Distributed Card Game

Group 27: Sebastian Sergelius, Henrik Nygren, Daniel Koch

Contents

Introduction **3**

Project Goal **3**

Core functionality **3**
- Creating and joining a game . . . 3
- Leaving a game . . . 4
- Playing the game . . . 4

Design Principles **4**
- Architecture . . . 4
- Communication . . . 4
- Process . . . 6

Functionalities **7**
- Naming and node discovery . . . 7
- Consistency and synchronization . . . 7
- Fault tolerance . . . 7
- Consensus . . . 8

System scalability **8**

Performance **8**

Lessons learned **8**

Appendix **8**

Introduction

In this project, we will create a distributed card game, where the goal is to get dealt the highest card. The participants will connect to each in peer-to-peer fashion, and the system will try to ensure that the card deck in the game will be shuffled fairly.

Project Goal

Our project will be a very simple card game where we have one card deck containing 52 cards. Cards are numbered from 1 to 52. First step is to shuffle the card deck with two nodes, one being leader and other being a random player in the game, in a manner where cheating is hard or impossible. After the deck has been shuffled, we deal each participant a card in ascending order based on their player number. Once each participant has been dealt their card, they can decrypt the whole deck and verify the game.

To ensure cheating doesn't happen while shuffling the cards, we will implement the Mental poker card shuffling algorithm: https://en.wikipedia.org/wiki/Mental_poker#Shuffling_cards_using_commutative_encryption.

Each participant will run the same software, which will be a Python program running on top of Docker. To ensure that participant can connect, we need to have a leader node, which admits new nodes to the game. Connections will be established using the leader nodes' IP. If a player connects to a non-leader node, this node will provide the correct IP to connect to.

Core functionality

Our environment is built on docker containers, each container running the same code. Commands from nodes will be taken from STDIN in terminal. Before each command we check that the leader node is alive and healthy.

The following commands are available:

- `join <ip>` - joins an ongoing game or creates a game if first participant
- `leave` - leaves the game
- `list` - list the state the node knows off
- `start_game` - the leader can start a game
- `debug` - to debug game state

Application: Big online card game tournaments

Creating and joining a game

Each participant will be running the same environment. First participant node will join another (leader) node, which will define the leader node. The first joining participant will provide the leader node its IP-address it used to join with. Once another participant node joins, he can join any of the participants in the game. If new node tries to join non-leader node, they will automatically be redirected to the leader node.

Leaving a game

A participant can leave the game. Once the participants leaves the game, the leader node will broadcast the updated player list to all nodes in game.

Playing the game

The game can start once players have joined and start_game command is issued at leader node. Leader node broadcasts that game is starting. Leader picks another node who he uses to shuffle the deck with. Once deck is shuffled, each player is dealt the card in order of player name.

Design Principles

Architecture

Each node is running the same environemnt, which is based on Python. These nodes run in a docker container where each node has two threads, which are the following:

- One thread for reading commands from a user from the terminal (UI thread)
- Other thread for running the flask http server for receiving request from other nodes (Web server thread)

The two treads communicate with each other using shared memory. Because we are running a distributed system that requires sometimes voting and waiting, we additionally spawn threads in these cases.

Communication

Either actions in the UI thread or the web server thread can trigger communication to the other nodes. Communication is done by directly starting an HTTP request to the other node when needed. Messages are JSON messages that are posted to the other node's port 6376. The most common HTTP requests and responses have Python types that act as documenting the messages in the code.

Endpoint	Data	Response	Description
POST /election	origin ip	taking_over: bool	Used by our Reverse Bully algorithm to send out election message to nodes with smaller number, as a response you will receive if there are nodes taking over, i.e. nodes with smaller number
POST /new-leader	origin ip	message: str	Endpoint used to set new leader for node after election is done.
GET /health	NaN	message: str	Used by nodes to check the leader health before each command, if no response in 5 seconds, we start election.

Endpoint	Data	Response	Description
POST /join	origin ip	message: str nodes: Dict[str, Player] your_player_number: int leader_node_number: int	Joining node sends join command in the command line. At this endpoint we handle giving the participants a name and maps the name with his IP and player_number. We also give the joining participant the distributed state and aswell broadcast to everyone that a new node has joined.
POST /leave	origin ip	message: str	Verifies that a participant is in game and deletes him from the player list, broadcasts to the current in-game players state.
POST /new-node-list	NewNodeListMessage: nodes: Dict[int, Player] next_player_number: int	message: str	Handles setting the game state for node if someone joins or leave.
GET /get-nodes	NaN	message: str nodes: state.NODES	This is used for Fault tolerance, if a node has crashed and tries to join with same IP, we can return him the current game and we don't need to reset his player number.
POST /game-starting	NaN	message: str	Sets game phase to ongoing, which means other players can't join the game during this phase.
POST /plz-help-with-encrypting	deck: str	deck: str	The leader sends to one randomly chosen node his encrypted deck as JSON, the helper node in this endpoint creates the deck from this data and encrypts it once again, shuffles it and broadcast the double encyprted deck to each player in game.
POST /double-encrypted-deck	deck: str	message: str	Each node receives the double encrypted deck here and saves it to his state.
POST /deal-result	who_get_what_cards: Dict[int, str] helper_player_number: int	message: str	Leader performs the deal-result to each node, once a node receives this information, he sends to the helper node that he has received the info that cards are dealt. Saves the dealt cards in state.
POST /helper-key	key: str	message: str	The helper node distributes its encryption key to node, node saves this to state for later game verification

Endpoint	Data	Response	Description
POST /leader-key	key: str	message: str	The Leader distributes its encryption key to node, node saves this to state for later game verification
POST /i-got-the-dealt-cards	origin ip	message: str	This endpoint is used by the helper node to determine that majority (50% or more) has received the dealt cards and then he can start distributing the private key to each player.
POST /winner	winner: int	message: str	Triggered when leader determines the winner. Each player starts verifying game and participating in fairness voting, this is done in its own thread.
POST /game-winner-verification-result	agree: bool	message: str	Each player has check from the double encrypted deck and the dealt cards by the leader that they match and the broadcasted winner is the one that leader claims. Every node sends to each and every node in game the verification boolean. If half of more agree to this fairness voting, the winner can be announced.

Process

- Leader maps the name to the player IP and player number.
  - Broadcasted to all nodes
- Game starts by each player joining leader node.
- Once players are have joined the leader can start the game (with the start_game command). Game follows pattern:
  - Leader broadcasts start_game, each players game phase should be ongoing.
  - Leader picks randomly another helper node to shuffle and encrypt the game deck with
    - Leader encrypts deck and shuffles
    - Sends encrypted deck to helper node
    - Helper node encrypts the deck again, shuffles and returns to leader
      - Helper node also sends a copy of the double encrypted deck to each player for later game verification
  - Once the deck is encrypte d by leader and fellow player, all the nodes know who got dealt what cards, because the cards are dealt from the shuffled double encrypted deck in order (by player name/number).
  - Each player broadcasts that they have received the double encrypted deck
  - Once most players have received the deck, the helper node broadcasts its private key
  - Once helper private key is received by leader, leader broadcasts his private key

- "Imaginally" each player flips their card and the leader checks who got the highest card
- Leader broadcasts to everyone the winner player name
- Each player decrypts the encrypted deck, and checks if the game result is correct
- Voting is started for winner
  * If 50% or more approve the winner, he wins.

Functionalities

Naming and node discovery

Each participant is named by a number that is picked by the joining order to the game.

Once a new participant joins to the game, it is given a list of all other nodes in the game. The leader also broadcasts the new list of nodes to all other nodes whenever a new node joins.

Consistency and synchronization

We have implemented a strong consistency, we try to post all updates to all nodes and keep the data consistent. Consistency is achieved by broadcasting all the changed information to everyone such as the joining and leaving nodes and the encrypted game decks. If a node has crashed, this node will not be able to send a leave request.

If the leader crashes and some other node writes a command, such as `list`, a leader election should start, once the leader election is done, once we have a new leader, it is broadcasted to everyone.

Fault tolerance

During programming, we noticed that some failures are possible, such as a participant dropping out of the game or not picking up a card. If a participant drops out of the game, he will be part of the game and able to join back later, if he has the same IP-address. Timeout is used by the leader once the game has started and the deck is shuffled. If a participant does not respond, he will not be part of the game.

Every time a node sends a command to the leader, it checks for the leader nodes health, if the leader doesn't answer within 5 seconds, we will start the Reverse Bully Algorithm (prefers small numbers).

Dropping of non-leader nodes does not prevent the game from continuing because in order to determine the outcome of the game, we need only the majority of the nodes to agree on the result

Some minor things we didn't have time to finish is that if enough non-leaders disconnect the game will eventually break because non-leader health is not checked. The breakpoint when n/2 non-leader nodes fail at the same time, n is the total number of nodes, including leader. This could be fixed by doing health checks on non-leaders too.

Consensus

Consensus is achieved when each participant sends their (encrypted) picked card up to every other participant, these participants then decrypt all cards with keys received from two shuffling partners. When leader broadcasts the winner, other nodes will get the result and check if they agree. They will broadcast OK or Not OK to everyone and wait for results from other nodes.

If majority opionion on the fairness of the game cannot be reached, we will just conclude that the game was not fair, and the round won't have a winner.

System scalability

The chosen approach should scale to some degree. It should be possible to add many participants to the game. One major limitation of scalability could be the fact that is that everyone is connecting to every other participant, meaning that at some point, each node has too many connections. However, the protocol used here is so lightweight that the number of participants would need to grow to be very large before anyone would be overwhelmed with messages. The scalability of the approach could be improved by utilizing a structured peer-to-peer network.

Performance

The performance of the game is asynchronous. Performance wise there is no issue when shuffling the game deck, as currently we have 52 cards. making the deck bigger does not affect the performance too much: only the leader and one other node will encrypt all cards once. Only problem is that the whole encrypted deck will be broadcasted to every node, so if there are many nodes and a very big deck this is a lot of data sent.

Also, performance wise broadcasting the whole NODES state to every player when there are many players, could cause an issue. To improve this we could broadcast just the player node number and then each participant could just remove that one from their list.

Lessons learned

- Python has no meaningful type errors
- VSCode Live Share works for code sharing
  - not for sharing terminals
- Lot of messages in distributed systems

Appendix

- We didn't completely implement Mental Poker, but we encrypted the deck two times and decrypted it in the same manner.

- Score given is not implemented when the winner is announced, so sharing the score given state with each node didn't finish, although it is not far away as we have the fairness voting.