Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Six Card Golf - Socket Project Final Report

Matthew Bulger

Manager Protocol - Format and Order of Messages

The following requests define the protocol for communicating with the manager process. This protocol uses a "request-response" format for communication. There is a single manager with a well-known, static IP address and port (4500 by default) receiving requests, and an arbitrary number of client processes receiving responses to their requests. The manager process listens for incoming messages from any of the clients and responds to them with the responses defined below. No additional acknowledgment messages should be sent back to the manager on receipt of the response; any acknowledgment sent will be incorrectly interpreted as a query to execute.

All requests and responses are strings encoded in the UTF-8 format. If at any point, the command is not recognized, rather than one of the responses below, the response “SYNTAX_ERROR” is returned.

Note that for all of the below requests and responses, double quotes (“ ”) indicates a message or subset of a message being sent. The quotes are not actually included in the messages.

Register Player

Registers a user with the manager by correlating a username with the network address (IP address and port) they will be listening at.

Request: *“register <username> <port>”*

<username> must be between 1-15 characters long and unique. *<port>* must be an integer and unique.

Note the user’s IP address does not need to be specified explicitly in the request, it will be taken from the datagram header. As well, the *<port>* is the smallest port the user has access to, but each client is actually allocated the next 4 ports as well (total of 5 ports) to be able to open multiple sockets each.

Response 1: *“SUCCESS”*

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

A response of "SUCCESS" indicates the username and port were unique, and successfully registered with the manager. The user with a corresponding username/port will be eligible to be assigned to a game.

Response 2: *"FAILURE PORT"*

This response indicates the manager was unsuccessful at registering the user, as the port has already been registered to another user. The client should re-register with the same username, but different port.

Response 3: *"FAILURE USERNAME"*

This response indicates the manager was unsuccessful at registering the user because another user already has registered the same username. It is possible the port specified was invalid, but the client should resolve the username issue before the port issue, hence this response takes priority.

Time Space Diagram

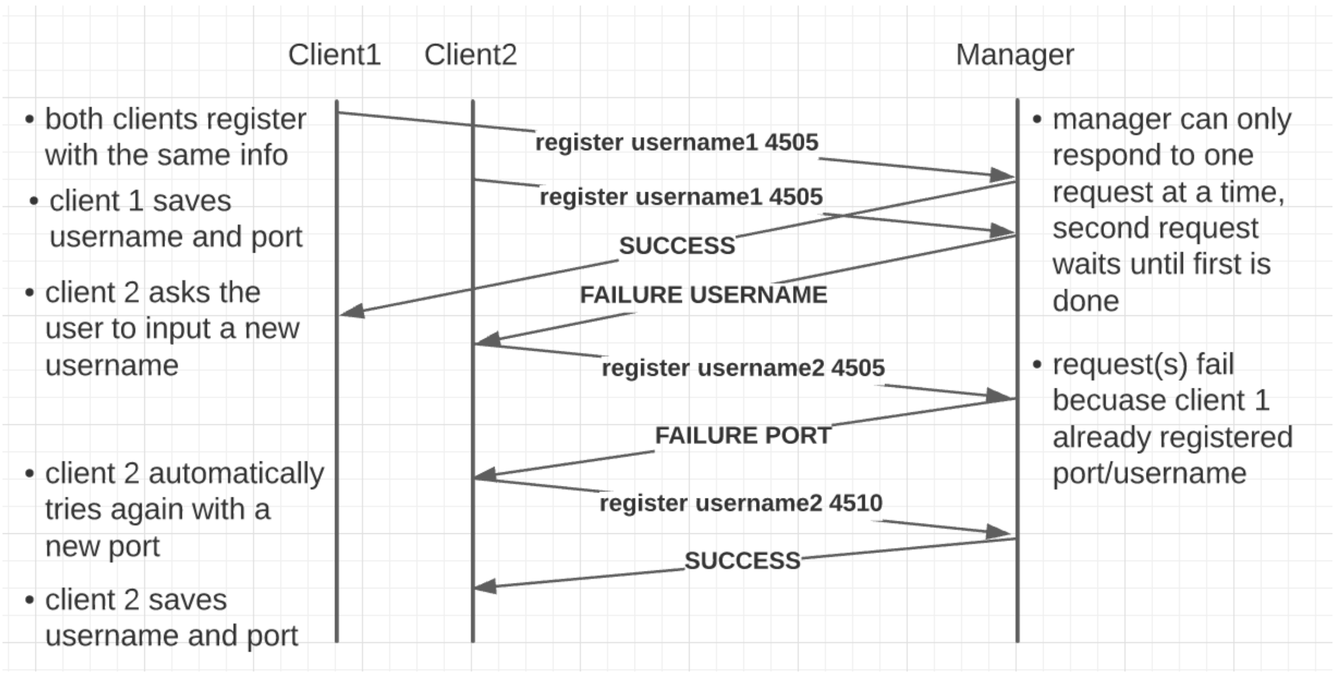

Client1
Client2
Manager
• both clients register with the same info
• client 1 saves username and port
• client 2 asks the user to input a new username
• client 2 automatically tries again with a new port
• client 2 saves username and port
register username1 4505
register username1 4505
SUCCESS
FAILURE USERNAME
register username2 4505
FAILURE PORT
register username2 4510
SUCCESS
• manager can only respond to one request at a time, second request waits until first is done
• request(s) fail becuase client 1 already registered port/username

Figure 1: Time Space Diagram of the "Register Player" request.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Query Players

Requests the number of players registers with the manager, as well as their user info (username and network addresses).

Request: *"query players"*

This request has no parameters.

Response: *"<num_players><players>"*

<num_players> is an integer of how many players are registered with the manager

This is followed by any number of players *<players>*, which have the format *"\n(<username>, <ip_address>, <port>)"*. Note the newline at the beginning, meaning each parameter and player will be separated by a single newline.

Time Space Diagram

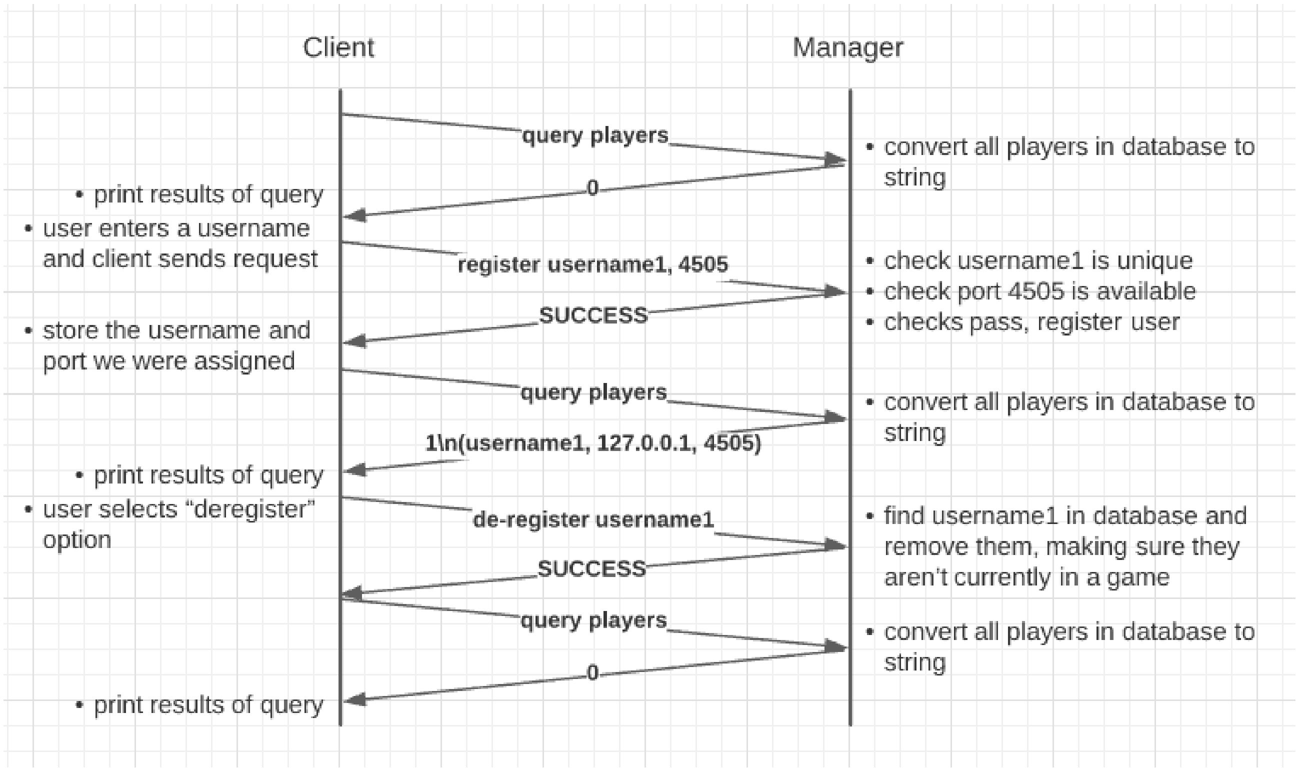

Client
Manager
query players
• convert all players in database to string
0
• print results of query
• user enters a username and client sends request
register username1, 4505
• check username1 is unique
• check port 4505 is available
• checks pass, register user
SUCCESS
• store the username and port we were assigned
query players
• convert all players in database to string
1\n(username1, 127.0.0.1, 4505)
• print results of query
• user selects "deregister" option
de-register username1
• find username1 in database and remove them, making sure they aren't currently in a game
SUCCESS
query players
• convert all players in database to string
0
• print results of query

Figure 2: Time Space Diagram of the "Query Players" request.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Query Games

Requests the number of games that are currently in progress, and for each game in progress, gets the users involved with that game.

Request: *"query games"*

This request has no parameters.

Response: *"<num_games><games>"*

<num_games> is an integer number of games that are currently ongoing, which is followed by the same number of *<games>*. Each of these games have the format *"\n(<game_id>, <dealer_username>, [<players>])"*. *<players>* is a list of each player in the game, including the dealer, each having the format *"(<username>, <ip_address>, <port>)"*, which each player separated by a newline. *<game_id>* is a unique integer id assigned to this game.

Time Space Diagram

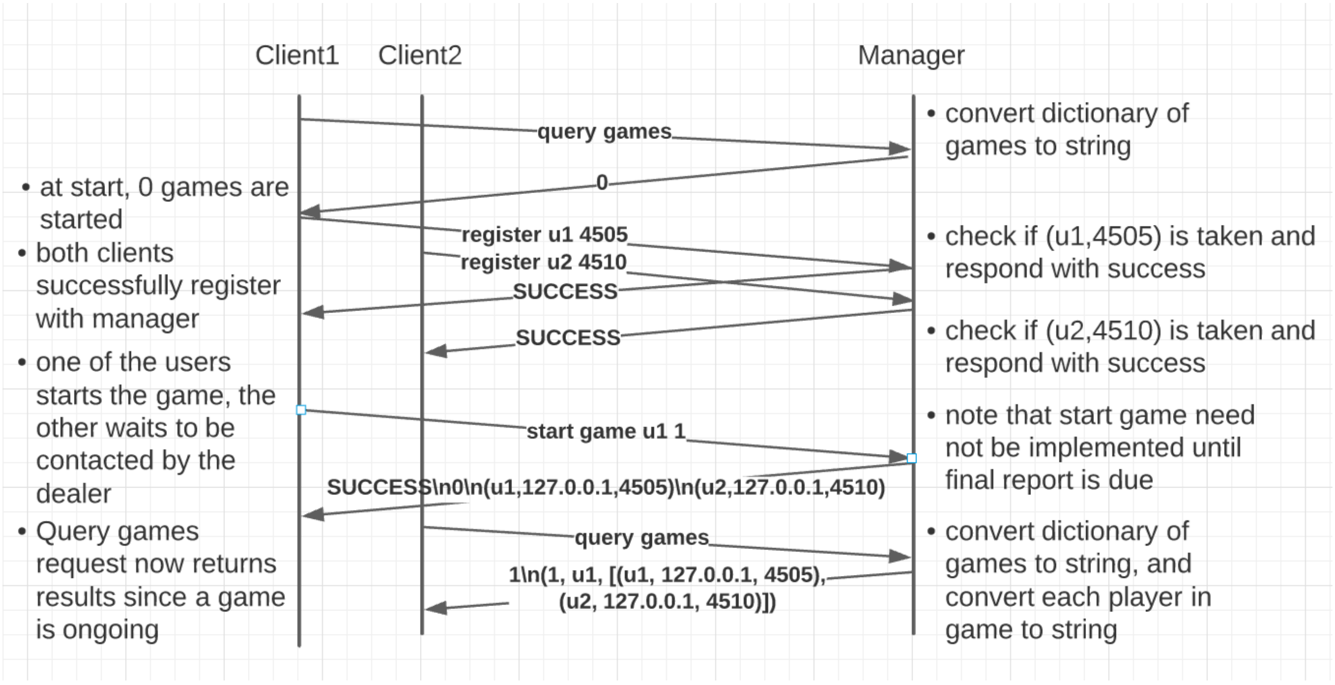

Client1
Client2
Manager
query games
convert dictionary of games to string
0
at start, 0 games are started
register u1 4505
register u2 4510
both clients successfully register with manager
check if (u1,4505) is taken and respond with success
SUCCESS
SUCCESS
check if (u2,4510) is taken and respond with success
one of the users starts the game, the other waits to be contacted by the dealer
start game u1 1
note that start game need not be implemented until final report is due
SUCCESS\n0\n(u1,127.0.0.1,4505)\n(u2,127.0.0.1,4510)
query games
Query games request now returns results since a game is ongoing
convert dictionary of games to string, and convert each player in game to string
1\n(1, u1, [(u1, 127.0.0.1, 4505), (u2, 127.0.0.1, 4510)])

Figure 3: Time Space Diagram of the "Query Games" request.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

De-Register Player

Stops the manager from assigning a specific user to any new games, and frees the username/port to be used by a new user who joins.

Request: *“de-register <username>”*

<username> must be the username of a user who has already successfully registered by calling the “Register Player” request.

Response 1: *“SUCCESS”*

This response indicates the username was found successfully, and that username and port block has been de-allocated. The client should close all of its open sockets and exit to allow another client who joins to use those resources.

Response 2: *“FAILURE”*

This response indicates the de-registration was not successful. This is usually because you are trying to de-register a username that has not previously been registered with the “Register Player” request above.

Time Space Diagram

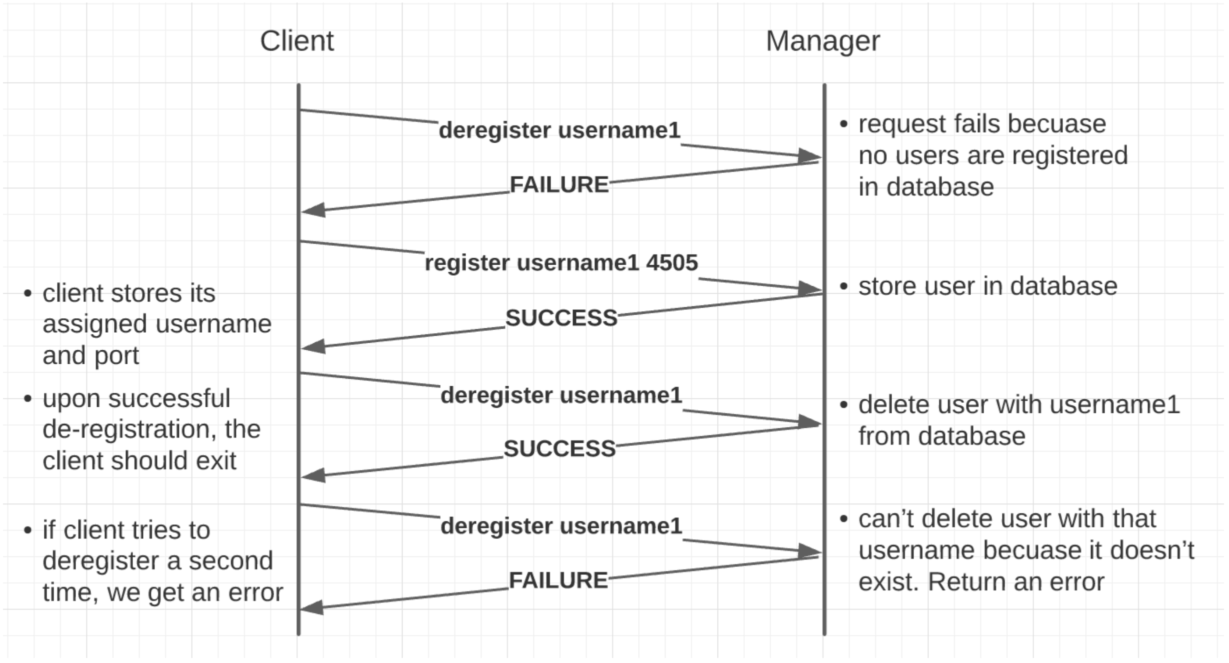

Client
Manager
deregister username1
• request fails becuase no users are registered in database
FAILURE
register username1 4505
• store user in database
• client stores its assigned username and port
SUCCESS
deregister username1
• upon successful de-registration, the client should exit
• delete user with username1 from database
SUCCESS
deregister username1
• can’t delete user with that username becuase it doesn’t exist. Return an error
• if client tries to deregister a second time, we get an error
FAILURE

Figure 4: Time Space Diagram of the "De-Register Player" request.

Start Game

Starts a new instance of a game of Six Card Golf. The player process sending the request becomes the dealer of the new game. The manager assigns *k* additional players to the game by marking them as "in-game", which prevents players from joining two games at the same time. Upon receipt, of a successful response, the dealer is responsible for contacting the other players and playing the game using the player protocol below.

Request: *"start game <username> <k>"*

*<username>*is the name of the user starting the game, who will be the dealer of the new game.

Response: *"FAILURE"*

This response indicates that the game was unable to be started because there were not enough players registered with the manager who are not already in a game.

Response: *"SUCCESS\n<game_id>\n<players>"*

<players> is a list of each player in the game, including the dealer, each having the format "(*<username>*, *<ip_address>*, *<port>*)", which each player separated by a newline. *<game_id>* is a unique integer id assigned to this game.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Time Space Diagram

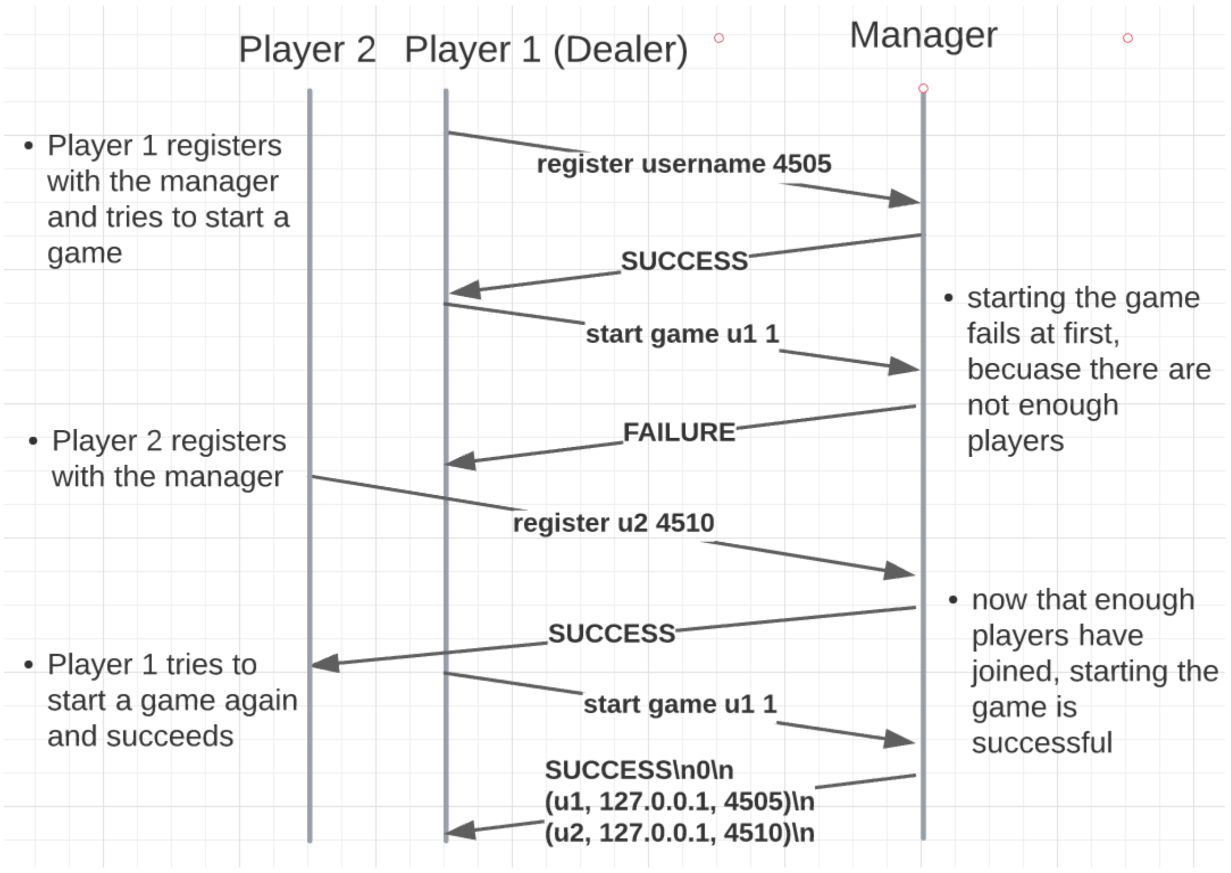

Player 2
Player 1 (Dealer)
Manager
• Player 1 registers with the manager and tries to start a game
register username 4505
SUCCESS
start game u1 1
• starting the game fails at first, becuase there are not enough players
FAILURE
• Player 2 registers with the manager
register u2 4510
SUCCESS
• now that enough players have joined, starting the game is successful
• Player 1 tries to start a game again and succeeds
start game u1 1
SUCCESS\n0\n
(u1, 127.0.0.1, 4505)\n
(u2, 127.0.0.1, 4510)\n

Figure 5: Time Space Diagram of the "Start Game" request.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

End Game

The dealer sends this request to the manager after the game they started with the *"Start Game"* request has completed all nine rounds, and a winner has been selected.

Request: *"end <game_id> <username>"*

<game_id> is the ID of the game that is ending, which was assigned to the dealer in the *"Start Game"* request. *<username>* is the name of the dealer who originally started the game.

Response: *"SUCCESS"*

This response indicates the game was ended successfully. All players who participated are no longer marked as "in-game", so they are now able to be assigned to a new game.

Response: *"FAILURE"*

This response indicates there was an error with the request; either the *<game_id>* does not exist, or the *<username>* is not of the dealer who started the game originally.

Time Space Diagram

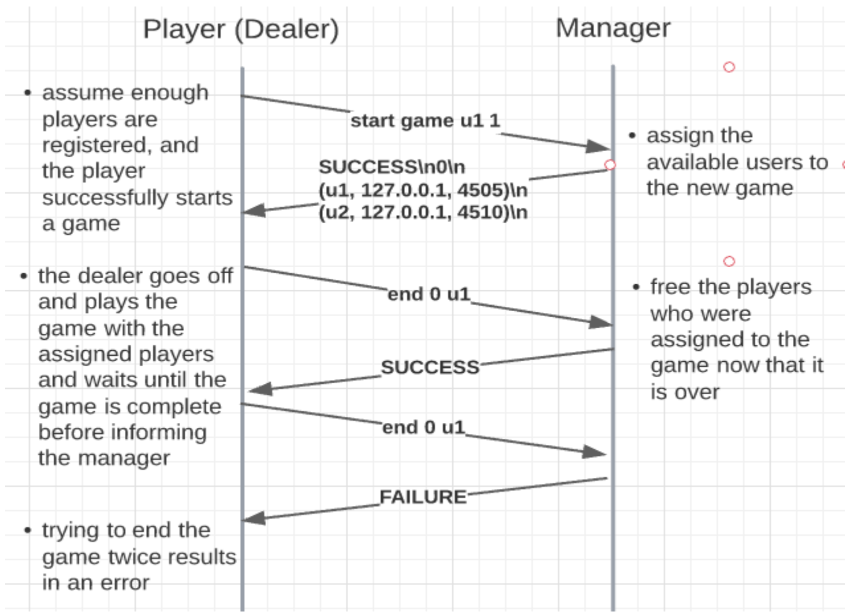

Player (Dealer)
Manager
• assume enough players are registered, and the player successfully starts a game
start game u1 1
• assign the available users to the new game
SUCCESS\n0\n (u1, 127.0.0.1, 4505)\n (u2, 127.0.0.1, 4510)\n
• the dealer goes off and plays the game with the assigned players and waits until the game is complete before informing the manager
end 0 u1
• free the players who were assigned to the game now that it is over
SUCCESS
end 0 u1
FAILURE
• trying to end the game twice results in an error

Figure 6: Time Space Diagram of the "End Game" request.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Player Protocol - Format and Order of Messages

The following messages/commands define the protocol used by the player process for playing my implementation of Six Card Golf. Each player process must first register with the manager process, which has a well-known, static IP address and port (4500 by default). A player can then opt to wait for a game to start, or start a new game, becoming the dealer of a new game. The dealer then assigns each player to the new game, shuffles the deck of cards, and broadcasts this data to every player process, allowing the game to begin. For a more detailed look at the relationship between these messages, see *Figure 13* which is a Finite State Automata representing the player process application logic.

This protocol uses a string format for each message. The first token of the string will correspond to one of the below messages, followed by some number of parameters depending on the request, which must be decoded on the receiving end. All strings are encoded in UTF-8 format. If at any point, the received command does not match the next expected command type, that message will be dropped with no response sent.

Unless otherwise noted, each request is meant to be broadcast to all players. That is, the same message should be sent to all players, and an acknowledgment should be received back from each to verify that all clients are up to date with the same information before moving on, to prevent synchronization issues.

Assign Player

The "Assign Player" command is sent by the dealer of a game. After creating a new game via the *"start game"* command with the manager, the dealer uses the list of players it received to send each of them a message indicating which game they have joined, and the other players in the game, so that each player can communicate with each other directly (including the dealer itself).

Message: *"assign player\n<extension>\n<num_players>\n<players>"*

<num_players> is the integer number of players in the *<players>* list, where *<players> is a list of <player> with the format* "(*<username>*, *<IP_address>*, *<port>*)". Each player is separated by a newline character. *<extension>* is a boolean; if true, the player extension mode will be used for this game, i.e. stealing cards from other players is allowed.

Response 1: *"ack assign player\n<extension>\n<num_players>\n<players>"*

The acknowledgment response is the same as the message received, with "ack" prepended.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Time Space Diagram

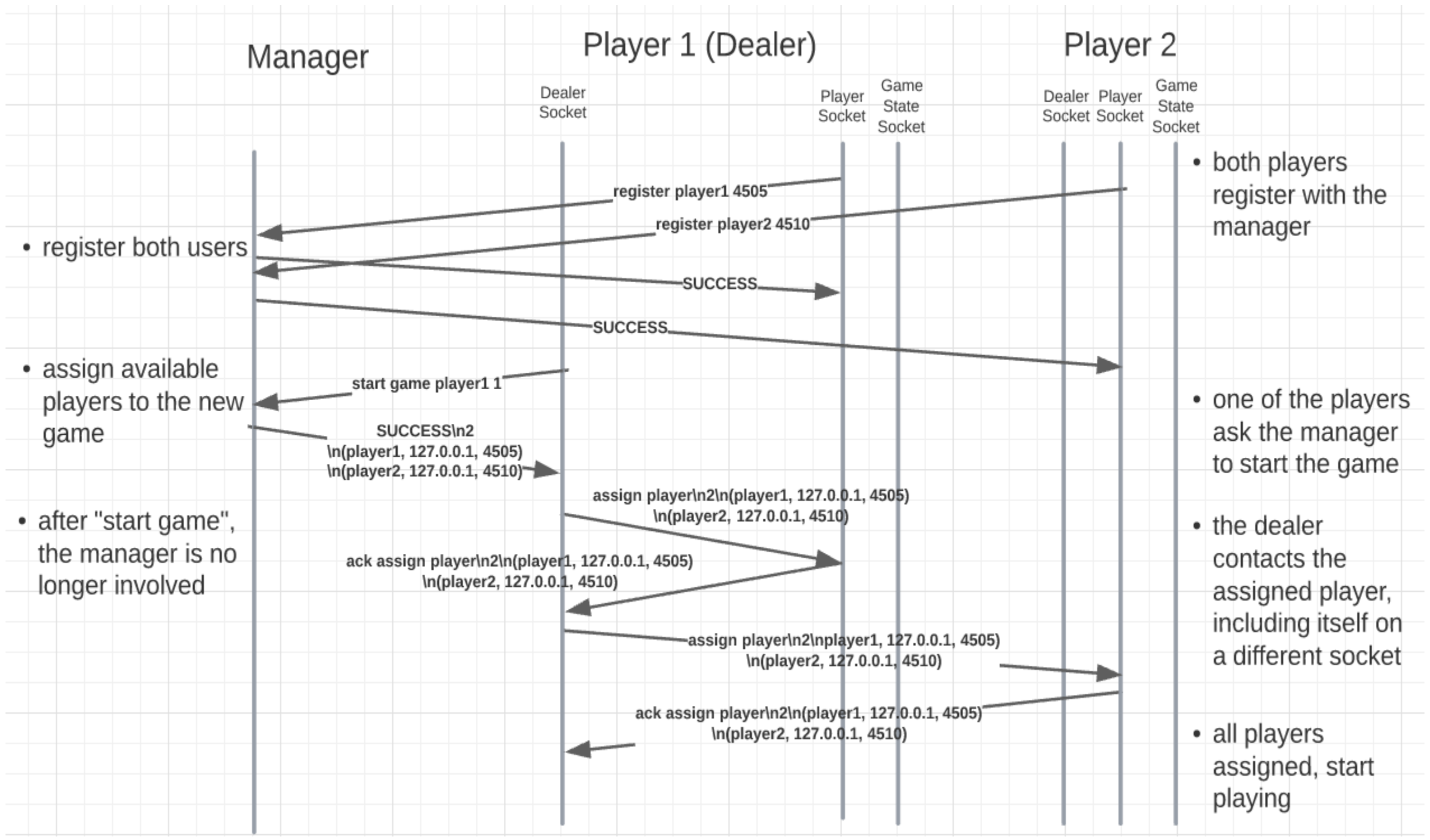

Manager
Player 1 (Dealer)
Player 2
Dealer Socket
Player Socket
Game State Socket
Dealer Socket
Player Socket
Game State Socket
register player1 4505
register player2 4510
SUCCESS
SUCCESS
start game player1 1
SUCCESS\n2 \n(player1, 127.0.0.1, 4505) \n(player2, 127.0.0.1, 4510)
assign player\n2\n(player1, 127.0.0.1, 4505) \n(player2, 127.0.0.1, 4510)
ack assign player\n2\n(player1, 127.0.0.1, 4505) \n(player2, 127.0.0.1, 4510)
assign player\n2\nplayer1, 127.0.0.1, 4505) \n(player2, 127.0.0.1, 4510)
ack assign player\n2\n(player1, 127.0.0.1, 4505) \n(player2, 127.0.0.1, 4510)
• register both users
• assign available players to the new game
• after "start game", the manager is no longer involved
• both players register with the manager
• one of the players ask the manager to start the game
• the dealer contacts the assigned player, including itself on a different socket
• all players assigned, start playing

Figure 7: Time Space Diagram of the "Assign Player" request.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Deal Card

Informs a player that a single specific card is being dealt from the shuffled deck to a specific player. This command is invoked only by the dealer of the game at the beginning of a round, immediately after generating the deck of cards and shuffling them. This message is not just sent to the player being dealt the card, but to all players so they can display the current state of everyone's hand (despite all cards being hidden initially). Upon receipt, a player should update their game state to append this card to the specific user's card array.

Message: *"deal card <card> <player>"*

<*card*> is the value of a specific card encoded in the format "(<*value*>, <*hidden*>)". <*value*> is the suit and value of the card combined as a string (i.e. "H10"), while <*hidden*> is a boolean representing whether the card is turned over or not. <*player*> is the username of the player receiving the card.

Response: *"ack deal card <card> <players>"*

The acknowledgment response is the same as the message received, with "ack" prepended.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Time Space Diagram

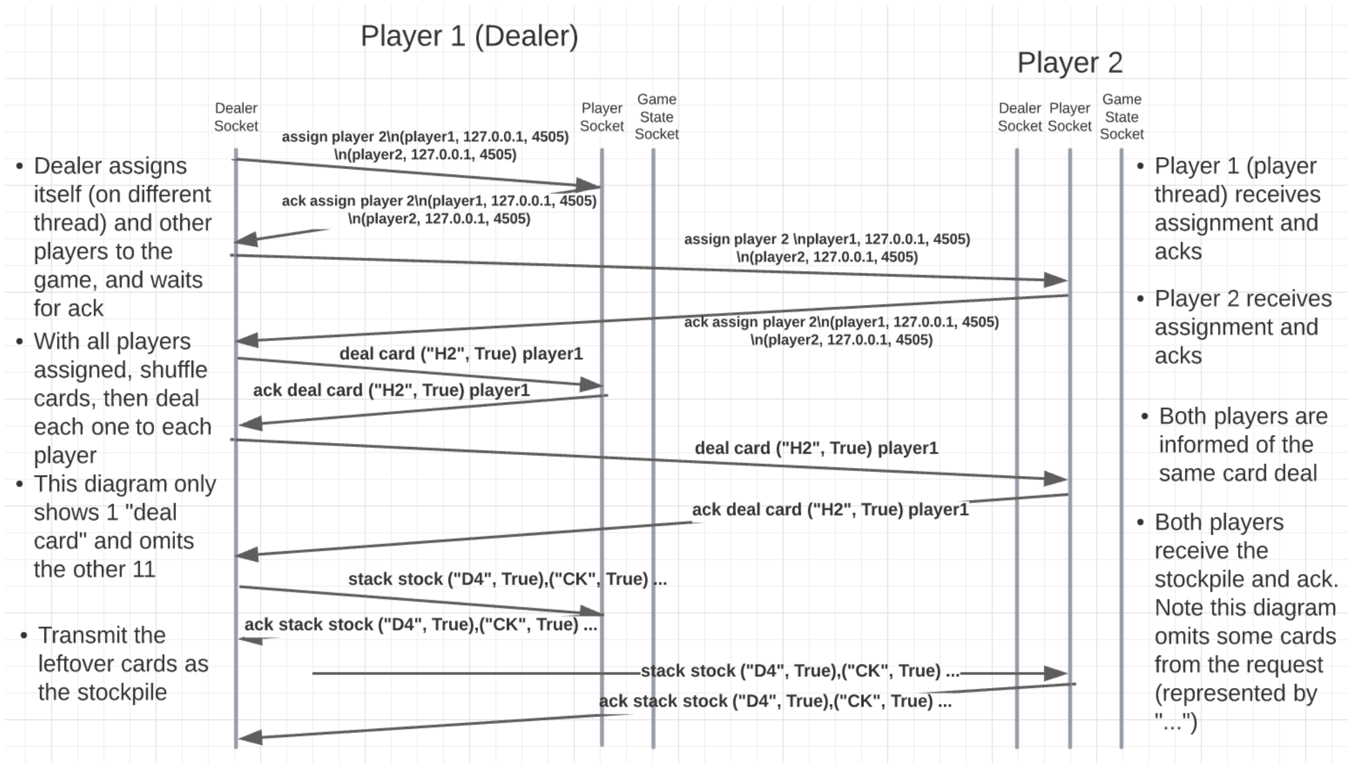

Player 1 (Dealer)
Player 2
Dealer Socket
Player Socket
Game State Socket
Dealer Socket
Player Socket
Game State Socket
assign player 2\n(player1, 127.0.0.1, 4505) \n(player2, 127.0.0.1, 4505)
ack assign player 2\n(player1, 127.0.0.1, 4505) \n(player2, 127.0.0.1, 4505)
assign player 2 \nplayer1, 127.0.0.1, 4505) \n(player2, 127.0.0.1, 4505)
ack assign player 2\n(player1, 127.0.0.1, 4505) \n(player2, 127.0.0.1, 4505)
deal card ("H2", True) player1
ack deal card ("H2", True) player1
deal card ("H2", True) player1
ack deal card ("H2", True) player1
stack stock ("D4", True),("CK", True) ...
ack stack stock ("D4", True),("CK", True) ...
stack stock ("D4", True),("CK", True) ...
ack stack stock ("D4", True),("CK", True) ...
• Dealer assigns itself (on different thread) and other players to the game, and waits for ack
• With all players assigned, shuffle cards, then deal each one to each player
• This diagram only shows 1 "deal card" and omits the other 11
• Transmit the leftover cards as the stockpile
• Player 1 (player thread) receives assignment and acks
• Player 2 receives assignment and acks
• Both players are informed of the same card deal
• Both players receive the stockpile and ack. Note this diagram omits some cards from the request (represented by "...")

Figure 8: Time Space Diagram of the "Deal Card" request.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Send Stack of Cards

This command sends an entire list of cards from the dealer to a player process, as opposed to the above *"deal card"* command, which sends only a single card. This command is sent by the dealer to every player to inform them of the state of the stockpile, which are the cards leftover after dealing six cards to every player. Upon receipt, a player should clear their current stockpile array, and replace it with the cards encoded in the message.

Message: *"stack <stack_type> <cards>"*

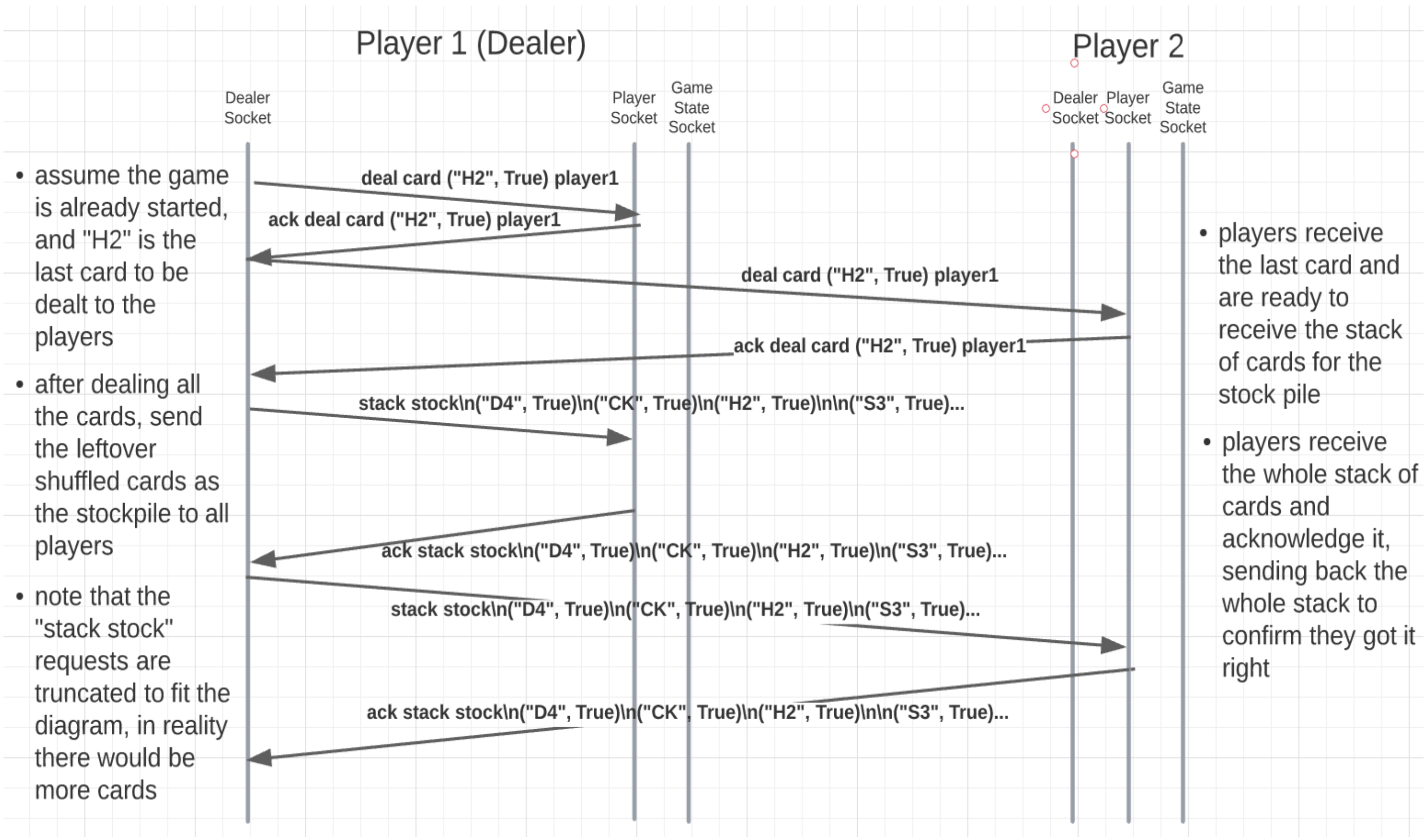
<stack_type> is the destination of the listed cards. Usually this is "stock", but "discard" is also valid. *<cards>* is a newline-separated list of cards, where each card has the format "(*<value>*, *<hidden>*)". *<value>* is the suit and value of the card combined as a string (i.e. "H10"), while *<hidden>* is a boolean representing whether the card is turned over or not.

Response 1: *"ack stack <stack_type> <cards>"*

The acknowledgment response is the same as the message received, with "ack" prepended.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Time Space Diagram



Player 1 (Dealer)
Player 2
Dealer Socket
Player Socket
Game State Socket
Dealer Socket
Player Socket
Game State Socket
• assume the game is already started, and "H2" is the last card to be dealt to the players
• after dealing all the cards, send the leftover shuffled cards as the stockpile to all players
• note that the "stack stock" requests are truncated to fit the diagram, in reality there would be more cards
deal card ("H2", True) player1
ack deal card ("H2", True) player1
deal card ("H2", True) player1
ack deal card ("H2", True) player1
stack stock\n("D4", True)\n("CK", True)\n("H2", True)\n\n("S3", True)...
ack stack stock\n("D4", True)\n("CK", True)\n("H2", True)\n("S3", True)...
stack stock\n("D4", True)\n("CK", True)\n("H2", True)\n("S3", True)...
ack stack stock\n("D4", True)\n("CK", True)\n("H2", True)\n\n("S3", True)...
• players receive the last card and are ready to receive the stack of cards for the stock pile
• players receive the whole stack of cards and acknowledge it, sending back the whole stack to confirm they got it right

Figure 9: Time Space Diagram of the "Send Stack" request.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Announce Reveal Cards & Reveal Card

These two commands, *"announce initial reveal"* and *"reveal card"* are used at the beginning of the game to synchronize the player processes, and allow each of the players to select two cards to turn over, as the cards are initially dealt face down.

Message: *"announce initial reveal"*

This message is broadcast to all players to let them know they can start to pick two cards to turn over. Having this command ensures that no players start revealing their cards before every player has received all their cards. If a player were to start revealing their cards before a player is ready, it will not receive those messages, so this command prevents such a scenario.

Response 1: *"ack announce initial reveal"*

Response sent from receiving player back to the dealer confirming receipt of the broadcast. Once a player sends this response, it can start sending *"reveal card"* messages (below).

Message/Response 2: *"reveal card <card> <username>"*

<card> is an integer between 0-5, representing the index in the array of one of *<username>*'s cards. Since the cards have already been distributed, it is easier to refer to the cards based on their index relative to a username as opposite to doing a linear search through the whole dictionary. On receipt, the indicated card's "hidden" property should be set to False.

Response 1: *"reveal card <card> <username>"*

Response sent from receiving player back to the dealer confirming receipt of the broadcast.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Time Space Diagram

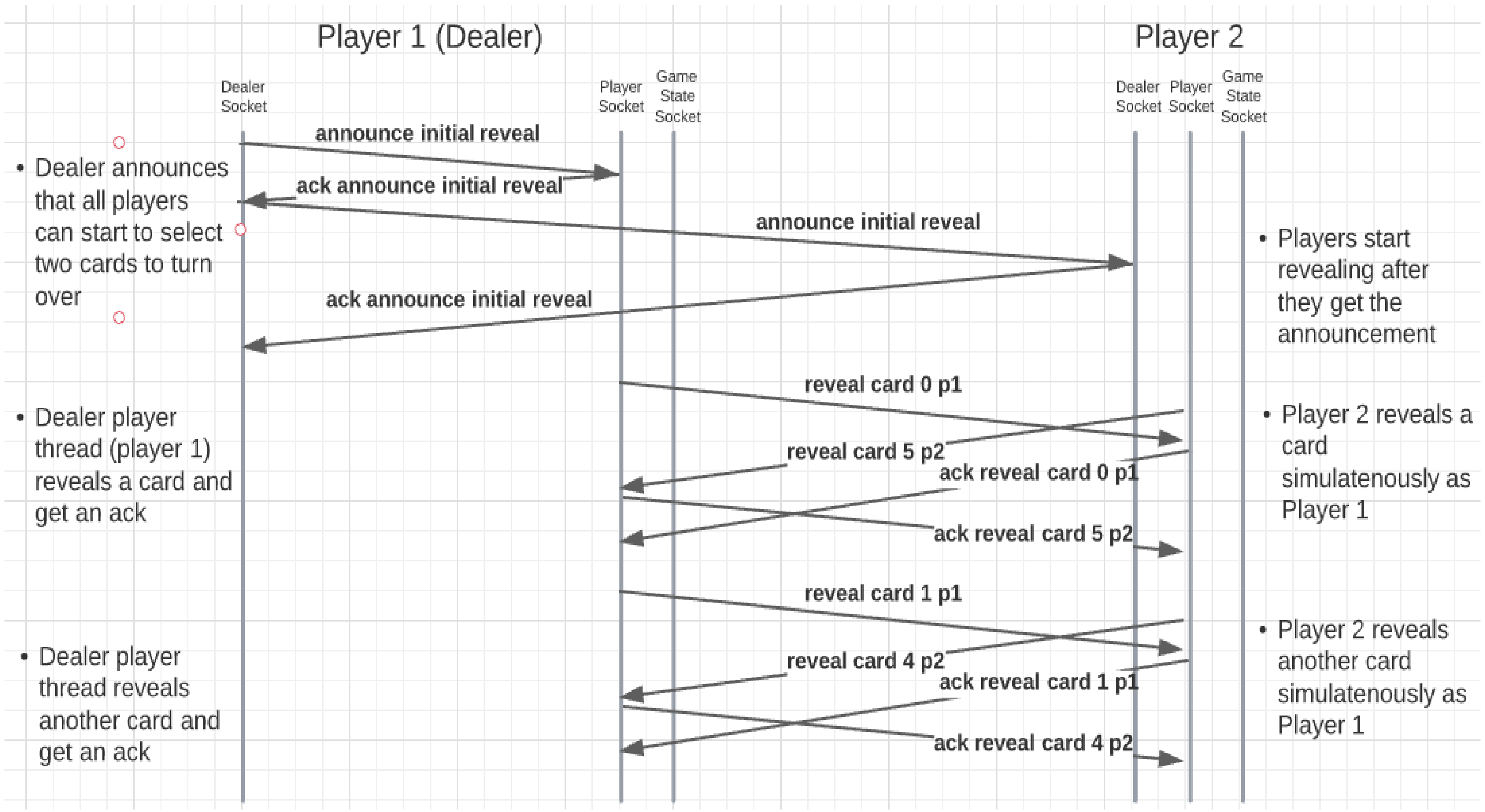

Player 1 (Dealer)
Player 2
Dealer Socket
Player Socket
Game State Socket
Dealer Socket
Player Socket
Game State Socket
announce initial reveal
ack announce initial reveal
announce initial reveal
ack announce initial reveal
• Dealer announces that all players can start to select two cards to turn over
• Players start revealing after they get the announcement
reveal card 0 p1
reveal card 5 p2
ack reveal card 0 p1
ack reveal card 5 p2
• Dealer player thread (player 1) reveals a card and get an ack
• Player 2 reveals a card simulatenously as Player 1
reveal card 1 p1
reveal card 4 p2
ack reveal card 1 p1
ack reveal card 4 p2
• Dealer player thread reveals another card and get an ack
• Player 2 reveals another card simulatenously as Player 1

Figure 10: Time Space Diagram of the "Announce Reveal Cards"& "Reveal Card" requests.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Pop Card & Replace Card

This command represents the "main move" or game play loop of Six Card Golf; when it is their turn, a player can *"pop"* from the stockpile, the discard pile, or, in the player extension mode, steal a card from another player. When a *"pop"* broadcast is received, all players store the indicated card in the "held card" position. The same player then decides which card to replace the "held" card using the *"replace"* command.

Message 1: *"pop <type>"*

<*type*> can take three values: "discard", "stock", or "steal":

- *"pop discard"*

  Takes the top card from the discard pile and puts it in the "held card" position.

- *"pop stock"*

  Takes the top card from the stock pile and puts it in the "held card" position.

- *"pop steal <victim_username> <victim_card> <stealer_username> <stealer_card>"*

  This move steals the card at index <*victim_card*>, an integer between 0-5, from <*victim_username*>, swapping it with their own card at index <*stealer_card*>. The stolen card must be a revealed card, while the swapped card must be a hidden card. Note that when this option is used, the *"replace"* command below is skipped.

Response 1: *"ack pop <type>"*

The acknowledgment response is the same as the message received, with "ack" prepended. After receiving all acknowledgments, the player whose turn it is can send the next message, *"replace"* (below).

Message 2: *"replace <card> <username>"*

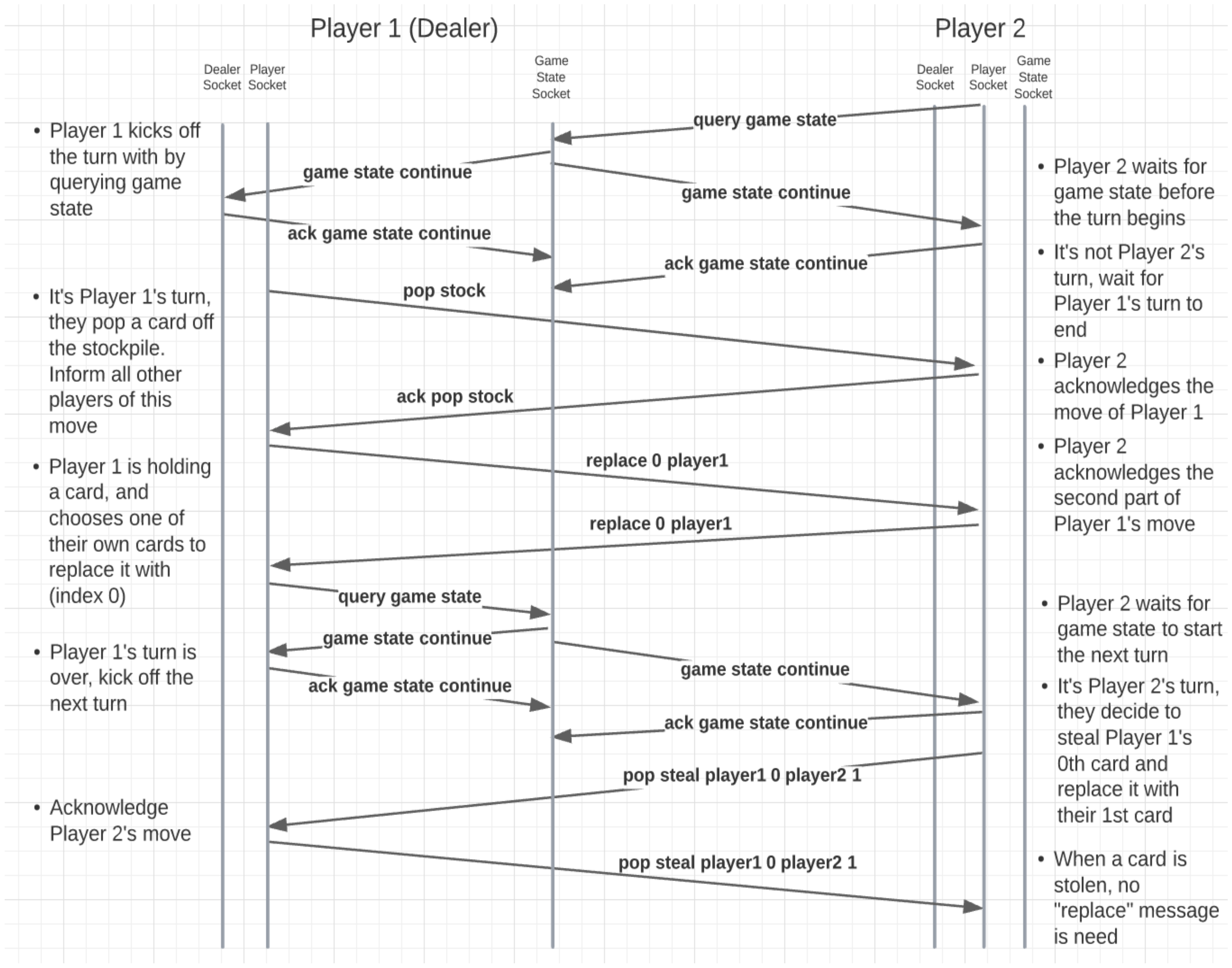
Indicates that the currently held card (which was set in the above *"pop"* command) should be replaced with the card at index <*card*> in the <*username*>'s card array. Upon receipt, players should swap the held card with the indicated card, and put the indicated card on top of the discard pile (face up).

Response 2: *"ack replace <card> <username>"*

The acknowledgment response is the same as the message received, with "ack" prepended.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Time Space Diagram



Player 1 (Dealer)
Player 2
Dealer Socket
Player Socket
Game State Socket
Dealer Socket
Player Socket
Game State Socket
query game state
game state continue
game state continue
ack game state continue
ack game state continue
pop stock
ack pop stock
replace 0 player1
replace 0 player1
query game state
game state continue
game state continue
ack game state continue
ack game state continue
pop steal player1 0 player2 1
pop steal player1 0 player2 1
• Player 1 kicks off the turn with by querying game state
• It's Player 1's turn, they pop a card off the stockpile. Inform all other players of this move
• Player 1 is holding a card, and chooses one of their own cards to replace it with (index 0)
• Player 1's turn is over, kick off the next turn
• Acknowledge Player 2's move
• Player 2 waits for game state before the turn begins
• It's not Player 2's turn, wait for Player 1's turn to end
• Player 2 acknowledges the move of Player 1
• Player 2 acknowledges the second part of Player 1's move
• Player 2 waits for game state to start the next turn
• It's Player 2's turn, they decide to steal Player 1's 0th card and replace it with their 1st card
• When a card is stolen, no "replace" message is need

Figure 11: Time Space Diagram of the "Pop" and “Replace” requests.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Query Game State & Game State

At the start of each round, a player is not allowed to make a move until it receives the current game state from the dealer. The game state indicates whether the round should continue, a new round is starting, or all nine rounds are over, which is the end of the game. This helps to synchronize the players, and also ensure all players are connected before starting the game in the first place.

Message 1: *"query game state"*

Rather than sending this message to the normal dealer address/port, a player querying the game state should send their request to the dealer's port *plus 1*. This is the Dealer's game state port. A third port is used because the dealer port is actively used throughout the game, so binding a new port allows the dealer to spawn a totally separate listener thread that is only responsible for handling *"query game state"* requests. This drastically reduces the complexity of implementation.

Message 2/Response: *"game state <state>"*

<*state*> can take one of three values: "continue", "end", "newround"

- *"game state continue"*

  This response indicates the current round is not yet complete, as there are still players with hidden cards. The turn should move to the next player, and the round should continue.

- *"game state newround"*

  This response indicates the current round is over, because all of the players' cards are turned over, but the game is not over because there have not been 9 rounds. The dealer should re-shuffle and re-deal the cards.

- *"game state end <winner>"*

  This response indicates both the round and game are over. The <*winner*> *username* should be displayed, and the players should return back to the main menu.

Not only is this response sent to the player who initiated *"query game state"*, it is also broadcast to all other players, who will also be waiting for the game state at the start of every turn. Note this response is sent to the normal player port.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Time Space Diagram

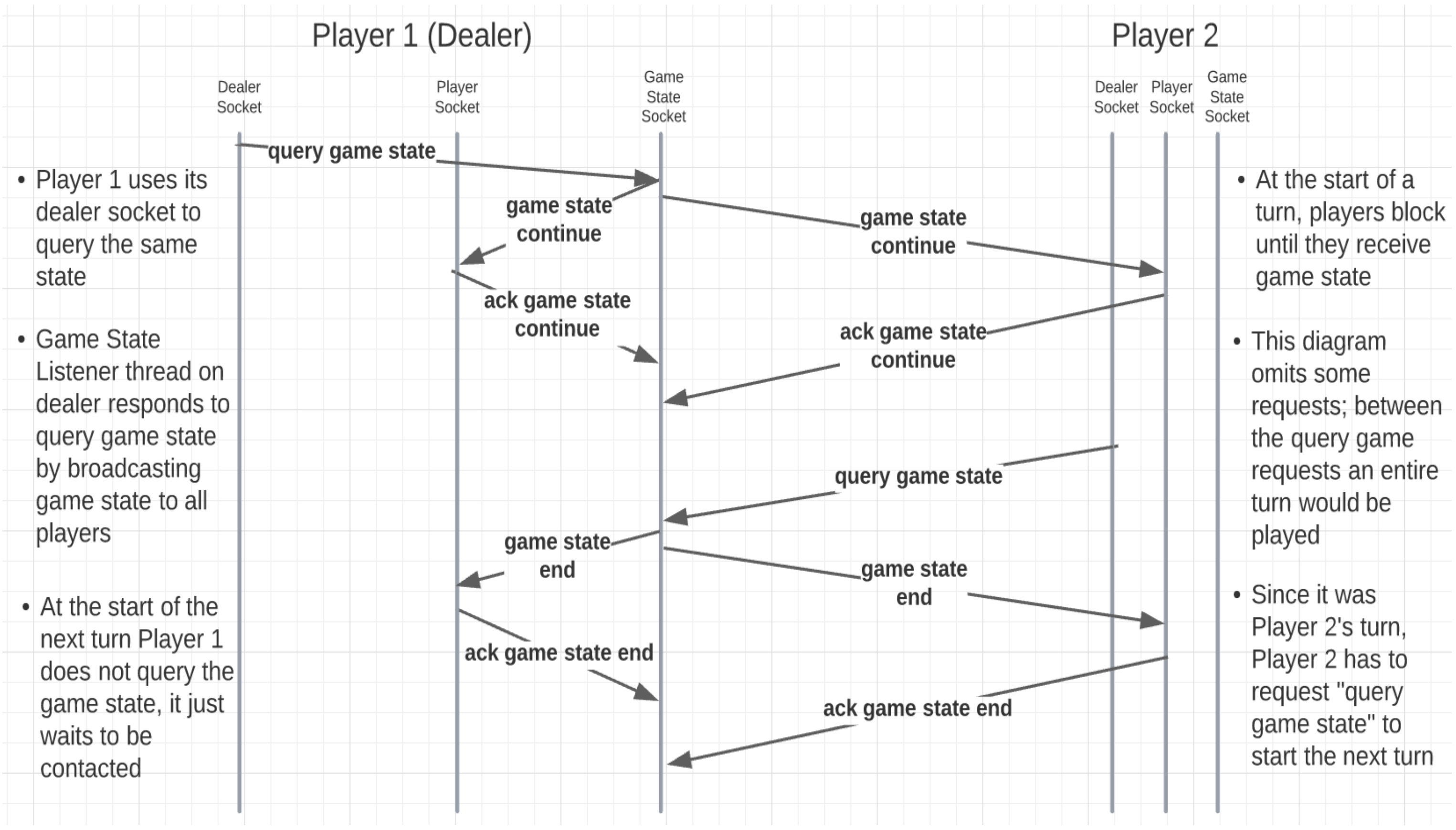

Player 1 (Dealer)
Player 2
Dealer Socket
Player Socket
Game State Socket
Dealer Socket
Player Socket
Game State Socket
query game state
game state continue
game state continue
ack game state continue
ack game state continue
query game state
game state end
game state end
ack game state end
ack game state end
• Player 1 uses its dealer socket to query the same state
• Game State Listener thread on dealer responds to query game state by broadcasting game state to all players
• At the start of the next turn Player 1 does not query the game state, it just waits to be contacted
• At the start of a turn, players block until they receive game state
• This diagram omits some requests; between the query game requests an entire turn would be played
• Since it was Player 2's turn, Player 2 has to request "query game state" to start the next turn

Figure 12: Time Space Diagram of the "Query Game State" & "Game State" requests.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Player Extension

For the player extension component of this project, I chose to implement the ability to “steal” cards from other players. When it is it their turn, rather than having two moves, “Draw from Stock” and “Draw from Discard”, in the player extension mode, there is now a third option, “Steal from Another Player”. When the player selects this option, they are prompted to select which player they want to steal from, the “index” of the card to steal, and the “index” of the player’s card to swap it with. The “index” is simply a position of the player’s current hand, with “0” being the top left card, and “5” being the bottom right card. Using the index over the literal card value is mostly helpful when playing the game manually, as it requires less typing, but from the computer AI perspective, there is no difference. When the player finalizes the move they want to make, a *“pop steal”* message is broadcast to all of the players, containing the information of the two cards that need to be swapped. It is important to note that you can only steal cards that are visible (face-up), and you can only swap the stolen card with one of your own cards that is face-down. This rule tweak and added validation prevents an infinite loop where the players continually steal the same cards back and forth instead of exposing all of their cards, which ends the round. In the normal game play mode, all the game play is the same, but you do not have the option to steal cards, only draw from the stockpile and discard pile.

To play a game in both modes, it is the responsibility of the dealer. When starting a game, you must enter how many other players should join, as well as whether the game should be the player extension mode or not. Given the user’s response to this, the dealer will inform all other players as it assigns them to the game which mode the game is so all the players are on the same page.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Design and Implementation Considerations

While the design of the Six Card Golf application was certainly challenging, it ultimately came together to form a functional and fun game card game. As I noted in my milestone report, I decided to use Python for this project because the syntax for setting up sockets is a lot simpler compared to C++, and my unfamiliarity with networking projects meant that I didn't want to have to deal with memory allocation. The manager protocol is a relatively simple client-server model, but the player protocol is purely peer-to-peer. Since the manager protocol used string messages for everything, I decided to use the same message format for each of my messages as opposed to using objects, even though in hindsight that would have made a lot of things easier, as using strings required encoding/decoding every card in a specific format that would not have been the case if I had simply sent my objects as bytes.

When designing the player process, and my Six Card Golf implementation as a whole, rather than immediately focusing on the code, I considered what it would be like to play this game with someone over the phone. Both players would first need to each a lot of messages to get the initial state of the cards correct, including the order of the stockpile and which cards each player has, but once the setup is complete, the players just need to tell each other the action they are taking, i.e. "I'm taking a card from the stockpile and replacing it with the card in the bottom left". In essence, this is the same framework I used in my implementation; have each of the players keep track of the entire game state, and take turns broadcasting to all other players the change they made during their move. Another reason I went with this strategy was in thinking about how online games work in real life. If each player is the sole source of truth of their cards, it would theoretically be possible to "lie" about your cards to other players; instead, with each player tracking the game state, they keep each other accountable for what is a valid move, ensuring the game state is always valid.

To implement the player, I needed to make several helper methods that were frequently used in letting communicate with each other, namely "broadcast" and "wait_for_command". The broadcast function loops through the addresses of all the players in the same game, and sends each of them a designated message, while waiting to receive an acknowledgment message back from each as well. Wrapping this functionality up made implementing acknowledgment responses for each of the above commands far simpler, and with far less repeated code that I was using initially. The next function I created was "wait_for_command", which is basically the opposite; it waits for a message (that is presumably being broadcast by another player), ignoring messages until we get the command we're expecting, which is simply the first token in the payload. As you can see in the Finite State Automata, there are many points where players are simply waiting for another player to finish something, like deciding what move to make, so again, abstracting away this complexity was very helpful.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

A huge consideration I had to make while creating this project was multi-threading. While the instructions listed multi-threading as optional, I found it to be almost a necessity, because waiting to receive a message on a socket is a blocking function call. This means that in order to receive messages from peers at the same time as you sent them, a child listener thread becomes necessary. Furthermore, multi-threading is also needed for the dealer. Since the dealer participates in the game themselves, and the dealer is responsible for dealing the cards to each player, a dealer player thread is needed to listen for the incoming requests, even though the dealer is really just sending messages to itself.

As soon as I added multi-threading, it became abundantly clear that one port per player would not be enough. Each player needs a way of communicating not only with the dealer to receive their cards, but they also need to interact with every other player to inform them of what move they are making when it is their turn. On the dealer side, I found the easiest way to distinguish between messages intended for the dealer and for the player (which, remember, are running on different threads of the same process) was not through the format of the messages themselves, but by delivering the messages to separate sockets. If only one thread is monitoring a socket at a time, there is no chance of race conditions or dropping a packet that was intended for the other role. Furthermore, I also found that a third socket was useful in my implementation. Since the dealer and player sockets are already used throughout the game, I wanted a new thread whose only responsibility was to monitor the current game state and detect when to move to the next round (all cards are turned over) or if someone has won the game (all nine rounds are over). While I theoretically could have made this work with the dealer socket, since I already allocated 5 ports to each process just in case, adding a new socket only for game state requests/messages also simplified the implementation. With all of these sockets set up, alongside the helper methods I mentioned above, the networking aspect of this project was resolved, and the rest of the implementation focused mostly on perfecting the game logic, such as how the cards are scored, ensure the computer does not select a move that is invalid or impossible, and moving the cards around when you have multiples of the same value.

The last hurdle I faced with this project was getting the computer AI functional. When I first began implementing the project, I assumed that while the computer was sending messages back and forth, an actual person needed to input the move each player would make. Eventually, the requirements were clarified to mean that the computers should play with each other automatically, deciding on the moves themselves. However, in the end, this modification did not end up being a major blocker, as I was able to implement a system that allows the computer to pick among choices that the human would have made because I already needed some validation to ensure the user could not input a wrong value. In fact, starting with a focus on the human aspect and these validation rules ultimately made the application far less fragile.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

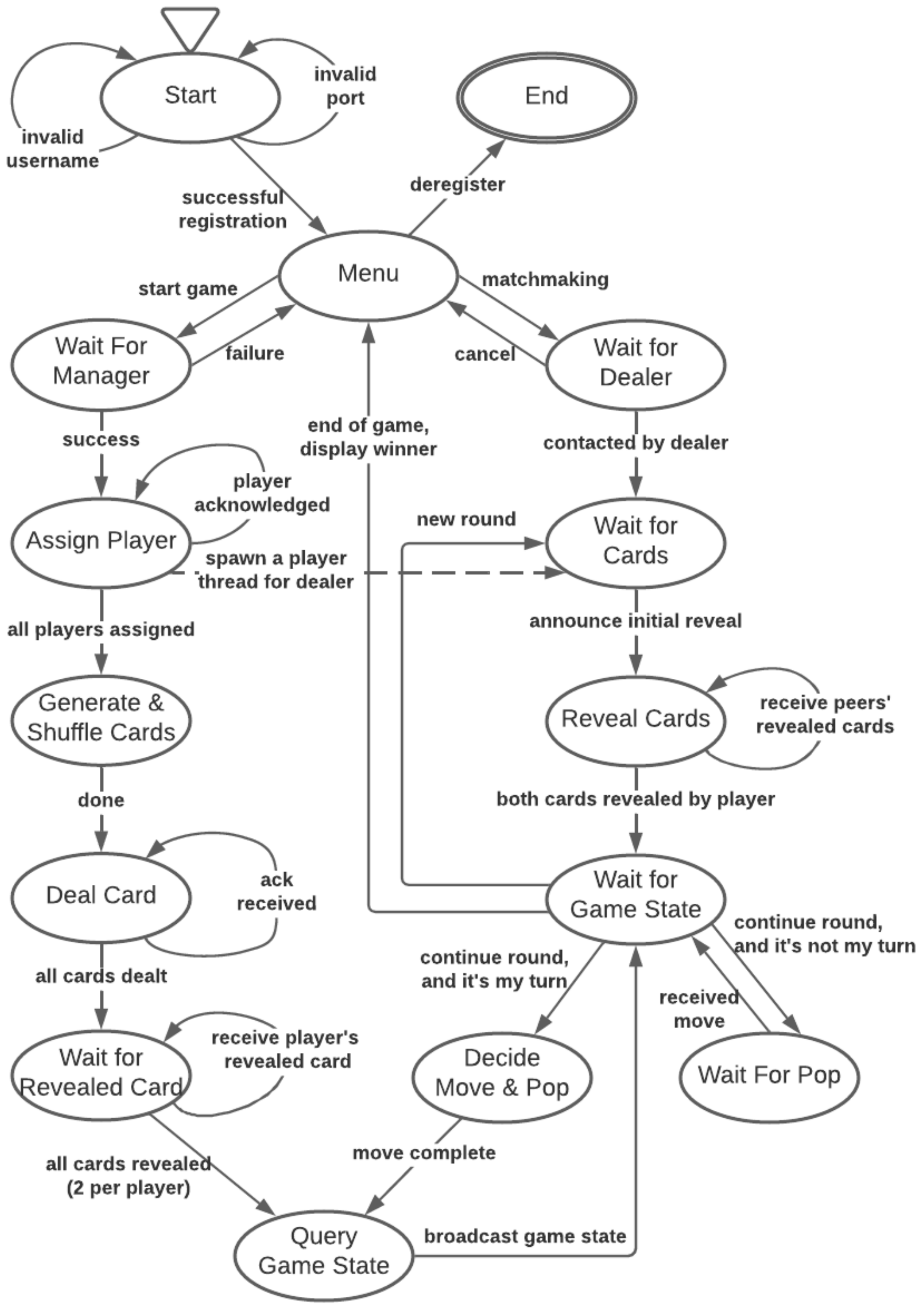
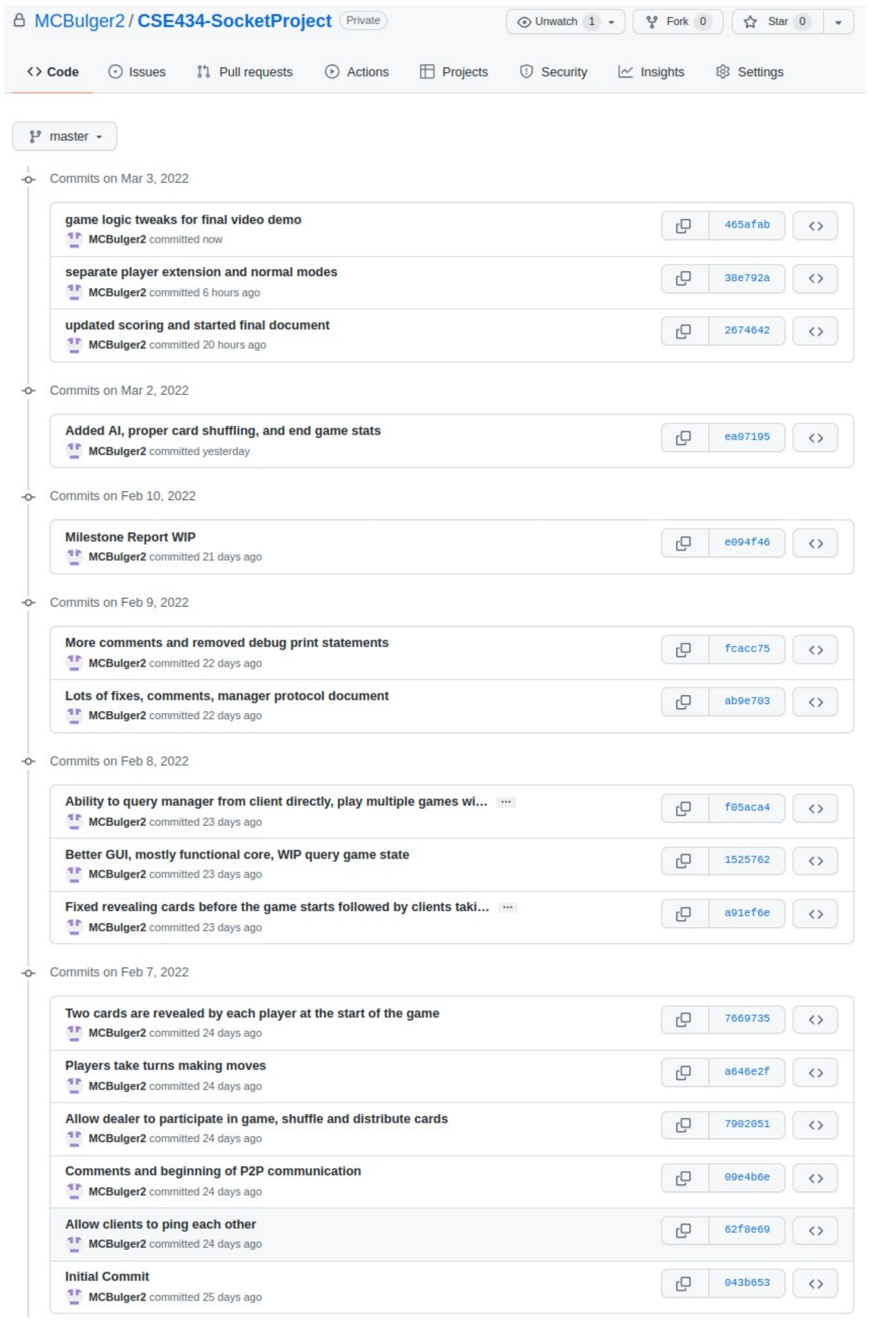


Figure 13: Finite State Automata of the player process.

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Version Control

A private GitHub repository was my choice for version control on this project. Please note that I was able to complete a lot more functionality than was required by the milestone before 2/13/22, which explains why there were fewer commits after the milestone.


MCBulger2 / **CSE434-SocketProject** Private

Unwatch 1 · Fork 0 · Star 0

Code · Issues · Pull requests · Actions · Projects · Security · Insights · Settings

master

Commits on Mar 3, 2022

- **game logic tweaks for final video demo** — MCBulger2 committed now — 465afab
- **separate player extension and normal modes** — MCBulger2 committed 6 hours ago — 38e792a
- **updated scoring and started final document** — MCBulger2 committed 20 hours ago — 2674642

Commits on Mar 2, 2022

- **Added AI, proper card shuffling, and end game stats** — MCBulger2 committed yesterday — ea07195

Commits on Feb 10, 2022

- **Milestone Report WIP** — MCBulger2 committed 21 days ago — e094f46

Commits on Feb 9, 2022

- **More comments and removed debug print statements** — MCBulger2 committed 22 days ago — fcacc75
- **Lots of fixes, comments, manager protocol document** — MCBulger2 committed 22 days ago — ab9e703

Commits on Feb 8, 2022

- **Ability to query manager from client directly, play multiple games wi…** — MCBulger2 committed 23 days ago — f05aca4
- **Better GUI, mostly functional core, WIP query game state** — MCBulger2 committed 23 days ago — 1525762
- **Fixed revealing cards before the game starts followed by clients taki…** — MCBulger2 committed 23 days ago — a91ef6e

Commits on Feb 7, 2022

- **Two cards are revealed by each player at the start of the game** — MCBulger2 committed 24 days ago — 7669735
- **Players take turns making moves** — MCBulger2 committed 24 days ago — a646e2f
- **Allow dealer to participate in game, shuffle and distribute cards** — MCBulger2 committed 24 days ago — 7902051
- **Comments and beginning of P2P communication** — MCBulger2 committed 24 days ago — 09e4b6e
- **Allow clients to ping each other** — MCBulger2 committed 24 days ago — 62f8e69
- **Initial Commit** — MCBulger2 committed 25 days ago — 043b653

Matthew Bulger
2 March 2022
Dr. Syrotiuk
CSE 434

Video Demonstration

Video demo link: https://www.youtube.com/watch?v=p-LpVZj3nac

Timestamps

(a) Compile the manager and player programs. - Not applicable since I used Python which is interpreted, not compiled. (0:30)

(b) Run the freshly compiled programs on at least four distinct end-hosts. (0:40)

(c) Registration of sufficient player processes to start two concurrent games of Six Card Golf, with and without the player extension component, with at least two players each. (1:07)

(d) Use another player to query both players and games. (2:29)

(e) Design scenarios to illustrate both successful and unsuccessful return codes of each command to manager. (8:36)

(f) Graceful termination of your application, i.e., de-registration of player processes. Of course, the server process needs to be terminated explicitly. (12:20)