Universität Zürich[UZH]

Decentralized Application: BlackJack

Group 10 (Demeter)

Jiaming Tong, Ying Fan, Yuhan Lin, Zishan Wei, Qianhui Wang

Github: https://github.com/WZishan/DappBlackJack

Department of Informatics IfI University of Zurich UZH
Binzmühlestrasse 14, CH---8050 Zürich, Switzerland

Abstract: Decentralized applications (DApps) are digital applications or programs that exist and run on a blockchain or peer-to-peer (P2P) network of computers instead of a single computer. This paper presents BlackJack, which is a famous gambling game in the world, being applied as a decentralized application. It is accomplished by different blockchain technologies and services: truffle, Ganache and metaMask. This Dapp provides a fair environment for people to enjoy BlackJack.

1 Introduction

As we know, once DApps are decentralized, they are free from the control and interference of a single authority. Benefits of DApps include the safeguarding of user privacy, the lack of censorship, and the flexibility of development (*Decentralized Applications (DApps) Definition*, 2021).

Based on the definition of Dapps, we create a decentralized application of the BlackJack gambling game. There are some tools and environments we used for implementing the Dapp: node (v: 14.19.1), truffle (v: 5.5.11), port: 8575, Ganache (for testing smart contracts) and metaMask (for visual interaction).

In this project, we use Smart Contract to achieve random (pseudorandom) distribution and fair game rules which will be introduced in section 2.3.

1.1 Requirements

The requirements we meet in this project are the following:

1) The core functionality is implemented and executed entirely within Smart Contract (SC).
2) The prototype validates the proposed use case.
3) The user can interact with the DApp via a Graphical User Interface (GUI)
4) A self-contained report documenting the SC, operation and the source code.

1.2 Game Rules and Assumptions

In this game, we have a deck of cards for the player and the dealer (without 2 jokers). The player makes their bets, they will have some initial bets. And they will get 2 cards. The dealer will also get 2 cards, and the 2nd card is hidden from the player.

The objective of the game is to achieve a higher total point than the dealer (but no more than 21, anything over 21 is an automatic loss called a bust) - if a player beats the dealer in this way, he wins from the casino what he bet (player can also win if the dealer busts). After the first round of dealing, the player has the option to hit (receive one more card) or stay (no more cards). If hitting results in the Player busting (total going over 21), the player's bet is lost.

After all the players are done hitting/staying, the dealer flips over the hidden card. If the Dealer's total is less than 17, then they need to hit (receive a new card). This process repeats until the Dealer's hand either totals to 17 or more or busts (goes above 21). After the Dealer is done, the final results are decided - if the Dealer busts, then the player who did not bust earlier wins his bet. If the Dealer does not bust, then the Dealer's total score is compared to the Players'. If the Player's total score is greater than the Dealer's, the Player wins money (in the amount that was bet). If the Player's total score is less than the Dealer's, the Player loses money. No money is exchanged in the event of a tie. Also, we set a function for the player to buy the bet. They can charge some money into the system and buy the bet. The result will directly show on the screen, and their bet will change by the result.

2 Design and Implementation

2.1 Software Architecture

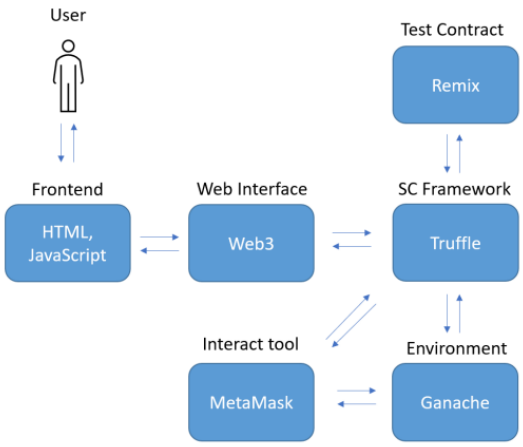

Figure 2.1.1: software architecture

Fig. 2.1.1 shows the architecture of this project. In our case, the system does not have any sort of database, as this task shall be carried out by the Blockchain itself.

The player opens the web interface which was written in HTML and Vue to play the game. The smart contract is deployed on truffle and transparent to people.

2.2 Graphical User Interface

The frontend code is mainly in HTML, CSS and JS, based on the node environment, using Vue and elementUI framework.

The frontend code is divided into four pages, the welcome page, the game details page, the victory page and the failure page.

Figure 2.2.1: welcome page

At the beginning is a simple welcome page (shown in Fig 2.2.1), consisting of text and two buttons, click the "start" button to start the game.

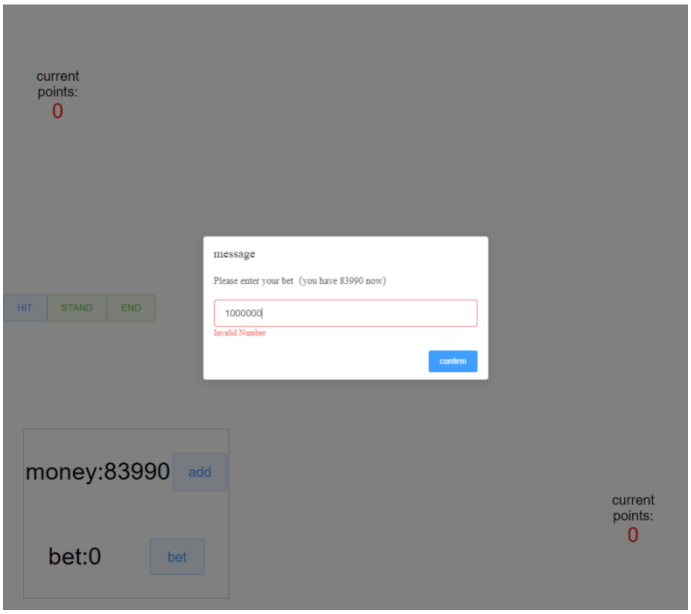

Figure 2.2.2: game details

Then jump to the game details page, first get the player's current balance by the *get_banlance* function, then the pop-up window asks the player to place a bet, the bet amount is less than or equal to the player's balance before the order is completed, otherwise, it will prompt the error 'Invalid Number'.

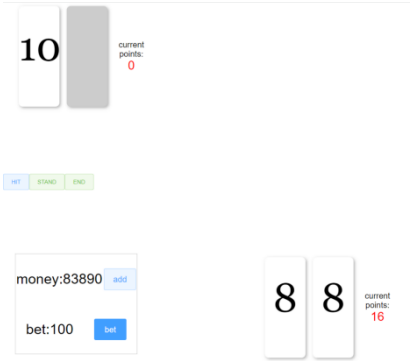

Figure 2.2.3: game details

If the input is correct, then the game will close the pop-up window and call the bet function. Then the *get_game* is called to start dealing, and the function determines the winner and loser. Every time calling to the *get_game*, "HIT" and "STAND" will appear, it is to ask the player whether to continue to take the cards. If the player chooses to continue, then it continues to call the *get_game* function to take the cards, and by returning the results If the player chooses to continue, the *get_game* function will be

called, and the system will determine the result of the game. If the player stops taking cards, the game is decided directly.

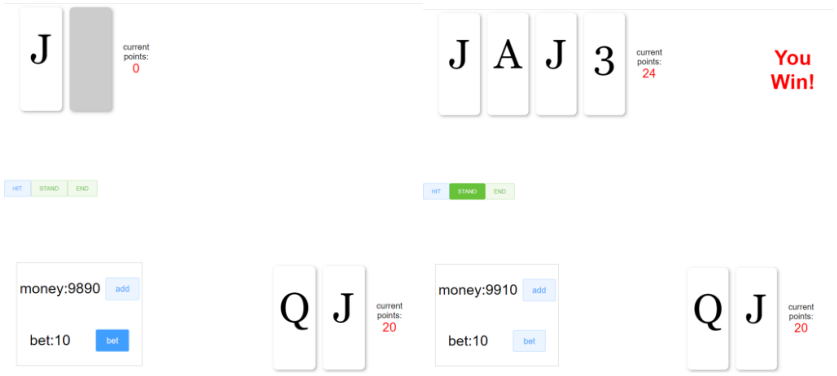

Figure 2.2.3: the player wins

If the player wins, the game will show this happy result to the player. The money will increase and the bet stays, which means the player gets double money.

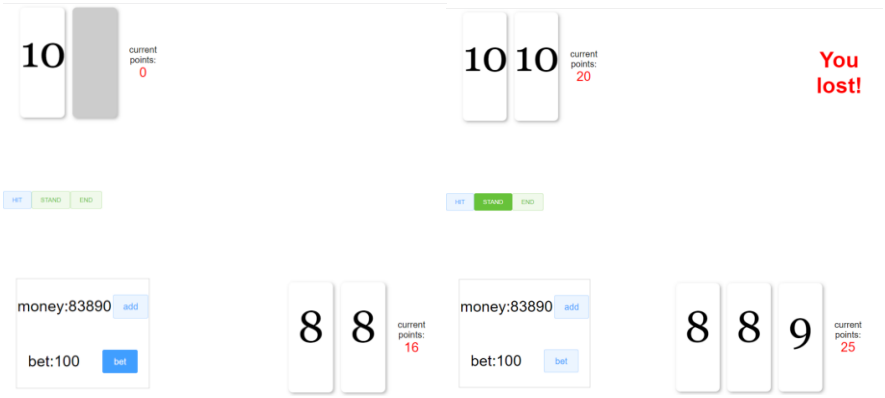

Figure 2.2.4: the player loses

If the player loses, the player can return to the initial page to start again.

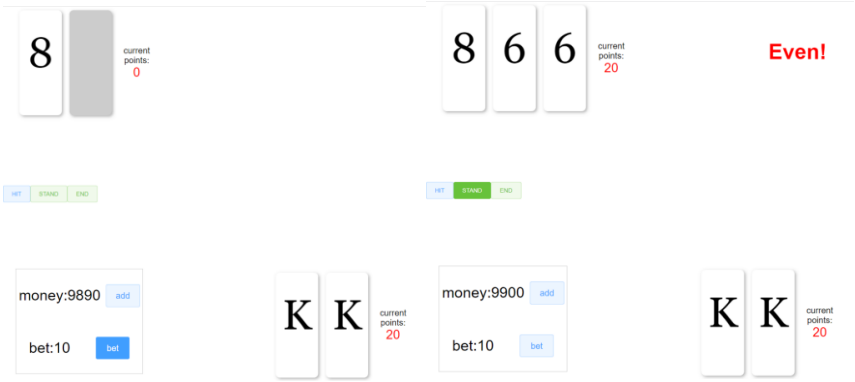

Figure 2.2.5: tie

If the scores are the same, the even result will be shown. And the bet will be returned.

2.3 Smart Contract

Smart contracts are simply programs stored on a blockchain that run when predetermined conditions are met. They typically are used to automate the execution of an agreement so that all participants can be immediately certain of the outcome, without any intermediary's involvement or time loss. They can also automate a workflow, triggering the next action when conditions are met (*What Are Smart Contracts on Blockchain? | IBM*, n.d.).

In this project, we use Solidity to implement SC (*Solidity — Solidity 0.8.14 Documentation*, n.d.). The following codes are applied as our SC in this project.

```
////////////////////////////////////////
//Errors
////////////////////////////////////////
string not_enough_account_balance = "Not enough account balance to charge!";
string not_enough_game_coins = "Not enough game coins! Please buy some coins.";
string amount_too_small = "The amount is too small!";
string account_not_exist = "Account does not exist!";
string please_bet = "please bet";
string bet_need_larger_than_zero = "Please bet larger than 0!";
string bust = "you have already busted!";
string already_bet = "you have already made the bet!";
string game_not_finish = "game is not finished yet!";
string game_finish = "game is already finished!";
```

Figure 2.3.0: Error

At the very beginning, we define the types of different errors. The system will respond correctly when the error occurs.

```
////////////////////////////////////////
//Struct
////////////////////////////////////////
struct Account{
    bool exist;
    uint balance;
    Game game;
}

struct Game{
    uint bet;
    uint dealer_hidden_card;
    uint[] dealer_showed_cards;
    uint[] player_cards;
    bool is_finish;
}
```

Figure 2.3.1: define the structure of the game and account

At first, we define the structure of the BlackJack game: the bet from the user, a hidden card for the dealer and cards for the player, whether the game is finished.

```
//Mapping
mapping (address => Account) accounts;

////////////////////////////////////////
//Function Public, Main Roadmap
////////////////////////////////////////
function get_balance() public returns(uint){ // login/create account
    if (!is_account_exist(msg.sender)) {
        accounts[msg.sender].exist = true;
        accounts[msg.sender].balance = initial_balance;
        accounts[msg.sender].game.is_finish = true;
    }
    return uint(accounts[msg.sender].balance);
}
```

Figure 2.3.2: mapping and function of *get_balance*

Before the player starts the game, we need to check the balance of his/her account, so this function returns to the account and checks whether they have enough money to bet.

```
function initialize_game() public {
    if (!is_account_exist(msg.sender)) {
        accounts[msg.sender].exist = true;
        accounts[msg.sender].balance = initial_balance;
        accounts[msg.sender].game.is_finish = true;
    }
    require(accounts[msg.sender].game.is_finish, game_not_finish);
    accounts[msg.sender].game.bet = 0; //also a way to judge if game has started, if bet = 0 means game is not started
    accounts[msg.sender].game.dealer_hidden_card = 99; //means no card
    delete accounts[msg.sender].game.dealer_showed_cards;
    delete accounts[msg.sender].game.player_cards;
    accounts[msg.sender].game.is_finish = false;
}

function get_game() public view returns(Game memory) { //get current information of the game
    require(is_account_exist(msg.sender), account_not_exist);
    return accounts[msg.sender].game;
}
```

Figure 2.3.3: initialize the game

Then we check the account. We want to send a requirement to the player's account and get the bet. If the account does not exist, it will be created.

```
function bet(uint value) public {//player makes his bet, initial cards delievered
    require(is_account_exist(msg.sender), account_not_exist);
    require(value >= 0, bet_need_larger_than_zero);
    require(accounts[msg.sender].balance >= value, not_enough_game_coins);
    require(accounts[msg.sender].game.bet == 0, already_bet);
    accounts[msg.sender].balance -= value;
    accounts[msg.sender].game.bet += value;

    //send initial cards
    uint id = rand(52, "1");
    accounts[msg.sender].game.dealer_hidden_card = id;
    new_card(true, "2"); //send new card for dealer
    new_card(false, "3");//send new card for player
    new_card(false, "4");
}
```

Figure 2.3.4: bet

The player can decide how many bets they want to bet, and the system will require from the player's account. If he/she does not have enough money, the game will give a hint about it.

We set 52 cards to be distributed. For the initial cards, 'true' means sending cards to the dealer, 'false' sending cards to the player.

```
function hit() public {
    require(is_account_exist(msg.sender), account_not_exist);
    require(accounts[msg.sender].game.bet!=0, please_bet);
    require(get_player_score()<=21, bust);
    require(!accounts[msg.sender].game.is_finish, game_finish);
    if (new_card_time) {
        new_card(false, salt1);//send new card for player
        new_card_time = false;
        // hit_times += 1;
    }
    else {
        new_card(false, salt2);//send new card for player
        new_card_time = true;
        // hit_times += 1;
    }

    if(get_player_score()>21){ // player bust
        accounts[msg.sender].game.is_finish = true;
        get_result();
    }
}
```

```
function stand() public {// player stands, dealer keeps taking cards until soft 17
    require(is_account_exist(msg.sender), account_not_exist);
    require(accounts[msg.sender].game.bet!=0, please_bet);
    require(get_player_score()<=21, bust);
    require(!accounts[msg.sender].game.is_finish, game_finish);

    while(get_dealer_score()<17){
        // uint dealer_hit = 0;
        if(new_card_time){
            new_card(true, salt1);//send new card for dealer
            new_card_time = false;
        }
        else {
            new_card(true, salt2);//send new card for dealer
            new_card_time = true;
        }
    }
    // hit_times = 0;
    accounts[msg.sender].game.is_finish = true;
    get_result();
}
```

Figure 2.3.5: status of the player

The Player has two statuses in the game: hit or stand. Whether hit or stand, the system needs to check the account, bet, score and card distribution first, then get the result. After the player decide to stand, the dealer will take cards according to the rule (sec. 1.2)

```
function get_result() public returns (uint){//1:player wins; 2:dealer wins; 3:tie
    require(is_account_exist(msg.sender), account_not_exist);
    require(accounts[msg.sender].game.bet!=0, please_bet);
    require(accounts[msg.sender].game.is_finish, game_not_finish);

    uint player = get_player_score();
    uint dealer = get_dealer_score();

    if(player>21){//player bust
        return 2;
    }
    else if(dealer>21){//dealer bust
        accounts[msg.sender].balance += 2*accounts[msg.sender].game.bet;
        return 1;
    }
    else if(player>dealer){
        accounts[msg.sender].balance += 2*accounts[msg.sender].game.bet;
        return 1;
    }
    else if(player<dealer){
        //accounts[msg.sender].balance += 2*accounts[msg.sender].game.bet;
        return 2;
    }
    else {
        accounts[msg.sender].balance += accounts[msg.sender].game.bet;
        return 3;
    }
}
```

Figure 2.3.6: get the result

To judge the winning or losing, we use the function *get_result*. This function can compare the score and decide the winner and loser, or it is a tie.

```
//////////////////////////////////////
//Function Public, Get Key Information
//////////////////////////////////////
function get_player_score() public view returns(uint){
    uint[] memory cards = accounts[msg.sender].game.player_cards;
    uint score = 0;
    uint num_ace = 0;

    for(uint i = 0; i < cards.length; i++) {
        score += id_to_val(cards[i]);
        if (id_to_val(cards[i])==11) num_ace++;
    }

    //if bust but have Ace, score -10
    while(score>21&&num_ace>0){
        score -= 10;
        num_ace --;
    }

    return score;
}
```

```
function get_dealer_score() public view returns(uint){
    uint[] memory cards;
    uint score = 0;
    uint num_ace = 0;

    cards = accounts[msg.sender].game.dealer_showed_cards;
    for(uint i = 0; i < cards.length; i++) {
        score += id_to_val(cards[i]);
        if (id_to_val(cards[i])==11) num_ace++;
    }
    uint hidden_card = accounts[msg.sender].game.dealer_hidden_card;
    score += id_to_val(hidden_card);
    if (id_to_val(hidden_card)==11) num_ace++;

    //if bust but have Ace, score -10
    while(score>21&&num_ace>0){
        score -= 10;
        num_ace --;
    }

    return score;
}
```

Figure 2.3.7: calculate the score

In BlackJack, there is a special rule: if the sum of values is smaller than 11, because the player's goal is trying to make the sum close to 21, So the Ace is 11 is the optimal solution. In this case, the system will make the value of Ace as 11; otherwise (bigger than 21), in order to prevent losing the game, Aces will be worth 1. Every other card is worth its face amount (face cards worth 10).

So in Fig. 2.3.7, we define a loop to judge the value of aces. In this case, the player does not need to calculate the value by themselves, our system can help them automatically.

```
function charge_game_coins() public payable{
    require(is_account_exist(msg.sender), account_not_exist);
    require(msg.value >= game_coin_exchange_unit, amount_too_small);
    accounts[msg.sender].balance += msg.value * exchange_rate / game_coin_exchange_unit;
}
```

Figure 2.3.8: charge

This function is used to charge the money into the game.

```
await this.contract.methods.charge_game_coins().send({from: this.account, value: Web3.utils.toWei(chargeStr),gas: gasCharge})
.catch(err => {console.error('error charge_game_coins ' + err);})
console.log("add_balance00",this.add_balance)
```

Figure 2.3.9

Collaborating with the frontend (shown in Fig. 2.3.9), players can transfer their money to the contract and charge game coins.

```
////////////////////////////////////////
//Function Private
////////////////////////////////////////
function is_account_exist(address _addr) private view returns(bool){
    return accounts[_addr].exist;
}

function rand(uint256 _length, bytes32 _salt) private view returns(uint256) {
    uint256 random = uint256(keccak256(abi.encodePacked(block.difficulty, block.timestamp, _salt)));
    return random%_length;
}

function new_card(bool is_dealer, bytes32 salt) private {
    uint id = rand(52, salt);
    if (is_dealer){
        accounts[msg.sender].game.dealer_showed_cards.push(id);
    }
    else{
        accounts[msg.sender].game.player_cards.push(id);
    }
}

function id_to_val(uint card_id) pure private returns (uint){
    require(card_id>=0&&card_id<=51, "invalid input!");
    uint card_val = card_id%13+1;
    uint val;
    if(card_val==1) val = 11;
    else if(card_val==11||card_val==12||card_val==13) val = 10;
    else val = card_val;
    return val;
}
```

Figure 2.3.10: private functions

For Fig. 2.3.10, these functions are private, which means they cannot be changed by other people (except the developer). In function *rand*, we realize random (actually pseudorandom, it will be introduced in detail in the following paper) card distribution. In function *id_to_val*, it defines the value of J, Q, and K cards.

3 Limitations and Improvement

For BlackJack Dapp, we set a *random* method to realize the random distribution of cards. In order to avoid taking cards at the time, the card will be the same (for example, if the player is distributed 2 cards at 8:00, the card might be the same), we also add a time gap to make the card different. However, because the difficulty and time stamp can be known and calculated, it is not a real random distribution, which means a smart contract cannot really generate randomness. But it is possible to be improved by a commit and reveal scheme, or a random oracle (Chainlink VRF or others).

Also, for now we can only accept one player to play the game. But in a real casino, there can be 2-6 people as players in one round. So maybe in the future, we can accept more players for this game.

4 Conclusion

This project has successfully completed the implementation of the Dapp and the requirements above, achieving openness and transparency in the rules of the blackjack gambling game and the division of cards, allowing players to immerse themselves in the game with more confidence. But please be noted that: play with your limit.

The prospect of Dapp and blockchain on gambling games is very promising, the transparency of blockchain allows online gaming to be provably fair because the entire transaction history is accessible on the ledger-the legitimacy of the game or transaction can be proven and users need not trust a third party to provide fair games(*Decentralization Of The Gambling Industry Through Blockchain - Gaming - United States*, 2021). We are looking forward to seeing more applications to be combined with blockchain in the future.

5 Reference

Decentralization Of The Gambling Industry Through Blockchain - Gaming - United States. (2021). https://www.mondaq.com/unitedstates/gaming/1142320/decentralization-of-the-gambling-industry-through-blockchain

Decentralized Applications (dApps) Definition. (2021). https://www.investopedia.com/terms/d/decentralized-applications-dapps.asp

Solidity — Solidity 0.8.14 documentation. (n.d.). Retrieved May 14, 2022, from https://docs.soliditylang.org/en/develop/

What are smart contracts on blockchain? | IBM. (n.d.). Retrieved May 14, 2022, from https://www.ibm.com/topics/smart-contracts