James Fulmer

Daniel Berry

**TCP Chess**

Our program is a simple game of chess that uses Transmission Control Protocol (TCP) to connect to a host server in order to carry out all of the various functions it needs. The program is comprised of 7 different classes: a Server class that needs to be run by the host, an Account class that allows a user to register an account or login to an existing one, a Multiplayer class that gathers a list of all online players and organizes them into a dropdown in order to challenge an opponent, a User class that is used to identify a player, a Cell class that is used by the game, a PasswordHash class which encrypts a user’s password for storage, and the Chess class which is the game and messaging system.

The Server class will constantly be listening for any inbound requests and creates a thread to handle each individual connection. This class also has two of the most important methods in the entire program, SendMessage and ParseCommand. SendMessage is used by many classes to send specific commands to the server and other clients, more on that later. ParseCommand is the method that takes those commands that are sent and makes sense out of them.

The Account class creates a login/register screen for a user to either send their credentials to the server to gain access or create a new account to use. If a user tries to register an account the client will send a message to the server in the form of “REGISTER||username||password||” and then the server will salt and hash the password, check if the username is already taken, and add the user to a csv file with the information given and the date/time of registration. If a user tries to log in, the client will send a command in the form of “LOGIN||username||password||port||” and the server will parse the command, compare the password hash of the corresponding account with the password has of the given password, and keep record of what port the user is using for the connection. With a successful login the user will now access the Multiplayer class.

The Multiplayer class immediately sends a message, “GET\_LIST||username||”, to the server to both inform the server that the user is now online and to get a list of all users currently online. The class then takes the list that the server returns and puts the usernames into a dropdown menu where a user can select and challenge an opponent. When challenging an opponent the client first sends a message to the server, “GET\_USER||opponentname||” in order to get the opponent’s username, IP, and port. The class will then take the username, IP, and port and user it to instantiate a User object that they can reference. The class then uses the opponent’s information to send a message, “CHALLENGE||username||opponentname||”, directly to the user. The opponent then receives your challenge request and pops up a confirmation screen to either accept or decline your offer. If the opponent accepts they will then store your information into their own User object that they can reference, send their confirmation back to the challenger, and open a game using the Chess class.

The Chess class requires two Users, player one and player two, and a boolean variable as parameters. This boolean variable is what determines which player is which color, the challenger is automatically assigned to white and gets to go first in the match. The constructor for this class then goes on to generate the chess board and pieces, and it calls two other methods in order to create a window to display taken pieces and their respective owners and a window to display chat between players and piece movement. While in a game if it isn’t your turn your client is constantly listening for a message from the other player, this could be one of three different messages; “Close||”, “MESSAGE||string||”, or a piece movement, “x1||y1||x2||y2||”. A “CLOSE||” message means that the other player has left the game, “MESSAGE||string||” is a chat message that is then parsed and displayed in the chat window, and piece movements just contain the original position of the piece and the new position of the piece which is parsed and used to update the current state of the board. If a piece is moved the game will also check if there were any opponent’s pieces captured, if there are then they are removed from the board and the graveyard is updated accordingly.

Transmission Control Protocol (TCP) is a transport layer protocol whose features make it incredibly reliable and resilient in transferring data over a connection. TCP is designed to be able to handle the loss of packets with its requirement of receiving acknowledgement or ACKS back from the recipient; if the sender doesn’t receive an ACK within an allotted amount of time it will resend the packet. TCP also uses sequence number to keep packets in order and to find and discard duplicate packets. The protocol also implements a checksum for each packet in order to make sure they aren’t damaged. Another feature of the TCP is “Flow Control”; this is the ability for the host on the receiving end of a connection to negotiate the amount of data being sent. Flow control is accomplished by having the receiver return a “window size” with each of its ACKs; this tells the sender how much space the receiver has to work with at any given moment and allows the sender to adjust accordingly. (<https://www.ietf.org/rfc/rfc793.txt>) TCP can, however, be less effective in wireless communications because of there being a lot more opportunities for a packet to get lost or damaged without a cable keeping it moving toward its destination.

Unfortunately no program can be perfect and ours is no different, there are bugs... The majority of these bugs aren’t even technically bugs they’re just things we didn’t get around to implementing. One bug, though, occurs in the chat window during a chess game, if the window fills up completely any future messages will not be visible, they are being added to the area but we never allowed for scrolling. Another bug is the fact that when it’s your turn in a game you can’t receive chat messages until after their next turn. We worked around this by adding any chat messages sent to a queue of sorts and sending them out alongside piece movements. There are also many features we wish we had time to implement, for example: a notification of check/checkmate, castling, en passant, pawn promotion, prevention of illegal movements while in check, a notification of victory/defeat, and encrypting messages to the server.

In our program, our security could definitely be improved, but we aren’t entirely insecure. It is clear that our program is very vulnerable to attackers who can capture our packets, we send all of our commands and messages in plain text, passwords included. On the other hand, however, we do use SHA-512 salt and hash algorithm to encrypt all of our passwords as we store them.