# Ruban: Mutually-distrustful Turn-based P2P Transactions

Abdu Mohamdy
*Stanford University*

## Abstract

We re-envision the classic Two Phase Commit (2PC) Protocol to build a P2P round-robin distributed system around the idea of offers and counters. In this system, a peer can propose an offer to all other participants in the system which they can independently accept or counter. Counters can be stacked to create a chain and reach a final state that all participants accept.

All participants are mutually distrustful and we use the P2P nature of the system as well as public-key cryptography to to introduce the concept of Contesting to provide majority-based authenticity and security without a large traffic overhead.

## 1 Introduction

P2P Protocols that require strong consistency and high-availability often have to strike a balance between performance and decentralization. For example, many of these systems rely on a central leader to achieve atomicity and consensus. A static central leader creates a single of point of failure where the system is unable to continue without the leader. On the other hand, dynamically elected leaders and leaderless system often suffer from poor performance, lead to traffic amplification, and are much harder to scale. [1]

Our goal is to build a distributed system for turn-based games where participants take turns playing in a round-robin fashion. Such gaming schemes provide a unique opportunity: they allow us to partition data in a linearizable fashion based on turns. Each player gets a chunk in a shared log according to their turn in the game. Only when it's a player's turn, can they start a new action. However, any action that a player makes (including, e.g., ending their turn) can be vetoed by any other player (e.g. by activating a trap card) even when it's not their turn. Moreover, those vetoes can be overridden and the overrides themselves can be countered in a recursive fashion.

Moreover, the competitive nature of these games means players are incentivized to cheat. So any protocol cannot be cooperative and needs to be mutually-distrustful. Not only do players need to make sure that all other players are following the rule, they also need to make sure that players do *not* ignore their vetos. For example, suppose player `P1` plays an Action `A1` which another player `P2` counters. `P2` needs to ensure that `P1` won't just send a `COMMIT` to all other players in the system and ignore the counter. We require that the majority of players are alive and honest.

These stronger necessary security requirements make traditional solutions impractical: a centralized system is not secure against a malicious coordinator and traditional distributed consensus protocols such as Paxos or Raft do *not* provide the desired security guarantees.

Our goal is to provide Byzantine Fault Tolerance along with Authenticity and Non-Repudiation while also keeping traffic congestion low. Traffic amplification is a serious problem since an immediate use of this (which we are in the process of doing) is to use this system on low-cost, low-resource chips such as Raspberry Pi's.

Specifically our goal is a system that provides:

1. **Extensibility**: Chains can be recursively countered.
2. **Authenticity**: Peers should be confident that all other participants endorsed a chain before committing it.
3. **Scalability**: Traffic (packets sent) should grow in $O(n)$ where $n$ is the number of nodes.
4. **Byzantine Fault Tolerance**: The system should be able to gracefully recover from a node failing.
5. **Liveness through Maliciousness**: A malicious player does *not* cause the game to halt for honest participants.
6. **Proposer Autonomy**: The decision for which counter to choose, if more than one is available, belongs to the original proposer.
7. **Architecture Agnosticism**: The system should be easily extensible to other session layer including, e.g., Bluetooth, POSIX Sockets.

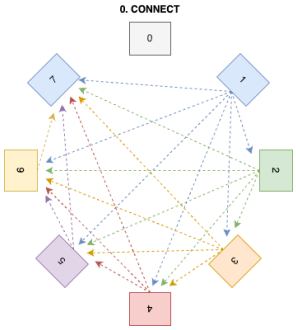


Figure 1: Nodes connecting to each other in an ascending fashion based on their PIDs provided by host (PID 0)

## 2 Background

### 2.1 Two Phase Commit

Two Phase Commit is one of the earliest distributed atomic commitment protocols that provides consistency and atomicity across node crashes. It allows many distributed nodes to either all commit or abort a transaction. One node is chosen as the leader and it's responsible for coordinating transactions to all other nodes.

As its namesake suggests, it works in two phases: PREPARE and COMMIT. During the PREPARE phase, the leader sends a PREPARE message to all the nodes asking them for a promise to commit a transaction. Each node independtly replies with a vote to either COMMIT or ABORT. Based on the outcome, the leader sends either a COMMIT or an ABORT which tells nodes whether or not to commit the transaction. Upon receiving a COMMIT, the nodes commit the transactions to a stable log. The Coordinator only commits the transaction once it receives ACKs from all nodes. Thus, if a node crashes during the protocol, it can ask the coordinator which references its log. At the end of the protocol, all nodes can be confident that the transaction was either committed by everyone, or aborted by everyone in an "All or Nothing" fashion.

Various optimizations of the protocol exist including "presume commit" and "presume abort" versions that allow nodes to assume that transactions committed or aborted to reduce latency and traffic. An notable optimization based on the "presume commit" version allows fewer log writes by the leader as well as fewer messages. [2]

### 2.2 Elliptic Curve Signatures

Elliptic Curve Digital Signature Algorithm (ECDSA) is a form of asymmetric public-key cryptography where a party can use a private key priv to create a digital signature s = sign(priv, d) on some data d and publish an associated public key pub that can be used to verify the signature was created by the owner of the private key (i.e. verify(pub, d, s) == TRUE). ECDSA signatures provide cryptographic Integrity, Authenticity, and Non-Repudiation, as they can be reduced to the Computational Diffie-Hellman Problem which is assumed to be difficult. Thus without knowledge of priv, a party cannot forge a signature s' such that verify(pub, d, s') == TRUE.

Other more popular Digital Signature Algorithms such as RSA are less scalable as they require longer keys for the same security, which is computationally expensive. Thus ECDSA is perfect for low-cost, low-compute devices.

## 3 Design

### 3.1 Setup

At its core, the system is really simple. We start with a basic setup where all players connect to a host that gathers player information (including connection information and public keys), assigns each player a unique Player ID (PID), and broadcasts the same connection information to all connected players. Players proceed to connect to each other in a round-robin tournament fashion using the connection information provided by the host.

One limitation with requiring our system to be architecture-agnostic is that some protocols do *not* allow a node to both listen for connections and attempt to connect to other nodes at the same time. We solve this by having the players initiate connections in an ascending order. Figure 1 shows this simple operation. A node i listens for connections until it is connected to i nodes, then it initiates connections to nodes (i+1)...n

The setup process creates a vulnerability if the host is malicious, and we plan on changing this in the future in favor of a Public Key Infrastructure as discussed in §5.1

### 3.2 Normal Operation

Once all players are connected, they use the unique PIDs to decide the order of play. A player P1 whose turn it is acts as a leader in a Two-Phase Commit. The protocol proceeds as follows:

1. P1 sends a PORPOSE message to all participants with the proposed chain of actions A1 as in Figure 2.

2. Upon receiving a PROPOSE message, players decide whether to:

   **Accept** : They hash the chain A1, sign it with their private key and send an OK message with the hash and signature.

   **Counter** : They append actions A2 to create the chain A1, A2 and send a COUNTER message back to the leader with the new chain.

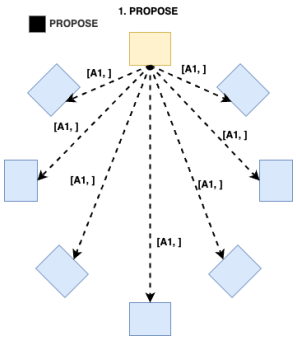


Figure 2: Proposer (yellow) proposes chain `[A1, ]` by broadcasting a `PROPOSE` message to all participants.

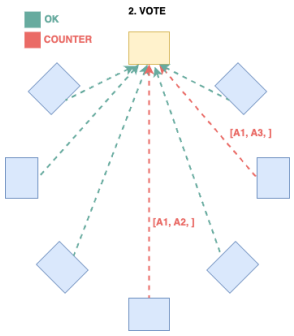


Figure 3: Players respond to `P1`'s `PROPOSE` message. In this case Players 2 and 4 (clockwise) send a `COUNTER` while all other players send an `OK` along with the signature using their private key

**Reject** : They reply with a `COUNTER` message with a hash of the chain.

Figure 3 shows an example of responses the original `PROPOSE` message.

3. Once `P1` receives all responses (or successfully contests a player, see §3.3):

   (a) If all players responded with `OK`, `P1` verifies the signatures using the public keys and sends a `COMMIT` with the hash of the chain and signatures.

   (b) If even a single player countered, `P1` either accepts the counter as-is, or counters the counter by appending more actions. Once done, `P1` repeats Step 1 with the new chain.

4. Once all players receive a `COMMIT` message, they verify the signatures using the public keys, and if they are valid, they commit the chain of actions and sends an `ACK` to the leader.

Since we assume CDH is hard, forging a player's signature is also hard. Thus, upon receiving a `COMMIT` with valid signatures, players can be assured that all players have indeed endorsed the chain of actions with no more counters.

## 3.3 Contesting

Contesting is a mechanism for the leader to alert others that a player `P2` either went offline, or played an illegal action. If a plyer `P1` has reason to believe that another player `P2` is dead or acting maliciously, they can send a `CONTEST P2 C` to all other players. A `CONTEST` is sent as a `PROPOSE` message and initiates the contest protocol where each player is independently responsible for checking the liveness and integrity of `P2`. This is useful for two cases:

1. `P2` is dead. We define dead as it lost connections with a majority of the players.

2. `P2` is acting maliciously by refusing to reply, or playing an illegal move to halt the game.

Each player `P` verifies that `P2` is alive and requests `P2`'s response to the chain `C` along with `P2`'s signature of the resulting chain (whether it's an `OK` or a `CONTEST`). The signature is necessary to ensure Non-Repudiation. Then if `P` is content with `P2`'s answer, they can `COUNTER P1`'s contest and forward `P2`'s answer, along with both `P` and `P2`'s signature to `P1`. If `P1` happened to simply drop its connection with `P2`, then `P1` can use any one of the signed responses that it received through another player on behalf of `P2`, and the game continues. Figure 4 shows an example where `P2` dropped connections with 3 players, but con still participate in the game.

**`P2` is malicious**

If a player `p` does *not* receive a response from `p` or receives a malicious response, then they can `OK P1`'s `CONTEST`. Once a majority of players `OK` the `CONTEST`, `P1` can send a `COMMIT CONTEST` to all players with the `OK`s and signatures and `P2` is kicked out of the game for all players. Thus `P2` is incentivized to respond honestly so they don't get kicked out.

**`P1` is malicious**

If `P1` is malicious and wants to kick `P2` out, they would *not* be able to do that unless a majority of the players agree, otherwise the `COMMIT CONTEST` message will not have an effect. Note that `P1` cannot grandiosely contest a majority of the players are offline to override a player's vote on another `CONTEST` since we require that a majority of the players are alive and honest.

Contesting is the true power of the P2P system setup as each player is able to independently verify `P2`'s status and no player, not even the leader, can maliciously claim a player is no longer in the game. While a player cannot trust another player, they can trust the majority.

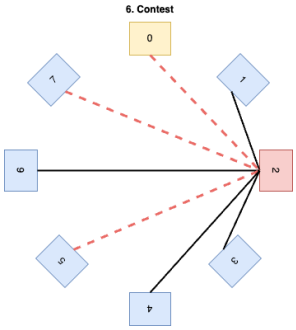


Figure 4: Player 2's links during a CONTEST. A Solid black line represents a live link and a dashed red line is a dead connection. The game is able to continue.

## 4 Implementation

We implemented Ruban in Python 3 as a static library of abstract classes that a user can implement by supplying the underlying session protocol. We provide an example using the open-source Python-P2P-Network [3]. Figure 5 shows the class design. Each class implements a certain functionality as follows:

### Trader

Abstracts the 2PC layer providing a clean interface to make new offers and respond to them. It keeps track of active transactions and relies on setting a `prev` hash in counter messages to replace and clean up offers. It exposes:

`offer(chain)`: Propose a new chain to all participants

`accept(chain)`: Accept a chain that was previously passed to `respond()` to accept the chain

`reject(old_chain, new_chain)`: Reject a chain that was previously passed to `respond()` to reject the chain. If `new_chain` is provided, a counter is sent.

`contest(pid)`: Contest `pid`

It also requires implementing the following abstract method:

`respond(chain)`: Called when a cohort receives a new proposed chain, or when a leader receives counters

### deCoordinated

Deals with the setup and maps `PIDs` to connections.

`setup(host_conn)`: If host, listen for new connections, else connect to host (`host_conn`)

`host_begin_round_robin()`: Send connection info to cohorts as a 2PC transaction to begin p2p connections.

`recv(pid, message)`: receive `message` from `pid`.

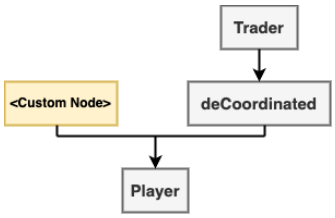


Figure 5: Class Design. Arrows represent an *is-a relationship* where the back of an arrow is an interface for what is being pointed to.

`on_new_connection(conn)`: new connection `conn`.

It also requires implementing the following abstract methods:

`send(pid, message)`: Send `message` to `pid`.

`connect(conn)`: Initiate connection to `conn`.

`listen()`: Listens for incoming connections.

### Player

Exposes turn-based game functionality:

`play(chain)`: play a chain by proposing it.

`end_turn()`: end player turn

### <Custom Node>

To be implemented by the user as the session and transport layer. Must reliably deliver messages and implement the `send, connect, listen` abstract methods of deCoordinated.

## 5 Discussion

The goal of the system is to provide security among a set of mutually distrustful players. Although players cannot trust any one single player, they are able to trust that the majority of players are honest and truthful. This allows players to collectively prevent any malicious player from both not playing according to the rules, or acting maliciously to kick another player out. However, the system does *not* address the case where a majority of players are either malicious or dead. In this case the system should hard-reset the game and start over.

There are a few additional limitations that we plan on addressing in the future like the setup and recovery from failure.

### 5.1 Setup

The current setup system is vulnerable to a malicious host. A malicious host for example, can share incorrect public keys and connection information during the round-robin connection step and impersonate other players. Although the players

need only trust a single player, the host, this is a huge limitation and calls for a secure public key infrastructure in place. Adding a server or trusted authority greatly limits the potential of the system as it would require connection to a central authority which means remote gameplay would *not* be possible, but it is potentially worth extending the setup so that a varity of solutions are possible. Another idea that is potentially worth exploring is setting up a model similar to Bitcoin where the addresses are based on public keys. We're not quite sure how to do this yet, but it is a work in progress. This would also allow the network to follow a more ad-hoc model where players can connect to any of the players already connected on the network to join the game, instead of a designated host.

### 5.2 Resilience To Crashes

Another problem that is incredibly important is what happens if a player crashes. If a player crashes when they are not a *leader* in a round (i.e. it's not their turn), the leader is able to contest the player and they would be removed from the game as long as the majority of players are honest. One potentally useful feature is to add a ystem in place to allow removed players to petition to re-join the game upon resetting. This could follow entail connecting to any of the players in the game (or all of them) and petitioning to join until a leader sends proposes an action to add the player back in the game. However, as the player is no longer in the game, there is *no* way for them to trust that any one leader would propose adding them back.

Another problem that is no stranger to Two Phase Commit is the leader failure. Currently we only allow leaders to contest other players, but in the future we plan on allowing any one to send contest messages. This, however, introduces new consistency problems against both malicious and honest players. The immediate plan is to allow players to contest the players before them out of turn and recursively build back to the last alive player.

## 6 Future Work

1. Consider the advantages and limitations of using an optimization of the 2PC protocol (either presume Commit or presume Abort)

2. Consider adding a flag for a majority consensus on the chosen counter instead of having the leader choose one. We don't see immediate applications for this, but it could be interesting.

3. Currently the system uses a cooperative mean to clean-up and abort transactions that have been countered or rejected where sendes set a `prev` field to a hash of the previous chain that this was built on. Each player confirms that this is in fact a valid counter chain of the old one and can then remove it so as to not exhaust memory. This introduces some vulnerabilities that need further studying as well as a great opportunity for a distributed garbage-collector that provides atomicity and consistency.

4. If a player drops their connection with a leader, they are able to remain in the game as the leader sends a contest which they reply to. On every subsequent round, as long as the p2p connection is dead, this contest needs to happen. This greatly amplifies traffic. We're considering allowing leaders to infer a dropped connection and upon initiating a contest and receiving back the responses, they can request one or more players that still have an up connection and forwarded a reply to serve as proxies for the player moving forward until the connection is reestablished. The signature scheme employed allows for this proxy to have authenticity and integrity.

5. The motivation for this project came as a step in building a Yu-Gi-Oh dueling disk (hence the player/game paradigm). The disk is a current WIP and we're really excited about it :)

## 7 Conclusion

We drew inspiration from the classic 2PC Protocol to introduce a distributed endorsement system that provides authenticity, integrity and non-repudiation. The characteristics of turn-based system provide a unique opportunity for simplifying many consistency concerns common to consensus protocols and allow for a straightforward implementation of 2PC without leader election concerns. Moreover, the P2P nature of the system affords an interesting contest protocol that allows the game to be resilient to maliciousness as long as the majority of players are honest and alive.

## References


[1] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*, pages 54–70, 2015.

[2] Butler Lampson and David Lomet. A new presumed commit optimization for two phase commit. In *19th International Conference on Very Large Data Bases (VLDB'93)*, pages 630–640, 1993.

[3] Maurice Snoeren. python-p2p-network. https://github.com/charlespwd/project-title.