The Rai Protocol (Draft)

Rui Morais

June 2025

1 Preliminaries

1.1 The Block-lattice

The block-lattice is a Directed Acyclic Graph (DAG) that serves as the underlying data structure for the system. It has the following properties:

- It consists of multiple independent blockchains, called account-chains, where each chain corresponds to a unique user account.
- Only the cryptographic owner of an account can append new blocks to their corresponding blockchain. Each block represents a single transaction from the owner's account to a receiver's account.
- Conflicting attempts to modify the same blockchain, known as forks, can lead to a double-spend if not properly resolved by the consensus protocol.

1.2 System Model

We assume a standard Byzantine Fault Tolerant (BFT) system model:

- The system consists of n nodes, among which a total of $3f+1$ voting weight is distributed in each epoch. Up to f of this voting weight may belong to Byzantine nodes, which can behave arbitrarily. The remaining voting weight belongs to correct nodes that follow the protocol. The distribution of voting weight can change between epochs.
- Nodes communicate over a *partially synchronous* network. This means that messages between any two correct nodes are guaranteed to be eventually delivered, but there is no known upper bound on message latency.

1.3 Goals

The Rai protocol is designed to implement Equivocation Tolerant Atomic Broadcast for state changes on the block-lattice. This abstraction provides the following guarantees:

- **Agreement.** If a correct node confirms a transaction, no other correct node confirms a conflicting transaction for the same account state.
- **Termination.** If a correct node proposes a transaction, then every correct node eventually confirms it.
- **Total Order.** If a correct node orders transaction t before transaction t', then no correct node orders t' before t.
- **Equivocation Tolerance.** If a Byzantine node proposes multiple conflicting transactions in an epoch e, then either all correct nodes confirm one of those transactions, or the node is able to propose again a transaction without conflicts and confirm it in a subsequent epoch e', $e' > e$.

2 The Rai Protocol

The Rai protocol provides fast agreement on the state of individual account-chains and reaches agreement on the state of the block-lattice from time to time. These two processes are run independently by having two types of elections:

- **Confirmation Elections**: A fast, leaderless agreement mechanism designed for low-latency confirmation of individual transactions. Each election is tied to a specific consensus instance i and an epoch e. While efficient, this mechanism does not guarantee termination in the presence of forks.
- **Ordering Elections**: A robust, epoch-based consensus mechanism that establishes a global order for a batch of confirmed transactions. This process also serves to define new epochs and definitively resolve any Confirmation Elections that failed to terminate, thereby providing the system's overall liveness.

2.1 Initialization

Algorithm 1 specifies the constants and variables required by each node p. The constant $\mathsf{THRESHOLD}$ is a system parameter that balances efficiency and latency: a lower value triggers Ordering Elections more frequently, resolving forks faster at the cost of increased overhead, while a higher value reduces overhead but may delay global ordering. The variables are categorized as follows:

- **Internal State**: $epoch_p$ tracks the node's current epoch, and $committed_p[]$ counts the number of committed transactions per epoch to trigger Ordering Elections.
- **Confirmation Elections State**: $votes_p[]$, $commits_p[]$, $confProposals_p[]$, and $forks_p[]$ store the state related to the agreement process for individual transactions.

- **Ordering Elections State**: $ordProposals_p[]$ and $orderings_p[]$ store the state for the epoch-level global ordering process.
- **Ledger State**: $confirms_p[]$ stores the final, confirmed transactions that are part of the ledger.

Algorithm 1 Initialization

```
Constants:
    THRESHOLD
Variables:
    $epoch_p := 0$
    $votes_p[] := nil$
    $commits_p[] := nil$
    $confirms_p[] := nil$
    $confProposals_p[] := nil$
    $ordProposals_p[] := nil$
    $committed_p[] := 0$
    $orderings_p[] := nil$
    $forks_p[] := nil$
```

2.2 Confirmation Elections

Confirmation Elections (Algorithm 2) are the protocol's fast path, designed to rapidly achieve agreement on non-contentious transactions. Each new transaction proposal initiates an election for its corresponding instance. The primary goal is to satisfy the Agreement property quickly for a single transaction.

The process for a node p in an election for instance i is as follows:

- **Proposal**: An account owner creates a transaction v for instance i and initiates the election by calling StartConfirmationElection(i, v), which broadcasts a $\langle \mathsf{CONF}, i, v\rangle$ message (line 2).
- **Voting**: Upon receiving a valid $\langle \mathsf{CONF}, i, v\rangle$ for an instance it has not seen before, a node stores the proposal, associates it with the current epoch $epoch_p$, and broadcasts a $\langle \mathsf{VOTE}, epoch_p, i, v\rangle$ message (line 8). If the node later receives a conflicting proposal for the same instance, it flags that instance as a fork (line 11).
- **Optimistic Voting**: To accelerate agreement, a node that receives $f + 1$ $\langle \mathsf{VOTE}\rangle$ messages for a proposal it has not yet seen can optimistically vote for it (line 16). This is safe because a quorum of $f + 1$ votes guarantees that at least one correct node has validated and voted for the original proposal.
- **Committing**: A node commits to a value v for instance i and epoch e once it observes a quorum of support (line 20) and no fork has been seen (line 23). This quorum is defined as either $2f+1$ $\langle \mathsf{VOTE}\rangle$ messages or $f+1$ $\langle \mathsf{COMMIT}, e, i, v\rangle$ messages for the same value (v, e, i). Once committed, the node broadcasts a $\langle \mathsf{COMMIT}\rangle$ message to signal its decision.

- **Ordering Election Trigger**: After committing a transaction in epoch e, the node increments its commit counter for that epoch. If this counter reaches the THRESHOLD (line 27), the node collects all its committed transactions without forks and initiates an Ordering Election by calling StartOrderingElection (line 31). This serves as the crucial link to the global ordering mechanism.
- **Confirmation**: A transaction is considered fully confirmed when a node receives $2f + 1$ $\langle \mathsf{COMMIT}, e, i, v \rangle$ messages (line 35). At this point, the node records the transaction as final (line 38) and garbage-collects the corresponding elections.

Algorithm 2 Confirmation Elections

```
1:  Function StartConfirmationElection($i, v$) :
2:    broadcast $\langle \mathsf{CONF}, i, v \rangle$

3:  upon $\langle \mathsf{CONF}, i, v \rangle$ do
4:    if $valid(v)$ then
5:      if $confProposals_p[i, *] = nil$ then
6:        $confProposals_p[i, epoch_p] \leftarrow v$
7:        $votes_p[i, epoch_p] \leftarrow v$
8:        broadcast $\langle \mathsf{VOTE}, epoch_p, i, v \rangle$
9:    else
10:     if $confProposals_p[i, *] \neq v$ then
11:       $forks_p[i] \leftarrow true$

12: upon $f + 1$ $\langle \mathsf{VOTE}, e, i, v \rangle$ do
13:   if $confProposals_p[i, *] = nil$ then
14:     $confProposals_p[i, e] \leftarrow v$
15:     $votes_p[i, e] \leftarrow v$
16:     broadcast $\langle \mathsf{VOTE}, e, i, v \rangle$
17:   else
18:     if $confProposals_p[i, *] \neq v$ then
19:       $forks_p[i] \leftarrow true$

20: upon $2f + 1$ $\langle \mathsf{VOTE}, e, i, v \rangle$ OR $f + 1$ $\langle \mathsf{COMMIT}, e, i, v \rangle$ do
21:   if $confProposals_p[i, *] = nil$ then
22:     $confProposals_p[i, e] \leftarrow v$
23:   if $confProposals_p[i, *] = v \wedge forks_p[i] \neq true$ then
24:     $commits_p[i, e] \leftarrow v$
25:     broadcast $\langle \mathsf{COMMIT}, e, i, v \rangle$
26:     $committed_p[e] \leftarrow committed_p[e] + 1$
27:     if $committed_p[e] = \mathsf{THRESHOLD}$ then
28:       for $v' \in commits_p[i, *]$ do
29:         if $forks_p[i] = false$ then
30:           $v \leftarrow v \cup v'$
31:       StartOrderingElection($e, v$)
32:   else
33:     if $confProposals_p[i, *] \neq v$ then
34:       $forks_p[i] \leftarrow true$

35: upon $2f + 1$ $\langle \mathsf{COMMIT}, e, i, v \rangle$ do
36:   $votes_p[i, *] \leftarrow nil$
37:   $commits_p[i, *] \leftarrow nil$
38:   $confirms_p[i, e] \leftarrow v$
39:   $confProposals_p[i, *] \leftarrow nil$
```

2.3 Ordering Elections

Ordering Elections (Algorithm 3) are the protocol's synchronous backbone, providing strong termination and total order guarantees. They are triggered periodically to establish a global ordering for a batch of transactions, resolve any pending conflicts from the preceding epoch, and advance the entire network to the next epoch.

The process for an Ordering Election for epoch e is as follows:

- **Proposal**: A node initiates the process by calling $\mathsf{StartOrderingElection}(e, v)$, broadcasting an $\langle \mathsf{ORD}, e, v \rangle$ message containing the set of transactions it has committed in epoch e (line 2). To be valid, this set must only contain transactions for which the node has observed a valid commit quorum and no forks have been seen (line 4).
- **Consensus**: Upon receiving valid $\langle \mathsf{ORD}, e, v \rangle$ messages from nodes representing $2f + 1$ of the voting weight (line 6), a node invokes an underlying Byzantine consensus primitive $\mathsf{Consensus}$ [1]. This primitive ensures that all correct nodes will agree on a single, final vector of transactions for epoch e that contains all confirmed transactions by correct nodes until then (line 7).
- **Post-Consensus**: Once the consensus primitive decides on an ordered vector for epoch e (line 8), each correct node performs a state update:
  - **Epoch Advancement**: The node increments its local epoch counter to $e + 1$ if the new epoch is more recent than the current one (line 10), formally transitioning to the next epoch and preparing for a new round of elections.
  - **Confirmation**: The node iterates through the decided vector. For any transaction (i, v) in the vector that it has not already confirmed, it now does so (line 15). This ensures that all correct nodes converge to an identical state for all transactions included in the global order.
  - **Fork Resolution**: The node purges all state related to proposals from previous epochs (line 18). This formally terminates any lingering Confirmation Elections, including those that were forked, preventing them from carrying over.
  - **Liveness for Honest Transactions**: For any non-forked proposal from the current epoch (line 22) that was not included in the decided ordering vector, the node ensures its liveness by re-broadcasting its vote for it in the next epoch, $e + 1$ (line 27).

[1] Any Consensus algorithm can be used, as long as it guarantees Agreement and Termination.

Algorithm 3 Ordering Elections

```
Function StartOrderingElection($e, v$) :
    broadcast ⟨ORD, $e, v$⟩

upon ⟨ORD, $e, v$⟩ do
    if $valid(v)$ then
        $ordProposals_p[e] \leftarrow ordProposals_p[e] \cup v$
        if $|ordProposals_p[e]| = 2f + 1$ then
            $orderings_p[e] \leftarrow$ Consensus($e, v$)

upon $orderings_p[e] \neq nil$ do
    if $epoch_p \leq e$ then
        $epoch_p \leftarrow e + 1$
    for $(i, v) \in orderings[e]$ do
        if $confirms[i, e] \neq nil$ then
            $votes_p[i, *] \leftarrow nil$
            $commits_p[i, *] \leftarrow nil$
            $confirms[i, e] \leftarrow v$
            $confProposals_p[i, *] \leftarrow nil$
    for $(i, e') \in confProposals_p[]$ do
        if $e' \leq e - 1$ then
            $confProposals_p[i, e'] \leftarrow nil$
            $votes_p[i, e'] \leftarrow nil$
            $commits_p[i, e'] \leftarrow nil$
        if $e' = e$ then
            if $forks_p[i] = false$ then
                $v \leftarrow confProposals_p[i, e]$
                $confProposals_p[i, e + 1] \leftarrow v$
                $votes_p[i, e + 1] \leftarrow v$
                broadcast ⟨VOTE, $e + 1, i, v$⟩
```

2.4 Security Proofs

Lemma 1. *For all $f \geq 0$, any two sets of nodes with voting power at least equal to $2f + 1$ have an intersection containing at least $f + 1$ voting power, which implies the intersection contains at least one correct node.*

Proof. As the total voting power is equal to $n = 3f + 1$, we have $2(2f + 1) = n + f + 1$. This means that the intersection of two sets with the voting power equal to $2f + 1$ contains at least $f + 1$ voting power in common, *i.e.*, at least one correct node (as the total voting power of faulty nodes is f). The result follows directly from this. $\square$

Lemma 2. *The Rai protocol satisfies the Agreement property of Equivocation Tolerant Atomic Broadcast.*

Proof. Suppose a value v is confirmed in a Confirmation Election of epoch e. Assuming that all correct nodes will see at least one vote, commit or the proposal of v while in epoch e, they will not commit a conflicting value v' in epoch e' ($e' > e$), because $forks_p[i]$ will be set to true, thus v' will not be confirmed due to Lemma 1. If $e' < e$, there is not enough voting weight on epoch e' to reach quorum.

The only way to commit v' is if the commit of v is cleaned up, but that is not possible because transaction v confirmed by a correct node in epoch e in a Confirmation Election is guaranteed to be included in the Ordering Election of epoch e or $e{+}1$. This is because all correct nodes propose their committed values without forks. By Lemma 1, any winning proposal from the Consensus primitive must have a non-empty intersection with the set of correct nodes, ensuring that a committed transaction by at least $2f+1$ voting weight is included. This is the same reason why v' cannot be confirmed in an Ordering Election.

□

Lemma 3. *Rai satisfies the Termination property of Equivocation Tolerant Atomic Broadcast.*

Proof. Consider a valid transaction t for instance i proposed by a correct node. The node broadcasts a $\langle\mathsf{CONF}\rangle$ message. t is guaranteed to be eventually confirmed through one of the following paths:

- **Confirmation Election**: If the network is responsive, t will quickly gather $2f+1$ votes and then $2f+1$ commits within its epoch, leading to its confirmation.

- **Ordering Election**: Assuming that new valid transactions are continuously proposed, all correct nodes will eventually start the Ordering Election for epoch e. If t is committed but not fully confirmed, it will be included in its proposer's submission to the next Ordering Election. Since all correct nodes eventually participate in this election and the underlying Consensus guarantees termination, an ordering will be decided. If t is included in the final vector, all correct nodes will confirm it.

- **Carry-Forward Mechanism**: If t is not forked but fails to be included in the ordering for its epoch, every correct node that voted for it will rebroadcast its vote in the next epoch, $e+1$. This gives t a new chance for confirmation in each subsequent epoch.

Since the Ordering Election process guarantees the system always progresses to a new epoch, a valid, non-forked transaction cannot be stalled indefinitely and will eventually be confirmed. □

Lemma 4. *Rai satisfies the Total Order property of Equivocation Tolerant Atomic Broadcast.*

Proof. Ordering Elections use a consensus primitive to produce a single, decided vector of transactions for each epoch e. The Agreement property of this primitive ensures all correct nodes decide on the exact same vector. The global total order is then established by processing all transactions from the vector of epoch 0, followed by epoch 1, epoch 2, and so on. Within each epoch's vector, a deterministic rule (e.g., lexical order of transaction hashes) ensures all nodes process transactions in the same sequence. This creates a single, unambiguous global history. □

Lemma 5. *Rai satisfies the Equivocation Tolerance property of Equivocation Tolerant Atomic Broadcast.*

Proof. Suppose a Byzantine node equivocates by proposing conflicting transactions for the same instance i. A correct node that has already received one proposal for i will detect the conflict upon receiving a second, different proposal. It will then mark instance i as a fork. Two outcomes are possible:

1. One of the conflicting versions gathers enough support to be confirmed by a correct node in epoch e.
2. Neither conflicting version gets confirmed. The instance remains in a forked state. When the Ordering Election for epoch $e + 1$ concludes, the undecided proposal is not carried forward to the next epoch. The stale election is terminated during cleanup and a non forked transaction can be re-submitted.

□

Theorem 1. *Rai satisfies Equivocation Tolerant Atomic Broadcast.*

Proof. It follows directly from satisfying its constituent properties: Agreement (Lemma 2), Termination (Lemma 3), Total Order (Lemma 4), and Equivocation Tolerance (Lemma 5). □

Exception to Lemma 2. Suppose value v is committed and confirmed in epoch e after the Ordering Election of epoch e has already started. This means that v will not be included in the Ordering Election and that v has received at least $2f + 1$ commits, let us call this quorum Q. And let us call the remaining f honest voting weight A.

Now suppose that f of Q is byzantine and that in epoch $e+1$ there is a shift of honest voting weight of at least 1 from the $f + 1$ honest voting weight of Q to the other f honest voting weight that is not part of Q.

If there is a conflicting value v', in which the byzantine nodes also vote (double vote) and A has not seen the proposal, nor at least $f + 1$ votes, nor at least $f + 1$ commits), A can vote for v' and even commit it, which can lead to a double spend.

The byzantine weight needed for this to happen is less the more the honest voting weight shifts from the honest voting weight of Q to A.