Team Name: concurrent_butt_kicking
Members: Johanna Ross, Jacob Bennert, Will Rusk

The Project

Our Project is an API that allows users to create concurrent card games that can be played over the internet. Our goals were to make the API general enough that it can be used to create a wide variety of card games while still making it powerful enough that the user doesn't have to do too much of the coding. Our API abstracts away all of the networking and synchronization needed to run the game clients and server.

The only thing that the users of the API have to write is the logic for the game itself. They may use provided library classes including gamestate, player, hand, etc. to make this easier. Once the business logic of the game is declared, the users declare the overall flow of the game and then let the API do the rest.

Overall our API greatly simplifies the process of creating these games. The user need only define the actual logic of the game itself, allowing for complex games to be defined in a surprisingly easy, declarative way. Our API is able to use this declaration of the game flow to run both the game server and clients, cutting the code required to create the game significantly.

What's the minimum/maximum deliverable?

Minimum deliverable:

- Minimally applicable API
- At least one game created
    - 'Bullshit'
- Limited flexibility in game structure
- Basic text-based/Ascii graphics

Maximum deliverable:

- Maximally generalizable API
- Multiple test cases
    - 'Bullshit'
    - Cards Against Humanity
    - Texas Hold 'em
- Minimal user effort
- Graphics + chat modules
    - User defined + built-in options

What did we achieve?

We were able to achieve the minimum deliverable. We made an API that abstracts away message passing and concurrency and allows a user to make a card game by providing functions to handle asynchronous and synchronous turns as well as checking win condition, etc. We are happy with what we achieved, however since we ended up changing ideas last

minute, we weren't able to get as much done as we had hoped, and we're sure that with more time, we could have made the API even better.

One of the things that we wished we had been able to clean up is the way a player interacts with their client program. At the current moment, there are some instances where the screen the player sees can get a little messy, especially (in the case of BS) with displaying their current hand or calling BS. These shortcomings are somewhat split between the API and the implementation of the BS game itself. Both could stand to be improved, but a certain amount of the work is up to the users of the API to complete.

Classes:

- CardGame
    - Allows the user to create the game and specify the control flow in the form of synchronous and asynchronous events, win checks, and turn advancements.
    - Takes in user-defined closures that specify what to do during each event.
    - Holds all of the information for how to run the backend of the client and the server, hidden from the user.
    - When run() is called, determines whether the user is the server or the client, then runs the game with the specified control flow.

The following classes are designed to be used by the user within the functions they're giving to CardGame. They are supposed to help take some of the load off of the users by providing helpful representations for common card game items.

- GameState:
    - Represents the current state of the game, and contains methods and instance variables for getting and setting different possible fields of the gamestate, such as decks, list of players, who's turn it is, etc.
    - Is initially defined by the user to hold whatever information suits them, and, because the gamestate is often a parameter to the user-defined closures given to CardGame, is mainly managed by the user. This way our API doesn't need to concern itself with how the user has chosen to represent the gamestate.
    - Contains get_json method so it can be sent back and forth across the network.
- Player
    - Contains name, points, and a hand for a given player.
- Hand
    - Represented as a list of cards, and contains helpful methods for the user such as pick_random_card(), contains(), and print_hand(), to name a few.
- Deck
    - Represented as a list of cards, and contains helpful methods for the user such as shuffle(), divide_deck(), and draw_card(), to name a few.
- PlayingCard
    - Contains number and suit, as well helpful methods for the user such as is_equal(card), rank_to_string(rank), and print_card(), to name a few.

- WordCard
    - Contains a single message.
    - Allows user to create and print Cards Against Humanity (Crabs Adjust Humidity) style cards.

Backend Design:

The way that we represent the control flow of the game within the CardGame class is by using an event queue. The CardGame class has functions the user can use to add different kinds of events to the event queue, as described in the description of the CardGame class above. When they add a specific type of event, they also have to provide a function (or functions, in the case of a sync event) to be run during that event. This function is put onto the event queue, along with an event-type signifier, and a unique ID. The server then iterates through this event queue, and runs the functions as they appear. Of course, depending on what the event-type is, a few different things might need to happen.

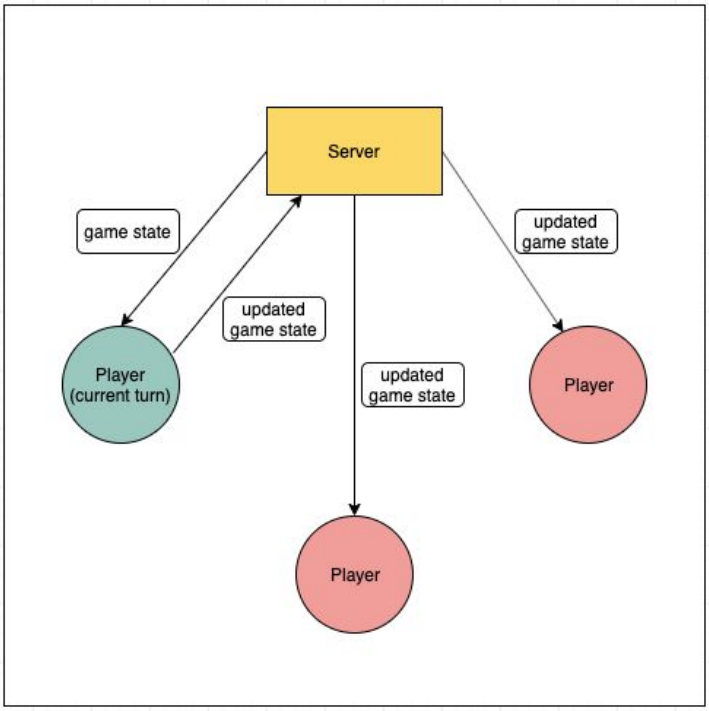
If we have an asynchronous event, then we look up who the current player is, and we send them a message telling them to run the function associated with that event. Because everyone is running the same code, we don't need to send the function over the network. The client already has it! It's an invariant that the function defined for an async event must take in just the gamestate, and return a tuple of the new gamestate, and any message that should be sent out to the other players. So we wait for the client to send us that message back, we update the gamestate to be the new one we just received, and we send the message out to all the players (except the current player).

If we have a synchronous event, then we create a thread for every player (except the current player), and we send a message to those players telling the to run the function associated with the event. As we receive their responses, we put them onto a message queue, and then consume messages using a user-defined function. This user-defined function takes in the gamestate and a continuation (among other things), and returns a gamestate and a continuation, which are passed back into the function along with the next message. At the end of that process, the gamestate is updated with the most recent gamestate.

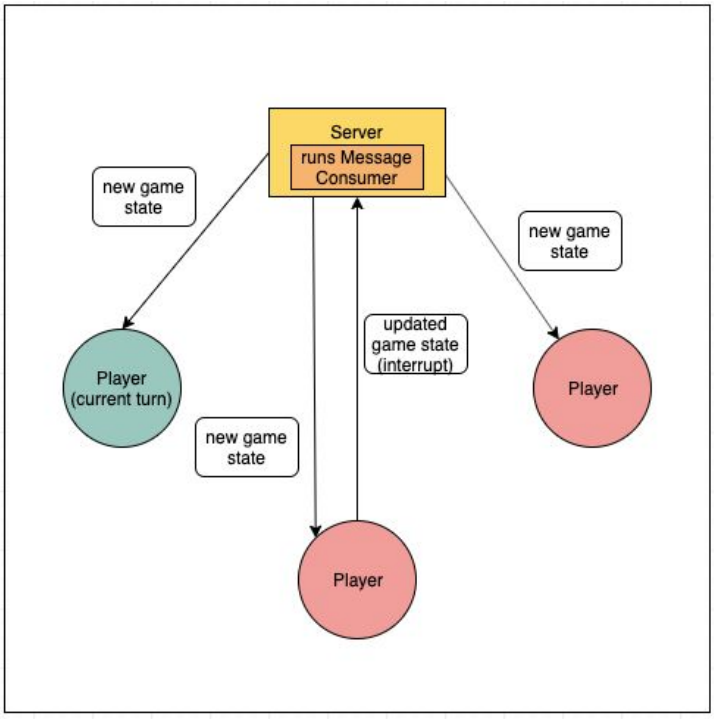
If we have a win check event, then we run the win check function the user provided, giving it the current gamestate. If it returns true, we stop the loop and announce the winner. Otherwise, we keep going.

Finally, if we have a next player event, then we call the next player function the user provided, giving it the current gamestate.

This is representative of what might be happening during an asynchronous event.



This is representative of what might be happening during a synchronous event.



Design reflection:

Best decision made?

An event queue was a great decision for this project it was easy to implement and made the API far more general. Before we implemented the event queue the user could only make games that had logic from a specific set that we had pre-defined in the system.

What would we do differently?

Ideally we would change the format of the GameState class to allow users to add any field that they wanted (as a key value pair) making it truly applicable to all games however we ran into problems with converting these user defined fields into jsons to pass between the server and client and didn't have time to make it work.

Distribution of Work:

We originally tried to completely split up the work completely and have each of us working on a different aspect of the game (one person working on server code, one person working on client code, one person working on the API test case "user" code). This became difficult since all of the parts are so interrelated and the client/server/user code all depend on each other. Once we realized this we did most of our coding together or split up who would write functions but define the function contract beforehand together.

Bug Report:

The most difficult bug we encountered was one where the server would think someone had called bullshit even if no one had. We were very confused because we looked through how we were initializing the variable that controlled that, and there was no problem there. We looked at how the player input the variable on their end and there was no problem there. We looked at how we were passing this variable through the message queue as the continuation (this variable was a Boolean), and there was no problem there. We were stumped. What we ended up doing was printing out the value of the variable at every time it was created or passed somewhere. While we were looking at these print statements, we noticed something strange. Sometimes the variable would print as 'True' and sometimes it would print as 'true'. We realized that what was happening was when we passed the variable over the network, we were converting it to a string, and then not converting it when we received it. Because the string 'false' evaluates to the Boolean True, the check was always passing. It took us 30 minutes to find this bug. In retrospect, it probably would have been simpler right from the beginning to just print out the values as a sanity check, because seeing a print statement that was something we didn't expect would have pointed us right to the pertinent spot in the code.

Files:

- GenCardGame.py
    - Contains all the classes that make up the concurrent card game API (described above)
- Bullshit.py
    - File in which the game 'Bullshit' is defined, using our API.
    - This file will be run with different command line arguments depending on if you want to set up a server or join an existing game.

Run instructions:

Server: python3 bullshit.py -h
Client: python3 bullshit.py [username] -c [hostname]

- Start server first then game will run after all players have joined
- **Important:** Make sure you have enough people joining to reach the minimum number of players described in bullshit.py. The server will only give the prompt to start the game if there are at least the minimum number of players in the lobby. Alternatively, you could decrease the min number of players needed.
- We know the game will work on tufts secure ssh-d into a vm, taking the vm name as the hostname
    - Ex. python3 bullshit.py johanna -c vm-hw01 if server is running on vm-hw01

Packages:

sys, socket, json, select, threading
(these already get imported in GenCardGame.py so you shouldn't need to worry about it)

API Documentation

This module defines the following classes and functions

CardGame Object:

The CardGame defines and holds the logic for running a card game. The logic of the game is specified by calling add_xxx_event functions, which insert the passed closure into the game's event queue. The functions in the event queue will be executed in the same order they are added in. Once all events are added to the game the CardGame::run() function may be called to start the game.

Depending on the command line arguments of the program, either the server or client will be run. To run the server argv[1] must be '-h'. To run the client argv[1] should hold the username to join the game with, argv[2] should contain the '-c' flag, and argv[3] must contain the hostname of a machine running the server.

There is no need to write specific client/server code, the definition of the game is enough for GenCardGame to run both the server and client.

class CardGame():

The constructor for a CardGame.

method add_async_event(closure):

Adds an asynchronous event to the event queue. When this event reaches the front of the queue the server will signal the client who's turn it is to execute the passed closure function. The closure will be passed the current gamestate as its first argument and must return the updated gamestate as of the end of the event.

method add_sync_event(client_closure, server_closure, initial_accumulator):

Adds a synchronous event to the event queue. When this event reaches the front of the queue the server will signal all clients to execute the passed client_closure function. client_closure will be passed the current gamestate as its first argument and must return a message to be sent back to the server. server_closure will be passed the game object, the current state of the game, the accumulator (at first has the value initial_accumulator), and a message on the message queue. server_closure will be folded across the message queue, passing along an updated state and accumulator.

method add_next_turn(closure):

Adds a next turn event to the event queue. When this event reaches the front of the queue the server will execute the passed closure function. The closure will be passed the current gamestate as its first argument and must return the updated gamestate as of the end of the event.

This function is meant to allow the user to change the player whose turn it is but may also be used to modify the state without signaling the clients for any other purpose.

method add_wincheck_event(closure):

Adds a wincheck event to the event queue. When this event reaches the front of the queue the server will execute the passed closure function. The closure will be passed the current gamestate as its first argument and must a player object, the player who has won the game, or None if no player has won yet.

method broadcast(message):

Broadcasts a message to all clients. Calling this method causes the message to print on all clients' screens.

Broadcast may only be executed in closures running on the server and has undefined behavior if called before run() is called.

method set_init_state(closure):

Sets a method to initialize the gamestate. closure is passed the gamestate and must return the gamestate with the fields within initialized as the user pleases. closure will be called by the server before the first action of the event queue is executed.

method set_num_players(num_players=(2,2)):

Sets the minimum and maximum number of players. num_players must be a tuple whose first element is the minimum number of players and second element is the maximum number of players (>= the minimum).

method run():

Runs the client/server. Run may not be called before at least one wincheck event has been added to the event queue, the initial state closure has been set, and the number of players has been set

Player Object:

The Player class holds the name, hand, and points for a specific player.

var **name:** a string representing the name of the player.

var **hand:** a Hand object representing the player's hand.

var **points:** an integer representing the number of points this player currently has.

class **Player(name):**

The constructor for the Player class.

method add_hand(cards):

Cards is a list of Card objects. Sets the hand for the player.

Hand Object:

The Hand class represents a hand of cards. Cards in the hand must export a print_card() function which should print an ascii representation of the card to stdout and a get_face() function which returns the width, height, and string representation of an ascii image of the card.

class Hand(cards):

Hand constructor takes a list of cards that will make up the hand initially.

method add_card(card):

Adds card to the hand of cards.

method pick_card(card):

Removes card from the hand and returns it if it was present, otherwise returns None if card was not in the hand.

method contains(card):

Returns a boolean True if card is in the hand and False otherwise.

method pick_random_card():

Returns a random card in the hand.

method size_hand():

Returns the number of cards in the hand.

method print_hand_vertical():

Prints all of the cards in the hand vertically by calling each card's print_card() function.

method print_hand():

Prints all of the cards in the hand horizontally by calling each card's get_face() function.

method to_string():

Renders all of the cards in the hand using get_face() for each card then returns a string representation of the entire hand. Useful in conjunction with broadcast to send the content of player's hands to clients.

Deck Object:

The Deck class represents a deck of cards. Cards in the deck must export a get_face() function which returns the width, height, and string representation of an ascii image of the card.

class Deck(cards):

Deck constructor takes a list of cards that will make up the deck initially.

method add_card(card):

Takes a Card object and adds it to the deck.

method draw_card():

Returns a Card object from the deck.

method shuffle():

Shuffles the deck.

method contains(card):

Takes a Card object and returns true if that card is in the deck.

method add_deck(cards):

Takes a list of Card objects and adds them to the deck.

method divide_deck(num_players):

Takes in a number of players, and returns a list of lists of Card objects, where each list contains a number of cards equal to (or as close as can be) the number of cards that were in the deck divided by num_players.

method print_deck():

Pretty prints the deck to the terminal.

method to_string():

Returns a string representation of the deck of cards by calling the get_face() method of each card in the deck.

GameState Object:

The GameState class holds the fields necessary to encode the current state of the game.

var players:

a list of all players currently in the game. Each player is represented by a player object

var cards_on_table:

a deck object containing all of the cards currently lying face-up on the table

var primary_deck:

a deck object of cards to be used in the game

var **played_primary:**
a list of all cards played from primary_deck in the game so far

var **secondary_deck:**
another deck object of cards to be used in the game

var **played_secondary:**
a list of all cards played from secondary_deck in the game so far

var **judge:**
a player object who should serve as the judge this turn

var **last_move:**
a field to contain information about the last move of the game

var **curr_rank:**
a field to contain information about the current turn of the game

var **curr_player:**
a player who should currently take their turn

class **GameState(players):**
The GameState constructor takes a list of players objects who are in the game. This will be called automatically by the API but is exposed for outside use as well.