Core Peras Design and Architecture

Modus Create Peras Team
March 2, 2025

Contents

Chapter 1

Introduction

This document describes an architecture for implementing the pre-alpha version of the Ouroboros Peras protocol for the Cardano blockchain. Ouroboros Peras is a novel protocol that enables fast settlement under optimistic conditions.

For a description of the problem of settlement/finality on Cardano, as well as possible use cases, we recommend reading to the corresponding Cardano Problem Statement.

`https://github.com/cardano-foundation/CIPs/blob/master/CPS-0017/README.md`

For a description of the Peras protocol, its high-level dynamics and a formal specification, we refer to the Peras Cardano Improvement Proposal and the Peras website.

`https://github.com/cardano-foundation/CIPs/blob/master/CIP-0140/README.md`
`https://peras.cardano-scaling.org`

This document is based on the work of the Peras Innovation team that created the above resources.

Overview. We reify the high-level description of the protocol by showing how Peras can be integrated into the existing node architecture, while reasoning about lower-level security properties such as bounded resource usage under adversarial activity. We proceed by giving an outline implementation plan for `cardano-node`, the Haskell implementation, and define the tests necessary to validate the implementation. Finally, we portray the interactions of Peras with other Cardano features (Mithril and Ouroboros Leios), and list the identified risks and opportunities.

Acknowledgements. We would like to thank members of the Peras research team, the Peras and Leios Innovation teams, the IOE Network and Consensus teams, the IOG Mithril team and the Intersect Technical Steering Committee for fruitful discussions.

Latest version and feedback. The latest version of this document is available at

`https://tweag.github.io/cardano-peras/peras-design.pdf`

For asking questions and reporting feedback, as well as the source code of this document, see the GitHub repository at

`https://github.com/tweag/cardano-peras`

Chapter 2

Architecture

Figure 2.1 gives a graphical overview of the data flow between the components involved in the architectural changes of this document. To keep the diagram simple, not all dependencies are indicated; for example, many components require access to a recent ledger state in order to validate votes or certificates or to decide whether to cast a vote. We elide all existing components that do we do not expect to modify non-trivially.

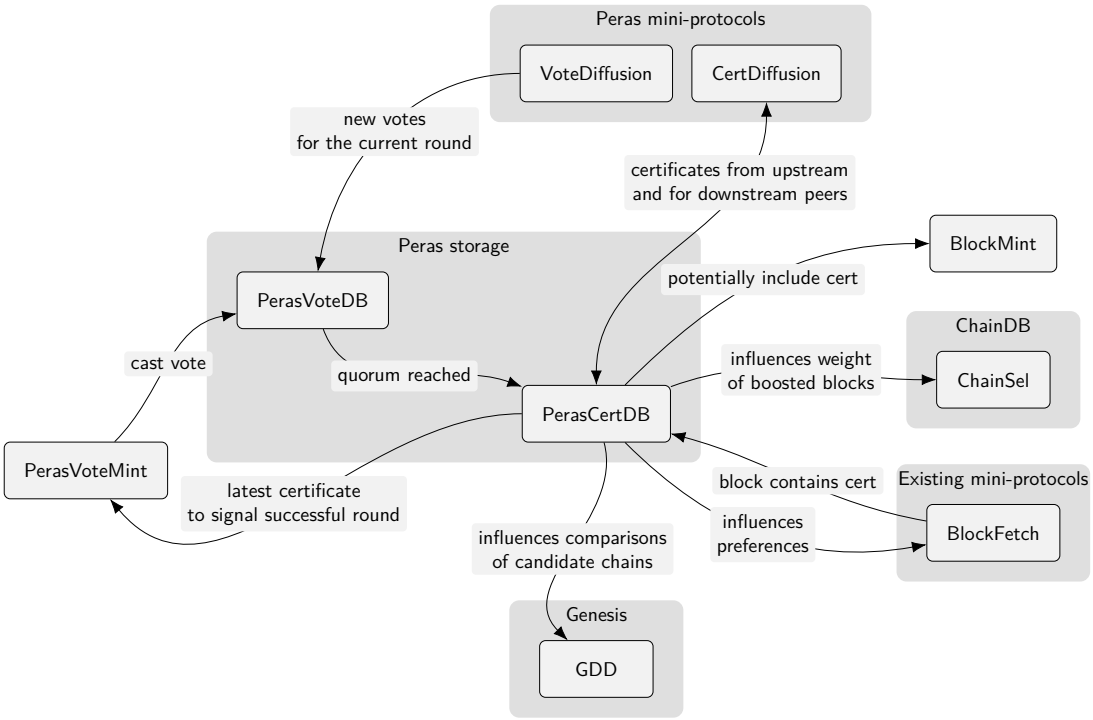

FIGURE 2.1: *Overview of the data flow between the components of this document*

2.1 Protocol parameters

For context, we list existing (Praos/Genesis) protocol parameters in table 2.1.

The Peras protocol is influenced by various new protocol parameters, also see [Bai+25], which we list in table 2.2. The values of these parameters are not yet final, and justifying their values (especially the ones related to cooldowns) is out of scope for this document. For explicitness and to avoid confusion, we use a relatively verbose naming scheme. The Cardano Ledger allows protocol parameters to be controlled via on-chain governance. We indicate whether this flexibility is likely to be useful.

`perasRoundSlots` Peras round length, number of slots per Peras round.

Description	Unit	Symbol	Mainnet value
Active slot coefficient	slot^{-1}	`asc`	$1/20$
Security parameter for chain growth	block	`kcp`	2160
Chain growth window size to achieve common prefix	slot	$T_{\texttt{cp}}$	$3 \cdot \texttt{kcp}/\texttt{asc} = 129\,600$
Chain quality window size to guarantee at least one honest block	slot	$T_{\texttt{cq}}$	$\texttt{kcp}/\texttt{asc} = 43\,200$
Genesis window	slot	$s_{\texttt{gen}}$	$T_{\texttt{cp}}$

TABLE 2.1: *Existing Praos/Genesis protocol parameters*

Name	Unit	Symbol	Feasible value	Governable?
`perasRoundSlots`	slot	U	90	✓
`perasBlockMinSlots`	slot	L	30 to 90	✓
`perasBlockMaxSlots`	slot	n.a.	$T_{\text{CQ}} = \texttt{kcp}/\texttt{asc}$	✓
`perasHealingSlots`	slot	$T_{\texttt{heal}}$	t.b.d.	✘
`perasCertMaxSlots`	slot	A	$T_{\texttt{heal}} + T_{\texttt{cq}}$	✘
`perasIgnoranceRounds`	round	R	$\lceil A/U \rceil$	✘
`perasCooldownRounds`	round	K	$\lceil (A + T_{\texttt{cp}})/U \rceil + 1$	✘
`perasBoost`	block	B	15	✓
`perasQuorum`	weight	τ	$3/4$	✘
`perasN`	committee seat	N	500 to 1000	✓
`perasVoteSizeLimit`	B	n.a.	200 B	✓
`perasCertSizeLimit`	B	n.a.	20 kB	✓

TABLE 2.2: *New Peras protocol parameters*

`perasBlockMinSlots` Peras block selection offset, the minimum age (in slots) of a block to be voted on at the beginning of a Peras round.

Note that the rather small values of this parameter allow relatively weak adversary to execute "vote splitting attacks", see appendix A.1.2 for more details.

`perasBlockMaxSlots` The maximum age (in slots) of a block to be voted on at the beginning of a Peras round.

This parameter is new compared to the CIP [Bai+25]. It is motivated to allow validating votes/certificates ahead of the current chain, especially while syncing.[1]

`perasHealingSlots` The amount of slots needed to heal from a failed Peras voting round, depending on `perasBoost`. A concrete value has yet to be chosen.

`perasCertMaxSlots` The maximum age of a certificate to be included in a block.

[1]We note that the exact details of forecasting are still subject to discussions, in particular pending feedback by the Peras research team, cf. `https://github.com/tweag/cardano-peras/issues/25`.

`perasIgnoranceRounds`, `perasCooldownRounds` Lengths of the chain ignorance period and the cooldown period. Determine for how long honest nodes will not vote after an unsuccessful Peras round.

`perasBoost` The extra chain weight that a certificate gives to a block.

`perasQuorum` The total weight of votes required to create a certificate, assuming that the total (expected) weight of a committe is 1.

`perasN` The expected number of committee seats, i.e. the total number of votes that can be cast per round.

`perasVoteSizeLimit`, `perasCertSizeLimit` Maximum size (in bytes) of Peras votes and certificates.

`kcp` The security parameter (reinterpreted to measure weight of chains instead of just blocks) needs to be increased to retain the same security as in pure Praos.

For a more detailed discussion of the interactions of the various parameters we refer to [Bai+25].

2.2 Votes and certificates

Peras introduces two new entities: *votes* and *certificates*.

2.2.1 Votes

Peras voting proceeds in rounds, lasting `perasRoundSlots` each. Every round has an associated *weighted committee* [GKR23, Section 4], a set of stake pools responsible for voting in this round, together with individual amounts of *voting power*, i.e. their *weight* in the committee. The exact committee election scheme has not yet been decided, but it must satisfy the following informal properties.

a) An adversary with total stake α can not get significantly more than α relative weight in any committee. The (expected) weight of a pool in the committee is proportional to its stake.

b) The (expected) size of the committee can be controlled via the `perasN` parameter.

At the start of every Peras round (lasting `perasRoundSlots`), all of the honest nodes in the corresponding Peras committee will usually vote for a recent block (otherwise, Peras is likely entering a cooldown period). Votes are *weighted* according to the weight of the pool that cast them in the committee, which allows for an improved size-security tradeoff compared to unweighted votes (where the same party might be allowed to vote multiple times), cf. [GKR23]. Under optimistic conditions, votes of total weight of at least `perasQuorum` are cast for the same block, in which case the round is *successful*.

A vote contains the following data:

- Round number.
- Point of the block that is being voted for.
- Stake pool identifier.
- Associated cryptographic material, i.e. a signature and potentially an eligibility proof.

The weight of a vote is implicit here and can be recomputed upon validation.

A vote from round r (starting in slot s) for a block in slot t is *valid* under the following conditions.

- The cryptographic material must be valid, ensuring that the vote was indeed cast by an eligible pool.

  This requires the stake distribution, certain protocol parameters and the epoch nonce. The natural choice is to use the same data that would be used to validate a header for slot s.

- $s - \texttt{perasBlockMaxSlots} \leq t < s - \texttt{perasBlockMinSlots}$.

- The slots s and t are in the same era, or at least, the eras of s and t both support Peras.

2.2.2 Certificates

Once a node observes votes of the same round for the same block with total weight of at least `perasQuorum`, then it will create (via *aggregation*) a corresponding (succinct) *certificate* proving this fact.

A certificate contains the following data:

- Round number.
- Point of the block that is being voted for.
- Associated cryptographic material certifying that there are votes of weight `perasQuorum` voting for the aforementioned block in the given round.

Validity of certificates is analogous to that of votes.

We note that the cryptographic material may vary depending on which votes are aggregated. Such certificates are only artificially different and should be treated interchangeably.

On the other hand, we assume that *equivocating* certificates, i.e. two certificates in the same round for *different* blocks, do not occur.

Assumption

For every Peras round, there is at most one block that has a valid certificate in that round.

This assumption is justified by an appropriate parameterization of `perasQuorum` and `perasN`, such that an adversary never attains sufficient weight in the committee to equivocate certificates.[2]

2.2.3 Realizing votes and certificates

The underlying cryptography used to realized votes and certificates is out-of-scope for this document, we refer to [Bus25] for more details. Here, we only summarize a few key metrics in fig. 2.2, while noting that these are still subject to change[3]; however, we do not expect them to get significantly worse.

The existing formal specification in Agda for the Consensus and Ledger layer [Día25; Kni+25] can be naturally extended with the details of vote and certificate validation, in particular for conformance testing.

[2] Future work on the Peras protocol might relax this assumption and hence allow a less conservative parameterization. However, certain design decisions (especially around certificate diffusion) in this document would need to be reconsidered to properly handle equivocating certificates, as we decided not to prematurely complicate the design without a clear picture of the required semantics.

[3] In particular, certificates might get significantly smaller ($\leq$ 1 kB).

	Votes	Certificates
Size	90 B to 164 B	7 kB to 10 kB
Generation	280 µs	63 ms to 93 ms
Verification	1.4 ms	115 ms to 157 ms

FIGURE 2.2: *Preliminary metrics for votes and certificates*

2.2.4 Handling votes and certificates from the future

A vote or certificate is *from the future* if its round has not yet begun according to the local wallclock. We suggest to handle such votes just like *headers* from the future are already being handled:

- Upon receiving a vote or certificate x from the *near* future (which is currently defined to be at most 2 s in the future), an artifical delay is introduced such that x is no longer from the future, after which x is processed further.

  This rule is motivated by the insight that it would be unreasonable to expect clocks between honest nodes across the world to be perfectly synchronized.

  The artifical delay might not be necessary for Peras votes/certificates as there is, in contrast to headers/blocks, no incentive to diffuse your vote as early as possible (as long as it still arrives within a Peras round). However, using the exact same logic as for headers has a conceptual simplicity.

- Upon receiving a vote or certificate from the *far* future (defined to be more than 2 s in the future), we immediately disconnect from the sending peer as this constitutes adversarial behavior.

2.2.5 Handling votes and certificates from the past

The formal specification of Peras in [Bai+25] allows to diffuse votes (and hence processes the resulting certificates) that lie far arbitrarily in the past. In an actual implementation, this is undesirable, as the node does not maintain the necessary state (e.g. arbitrarily old stake distributions) in order to even be able to validate such votes/certificates. We now describe an approach for resolving this in an actual implementation.

Distant past If a caught-up node observes a vote or certificate for the first time while its round is already from the distant past (i.e. more than T_{cp} slots old), it can safely ignore/discard it, as such votes and certificates can only affect (and hence further strengthen compared to alternative chains) the common prefix of the honest nodes, i.e. the already-immutable part of their selection.

A consequence of this rule is that two honest caught-up nodes can differ harmlessly in whether they have seen a certificate for a historical round, namely when an adversary diffuses certain votes/certificates very late, such that some honest nodes still store a particular certificate for an old (but still just young enough) round, but other nodes ignore it as it is already (barely) too old.

Near past In contrast, we handle votes and certificates from the near (i.e., not distant) past by disallowing votes from past rounds (with some tolerance to account for an acceptable clock skew), but still allowing to diffuse such certificates.

This is justified by the fact that honestly cast and diffused votes have ample time to be diffused within a Peras round. On the other hand, an adversary can delay their votes for some round r arbitrarily, for example such that some honest nodes consider them to have arrived on time for round r and hence observe a quorum, while other honest nodes do not. However, despite not receiving these votes, the nodes in the latter category still quickly learn of the quorum in round r as the certificate is still diffused separately.

This approach is compatible in behavior with the formal specification, and directly bounds the amount of vote diffusion work that an honest client has to perform per round (namely, to download the votes of that round). In particular, it prevents attackers from releasing lots of votes from past rounds in a burst.

2.2.6 Monotonicity of honestly observed certificates

For future reference, we observe that an honest nodes observes Peras certificates in *almost* monotonically increasing order of round numbers in the sense below.

If an honest caught-up node observes a certificate for round r but has not yet done so for round $r-1$, this means that

- either a cooldown just ended, during which no honest nodes voted and so no certificate was created for many rounds, so the node has definitely seen all certificates for rounds smaller than r,
- or other honest nodes voted in round r due to them observing a certificate for round $r-1$ at that point, so it is only a matter of time until the node also receives that certificate (which we can assume to happen before round $r+1$ starts, as `perasRoundSlots` $\gg \Delta$, where Δ is the maximum network delay).

  An adversary with sufficient stake $\alpha > 1 -$ `perasQuorum` can cause this scenario by withholding its votes during round $r-1$ until closely before the start of round r, and then diffusing them only to a subset of the honest nodes in time before round r. Also see appendix A.2.1.

In both cases, the sequence $(r_k)_{k\in\mathbb{N}}$ of round numbers of certificates observed by an honest node increases *almost* monotonically, i.e. $\max_{i<k} r_k \leq r_{k+1} - 1$ for all $k \in \mathbb{N}$.

2.3 Changes to the Ledger

The Cardano Ledger [CVG23; Kni+25] needs minor modifications to accomodate Peras.

Peras requires new protocol parameters, see section 2.1, which should be a routine change.

Apart from that, the Ledger needs to be modified to account for Peras certificates which are included on chain to coordinate the end of a cooldown period. We note that certificate are morally part of the Consensus layer (in particular, validating them requires the epoch nonce, which no other ledger rule does); however, certificates are likely too large to be stored in a header.

Concretely, we propose the following checks for a block body B in round r containing a certificate `cert`:

a) The certificate `cert` must be valid.

b) The round of `cert` must be strictly greater than the round of the previous certificate on chain, and it must lie between $r -$ `perasCertMaxSlots` and r, i.e.

$$r - \texttt{perasCertMaxSlots} \leq \text{round}(\texttt{cert}) \leq r \ .$$

This ensures that `cert` is neither too old nor too new, such that the ledger state that B is validated against contains the necessary information to validate r.

c) The size of the certificate must be bounded. Concretely, we propose a protocol parameter `perasCertSizeLimit` for the size of a certificate, and additionally, we let the size of the certificate count towards the maximum block body size.

The only change to the ledger state is the addition of the round number of the last certificate on chain, and the stake distribution for the previous epoch in case `cert` is in a previous epoch compared to B.[4]

2.3.1 Analysis

Note that these rules allow the adversary to needlessly include certificates in chain even if there is no current/upcoming cooldown period. This does not impact the purpose of on-chain certificates: If the system does enter a cooldown (via honest nodes stopping to vote), the adversary will run out of certificates to include on chain due to the monotonicity property of round numbers.

Moreover, this monotonicity property also ensures that the adversary can (on average) only include one certificate per Peras round.

- This means that, on average (assuming no cooldown periods), an attacker with sufficient stake to be elected at least once per round on average[5] can include

$$\frac{\texttt{perasCertSizeLimit}}{\texttt{perasRoundSlots}}\,\mathrm{B/slot}$$

without anyone paying for the associated cost of bandwidth/storage.

For comparison, the fee for including `perasCertSizeLimit` bytes on the chain as part of a transaction is given by $\texttt{minfeeA}\cdot\texttt{perasCertSizeLimit}$. For realistic values on the conservative end (using $\texttt{minfeeA} = 44$ and $\texttt{minfeeB} = 155381$ on Cardano mainnet as of epoch 537[6], $\texttt{perasRoundSlots} = 90$ and $\texttt{perasCertSizeLimit} = 20\,\mathrm{kB}$, cf. fig. 2.2), this would correspond to a cost of

$$\frac{\texttt{minfeeA}\cdot\texttt{perasCertSizeLimit} + \texttt{minfeeB}}{\texttt{perasRoundSlots}} \approx 11\,504\,\mathrm{lovelace/slot} \approx 41.42\,\mathrm{ADA/h}$$

assuming a slot length of one second.

For context, note that implementing Peras also entails diffusing `perasN` votes per round (and the total size of the votes is significantly large than `perasCertSizeLimit`), and the bandwidth induced by this is also currently not planned to be compensated/incentivized.

- Additionally, we need to validate these uselessly included certificates. As they do not affect the ledger state, this validation can in principle be performed in parallel to the ledger rules, partially mitigating the impact.

Finally, note that otherwise-honest nodes have no incentive to include such useless certificates, in particular as they usually could instead include more transactions (which pay fees) in their blocks.

[4] The ledger state already keeps the stake distribution of the previous epoch around for its reward calculation.

[5] Concretely, this is satisfied for an adversary with stake α if $\phi(\alpha)\cdot\texttt{perasRoundSlots} \geq 1$, so for $\alpha \geq 21.8\%$ if $f = 1/20$ and $\texttt{perasRoundSlots} = 90$.

[6] See e.g. `https://cexplorer.io/params`.

2.3.2 Alternatives

If this cost is considered to be unacceptable, a drastic measure would be to require pools to pay a comparable fee when they include a certificate in a block (which is a rare thing for honest nodes). However, such a fee for pools is unprecedented, and raises various unresolved questions regarding incentives and what to do when honest pools run out of funds to pay this new fee.

Another option would be to modify the Peras protocol to reliably determine whether an honest node could have included a certificate into a particular block. However, we are not aware of an easy way to do this.

2.4 Storing historical certificates for Ouroboros Genesis

Nodes bootstrapping via Ouroboros Genesis (to minimize trust assumptions) need to be able to assess the weight of competing historical chains in order to resist adversarial *long-range attacks* [Bac+25]. Computing the weight of a chain requires the corresponding certificates indicating which blocks are boosted. See appendix A.2.2 for more details.

Even if a syncing node did not encounter any adversarial challengers (and hence did not ever need to compute weight), it still must get all certificates: Indeed, *other* peers syncing from that node (e.g., in the future) might encounter adversaries and therefore require those certificates. Briefly, there is a "duty to remember" certificates even if they are not directly useful for the caught-up node anymore.

We describe the protocol for retrieving historical certificates in section 2.6.2, the associated storage component in section 2.9, and the adjustment to comparing chains by weight instead of density in section 2.7. Furthermore, it is necessary to prevent block synchronization from outpacing certificate acquisition, as this would require retaining historical state longer than necessary for certificate validation.[7] To accomplish this, we propose to temporarily stop selecting blocks if doing so would cause the immutable tip of the chain to advance while certificate acquisition is lagging behind. This is very a mechanism very similar to the *Limit on Eagerness* in the Cardano Genesis implementation [Bac+25], which avoids permanently committing to any chain if a disagreement between competing header chains is yet to be resolved by a density comparison.

2.5 Object diffusion mini-protocol

This section presents a generic object diffusion mini-protocol. Heavily inspired by Tx-Submission [Cou+25, Section 3.9] and the related "Decentralized Message Queue" CIP [Ray+24], we hope to use it for certificate and vote diffusion. This protocol aims are synchronising a data base of objects (e.g., for Tx-Submission, their mempool) between peers:

- Clients and server maintain the knowledge of a queue of objects of which the server has knowledge and the client presumably does not. Importantly, the queue is of bounded size, and the maximal size is a hardcoded parameter of the implementation. The objects are represented by *identifiers*, typically orders of magnitude smaller than the objects themselves.
- Clients can ask their servers to provide them with an ordered list of the *next* objects that they wish to diffuse. The notion of order—and therefore of which objects are *next*—is internal to each

[7] We do not expect certificate validation to become the bottleneck of syncing, as `perasRoundSlots·asc` $= 4.5$ and 4 to 5 blocks empirically take longer to validate than a certificate. Also, we note that certificates can be trivially validated in parallel.

server; it might for instance be the order in which they themselves got aware of the objects in question.

- Clients can then, given an object identifier, request the corresponding object to the server. This is not mandatory, and, if several servers propose to diffuse the same objects, the peer might decide to download only from a subset of them. The strategy may vary depending on the objects, how promptly the client wants to get them, and how important it is for them not to miss one.
- When requesting the next batch of object identifiers, the client informs the server that they are not interested in some of the objects anymore, possibly because they have acquired them from this server or another one. They do that by *acknowledging* a number of identifiers from the beginning of the queue. They must, in doing so, ensure that the queue remains under the maximal allowed size.

Figure 2.3 gives a graphical representation of the state machine of the proposed mini-protocol, while fig. 2.4 provides a table with the states' agencies. The coming subsections give more details on the mini-protocol.

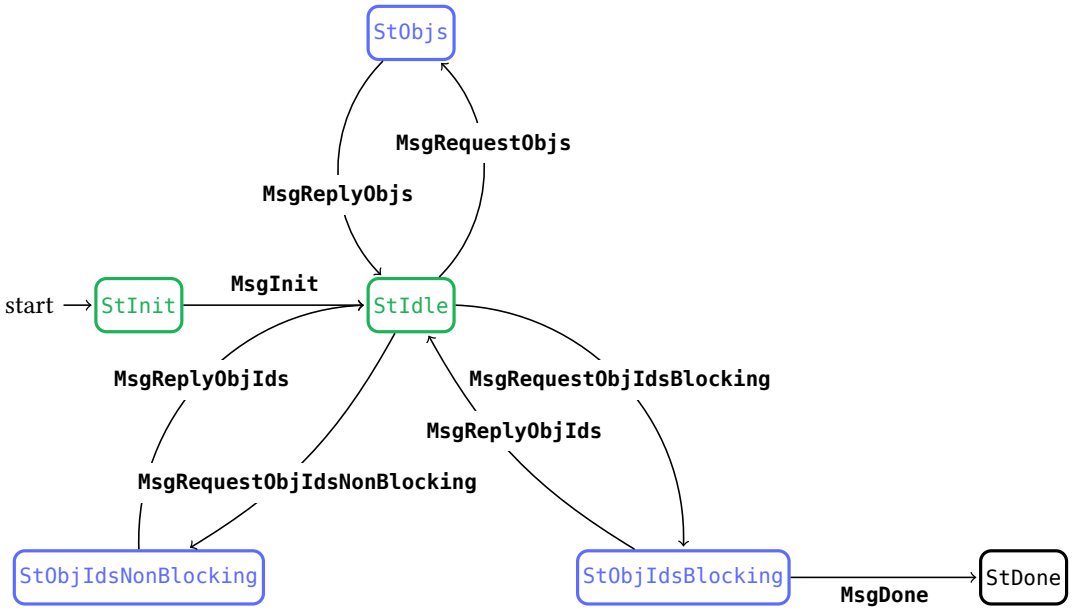

FIGURE 2.3: *Object diffusion state machine*

2.5.1 Parameters

This subsection describes the parameters of this generic protocol. They are notions that need to be made concrete for each implementation of the protocol.

object The abstract type of objects diffused by the protocol.

id Identifier that uniquely identifies an object.

state	agency
`StInit`	Client
`StIdle`	Client
`StObjIdsBlocking`	Server
`StObjIdsNonBlocking`	Server
`StObjs`	Server

FIGURE 2.4: *Object diffusion state agencies*

objectIds An opaque type returned by the server when asked for object ids. It is not explicitely a list of ***id***, because the server may want to add metadata to the ids, and/or run a compression scheme to limit the size of its response.

responseToIds A function with type ***objectIds*** $\to$ [***id***].

initialPayload An abstract payload that helps initialise the state of the server. For instance, a slot number before which the client does not care about the objects.

2.5.2 Protocol messages

`MsgInit` (***initialPayload***) Initial message of the protocol, with its payload.

`MsgRequestObjIdsNonBlocking` (***ack***, ***req***) The client asks for up to ***req*** new object ids and acknowledges ***ack*** old ids. The server immediately replies (possibly with an empty list).

`MsgRequestObjIdsBlocking` (***ack***, ***req***) The client asks for up to ***req*** new object ids and acknowledges ***ack*** old ids. The server will block until new objects are available.

`MsgReplyObjIds` (***objectIds***) The server replies with the ids of its available objects. In the blocking case, the reply is guaranteed to contain enough information to build at least one object id with ***responseToIds***. In the non-blocking case, the reply may not contain any actual data.

`MsgRequestObjs` ([***id***]) The client requests objects by sending a list of object ids. The total size of the corresponding objects MUST not be bigger than the size limit in bytes.

`MsgReplyObjs` ([***object***]) The server replies with the list of all the objects that were requested.

`MsgDone` The server terminates the mini protocol.

Table 2.3 presents the state transitions and the associated messages.

2.5.3 Client and server implementation

The protocol has two design goals, inherited from Tx-Submission [Cou+25, Section 3.9]: It must diffuse objects with high efficiency and, at the same time, it must rule out asymmetric resource attacks from a client against a server.

Typically, downstream nodes will run one instance of the client per upstream peer that they have, and upstream nodes will run one instance of the server per downstream peer that they have.

from state	message	to state
StInit	**MsgInit**	StIdle
StIdle	**MsgRequestObjIdsNonBlocking**	StObjIdsNonBlocking
StIdle	**MsgRequestObjIdsBlocking**	StObjIdsBlocking
StObjIdsNonBlocking	**MsgReplyObjIds**	StIdle
StObjIdsBlocking	**MsgReplyObjIds**	StIdle
StIdle	**MsgRequestObjs**	StObjs
StObjs	**MsgReplyObjs**	StIdle
StIdle	**MsgDone**	StDone

TABLE 2.3: *Object diffusion mini-protocol state transitions*

The protocol is based on two pull-based operations. The client can ask for a number of object ids, and it can use these object ids to request a batch of objects. The client has flexibility in the number of object ids it requests, whether to actually download the object and how it batches the download of objects. For instance, a downstream node might choose to download an object from a subset its upstream peers that shared its id.

The client can also switch between requesting object ids and downloading objects at any time. It must, however, observe several constraints that are necessary for a memory-efficient implementation of the server.

Conceptually, the server maintains a limited size FIFO of outstanding objects per client. (The actual implementation can, of course, use the data structure that works best.) The maximum FIFO size is a protocol parameter. The protocol guarantees that the client and server agree on the current size of that FIFO and on the outstanding object ids. The client can use a variety of heuristics to request object ids and objects. One possible implementation for a client is to maintain a FIFO that mirrors the server's FIFO but only contains the object ids (and the size of the objects) and not the full objects.

After the client requests new object ids, the server replies with a list of object ids and puts these objects in its FIFO. The server may reply with less ids that requested, indicating that it has no more objects after those. As part of a request, a client also acknowledges a number of objects from the beginning of the server's FIFO that it is no longer interested in. The server then removes them from its FIFO. The server checks that the size of the FIFO, *i.e.* the number of outstanding objects, never exceeds the protocol limit and aborts the connection if a request violates the limits.

The choice of a FIFO is what allows for acknowledgement by a simple integer, which is more network-efficient and allows for simple implementation. However, it somewhat constrains the order in which we request the objects. Note that we could use another structure and implementation, as long as the acknowledgement scheme still makes sense.[8]

The protocol supports blocking and non-blocking requests for new objects ids. If the FIFO is empty, the client has nothing better to do than to wait for the server to acquire more object ids; in that case it should use a blocking request, in order to avoid polling. The rest of the time, the client might prefer non-blocking requests to prevent giving agency to the server for too long a time, and being able to download objects in the meantime. The server must reply within a small timeout to a non-blocking

[8]In the case of Tx-Submission, the use of a FIFO matters because we also care about the order of transactions themselves, as they may depend upon each other.

request, possibly with an empty list. A blocking request, on the other hand, waits until at least one object is available.

The client can request any batch of objects from the current FIFO in any order. The server must respond with all the requested objects.[9]

The client must ensure that the total size of the object batch it requests does not exceed the protocol's size limits. In order to do so, the ***`objectIds`*** type should contain enough information to infer the size of each object, or the protocol should exchange objects with fixed size.[10]

2.5.3.1 Blocking vs. non-blocking ids requests

When requesting object ids, a client has the choice between emmiting a blocking or a non-blocking request. Blocking requests are kinder to the network because they avoid round-trips for nothing. However, they abandon the agency to the server, preventing the client to request objects in the meantime.

We propose the following rule to govern this decision:

If the pipeline of that specific client-server pair contains only **`MsgRequestObjs`** messages, and these messages span over all the objects in the FIFO that we have not yet obtained, then send **`MsgRequestObjIdsBlocking`**. Otherwise, send **`MsgRequestObjIdsNonBlocking`**.

Note that this criterion covers the simpler case of empty pipelines and FIFO. Note that, if we send **`MsgRequestObjIdsBlocking`** while the messages in the pipeline do not cover all of the objects in the FIFO that we are missing, then we might get stuck with ids that we will not be able to process until the server gets net ones. This point holds also if we have requested the objects from other peers, as they might fail to deliver. Finally, note that, if we send **`MsgRequestObjIdsBlocking`** while there are messages in the pipeline that are not **`MsgRequestObjs`**, then we might receive new ids that we will not be able to process (with this server) until the server gets new ones.

We believe that this rule covers all the use cases one might have of this protocol. Variations from this rule for specific protocols would domain knowledge that for sure we will not need specific objects, but then nothing prevents us, in such a case, from first acknowledging those objects with a **`MsgRequestObjIdsNonBlocking`**, and only then send **`MsgRequestObjIdsBlocking`** if the answer was empty.

2.5.3.2 Orchestration

A downstream node should run an instance of the object diffusion mini-protocol for each of its upstream peers, with the client side running on the node, and the server side running on its peers.

If the network is well-connected, it means that the node should receive the same object id from multiple peers. The choice of which peer(s) from which to actually download the object depends on a large variety of parameters, specific to each parameterization of the mini-protocol:

- the latency/throughput constraints,
- the expected network load,
- the quality of each node-to-node connection.

[9] This is different from the behaviour of Tx-Submission, that allows the server to omit transactions in its response, indicating by doing so that they are invalid. This behaviour can be recovered by using transaction *options* as ***`object`***s.

[10] For votes and certificates, we will use the latter. See Section 2.6.

2.5.3.3 Use-case specific functions

As the object diffusion protocol is generic, its implementation may benefit from having specific key functions where use-case specific behavior is implemented. This would allow code sharing between implementations. We identify the following use-case specific functions:

In the server:

- ***initialise***, consuming ***initialPayload***.
- ***nextIds*** to retrieve the ids of objects it can serve, when receiving **MsgRequestObjIdsBlocking** or **MsgRequestObjIdsNonBlocking**.
- ***idsToObjects*** to retrieve objects given their ids, when receiving **MsgRequestObjs**.

In the client:

- ***onRecvId*** to add object ids to the FIFO, and to reward/penalize their peer depending on how much time they took and/or how many ids they sent.
- ***onRecvObject*** to register new objects in the database.
- ***objectStatus*** to retrieve data about an object, given its id. Typical data may include the expected size of the object, and whether it has been/should be requested from the server to which this client instance of the mini-protocol is connected. Reasons for no longer wanting to retrieve an object could be that it has already been retrieved from another peer, has been considered invalid, or has been superseded (for instance, if the object is a vote, but a certificate for the same round has already been received). In any case, this hook should provide enough information to decide whether to acknowledge the object's id when sending the next **MsgRequestObjIdsBlocking** or **MsgRequestObjIdsNonBlocking** to the server.

I may be possible to have a fully generic implementation of the object diffusion mini-protocol, where the use-case specific behavior is entirely contained in the key functions above. In that case, those could be provided as hooks to the generic implementation.

2.6 Vote and certificate diffusion

This section presents our proposed protocols for diffusion of votes (section 2.6.1) and certificates (section 2.6.2) between peers, based on the object diffusion mini-protocol described in section 2.5. Finally, section 2.6.3 presents a discussion about alternative designs that we have considered.

2.6.1 Vote diffusion mini-protocol

Requirements. Diffusion of votes takes place between caught-up nodes in every Peras round. The core requirement is that all honest pools observe a quorum before the end of a round, as long as votes with weight of at least `perasQuorum` have been cast for the same block in that round.

Concretely, assuming $\texttt{perasN} = 1000$ and a vote size of $164\,\text{B}$ (table 2.2 and fig. 2.2), the total size of the votes of a round is $\texttt{perasN} \cdot 164\,\text{B} = 164\,\text{kB}$, which needs to be diffused within $\texttt{perasRoundSlots} = 90$ slots/seconds. This is a more relaxed timeliness constraint compared to block diffusion, where blocks of size up to $90\,\text{kB}$ need to be diffused within $\lesssim \Delta = 5$ slots/seconds.

Using object diffusion. We propose to use the generic object diffusion mini-protocol (section 2.5) instantiated for votes. In particular, we want:

object A Peras vote.

id A vote ID, i.e. a pair of a round number and an identifier for a committee seat of that round, determining the pool identity.

Concretely, in scheme described in [Bus25], this is either the hash of the cold key of the voting pool, or (as an optimization) an index into the stake distribution.

objectIds, ***responseToIds*** A list of ***id***s. Correspondingly, ***objectIds*** is the identity function.

initialPayload No payload is necessary; only votes near the current round of the system are ever diffused.

The server advertises the vote IDs of the votes it has received for the current round (sorted by weight in decreasing order), cf. section 2.2.5. Votes for older rounds are not advertised, cf. section 2.2.5. Additionally, it serves the votes corresponding to the unacknowledged vote IDs.

We now describe the dynamics of an honest node engaging as the client in vote diffusion with its peers, assuming that it is caught-up.

- The node continually requests more vote IDs from its peers (with a limit on unacknowledged votes), using non-blocking or blocking requests depending on whether there are are outstanding unacknowledged votes.
- The client disconnects from the server if the round of the vote ID is from the future or the past (modulo some acceptable clock skew), cf. sections 2.2.4 and 2.2.5.
- At the beginning of a Peras round, the node will start receiving lots of new vote IDs for that round from all of its (honest) peers. For each such vote ID, the node will ask one peer (or a small number in parallel) out of the peers that offered this vote ID for the corresponding vote, relying on protocol-level timeouts for a prompt delivery (or otherwise disconnecting from the peer).

  By limiting the number of votes that are being downloaded in parallel, the traffic is implicitly spread out over the first few seconds of the round, bounding the spike of network activity.

  This strategy naturally handles adversarial vote equivocation, cf. appendix A.2.3.
- The client stops downloading votes as soon as it has votes with weight of at least `perasQuorum`. However, it still continues to offer these vote IDs/votes during the round, as it can be more efficient for downstream peers to download a few remaining votes to observe a quorum by themselves instead of downloading a certificate.
- The client always checks that the votes indeed correspond to the advertised vote IDs, and that the votes are in fact valid, and disconnects from the offending peer otherwise. Together with the fact that the number of votes per round is bounded by the size of the Peras committe, this bounds the work of the client.

The node does not send requests for vote IDs or votes while it is syncing (in particular, it could not even validate them at this point), which can be determined by the Genesis State Machine [Bac+25] or a more ad-hoc criterion like the proximity of the chain tip to the current wall-clock time.

This protocol shares several similarities with Tx-Submission, and we anticipate that the implementation can benefit from the insights of the design in `cardano-node`. In particular, the IOE Network team

has been reworking the inbound side of Tx-Submission[11] to more efficiently download transactions from different peers (avoiding repeated downloads).

Finally, we mention that nodes that do not intend to participate in relaying traffic through the Cardano network (for example certain wallets or nodes used by dApp developers) can reduce bandwidth requirements both for them and their upstream peers by requesting certificates (section 2.6.2) instead of votes. We propose to add an appropriate configuration flag to toggle this behavior.

2.6.2 Certificate diffusion mini-protocol

Requirements. Diffusion of certificates is required both for syncing (a)) and caught-up (b)) nodes:

a) Primarily, nodes/pools syncing via Ouroboros Genesis need to be able to obtain historical certificates in order to choose the correct historical chain, cf. section 2.4.

   The Cardano implementation of Ouroboros Genesis [Bac+25] requires that syncing nodes are always connected to at least one honest peer. (This requirement is called the "honest availability assumption".) In practice, this is guaranteed by connecting to a sufficient number of appropriately sampled peers. To reduce the load on these peers (and for a more efficient use of resources generally), downloading the same certificates from multiple peers must be avoided.[12]

b) Additionally, diffusing certificates even between caught-up nodes is necessary in certain cases involving adversarial activity, such as late votes (section 2.2.5) and equivocation attacks (appendix A.2.3).

We note that certificates can also be diffused in blocks (in order to coordinate the end of a cooldown); this is orthogonal to the discussion in this section.

Using object diffusion. We propose to again use the generic object diffusion mini-protocol (section 2.5) instantiated for certificates. In particular, we want:

object A certificate.

id A Peras round number.

objectIds, ***responseToIds*** In its simplest form, a list of Peras round numbers, and the identity function.

As an optimization, compact/compressed representations are possible, for example

- a round followed by a bitset for the subsequent rounds, indicating whether a certificate is present, or
- a round followed by a run-length encoding of the subsequent rounds. This is motivated by the observation that in Peras, rounds are (not) successful in larger runs, either because all rounds are successful due to sufficient honest votes, and if not, a lengthy cooldown period (with no successful rounds to due honest abstention) of unsuccessful rounds.

The ***responseToIds*** function is then decoding this compact representation.

[11] `https://github.com/IntersectMBO/ouroboros-network/pull/4887`

[12] We find this motivation in other parts of Ouroboros. For instance, as part of the Cardano Genesis implementation, we ensured that both block bodies and headers were downloaded only once.[Bac+25]

initialPayload A Peras round number.

Partially synced/recently caught-up peers can use this to receive certificates starting from the first round for which they do not yet have a certificate, avoiding the transfer of older, already-downloaded data.[13]

An honest server answers requests for more round numbers by sending those for which it has a certificate in ascending order, until the client is caught-up, in which case the server blocks (or returns empty results) until it observes new certificates arising from successful Peras voting rounds.[14] At any time, it will serve the certificates corresponding to unacknowledged round numbers, while enforcing an upper bound on the this quantity.

This design leverages the fact that there can only be at most one certificate per round, which allows us to elide data like the point of the block that is being certified/boosted, justifying the use of round numbers as certificate IDs. In particular, it does not matter who we download a certificate for a particular round from.

We now give a high-level description of the envisioned dynamics of this protocol.

- Consider a syncing node via Ouroboros Genesis which still needs to catch up a significant part of the historical chain. The node continually requests sizeable chunks of round numbers via **`MsgRequestObjIdsNonBlocking`** for each peers. For every round number that is advertised by at least one peer, the corresponding certificates are downloaded from the peers via **`MsgRequestObjs`** (as a first step, a singular designated peer which a simple batching strategy could be used; but more sophisticated strategies, similar to the existing BlockFetch logic are conceivable). The client uses protocol-level pipelining to avoid round-trip delays and to make full use of the available bandwidth.

- As the node is eventually done catching up, its peers run out of certificates to serve, and the node will start sending **`MsgRequestObjIdsBlocking`** instead.[15] Usually, it will receive one new round number per peer every `perasRoundSlots` slots, but given that the node now participates in vote diffusion, it is not necessary to actually download the certificate.

  An exception is the scenario where the node has downloaded all votes for a round without observing a quorum, but still received this round number via the certificate diffusion protocol. In this case, the node will download the certificate via certificate diffusion. This can only happen due to adversarial activity like vote equivocation, see appendix A.2.3.

Clients can ensure progress in this protocol w.r.t. adversarial servers by enforcing appropriate timeouts and a monotonicity property on the advertised round numbers:

a) Prompt delivery of requested certificates can be ensured by protocol-level timeouts, or via a more elaborate mechanism like a *leaky token bucket* as used in the Cardano Genesis implementation [Bac+25] which handles temporary latency spikes more gracefully.

b) The sequence of round numbers sent by the server must increase *almost* monotonically, as honest nodes (acting as servers) can observe the certificate for round r before the one for round $r-1$ in edge cases, see section 2.2.6.

[13] This is conceptually analogous to **`MsgFindIntersect`** in the ChainSync protocol.

[14] The resulting sequence of round numbers is *almost* monotonically increasing, with a possible exception once the client has (almost) caught-up, see section 2.2.6.

[15] In particular, an incomplete/empty reply to **`MsgRequestObjIdsNonBlocking`** is analogous to **`MsgAwaitReply`** in the ChainSync protocol.

This almost-monotonicity requirement can be enforced implicitly by an appropriate leaky token bucket (also see a)) via an approach analogous to the "Limit on Patience", the mechanism used in the Cardano Genesis implementation [Bac+25] to guarantee progress in ChainSync while syncing.

In short, the idea is to make sure that the server sends round numbers higher than anything it has sent before at a minimum rate on average, or that the server advertises that it has no more certificates. An honest server will have no trouble in doing that, as it will only ever send round numbers in non-monotonic order when the client is almost caught-up. Once it is caught-up, this mechanism can be disabled, just like the Limit on Patience.

Finally, we note that in order to conclude that a round *does* have a corresponding certificate, it is enough to download such a cert from any peer, while concluding that there is *no* certificate for a round r requires information from *all* peers (in the form of advertising a certificate for a round sufficiently larger than r as per b), or by the peer indicating that they have no more certificates at the moment).

2.6.3 Alternatives

We briefly discuss two alternatives to the design of vote and certificate diffusion presented above.

- One could combine vote and certificate diffusion into one custom protocol. This would allow for some optimizations, such as responding with a certificate when a client asks for a vote that it subsumes, and could make certain interactions between the vote and certificate clients explicit.

  However, the separation of votes and certificates actually allows the individual protocols to be simpler and more focused, and the aforementioned optimization does not seem relevant in practice, in particular as one still needs synchronization between the clients of different peers.

  Also, there is an advantage in reusing object-diffusion mini-protocol (section 2.5) due to its similarity to the existing Tx-Submission protocol, and the planned use for Mithril [Ray+24] and Ouroboros Leios.

- Certificate diffusion could use *two* protocols similar to how chains are diffused in Cardano. Concretely, the first protocol would be similar to ChainSync and diffuse just certificate IDs (i.e. round numbers) and the second protocol would be an instantiation of BlockFetch with certificate IDs instead of block points and certificates instead of blocks.

  However, this approach seems overly complicated for the purpose of certificate diffusion. The primary motivation for having separate ChainSync and BlockFetch protocols is the *header-body split* [Cou+20], and ChainSync is specifically designed for stateful chain following, whereas certificates do not have an inherent chain structure (despite voting for blocks on a block chain).

2.7 Using chain weight instead of chain length

Any certified block receives extra weight corresponding to `perasBoost` per certificate.[16] Various components that currently consider chain *length* must instead consider chain *weight*.

Chain selection Most prominently, chain selection must no longer select the longest, but instead the *weightiest* chain, using all currently available certificates. In particular, the selection of a node

[16] Usually, blocks will be boosted by at most one certificate, but it is possible for the same block to be boosted by multiple certificates, for example if there is no active slot within a Peras round, which happens with probability $(1 - \texttt{asc})^{\texttt{perasRoundSlots}} \approx 1\,\%$ for $\texttt{asc} = 1/20$ and $\texttt{perasRoundSlots} = 90$.

can now additionally change solely due to the emergence of a new certificate, without any new blocks.

Volatile vs. immutable chain The node maintains its selection as an immutable prefix and a volatile suffix, where only the latter is subject to rollbacks. Without Peras, the volatile suffix is defined to be the `kcp` most recent blocks. With Peras, it is instead defined to be the largest suffix containing blocks of weight at least `kcp`.[17]

When Peras is not in a cooldown phase, this means that the length of the volatile suffix decreases from `kcp` blocks to

$$\left\lceil \frac{\mathtt{kcp}}{1 + \frac{\mathtt{perasBoost}}{\mathtt{perasRoundSlots} \cdot \mathtt{asc}}} \right\rceil = 499$$

blocks in expectation, assuming no adversary and full honest participation, as well as $\mathtt{perasBoost} = 15$, $\mathtt{asc} = 1/20$ and $\mathtt{kcp} = 2160$.

As a consequence, and because we only need to store ledger states for each volatile block and the immutable tip, fewer ledger states need to be stored in that case. However, this expected to only result in a small gain in efficiency, as ledger states corresponding to subsequent blocks only differ by a small amount in relation to the total size, and existing implementations exploit this fact by using implicit structural sharing or by explicitly maintaining these diffs.

Ouroboros Genesis rule The Cardano Genesis implementation [Bac+25] performs density comparisons between competing header chains while syncing. With Peras, this needs to be modified to use the *weighted* density instead (cf. section 2.4 and appendix A.2.2).

Concretely, the *Genesis Density Disconnection* governor needs to be revised to become aware of certificates. In general, disconnecting from one of two peers offering competing historical chains requires definite knowledge of all certificates in rounds that can boost a block in the first s_{gen} slots after their intersection. We achieve this via the `perasBlockMaxSlots` parameter (establishing an upper bound on the maximum relative age of a block that a certificate can boost) and the discussion of progress of certificate diffusion in section 2.6.2.

BlockFetch decision logic The BlockFetch decision logic is responsible for downloading blocks corresponding to candidate header chains that are preferable to the current selection. The comparison between candidate header chains and the current selection needs to take weight into account.

Hard Fork Combinator The Hard Fork Combinator (HFC) is Cardano's mechanism for transparently handling transitions between different Cardano eras. Before enacting an era transition decided by on-chain governance, the HFC requires at least `kcp` blocks after the *voting deadline*, i.e. the last slot that could still affect the on-chain governance.[18] This mechanism is called *block counting*.

Relying on Praos Chain Growth, these `kcp` blocks must arise in T_{cp} slots. In Peras however, Chain Growth guarantees instead that T_{cp} slots contain blocks having *weight* of at least `kcp` blocks on any chain, which does not imply that there actually are `kcp` blocks. See appendix A.1.3 for more details on how the adversary can cause the chain to be of high weight, but low (unweighted) density.

[17] Note that it is acceptable (but overly conservative) to not capitalize on this observation, and still consider the last `kcp` blocks to be volatile.

[18] See `https://ouroboros-consensus.cardano.intersectmbo.org/docs/for-developers/CivicTime` for more context.

As block counting is a part of ledger validation, it is not possible to directly change this rule to use weight instead, as the set of certificates is not availabe in this context, and it is not desirable to make block validation dependent on external data like certificates.

It is conceivable that a more fine-grained analysis could result in the probability to have less than `kcp` blocks (or an appropriately increased quantity) to still be acceptably low for this purpose, such that the HFC can continue to use block counting. We also not that block counting has been under closer scrutiny and been subject of discussions to potentially replace it for other, unrelated reasons.

2.8 Vote storage (PerasVoteDB)

The node needs to store votes for the current round (cf. section 2.2.5) in order to observe a quorum, and to diffuse these votes to downstream peers while the round is still ongoing even after a quorum has been observed.

- As soon as the PerasVoteDB contains votes with weight of at least `perasQuorum`, it starts aggregating these votes into a certificate and adds it to the PerasCertDB (section 2.9).
- At the beginning of a Peras round r, all votes prior to r are deleted from the PerasVoteDB.

We propose to store these votes in memory only; not persisting its state to disk. Restarting a node is realistically going to take longer than `perasRoundSlots`, so any persisted votes would become stale immediately after a restart.

2.9 Certificate storage (PerasCertDB)

2.9.1 In-memory storage of recent certificates

Various components of the node need quick access to all certificates applying to a particular candidate chain in order to compute its weight, cf. section 2.7. In addition, the Peras aspect of the block minting logic (section 2.10.1) needs access to the most recent certificate.

We propose to maintain an in-memory cache in the PerasCertDB of all certificates that can apply to a block that the node could select. Older certificates (e.g. those with a round that ends before the tip slot of the immutable chain) can be garbage-collected from this in-memory cache, which bounds the size of the cache. Therefore, as there is at most one certificate per round, this cache contains at most $\lceil T_{\mathsf{cp}} / \texttt{perasRoundSlots} \rceil = 1440$ certificates, so in total $28.8\,\mathrm{MB}$ for the proposed parameters (section 2.1).

2.9.2 On-disk storage of historical certificates

Nodes serving the historical chain need to retain all past certificates in order to provide them to peers syncing from them via Ouroboros Genesis, see appendix A.2.2. Naturally, due to its unbounded size (linear to the age of the chain), these certificates need to be stored on disk.

We store certificates on disk that are boosting a block on the immutable chain.[19] Therefore, the on-disk store is *append-only*, and all certificates stored in it are *immutable*.

[19] The only certificates that do not have this property must be from rounds shortly before a cooldown period, so they are rather rare.

These properties are similar to those of the ImmutableDB in the `cardano-node` implementation, [VWC20, chapter 8], and the on-disk storage has similar requirements:

a) The store must efficiently add new certificates, but it *is* acceptable if recently added certificates are lost on a crash, as they can be re-downloaded.

b) The store must provide performant functionality to implement the server side of certificate diffusion section 2.6.2. Concretely, we require efficient lookups and streaming of certificates indexed by their round number.

A simplified variant of the ImmutableDB as implemented in the `cardano-node` fulfills these requirements, as does any standard key-value store (which usually provide many additional features/guarantees that are not strictly required here).

2.10 Minting

2.10.1 Voting component (PerasVoteMint)

At the beginning of every Peras round, a pool needs wake up and perform the following tasks in order:

a) Check if it is eligible for a seat in the Peras committee of that round. If not, exit here.

   If the node is currently syncing, this might be impossible to determine, just as for the block minting logic. In this case, the node does not vote.

b) Determine the candidate block B to potentially vote for. If none[20], exit here.

   In the pre-alpha version of Peras described in this document, this is defined by a simple rule (the latest block on the current selection that is at least `perasBlockMinSlots`, but at most `perasBlockMaxSlots` old compared to the first slot of the current round). Future versions of the Peras protocol might include a more sophisticated pre-agreement rule here.

c) Check if it should participate in voting this round, using [Bai+25, Rules for voting in a round]. If none, exit here.

   This requires access to the most recent certificate, which is available via the PerasVoteDB (section 2.8), and the most recent certificate stored on the current selection, which is stored in the corresponding tip ledger state.

   Note that the node might not vote in a round, but continue voting in the next round, see appendix A.2.1 for details. However, usually, honest nodes will only stop voting when the protocol enters a cooldown period.

d) Cast a vote for B and add it to the PerasCertDB (section 2.8), which will cause it to be diffused to the downstream peers.

2.10.2 Modification to the block minting logic

Honest nodes need to include a certificate on chain when the protocol is in a cooldown period in order to coordinate the *end* of that cooldown period. We refer to [Bai+25, Block creation] for the precise condition. This extra check is computationally trivial.

[20] This can only happen when the chain is very sparse.

As mentioned in section 2.3, the size of the certificate count towards the maximum block body size, so logic for selecting transactions to include in the block needs to be appropriately tuned to select fewer transactions in that case.

2.11 Node-to-client protocols

In order to make use of Peras and its improved settlement guarantees under optimistic conditions, applications need to monitor whether or how many of the block containing their transaction of interest and its descendants are boosted via a Peras certificate.

For this, we propose to run certificate diffusion (section 2.6.2) also as a node-to-client protocol, such that clients can process these themselves for their use case. For example, such a separate service can provide an easy-to-use API that can answer queries about recent blocks with concrete settlement probabilities based on the observed certificates.

We do not see a strong use case for exposing votes as a node-to-client protocol; monitoring tools for Peras can likely use the node-to-node protocol directly.

Chapter 3

Interactions with other features

We provide a preliminary discussion of the interactions of Peras with Mithril[1] and Ouroboros Leios[2].

3.1 Mithril

Mithril is a protocol for generating stake-based threshold multi-signatures for certain entities related the Cardano blockchain, for example the stake distribution, the node database or even individual transactions.

3.1.1 Bootstrapping Peras certificates

Mithril[1] is often used to bootstrap a node by fetching a pre-synced chain database which has been signed via Mithril instead of syncing manually from other nodes. Given that Peras is adding *certificates* to the chain database, we now sketch how these could be handled by Mithril.

- Applications that are not interested to serve the chain to other nodes themselves[3] do not need Peras certificates, so Mithril does already work for this use case without any additional modifications.
- Nodes that *do* want/need to serve the historical chain to other nodes (e.g., pools) need to obtain all historical certificates. This is non-trivial, as there can be many different valid certificates for the same block in some round, and even further, honest nodes do not necessarily have the exact same collection of historical certificates, see section 2.2.2. This is in contrast to the immutable chain, on which all honest peers agree on eventually up to a specific slot.

  It seems plausibly possible to us to work around these complications, but it requires further thought and discussion with the Mithril team.

3.1.2 Reusing votes across Peras and Mithril

Just like Mithril, Peras needs to diffuse votes in order to create certificate, which is a stake-based threshold multi-signature for the block that is being voted for.[4] Conceptually, it is therefore possible to use the same votes for Peras *and* for Mithril, adding the data to be signed for both applications to individual votes (concretely, the point of the block that Peras is voting for, and e.g. the hash of a (stable) stake distribution for Mithril). Peras and Mithril would aggregate these votes into certificates independently from each other.

[1] `https://mithril.network/`

[2] `https://leios.cardano-scaling.org/`

[3] We note that such nodes might also benefit from other optimizations, such as eliding the transaction signatures from blocks ("segregated witness").

[4] The Mithril team intends to use a protocol very similar to Tx-Submission, see [Ray+24].

The main benefit of this approach is that all pools would automatically and at essentially no extra cost for themselves on top of Peras (assuming that the data that Mithril requires to be signed is sufficiently lightweight, which should be the case for e.g. stake distributions) participate in Mithril voting. In particular, no bandwidth is wasted for duplicate

Of course, this approach requires that Peras and Mithril use the same or compatible cryptography for votes and certificates. The current design for votes and certificates in Peras [Bus25] is different from Mithril's certificates [CK24]; however, we understand that there is an effort to use a unified approach of stake-based threshold multi-signatures across various Cardano-related projects, in particular for Peras and Mithril.

Another complication is the existence of Peras cooldown periods. As presented in the Peras CIP [Bai+25] and this document, honest nodes stop voting during such a cooldown period, which, even if rare, would also disable Mithril as a side effect, which is undesirable. A natural mitigation is to let nodes still vote even during Peras cooldown periods, but indicate in the vote that it must not be considered as a Peras vote, only as a Mithril vote.

In general, the main drawback of this idea is that it introduces coupling between components that are logically separate, and hence imposes accidental complexity that needs to be weighed against the aforementioned advantages. In particular, this coupling makes a separate evolution of Peras and Mithril more difficult.

3.2 Ouroboros Leios

Ouroboros Leios[2] is a high-throughput protocol for Cardano.

3.2.1 Combining Peras and Mithril

From a very high level point of view, Peras and Leios are orthogonal features: Leios is a significant change to the protocol with many new kinds of blocks as well as votes and certificates. However, in the end Leios, still establishes a chain made out of *ranking blocks* that follows the longest chain rule as in Praos. Peras can still be applied to this chain (i.e. votes would be case for ranking blocks), providing faster settlement under optimistic conditions.

In other words, interacting with the Cardano blockchain via a transaction involves two steps:

a) First, the transaction needs to be propagated through the network and be included in a block.

b) Second, the block needs to be settled with a probability appropriate for the concrete use case.

Leios accelerates step a) (by increasing throughput), while Peras speeds up step b) (by boosting the block or one of its descendants).

On a lower level, Peras and Leios fundamentally compete for bandwidth in the system (as Leios is all about making full use of it, in contrast to Praos). The Leios innovation time is currently working on elaborate realistic simulations of Leios to assess its dynamics and performance profiles. Enriching these simulations with Peras (concretely, its vote diffusion) can provide further insights into this tradeoff. However, we note that a future version of Peras might incorporate a dedicated *pre-agreement* mechanism of unclear specifics.

Overall, there is no explicit dependency between Peras and Leios, but rather common dependencies like the work on a cryptographic scheme for votes and certificates. Moreover, Leios will likely also use a variant of Tx-Submission do diffuse its various kinds of new entities, which provides an opportunity for collaboration.

3.2.2 Reusing votes

Like Peras and Mithril, Leios also makes use of stake-based threshold multi-signatures (in order to certify data availability), and needs to diffuse votes for this purpose. Similar to the discussion in section 3.1.2, it is therefore in principle conceivable to share these votes between Peras and Leios (and Mithril), potentially mitigating the competition for bandwidth.

At present, it is unclear whether or how Peras could be more tightly integrated with Leios, such as voting for input blocks or directly reusing the endorsement certificates for faster finality. It seems likely that any such approach would have to differ signficantly from Peras in its current form.

Chapter 4

Outline implementation plan

We sketch a possible implementation plan to integrate the design presented in this document into `cardano-node`, the Haskell implementation of a Cardano node. However, similar considerations would plausibly apply to other implementations.

We exclude the implementation of the underlying cryptography used for votes and certificates from this discussion, and assume the availability of appropriate Haskell bindings. In particular, the scheme described in [Bus25] requires pools to register new keys, which necessitates appropriate tooling to generate and manage those.

4.1 Prototype without historical certificates

As a first step, Peras can be implemented without storing any historical certificates. This prevents syncing via Ouroboros Genesis with reduced trust assumptions, but syncing from trusted peers is still possible. In this form, Peras can already be used to run a testnet to start to empirically explore and test the implementation in a real-world environment.

Such a prototype involves the following work items.

a) Implementation of votes and certificates (section 2.2), the associated validation logic and serialization.

b) Changes to the Cardano Ledger to include certificates in block bodies (section 2.3).

   We propose to make the Ledger only minimally aware of certificates, as they are morally a Consensus concern and only included in block bodies due to their size. More concretely, the Ledger logic would consider certificates to be opaque objects, with the actual validation logic being provided by the Consensus layer when it invokes the ledger rules.

   The new protocol parameters (section 2.1) which are subject to governance can initially be hard-coded, but we expect it to be a routine operation for the Ledger team to add them to a new Ledger era.

c) Implementation of the (in-memory) PerasVoteDB (section 2.8) and associated vote minting logic (section 2.10.1) and vote diffusion (section 2.6.1).

   This requires defining the object diffusion mini-protocol using the `typed-protocols` library, and implementing appropriate client and server logic. The client logic is more complicated, and can initially download all votes for every peer (similar to how Tx-Submission worked before recent work by the Network team[1]).

d) The in-memory part of the PerasCertDB (section 2.9).

e) The (straightforward) modification of the block minting logic to include certificates (section 2.10.2).

[1] `https://github.com/IntersectMBO/ouroboros-network/pull/4887`

f) Support for efficiently computing the *weight* of chains, and making use of this functionality in chain selection and the BlockFetch decision logic (section 2.7).

g) Integration with the Hard Fork Combinator, see section 2.7.

h) Optionally, a restricted version of certificate diffusion that does not support receiving historical certificates (section 2.6.2). We recall that certificate diffusion between caught-up nodes is only necessary in certain scenarios involving adversarial activity. For the purpose of using this prototype in a testnet, it therefore is possible to elide this step and instead implement it as part of fully implementing certificate diffusion in section 4.2.

This work largely falls into the scope of the Consensus team, with interactions with the Ledger team (b)) and the Network team (c)).

4.2 Adding certificate diffusion and historical certificates for Ouroboros Genesis

To support syncing via Ouroboros Genesis, the following additional steps are necessary.

a) Implementation of the on-disk part of the PerasCertDB (section 2.9).

   For a straightforward initial implementation, we recommend to use a key-value store, such as LMDB, which is currently used by the UTxO-HD project to store the UTxO map on disk, or lsm-tree[2], which is currently developed by Well-Typed as a replacement for LMDB in the aforementioned use case. We note the use cases of storing the UTxO map on disk and the PerasCertDB are pretty different, so certain features of these libraries that are important for the former case (such has supporting random access write-heavy workloads) are not relevant for us.

   Alternatively, adapting and simplifying[3] the existing implementation of the ImmutableDB can be considered.

b) Full implementation of certificate diffusion (section 2.6.2).

c) Update the Cardano Genesis implementation [Bac+25] to compare competing header chains using their weight (instead of their unweighted density) (section 2.7).

   Concretely, this requires changes to the Genesis Density Disconnection governor, the component that disconnects peers who offer a header chain that is definitely less dense/has less weight than another one.

d) Adjust chain selection to ensure that certificate acquisition does not fall behind block adoption (section 2.4) as part of the "duty to remember" certificates.

4.3 Discussion of implementing Peras externally to the node

The implementation approach described in the preceding sections assumes that Peras will be directly integrated into the existing node. In this section, we briefly discuss to what extent Peras could be implemented externally as a separate component. This is motivated by Cardano features like Mithril

[2] `https://github.com/IntersectMBO/lsm-tree`

[3] For example, complications due to epoch boundary blocks are not relevant for the PerasCertDB.

(section 3.1) being implemented externally (using the *Extensible Ouroboros Network Diffusion Stack*[4]), and Project Caryatid[5], a prototype for a potential node architecture based on microservices.

We note that the interaction between the core node component and an external Peras component must be *bidirectional*, as the validity of votes and certificates and the voting logic are determined by the current ledger state (maintained by the node), and in turn, these certificates (from the Peras component) fundamentally affect the chain selection process of the node. Therefore, implementing Peras externally still requires non-trivial changes to the core node. This stands in contrast to e.g. Mithril, where the Mithril component requires certain data from the node (e.g. the stake distribution), but the node does not have to be aware of Mithril.

In other words, the degree of coupling between a Peras component and the node will be relatively high, contrary to the usual requirement of loose coupling in a microservice architecture. Therefore, we anticipate that an implementation that integrates Peras directly into the existing node will be quicker to execute and easier to maintain.

Finally, we sketch how Peras could fit into a hypothetical future microservice architecture for the node with fine-grained components. In particular, assume the existence of a dedicated *Consensus* component responsible for e.g. chain selection and block minting, which defers to other components for e.g. block validation and maintenance of the transaction mempool. (This is similar to the separation of consensus and execution clients in Ethereum[6].) In this case, almost all of the changes mandated by Peras would only affect this Consensus component, with the exception of certificates contained in blocks to coordinate the end of a cooldown period.

[4]Also known under the name "reusable diffusion", cf. `https://github.com/IntersectMBO/ouroboros-network/wiki/Reusable-Diffusion-Investigation`

[5]`https://github.com/input-output-hk/caryatid`

[6]`https://ethereum.org/en/developers/docs/nodes-and-clients/`

Chapter 5

Testing

We present a series of tests for Peras to ensure its functionality when being implemented into `cardano-node`, the Haskell implementation, both under optimistic and adversarial conditions.

We exclude considerations of testing the cryptographic scheme underlying votes and certificates.

5.1 Component-level tests

We describe how the components of the Peras architecture (both new and modified ones) can be tested.

- The IOE Consensus team is currently collaborating with the IOE Formal Methods team in establishing *conformance tests* for header validation, comparing the actual implementation to the formal specification in Agda [CBV24]. We advise to consider similar tests for ensuring the correctness of vote and certificate validation.
- To test the enriched chain selection algorithm which accounts for weighted chains via boosted blocks, it is natural to modify the existing state machine test in lock-step style, which uses property tests to ensure the equivalence to a simple model implementation.

  Similarly, we recommend state machine tests for the PerasVoteDB and the PerasCertDB. In case of the latter, explicit fault injection can be used to check for appropriate behavior under e.g. disk corruption that are otherwise hard to test.
- The modification to the Cardano Genesis implementation (using weight to compare candidate chains) can be checked by adapting the existing extensive test suite for Genesis. For example, the Genesis tests rely on generating a *block tree* that indicates all possible chains that exist for this particular test run, and could hence be served via a (randomly generated) adversarial strategy. For Peras, it is necessary to generate *weighted* block trees such that the new logic is exercised non-trivially.

5.2 Simulation tests with generated environments

We propose to simulate the behavior of an honest Peras node in a generated environment, in order to catch regressions in the overall dynamics of the system, as well as to exhibit scenarios that are hard to test due to requiring complicated setup and tooling in an integration test.

- An optimistic environment where all nodes are behaving honestly. In this case, all Peras rounds must be successful.
- Temporarily decreased participation in voting, leading to unsuccessful rounds. Here, the node must pause voting for some time, potentially include a certificate on chain, and once the cooldown period ends, resume again.

- Environments involving adversarial behavior of pools with a given amount of stake. These may either be specific strategies from a pre-defined list (such as equivocation attacks), or randomly generated "chaotic" adversaries.

  Here, the properties to test for are the core guarantees of the system, i.e. safety and liveness.

- The Peras conformance tests[1] provided by the Peras Innovation team also constitute an environment to test against. In this case, success is decided by the conformance suite (backed by a model implementation).

  This requires an appropriate layer of glue (for example, to translate between the pull-based mini-protocols and the push-based replies expected by the conformance test).

These simulations can leverage the `io-sim`[2] library for deterministic tests even in the presence of concurrency and time that is inherent in such a context.

5.3 Integration tests and benchmarks

The IOE Cardano Performance & Tracing team maintains an elaborate cluster benchmarking setup using a variety of high-load scenarios. It is used to validate the performance of the node, in particular by catching regressions. Naturally, this setup can be extended to measure the impact of Peras on key metrics such as block diffusion time and resource usage per node, while monitoring that Peras behaves as expected, i.e. that all Peras rounds are successful (due to the absence of an adversary).

We also recommend to consider setting up a dedicated testnet for Peras, similar to the past "Sanchonet" whose purpose was to assess the in-progress implementation of Voltaire governance. This enables an early experimentation environment for pool operators as well as developers wanting to integrate the settlement guarantees of Peras into their application.

[1] `https://github.com/input-output-hk/peras-design/blob/63e4224c4f6c286c58121acabe366d9be1d90998/peras-simulation/Conformance.md`

[2] `https://github.com/input-output-hk/io-sim`

Chapter 6

Identified risks and opportunities

We have identified the following risks inherent in the protocol that we do not see how to address at the implementation level:

- An adversary can include useless certificates on chain while the system is not in a cooldown period, cf. section 2.3. This is not a fatal flaw, and might be of limited relevance if the cryptographic scheme for Peras allows for very small and cheap-to-validate certificates.
- The pre-alpha version of Peras is characterized by the lack of a dedicated *pre-agreement* mechanism for avoiding cooldown periods as much as possible. We stress that this implies that adversaries with less than 25 % of the stake can cause (long!) cooldown periods with decent probability, cf. appendix A.1.2.

  The additional implementation complexity of a pre-agreement mechanism has not been analyzed so far. Therefore, we recommend to conduct further research to determine if the consequences of the lack of pre-agreement are acceptable for the anticipated users of Peras.
- In its current form, there is no direct monetary incentive for pool operators to participate in Peras. This is relevant as low honest participation in voting (even without any adversary) causes Peras to be ineffective due to its lengthy cooldown periods on unsuccessful rounds. It is not obvious how to realize a reward mechanism for voting in general as there is no on-chain track record of who participated in Peras voting. Even then, further research in the form of a game-theoretic analysis would be required.

  We propose to transparently communicate this fact to pool operators to gather feedback.

Finally, we highlight an opportunity (discussed with the Peras research team) for further study of the Peras protocol that *could* avoid a significant chunk of the work necessary to implement Peras. Concretely, if such research results in the conclusion that it is possible to instantiate Peras in such a way that it is acceptable to perform *unweighted* Genesis density comparisons instead of weighted ones, it is not necessary to store historical certificates, cf. section 2.4. Here, by "acceptable" we mean that the resulting loss in security (for example due to attacks as described in appendix A.1.3) is negligible. In that case, the implementation steps described in section 4.2 would not be required anymore. In particular, Peras would not require any additional disk space.

However, since the outcome of this (unstarted) research is unclear, both in terms of its outcome and the time frame required, the architecture presented in this document does not make use of this hypothetical simplifying assumption.

Bibliography

[Bac+25] Nicolas Bacquey, Facundo Domínguez, Alexander Esgen, Nicolas Frisby, Nicolas Jeannerod, and Torsten Schmits. *Cardano Genesis*. https://github.com/IntersectMBO/ouroboros-consensus/pull/1174. 2025.

[Bad+18] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. "Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 913–930.

[Bai+25] Arnaud Bailly, Brian W. Bush, Sandro Coretti-Drayton, Yves Hauser, and Hans Lahe. *CIP-0140 | Ouroboros Peras - Faster Settlement*. https://github.com/cardano-foundation/CIPs/blob/aa314c019b857cbaea38e16b1570af90cecefd38/CIP-0140/README.md. 2025.

[Bus25] Brian W. Bush. *Peras certificates revisited: January 2025*. https://github.com/input-output-hk/peras-design/blob/febca47e37efcd980fca08d4a96e4d2d63664ef9/analytics/certificates-jan2025.md. 2025.

[CBV24] James Chapman, Arnaud Bailly, and Polina Vinogradova. "Applying Continuous Formal Methods to Cardano (Experience Report)". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Software Architecture*. 2024, pp. 18–24.

[Cha+24] Pyrros Chaidos, Aggelos Kiayias, Leonid Reyzin, and Anatoliy Zinovyev. "Approximate lower bound arguments". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2024, pp. 55–84.

[CK24] Pyrros Chaidos and Aggelos Kiayias. "Mithril: Stake-based threshold multisignatures". In: *International Conference on Cryptology and Network Security*. Springer. 2024, pp. 239–263.

[Cou+20] Duncan Coutts, Neil Davies, Marcin Szamotulski, and Peter Thompson. *Introduction to the design of the Data Diffusion and Networking for Cardano Shelley*. Tech. rep. 2020. URL: https://iohk.io/en/research/library/papers/introduction-to-the-design-of-the-data-diffusion-and-networking-for-cardano-shelley/.

[Cou+25] Duncan Coutts, Neil Davies, Marc Fontaine, Karl Knutsson, Armando Santos, Marcin Szamotulski, and Alex Vieth. *Ouroboros Network Specification, version 1.5.0 of 25th February 2025*. https://ouroboros-network.cardano.intersectmbo.org/pdfs/network-spec/network-spec.pdf. 2025.

[CVG23] Jared Corduan, Polina Vinogradova, and Matthias Güdemann. *A Formal Specification of the Cardano Ledger*. https://github.com/intersectmbo/cardano-ledger/releases/latest/download/shelley-ledger.pdf. 2023.

[Dav+18] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain". In: *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37*. Springer. 2018, pp. 66–98.

[Día25] Javier Díaz. *A Formal Specification of the Cardano Consensus*. https://ouroboros-consensus.cardano.intersectmbo.org/docs/about-ouroboros/References. 2025.

[GKR23] Peter Gaži, Aggelos Kiayias, and Alexander Russell. "Fait accompli committee selection: Improving the size-security tradeoff of stake-based committees". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 845–858.

[Kni+25] Andre Knispel, Orestis Melkonian, James Chapman, Alasdair Hill, Joosep Jääger, William DeMeo, and Ulf Norell. *IntersectMBO/formal-ledger-specifications*. `https://github.com/IntersectMBO/formal-ledger-specifications`. 2025.

[Ray+24] Jean-Philippe Raynaud, Arnaud Bailly, Marcin Szamotulski, Armando Santos, Neil Davies, and Sebastian Nagel. *CIP-0137|Decentralized Message Queue*. `https://github.com/cardano-foundation/CIPs/blob/master/CIP-0137/README.md`. 2024.

[VWC20] Edsko de Vries, Thomas Winant, and Duncan Coutts. *The Cardano Consensus and Storage Layer*. Tech. rep. 2020. URL: `https://ouroboros-consensus.cardano.intersectmbo.org/docs/for-developers/TechnicalReports`.

Appendix A

Attack scenarios

The goal of this appendix is to give an overview of various attacks on Peras, which either motivate a Peras certain rule/requirement or highlight what we can expect from sufficiently strong attacker within our margin of security.

A.1 Attacks against Peras

Here, we list attacks that a sufficiently strong adversary (within the margin of security) can execute and degrade the user experience this way (for example, due to reduced throughput or longer settlement times), but which do not threaten the core security properties of the system.

A.1.1 Abstaining during voting

An adversary with α stake will have $\alpha \cdot$ `perasN` votes on average, so if they simply abstain, the honest nodes will likely fail to reach quorum even when they all vote for the same block when $\alpha \geq 1 -$ `perasQuorumperasN` $= 1/4$ for `perasQuorum` $= 3/4 \cdot$ `perasN`. Therefore, Peras doesn't provide fast settlement in the presence of an adversary with $\alpha \geq 25\%$, which is expected.

As usual, this is assuming full honest participation; smaller participation levels can only tolerate proportionally weaker adversaries.

Additionally, this attack (and others that are about preventing a quorum) gets more effective if the cryptographic scheme used for certificates requires votes of weight larger than `perasQuorum` to certify a quorum, which is for example the case for ALBA due to its n_p/n_f parameters [Cha+24].

A.1.2 Honest vote splitting

In the pre-alpha version of Peras, honest nodes vote for the newest block on their selection that is at least `perasBlockMinSlots` slots old.[1] It is relatively easy for an adversary to cause different honest nodes to vote for different blocks.

For example, this allows an adversary with $\alpha < 0.25$ to prevent quorum formation with decent probability (which they couldn't do by simply abstaining) by splitting the honest votes such that neither block reaches quorum, causing the system to enter a cooldown period. Future versions of Peras will improve on this front by adding a dedicated *pre-agreement* mechanism that *avoids* a cooldown in this scenario.

Write $\text{tip}(C)$ for the tip of a chain C, and $\text{trunc}(s, C)$ for the prefix of C up until (but excluding) slot s.

Concretely, consider the voting phase at the beginning of round r starting in slot s, with all honest nodes having selected chain C_1 just before.

[1] There are potential variants of this rule (such as choosing the tip of the best chain made out of blocks that at least `perasBlockMinSlots` slots old) that might mitigate the attack discussed here in certain scenarios, but they do not change the overall picture.

Suppose that the adversary with stake α has a chain C_2 that is preferrable to C_1, and $D_1 \neq D_2$ where

$$D_i = \text{tip}(\text{trunc}(s - \texttt{perasBlockMinSlots}, C_i))$$

for $i \in \{1, 2\}$. If the adversary now diffuses C_2 to only a subset of the honest nodes right before r starts (such that the remaining honest nodes only receive C_2 after r began), these nodes will vote for D_2, while the rest will vote for D_1.

We can calculate a lower bound on the probability that an adversary has such a chain C_2 by considering a particular outcome of the leader schedule that guarantees the existence of C_2 when the adversary simply abstains while honest chain is growing before round r.

a) There is at least one adversarial active slot in the interval $(h, s - \texttt{perasBlockMinSlots})$ where h is the last honest active slot before $s - \texttt{perasBlockMinSlots}$.

b) There are at not more honest than adversarial active slots in the interval $[s - \texttt{perasBlockMinSlots}, s)$.

Condition a) guarantees that $D_1 \neq D_2$, and condition b) makes sure that C_2 is preferrable to C_1.

We stress that this is not the only scenario where the attack can be executed; for example, we have not considered tiebreakers and honest short forks.

Lemma A.1.1. *For an adversary with stake α the probability that the events a) and b) occur before some Peras round is given by*

$$P(H \leq A) \cdot P(A' \geq 1)$$

where $H \sim \text{Binom}(L, \phi(1-\alpha))$, $A \sim \text{Binom}(L, \phi(\alpha))$, $A' \sim \text{Binom}(H', \phi(\alpha))$, $H' \sim \text{Geom}(\phi(1-\alpha))$[2], and where $L = \texttt{perasBlockMinSlots}$ and $\phi(\sigma) = 1 - (1 - \texttt{asc})^{\sigma}$ [Dav+18, (1)].

Proof. Note that a) and b) are independent as they refer to disjoint sets of slots.

The number of active slots out of n total slots for a party with stake σ is binomially distributed via $\text{Binom}(n, \phi(\sigma))$, so $P(\text{“}a\text{”}) = P(H \leq A)$ is clear.

The number of successive inactive slots until an active slot for a party with stake σ is geometrically distributed via $\text{Geom}(\phi(\sigma))$, so H' counts the size of the interval $(h, s - L)$, and therefore $A' \sim \text{Binom}(H', \phi(\alpha))$ counts the number of adversarial active slots in that interval. Thus, $P(\text{“}b\text{”}) = P(A' \geq 1)$. □

In lemma A.1.1, we assume that the gap H', i.e. the length of the interval $(h, s - \texttt{perasBlockMinSlots})$, is sampled geometrically due to the leader schedule. However, it can also be instructive to set it to a concrete value (letting H' have a one-point distribution), especially if an adversary could force a long gap in some other way.

We plot the propabilities of lemma A.1.1 for realistic parameters in fig. A.1. We observe that for small values for `perasBlockMinSlots`, even relatively weak adversaries can execute a vote splitting attack rather frequently. This is relevant in particular as it allows them to trigger a cooldown phase.

A.1.3 Density reduction via boost-induced rollbacks

We now describe an adversarial strategy that results in the honest historical chain to have low density, but high weight, i.e. in the extreme case, there is one boosted block per Peras round, with every voting round being successful. In that case, a simple (private) fork purely made out of adversarial blocks has a good chance to have higher density (but significantly lower weight) than the honest chain.

[2]Here, we use the convention of counting the number of failures until the first success.

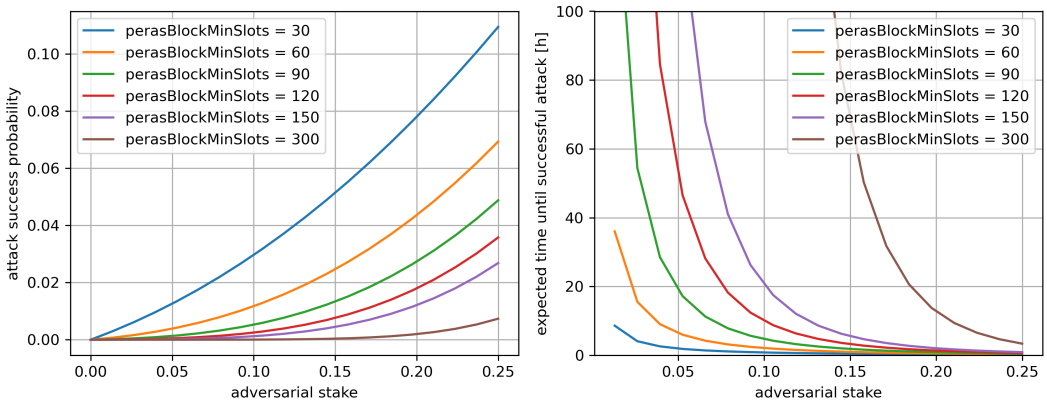

FIGURE A.1: *Probabilities (lower bounds) for lemma A.1.1 for realistic parameters. The right plot assumes a slot length of one second and* `perasRoundSlots` $= 90$.

See appendix A.2.2 for how this attack influences the Peras design.

Consider a Peras round r starting in slot s. During round r, the adversary (with stake α) does not diffuse any votes or blocks until shortly before the end of round r in slot $s +$ `perasRoundSlots`. Let C be the best chain that any honest node has selected in slot $s +$ `perasRoundSlots` $- 1$. The adversary arranges it such that during the voting phase of round r, no quorum is reached without the adversarial votes, and further, that honest votes with weight at most `perasQuorum` $- \alpha$ are cast for a block B that is *not* on C. The idea now is for the adversary to diffuse its $\alpha \cdot$ `perasN` votes for B near the end of round r such that round r is considered successful and B receives a boost. Then, under reasonable assumptions, the chain ending in B is preferrable to C (as the latter doesn't contain a block that was boosted during round r), so the adversary succeeded in creating a low density but high weight chain during round r.

Let us consider concretely how and under what conditions an attacker can execute this attack.

a) To start the attack, the attacker needs to prevent a quorum out of honest votes during round r, and needs to ensure that votes of weight at least `perasQuorum` $- \alpha$ are cast for a block that the honest nodes will not build upon during round r.

   One way to accomplish this is to execute an honest vote splitting attack, see appendix A.1.2.

   If $\alpha < 0.25$, then the adversary must proceed exactly as described there, in particular, diffuse an appropriate chain right before round r starts, in order to prevent a quorum purely made out of honest votes. On the other hande, if $\alpha \geq 0.25$, preventing an honest quorum is trivial, see appendix A.1.1, so the adversary also has the option to diffuse the better chain even after r started.

b) The round length `perasRoundSlots` must not be too long in relation to `perasBoost`, i.e.

$$\texttt{perasBoost} \geq \phi(1-\alpha) \cdot \texttt{perasRoundSlots}\ .$$

   Otherwise, the honest chain built during round r might have more weight than the block B (plus additional adversarial blocks on top of B) despite not having a boost. The attacker can still execute a less effective version of the attack by diffusing their votes signifcantly *before* the end of round r.

However, for realistic/useful Cardano mainnet parameters, such as $\texttt{perasRoundSlots} = 90$ and $\texttt{perasBoost} = 15$, this is not a problem for the adversary.

c) Once an adversary successfully performed the attack in a round, they can repeat it with good probability like this. The idea is for the adversary to mint two blocks B_1, B_2 on top of B between slot $s - \texttt{perasBlockMinSlots}$ and $s + \texttt{perasRoundSlots} - \texttt{perasBlockMinSlots}$, where B_1 is preferrable to B_2. Then, they execute an honest vote splitting attack between B_1 and B_2 for round $r + 1$, such that neither block reaches quorum just due to honest votes, but such that B_2 votes having weight $\texttt{perasQuorum} - \alpha$ at least. This is exactly the necessary setup to continue with the attack.

The number of elections of a party with stake σ within n slots is given by

$$E_{n,\sigma} \sim \text{Pois}(-n\sigma \log(1 - \texttt{asc}))$$

in the limiting case when the stake is distributed across infinitely many stake pools. Therefore, the probability that the adversary gets two elections in the aforementioned slot interval of size $\texttt{perasRoundSlots}$ is bounded by $P(E_{\texttt{perasRoundSlots},\alpha} \geq 2)$. For example, for $\texttt{perasRoundSlots} = 90$ and $\alpha = 0.4$, this evaluates to 55.09%.

The number of successive successful rounds is hence geometrically distributed. Note that all blocks added to the chain in the meantime are adversarial, hence impacting chain quality.[3]

These probabilities seem sufficiently large to us to take this attack seriously. A more detailed analysis (potentially simulating the resulting Markov chain) is out-of-scope for this document.

A.2 Attacks motivating aspects of the Peras design

Here, we list attacks that motivate (and are hence prevented by) certain rules of the Peras design.

A.2.1 Attack on a variant of the block creation rule

Peras enters a *cooldown period* when a round does not give rise to a certificate. In order to coordinate the *end* of the cooldown period, a certificate is included on chain.

When an honest node is elected in a slot in round r, it includes the latest certificate $\texttt{cert}'$ it has seen if and only if all of the following hold:

a) The node has not seen a certificate $\texttt{cert}$ with $\text{round}(\texttt{cert}) = r - 2$.

b) $r - \text{round}(\texttt{cert}') \leq \texttt{perasCertMaxSlots}$.

c) $\text{round}(\texttt{cert}') > \text{round}(\texttt{cert}^*)$.

Here, rule c) makes sure that an honest node never includes a certificate that has already been included in our current selection. Rule b) allows us to disregard votes/certificates beyond a certain age.

Rule a) makes sure that all honest nodes have stopped voting, preventing useless certificate inclusions. Concretely, rule a) implies that no honest node voted in round $r - 1$, assuming that round $r - 3$ was successful (i.e. we were not in a cooldown phase, so the voting rule (VR-2A) doesn't apply in round $r - 1$).

[3] However, such reductions in chain quality are not necessarily too unexpected in the presence of such strong adversaries, and the situation might overall still be better than with pure Praos.

To see this, we consider the contrapositive, i.e. assume that an honest node *did* vote during round $r-1$. Due to voting rule (VR-1A), it must have observed a certificate for round $r-2$ at the beginning of round $r-1$. However, as `perasRoundSlots` $\geq$ `perasDelta`, the votes for that certificate (including adversarial ones) must have been diffused to all honest nodes before round r. Therefore, rule a) is not satisfied for any honest node during round r. □

In contrast, if we were to modify rule a) to be about the absence of a certificate in round $r-1$, then an adversary could force nodes to unnecessarily include a certificate on chain. Concretely, the adversary can diffuse its votes shortly before the start of round r, such that some honest nodes see a quorum for round $r-1$, while others do not. If the latter category of nodes is sufficiently small, then them not voting during round r does not preclude the possibility of round r being successful, in which case they will vote again in round r as normal. However, the modified rule a) would still force them to include the most recent certificate on chain when they are elected before they receive the adversarial votes for round $r-1$, which we want to avoid.

It may be useful to clarify that the harm of honest nodes unnecessarily including a certificate on chain is not necessarily the presence of the certificate itself. Note that an adversary can include a certificate in any block they mint, for example, with the only risk being reputational harm. Instead, the harm done by honest nodes including unnecessary certificates in the honest blocks they mint is that the limit on block size means the bytes occupied by the certificate could have otherwise been occupied by transactions.

A.2.2 Adding weight to Genesis density comparisons

The implementation approach of Ouroboros Genesis in the Cardano node fundamentally relies on the following property, justified by the analysis of the Genesis chain selection rule in [Bad+18]:

Density of Competing Chains

Let p be any historical point on the honest chain. The honest chain of a net that has always executed Praos under nominal conditions will have strictly higher density in the s_{gen} slots immediately following p than any alternative chain that intersects at p.

Here, the *density* of a chain in a range of slots is defined to be the number of blocks in that range.

With Peras, it is natural to modify this property to talk about the *weight* in the s_{gen} slots instead of just the number of blocks:

Weighted Density of Competing Chains

Let p be any historical point on the honest chain. The honest chain of a net that has always executed Peras under nominal conditions will have strictly higher weight in the s_{gen} slots immediately following p than any alternative chain that intersects at p.

In the implementation, relying on this rule instead of Density of Competing Chains results in additional complexity and operational costs:

- A syncing node must download certificates in order to perform Genesis density comparisons. This requires modifications to the existing Genesis logic, which currently gets by with looking purely at header chains.

- Nodes need to store certificates boosting blocks on the historical/immutable chain stored indefinitely, such that they can be given to syncing peers. This increases disk and outbound bandwidth requirements.

However, this complexity is necessary, as appendix A.1.3 describes an attack that would be possible if we were to keep using Density of Competing Chains instead of implementing Weighted Density of Competing Chains. Concretely, the adversary can use the attack to let the honest chain to have less than $\phi(\alpha) \cdot s_{\mathsf{gen}}$ blocks (i.e. the average number of blocks on a chain they can create completely by themselves) in a window of s_{gen} slots, despite having higher weight. Once successful, the adversary can cause syncing nodes to commit to the adversarial chain permanently, violating safety.

A.2.3 Spamming equivocating votes

The adversary can equivocate any of their seats in the voting committee of a Peras round by creating more than one vote for each seat, voting for different blocks. An adversary with $\alpha < 1/2$ stake can not use this to cause a quorum for different blocks in the same round due to the choice of `perasQuorum` $= 3/4$ and a quorum intersection argument.

However, in a naive implementation, the adversary can use equivocating votes to cause unbounded additional network traffic for honest nodes. Most directly, they can send many equivocating votes to any of their peers individually. If this is disallowed (by requiring nodes to only forward the first vote per voting committee seat per round), the adversary can diffuse different equivocating votes to different peers, such that nodes still download many different equivocating votes from their peers.

We now describe how this can be avoided; specifically, honest nodes will only ever download at most one vote (or alternatively, a bounded amount of votes) per voting committee seat per round.

An honest node downloads votes from its peers in two stages:

a) In the first stage, it requests and downloads *vote IDs* from its peers.

   The vote ID must be chosen as to uniquely identify a voting committee seat in a particular round. Concretely, it can be represented as the round number and a proof of eligibility, which might be the stake pool identity or a VRF proof.

   Also, we assume vote IDs to be significantly smaller than full votes.

b) In the second stage, the node downloads votes corresponding to all vote IDs *without duplicates*. Upon receiving a vote, it is checked that the vote matches the supplied vote ID.

   This way, observing equivocating votes is impossible.

As an optimization (for example to reduce the impact of slow peers), the same vote ID could also be requested from multiple peers (bounded by a small constant) in b). While possible and sound, it is not necessary to discard equivocating votes received this way.

Additionally, this approach requires us to diffuse of certificates in addition to votes in general (even between caught-up peers). To see why, consider two honest nodes H_1, H_2 and an adversarial node A which are pairwise connected. Suppose that we are in round r, both H_1 and H_2 are only one vote short of a quorum for the block B in round r, and all honest votes for round r have already been diffused. Now the adversary sends H_1 and H_2 a new vote ID `vid`, and then equivocates `vid` to send a vote v_B for B to A, but a vote $v_{B'}$ for a block $B' \neq B$ to A'. Now A observes a quorum for B in round r, creating a certificate, while A' does not. The vote diffusion mechanism described above causes A' to

not request a vote for `vid` from A, because A' already has received $v_{B'}$.[4] Overall, the honest nodes A and A' now disagree whether round r was successful, which they would not have if they had exchanged all equivocating votes.

However, by letting nodes also exchange *certificates* in addition to votes, this scenario can be avoided. Concretely, A' would ask A for a certificate for round r, and then download the certificate for B. It does not matter that the certificate was built using an equivocating vote; in general, depending on the cryptographic scheme used for building certificates, it can even be impossible to detect this. By construction, there can be at most one certificate per round, so there is no risk of equivocating certificates.

[4]Potentially, if A' implements the optimization mentioned above, i.e. to download votes for the same vote id from multiple peers opportunistically, they might download v_B from A, but this is not something we can rely on in general.