A Distributed Card Game with a Replicated File System

CS 550: Advanced Operating Systems
November 30th, 2017

Brandon Dotson
A20313199

Table of Contents

Introduction

The purpose of this project was to create a card game that utilized a distributed system while having a replicated file system embedded within.

Background

Before going into detail about the implementation, I believe it is a good idea to provide some background information about the card game I choose and the concept of replicated file systems.

The game chosen for this project was Blackjack, a game played with a dealer and a varying number of players. The object of Blackjack is to score more points than the dealer without going beyond 21 points. To start the game, the dealer deals two cards to everyone at the table. Once this is complete, each player has an opportunity to draw additional cards to improve their hand, with the dealer having his turn once the players have finished. The round ends with, the dealer comparing his hand against the players' to determine the winner(s). The following table has the card values:

Card Type	Value
Numerical Cards	Worth the same as the number on the card
Face Cards	Worth 10 points
Aces	Worth either 1 or 11*

*The ace is worth 11 only when adding 11 to the score would not cause it to go over 21.

Table 1: Card Values

There are additional rules that may be used in Blackjack. For example, you may bet double on the next card draw or split your hand into two and bet on both separately. Since this project does not involve betting and has limitations on how the “player” behavior programmed, the advanced rules were not included in the implementation.

In addition to the basic rules of Blackjack, some information about replicated file systems should be known. The basis of a replicated file system is to have “back-up” servers that hold copies of the files on your main server. Having additional copies of important files is useful in that the main server now has added redundancy in case things do not go as planned. If the main server were to go offline unexpectedly, all important files would still be accessible. Additionally, having copies may also serve as a small countermeasure to those with the ability to access and modify the files without permission. If the originals were modified, a simple comparison with the copies would show that there were unauthorized changes.

Implementation

This project relied on a library called rpclib to implement remote procedure calls for the distributed system. The system was made up of one main server, two back-up servers, and up to four clients, with each of the clients acting a player for the card game. The main server was centralized and used five threads to handle game management and server-client interactions.

The server starts by creating a deck of cards in the form of a vector of strings. Each string is made up of two characters that corresponds to one of the 52 cards in a standard deck. For example, the string “2C” would represent the 2 of Clubs while “KD” would represent the King of Diamonds. Using the vector’s shuffle function, the cards are randomized and the

server, now acting as the dealer for the game, waits for 4 clients to join and become players. Once the players are present, the server waits for 5 seconds before beginning.

At the start of a round, the server pops strings off the vector and allocates them to the players and dealer. This is done by opening a file and assigning the values of the popped strings to different player slots, following the order one would in an actual game. Once the card strings are assigned to the players, the file is copied to the back-up servers and shared with each of the players. At this point, each of the players is given an opportunity to act. When it is a player's turn, they search the file for their player number, a number from 1 – 4 representing the player id, and convert their cards into a numerical sum. If the total value is lower than 17, they request a new card from the dealer until they reach at least 17. If they are already at 17, they decide not to act. Once the players have completed their turns, the dealer decides to whether it should draw more cards before comparing its hand with those of the players. This process repeats until there are not enough cards to start a new round.

Results

The following figures show the output of the server and client during runtime. The server images display the process of beginning a game and dealing out cards while the client images convey the decisions of each of the clients.

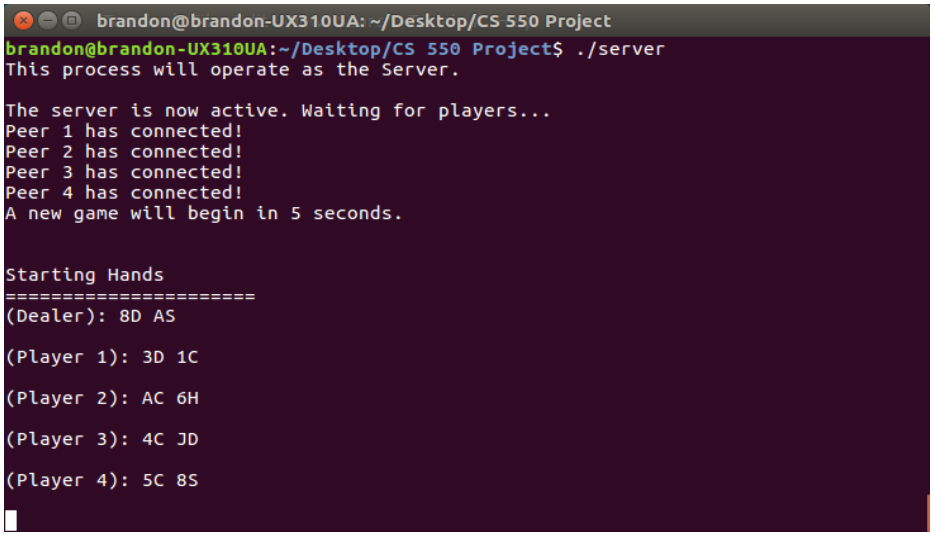

Figure 1: Server Output 1

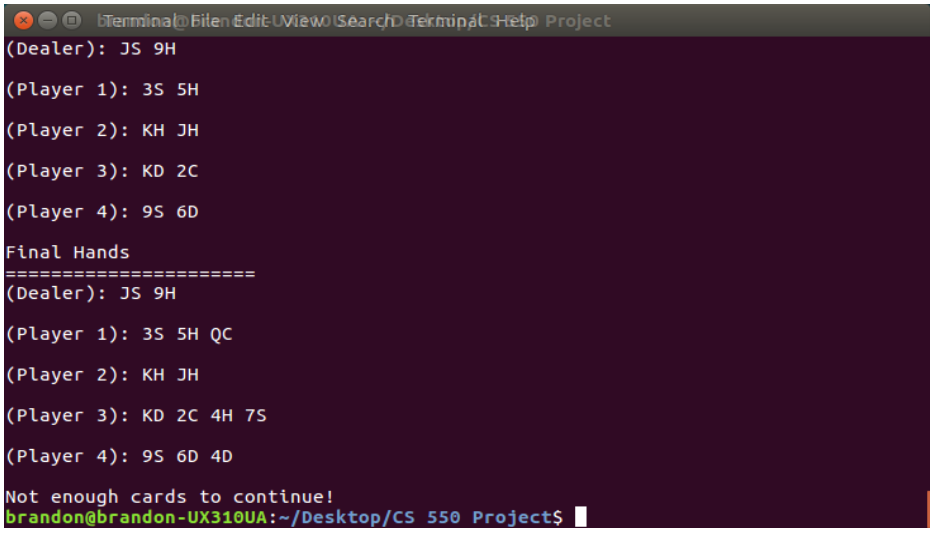

Figure 2: Server Output 2

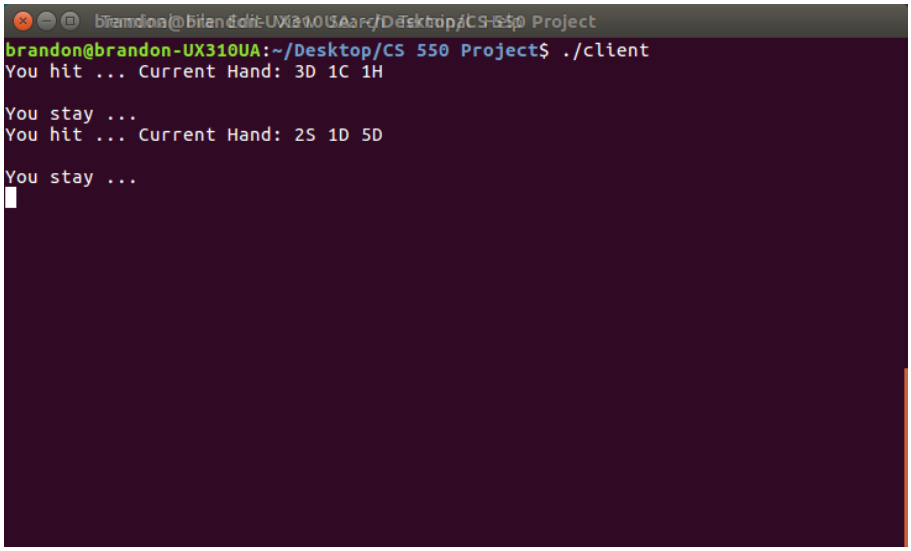

Figure 3: Client Output 1

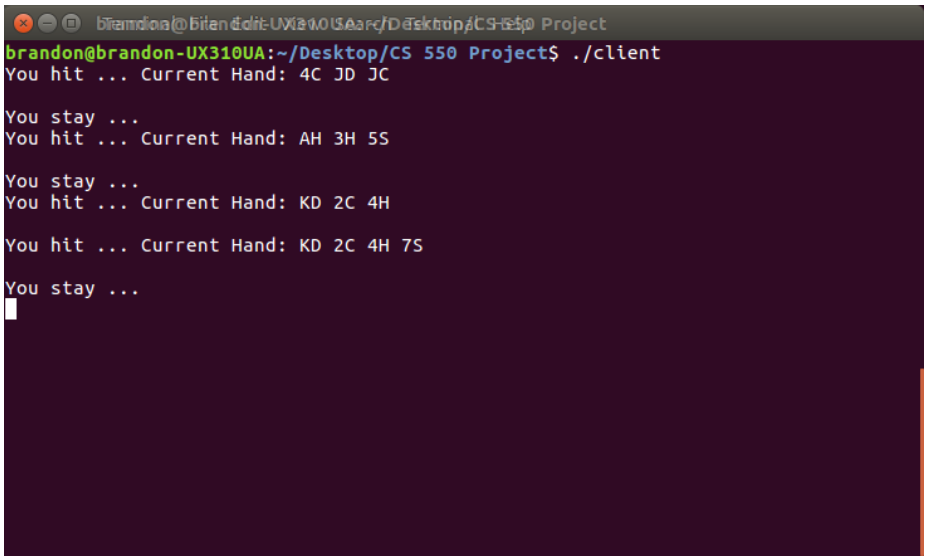

Figure 4: Client Output 2

Discussion

As technology advances, we rely more on distributed systems. Many of the popular websites have many servers, and a majority of online, multiplayer games use client-server models for communication. Consequently, many opportunities for exploiting these systems have also to appeared. Realizing this connection between concepts from our class and leisurely entertainment is what initially attracted me to this type of project.

While working on this project, I was able to familiarize myself with the important aspects of remote procedure calls. I also had the opportunity to program a card game for the first time, only having experience with basic, text-based games before. Finally, I learned a little more about network and system topology in order to have servers and clients communicate properly.

Challenges

While developing the project, I had several challenges. One of the main issues was the limitations of the library I used for remote procedure calls. One case of this, is that the client/server objects created were not capable of being copied. This meant there was some reliance on global variables because of my inability to pass them to functions. There were also synchronization issues with the main server and clients. Once the games were complete, there was a chance that one of the four clients was left running without any way of closing properly. Finally, I wanted to add graphics to make it easier to recognize the game flow, but the actual implementation was difficult to achieve.

Improvements

The following improvements were considered:

- Allow clients to connect/disconnect to server in between rounds
- Allow clients to behave differently using a “risk factor”
- Introduce advanced Blackjack rules
- Include endless rounds and improved deck shuffling variety
- Add graphical representation for game flow
- Make text output for server/client more clear

Sources

RPClib Library: http://rpclib.net

Blackjack Background: https://en.wikipedia.org/wiki/Blackjack

SDL Library: https://www.libsdl.org/