1 Introduction

Welcome to the Albatross Communication System! We’ve labored hard to bring you this exciting, and reliable product, from our laboratory to your home and your heart. The ACS is particularly tailored to withstand large BER fluctuations, competing radio traffic, and guarantees reliability to the applications that require it. We hope that our sample file transfer application will whet your appetite for more of our future products, such as ad-hoc routing (without leave your couch!), 802.11 compliance package, and the perpetual energy patch to help you save on transmit power, all coming soon near you.

2 ACS Link Protocol (ACSLP)

The ACS link protocol allows machines to address one another, and to multiplex application connections. Machine addresses are strings of 8 characters. Machine ports, used to multiplex connections, are strings of 2 characters.

3 ACS Transport Protocol (ACSTP)

The ACS transport protocol is bi-directional, and a reliable protocol for the software radio platform, which runs on top of ACSLP. ACSTP version 1.0 is similar to TCP in that our protocol has a connection handshake and a disconnection handshake, but ACSTP has fewer closing states and does not allow half-open connections.

ACSTP implements flow control in a sliding window fashion. The sender uses the remote receiving window size that it received during the connection handshake to limit its sending rate. Congestion control in ACSTP is experimental and loss-based.

Reliability in ACSTP is assured by a stream of acknowledgment from the sender. Each received data packet generates an acknowledgment packet that carries a cumulative acknowledged sequence number. Since ACSTP is bidirectional, the flow of acknowledgments and data traverses the airways in both directions asynchronously.
4 Implementation

The ACS is implemented in layers that are modular enough to make modification a simple matter. Right now we have two threads providing ACSTP support above the GNU radio stack. The first thread maintains a list of sending queues- one for each connection. It schedules packets onto the radio in round robin fashion to make sure that each connection is fairly allocated its air time proportionally to the number of competing flows on the machine. The second thread implements the protocol logic- generating acks to data packets, maintaining sending and receiving buffers for each connection, and communicating to the application when a particular event has occurred.

Packet reception happens in the GNU radio thread set, and incoming packets are distributed amongst connection queues as well- making it difficult for a single connection to flood the incoming queue with its own packets, thus making the library drop incoming packets for other connections.

5 API

We’ve implemented a socket like API for the application layer. The API is especially simple to use for developing complex applications. It allows the application to open multiple connections, use them from multiple threads and care little for what’s under the hood of the library. Yet the API allows for blocking and nonblocking modes for sending and receiving packets, timeout specification for initializing or closing the connection and other functionalities that offer the application designer fine tuned control over the connection.

For example, the basic syntax for connecting, sending two buffers, and disconnecting from the active connection side looks like the following:

```python
1 ctl = Control(options, args, 'my_hostname')
2 conn = ctl.new_connection()
3 ret = conn.connect('remote_hostname',1024) # port 1024
4 conn.send(buffer) # by default, blocking = False
5 conn.send(buffer, True) # explicitly, blocking = True
6 conn.close()
```

While from the passive connection side, the code to connect, receive two buffers, and to disconnect would look like the following:

```python
1 ctl = Control(options, args, 'my_hostname')
2 conn = ctl.new_connection()
3 ret = conn.listen(1024, 10) # listen on port: 1024, with 10s timeout
4 packet = conn.recv(True) # explicitly, blocking = True
5 packet = conn.recv(True) # explicitly, blocking = True
6 conn.close()
```

In the future we intend to extend our API to allow the application to specify routing priorities if it has any, packet loss thresholds, error rate thresholds, and other parameters that would allow the underlying layers to fit the reliability logic to the needs of the application.
6 ACS Physical Layer Experiments

The ability to experiment with the physical layer is one of the most important features of a software radio system. In this phase, we conducted many measurement experiments to determine the properties of the communications channel between the devices. In addition, we added reliability to our communications channel as well as a little bit of intelligence in selecting bit rates, transmit power levels, and error correction.

6.1 Measurements

We implemented a way to measure packet drops as well as bit errors across an unreliable channel using the following technique. Both the sender and receiver have access to a shared source of randomness (a pseudo-random number generator using Knuth’s method of Linear Congruence Sequences [citation here]), and the sender sends some number of packets asynchronously to the receiver. The receiver compares the packets it decodes against the next few packets it is expecting to receive, and selects the most similar as the match to the transmitted packet. In this way, when packets are dropped the receiver notices and automatically skips over that chunk of the random stream. The receiver also keeps a running total of bit errors in the packets it receives.

Our measurement techniques provided us with some interesting, and puzzling, results. For instance, established theory of communications implies that a higher signal-to-noise ratio (SNR) should result in fewer errors in transmissions. However, we did not find this to always be the case. Consider Figure 1: this figure shows the BER vs transmit power for varying bit rates. The theory claims that at a fixed bit rate, BER should decrease exponentially as power increases; our figure supports this claim. However, the theory also claims that as bits become shorter, i.e. the bit rate increases, a higher SNR is needed to maintain the same BER. The data for the 100kbps and the 150kbps streams agree with the theory, but in the transmit power range used in this experiment, the 125kbps transmissions were received perfectly, even at low power levels that where the 100kbps stream was not.

To attempt to explain this deviation from expectation, note that at 100kbps, the USRP sends 4 samples per symbol and decimates the input 64mmps (megahertz per second) stream by 160 to achieve this rate. At 150kbps, there are 3 samples per symbol and a decimation factor of 142, and at 125kbps, there are 2 samples/symbol and a decimation factor of 256. Further investigation into the physical layer showed that the USRP uses a decimation filter that is usually presented as decimating by a factor of 2. Our suspicion is therefore that something in that filter behaves poorly at other decimation factors, and led us to constrain our choice of bit rates to those that divide into 64,000,000 evenly as a power of two.

6.2 Reliability

There can be many reasons for the introduction of bit errors in a packet. It can be due to noise in the channel or due to the choice of bit rate, but even adjusting the physical parameters under our control is not guaranteed, or even likely, to result in a lossless channel. Therefore, for a sender to know that a packet is received by the receiver correctly, acknowledgments alone do not suffice. There needs to be a mechanism for the receiver to detect errors.
We have added cyclic redundancy check (CRC) into our packets. A CRC is a type of hash function used to produce a checksum - a small, fixed number of bits - on a block of data, such as a packet of network traffic. The checksum is appended to the packet before transmission and compared to a newly calculated checksum on packet payload after reception, allowing the receiver to detect corruption with high probability. We have used 32 bit CRCs provided by the GNU Radio. These CRCs are appended below the ACSTP layer.

There are many benefits to using CRCs. First of all they provide a mechanism to detect errors with a false positive (i.e. claiming a corrupted packet is intact) of $2^{-32}$, or approximately $10^{-10}$. Also, they can be computed at a very fast rate: the machine in Sieg Hall can compute and append a 32 bit CRC of 1 KB packet in about 80 microseconds, and it can also verify and delete CRC from packets of size 1 KB in 36 microseconds. Hence CRCs do not add much overhead in processing the packet by the nodes, i.e. this operation does not limit the bandwidth. CRCs are also easy to analyze mathematically, and are particularly good at detecting common errors caused by noise in transmission channel.

When the CRC check detects an error in the packet, the MAC layer throws the packet away and does not send an ack back to the sender. The sender, failing to receive an ack rightfully thinks that the packet was lost and resends the packet. This happens until the receiver gets the right packet. Let’s do some simple analysis to compute the total number of expected resends to ensure successful transmission of the packet.

Let bit error rate be $b$, the packet size be $X$ bits and the number of bit errors be $B$. Then, $b = B/X$ and $P[\text{packet is Good}] = (1 - b)^X = (1 - B/X)^X \simeq e^{-B}$ for small values of $B$. Hence the expected number of transmissions is about $e^B = e^{bX}$, when $b << 1$. 

6.3 Reducing BER via Error-Correcting Codes

As Figure 2 suggests, having the capability to detect errors only is just not enough. It will lead to very large number of retransmissions for each packet. Because of this rationale we have included error correcting codes in our communication channel.

Error-correcting code (ECC) is a code in which each data signal conforms to specific rules of construction so that departures from this construction in the received signal can generally be automatically detected and corrected. We have used ECC for the above mentioned reasons. If we are able to detect and correct errors, up to a limit then this highly reduces the total number of retransmissions. Which in turn increase the throughput of the network. To transfer each packet successfully we have to send lesser number of packets, on average. But all of this comes with a price. There is a small fraction of overhead involved with ECC. Unlike CRC where the overhead was constant irrespective of the packet size, here the overhead is proportional to packet size.

We have used one of the highly studied ECC, Hamming Codes. They are able to detect errors of 3 bit and correct errors of 1 bit. Important thing to note here is that, we divide the packet to small size blocks and encode them independently. This gives us more resilience.

The average retransmissions for a packet is \( p \), then from previous \( \log(p) \) can be seen as the equivalent bit errors in the packet if there were not ECC. This will be helpful to understand the benefits we reap by adding the ECC. Figure 2 suggests that by adding ECC we have made our channel appear as a channel with very low bit error rate.

But, we also have to make sure that in doing encoding and decoding of packets, we are not slowing down the system. Both encoding and decoding will take lot of time if we don’t do it intelligently. For example the encoding and decoding which used higher level functions of pythons took about 20 milliseconds to decode and encode a 4Kb packet. But we can pre-generate the codes/decoded message and hash them and use them to quickly code and decode. We used the above idea to implement the coding and decoding scheme. It takes about 1 millisecond to encode and decode 4Kb size packet. This is implementation will be much faster if it was done in C or assembly level language. And our results are for implementation in python.
7 Conclusion

The ACS is designed for reliable data transmission. It has a simple to use API that shields the application from low level details and yet exposes enough functionality to make connection details configurable. The current sliding window method allows for robust flow and congestion control, and the link layer (ACSLP) is extensible enough to support ad-hoc routing in the future. The inclusion of error correction (on top of the default checksumming operations) has enabled us to transfer at higher bit rates and to utilize the capacity of the air waves more fully. We hope to improve the physical layer adaptation further as we learn more about the device, our network, and the situations in which we operate in future projects.