CSE561 Project Phase1
2/3/2007
The A Team: (aka Group A)
Andrew ‘Murdock’ Guillory, Jonathan ‘Hannibal’ Hsieh, Nathan ‘Face’ Kuchta,
Nicolas ‘B.A.’ Murphy.
"I love it when a plan comes together!"

Goal:

We designed our application (phase1.py) to demonstrate basic reliable communication between two nodes with a file piping capability similar to netcat (a simple TCP/UDP socket piping program). We had two specific uses cases in mind for demonstration purposes.

- Bulk file transfer.
- Remote shell terminal.

Introduction

This paper is broken up into several section. First, we describe initial tests that motivated the direction of our work. Next, we describe the packet format, its header, justification for its contents and our ARQ method. After that, we describe the modes, and the tests we created to demonstrate the communications reliability and versatility. Finally we describe the results of our tests, and discuss what these results mean and how we could improve upon it.

Physical layer

Initially we started with the example echo.py program. It used DBPSK (Differential Binary Phase Shift Keying) and a bit rate of 200k. We implemented simple tests -- test-rx.py and test-tx.py --that transmitted known values and checked and displayed the received values. This gave us a feeling of what kinds of physical errors to expect, and helped us get familiar with the gnuradio system.

Playing with DBPSK we modified the bit rate, and observed some results that directed us.

- Lowering the bit rate to near 100k consistently caused problems. We noticed that oftentimes the receiver got bit shifted values. For example, when sending an ASCII ‘A’, 0x41, the receiver would show an 0x82. We believe that this is a timing issue and that line encoding could mitigate the problem.
- Upping the bit rate was ok until we reached 345k. At this point, we noticed the PC started acting sluggishly. When we upped the rate to >=350k, the PC became unresponsive and we had to resort to a hard reboot.

We initially implemented 4B5B encodings and 4bit and 5bit chunkers. The “differential” part of DBPSK provided NRZI encoding. First attempts at tests placed this module into the MAC layer. In order to move it to the physical layer (into the graph) we explored the gnuradio system and in the process found other modulation packages. This lead us to shelve the 4B5B work.

We found modules for DQPSK (Differential Quaternary Phase Shift Keying) and GMSK (Gaussian Minimum Shift Keying, apparently a frequency modulation currently used for GSM/GPRS cellular systems) modulations. First we tested DQPSK but got poor results – we had difficulty establishing any communications during testing.

GMSK was much more successful. Simple tests showed reasonable error rates and we were able to use a wider range of bit rates.

- At lower rates and medium rates (100kbps-300kbps) we encountered a significant percentage (~20%) packet bit error rate. We used this fact later when we tested our reliability mechanism.
As rate increases, (300kbps-1000kbps) the error tends to be less severe.
At high rates (>1000kbps) we locked the PC up and required a hard reboot.

In our testing we found that the provided carrier sense code was backing off very frequently—on the order of 10 times per packet in some cases. We experimented with changing the exponential backoff times but this did not seem to help. As a temporary measure, we have simply disabled the carrier sense, which we found greatly increased our transfer rates. We speculate this is due to latency between the software and the radio. This is an issue we may revisit in future phases.

**Packet format**

One of the first steps taken was to create a separate class for packets. This isolates packet layout details and simplifies minor changes due to future modifications to the packet format.

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Rx Access Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length (x2)</td>
<td>CRC32</td>
</tr>
<tr>
<td>Tx Access Code</td>
<td>Sequence Number</td>
</tr>
<tr>
<td>Flags</td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td></td>
</tr>
</tbody>
</table>

The current packet format is shown in the diagram above. Much of the format is derived from the baseline echo program. The preamble, RX access code, and length (x2) are taken directly from original format. Our additions include the CRC-32, the TX access code, a sequence number, and a set of flags. The RX/TX access codes are used for addressing the sender and the receiver, and were provided primarily for the next phase of the project. The sequence number provides details about the state of the transmission. The flags include whether a packet is an ACK, as well as provide future flexibility for additional features.

We compute CRC-32 checksums using the built-in Python function crc32 in the binascii module. The checksum includes everything after it excluding the buffer (e.g.: Tx Accesscode, sequence number, flags and payload). The preceding fields are stripped by the gnuradio graph and belong in the PHY. We initially considered using a more complicated error correcting code method such as Turbo codes, but start simple first. We wanted to have a baseline to justify and also to potentially compare error rates if we went to the more complicated code. More often then not, we found that for certain encoding schemes and bit rates caused entire packets drops as opposed packets with bit errors. This makes the checksum less relevant.

**Automatic Repeat reQuest (ARQ)**

One of the key requirements in providing reliable communications is some form of Automatic Repeat reQuest (ARQ). We began by starting simple, with the option of being able to modularly replace mechanisms. One assumption we made was that our point-to-point wireless transmission would not have to deal with out of order packets. Sequential packet transfers implied that we could use the simplest ARQ algorithm, stop-and-wait, and if necessary, move to sliding window. We initially had a one bit flag for stop-and-wait ordering, but decided that for the future a sequence number would allow us to smoothly transition to sliding window and provide us with more general flexibility.

Acknowledgments are done at the ARQ level as well. ACK packets are sent in response to each valid data packet received. If a packet is found to be invalid, no ACK packet is sent back, nor is any explicit message sent to alert the sender of the problem. Instead, a timeout on the sender will cause the packet to be resent. If a
timeout occurs, the receiver may acknowledge that both packets were sent, which may cause problems. Our approach resolves this issue by including the sequence number to which they are referring in ACK packets.

We needed a way to deal with an unusable channel or a receiver not responding for some reason. Simply resending the packet without some upper-bound on the number of resends will cause unnecessary traffic. Instead, after a specified number of failed re-sends, the ARQ gives up and throws an exception to signal the application of the problem.

We found it necessary to assert that a payload was not longer than a specified size. Our ARQ fragments large payloads into multiple packets of specified maximum size. The work of reconstructing received packets is delegated to the application.

**Reliable payload modes**

We created several modes in order to handle different input types and to force the testing of different types of traffic. Four send modes were created:

- ‘line’ mode acts like the original echo program. Any ‘\n’ character acts as an input delimiter and forces a send. All unfragmented packets end with a \'n' character.
- ‘char’ mode sends only one character payloads per packet. This is inefficient but very useful for testing. Initial remote shell testing efforts used this mode to debug ARQ issues.
- ‘blk’ mode sends variable sized packets when input is received. This mode uses poll (similar to select) and is useful for more efficient bulk transfers. This also made remote shell operations more efficient.
- ‘term’ mode modifies the posix termios settings (unix terminal settings) in order to provide more interactive command line features common in bash and tcsh shells such as tab-command completion and up-arrow command history. This feature is not complete.

**Usage and Demo**

Here’s a summary of the command line parameters we added:

- Modulation selection ("-m"). The user may select from between gmsk, dbpsk, dqpsk. Since it worked most reliably, the default is gmsk.
- Verbose mode ("-v"). Normally all non-duplicate received payloads are sent to stdout. When in Verbose mode, detailed diagnostic information is displayed including acks, the payload at different layers, carrier backoff, etc. The default mode is not verbose.
- Mac/read mode selection ("-M"). The user may select between line, char, blk or term modes. blk mode is the default.

For file transfer, we can directly use the piping mechanisms from unix. The receiver must be started before the sender:

```
receiver$ ./phase1.py -f2.412G -r 200k -m dbpsk> destfile
sender$ ./phase1.py -f 2.412G -r 200k -m dbpsk < sourcefile
```

For remote shell we use these command line. The receiver must be started before the sender:

```
# client.sh
# example is in char mode (generally use block or term mode)
```
Statistics.

To aid in testing our application we added a logging features. The application logs the following information and writes this information out to a file whenever the application exits (cleanly or otherwise):

- The total number of bytes of data sent by the application.
- The number of times the high level send method of the protocol was called. (number of payloads before fragmentation).
- The number of packets successfully sent by the application, excluding acknowledgements.
- The number of times the application attempted to send a packet, excluding acknowledgements.
- The number of small (< 20 bytes of payload) packets successfully sent by the application, excluding acknowledgements.
- The number of packets received by the application.
- The number of bad packets received by the application. (bad CRC)
- The number of back off's performed by the carrier sense.
- The maximum back off time used by the carrier sense.

From this information, other statistics can be calculated. For example, the number of acknowledgements sent by the application is always equal to the number of good packets received by the application, and the average number of retries per packet is the number of packet send attempts divided by the number of successfully sent packets. By comparing the logs from the sender and receiver, we can also infer information about the number of packets not received.

We tested the file transfer use case for our application on three files with sizes of approximately 2 KB, 10 KB, and 100 KB. Each file was sent 5 times using three different modulation and bit rate settings. All the transfers were successful, except for one of the 100KB transfers using the DPBSK encoding scheme. This transfer timed out after the maximum number of re-sends (30) was exceeded. This seems to have been an issue with interference, since the other 4 sends with the same settings succeeded. Below is a graph of the transfer times for each setting. The best encoding scheme and bit rate (GMSK 500k) gave a transfer rate of around 10 KB per second.
We also examined the logs from 100KB transfer test runs and calculated the average number of sends per packet and the overall percentage of bad packets averaged over all the runs. As seen below, the GMSK modulation scheme at 500k had significantly fewer re-sends. Interestingly, despite having a high resend rate, the GMSK modulation scheme at 200k had a relatively small percentage of bad packets. At that bit rate, the GMSK modulation scheme seems to completely drop packets as opposed to receiving packets with bit errors.
Errata:

- Verbose mode: There were a few items we could not remove including the initial messages of the form ">>>g_fir_ccf: using SSE" and "UuUuUu" messages which represent under and overflow conditions. We believe these come from the C++ parts of the gnuradio library.
- Exiting and retry fail out: Currently the application does not fail gracefully.
- Remote Shell: In general, command shells output prompts to stderr while the programs they run output to stdout. Because we are combining two streams into one, the order of piping output is not always consistent.
  - 'blk' mode: During interactive use, the application waits for "\n" before sending packet.
  - 'term' mode: During interactive use, the application sends individual chars (even before "\n") but control chars (delete, backspace) are not echoed properly. Ideally terminal features should be orthogonal to the packet send method.
- Undetermined bit packing bug: In recent tests while attempting to transfer a 1 MB file, we encountered a bug where a specific packet (the 805th) would become shifted in transmission every time and fail the CRC check. We are still investigating this bug, but we suspect it is a problem with our packet encoding scheme.