Data Compression and Data Integrity: Projects for Data Communication Courses

Sub Ramakrishnan, Mohammad B. Dadfar

Department of Computer Science
Bowling Green State University
Bowling Green, Ohio 43403
Phone: (419) 372-2337 Fax: (419) 372-8061
email: datacomm@cs.bgsu.edu

Abstract

This paper describes two software projects that are assigned in our undergraduate data communications course. The projects help students understand Lempel-Ziv algorithm and CRC generation of the OSI data link layer. The students write software to implement these two schemes. Students are particularly excited about implementing Lempel-Ziv because popular Unix utilities, such as *compress*, use a variant of this algorithm.

1. Introduction

Study of operating systems and data communications concepts is an important subject area for most undergraduate computer science programs. In our department we have offered a sophomore level mandatory course that introduces both of these concepts. Following this course, we have elective courses in operating systems and data communications. This paper deals with a project in the latter course.

The elective data communications course covers a range of topics including protocol architecture, client server communication and remote procedure calls, compression and encryption, multiplexing and transmission media. Where possible, hands-on programming projects are used to enhance the learning process and to gain additional insight into specific topics. A good number of text books address these topics[1, 2, 3, 4]. In the past few years we have used different textbooks including *"Understanding Data Communications and Networks,"* by *William Shay, Second Edition* (*1999*). In this paper, we focus on projects that deal with two of the topics in the course, error checking in communications and data compression to provide for bandwidth efficiency.

The first project deals with computing *CRC*-16 (Cyclic Redundancy Check) in software. The generator polynomial and the payload message are both given as input. The software emulates a well-known hardware implementation so it is very fast. It is written as a client server program,

the client does the *CRC* computation and transmits the payload and checksum and server determines the validity of the received data. We simulate transmission errors by occasionally flipping payload or checksum bits before it is sent to server. However, the server is not told about it.

In a follow-up project students implement the *Lempel-Ziv* algorithm for data compression/decompression. It is then integrated into the above project by doing compression, at client side, prior to *CRC* generation. Similarly, decompression is employed at server side following *CRC* verification. The students may implement the projects in a language of their choice though most choose C++ or Java.

Our students feel that completing these projects have helped them to gain a better understanding about operating systems concepts and data communications and networking between processes. Students are particularly excited about implementing Lempel-Ziv because popular Unix utilities, such as *compress*, use a variant of this algorithm. Though CS 429 is an elective course, over 80% of our undergraduate students take it.

In Section 2 we discuss the CRC generator project. In Section 3 we discuss the data compression project. The paper concludes with some remarks in Section 4.

2. CRC Generator

We usually spend over two weeks on protocol architecture and the OSI model. The OSI model introduces the notion of reliable communication in the bottom three layers which correspond to the three layers of the X.25 model. Reliable communication provides for loss-free, duplication-free, error-free and in-sequence communication of message between nodes. X.25 uses a cyclic redundancy checksum to protect against loss of these messages. The checksum is 16 bits and is formed by a division algorithm. The dividend is the message (in polynomial from) and the divisor is a well-known generator polynomial of degree 16. The reminder from this computation is 16 bits wide and is the checksum that is transmitted with the message. The receiver does a similar computation and computes the checksum and compares it with the incoming checksum. If both match the message is assumed to be error-free otherwise it is in error. In the latter case, the message is discarded.

It turns out the CRC mechanism can be implemented very efficiently using shift register mechanisms. The construction of the shift register is dependent on the divisor polynomial. The register is 16 bits wide for a divisor of degree16. At the transmit side, we append sixteen 0s to the message to form the dividend polynomial. This is then fed in one at a time to the CRC logic. When the last bit is fed what is left in the registers is simply the reminder of dividing the message polynomial by the divisor polynomial. For brevity, we do not include the CRC logic diagram. We refer the reader to page 240, Figure 4.8 of Shay's second edition or page 269, Figure 6.6 of the third edition book[1]. It shows the CRC logic for a degree 4 divisor polynomial, $X^4 + X^3 + 1$.

The transmitter transmits the original message and the 16 bits remainder. The receiver employs the same CRC logic and feeds what is received one bit at a time to the logic. When the last bit is fed, the remainder in the register is used to detect errors. If the register (all 16 bits) is zero then

no error has occurred, else there is an error. Students are very interested in understanding why it works the way it does and the error detection properties of the divisor polynomial itself. We explain some of these ideas in class and assign a software project to implement this computation.

In this project we ask the students to implement and test the CRC computation in software. A statement of the problem is given in Figure 1. The implementation is done in two phases. In Phase 1, one program does the transmit and receive functions. In phase 2, the transmit program is invoked as a client and the receive program is invoked as a server and TCP/IP communication is used between the two programs. The Phase 1 implementation is shown in Figure 3(a-b). Figure 3a is the driver module that is used to call the transmit or receive side and display the results on the screen. Figure 3b provides the actual implementation of the transmitter and receiver. This is modular and as seen from Figure 3b the CRC-16 class has two member functions, *transmit* and *receive*. The code for *transmit* module is shown in that figure. The *receive* module is similar and it is not shown for brevity.

A run snapshot of Phase 1 is given in Figure 2. The first two runs do the transmission and corresponding reception and show that the message is error-free. In the third run, the CRC is intentionally altered to show that the receiver can detect the error. Note that the original CRC is `0000000000001010` while the third run gives the receiver an erroneous CRC, `1000000000001010`.

The Phase 1 program can easily be expanded to accomplish the requirements of Phase 2. The expansion includes TCP/IP client and server communication modules. The payload for the communication is simply the message and the CRC from transmit (client) module to receive (server) module. For brevity, code for Phase 2 is not shown.

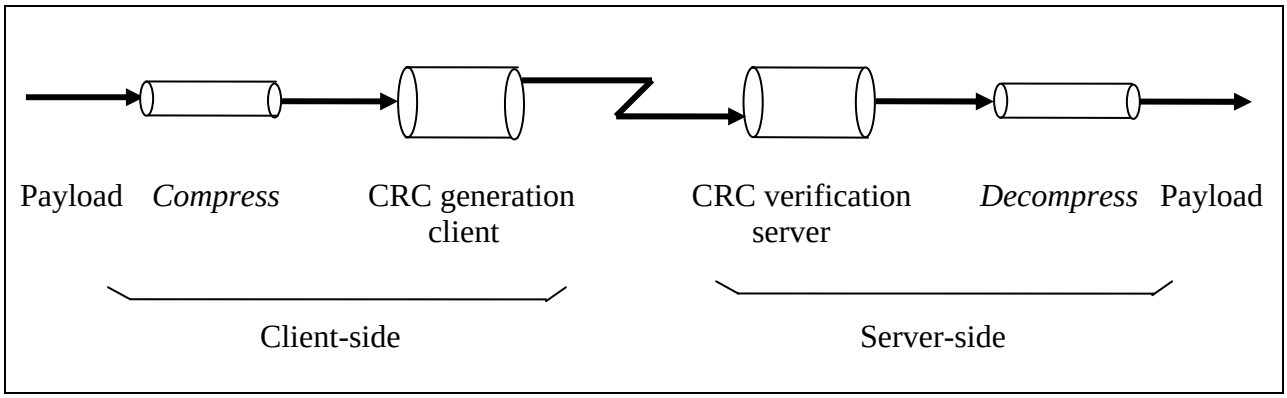
3. Data Compression

Many applications require communication of data and data compression is routinely used to reduce transmission time and message bandwidth requirements. In our data communications course, we spend two weeks on data compression techniques. Following an overview of lossy and lossless schemes, we discuss Huffman and run length encoding techniques. Then, we have an interesting class discussion on the *UNIX* utility *compress*, followed by an introduction to the application of Lempel-Ziv encoding schemes. Then, we discuss the algorithmic details of Lempel-Ziv encoding. A good discussion of this scheme can be found in Shay's second edition, pages 195-201 or pages 229-235 of the third edition[1]. The scheme allows us to recap and reinforce data structure topics taught in freshman classes. Students also appreciate the tradeoff between processing and memory requirements.

To help students better understand the intricacies of this algorithm the students implement a Lempel-Ziv C++ class - member functions include *compression* and *decompression*. Then, they write driver programs to test the working of these two member functions. A formal statement of the problem assigned to students is given in Figure 4.

Then, we ask the students to use this project as a wrapper for the CRC generator of Section 2. This process is illustrated below. This is easily accomplished by invoking *compression* on the payload, then feeding through CRC generator at the client. The compressed message and its

CRC are then fed to CRC generator at the receiver. Following CRC verification we feed it through the *decompression* function which extracts the original payload.



Recall that the client-side includes *compression* and CRC generation while the server-side includes CRC verification and *decompression* to extract the original payload. It is important to make sure the students' implementation confirms to Lempel-Ziv. We hammer the idea of object reuse in all of our courses and this principle is used here as well. The testing phase may employ client-side modules, X, of a student group with server-side modules, Y, of a *different* student group. This process helps them to realize the importance of clean interfaces and eventually help them to discover bugs in their implementation. Of course, if X and Y both contain the same bugs this would not help. A convenient testing scheme we occasionally employ is to replace X or Y with an equivalent instructor-written modules, I_X and I_Y. Then, X may be tested with I_Y or Y may be tested with I_X.

Students get perfect scores only when the behavior of their client-side and server-side modules is identical to that of replacing either side with I_X or I_Y as the case may be. Occasionally, we provide the students with the binary versions of our, instructor-written, modules I_X and I_Y. For brevity, the detailed code is not shown here.

4. Concluding Remarks

In this paper we discussed software implementation of two popular topics, CRC generator and data compression. The projects can be assigned in phases and as shown can be integrated at the end to provide for some continuity and a sense of accomplishments for the students.
The projects help reinforce a number of topics in data communications including protocol architecture, computer organization, emulation of hardware architecture in software, data compression and client-server communication.

Students are often curious about the error detection properties of the CRC logic. However, the theoretical treatment of linear cyclic block codes itself is left for an advanced graduate course on reliable computing since it usually involves many linear algebraic concepts and syndrome generation.

Students are particularly excited when these are assigned as group projects. It is important the instructor approves the composition of the teams.

- **Phase 1:**

  Write a C++ crcClass to implement the CRC-16 hardware bit shift mechanism. Include appropriate member functions to compute CRC, verify CRC.

  - Make this class general enough so you can easily plug it into Phase 2 (see below).
  - Implement a driver that calls the above with a payload data (X bits where X need not be a multiple of 8) as would be done at transmit side, or payload and CRC (as would be done at receiver).
  - The command line parameters include {tx or rx}, payloadFileName, CRC value {if receive}.
  - Submit detailed program listing, your comments, multiple input and output run snapshots, readme file, and what each one (in the group) did.

- **Phase 2:**

  - Client-server homework. Run client and server separately, as shown below.
    Client tx payLoadFile flipSomeBits serverMachine serverPortNumber Server
  - Client side does transmit functions (of Phase 1). Then, transmits to server. Flip a bit or two before transmission depending on the flag, `flipSomeBits`. Do `cout` as appropriate.
  - Server side does receive functions (of Phase 1). Receives the message and CRC and checks it. Do `cout` as appropriate.

Figure 1: Problem Description for CRC Implementation

```
hostname% a.out rx testFile 0000000000001010
Receive or CRC checking process

This is the original message bit string, in the file:  00101
This is the standard polynomial for CRC-16
G(x) = x^16+x^12+x^5+1
This is the CRC received: 0000000000001010
T(x) = B(x)-R(x)
This is T(x):
001010000000000001010
T(x)/G(x) to get the result
This is the result, which is the final values in register: 0000000000000000
Congrats! The transmitting process was successful!

//Below I made a CRC error before running receive module

hostname% a.out rx testFile 1000000000001010
Receive or CRC checking process

This is the original message bit string, in the file:  00101
This is the standard polynomial for CRC-16
G(x) = x^16+x^12+x^5+1
This is the CRC received: 1000000000001010
T(x) = B(x)-R(x)
This is T(x):
001011000000000001010
T(x)/G(x) to get the result
This is the result, which is the final values in register: 1000000000000000
the result is not 0, so the transmitting process was not successful!
hostname%
```

Figure 2: Run Snapshot of CRC Modules

```
/*********************************************************************
*                                                                    *
*  Script started on Tue Apr 13 12:23:07 2004                        *
*  $ cat README                                                      *
*  We run this program 4 times for each string in the file load.     *
*  The file load contain the message.                                *
*  1) transit---get the CRC for the message in the file.             *
*  2) receive---check to see if the message is valid by passing in   *
*               the correct CRC.                                     *
*               Result should be "no error"                          *
*  3) receive---check to see if the message is valid by passing in   *
*               the messed up CRC. Do so by mistype the CRC.         *
*               Result should be "error"                             *
*                                                                    *
*  4) receive---check to see if the message is valid by passing in   *
*               the correct CRC. But going into the load file,       *
*               manually mess up the original message.               *
*               Result should be "error"                             *
*********************************************************************/

/*********************************************************************
*     CS429   Assignment #4                                          *
*     Description: This is the driver of the CRC-16 hardware         *
*                  bit shift mechanism.                              *
*********************************************************************/

#include <fstream.h>
#include <string>
using namespace std;
#include <stdlib.h>

#include "crcClass.h"   //Implements CRC class

signed int main(int argc, char * argv[])
{
      if(argc == 3)
      {
         cout << "Transmit or CRC generation process" << endl;

         crc16 myCrc(argv[2]);
         myCrc.transmit();
      }

      else if(argc == 4)
      {
         cout << "Receive or CRC checking process" << endl;

         crc16 myCrc(argv[2],argv[3]);
         myCrc.receive();
      }

      else
      {
         cout << "Usage: a.out tx payloadFileName" << endl;
         cout << "Usage: a.out rx payloadFileName crc" << endl;
      }

      return 0;
}
```

Figure 3a: CRC Student Implementation - Driver Module

```
/* crcClass.h
#include <fstream.h>
#include <string>
using namespace std;
#include <stdlib.h>

class crc16 {
  public:
    crc16(char * filename) {
      strcpy(fileName, filename);
    }
    crc16(char * filename, char * crcIn) {
      strcpy(fileName, filename);
      strcpy(crc, crcIn);
    }
    void transmit();
    void receive();
  private:
    char fileName[100];
    char crc[100];
};


void crc16::transmit()
{
     //open payload file
     ifstream datFilePtr;
     string datFile = fileName;
     datFilePtr.open(fileName, ios::in);
     if( datFilePtr == 0 )
     {
          cout << "unable to open the payLoadfile "
               << datFile << endl;
          exit (1);
     }

     //read in the original string-----message
     char buffer[100];
     datFilePtr.getline(buffer,100,'\n');
     cout <<"This is the original message bit string "
          << " in the file: " << buffer << endl;
     cout << "This is the standard polynomial for "
          << "CRC-16 " << endl
          << "G(x) = x^16+x^12+x^5+1" << endl;

     //form B(x) by appending degree number of 0s to
     //the end.  in This case is 16, since it is crc 16.
     strcat(buffer,"0000000000000000");
     cout << "This is the message, B(x), with appended"
          << " 0s" << endl << buffer << endl;

     cout << "Divide this to get the R(x), which is the "
          << "CRC"
          << endl;
```

```
//division by using shifting register
     char regis[16];
     int length = strlen(buffer);
     int i = 16;
     int j = 0;
for (j=0; j<16; j++)
          regis[j] = buffer[j];

     while(i < length)
     {
          char temp = regis[0];
          //exclusive or for register 1
          if( (regis[0] == '0' && regis[1] == '0')
               || (regis[0] == '1' && regis[1] == '1') )
          {
               regis[0] = '0';
          }
          else
          {
               regis[0] = '1';
          }
           //copy over through register 14
          for(j = 1; j<14; j++)
          {
               regis[j] = regis[j+1];
          }
          //exclusive or on register 15
          if( (temp == '0' && regis[15] == '0')
               || (temp == '1' && regis[15] == '1') )
          {
               regis[14] = '0';
          }
          else
          {
               regis[14] = '1';
          }
          //exclusive or on register 16
          if( (temp == '0' && buffer[i] == '0')
               || (temp == '1' && buffer[i] == '1') )
          {     regis[15] = '0';  }
          else
          {     regis[15] = '1';   }

          i++;
     }

     cout << "This is the R(x), which is the CRC: ";
     for (j=0; j<16; j++)
     {
          cout << regis[j];
     }
     cout << endl;
}
```

Figure 3b: CRC Student Implementation - Transmit Module

Data Compression

Knowledge Acquisition Goals

- Study compression efficiency.

Skill Goals

- Program the Lempel-Ziv algorithm for compression and decompression.

Assignment Specification

- How to run the program?
    - `Program task fileNameIn fileNameOut`
    - `task` is either compression or decompression
    - `fileNameIn` is name of file to be compressed or decompressed.
    - `fileNameOut` is the output file
- Develop a class with appropriate member functions for compression and decompression.
- Do a `diff` (UNIX command) of the original file and decompressed output.
- Do a `wc` (UNIX command) of the original file and decompressed output.
- The above two items may help you uncover program bugs.
- Run the program for a variety of inputs.
- Submit readme, snapshot, program listing and the whole works.

Additional Work: Integration with CRC client-server modules

- Use this class as a wrapper for the previous CRC assignment
- Ensure that the following things happen, in order at the client side: compress payload, generate CRC, transmit compressed payload and CRC
- Ensure that the following things happen, in order at the server side: verify incoming CRC, extract compressed payload, decompress to extract original payload
- Provide multiple test runs
- Comment on your findings.

Figure 4: Problem Description for Lempel-Ziv Implementation

Bibliography

1. Shay W., "Understanding Data Communications and Networks," (Third Edition), Brooks/Cole, 2004.

2. Silberschatz, A., Galvin, P., and Gagne, G., "Operating System Concepts," (Sixth Edition), Wiley & Sons, 2003.

3. Stallings, W., "Operating Systems," (Third Edition), Prentice-Hall, 1998.

4. Tanenbaum, A. S., "Modern Operating Systems," (Second Edition), Prentice-Hall, 2001.

MOHAMMAD B. DADFAR
Mohammad B. Dadfar is an Associate Professor in the Computer Science Department at Bowling Green State University. His research interests include Computer Extension and Analysis of Perturbation Series, Scheduling Algorithms, and Computers in Education. He currently teaches undergraduate and graduate courses in data communications, operating systems, and computer algorithms. He is a member of ACM and ASEE.

SUB RAMAKRISHNAN
Sub Ramakrishnan is a Professor of Computer Science at Bowling Green State University. From 1985-1987, he held a visiting appointment with the Department of Computing Science, University of Alberta, Edmonton, Alberta. Dr. Ramakrishnan's research interests include distributed computing, performance evaluation, parallel simulation, and fault-tolerant systems.