2023

Project 2

Jeremy Varghese

jv852856

Contents

[System Documentation 2](#_Toc150434920)

[1. Data Flow Diagram 2](#_Toc150434921)

[2. Routines 4](#_Toc150434922)

[3. Implementation Details 5](#_Toc150434923)

[Test Documentation 7](#_Toc150434924)

[1. Method of Testing 7](#_Toc150434925)

[2. Testing Outputs 8](#_Toc150434926)

[3. Test Sets 8](#_Toc150434927)

[User Documentation 8](#_Toc150434928)

[Source 8](#_Toc150434929)

[How to run 8](#_Toc150434930)

[Parameters 8](#_Toc150434931)

# System Documentation

## Data Flow Diagram

A diagram of a diagram

Description automatically generated

## Routines

* 1. int server() – First process to be run. Accepts 6 clients, any further clients wishing to join have to wait until one of the current six clients logs out or disconnects.
  2. int client() – Connects to server IP and port. Has to provide 8 character unique username starting with a letter. ANY TEXT INPUT FED NEEDS TO BE TYPED WITH ` (Backtick or Grave accent) AT THE END FOLLOWED BY ENTER.
  3. int producer(char\* port, char\* fl, char\* buffer,char\* username, char\* to, char\* frameNum) - Process that receives input from user and sends data in chunks of 64 characters to be encoded.
  4. int encode(char \*inData,char \*len, char\* fdOut\_One,char\* isCap,char\* flag,char\* username, char\* to) – encodes incoming data frame of <=64 characters to 7 bits of binary for each character. Number of characters is encoded and added to frame.
  5. void addParity(char \*inData, char\* fdOut\_One,char\* isCap,char\* flag,char\* username, char\* to) – adds parity to the encoded frame with number of characters and the characters.
  6. void addHamming(char \*inData, char\* fdOut\_One, char\* isCap,char\* flag,char\* username, char\* to) – performs hamming encoding on given frame. Also sets a random error bit if isCap is ‘1’.
  7. void buildFrame(char \*inData, char\* fdOut\_One,char\* isCap,char\* flag,char\* username, char\* to) – adds SYN characters (hardcoded) to front of frame and writes to socket if flag is ‘h’ (hamming). If flag is ‘c’, it sends the full frame to addCrc.
  8. void addCRC(char\* inData, char\* fdOut\_One, char\* isCap,char\* flag,char\* username, char\* to) – performs CRC32 on given frame. Builds <MSG><FROM></FROM><TO></TO><ENCODE></ENCODE><BODY></BODY></MSG> frame and writes to socket. Adds random error bit to crc32 encoded string if isCap is ‘1’.
  9. void checkCRC(char\* inData, char\* fdIn\_One,char\* flag,char\* file) – performs CRC32 division on received frame and checks if remainder is 0. If not, prints that Data has been corrupted.
  10. int deframe(char \*inData, char\* fdIn\_One) – checks and removes SYN characters, and sends deframed data
  11. void removeHamming(char\*\* characters) – checks and corrects 1 bit errors in hamming encoded message.
  12. int checkRemoveParity(char\*\* characters) – checks parity bits for all characters and removes them.
  13. int decode(char\*\* blocks) – converts encoded binary characters to ascii values. Prints received message.

## Implementation Details

I will describe the data flow.

The server is run first. It is given a port as an argument. The server waits for clients to connect. Every client that registers with a username is added to the clientList.txt file. CTRL+C on the server prompts the user to confirm shutdown. On confirmation, all .txt files starting with usernames currently in the client list are deleted, clients are disconnected, and the server is shut down.

Clients are run. They are given the IP address and port to connect to. On successful connection to the server, the first 6 receive a welcome message and a list of current users, and are asked to input a username that begins with a letter. The username must be unique.

On successful registration, the client forks, the child listens for input from the server, and the parent listens for input from stdin (the user). The client is presented with a menu with 3 options:-

* + - 1. Get a list of clients on the server
      2. Send a message to another client
      3. Logout

Typing 1 fetches the client list stored on the server, and communicates it to the client within <LOGIN\_LIST></LOGIN\_LIST> tags.

Typing 2 asks the client to enter the username of the client they would like to send the message to. Entering a valid username will lead to the system asking the client which encoding format they would like to use, ‘h’ for hamming or ‘c’ for CRC.

Upon typing ‘h’ or ‘c’, the client is then asked for a frame number where they would like the error to be included. On typing a digit where the error is to be entered, the client is then asked for the message they would like to send.

PLEASE NOTE: To avoid an issue with copy pasting the content from a .txt file, where every ENTER in the txt file triggered a new prompt, I set the delimiter to ` (grave accent or Backtick. Located above the tab key on a keyboard). After typing your message, type ` and hit Enter to submit the message. Same goes for copy pasting a txt file. After pasting, enter ` and hit Enter to submit the message.

On submitting the message, the system will send the message to the producer for processing with the port, the encoder flag, the input, the client username, the “To” username, and the error frame number as parameters.

The producer csteps through every character, and saves them to a buffer. It counts the number of characters read, and every 64 characters, it forks and sends the buffer, the length of the frame, the socket, and “0” as isCap for a non error frame, or “1” as isCap if that frame is to have an error added to it, the encoder flag, the sender username and the receiver username. The frames are sent to encoderService in the child process by calling exec.

If there are <64 characters, the producer sends the remaining characters, the length, socket and “1” or “0” to encoderService via an exec.

The producer also keeps track of how many frames have been sent.

The encoder converts the length of the string into its 7-character binary representation, and all the characters passed in its argument to their ASCII value’s 7-character binary representations. It forks and sends this encoded information, along with the socket to write to, isErrorFrame, encoder flag, sender username and receiver username to the parityAddService via an exec.

The parityAddService calculates the odd parity for each of the 7 character blocks and converts each of them into 8 character blocks. It forks and sends this encoded information, along with the socket to write to, isErrorFrame, encoder flag, sender username and receiver username to the hamming service if encoder flag is ‘h’, or to the buildFrameService if encoder flag is ‘c’, via an exec.

The hamming service calculates the hamming code for each frame. P1,p2,p4,p8,p16 and all the bits in powers of 2 are added to the frame, and the even parity for each of those bits is calculated and set. This helps to check for single bit errors, and by adding the positions of the incorrect parity bits, we can pinpoint the error and correct it.

The buildFrameService creates one long string, with parity bit binary encoded SYN characters (2222) in the front, followed by the data sent as an argument. It then writes to the socket. If the encoder flag is ‘h’, the service will build the <MSG><FROM></FROM><TO></TO><ENCODE></ENCODE><BODY></BODY></MSG> tag frame and write to the socket.

If the encoder flag is ‘c’, the buildFrameService sends the non tag frame to the crcAdd service.

The crcAdd service computes the CRC32 encoding of the entire frame by calculating the remainder of the bitstring with the CRC32 generator. (x32+ x26+ x23+ x22+ x16+ x12+ x11+ x10+ x8+ x7+ x5+ x4+ x2+ x + 1). The remainder is added to the frame, and then the <MSG><FROM></FROM><TO></TO><ENCODE></ENCODE><BODY></BODY></MSG> tag frame is built and written to the socket.

The server receives the tag frame text from the client. The main tag is either <LOGIN\_LIST>, <MSG> or <LOGOUT>

If the server receives <LOGIN\_LIST>, it reads the current clientList.txt file and sends it to the requesting client.

If the server receives <MSG>, It breaks the message down into its FROM, TO, ENCODE and BODY components. The server concatenates the FROM username and TO username and “.txt” to get the chat history file name. The server forks, the child process sends the text within the <BODY>text</BODY>, and the chat history file name to the deframe service if the ENCODE is ‘h’ for hamming, or it sends the text to the crcCheck service if the ENCODE is ‘c’ for CRC. The decoded message will be saved to the c1c2.txt file if c1 sent the message to c2.

The parent process sends the received tag frame message to the recipient client.

If the server receives <LOGOUT>, it deletes all chat history files for the client that is logging out and updates the client list.

The client child listening for input from the server receives the tag frame, and breaks breaks the message down into its FROM, TO, ENCODE and BODY components. It sends the encoded BODY text, the socket, and the Encoder flag to the deframe service if encoder is ‘h’, or to the crcCheck service if the encoder is ‘c’.

The crcCheck service receives a CRC32 encoded frame. It divides the encoded frame by the CRC32 generator and checks the remainder. If the remainder is >0, then it flags an error and prints Data corrupted. It removes the remainder bits from the encoded frame and sends it to the deframe service for further processing.

The deframe service receives a fully encoded frame. It checks the first 2 8 character blocks, and compares them to the SYN encoded string. If they match, then there is no issue in transmission and the service removes those 16 characters, and sends the rest of the frame and the file descriptor to write to, to the checkParityRemoveService via an exec if the encoder flag is ‘c’, or to the hammingDecode service if the encoder flag is ‘h’.

The hammingDecode service receives the hamming encoded frame, and checks all the bits at the powers of 2 positions. P1,p2,p4,p8 and so on. Check 1 skip 1, check 2 skip 2, check 4 skip 4 and so on. It gathers all the parity bits that have incorrect parity. The sum of all the incorrect parity bits indicates the position of the error bit. If an error bit is detected, the service prints the location of the error, and corrects it. The decoded frame is then sent to the checkParityRemoveService for further processing.

The checkParityRemoveService checks each 8 character block for its odd parity, and compares it to the Most Significant Bit. If they match, the data isn’t corrupted. This service grabs the length of the string passed in the encoded message, and loops through the string and extracts 8 character blocks. It checks those blocks for their parity and removes the bit. All blocks are now 7 characters long. This service then forks and exec calls the decoder service, passing the raw encoded data and chat history file to write to.

The decoder service loops through the 7 character blocks and converts them into their ASCII values. These values are stored in a char pointer. Once all the information has been decoded, it prints the message and then the service checks the chat history file input. If the input is ‘\n’, then it doesn’t store to file as it was the client that called this process chain. If the chat history input is anything else, it stores the decoded message to that file location. The service then frees the memory and returns.

# Test Documentation

## Method of Testing

Initial testing began with <64 characters. 11111111 was used to get basic process functionalities in place.

11111111 was then used to test data flow with hamming and crc services implemented.

When 11111111 was correctly processed, the error handling services were integrated into Project 1 workflow. This integration was tested with 11111111 and bugs and issues were fixed.

The test input given in Project 2 was used to reach >64 characters, and functionalities were tested and tweaked to correctly process the data.

The client server implementation was then started. First connection was tested. Once that was working, username features were added and tested. Following the logic, functionalities were added one by one, with appropriate testing.

The tag system was then implemented. Initially <info> tag was used to test correct transmission. Then the <LOGIN> tag was implemented, and logic tested. <LOGIN\_LIST> was next, and on detecting the tag, the server correctly transmitted the contents of the current clientList.txt file. Then <LOGOUT> was implemented, and on detecting that flag, the user was disconnected.

At the end, the first scene of Jurassic Park’s script was used to test correct transmission and error handling. Issue with CRC was discovered and fixed.

Diff between the jurassicPark.inpf and chat history file, correctly identifies area of difference.

Random error was included in the frame number specified by the user.

## Testing Outputs

1. Tag language commands and responses –
   1. <LOGIN\_LIST> - server sends clientList.txt contents to requester
   2. <LOGOUT> - server deletes files associated with client logging out and removes username from client list
   3. <MSG> - server detects <msg> tag and extracts following information
      1. <TO> - receiver username
      2. <FROM> - sender username
      3. <ENCODE> - h for hamming or c for crc
      4. <BODY> - encrypted text message
2. CRC Error Detection – Client is prompted to enter frame number where error is to be randomly inserted. Message received by receiver is checked and Data Corrupted is printed before the frame where the error occurred is printed.
3. Hamming detection and correction – Client is prompted to enter frame number where error is to be randomly inserted. Message received by receiver is checked, and Error At Bit # followed by Correcting Error At Bit # is printed.
4. One to one chat – A large .txt file is copy pasted into the message prompt from client 1, and sent to client 2. Message is printed on both server side and receiver client, and the message is also saved to client1client2.txt

## Test Sets

* 1. Joke.inpf – input data. Contains the quote from project 1.
  2. Intext.txt – input data containing quote from project 2.
  3. jurassicPark.inpf – input file containing the first scene from Jurassic Park’s script.
  4. jurassicPark.compare – Correct fully capitalized output version of Jurassic Park’s script.

# User Documentation

## Source

* My source is in the zip folder. Simply download, navigate to and run server and client from the folder.

## How to run

* Simply navigate to folder with the source code and compiled services.
* Type ./server 9000 in console.
* Open more consoles, and in each one navigate to the folder with the source code and type ./client 127.0.0.1 9000
* Follow the instructions on screen for each client
* Type h for hamming, or c for crc before every message
* Type the frame number to insert a random error.
* Copy and paste the contents of jurassicPark.inpf into the message prompt.
* Run diff username1username2.txt jurassicPark.inpf to check successful transmission and error correction for hamming, or transmission with error for crc.

## Parameters

* Server – Port number
* Client – IP and port number of server