CMPT 434
Computer Networks
Group Project

Kale Yuzik
kay851
Jason Goertzen
jag190
Ben Haubrich
bjh885

April 6th, 2020

Checklist (Temporary)

Introduction

This project for CMPT 434 (Computer Networks) will focus on methods that can be used to implement anti-cheat measures in multiplayer games. The game being developed is a command-line interface card game similar to War, but with extra rules and moves. The game will connect 2 players over a network into a session where they play the card game until one player wins or quits. The game will feature a command-line interface, and no GUI will be developed.

Several ways in which a malicious actor may attempt to gain unfair advantage in the game will be examined and defences to these weaknesses will be proposed. Players will be prevented from making a move that doesn't align with the rules of the game such as moving a card more than once per turn, or moving more than one square (the game is based off a grid) at a time. Players will not be able to spoof any game states such as claiming to own cards they do not have or claiming to have cards on a square. The cards that are drawn in the game must be truly random, and not influenced by the players or easily predicted.

The finished project should eliminate avenues of cheating for the purpose of winning a game. The game will allow at least 2 players to play over a network and will be command-line based (i.e no GUI will be developed).

Objectives

- Should the project use the Ethereum blockchain via the Geth Ethereum client, the solution is stronger
- Use smart contracts to validate the legitimacy of any given decision a player makes to protect against cheating, in a distributed, public manner
- Use Solidity to write the Ethereum smart contracts
- The more the players can visualize the board, the better
- The more defences against cheating, the better
- Implementing game connectivity to be played over wide area networks
- Enabling spectators to watch a game being played
- Allow players to gamble and gain rewards based on the outcome of a game
- The more random the drawing of cards is, the better

Functions

The finished project will:

- Allow both players to make a turn

- Allow two players to connect and play a game
- Allow players to see each other's turns
- Have some concept of a deck of cards
- Ensure a player's hidden cards are only visible to others when the rules of the game dictate they should be while ensuring the cards were selected randomly
- Allow players to see the turn that the opponent made
- Disallow playing the game outside of the set rules (cheating)
- Determine a winner

Constraints

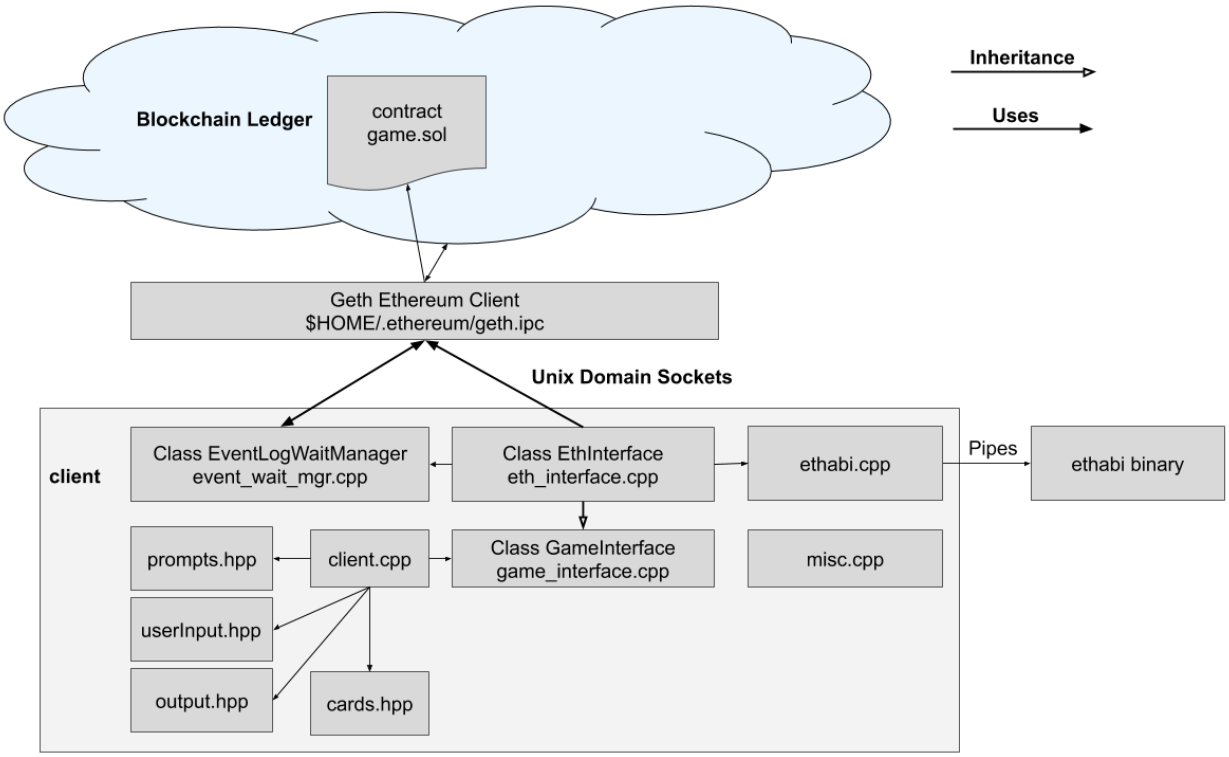
The project must:

- Involve computer networking

Technology Used

This project uses numerous dependencies. Please see the readme for installation and configuration instructions.
A private Ethereum blockchain network is used through the Geth Ethereum client. Ethereum's smart contracts are used to store the game state and verify the game rules are enforced. The contract is implemented in Solidity.

Design



Contract (Distributed Code)

Files

- contracts/game.sol

This is written in Solidity, a language that compiles into Ethereum Virtual Machine bytecode using the Solidity compiler (solc). This is the distributed code that runs on mining nodes on the blockchain. This code performs all business logic for the game and ensures there is consensus about the legality of a move in the game.
The contract stores the game state and all data in it is publicly visible. This creates some challenges:

1. Drawing random cards
2. Preventing either player from being able to see the other players hand (hidden cards), while preventing a player from being able to lie/falsify the cards they are playing

Drawing Random Cards

Random numbers can not be generated in Solidity since blockchain miners simultaneously execute the contract code, for which the results must be agreed upon. If each miner were to generate a random number the results would not be consistent and agreement would not be possible. To overcome this and generate random numbers for use in the contract, both players submit 52 random 8-bit unsigned integers (for a deck of 52 cards) that are XOR'ed together. This ensures that neither player can explicitly control the result. This forms what we term the "deck seed".

Keeping Player's Hands Hidden

Both player's can know the "deck seed", since it is on the blockchain (public ledger). These random numbers cannot be used directly to determine what cards will be drawn. To form each player's private hands, numbers are drawn from the "deck seed". These numbers, along with a nonce of the time the game started, are then signed with the players encryption keys on the blockchain, which results in a 65 byte signature. These 65 bytes are XOR'ed together to reduce it to 1 byte. A modulo 52 is performed on the number to obtain the card which it will represent. This ensures the other player is unable to know what cards the other player holds in their hand. When a player plays a specific card, making it visible to the other player, we must ensure that the player cannot falsify/lie about what the card was. To do this, the player must submit the seed value of the card that they are playing, the claimed card ID, and their signature of the seed value. The signature can then be verified in the contract.. If the result of this computation is the signers public Ethereum address, the signature is valid and the signature is then reduced to 1 byte using XOR and a modulo 52 operation. If this number equals the claimed card ID, the card is valid and has not been falsified.

Blockchain Framework

Files

- eth_interface.cpp
- event_wait_mgr.cpp
- ethabi.cpp

Class EthInterface (eth_interface.cpp)

EthInterface provides methods to request services of the Geth Ethereum client, including:

- Compile and upload a new contract to the blockchain
- Perform a call to an accessor function of a contract (does not modify contracts internal state)
- Perform a call to a mutator function of a contract (known as a transaction), which may modify the internal state of the contract
- Get the Ethereum address(es) of the Geth client

- Sign a block of data with the encryption keys of the local Ethereum account
- Various functions to format responses from Geth to required formats/data types

Requests are sent to Geth via a Unix Domain socket with Geth's JSON API.

Class EventLogWaitManager (event_wait_mgr.cpp)

EventLogWaitManager consists of a server thread which is responsible for primary communications with the Geth Ethereum client via a Unix Domain Socket. When certain points in the contract are reached, the contract can "emit" events, which EventLogWaitManager subscribes to. Communication with Geth is done with Geth's JSON API. When the server thread starts and the connection is opened, numerous event signatures are subscribed to.
These events are used to indicate to a client that their requests of the contract have completed. This is required, since mutator functions calls to a contract are asynchronous and do not return immediately. Events are also used to notify a client when the other player has completed their turn.
When the GameInterface class needs to wait for one of these logs, it calls EventLogWaitManager::getEventLog(). This function is blocking and the server will unblock the calling function when the required log is available (if it was not already available when the function call was first made). This design of the server allows for far more robust function than what is required for this game. Many concurrent threads could make requests of the server in a fast and efficient manner.

ethabi.cpp

This file implements functions to request encoding/decoding services from the `ethabi` binary. This is required to provide and receive messages from Geth. Ethabi is called using the popen() family of system calls to open a pipe to the binary.

Game Client

Files

- game_interface.cpp
- Client.cpp

Class GameInterface (game_interface.cpp)

GameInterface inherits from EthInterface and provides methods to access each of the functions implemented in the contract. No game logic is implemented in this class.

client.cpp

The game client was designed to allow the user to make any move they want, whether it is legal or not. The goal is to have all rule enforcement occur in the contract as described above. There are certain cases where the client performs some computation before sending a move to the contract (such as finding adjacent squares), but the computation performed by the client is still

verified by the contract. The reason more computation is performed in the contract is because the contract must be as light/small as possible. The client may look like it is restricting plays as it highlights valid moves the player can make, however, these are merely visual guides and the client doesn't check to see if the user's input is legal.

Challenges and Oddities

Geth

We experienced an issue where geth would return corrupted data. After trying several versions, we settled on version 1.9.9-stable. Issues with corrupted board state still arise every so often, mostly on Mac OS X, however it is a much rarer occurrence with this version. In an attempt to mitigate the issue, we developed a function that will try up to 25 times to get a non corrupted board state. Unfortunately, sometimes the returned state appears to not be corrupted when it actually is, so we sanity check the state. This doesn't catch all incorrect cases, but also reduces the number of corrupted boards that were displayed. A Git issue was created on the Geth repository as a result of this: https://github.com/ethereum/go-ethereum/issues/20890

Ncurses

Due to Apple shipping their own version of ncurses, we were unable to get the ncurses version of the main menu working for the Mac OS X build. There appears to be a collision or configuration issue between the homebrew version of ncurses and the version Apple ships with OS X.

Outcome

We were able to successfully develop our turn-based card game on an Ethereum private network, such that all moves are verified and added to the ledger. As our client doesn't enforce the rules of the game, we were able to test and verify that illegal turns were in fact rejected by the contract, and the player is then prompted to make another, hopefully, legal move. If the new move isn't legal, the contract will still reject it, and so on and so on. One negative of the approach we have taken is we rely on Geth to interface with the Ethereum Virtual Machine, and as mentioned above, means that if there is a bug in Geth we can't fix it. Ultimately we believe the project to be a success and has potential to be built upon in a future date

Contributions

Ben

- Text user interface
- Documentation
- Testing and debugging

Jason

- Game design
- Client design/development
- Testing and debugging
- Documentation
- Mac build

Kale

- Contract Development
- Ethereum interface development
- Game interface development
- Testing and debugging
- Documentation
- Linux and Ethereum Build