Mental Poker: A Decentralized Implementation

Version 3.0.0

Galeri Marco
marco.galeri@studio.unibo.it

Patrignani Luca
luca.patrignani3@studio.unibo.it

December 5, 2025

Contents

This project implements a decentralized poker game (Texas Hold'em) without requiring a trusted third party. The implementation guarantees fairness even when $k-1$ out of k players form a malicious coalition attempting to rig the game. The system consists of three main components: a cryptographic deck library implementing the Wei & Wang mental poker protocol[15], a poker game engine managing game logic and state, and a blockchain-based distributed ledger ensuring non-repudiation of player actions. The architecture follows a peer-to-peer design with broadcast-based communication primitives, enabling fully distributed gameplay over local area networks. The delivered artifacts include a Go library with all the core functionalities and a command-line interface that exposes this library as a usable application, allowing end-to-end experimentation and gameplay while deliberately minimizing effort on user interface concerns so that the project work could focus more on the distributed and cryptographic aspects.

1 Goals of the project

The project's goal is to implement a decentralized poker game without a trusted third party. In particular, our implementation guarantees that given k players, an adversary cannot manipulate the game even when controlling $k - 1$ players.

1.1 Detailed Description

Mental poker is a classic problem in cryptographic protocol design, first formulated by Shamir, Rivest, and Adleman in 1979[12]. The challenge is to enable multiple distrustful parties to play a fair card game remotely without requiring a trusted dealer or central authority. Traditional online poker platforms rely on a centralized server that acts as the trusted dealer, shuffling cards and dealing them to players. However, this approach requires players to trust the server operator.

Our implementation addresses this trust problem by distributing the dealer's responsibilities among all players using cryptographic protocols. Each player participates in deck preparation, shuffling, and card dealing operations. The system ensures that:

- No single player or coalition of $k - 1$ players can determine the order of cards in the deck
- Players cannot see cards dealt to others until those cards are revealed
- All players can verify that cards shown by others are legitimate and were not fabricated
- Player actions (bets, raises, folds) are immutably recorded and cannot be repudiated

The implementation targets local area network gameplay, where players connect directly to each other in a peer-to-peer fashion. This use case is particularly relevant for casual games among friends or small tournaments where players want to ensure fairness without relying on external services.

1.2 Usage scenarios

1.2.1 Scenario 1: Casual Game Among Friends

Alice, Bob, and Charlie want to play poker remotely. They each run the mental poker application on their computers. Once all players are connected, they can start a game session. The system handles all cryptographic operations transparently - players simply interact through the command-line interface to make their betting decisions (call, raise, fold). The blockchain ledger maintains a complete history of the game for post-game verification if needed.

1.2.2 Scenario 2: Tournament Play

A group of players organizes a small poker tournament. Multiple game sessions run concurrently with different player groups. The decentralized architecture ensures that no central authority can manipulate outcomes. Players can verify the integrity of their specific games by examining the blockchain ledger. The absence of a central server eliminates single points of failure and potential manipulation.

1.2.3 Scenario 3: Educational Demonstration

Computer science students learning about cryptographic protocols use the mental poker implementation as a practical example. They can examine the source code to understand how the Wei & Wang protocol works[15], run games while monitoring network traffic to see the cryptographic operations, and modify the code to experiment with different protocol parameters or attack scenarios (demonstrating security guarantees when attempting to cheat).

1.3 Definition of done

The delivered artifacts are:

- A golang library (`domain/deck`) which implements a decentralized deck and operations such as shuffling, dealing face-down cards, and revealing cards. The implementation follows the paper by Wei and Wang [15].
- A golang library (`domain/poker`) which implements the actual game of poker, including game state management, betting rounds, hand evaluation, and winner determination.
- A golang library (`ledger`) implementing a blockchain-based distributed ledger for recording player actions with consensus voting mechanisms[6].
- A golang library (`network`) providing peer-to-peer communication primitives (broadcast and all-to-all operations) over TCP.
- A command-line interface (`cmd/`) for playing poker games with a text-based user interface.

The functionalities are tested automatically with comprehensive unit tests for each component. Integration tests validate the end-to-end workflow of complete poker games with multiple players.

2 Background and link to the theory

2.1 Architectural Styles

The project employs a **peer-to-peer (P2P) architectural style**[14], where all nodes (players) have equal responsibilities and capabilities. Unlike client-server architectures,

there is no central coordinator or privileged node. Each player runs identical software and participates equally in all cryptographic protocols.

This choice aligns with the project requirements for decentralization and fault tolerance. A centralized architecture would introduce a trusted third party, contradicting the fundamental goal. The P2P style ensures that:

- No player has privileged access or control
- Security properties hold even with $k - 1$ malicious players

2.2 Interaction Patterns

The implementation uses **broadcast-based communication patterns** inspired by high-performance computing frameworks, specifically the Message Passing Interface (MPI) [10]. Two primary communication primitives are employed:

1. **Broadcast (one-to-all)**: A single player sends data to all other players. This is analogous to `MPI_BCAST` in MPI. Used when one player needs to share information with the group (e.g., broadcasting encrypted cards during shuffling).
2. **All-to-all (all-to-all)**: Every player sends data to every other player simultaneously. Analogous to `MPI_ALLTOALL`. Used when all players must contribute information (e.g., generating random elements for card initialization).

Both primitives implement **barrier synchronization**, blocking until all players complete the operation. This ensures consistent state across all nodes and simplifies reasoning about protocol correctness.

The communication model follows the **Single Program, Multiple Data (SPMD)** paradigm [16], commonly used in parallel programming. All nodes execute the same program (protocol logic), but each operates on different data (its own secret key, hand, etc.) and may take different execution paths based on its player ID.

2.3 Cryptographic Frameworks

The project uses **Kyber**[8], a Go cryptography library providing elliptic curve operations. Specifically, the Ed25519 curve suite[1] is used for:

- **Scalar operations**: Secret key generation and manipulation
- **Point operations**: Card representation as curve points, encryption through scalar multiplication
- **Homomorphic properties**: Enabling multiple players to collectively encrypt the deck without revealing individual secret keys[7]

The Wei & Wang protocol leverages the discrete logarithm problem on elliptic curves to provide computational security guarantees[4].

3 Requirements Analysis

3.1 Functional requirements

The player is able to play a game of Texas Hold'em poker following standard rules[13]:

1. **Initial Deal**: The player receives two private cards (hole cards), which are not visible to any other players.
2. **Betting Actions**: When it is the player's turn, they can perform one of the following actions:
   - **Call**: Match the current highest bet
   - **Raise**: Increase the current bet amount
   - **Fold**: Exit the current hand, forfeiting any bets made
   - **Check**: Pass without betting (only when no bet has been made)
   - **All-in**: Bet all remaining chips
3. **Flop Phase**: When the initial betting round completes (big blind checks or all players call the raise), three community cards are dealt face-up on the board.
4. **Turn Phase**: After the flop betting round completes, one additional community card is dealt face-up.
5. **River Phase**: After the turn betting round completes, a final community card is dealt face-up.
6. **Showdown**: After the river betting round completes, all remaining players reveal their hole cards. The winner is determined by the best five-card poker hand using any combination of hole cards and community cards.
7. **Early Termination**: If only one player remains in the game (all others have folded), that player wins immediately without proceeding to showdown.
8. **Pot Distribution**: The winner(s) receive the pot. In cases of all-in scenarios, side pots are calculated and distributed appropriately to eligible players.

3.2 Non functional requirements

3.2.1 Deck's requirements

Each of the following requirements must be satisfied even when a coalition of $k - 1$ players (out of k total) attempts to manipulate the game without the remaining player detecting the manipulation:

- **Shuffle Fairness**: When shuffling the deck, no player or coalition can control the final order of the cards. The order must be uniformly random from the perspective of any player.

- **Shuffle Privacy**: After the deck is shuffled, no player knows the order of the cards (beyond their own influence on the randomization).
- **Deck Completeness**: All 52 cards are present in the deck exactly once. No cards are missing, duplicated, or fabricated.
- **Card Privacy**: When a card is dealt to a player face-down, no other player can determine which card it is. The card remains private until the owning player chooses to reveal it.
- **Card Verifiability**: When a player reveals a card, other players can verify that the player is not lying about which card they hold. The revealed card must be provably one of the cards from the shuffled deck.
- **Cryptographic Security**: The protocols provide computational security based on the discrete logarithm problem[4]. Breaking the security would require solving computationally infeasible problems.

3.2.2 Poker game's requirements

- **Fair Deck Usage**: The game uses a deck which meets all the cryptographic security requirements described above.
- **Action Non-repudiation**: When a player makes a move (call, raise, fold, check), they cannot later deny or repudiate that choice. All actions are recorded immutably in the blockchain ledger with consensus verification.
- **State Consistency**: All players maintain a consistent view of the game state (current round, pot size, player positions, community cards). Consensus mechanisms ensure agreement on state transitions.
- **Byzantine Fault Tolerance**: The game continues correctly even when some players behave maliciously (within the $\frac{2}{3}k$ coalition limit)[2].
- **Liveness**: The game makes progress as long as a sufficient number of honest players remain active. Malicious players cannot indefinitely stall the game.
- **Auditability**: Complete game history is recorded in the blockchain, enabling post-game verification and dispute resolution.

3.3 Top-down analysis

3.3.1 Deck implementation

Due to the decentralized nature of the project, every communication is modeled as a *broadcast*, where a player communicates with all other players simultaneously by sending identical information to everyone. This constraint arises from the requirement that no

player can have private communication channels with subsets of players, which could enable collusion.

The **broadcast communication model** has several implications for the architecture:

- **Symmetry**: All players observe the same messages in the same order, ensuring consistent state.
- **Transparency**: All protocol steps are visible to all participants, enabling verification but also requiring careful cryptographic design to protect privacy.
- **Simplicity**: The communication topology is fully connected, simplifying protocol design compared to partial network graphs.
- **Scalability limits**: Broadcast overhead grows with player count, making the approach most suitable for small to medium-sized games (up to 10-20 players).

3.3.2 Interaction pattern selection

The **MPI-inspired broadcast primitives**[10] were chosen because:

- **Protocol compatibility**: The Wei & Wang paper[15] describes operations that naturally map to broadcast and all-to-all patterns.
- **Barrier synchronization**: Built-in synchronization simplifies protocol implementation and reasoning about correctness.
- **Proven model**: MPI is a well-studied standard with decades of use in high-performance computing, providing confidence in the design.
- **Composability**: Complex protocols can be built from these simple primitives.

3.3.3 Technology framework selection

Go (Golang)[5] was selected as the implementation language because:

- **Concurrency**: Built-in goroutines and channels simplify concurrent network communication.
- **Performance**: Compiled language with performance suitable for cryptographic operations.
- **Networking**: Excellent standard library support for TCP/IP networking.
- **Cryptography**: Rich ecosystem of cryptographic libraries, including Kyber for elliptic curve operations.
- **Testing**: Strong testing framework and tooling for verification.
- **Simplicity**: Clear syntax and design facilitate collaboration and maintainability.

The **Kyber cryptography library**[8] (Dedis Advanced Crypto Library for Go) provides:

- **Homomorphic cryptography**: Kyber supports various suites of homomorphic cryptographic encryptions such as curve operations[1] required by the Wei & Wang protocol.
- **Security**: Well-audited, academically-developed library from EPFL.
- **Abstraction**: Clean API for scalar and point operations.

4 Design

The design process employed a hybrid top-down and bottom-up approach. The high-level architecture was determined by the requirements for decentralization and cryptographic security (top-down), while specific implementation details emerged from practical constraints and library capabilities (bottom-up).

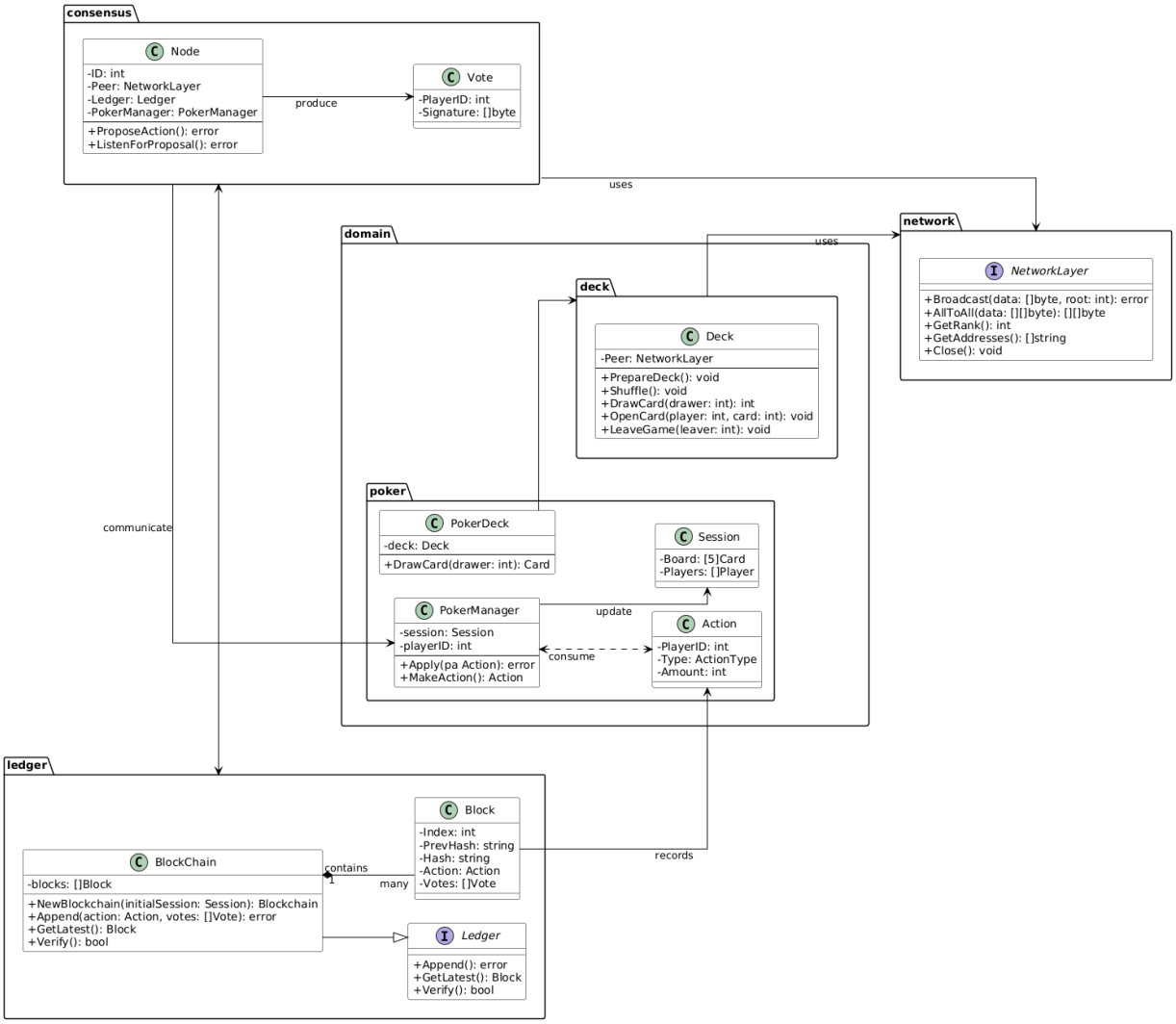

Figure 1: UML class diagram showing the main domain entities and relationships

The solution was designed with four main components:

- **The Deck**: Cryptographic layer implementing the Wei & Wang mental poker protocol[15]
- **The Game Logic**: Poker game mechanics, state management, and rules enforcement
- **The DLT (Distributed Ledger Technology)**: Event-sourcing blockchain[6] to record actions
- **The Consensus Node**: Voting mechanisms for action validation[2]

Additionally, supporting components provide:

- **Network Layer**: Peer-to-peer communication primitives
- **User Interface**: Command-line interface for player interaction (not shown here)
- **Discovery Service**: Service for announcing and discovering peers in the local network

4.1 Components

4.1.1 Domain: Deck Component

The Deck component implements the cryptographic layer based on the Wei & Wang mental poker protocol[15]. It ensures that no player can control or predict the card order, and that cards dealt face-down remain private until revealed.

Deck Entity The `Deck` struct is the core cryptographic abstraction representing a secure card deck. Each player maintains their own Deck instance, participating in collective protocols for shuffling, dealing, and revealing cards. The deck uses elliptic curve cryptography (Ed25519)[1] through the Kyber library[8], where each card is represented as a point on the curve. Cards are encrypted with each player's secret key during the shuffle protocol, ensuring that the final deck order incorporates randomness from all participants.

Key operations include:

- `PrepareDeck()`: Initializes n cards as cryptographic points using distributed random element generation
- `Shuffle()`: Sequential protocol where each player encrypts the entire deck with their secret key
- `DrawCard(drawer)`: Collective decryption allowing only the specified player to learn the card value

- `OpenCard(player, card)`: Broadcasts and verifies a player's revealed card
- `LeaveGame(leaver)`: Removes a disconnected player's encryption layer from the deck

4.1.2 Domain: Poker Component

The Poker component manages game state, enforces Texas Hold'em rules[13], and evaluates hand strengths. It operates independently of the cryptographic deck, focusing purely on game mechanics.

Session Entity The `Session` struct maintains the complete game state at any point in time. It tracks community cards on the board, player states, pot distributions, betting information, and the current game phase (PreFlop, Flop, Turn, River, Showdown). The session serves as the single source of truth for game state.

Critical methods handle state transitions:

- `ApplyAction()`: Executes validated player actions (fold, bet, raise, call, check, all-in)
- `recalculatePots()`: Manages main and side pots for all-in scenarios
- `advanceRound()`: Transitions between betting rounds when all players have acted
- `winnerEval()`: Determines winner(s) at showdown using hand evaluation

Player Entity Each `Player` represents a participant with their hole cards, chip count, current bet, and fold status. The poker manager coordinates player turns and validates that actions are taken by the correct player.

Card and Hand Evaluation The `Card` struct represents standard playing cards with rank and suit. The `Evaluator` module implements poker hand ranking algorithms[13], comparing hands to determine winners according to standard Texas Hold'em rules (Royal Flush through High Card).

Action Entity The `Action` struct represents a proposed poker move with a unique ID, player ID, poker action payload, timestamp, and Ed25519 signature[1].

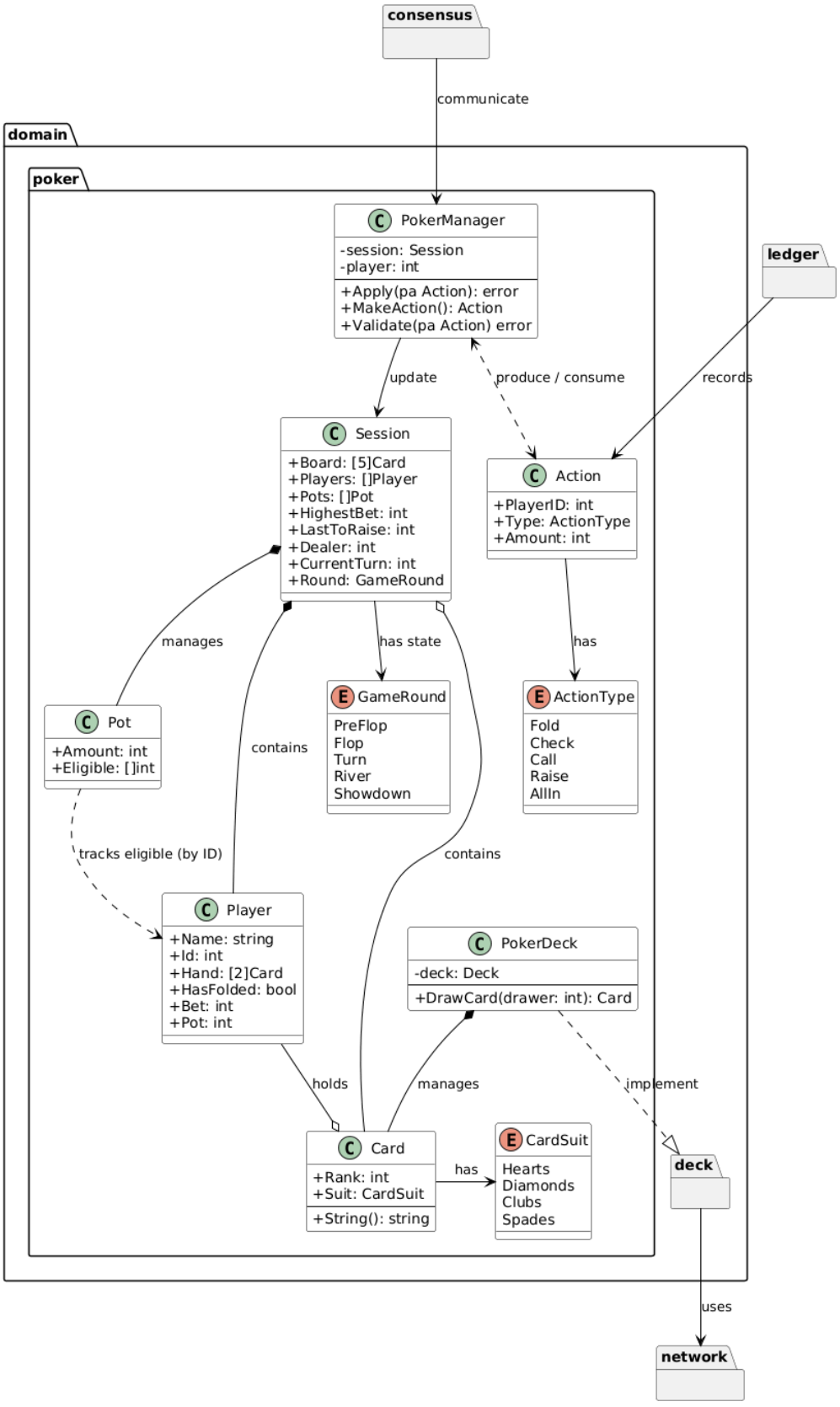

Figure 2: UML class diagram showing the package poker details

4.1.3 Ledger Component

The Ledger component implements an event-sourced[6], blockchain-based replicated log ensuring immutability and non-repudiation of all game actions.

Blockchain Entity The `Blockchain` struct maintains an append-only chain of blocks, each cryptographically linked to its predecessor via SHA-256 hashing[11]. The genesis block records the initial game state with all players and starting chip counts. Each subsequent block contains a player action, the resulting game state, consensus votes, and metadata.

The blockchain provides:

- `Append()`: Validates and adds consensus-approved actions with quorum votes
- `Verify()`: Checks the entire chain integrity by validating hashes and vote counts
- `GetLatest()`: Retrieves the current game state from the most recent block

Block Structure Each `Block` is self-contained and includes:

- Sequential index and timestamp
- Cryptographic hash of the previous block (forming the chain)
- The `PokerAction` that caused the state transition
- Consensus `Votes` from all players (with signatures)
- Metadata including proposer ID, quorum threshold, and optional extra data (e.g., ban reasons)

This structure enables post-game auditing, dispute resolution, and verification that all state transitions received proper consensus approval.

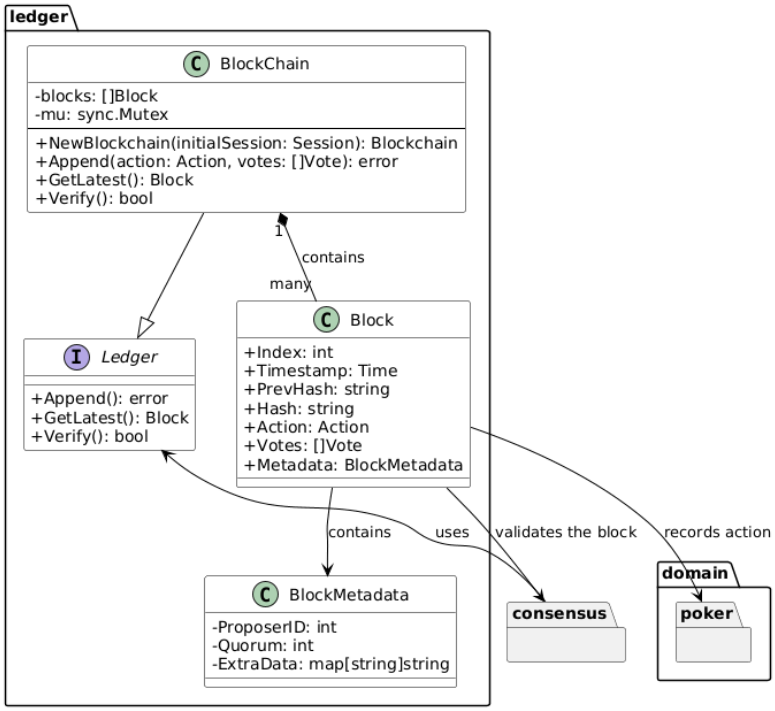

Figure 3: UML class diagram showing the detail of the ledger component

4.1.4 Consensus Component

The Consensus component implements a Byzantine Fault Tolerant (BFT) voting protocol[2] ensuring that all players agree on action validity before state changes occur.

Node Entity The `ConsensusNode` manages the distributed consensus protocol for each player. It maintains cryptographic key pairs (Ed25519)[1] for signing and verifying messages, tracks peer public keys, calculates quorum thresholds using the BFT formula[2] $\lceil \frac{2n+2}{3} \rceil$, and coordinates with the poker state machine and blockchain ledger.

The consensus workflow follows a propose-vote-commit pattern:

- `ProposeAction()`: The current player broadcasts a signed action proposal
- `WaitForProposal()`: Other players receive and validate the proposal
- Vote broadcasting: Each player broadcasts ACCEPT or REJECT with signature
- `checkAndCommit()`: If quorum accepts, the action is applied and recorded; if quorum rejects, the proposer is banned

Vote Entity The `Vote` struct contains the voter's decision (ACCEPT/REJECT), validation reason, and cryptographic signature[1]. All signatures are verified using the voter's public key to prevent forgery.

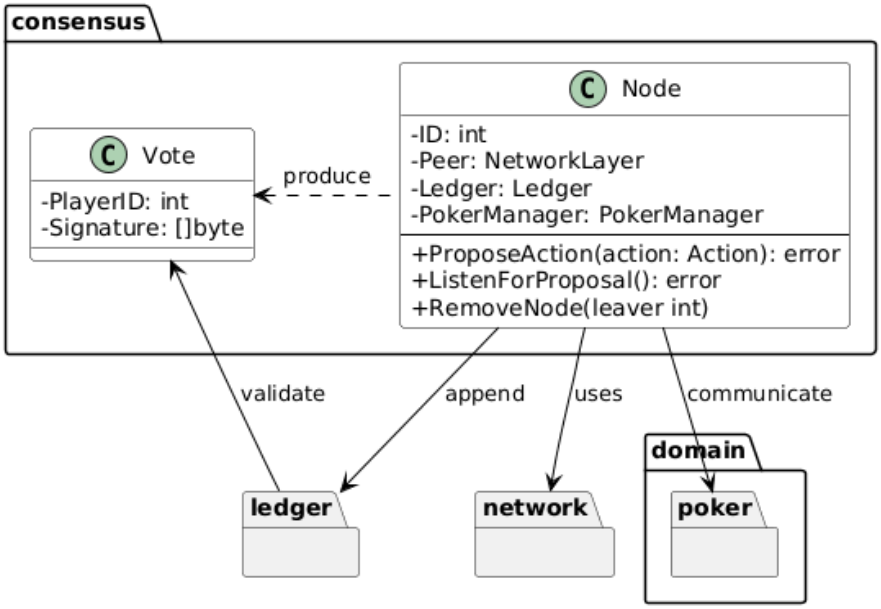

Figure 4: UML class diagram showing the consensus component details

4.1.5 Network Component

The Network component provides peer-to-peer communication primitives[14] abstracting the underlying TCP connections.

NetworkLayer Interface The `NetworkLayer` interface defines two fundamental communication patterns inspired by MPI[10]:

- `Broadcast(data, root)`: One player sends identical data to all peers (analogous to `MPI_BCAST`)
- `AllToAll(data)`: Every player sends data to every other player simultaneously (analogous to `MPI_ALLTOALL`)

Both operations include timeout variants and implement barrier synchronization, blocking until all participants complete the communication. This ensures consistent ordering of messages and simplifies protocol implementation.

Peer Implementation The `Peer` struct implements the `NetworkLayer` interface using TCP sockets. Each peer maintains connections to all other peers in a fully-connected topology. The implementation handles connection management, message serialization/deserialization, and timeout enforcement. The `p2p.go` wrapper provides stable interfaces for the cryptographic deck operations.

4.1.6 The discovery service

A simple discovery service is implemented using UDP broadcasting to announce and discover peers on the local network. Each instance periodically broadcasts its presence with its IP address and listening port. Peers listen for these broadcasts to build a list of available players before establishing TCP connections for gameplay.

4.2 Communication

As previously mentioned, each player communicates only via two primitives[10]:

- `broadcast`: A player sends information to all other players
- `all-to-all broadcast`: Every player sends information to every other player

Both operations implicitly behave as barrier synchronizations, blocking thread execution until all nodes complete the communication. This design decision was made while studying the Wei and Wang paper [15], whose protocols exclusively use these primitives.

4.2.1 The high performance computing similarity

The communication primitives are inspired by the *Message Passing Interface*[10], an open library standard for distributed memory parallelization widely used in high-performance computing. For those familiar with MPI, our primitives correspond to:

- `broadcast` $\leftrightarrow$ `MPI_BCAST`
- `all-to-all broadcast` $\leftrightarrow$ `MPI_ALLTOALL`

We observed that the Wei & Wang protocols are executed by all nodes identically. This resembles the *Single Program, Multiple Data (SPMD)* computational model [16], the most widely used paradigm in parallel programming. SPMD states that all nodes run the same program, but each node may choose different execution paths based on its ID.

We believe this model extends naturally to decentralized systems like ours, where:

- All players run identical protocol code
- Each player uses its unique ID to determine its role in each protocol step
- Different execution paths are taken based on player ID (e.g., "if GetRank() == dealer")

This insight guided our architectural decisions and simplified implementation by enabling code reuse across all nodes.

4.2.2 Consensus Protocol Sequence

The consensus protocol ensures that all players agree on the validity of each action before it modifies the game state. Figure 5 illustrates the complete message flow for a successful action proposal.

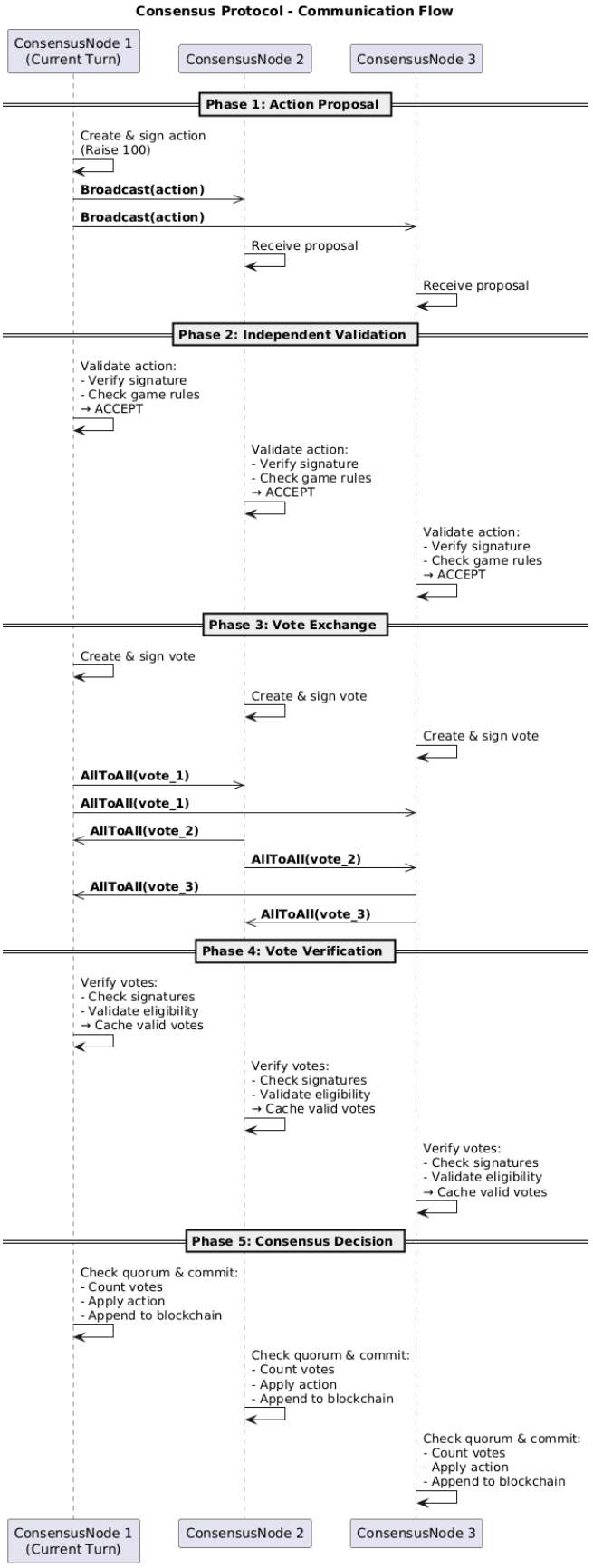

Figure 5: Sequence diagram of the consensus protocol showing the five-phase distributed workflow

The protocol proceeds through five distinct phases:

Phase 1: Action Proposal The player whose turn it is creates a poker action (e.g., CHECK, FOLD, RAISE) and submits it. The node generates a unique action ID by combining the player ID, the action payload hash, and 128 bits of cryptographic entropy. The action is timestamped and signed with the player's private key through the elliptic curve cryptography[1]. The signed action is then broadcast to all peers using the `Broadcast` primitive.

All other players are executing `WaitForProposal()`, blocking until they receive the broadcast from the current player or the default action in case of timeout.

Phase 2: Proposal Validation Upon receiving a proposal, each consensus node locally validates it:

1. **Signature verification**: The node verifies the signature using the proposer's public key to ensure authenticity and prevent tampering.
2. **Game rule validation**: The proposal is passed to the poker manager, which validates turn order, action legality (e.g., sufficient chips for a bet) and compliance with Texas Hold'em rules[13].

If any validation check fails, the node prepares to vote `REJECT` with a descriptive reason (e.g., "out-of-turn", "insufficient-chips", "bad-signature"). Otherwise, the node votes with `ACCEPT`.

Phase 3: Vote Broadcasting Each node creates a `Vote` struct containing:

- The action ID being voted on
- The voter's node ID
- The vote value (`ACCEPT` or `REJECT`)
- A human-readable validation reason
- A signature on the serialized vote data

All nodes broadcast their votes simultaneously using the `AllToAll` primitive. This operation blocks until every node has both sent its vote and received votes from all peers, implementing barrier synchronization. The 30-second timeout ensures the protocol progresses even if a peer disconnects.

Phase 4: Vote Collection and Verification Each node receives the vote array from the `AllToAll` operation and processes it through `onReceiveVotes()`:

1. **Proposal consistency**: Verify that all votes reference the same action ID.
2. **Signature verification**: For each vote, verify the Ed25519 signature[1] using the voter's public key from the peer map.
3. **Voter eligibility**: Ensure the voter is a current player in the session and has not been banned.
4. **Vote caching**: Store valid votes in a local map indexed by voter ID.

Malformed votes or votes with invalid signatures are logged but do not halt the protocol, providing resilience against Byzantine failures.

Phase 5: Quorum Check and Commit The `checkAndCommit()` function evaluates whether consensus has been reached. The quorum threshold is computed using the Byzantine Fault Tolerance formula[2]:

$$\text{quorum} = \left\lceil \frac{2n+2}{3} \right\rceil$$

where n is the current number of players. This ensures safety with up to $\lfloor (n-1)/3 \rfloor$ Byzantine (malicious or faulty) nodes.

The function counts `ACCEPT` and `REJECT` votes and determines the outcome:

- **Accepts $\geq$ quorum**: The action is legitimate. The consensus node applies the action to the poker state machine, appends a block to the blockchain[6], and clears cached proposal and votes.
- **Rejects $\geq$ quorum**: The proposal is invalid and the proposer is Byzantine. The consensus node bans the player, records the ban action in the blockchain[6], and recalculates quorum.
- **Neither quorum reached**: Insufficient votes have been collected. The function returns an error, and the node waits for more votes to arrive.

Even if some nodes are Byzantine (sending different votes or proposals), the quorum mechanism ensures that the supermajority of honest nodes drives the consensus outcome[2]. The blockchain records the complete set of votes, enabling post-game auditing to identify Byzantine actors.

4.2.3 Network Topology and Message Flow

The system operates in a **fully-connected peer-to-peer topology**, where each node maintains direct TCP connections to all other nodes. This design choice simplifies the implementation of broadcast and `AllToAll` primitives, as there are no routing concerns or intermediate hops.

Message serialization uses JSON encoding for human readability and debugging convenience and then in binary to reduce bandwidth. The communication model follows the **Single Program, Multiple Data (SPMD)** paradigm[16] from parallel computing: all nodes run identical code but make different execution decisions based on their node ID and the current game state. For example, only the node whose turn it is executes `ProposeAction()`, while others execute `WaitForProposal()`.

This architecture ensures that the system scales linearly with the number of players in terms of network traffic (each message must be sent to $n - 1$ peers), but maintains constant time complexity for consensus rounds (one broadcast and one `AllToAll` per action).

4.3 Behaviour

We can model the behavior of the poker game as a state machine with the following states:

- **PreFlop**: Initial betting round after dealing hole cards
- **Flop**: Betting round after dealing 3 community cards
- **Turn**: Betting round after dealing the fourth community card
- **River**: Betting round after dealing the fifth community card
- **Showdown**: Hand evaluation and pot distribution
- **GameOver**: Terminal state when the game ends

4.3.1 State Transitions

PreFlop → Flop Triggered when the betting round completes (all active players have called the current bet or checked). Entry action: Deal 3 community cards.

Flop → Turn Triggered when the betting round completes. Entry action: Deal 1 community card.

Turn → River Triggered when the betting round completes. Entry action: Deal 1 community card.

River → Showdown Triggered when the final betting round is complete. Entry action: All remaining players reveal their hole cards.

Any State → Showdown Triggered when only one player remains (all other players have folded). The remaining player wins without revealing cards.

Showdown → PreFlop Triggered when starting a new round with the same players. Entry action: Reset player states, collect antes/blinds, deal new hole cards.

Showdown → GameOver Triggered when players choose to end the game or insufficient players remain.

4.3.2 State Invariants

Each state maintains specific invariants:

- **PreFlop**: Each player has exactly 2 hole cards; no community cards are visible
- **Flop**: Exactly 3 community cards are visible on the board
- **Turn**: Exactly 4 community cards are visible
- **River**: Exactly 5 community cards are visible
- **Showdown**: All community cards are visible; remaining players' hole cards are revealed

4.3.3 Player State

Individual players also have internal state:

- **Active**: Player is in the current hand
- **Folded**: Player has folded and cannot win the current hand
- **All-in**: Player has bet all chips and cannot make further bets
- **Spectator**: Player has lost all of his chips and is watching the game without influencing it.

The `ApplyAction()` method handles state transitions based on player actions, ensuring consistency through consensus validation.

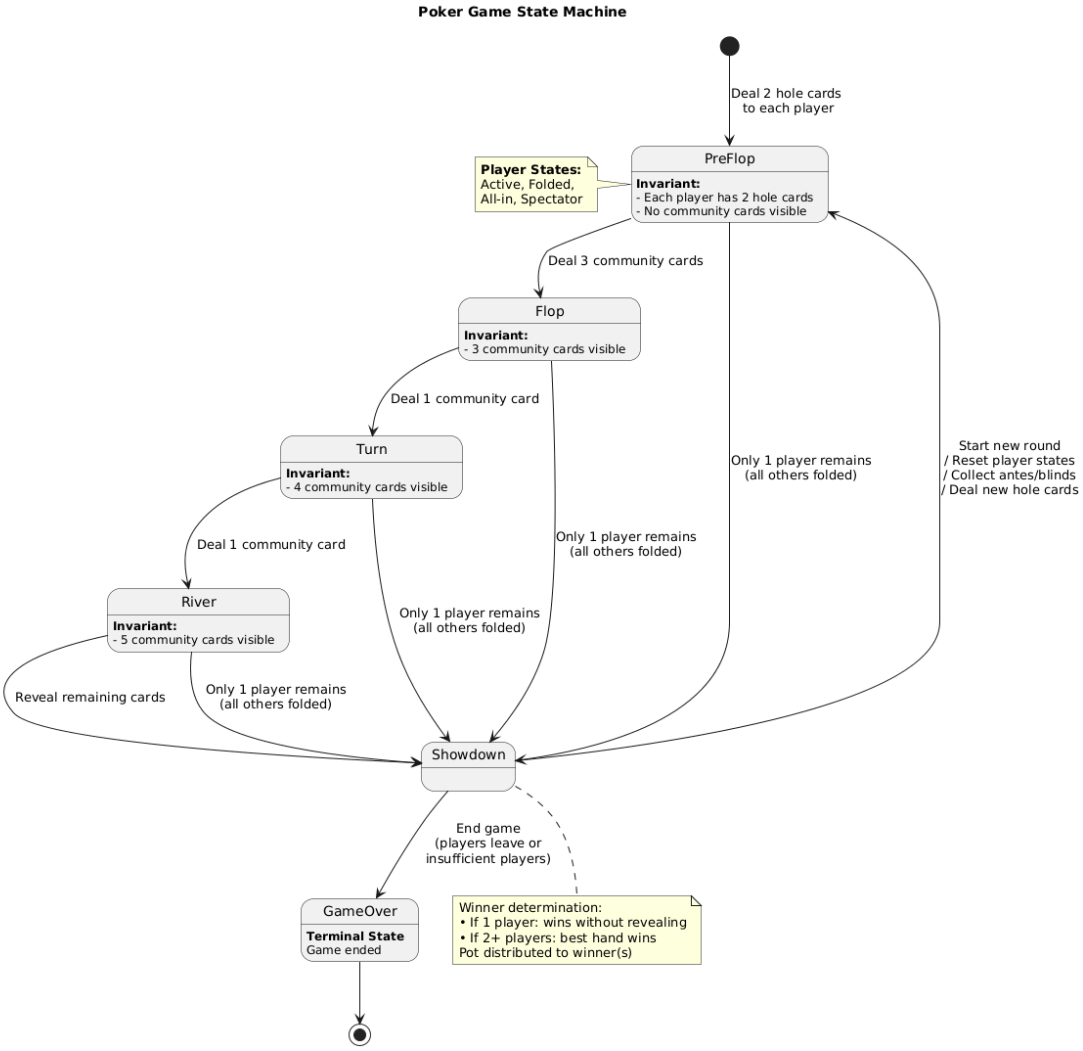

Figure 6: Finite state machine of a Texas Hold-em poker game

4.4 Distributed Ledger Technology

The system employs a **permissioned blockchain** as a distributed ledger to record all game actions immutably. This ensures transparency, auditability, and non-repudiation of player actions. Since the set of players is known and controlled, a permissioned model is appropriate. A **replicated log** similar to a *blockchain* data structure keeps track of the state of the game in an **event sourcing** fashion[6], recording all game actions with consensus verification. Each block contains[11]:

- Player action (type, amount)
- Consensus votes from all players
- Cryptographic hash linking to the previous block

This provides **non-repudiation**, so players cannot deny actions they performed, as all actions are cryptographically signed and verified by the group[2]. The replicated log also enables post-game auditing and dispute resolution.

4.5 Corner cases

4.5.1 Faults detection

The system detects several types of faults:

- **Malicious Stalling**: Timeouts on broadcast/all-to-all operations detect stalling players
- **Invalid cryptographic operations**: Card verification failures detect attempts to reveal fake cards
- **Consensus violations**: Actions without sufficient votes are rejected[2]
- **Blockchain integrity violations**: `Verify()` method detects tampered history[11]
- **Protocol violations**: Invalid game actions are rejected

4.5.2 Recovery strategies

Recovery mechanisms include:

- **Detected cheating**: Malicious player is banned via consensus vote[2]
- **Blockchain fork**: Not applicable - no forks possible with consensus-based append-only design
- **State inconsistency**: Detected via consensus; inconsistent players must resync or are excluded

4.5.3 Graceful shutdown

The system supports graceful shutdown:

1. Player signals intent to leave (via CLI)
2. The current hand completes if in progress
3. The Player's `LeaveGame()` method is called
4. The Player's encryption is removed from the deck
5. Network connections are closed
6. Application exits cleanly

5 Salient implementation details

This section highlights key implementation details extracted from the actual codebase, demonstrating how theoretical protocols are realized in Go[5].

5.1 Cryptographic Deck Implementation

The deck uses Kyber's[8] Ed25519 elliptic curve suite[1]. Each card is represented as a cryptographic point, with randomness contributed by all players to prevent manipulation.

Listing 1: Distributed random element generation from deck.go

```
func (d *Deck) generateRandomElement() (kyber.Point, error) {
  // initialize random generator of cyclic group G
  gj := suite.Point().Mul(suite.Scalar().Pick(
    suite.RandomStream()), nil)
  hj := suite.Point().Mul(suite.Scalar().Pick(
    suite.RandomStream()), nil)
  for gj.Equal(hj) {
    hj = suite.Point().Mul(suite.Scalar().Pick(
      suite.RandomStream()), nil)
  }

  lambda := suite.Scalar().Pick(suite.RandomStream())
  gPrime := suite.Point().Mul(lambda, gj)
  _, err := d.allToAllSingle(gj)
  if err != nil {
    return nil, err
  }
  _, err = d.allToAllSingle(gPrime)
  if err != nil {
    return nil, err
  }
  _, err = d.allToAllSingle(hj)
  if err != nil {
    return nil, err
  }

```

```
27    hPrime := suite.Point().Mul(lambda, hj)
28    hArray, err := d.allToAllSingle(hPrime)
29    if err != nil {
30      return nil, err
31    }
32
33    hResult := hArray[0]
34    for i := 1; i < len(hArray); i++ {
35      hResult.Add(hResult, hArray[i])
36    }
37    return hResult, nil
38  }
```

The `generateRandomElement()` method implements Protocol 2 from the Wei & Wang paper[15]. Each player contributes randomness through `AllToAll`[10] exchanges of cryptographic points, ensuring that no single player can predict or control the final card values.

5.2 Shuffle Protocol

The shuffle implementation follows Protocol 3 from the Wei & Wang paper[15], where each player sequentially encrypts and permutes the deck.

5.3 Card Drawing Protocol

Drawing a card requires collective decryption, allowing only the designated player to learn the card value using Ed25519 elliptic curve cryptography[1].

5.4 Documentation

Comprehensive GoDoc documentation[5] is provided for all public APIs. Examples:

- Package-level documentation in `doc.go` files
- Function-level documentation with parameter and return descriptions
- Example usage in test files

The documentation can be found on the official go module site.

6 Validation

6.1 Testing Strategy

The testing strategy employs automated testing for **Unit tests**

6.2 Test Coverage

The test coverage statistics can be found in the CodeCov page of the project here.
Coverage is measured with:

```
go test -coverprofile=coverage ./...
go tool cover -html=coverage
```

6.3 Manual Testing

Manual testing scenarios:

1. **3-player game**: Alice, Bob, Charlie play a complete game. Verify all betting rounds, community cards, showdown.
2. **Network disruption**: Verify graceful termination handling.
3. **Malicious stalling**: Verify timeout default action works properly.

7 Deployment Instructions

7.1 Download Prebuilt Binaries

Prebuilt binaries for end user are available for download in the release section of the GitHub repository:

```
https://github.com/luca-patrignani/mental-poker/releases
```

7.2 Build from source

7.2.1 Prerequisites

- **Go**: Version 1.21 or later

  ```
  # Verify Go installation
  go version
  ```

- **Git**: For cloning the repository

  ```
  git --version
  ```

- **Network**: All players must be on the same local network or have direct connectivity

7.2.2 Installation Steps

1. Clone the repository:

   ```
   git clone https://github.com/luca-patrignani/mental-poker.git
   cd mental-poker/v*
   ```

2. Install dependencies:

   ```
   go mod download
   go mod verify
   ```

3. Build the executable:

   ```
   go build -o mental-poker ./cmd
   ```

This creates a `mental-poker` executable in the current directory.

7.3 Network Setup

For each player, determine their network address:

```
# On Linux/Mac
ip addr show
ifconfig

# On Windows
ipconfig
```

Note that all players must be on the same local network (LAN) or have direct connectivity to each other. The application does not support NAT traversal or internet play. Furthermore the application uses multicast UDP address 239.0.0.1 at port 53551 for UDP discovery, so firewall must be configured to allow traffic on this address and port.

7.3.1 Running the Application

Default startup:

```
./mental-poker <Your IP address>
```

This starts the application and displays connection information for other players with the default timeout.

Specified Timeout:

```
./mental-poker -t <TIMEOUT> <Your IP address>
```

This starts the application and displays connection information for other players with the specified timeout.

8 Conclusions

This project successfully implemented a decentralized poker game meeting all specified security and functionality requirements using the Wei & Wang mental poker protocol[15], Byzantine Fault Tolerant consensus[2], blockchain-based event sourcing[6], and Go programming language[5].

8.1 Achievements

The system enables trustless poker gameplay among multiple players without requiring a central authority, with all cryptographic primitives[1] properly implemented and validated.

8.2 Future Works

Several extensions could enhance the system:

- **Graphical user interface**: A web-based or desktop GUI would improve usability
- **Chip transactions**: A cryptocurrency or token system for real-money betting
- **Zero-knowledge proofs**: More efficient card verification[7]
- **Tournament management**: Support for multi-table tournaments
- **Reputation system**: Tracking player behavior to identify cheaters[2]

8.3 Personal Consideration

This project provided valuable experience in cryptography[1][4], distributed systems[16][10], blockchain technology[6], and software engineering practices. The successful implementation demonstrates that complex theoretical concepts can be realized as practical, working systems[15][2].

References

[1] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 124–142. Springer, 2011.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186, 1999.

[3] Giovanni Ciatto, Alfredo Maffi, Stefano Mariani, and Andrea Omicini. Smart contracts are more than objects: Pro-activeness on the blockchain. In Javier Prieto,

Ashok Das Kumar, Stefano Ferretti, António Pinto, and Juan Manuel Corchado, editors, *Blockchain and Applications*, volume 1010 of *Advances in Intelligent Systems and Computing*, pages 45–53. Springer, 2020.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. 2009.

[5] Alan A. Donovan and Brian W. Kernighan. The go programming language. 2015.

[6] Martin Fowler and Rebecca Parsons. Event sourcing. *Martin Fowler's Bliki*, 2005.

[7] Craig Gentry. Fully homomorphic encryption over the integers. *SIAM Journal on Computing*, 43:1–50, 2014.

[8] DEDIS Laboratory. Kyber: A modular library for cryptographic primitives in go, 2024.

[9] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *ACM Transactions on Programming Languages and Systems*, volume 4, pages 382–401, 1982.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*, nov 2023.

[11] National Institute of Standards and Technology. Fips 180-4: Secure hash standard (shs), 2015.

[12] Adi Shamir, Ronald L. Rivest, and Leonard M. Adleman. Mental poker. *The Mathematical Gardner*, pages 37–43, 1981.

[13] David Sklansky. *The Theory of Poker*. Two Plus Two Publishing, expanded edition edition, 1994.

[14] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160. ACM, 2001.

[15] Tzer-jen Wei and Lih-Chung Wang. A fast mental poker protocol. *IACR Cryptology ePrint Archive*, 2009:439, 05 2012.

[16] Wikipedia contributors. Single program, multiple data — Wikipedia, the free encyclopedia, 2025. [Online; accessed 13-February-2025].