that the coding mechanism is perfect (i.e. all messages are randomly and evenly mapped to the code value space), Table 1 shows the possibilities of successful recovery for one character error with different data segment sizes and error detection code lengths.

	16 bytes cell length	32 bytes cell length	64 bytes cell length	256 bytes cell length
1 byte codes	1×10^{-7}	1.2×10^{-14}	1.4×10^{-28}	4×10^{-112}
2 byte codes	0.94	0.88	0.78	0.37
3 byte codes	0.9998	0.9995	0.999	0.996
4 byte codes	0.999999	0.999998	0.999996	0.999985

Table 1. Possibilities of successful recovery for different cell size and code length

From the table we can see that, one or two byte error detection codes are not strong enough to guarantee successful error recovery, while with three or more bytes of code, the possibility of a successful recovery is high. Since four byte codes do not improve the success rate very much over three byte codes, in our protocol we have chosen a three byte code length.

Initially we considered using a cyclical redundancy check (CRC) [4] of 24 bits. Although CRC is good at error detection, it is weak in message authentication. However, the most commonly used authentication codes such as MD5 [5], SHA1 [6], etc., are too long to be used here because they add to the overhead of cell transmission. In the end, we found a variation of CRC-32 called XUM-32 [7] which adds a hash mechanism to the CRC-32 making it stronger than CRC-32 for authentication. However, since we only need 24 bits of code, we do another step of hash on XUM-32 to get down to 24 bits.

Header Compression In Data Cells

During the transmission of a message, the major type of packet exchanged between sender and receiver are Data Cells. In each cell, besides the data fragment (payload), there is also one byte of "Packet Type" field, 2 bytes of "Segment ID" field, three bytes of "Error Detection Code" field, as shown below:

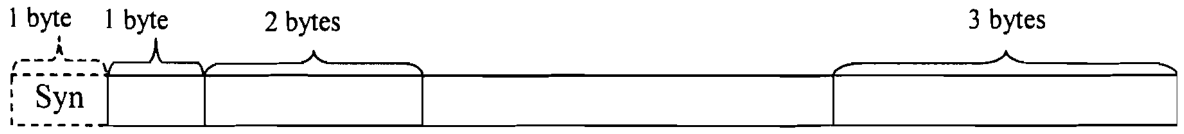


Figure 8. Data Cell format

In our implementation, there is also a synchronization byte before each packet. Thus, there are seven bytes of overhead in each data segment. However, we found that among these overhead fields, the "Type" and "Segment ID" fields can be compressed. The reason is that while transmitting Data Cells, the "Type" field will always be "Data_Cell" type, while the "Segment ID" will always be incremented by one between adjacent cells. Hence these two fields can be removed and we can compress the data cell format to that shown in Figure 9.

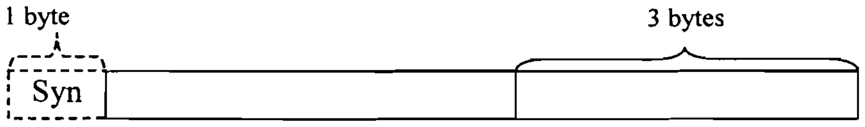


Figure 9. Data Cell Format with Header Compression

It should be noted that although the "Type" and "Segment ID" fields are removed when transmitting, they are included in the calculation of the Error Detection Code, thus the decompressed result at the receiver can be verified for correctness through verifying this code.

The header compression mechanism works as follows. Uncompressed data cells are transmitted with an interval of *n* cells. Between these *n* cells are the compressed data cells. Both sender and receiver maintain state variables which indicate the current data Fragment ID. The uncompressed data cells are used for the synchronization of sender and receiver states, in case some cells are lost. The compressed data cells can be easily decompressed using the state variable. Simply adding one to the last Data Segment ID yields the current Data Segment ID.

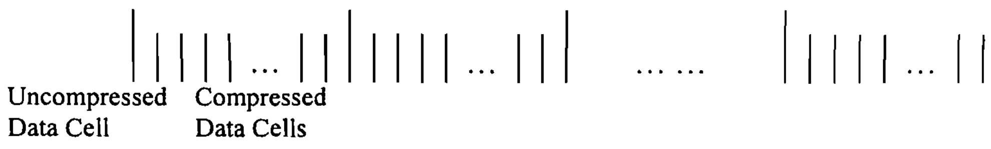


Figure 10. Header Compression for Data Cells

It is possible that some cells are missing due to transmission error. In this case, simply adding one to the last Data Segment ID will not generate the correct Data Segment ID, so the verification of the Error Detection Code will fail. When this happen, our algorithm will try to guess the correct Data Segment ID by adding 2, 3, 4..., M, until we finally find the correct ID or give up after M tries and wait for the next uncompressed data cell to arrive for resynchronization.

Current Implementation

Currently, we have a rudimentary implementation of the ERMTP protocol based on the *PSK31Core.DLL Version* 1.15 by Moe Wheatley, AE4JY. PSK31Core.dll is a dynamical link library in the Microsoft Windows platform. The library provides a character level interface for half or full duplex

PSK31 communication for both BPSK and QPSK modes. In our implementation of ERMTP, we only use the BPSK mode.

Our implementation is contained in two separate programs implementing sender and receiver protocols. We implemented the programs in Visual C++ 6.0 on Windows XP. Both sender and receiver protocol programs provide a simple and easy to use interface.

For the sender protocol program there are only three functions:

- int InitSendProto(HWND hWnd);
- void SendMSG(int txFreq, int rxFreq, unsigned char *msgBuf, int msgLen, int segLen);
- void StopSendProto();

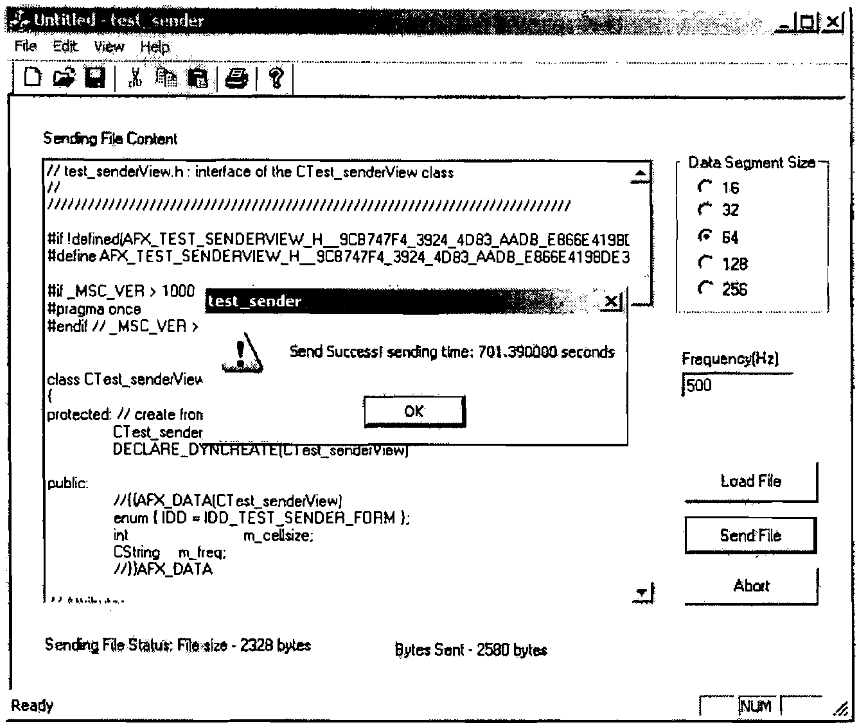
Before sending any messages, InitSendProto() should be called to initialize the sending protocol. SendMSG() is the function to send a message. The transmission frequency and the receive frequency (if different), the message body and length, and the segment length can be designated as its arguments. SendMSG() is an unblocked call, so it returns immediately. When the sending process finishes (whether successful or not), the program will generate a Windows message called MSG_SENDMSGEND, with its WPARAM indicating success or failure. StopSendProto() is called for clean up the sending protocol.

Similar to sender protocol program, receiver protocol program also provide three functions:

- int InitRecvProto(HWND hWnd);
- void RecvMSG(int txFreq, int rxFreq, unsigned char *msgBuf, int *msgLen);
- void StopRecvProto();

Before receiving any message, InitRecvProto() needs to be called to initialize the receiving protocol. RecvMSG() is then called to start receiving a message. The receiving frequency and the replying frequency (if different), the address to save the message and its length are specified as arguments to this call. RecvMSG() is also an unblocked call, so it also returns immediately. When the receiving process for the incoming message finishes (whether successfully or not), the program generates a Windows message called MSG_RECVMSGEND, with its WPARAM parameter indicating success or failure. StopRecvProto() is called for clean up the receiving protocol.

Our test programs are designed to use ERMTP for document file transfer. The screenshots of the sending and receiving programs are shown as below:



Sending File Content

```
// test_senderView.h : interface of the CTest_senderView class
//
///////////////////////////////////////////////////////////////////////

#if !defined(AFX_TEST_SENDERVIEW_H__9CB747F4_3924_4D83_AADB_E866E4198I
#define AFX_TEST_SENDERVIEW_H__9CB747F4_3924_4D83_AADB_E866E4198DE3

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER >

class CTest_senderView
{
protected: // create fron
        CTest_sender
        DECLARE_DYNCREATE(CTest_senderView)

public:
        //{{AFX_DATA(CTest_senderView)
        enum { IDD = IDD_TEST_SENDER_FORM };
        int             m_cellsize;
        CString   m_freq;
        //}}AFX_DATA
```

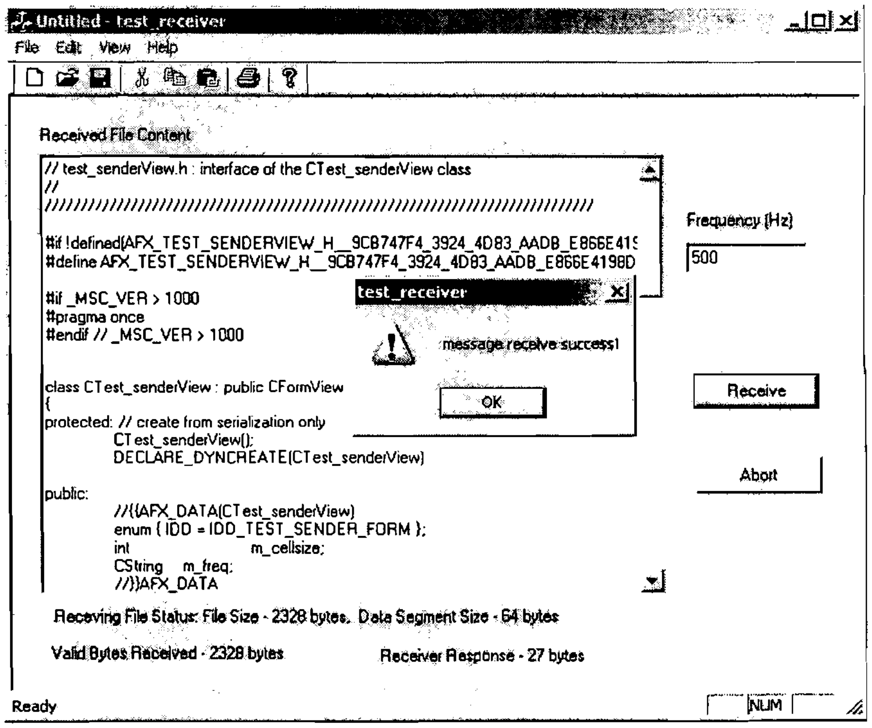
test_sender: Send Success! sending time: 701.390000 seconds

Data Segment Size: 16 / 32 / [✓ 64] / 128 / 256

Frequency(Hz): 500

Sending File Status: File size - 2328 bytes   Bytes Sent - 2580 bytes

Figure 11. Sender Side Program for File Transmission



Received File Content

```
// test_senderView.h : interface of the CTest_senderView class
//
///////////////////////////////////////////////////////////////////////

#if !defined(AFX_TEST_SENDERVIEW_H__9CB747F4_3924_4D83_AADB_E866E41
#define AFX_TEST_SENDERVIEW_H__9CB747F4_3924_4D83_AADB_E866E4198D

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CTest_senderView : public CFormView
{
protected: // create from serialization only
        CTest_senderView();
        DECLARE_DYNCREATE(CTest_senderView)

public:
        //{{AFX_DATA(CTest_senderView)
        enum { IDD = IDD_TEST_SENDER_FORM };
        int             m_cellsize;
        CString   m_freq;
        //}}AFX_DATA
```

test_receiver: message receive success!

Frequency(Hz): 500

Receiving File Status: File Size - 2328 bytes, Data Segment Size - 64 bytes

Valid Bytes Received - 2328 bytes   Receiver Response - 27 bytes

Figure 12. Receiver Side Program for File Transmission

Experimental Results

As of the time of writing, ERMTP was not tested with real HF radio transmissions. However, we tested the correctness and effectiveness of our current ERMTP implementation by directly connecting the sound cards of two computers so that the audio out of one is fed into the line-in port the other. In this way, the radio transmission process is by-passed but the PSK31 and ERMTP programs on each computer could communicate.

We successfully transmitted file sizes of up to 128K bytes, as shown in Table 2. Due to the time it takes PSK31 to transmit even a small file, we were not able to test larger file sizes. We believe, however, that the results will be proportional for the larger sizes. We intend to do more testing with larger file sizes as time becomes available.

File Size (bytes)	**Transmitted Size (bytes)**	**Transmission Time (seconds)**	**Overhead**
13178	14041	3552.4	6.55%
28501	30345	6884.7	6.47%
130479	138901	28979.2	6.5%

Table 2. Tests of transferring files with different sizes (data segment length: 64 bytes)

We also measured the protocol overhead for different data segment sizes as shown in the table below.

Segment Size (bytes)	**File Size (bytes)**	**Transmitted Size (bytes)**	**Transmission Time (seconds)**	**Overhead**
16	5961	7535	1861.7	26%
32	5961	6915	1629.1	16%
64	5961	6369	1414.6	6.8%
128	5961	6182	1344.7	3.7%
256	5961	6089	1307.5	2.1%

Table 3. Protocol overhead measured for different data segment sizes

The direct sound card connection provides a communication channel with a low error rate and no fading. For lack of time and resources we could not do on-the-air experiments, so we simulated a noisy channel with random errors and fading with bursty errors. To meet the requirement for Brute Force Error Recovery, i.e., not having more than one error in a packet, we chose the packet size to be about 1/5th of average interval between random errors. Bursty error rates are set to be 1/10th of random error rates, with a long average burst length of 20 bytes. The testing results are shown below.

Random Error Rate	**Segment Size (bytes)**	**File Size (bytes)**	**Transmitted Size (bytes)**	**Transmission Time (seconds)**	**Overhead**
1/100	16	5961	8085	2032.9	35.6%
1/150	32	5961	7150	1669.9	19.9%
1/300	64	5961	6521	1456.5	9.4%
1/600	128	5961	6740	1508.9	13.1%
1/1000	256	5961	6895	1516.3	15.6%

Table 4. Transmission overhead measured in simulated noisy channels

It can be seen from the above results that ERMTP is efficient in bandwidth utilization. The protocol overhead can be as low as 2%. ERMTP can also achieve low overheads for error recovery. With a properly selected data segment size (larger than four times the average error interval), the additional error recovery overhead can be kept as small as 3% to 13%.

Further Discussion of Error Recovery

In our experiment we found that only when the average random error interval is much larger than the cell size, will the brute force error recovery effectively correct the errors. However, since Brute Force Error Recovery can only correct one character error per segment, any more than one random error or a bursty error will always cause the data cell to be retransmitted, leading to a large overhead.

We are considering a solution to this problem of having to retransmit the whole cell in all cases where there is more than one error per segment or cell. Instead of retransmitting the whole cell from sender to receiver when the receiver indicates that an error has occurred in the initial transmission, we will send a pre-calculated Error Correction Code (ECC) for that segment and transmit that. Since the ECC for a segment is smaller than the original data segment, bandwidth consumption will be saved and overhead reduced.

In the method we propose here for optimizing ERMTP, during the first round of sending data cells, the sender pre-calculates an ECC code (such as a Reed-Solomon code [8]) for each cell. If the receiver cannot recover a data cell through Brute Force Error Recovery, it retains the cell's damaged content and its XUM (checksum). code The receiver's response message to the sender will contain two additional pieces of information: all missing data segment IDs and the checksums of the data cells that are received incorrectly. The response message itself will be protected against further errors with an ECC code. The sender will send the ECC codes for the damaged data cells identified by the returned XUM codes and resend other missing data cells. The receiver will then do error recovery on the incorrect data cells it has stored by applying the appropriate ECC code to the damaged cells. If a data cell cannot be recovered even when its ECC code is applied, the next response message from receiver to sender will indicate that there is a missing data cell, so the sender can potentially retransmit the whole cell.

The chief benefit of sending ECC codes in place of whole data cell retransmission is that the maximum error rate for the recovery of a data segment can be higher than what the Brute Force Error Recovery mechanism allows, namely, one per cell. Indeed, the error rate can be as high as the strength of the ECC allows.

In other uses of ECC, ECC codes are sent as part of Forward Error Correction (FEC). FEC generally adds to the bandwidth of a transmission and is meant to reduce the chance of errors at the receiver by sending redundant data. Unfortunately, low-bandwidth HF communication has a scarcity of bandwidth, so the tradeoff of bandwidth for delay is not a good choice because of FEC's high bandwidth overhead. In our proposed ERMTP optimization, ECC codes are sent instead of the data rather than accompanying it, so bandwidth is actually saved and overhead reduced.

At this time, this ECC-based error recovery mechanism is still under development. We plan to test its effectiveness, and then add it into our protocol library at a later date.

Conclusion and Future Work

This paper proposes a new protocol, the Eventually-Reliable Message Transport Protocol (ERMTP), that can provide reliable communications over low-bandwidth noisy radio channels such as commonly found with PSK31 on HF. PSK31 is a popular digital mode in Amateur Radio, but a weakness of PSK31 is that it only supports unreliable communication. This prevents it from being used in applications such as Email, File Transfer, WWW, etc. ERMTP enhances PSK31 with reliability, making the applications above viable in a PSK31-like environment.

ERMTP has been implemented in Visual C++ 6.0 on Windows XP. Our experiments on a simulated noisy PSK31 channel show that, with properly selected parameters such as data segment length, ERMTP can effectively achieve reliable communications on noisy channels with low overhead.

Our next step is to add an ECC-based error recovery mechanism into ERMTP. Also, since a bit stream level interface is more suitable to ERMTP, we plan to modify the PSK31Core library to provide such a bit level interface. The software will be released as a dynamic link library with simple interface functions. We will also conduct on-the-air testing to observe the performance of ERMTP in real HF radio communications.

References

[1] Steve Ford, WB8IMY, PSK31--Has RTTY's Replacement Arrived?
http://www.arrl.org/tis/info/HTML/psk31/index.html

[2] Steven L Karty, N5SK, PACTOR-II,
http://www.arrl.org/FandES/field/regulations/techchar/PACTOR-II.html

[3] Moe Wheatley, AE4JY, PSKCore.DLL Software Specification and Technical Guide,
http://www.qsl.net/ae4jy/files/pskcoredll133.pdf

[4] Cyclic Redundancy Check (CRC), http://www2.rad.com/networks/1994/err_con/crc.htm

[5] R. Rivest, RFC 1321: The MD5 Message-Digest Algorithm, April 1992,
http://www.faqs.org/rfcs/rfc1321.html

[6] SHA1 Hash Algorithm - Version 1.0, http://www.w3.org/TR/1998/REC-DSig-label/SHA1-1_0

[7] Ilya O. Levin, XUM32 and its implementation, http://www.nattyware.com/xum32txt.html

[8] Martyn Riley, Iain Richardson, An introduction to Reed-Solomon codes: principles, architecture and implementation, http://www.4i2i.com/reed_solomon_codes.htm