Introduction to the design of the Data Diffusion and Networking for Cardano Shelley*

AN IOHK TECHNICAL REPORT

Duncan Coutts
duncan@well-typed.com
duncan.coutts@iohk.io

Neil Davies
neil.davies@pnsol.com
neil.davies@iohk.io

Marcin Szamotulski
marcin.szamotulski@iohk.io

Peter Thompson
peter.thompson@pnsol.com
peter.thompson@iohk.io

Version 1.9, August 2020

Contents

*Supplemented by a Part 2 which is a Reference Document aimed at implementers of Shelly. https://input-output-hk.github.io/ouroboros-network/pdfs/network-spec

List of Tables

1 Revision History

Version	Date	Change	Reason
1.5	2019-10-14	Review	Internal document shared with selected reviewers.
1.6	2020-05-21	Revised	Preparation for external publication.
1.7	2020-05-28	Revised	Edits to remove irrelevant and redundant material.
1.8	2020-08-05	Revised	Resolving remaining points and correcting headings.
1.9	2020-08-09	Format	Converted from Google doc to LaTeX (via pandoc).

2 Executive Summary

Purpose of document. We explain the technical requirements and key constraints for the networking layer of the Cardano Shelley implementation of Ouroboros Praos. The key functional role of this layer is the data diffusion of blocks, transactions and(and other information to be

shared) in Cardano Shelly. The general design was created to meet the likely needs of highly distributed, publicly facing PoS blockchain system in general.

The technical requirements are expressed in terms of the key business requirements provided at the project's outset then refined with the IOHK business team.

We give design requirements, compare with existing protocols, and describe the rationale for the solution adopted by the IOHK Networking Team.

Who the original was for. It was intended to provide a semi-technical, high-level overview of the Cardano Shelley networking layer to inform strategic decision making and to provide confidence in the proposed networking implementation. This is supplemented with a detailed technical analysis explaining the design and implementation to the Engineering and Quality Assurance team.

Where does it fit in the overall documentation scheme (dependencies, technical/management etc)? We build and expand on the design requirements from the Ouroboros Praos paper. It links with the Shelley Ledger specification.

The networking solution is being deployed as part of the Shelley *mainnet* implementation, linking with the *ledger, wallet* and other implementations. Design decisions will also be used to inform the future *testnet* implementations, as appropriate to ensure the necessary rapid deployment. The document will be used to derive information for subsequent testing and deployment.

Achievements and Status. The business requirements necessitated a bespoke protocol design, compatible with hard real-time concerns, security concerns, and a globally distributed network. This is a novel, technically complex issue. The resulting implementation is carefully designed but relatively simple. It has been validated, substantially implemented and tested.

3 Technical Summary and Motivation for the New Design

The networking requirements for a Proof-of-Stake (PoS) system, such as Ouroboros, differ from those of a Proof-of-Work (PoW) system, such as Bitcoin or Ethereum.

In PoW, adversaries must expend hashing power, which honest players can check in a cheap and stateless way. Honest nodes in a PoS system lack this advantage, which tips the balance towards the attacker, so it is necessary to mitigate this with careful design. Notably, off-the-shelf overlay network solutions tend not to consider such attacks. Existing solutions have been assessed, but the cost of retrofitting them to address the new balance of power between the adversary and honest players above is easy to underestimate and they would require substantial modification which would negate any advantages of code reuse. This is a major reason we undertook a from-scratch development.

Our design is simple and modular and interleaves the three parts of the Ouroboros distributed consensus algorithm: chain transmission, validation and selection. The design uses a stateful chain synchronisation protocol[1], both for the node-to-node case and for the case of local IPC with wallets/explorers etc. The design is extensible for future flexibility, and has a plug-in consensus algorithm to support the anticipated evolution of Ouroboros. It includes a network protocol framework that currently is used for three application-level protocols. Protocols can be added or changed for future requirements. Protocols can be pipelined to enable effective use of network resources (all the node-to-node protocols are fully pipelined), and have been designed for decentralisation. The framework is reused for local client IPC, such as the explorer backend.

[1]We follow a stream processing paradigm adapted for the eventual finality behaviour of blockchain systems.

The implementation has been kept simple to minimise complexity and attack surface. It is fully integrated with the rest of the node and is working in the Shelly mainnet. At the time of writing only one component has not yet been deployed, namely the peer selection and subscription management (outline design in Section 5.7).

4 Introduction

A blockchain is a mechanism for computing *distributed consensus* on a global scale, i.e. coming to global agreement on a sequence of events, in the presence of 'bad actors' trying to bias or disrupt the system, and without a centralised authority.

Distributed consensus algorithms fall into two main camps: proof-of-work (PoW) and proof-of-stake (PoS).

- *Proof-of-work* algorithms, like Bitcoin's, are a race to 'mine' blocks. The winner gets to add a block on the blockchain, and collects rewards for doing so.
  Miners consume vast computational resources as they compete on a global scale.
- *Proof-of-stake* algorithms take turns to produce blocks in proportions that are determined by the users' stakes in the system.
  Blocks can be produced efficiently and frequently, improving throughput and settling time[2], but at the risk of creating forks if a previous block fails to reach the next block producer in time.

Cardano is a third-generation blockchain[3] based on a new proof-of-stake consensus algorithm called Ouroboros [BGKR17, BGK+19]. Cardano's first implementation, Byron, was introduced in September 2017.

Shelley is a full, modular, reimplementation of Cardano, organised into two main functional components:

1. the *data diffusion* layer, which governs how data propagates between Cardano nodes, and
2. the *consensus* layer, which implements the Ouroboros algorithm and applies the ledger rules.

Cardano is a distributed algorithm. Consequently in the Shelley implementation each network node runs a Shelley *instance*, whose functions interact via the data diffusion component.

This document describes, for the Shelley implementation of Cardano:

1. the goals and constraints of the data diffusion component,
2. the design decisions that were made in meeting these goals and constraints, and
3. why those particular decisions were made.

A note on terminology: while "data diffusion" and "network" mean slightly different things – technically, data diffuses *across* a network – for simplicity we may use these terms interchangeably in this document.

[2] The amount of time that must pass for the consensus to be considered immutable.

[3] Bitcoin was the first generation. Ethereum is an example of a second generation blockchain.

4.1 Structure of the document

This document is structured as follows:

1. In Section 5 we describe the overall implementation of the Ouroboros algorithm, and summarise the key constraints and design decisions, in particular how validation and forwarding have to be interleaved in order to deliver a robust implementation.
2. In Section 6 we discuss how the constraints of the physical world impact on a distributed ledger system and also the implications of running an autonomous collaborative algorithm with hard real-time constraints on a global-scale. We consider the threats that such a system faces because it is global and open, including the possibility of *adversarial nodes* within the system and the vulnerability of being on the global Internet, such as exposure to DDoS attacks.
3. In Section 7 we consider a range of published approaches to transferring information between nodes and the extent to which they are able to meet the constraints and deal with the threats that were discussed in Section 6.
4. In Section 8 we enumerate the operational environment and the constraints that are derived from the business requirements from Section 11.1.
5. In Section 9 we discuss our development approach and design decisions.
6. In Section 10 we look at currently unresolved issues.
7. In the appendix we provide: the previously-agreed business requirements (Section 11.1); numerical models of TCP performance (Section 11.2), the network scaling (Section 11.3), and the leadership distribution in Ouroboros Praos (Section 11.4); and give a detailed description of the software components that are required by our solution (Section 11.7).

5 Overview

This document concerns the requirements, constraints and design decisions of the network layer of Cardano. However, the network layer design is fundamentally shaped by

- the constraints and design of the Cardano consensus layer, and
- keeping extensibility in mind for the product roadmap.

Thus far, the network parts exist almost exclusively to serve the needs of the consensus implementation[4]. So to understand the design constraints and design decisions for the network layer implementation we must first consider the requirements, constraints and design decisions of the consensus layer implementation.

5.1 Consensus constraints and design decisions

Cardano as a cryptocurrency system fundamentally relies on an implementation of Ouroboros [BGKR17, BGK+19]. The underlying properties of Ouroboros[5] ensure, *by design*, that Cardano users can submit valid transactions and can rely on them being incorporated into the *immutable* prefix of the cryptocurrency ledger. This is despite the system being open (accessible to those with minimal or no stake) where adversaries may attempt to steal money or cripple the system,

[4] Although they have been written in a modular way to facilitate extension and re-use.

[5] Specifically, liveness and persistence – see below.

and operating over the public Internet where the network load, bandwidth and latency may be highly variable and unpredictable.

It follows that a key part of the commercial offering is that Cardano, being based on an implementation of Ouroboros, can actually achieve these goals and so provide a sound underpinning of the cryptocurrency, while operating on the public internet network with a public IP and thus fully exposed to bad actors.

The key requirements for the Ouroboros implementation come from the Ouroboros specification as embodied in the published and peer-reviewed papers, and from business requirements (see Section 11.1) including quantitative non-functional requirements.

There are three levels of refinement:

1. The underpinning academic papers (e.g. [BGKR17, BGK+19]) that give a *mathematical description* of the algorithm;
2. An *implementation design* that:
   (a) Can be shown equivalent to the mathematical description (informally at first, formal proof to follow as and when time permits);
   (b) Can be implemented in a real-world distributed computational setting;
   (c) Does not introduce new possibilities – with respect to the mathematical description – for attackers to disrupt or subvert the system, or at least includes mitigations for any vulnerabilities that it does introduce;
3. A *software implementation* that follows from the implementation design;
   (a) The correspondence between the two being checked through tests, property checking etc.;
   (b) That is efficient, modular and easy to maintain; and
   (c) That does not introduce any further possibilities – with respect to the implementation design – for attackers to disrupt or subvert the system through refinements of or changes to the design, or at least includes mitigations for any that it does introduce.

This section focuses on the first refinement step, from the mathematical description to the implementation design.

The mathematical description of the Ouroboros algorithm is not intended to be a directly implementable design. For example, transmitting entire blockchains is impractical, but is a valid specification of an outcome to be achieved by a practical design. Substantial work was required to refine the mathematical description to an implementation design.

Ouroboros establishes strong properties of *progress* (liveness) and *persistence*[6] , based on surprisingly weak assumptions [BGK+19, Section 1], and in an environment with potentially powerful adversaries. This environment combined with the capabilities of the adversaries collectively comprise the threat model (see Section 6.3).

5.1.1 Interleaving transmission and validation

The Ouroboros specification Ouroboros *as an algorithm specification* is elegantly modular, comprising:

1. Reliable chain broadcast; followed by
2. Chain validation; followed by

[6]These two properties are the underpinning of the Cardano cryptocurrency ledger.

3. Chain selection

All current members of the Ouroboros family have this same structure, which suggests using a parameterized consensus design that can be instantiated for BFT, Praos and the other variants[7] that are used in Cardano Byron and Cardano Shelley.

In the specification, the reliable chain broadcast is expected to achieve delivery within fixed deadlines. In practice this is, of course, impossible: delivery with high probability is the best that can be achieved. Failure to meet deadlines in the *typical* case or for an extended period of time is fatal to the system, but occasional failure simply eats into the "adversarial stake budget". IOHK's Ouroboros researchers have set a practical target of achieving the deadline in 95% of cases.

A simple broadcast implementation is impractical because adversarial players can use it to DoS the system with invalid chains. By deliberately over-using the broadcast mechanism, adversarial players can consume the system's limited broadcast capacity to the point where chains sent by honest players *consistently* fail to reach the next slot leader within the required deadline.

It is also impossible to make a *simple* alteration to a broadcast algorithm to filter out bad data sent by adversarial players: A simple alteration to a broadcast algorithm would require a *stateless* validation check. However, no known practical[8] stateless check can exclude invalid chains for Ouroboros. The validation check for Ouroboros relies on having a full and (nearly) up-to-date ledger state. Having a full ledger state depends on the other two pieces of Ouroboros functionality, the chain validation and chain selection. Thus, to validate new chains/blocks, we are required to run *all* of the consensus implementation. We cannot, therefore, use a straightforward modification of a broadcast algorithm.

Therefore, existing off-the-shelf implementations of the broadcasting paradigm (*pure gossip* and/or *pub/sub*) are hard to reuse; any form of broadcast in a PoS context opens up trivial DoS attacks, in which the asymmetric power of broadcast is weighted in favour of the attacker. Mitigating that problem is impossible in practice, since every end user would be susceptible to attack. At best we can use broadcast as an inspiration for the overall consensus design.

As if validation was not challenging enough, broadcast can also be used by adversaries who become (or have recently become) slot leader in order to send an unbounded number of *valid* chains. Avoiding this requires the Ouroboros chain selection and chain validation features, which can be used to reduce the number of chains broadcast to within a plausible bound that could fit within the available resources.

Consequences of PoS vs PoW A crucial difference exists between PoS and PoW at the network layer, with significant design consequences: in PoW-based systems, proof-of-work itself gives honest nodes an advantage over adversarial nodes (as listed below), and this enables system designs that are simpler and more modular. There is no such advantage for honest nodes in PoS-based systems such as Ouroboros.

In PoW systems:

- The number of different block headers with a valid PoW that can be constructed (over any given period of time) is bounded by the total available hashing power in the world. In Bitcoin for example this is one header every ten minutes on average.

[7]The new Byron-compatible consensus algorithm is Ouroboros "Permissive" BFT, a custom variation designed to handle the transition from Ouroboros Classic to Ouroboros BFT. The Shelley consensus algorithm is the sequential composition of Ouroboros Permissive BFT followed by the combination of Ouroboros Praos with a transitional BFT overlay schedule.

[8]There is of course an *impractical* stateless check, which is to validate the entire broadcast chain from the Genesis.

- The header PoW can be checked with little computational cost. This does not require any significant or recent state, only a vaguely-recent lower bound on the hashing difficulty value is needed.
- Such a cheap and simple test can be easily integrated into existing distributed algorithms such as broadcast algorithms.

By contrast, with PoS in Ouroboros:

- There is no equivalent of the PoW check that is expensive for the adversaries and cheap for the honest nodes: adversaries can create many apparently valid or actually valid candidate headers or whole chains.
- Block headers can only be fully validated with access to a very recent copy of the full ledger state, and the other preceding headers – which is not a simple stateless check.
- Having the full ledger state relies on the other two pieces of Ouroboros functionality: chain validation and chain selection.

Thus we conclude that the Ouroboros broadcast functionality cannot be implemented in isolation, and that an integrated design is required that combines all three areas of Ouroboros functionality: chain broadcast, validation and selection.

Bounding the resource use of honest nodes An important consideration in making a more detailed design for Ouroboros stems from the nature of the proof of the Ouroboros properties: the proof can be thought of as starting with

"For all possible actions of the adversaries . . . "

As we refine our design by making more detailed and realistic the mechanisms by which (honest and adversarial) nodes interact, so we also expand the threat model, because the more detailed design gives adversaries more ways to interact with the honest nodes.

For example, a single abstract broadcast interaction may be replaced by a collection of RPC interactions running atop layers of lower level protocols, each with their own complex interactions. We discussed in Section 5.1.1 the problems that arise when the resource use of honest nodes is under the control of adversaries; this too can only get worse during design refinement or implementation, as we provide new mechanisms to interact.

Thus, a fundamental constraint when evaluating consensus designs is to find one with a plausible argument for bounding the resource use of honest nodes, given *all* the possible actions of the adversary that are enabled by the design – otherwise, the security guarantees of the mathematical consensus algorithm are negated. This is a high bar – even with the caveats of "plausible" and "informal". This contrasts with the orthodox approach, which only solves the simpler problem of solving or mitigating against known classes of attack[9].

After much effort evaluating various designs that incorporated some aspect of broadcast (e.g. broadcast of block headers), we were unable to find a design for which we could make even an informal resource bounds argument. The designs we studied also tended to be excessively complex.

In abandoning the notion of broadcast we move from a situation in which a participating node receives information from *all* other nodes to one in which it receives information from only *some* of the other nodes, which we refer to as its 'peers'. Nodes are then connected in a graph through which information flows.

[9]For example, much of the critique of Kademlia (ours in Section 7.2 and others' [MHG18]) comes down to the fact that it followed the orthodox design approach. It does not start with a threat model and establish positive results. Many academic papers on Kademlia establish negative results: they find flaws and then fix or mitigate them. We must however suspect that there are more flaws to be found, because none establish a positive result.

Our final design thus interleaves (in a sense made more precise below) the three parts of the Ouroboros algorithm functionality:

- chain transmission,
- chain validation and
- chain selection.

These three parts are used at every node in the graph that indirectly connects all (or at least most) nodes. This design turns out to be relatively simple and could plausibly be refined further to (at least informally) establish resource bounds.

Our design interleaves Ouroboros functionality in the following sense: along any path through the graph of nodes, there will be multiple rounds of chain transmission, validation and selection, with one round per hop in the path. After the first design refinement[10] step, our high level algorithm is as follows:

- each node transmits its current chain to its immediate neighbours;
- each node validates the incoming chain from each neighbour;
- each node does chain selection on the candidate chains;
- the selected chain becomes the node's new current chain, ready to be transmitted to its neighbours again; and
- when a node is a slot leader, it extends its own current chain with the new block.

By contrast, recall that in the original mathematical description of the algorithm, the chains are broadcast to *all* nodes, and *then* each node locally performs chain validation and selection over all the chains that it receives.

It is worth noting again that this design is not 'broadcast' in the classical sense, and hence, as discussed earlier in Section 5.1.1, existing broadcast implementations (including multicast and pub/sub) cannot easily be adapted to provide an implementation of this design. It is nevertheless a relatively simple design and can be implemented directly, with some further essential refinement.

5.1.2 Block/body splitting

An essential and uncontroversial design refinement in any blockchain implementation is to separate block headers and block bodies:

- If blocks can be almost fully validated in O(1) time based on looking at only a small fixed size block header, then honest nodes can validate candidate chains with a small bounded amount of work.
- It also enables a design where a node can see blocks available from many immediate peers but can choose to download each block body of interest just once (from a peer of its choosing from which it is available). This saves network bandwidth[11].

[10] This design is an intermediate design refinement. Like the high level Ouroboros algorithm, it still uses elements that are not directly implementable (like transmitting whole chains). Further refinements will attain directly implementable design. This refinement process is a standard technique for modular design, allowing design options to be considered in relative isolation. This modularity is especially helpful if and when we apply formal methods to prove correctness of low level designs with respect to high level specifications.

[11] We also exploit this to choose a peer that we have reason to believe will deliver the block soonest.

In the case of Ouroboros, we can pack all the cryptographic consensus evidence into the block header, leaving the block body containing only the ledger data, and check that the block has been signed by a node that is the slot leader. If we validate this in the context of a chain of headers[12], then we can establish this is a plausible candidate chain, thus we eliminate several potential resource draining attacks.

So the design at this stage involves transmitting chains of headers rather than whole blocks, and using a secondary mechanism to download block bodies of interest[13].

5.1.3 Stateful chain-following

The next design decision revolves around how to implement the transmission of chains of headers in a resource-bounded way.

The general approach is to take advantage of the fact that we do not need to transmit whole chains if most headers are already on the destination node. Thus, we need a way to *synchronise* chains of headers, and interleave receiving headers and validating them. The interleaving resists asymmetric resource attacks, by minimising and bounding the resources expended before discovering an invalid header.

Alternatives considered included chasing chains from the tip, or establishing an intersection and chasing from there. The chosen design is a *connection-oriented* application-level protocol where the producer side[14] keeps track of the intersection point between the producer's chain and the consumer's copy of the same chain. Consumer and producer can operate in bounded resources and the consumer can progress even with concurrent forks in the chain – again considering the power of the adversary.

This protocol[15] is application-level in the sense that it is part of the consensus application-level logic and relies only on exchanging an ordered sequence of messages. It is simple in the sense that the consumer can follow a very simple algorithm using the protocol to achieve its ends of keeping in sync with the producer's chain.

Alternative high-level protocols for chain synchronisation were considered but none had a better combination of simplicity, efficiency and a clear argument for working in bounded resources. In particular "stateless" versions of chain synchronisation are possible but are either more complex or are less efficient in both the typical case and adversarial cases. The ones that have been considered also suffered from asymmetric resource attacks that can be avoided in the stateful version. See Section 9.1 for more details.

It may appear unfortunate that we have chosen a stateful protocol since stateless protocols enjoy wider support in existing protocol frameworks such as HTTP implementations[16]. Adding the required support on top of an existing stateless protocol amounts to making it stateful e.g. by adding the concept of a session. It is worth noting that stateless protocols can only support arguments for worst case resource bounds whereas stateful protocols can also support arguments for worst case *amortised* resource bounds. It is often easier to find effective algorithms that have amortised bounds. In the final design we rely on amortised bounds for two of the three consensus protocols.

It is perhaps not surprising that a stateful protocol is a natural fit. A blockchain-based distributed information system is close to a distributed information system based on the modern paradigm of event sourcing, or event stream processing. Architectures based on event sourcing tend to use a stateful or connection-oriented network protocol to join and then incrementally consume the sequence of events. This involves the event stream server maintaining state to

[12] We also need the ledger state, and there are some other technical constraints.

[13] Blocks can be fetched from any peer who announces it, or from multiple sources if desired.

[14] The 'producer' is the side that has new data, and the 'consumer' is the side that may want it.

[15] Meaning an algorithm where two parties maintain local state and exchange information.

[16] Of course TCP is a connection-oriented reliable sequential protocol, which supports all manner of higher level stateful protocols. It is undoubtedly the most widely deployed, supported and used protocol on the public internet.

know where in the sequence the consumer is. This allows it to send exactly the right events, and to do so promptly as soon as new events arrive. This architecture is very similar to a design for a consensus node. The "events" are the blocks. The consumer applications find their point on the chain and incrementally move forward, maintaining local state[17] and reacting as appropriate. The key difference between normal event sourcing and consuming a blockchain is that normal event sourcing has *immediate* finality of events, whereas blockchain algorithms such as Ouroboros have *eventual* finality, meaning that there are forks that consumers have to follow. A stateful protocol for consuming a sequence of events can be adapted for eventual finality relatively easily. The simplest possible version of the chain synchronisation protocol is essentially this, and our final version is not significantly more complicated.

5.1.4 Storage subsystem

This is not immediately obviously connected to the network design decisions, but there are some areas of contact.

A design decision with wider consequences that is motivated by the storage subsystem is to identify *points* on the blockchain not by their block hash but by the *pair* of the block's slot number and the block hash.

This helps with the design of the chain storage subsystem. The block points are ordered by time in the same order as the chain and can be used as a physical pointer to near to where the data can be found whereas block hashes have no useful order. This eliminates the need to maintain a massive mutable index of all block hashes. This in turn allows the entire storage system to be designed without a traditional database and to use only simple file operations, and in particular to exclusively use immutable append-only files. This has significant I/O performance and reliability benefits, however the details are out of scope for this document.

A consequence of using points to identify blocks internally is that the network protocols must also use points when referring to blocks, such as in the chain synchronisation and fetching of blocks. This might sound like leaking an implementation detail but it in fact has advantages for the network layer too. It gives us the property that points can be ordered in the same order as the blocks appear in the chain. We take advantage of this, in particular in the implementation of in-memory fragments[18] of chain headers where it allows for a simple and efficient representation.

5.2 Consensus components

In addition to chain synchronisation, the consensus implementation needs to:

- download block bodies;
- validate new transactions;
- forward transactions towards block producing nodes; and
- use chain synchronisation, block download and transaction submission with multiple upstream and downstream peers at once.

None of these interactions needs to be synchronised with any of the others. The high level design can support a fully asynchronous implementation approach.

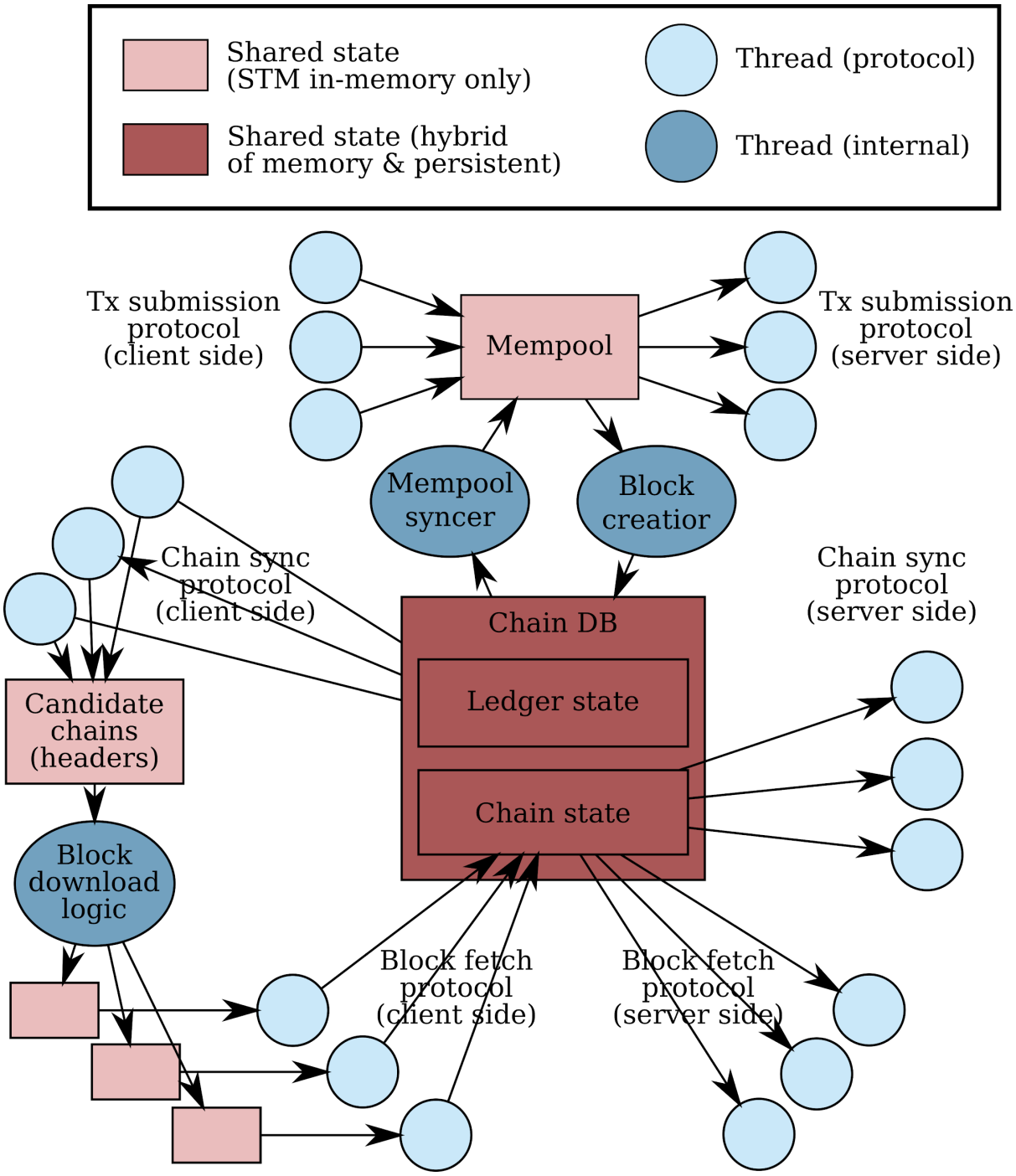
The high level consensus design following the various considerations above consists of the following components:

[17] Ledger state in the case of a validating consensus node, or other application-specific state in the case of other applications such as wallets.

[18] The in-memory chain fragment data structure is used throughout the network and consensus layers. It is briefly covered in Section 11.7.3. The implementation represents the sequence of headers using a finger tree data structure that is instantiated for block headers such as to offer efficient O(log n) operations based on slot numbers and block numbers, in addition to the normal efficient sequence operations.

- The *chain database* component wraps several other sub-components and covers several areas of consensus functionality:
  - It stores the node's current chain and the corresponding ledger state.
  - It manages the on-disk persistence of both pieces of state.
  - It maintains a set of blocks that have been recently downloaded but do not yet link onto the current chain, or are part of a recent fork that is not the current chain.
  - It performs the final stages of block validation: validating the contents of blocks according to the ledger rules.
  - It performs the final chain selection and adoption of a new chain as the current chain, keeping the current ledger state in sync.
  - It provides read access to the current chain and ledger state.
  - It provides a method to add new blocks to the database, which also triggers block body validation and chain selection.
- The *chain sync client* component engages in the consumer side of the chain synchronisation protocol to continuously get the candidate chains of headers from the immediate upstream peers, interleaved with chain header validation.
- The *block fetch* component selects plausible candidate chains based on the available valid chains of headers, and decides which block bodies to download from which peers (the *block fetch logic*). It uses the block fetch mechanism for downloading the selected blocks from the selected peers (the *block fetch client*). The implementation of this component sits within the network layer package.
- Two simple components, the *chain sync server* and *block fetch server,* provide the producer side of the chain synchronisation and block fetching. These draw their data from the chain database and allow downstream peers to synchronise a copy of the node's current chain.
- The *mempool* component stores and manages a set of valid pending transactions. It deals with keeping the mempool in sync with the current ledger state. It also deals with validating new transactions that are added to the mempool.
- The *tx-submission client* component consumes transactions from the mempools of downstream peers and tries to add them to the local mempool. This enables downstream peers to submit their transactions to this node. It deals efficiently with the common case that the same transaction is available from many peers.
- The *tx-submission server* is a simple component to provide the producer side of the transaction submission system. This enables the node to submit transactions to other upstream peers.
- A *block forge* component creates new blocks when the node becomes the slot leader.

Each of these components is illustrated in the diagram below, which illustrates the key data flows between the components. (The diagram also distinguishes between passive state and active threads that interact with the state, but this distinction is unimportant for the high level summary.)



The validation and chain selection functionalities of Ouroboros are in four separate places:

1. Peer chain header validation is performed in the chain sync client. This covers all the Ouroboros chain validation protocol checks, but none of the checks on the block bodies.

2. Initial selection of plausible chains takes place in the block fetch logic. It considers all the valid chains of headers maintained by the chain sync clients, and looks for ones that are longer than the current chain. It then picks one or more chains to download block bodies of. It picks on the basis of a combination of prioritising the longest chain, the available network resources and the expected ability to complete within deadlines. This should be considered part of chain selection.

3. The evidence connecting the block header to the block body is checked by the block fetch client upon completing the download of the block body.

4. The final block body (ledger) validation and chain selection are performed in a single integrated algorithm in the chain database.

The fact that these are split up is deliberate. The checks are moved to the earliest point in the operational cycle where the data is available, and where the least resources have been expended. This helps build an argument about bounded resource use and preventing asymmetric resource attacks.

Some parts of the network layer are thus inherently specific to the requirements of the consensus design[19], a trade-off that is forced upon us by operating in an environment where the adversary has so many cheap options[20] but the other parts are as general as reasonably possible. This provides more modularity and flexibility and makes the code amenable to rigorous component level testing and it also simplifies the review process (whether internal or external).

5.3 Network constraints and design decisions

There is no clear line between the consensus and network layers of an Ouroboros design. After all, Ouroboros itself is a high-level message-based networked protocol. But generally speaking, the consensus layer provides the *policy* and the network layer provides the *mechanisms*:

- The consensus layer handles issues deriving from the Ouroboros algorithm, whereas
- the network layer handles issues of effective use of network resources.

Both layers must be concerned with adversarial behaviour and asymmetric resource attacks. The block fetch logic component is an exception in that it has aspects of both layers: implementing parts of chain selection, but also effective use of network resources.

The final high level consensus design has information exchange requirements on three areas:

1. *chain sync*: a requirement to support a stateful application-level protocol for synchronising chains of headers;
2. *block fetch*: a requirement for downloading block bodies; and
3. *tx submission*: a requirement for transmitting new transactions from the mempool in one node to that of another.

5.3.1 Stateful versus stateless protocols

Requirement 1 above (to support a stateful connection-oriented application-level protocol) implies some notion of connection or session.

This is straightforward by layering on top of a lower level connection-oriented protocol, or as noted above, we could also add a notion of session to a stateless protocol (at a price of more complex resource management).

This design decision is discussed in Section 9.1. The summary is that there are not as many off-the-shelf solutions available as it might first appear, since most protocol implementations are designed for use within a data centre (thus within a controlled operational environment) and not in the adversarial environment of the public Internet. There does, however, exist a good stateful option: TCP; and there is also a good stateless option: HTTP.

TCP and HTTP are common and well-supported, with robust and reliable software libraries and tools. In terms of lines of code and maintenance cost, there is little difference between

[19]Though the implementation is highly parametric and can support the different protocol and ledger checks needed to implement many instances of the Ouroboros family.

[20]Again, contrast this with PoW where to get any higher than low-level network attacks using blocks, the adversary must provide a PoW which immediately shifts the resource argument in favour of the defenders.

approaches based on these two protocols. Our chosen solution has two components layered on top of TCP, each of which is less than 1000 lines of code.

To get to a roughly equivalent level of functionality with an alternative solution (based on exclusively stateless application-level protocols) on HTTP would likely involve picking existing libraries such as "servant", that are built on top of the "wai" and "warp" HTTP server stack. That would indeed save 2k lines of code, but there would then be tens of thousands of extra lines of code to audit. There would also be much more unnecessary functionality and network-exposed surface area for attackers to exploit.

A further constraint stems from the business requirement to support block-producing nodes on consumer-grade network connections. Such nodes are usually behind firewalls. In practice this means they can only establish outbound TCP connections, because inbound TCP connections are blocked by the firewall. Protocols such as HTTP establish new TCP connections for each new batch of requests, and those connections can only be used for requests in one direction[21]. This means that nodes behind firewalls can only make HTTP requests; they cannot receive them. This places a general limitation on the design of higher level consensus protocols, or requires a special asymmetric case in the design to cope with such nodes. For example, in an RPC based design layered on top of HTTP, RPC interactions can only be initiated by the node behind the firewall. This

- introduces an asymmetry into the design, and
- makes it harder to push notifications promptly, such as block header announcements, which puts "home" stake pools at a further disadvantage in meeting the Ouroboros timing constraints.

By comparison, with a bi-directional connection-oriented protocol such as TCP, a connection can be established by the node behind the firewall but once established the connection can be used in both directions. This allows the application level protocols to be fully symmetric, which is the natural design for a peer-to-peer rather than client/server system.

There are also significant advantages to being able to use stateful application level protocols. Again, as noted above, stateful protocols can rely on an amortised analysis for their resource bounds, which simplifies the design of protocols that are (or can be made) resistant to asymmetric resource attacks. For example, in an HTTP REST API, if there is a moderately expensive operation that must be available to be used as part of normal peer interactions then it is hard to mitigate an attack where that API is used repeatedly – without that mitigation also affecting legitimate users of the API[22]. In a stateful protocol that same interaction can be tied to the previous actions of the same peer. This enables mitigations that can add extra cost to the attacker, or can slow the attacker down without slowing down other legitimate peers.

Ultimately, the choice of stateful or stateless chain synchronisation protocol is a design decision on which reasonable people can and do disagree. Maintenance costs are similar either way and both approaches allow future flexibility, but

> *this design decision in the consensus layer cannot easily be changed later*.

The stateful chain sync pattern is relied upon in many parts of the consensus design and implementation: the chain database, the peer interactions and the local client IPC. Indeed, it is a unifying aspect of the design that simplifies many aspects of the implementation.

An additional feature of adopting a contention-based, stateful association with a peer is that, for the duration of that connection, we can integrate information derived from that connection to inform local decision making.

[21] HTTP persistent connections are a performance optimisation, and cannot be used for HTTP requests in the opposite direction. HTTP2 does allow some degree of server-push, but it is not a symmetric bi-directional connection.

[22] For example through some form of rate limiting.

5.3.2 Concurrency

Recall, from the beginning of Section 5.3, that the consensus information exchange requirements cover three areas: chain sync, block fetch and tx submission. Each relationship with a direct upstream and downstream peer needs to cover all three areas, however there is no requirement for any synchronisation between them when talking to a single peer. They are independent when talking to different peers, and at least semi-independent for a single peer.

Concurrency is of course a technique to structure programs in a modular way, for those programs that need to interact with multiple external agents [Mar13, Chapter 1]. It is a natural design decision to use concurrency for talking to independent peers. The same motivations lead to the decision to handle the three areas (chain sync, block fetch and tx submission) concurrently when talking to a single peer.

Using concurrency allows each of these three features to be handled in a modular way. It results in three simple protocols rather than one complex one, and a clear way to add new protocols or upgrade existing ones. Chain sync, block fetch and transaction submission are independent application-layer protocols that all run concurrently as required by the consensus layer.

We thus end up with one Haskell thread per application-level protocol per upstream and downstream peer. In extreme cases, this could be a large number of threads but managing this is well within the capabilities of the GHC runtime system [Mar13, Chapter 15].

5.3.3 Bearer and multiplexing

We must pick some underlying lower-level *bearer* protocol over which to run the collection of application-level protocols. Having decided to support stateful application level protocols we must either pick a stateful lower-level protocol or add session or connection support to a stateless protocol. The clear and obvious choice is then to use TCP. TCP is the best supported internet protocol when it comes to the thorny issues surrounding consumer NAT and firewall hardware: outbound TCP connections can almost always get through. It would be plausible to use UDP-based options but that would either require significant additional development work[23], or it would be necessary to rely on some relatively new UDP-based protocols with immature implementations. Similarly, adding sessions to another higher level protocol such as HTTP would add complexity, but would have little benefit.

The only significant drawback of selecting TCP is that we wish to run multiple application level message-based protocols with a single peer concurrently, whereas TCP supports a *single* reliable ordered byte stream. Adapting stream-based communication to provide message-based communication just needs simple *framing*. To support concurrent protocols requires either using multiple TCP connections or else running the collection of protocols over a single TCP connection. Using multiple TCP connections requires more resources[24] and also has other disadvantages such as more complexity in fully disconnecting from a peer that is deemed to be adversarial.

The standard solution to this well-known networking problem is to use multiplexing. That is, at one end we multiplex by chopping up the data streams from the individual protocols into bounded sized chunks and interleave them on the single TCP data stream, and at the other end we demultiplex to reverse the transformation. The abstraction that is presented by the multiplexer is essentially the same as that of TCP itself: a reliable ordered byte stream. This is described in more detail in the Section 11.7.2.

[23] And take on future risk as telecommunications start deploying Carrier Grade Nat to migitage IPv4 address exhaustion.

[24] Kernel buffers, file/socket handles etc.

5.3.4 Performance

Performance is a very important consideration for the network layer, deriving from:

1. the Ouroboros timing constraints;
2. the business requirements to support certain latencies and throughputs;
3. the business requirement to support worldwide distribution; and
4. the requirement to support block-producing nodes on consumer-grade network connections.

In the context of networks, performance amounts primarily to trying to use the network resource effectively whenever possible. Time that is expended without sending or receiving data is time that can never be regained. Point-to-point connections over a network can very crudely[25] be characterised as having a certain bandwidth and latency. Network connections do not support bursts of traffic well, since they have a maximum bandwidth. Full network utilisation requires that they be used at their maximum bandwidth continuously.

Network latency poses a significant challenge to effectively utilising a network connection, and this challenge tends to leak all the way up to the application-level protocols and application design. Network latencies also vary considerably, with many orders of magnitude difference between peers within a data center versus peers on other continents (see Section 11.2). Catering for this very wide range of latencies requires careful design. Ideally, protocols should be capable of adapting to the latency that is actually experienced, rather than being targeted to a specific latency.

There are two major approaches to dealing with network latency: *batching* and *pipelining*. Batching amounts to sending a large amount of data in one logical interaction, for example requesting a multi-megabyte file over HTTP. This hides latency in the sense that the full end-to-end round trip is amortised over the amount of data moved. Batching is relatively easy to implement, though it does require changes in the application level protocols. Moreover, picking the right batching size requires some care and thus becomes an application level concern. Batching by itself cannot fully utilise a network connection on a continuous basis due to the need to wait for the end of each batch. This improves as the batch size gets bigger, but there are also negative trade-offs with large batch sizes. By contrast, a fully pipelined protocol can fully saturate a network connection. Pipelining involves sending multiple messages back to back without waiting for replies, and handling replies as they come in. The primary disadvantage of pipelining is that it tends to complicate the data flow and control flow in the application[26].

5.3.5 Binary formats

We must choose an encoding format for messages in the protocols. Choices include

- standardised text formats like JSON, and
- standardised binary formats like ASN.1, MsgPack, CBOR and those used by Thrift and ProtoBufs.

The implementation (not just the encoding format) must be robust against untrusted input from the network. Further considerations, in roughly[27] decreasing order of importance, include:

[25] A much more precise characterisation is based on ΔQ, see [TD20].

[26] Surprisingly few HTTP client applications take advantage of HTTP pipelining for example.

[27] The order is of course somewhat a matter of opinion, and it by no means implies a strict lexicographical order for evaluating different choices.

1. standardised format or not
2. stability of standard
3. availability of format documentation
4. availability of existing implementations
5. availability of existing implementations for 3rd party implementations
6. availability of existing support tools
7. having a schema language for documentation and validation
8. availability of schema tools
9. ease of forwards/backwards compatibility
10. encoding size
11. performance of encoding and decoding
12. minimising excessive code dependencies

There are several perfectly adequate choices that score well against all the criteria, including one – CBOR (RFC 7049) – that is already in use in the system as the binary format for the blockchain itself that scores well on all the criteria[28]. Reusing the same choice leads to significant savings:

- reduces dependencies,
- simplifies documentation,
- reduces the cognitive load for developers,
- reduces audit costs, and
- minimises the maintenance burden.

This is true for the present Cardano implementation and any other compatible implementation. The present implementation uses an existing Haskell CBOR implementation that has been tested and externally audited to deal robustly with untrusted input.

5.4 Network libraries and components

Following from the high-level design choices above, the network layer design consists of the following libraries and components:

- A *multiplexer* component that carries multiple concurrent reliable ordered streams over a single reliable ordered bearer such as TCP[29].
- The *typed-protocols* framework for describing and using application level protocols, with enforcement via Haskell types. This is a form of session typing. It also has direct support for pipelining. It is described in more detail in this section below.

[28]Thrift and ProtoBufs do score better on the support tools, particularly schema tools, but the CBOR schema schema language is standardised (as RFC 8610) and the validation tools are adequate and are already used in the CI tests for the new Cardano implementation.

[29]It can use any reliable ordered bearer connection.This greatly aids testing, but also adds flexibility.

- An *IO simulator* library was developed for the purpose of testing the network and consensus components. It is described in more detail in this section below.
- The three application level protocols (node to node):
  - chain sync;
  - block fetch; and
  - tx submission.

These define the mechanism but not the policy for using these protocols. They are defined using the typed-protocols framework.

- A *connection handshake* protocol used during the initial set-up of connections with peers. It covers protocol version negotiation and checking compatibility of parameters like the network magic[30]. This is also defined using the typed-protocols framework.
- A *subscription manager* component that decides when to make new outbound connections to peers, which peers to connect to, and handles the process of doing so.
- A *server* component that handles accepting and managing inbound connections from other peers.
- A *block fetch* component. This was also listed as a consensus component but because it incorporates many concerns from the network layer such as achieving good network performance it was developed by the network team and is included within the network layer package. As mentioned previously this is a relatively sophisticated concurrent component: it selects plausible candidate chains based on the available valid chains of headers, and decides which block bodies to download from which peers. It provides the policy for the block fetch protocol.
- A *local tx submission protocol* which is tailored to local clients (wallets, explorers, etc.). Since it runs in a trusted environment it can be simpler, which reduces implementation complexity for clients. It also gives immediate verbose feedback to a client application on tx validation failures.
- A *chain fragment* module that is shared with the consensus layer. It defines an in-memory data structure for dealing efficiently with sub-sequences of blockchain headers.

As described in the previous section, the multiplexer satisfies the need to run multiple independent application level protocols concurrently over a single TCP connection[31]. It is perhaps worth noting that multiplexing is not unusual: the old cardano-sl network implementation also relied on a multiplexer as part of the network-transport-tcp package. The new implementation is simpler, smaller, more modular, and a better fit for the new design requirements.

We have the requirement (Section 11.1.1) that small stakeholders with reasonable consumer-grade network connections and equipment should be able to operate a stake pool. This translates to the technical requirement that nodes behind firewalls should be able to take part without having to make manual alterations to their firewall configuration. To satisfy this, the design relies on establishing outbound TCP connections and then using the connections in a bi-directional manner.

We have the requirement to support multiple application level protocols and to bridge the gap between the reliable ordered byte stream abstraction presented by the multiplexer and the

[30] This provides early failure and feedback when nodes accidentally connect to the wrong network, e.g. testnet vs mainnet.

[31] While avoiding issues of "TCP Meltdown" (running TCP over TCP)

reliable ordered message stream required by application level protocols. We must also make effective use of network resources, and must do it all in a way that respects the constraints stemming from the adversarial environment. This collection of requirements is met by the typed-protocols library developed for the purpose. It meets the requirements and brings with it a simple conceptual framework for thinking about and describing application level protocols.

The typed-protocols library is the jewel in the crown[32] of the network layer implementation and it has already been presented at three public events[33]. It casts application level protocols in the form of state machines with the constraint that both peers agree on the state that they are in. The transitions between states correspond to messages sent between the peers, so the protocol state dictates which messages may be sent or received. A further constraint is that each protocol state is labeled with the peer that has agency to send messages in that state. Thus in each protocol state one peer may send any message (valid for the state) and the other peer must accept any valid message (valid for the state). These constraints impose some simplicity on the application level protocols and make the protocols deadlock-free by construction. The library allows encoding protocols of this form in Haskell types, and thereby enforcing at compile time that implementations of the protocols comply with the constraints implied by the protocol state machines. In combination with compiler warnings it ensures that protocols only send when they are allowed to and only messages appropriate for the state, and conversely are prepared to receive when required and will handle all the messages allowed for the state. This provides structure to help writing protocol implementations.

Performance is a major consideration for the protocol design, as discussed in the previous section. The typed-protocols library has an innovative and effective solution. Many protocols of interest, including the three consensus protocols, can be pipelined *without any change to the protocol state machine*. The library provides a way to write a protocol implementation in a pipelined way, with the exact same protocol-compliance guarantee enforced by the types. Indeed it also provides type safety for certain pipelining invariants which is immensely helpful for writing pipelined implementations. The pièce de résistance is that a pipelined client can be paired up against a "normal" server that is oblivious to pipelining with the result being a pipelined use of the protocol. This makes using pipelining pervasively a very tractable proposition, and all three consensus protocols have fully pipelined implementations.

The individual consensus protocols are each designed with a few constraints and qualities in mind. They are designed:

- to allow implementations that can work in bounded space;
- to maximise the opportunities for pipelining;
- to exclusively use consumer driven control flow, without polling;
- to be highly parametric to maximise flexibility and simplify testing;
- to otherwise minimise the complexity of protocol implementations.

The motivations for these are all obvious except perhaps the consumer driven control flow. The motivation here is again about resource use. To avoid overload, malicious or otherwise, the consumer should decide when it is ready to accept new information to process. This does not prevent prompt transmission of, for example, new headers or transactions, and no costly polling is required. This fits naturally in a stateful protocol.

[32] It is a session type framework. Session types are actively researched for more than 20 years, see [Hon93, THK94, HVK98], and various implementations exists: Mungo (Java), Scrible, or [MV11] (Erlang), and many others.

[33] Notably at the "Monadic Party" 2019 Haskell Summer School, Poznan, PL, and at the "Haskell eXchange" 2019 London.

The IO simulator was developed to aid testing. It is used in tests for network components, individual protocols, application level protocol implementations, components within the consensus layer, and the combination of the network and consensus layers as a whole. It provides deterministic execution of concurrent code. It can produce a trace of interesting execution events. This is useful because many properties of concurrent code we wish to test are expressed in terms of event traces. It provides faster than real time simulation of time delays which is helpful for testing network protocol timeouts. It will in future allow testing consensus between nodes with clocks that are not completely in sync, which is otherwise a very awkward test to set up and run automatically.

The bulk of the code in the network and consensus implementations are parameterized over the type classes from the io-classes package, which enables running *precisely the same* code in IO for production or in the IO simulator for testing. For example the consensus storage subsystem is tested this way with simulated I/O failures and file corruption.

5.5 Related work (data diffusion)

As discussed in Section 5.1.1, there are strong reasons to conclude that pure broadcast cannot be used in any Ouroboros implementation that respects the Ouroboros threat model, and reasons to conclude that modifying a broadcast algorithm with the necessary checks is not a reasonable engineering trade-off. This excludes many implementations of broadcast, including multicast and pub/sub (publish/subscribe) algorithms.

5.5.1 PolderCast

PolderCast [SvVV12] is an implementation of pub/sub. As such it is not useful for an implementation of Ouroboros data diffusion as discussed in Section 5.1.1, (namely the asymmetry in favour of adeversies). PolderCast, however, also establishes a P2P graph which is a necessary feature of a fully distributed Cardano implementation. This is discussed in summary in Section 5.8 and PolderCast is considered in more detail in Section 7.3.

5.5.2 Other blockchain systems

Significant related work is the data diffusion function of other existing (PoW) blockchain systems. As discussed in Section 5.1.1, PoW provides a balance between adversaries and honest nodes that allows such systems to use broadcast algorithms (at least for block headers) modified with a stateless verification filter. PoW systems are also typically asynchronous, unlike Ouroboros which must ensure block delivery within deadlines.

Bitcoin, Ethereum and the previous Cardano implementation are considered in more detail in Section 6.2.2.

5.5.3 Other related work

Much related work exists on the choice of lower level network protocols. HTTP and the issue of stateless protocols was discussed in Section 5.3.1 and is also discussed in Section 9.1.

Other stateful protocol layers include libraries such as ZeroMQ or servers for message queue protocols like AMQP such as RabbitMQ. Unfortunately, as discussed in Section 9.1 the widely used choices are not designed to be used on the public internet.

The event sourcing architecture is however a direct inspiration for the architecture for the Cardano node. This architectural style is designed, amongst other things, to achieve very loose coupling in distributed applications. In particular, the relationship between the node as a data source and local node clients as data consumers is directly based on this idea. The

key impediment to reusing implementations, rather than just ideas, is the fact that blockchain algorithms only provide eventual finality rather than instant finality.

UDP is a lower-level internet protocol than TCP. TCP was chosen versus UDP because TCP delivers the appropriate bearer characteristics needed for the mini-protocols, is supported in all firewalls and has tried and tested methods for dealing with IP packet fragmentation. This is however a decision that would be reasonable to revisit in future.

Establishing connections between consumer nodes that are both behind firewalls is difficult but it would enable nodes on consumer broadband to make a greater contribution to the P2P network. Further research would be required to evaluate potential solutions, but a solution may involve augmenting the system with an additional non-TCP based bearer protocol specifically for use between nodes behind firewalls.

The typed-protocols library can be considered as a limited and very lightweight form of session typing. Related work is thus the existing literature on session types. The ABCD project, in which Professor Wadler is involved, maintains a large collection of references and other resources.

5.6 Decentralisation constraints

The requirements for decentralisation come from a combination of the business requirements and technical requirements deriving from Ouroboros. The business requirements are set out in Section 11.1. The dominant constraints derive from the nature of reality and from characteristics of the Internet as it currently exists. We will review the key requirements and constraints.

Ouroboros is a system with hard real-time deadlines. They are deadlines measured in seconds, not microseconds, but they are deadlines nevertheless. The deadlines can be missed from time to time by eating into the adversarial stake budget, but consistent failure to meet the deadlines destroys the properties of the system[34]. We get to pick the exact numbers for slot times or block sizes, but whatever values we pick, we must be able to keep to the deadlines in the vast majority of cases. IOHK's Chief Scientist Aggelos Kiayias set a practical target of achieving the deadline in 95% of cases. Compare this, for example, with a PoW system such as Ethereum, which is an asynchronous system without hard deadlines. It can be, and has been, adjusted over time, with longer block times during periods of high load (see Section 6.2.2 for details). This is not an option for Ouroboros as it exists now: we must pick the slot length parameter and we have no mechanism to adjust it on short time scales in response to network events. If we pick too relaxed a target then we leave transaction throughput on the table, and too tight a target risks system collapse. Thus a clear priority is *consistency* of block delivery times. We wish to avoid a long tail[35].

The other key business requirements are:

- the network layer should be decentralised in the sense that "IOHK should be in the same position on the network as any other stakeholder with an equivalent amount of stake";
- to support certain transaction latencies and throughput rates;
- to support worldwide distribution;
- 1,000 stakepool nodes holding about 80% of the system's stake[36];
- 10,000 other small-scale private stakepool nodes; and

[34] Praos does make this better in that it becomes a gradual degradation rather than a cliff edge. This lets it cope better with variance in block delivery times, but it does not help substantially with the whole distribution being shifted.

[35] See Section 6.2.2 which touches on an analysis of the Bitcoin network and the long tail of block delivery times

[36] The original business requirement was for 100 large-scale stake pools (see Section 11.1.1) but this was later raised to 1000.

- the requirement to support private stake pools operating on consumer-grade network connections;

There is a trade-off between block size and the transmission time of a block from one slot leader to the next. Of course the bigger the blocks that can consistently be delivered within the deadline, the higher the transaction throughput. The interesting question is how well we can do within this trade-off: can we easily meet the TPS targets or is this a harder problem? If the problem is easy then we may be able to use off-the-shelf solutions that do not get the highest possible TPS but at least cover the TPS targets. If the constraints make the problem more difficult then we may require a customised solution simply to hit our TPS targets. Thus we should review the factors that may impinge on transferring blocks of various sizes across a network, since that determines the envelope within which we can adjust the block size versus transmission-time trade-off.

In general the consensus nodes must arrange themselves in a graph. It does not scale very far to have every node linked to every other node. To work in limited memory and network resources we must limit the *valency* for each node – that is each graph node's in-degree or out-degree. This means that in general a block must transit multiple hops through the graph to get from one slot leader to the next. Since we are interested in the overall transmission time between slot leaders then we must be interested in both the number of hops and the transmission time on each hop. A useful technical graph metric here is one known as the characteristic path length [Wat99]. Informally it can be thought of the typical path length, taking into account both the number of hops and the hop transmission times.

Let us consider the number of hops. The typical number of hops clearly depends on the size of the graph, the typical valency and the shape of the graph. Section 11.3 provides tables relating hop counts and valencies with maximum graph size (at least in the perfect case of a homogeneous spanning tree). This tells us that for our target of 1000 stake pools, related relay nodes, and other participants, we can expect to have a hop count of at least 5 and a valency of 5 or more. This valency appears reasonable, though in reality it cannot be homogeneous. We would hope to allow edge nodes to use a lower valency and server-class nodes to be able to support a substantially higher valency.

Let us consider the transmission time per hop. This clearly depends on the characteristics of the Internet and of TCP, the underlying transport protocol that we use. Section 11.2 presents tables of transmission delays based on a model of TCP, calibrated with measurements between AWS data centres in various regions. For larger block sizes, such as 1Mb, the transmission times for intercontinental hops become quite large, in the 0.5–2.5 second range. Transmission times in this range are a significant fraction of our overall slot length budget.

The conclusion from the per-hop transmission time numbers is that if we are to use larger block sizes then we will need to ensure that on a typical path we do not get too many intercontinental hops. This can be ensured either by keeping the overall hop count low, or by selecting hops in a way that is sensitive to their transmission delays (or of course a combination). We already know, however, that we must expect hop counts of at least five, and five intercontinental hops brings us perilously close to our overall slot time budget once one takes into account other delays such as processing time, clock skews and other factors mentioned in Section 6.2.1. So this suggests that at least for larger block sizes, we are not in the "easy" scenario since we must keep both hop count and hop transmission times under control.

How large must the blocks be to hit our TPS targets? This is a straightforward calculation. Section 8.2 has a table relating average transaction size, block size and TPS, for 20s slot times. For example, for 1kb (kilobyte) transactions, 50 TPS requires 1Mb (megabyte) blocks. In the existing Byron system the smallest practical transactions are over 350 bytes, while exchanges chafe at the 8kb transaction size limit, and new features such as metadata, multi-sig and scripts will all require larger transactions.

Once we take into account future user base growth and the expected growth in transaction

sizes then our conclusion must be that if we wish to scale to our stretch goals, or perhaps even to hit our target goals, we must use a decentralised P2P graph construction mechanism that is sensitive to the combination of path hop counts and hop transmission times. Or conversely, we can only be oblivious of hop transmission times if we have small block sizes and hence low overall throughput.

Ideally any design we pick now should be one where we can have confidence that we can meet conservative initial throughput goals, but that does not preclude scaling to more users, higher throughputs etc.

A threat that is of significant concern in existing distributed blockchain systems is that of *eclipse* attacks. An eclipse attack is where attackers try to insert themselves in the network graph in such a way as to prevent a target node from discovering the "true" blocks from the system. To prevent the target node from becoming aware that it is eclipsed, the typical approach is for the attackers to supply the target with their own valid chain.

Eclipse attacks are a relevant problem for PoW blockchain systems. The crucial difficulties for PoW systems appear to be these:

- attackers (with moderate hashing power) executing an eclipse attack can create growing – albeit slowly growing – valid chains; and
- a node does not generally know the amount or distribution of hashing power.

Ouroboros does not share these difficulties. While it remains the case that attackers (with moderate stake) attempting to execute an eclipse attack can create valid slowly-growing chains, the difference is that validating nodes inherently know the current stake distribution. The node that is the target of the attack can compare the chain it observes with the current stake distribution and gather probabilistic evidence that it is being eclipsed. Section 11.4 provides an analysis of the speed and confidence with which this can be established. The conclusion is that at least for low stake attackers the time scales are very practical. More analysis is needed for medium and high stake attackers. Our threat model dictates that we must at least handle low stake adversaries.

5.7 Decentralisation design

Analysis of "S/Kademlia" suggests that one of the main drivers for its complex design is that it attempts to establish a degree of resistance to eclipse attacks based on very weak assumptions. It appears that starting from so few assumptions makes it difficult to establish resistance to eclipse attacks. As described above, however, Ouroboros furnishes us with the ability to probabilistically detect eclipse attacks on relatively short time scales. It is also worth noting that a deliberate eclipse attack and an accidental network partition are essentially identical from the perspective of the victims. Thus a solution that can *recover* from eclipse attacks already has many of the characteristics needed to recover from accidental partitions[37].

In our use case we must build a large graph in a decentralised way, initially from nodes contributed by IOHK and later by state pool operators, and grow it to support all stake pools and users. We would like small hop counts and we would also like to avoid the danger of there being too many long hops.

A literature review did not reveal existing complete designs that could meet the combination of timing and throughput goals, when scaled up to the expected number of nodes, when globally distributed, and subject to the behaviour of TCP[38]. Selected related work is discussed in the next section, and the constraints are analysed in more detail in Section 6.2.1 and Sections 11.2 to 11.6. The literature review does however reveal useful theory.

[37]The consideration of the appropriate mitigations to large scale network disruption (e.g. BGP routing issues, undersea cable damage) including their likely time to repair etc is part of the planned work.

[38]Using TCP is not a theoretical constraint, but it is a practical engineering and development schedule constraint.

Graph theory [Wat99] tells us that small hop counts[39] (formally, the characteristic path length) can be achieved in large graphs, including random graphs. Furthermore it tells us that these graphs can be "grown" from smaller ones. This is a very helpful combination of characteristics for our use case, as we want small hop counts, and random graphs are easier to build in a decentralised way than highly structured ones, especially in the presence of failures and adversarial behaviour. To achieve the low characteristic path length, the theory requires that nodes have a high enough mean valency, and a high enough standard deviation in the valency. Under these conditions the low characteristic path length is remarkably stable – for a range of starting graph types including random graphs[40]. Typical numbers are a mean of 10 and standard deviation of 3, which give a rough range of valencies of 4–16. This would appear to be eminently achievable, using the middle of the range for edge consumer nodes, and using the upper end for server nodes.

Bearing in mind the requirements, constraints and theory, our approach is to solve the problem directly, using a single simple flexible design.

The theory tells us that we can use random graph construction, which we can do with standard *random gossip* techniques. This deals with the decentralisation requirement and the low hop count constraint. The remaining significant issues are avoiding eclipse attacks, as previously discussed, and avoiding too many long hops. Our design is to augment random gossip with two layers of control loop to address these two issues. This general design is relatively simple, and has the significant virtue that the policy for the control loops can be adjusted after initial deployment with relatively few compatibility impacts. This should enable the policy to be optimised based on real-world feedback, and feedback from simulations of scale or scenarios that are hard (or undesirable) to test in a real deployment.

Each node maintains three sets of known peer nodes:

- *cold peers* are peers that are known of but where there is no established network connection;
- *warm peers* are peers where a bearer connection is established but it used only for network measurements and is not used for any application level consensus protocols;
- *hot peers* are peers where the bearer connection is actively used for the application level consensus protocols.

Limited information is maintained for these peers, based on previous direct interactions. For cold nodes this will often be absent as there may have been no previous direct interactions. This information is comparable with "reputation" in other systems, but it should be emphasised that it is purely local and not shared with any other node. It is not shared because this is not necessary and because establishing trust in such information is difficult and would add additional complexity. The information about peers is kept persistently across node restarts, but it is always safe to re-bootstrap – as new nodes must do.

For an individual node to join the network, its bootstrapping phase starts by contacting root nodes and requesting sets of other peers, which are added to the cold peer set. It proceeds iteratively by randomly selecting other peers to contact to request more known peers. This gossip process is governed by a control loop that has a target to find and maintain a certain number of cold peers. Bootstrapping is not a special mode, rather it is just a phase for the control loop following starting with a cold peers set consisting only of the root nodes. This gossiping aspect is closely analogous to the first stage of Kademlia, but with random selection rather than selection directed towards finding peers in an artificial metric space.

The root nodes used in the bootstrapping phase are the stakepool relays published in the blockchain as part of the stakepool registration process. See the Shelley delegation design

[39] This research provides a basis for the pre-existing popular "small world" idea of six degrees of separation.

[40] Interestingly, rings are a pathological starting case

specification, Sections 3.4.4 and 4.2. As with Bitcoin, a recent snapshot of this root set must be distributed with the software.

The inner control loop governs the following activities:

- the random gossip used to discover more cold peers;
- promotion of cold peers to be warm peers;
- demotion of warm peers to cold peers;
- promotion of warm peers to hot peers; and
- demotion of hot peers to warm peers.

The inner control loop has the following goals to establish and maintain:

- enough cold peers (e.g. 1000);
- enough hot peers to meet the valency target (e.g. order of 5–20);
- enough warm peers in total (e.g. order of 10–50)
- a set of warm peers that are sufficiently diverse in terms of hop distance.
- a target churn frequency for hot/warm changes
- a target churn frequency for warm/cold changes
- a target churn frequency for cold/unknown changes

The target churn values are adjusted by the outer control loop, which we will address below.

Local static configuration can also be used to specify that certain known nodes should be selected as hot or warm peers. This allows for fixed relationships between nodes controlled by a single organisation, such as a stake pool with several relays. It also enables private peering relationships between stake pool operators and other likely deployment scenarios.

Using 5–20 hot peers is not as expensive as it might sound. Keep in mind that only block headers are sent for each peer. The block body is typically only requested once. It is also worth noting that the block body will tend to follow the shortest paths through the connectivity graph formed by the hot peer links. This is because nodes will typically request the block body from the first node that sends the block header.

While the purpose of cold and hot peers is clear, the purpose of warm peers requires further explanation. The primary purpose is to address the challenge of avoiding too many long hops in the graph. The random gossip is oblivious to hop distance. By actually connecting to a selection of peers and measuring the round trip delays[41] we can start to establish which peers are near or far. The policy for selecting which warm peers to promote to hot peers will take into account this network hop distance. The purpose of a degree of churn between cold and warm peers is, in part, to discover the network distance for more peers and enable further optimisation or adjust to changing conditions. The purpose of a degree of churn between warm and hot peers is to allow potentially better warm peers to take over from existing hot peers.

The purpose in maintaining a diversity in hop distances is to assist in recovery from network events that may disrupt established short paths, such as internet routing changes, partial loss of connectivity, or accidental formation of cliques. For example, when a physical infrastructure failure causes the short paths to a clique of nodes to be lost, if some or all of the nodes in that clique maintain other longer distance warm links then they can quickly promote them to hot

[41] Or more precisely the two unidirectional ΔQ measures

links and recover. The time to promote from warm to hot need be no more than one network round trip.

Overall, this approach follows a common pattern[42] for probabilistic search or optimisation that uses a balance of local optimisation with some elements of higher order disruption to avoid becoming trapped in some poor local optimum.

The local peer reputation information is also updated when peer connections fail. The existing implementation classifies the exceptions that cause connections to fail into three classes:

- internal node exceptions e.g. local disk corruption[43];
- network failures e.g. dropped TCP connections; and
- adversarial behaviour, e.g. a protocol violation detected by the typed-protocols layer or by the consensus layer.

In the case of adversarial behaviour the peer can immediately be demoted out of the hot, warm and cold sets. We choose not to maintain negative peer information for extended periods of time; to bound resources and due to the simplicity of Sybil attacks.

The outer control loop deals with the problem of eclipse – whether malicious or accidental – and also adjusts the behaviour of the inner control loop over longer time scales. The outer control loop controls:

- the target churn frequencies of the inner control loop for promotion/demotion between the cold/warm/hot states
- partial or total re-bootstrapping under certain circumstances

The outer control loop monitors the chain growth quality, comparing it with the stake distribution. The probability of being in a disconnected clique or being eclipsed is calculated. As this rises the control loop increases the target frequencies for the churn between the hot, warm, cold, and unknown states. In the worst case it can re-bootstrap the peer discovery entirely by clearing the hot and warm sets and resetting the cold set to the bootstrap peers.

As previously stated, this design has a lot of opportunity to tweak and tune the policies used in the two control loops. The points of interaction in this design is limited to a mechanism to request known peers, and the mechanism to change a peer connection between the warm and hot states. All the rest is governed by purely local policies and hence there is significant scope to adjust these over a series of releases without introducing significant compatibility problems.

5.8 Related work (decentralisation)

As discussed in Section 5.7, prior graph theory [Wat99] tells us that we can construct large graphs with small hop counts from a range of starting graph types. To do so however requires a high enough mean and variance for the valency, well above 2-3. The results also demonstrate that using rings as the starting graph type is dramatically worse than using random graphs for the starting graph type.

As discussed in Section 5.6, poor network topography can sometimes be tolerated provided that the block size, and hence TPS, are suitably low. For example if 32k blocks were used (similar to the typical Ethereum size[44]) then the effect of using long distance links is reduced, but even with the minimum useful transaction size we would hit only 4.5 TPS, which falls well short of the minimum TPS target from the business requirements of 8 TPS.

[42] Simulated annealing for example

[43] The recovery from detected disk corruption involves resetting to a known-good state, which involves shutting down connections to all peers.

[44] This fits in two TCP round trips given typical parameters.

A recent paper [CBT+19] applies fountain codes to the problem of speeding up block broadcast. The essence is to improve the time to complete, compared even to a perfect spanning tree, by taking greater advantage of the full cross-sectional capacity of the graph. It works by breaking a large block into small chunks and sending different chunks across different paths in the network. It has an analysis of its resistance to adversaries trying to interfere with the broadcast. Overall, it is an innovative idea and while as a broadcast algorithm it does not easily fit our design, it merits further analysis to determine if the core ideas could be adapted.

A network performance problem with classic blockchain algorithms is that they cannot fully utilise the available network capacity. We noted previously that time that is expended without sending or receiving data is time that can never be regained. Sending a block along a path in a graph from one slot leader to the next means that at any one time, all but one of the network links are idle and the capacity is wasted. The approach described above with fountain codes is one approach to try to use more of the links more of the time.

Another approach to achieve true scalability is by running a collection of parallel but related blockchains, allowing different machines to operate different subsets of the chains. Even in the special case that every node in the system manages all chains, there is an opportunity for better use of network capacity. By running multiple chains out of phase (systematically or randomly), the times at which blocks need to be transmitted can be spread out over time, leaving less wasted time when the network links are idle.

As part of our literature survey on decentralised connectivity graph construction we have sought out potential academic experts, including attending "The Mathematics of Networks" EPSRC-funded seminar series in the UK. Our general conclusion from the literature and from discussions with experts is that P2P graph construction is not a mature academic topic and would benefit from further study.

Other implementation approaches to decentralization are discussed in Section 7.

6 Distributed consensus on a global scale

6.1 Characteristics of Cardano

We start with a PoS algorithm called *Ouroboros*, and a mathematical proof that it works even with a substantial proportion of bad actors (adversaries) and a certain amount of delay in communication between nodes [BGKR17]. As described in Section 5, the engineering design of Cardano – and its Shelley implementation in particular – refines Ouroboros[45] to a robust, real-world, and computationally efficient implementation. A globally distributed network of nodes implementing Ouroboros in the real world must be robust against communication delays and failures, constraints in capacity, hostile actors, etc..[46]

Cardano is ultimately a cooperating system of autonomous nodes. It is not a client-server design, so there is fundamentally no central point of control nor any privileged class of centrally managed servers. However, its original implementation codenamed 'Byron' has seven federated nodes that produce blocks. Section 5.7 is the goal of the second phase of Shelley.

6.2 Fundamental requirements of Cardano data diffusion

The Shelley network component has a specific problem to solve: it must meet the *information exchange requirements* of the consensus algorithm, in particular:

- diffusion of blocks to all potential creators of new blocks

[45] Shelley includes an array of proofs and tests to ensure that desirable properties of the Ouroboros algorithm are preserved in this transition from theory to implementation.

[46] The CAP Theorem (https://en.wikipedia.org/wiki/CAP_theorem) places an upper bound on real-world performance which sums up as "Consistency, Availability, and Partition-tolerance: pick two.".

Table 2: Data diffusion budgets

	Threshold	Target	Stretch
Max hops	**20s budget**	**15s budget**	**10s budget**
2	10.0s	7.5s	5.0s
3	6.6s	5.0s	3.3s
4	5.0s	3.75s	2.5s
5	4.0s	3.0s	2.0s

  - within the required timing constraint.

- propagation of transactions for inclusion into blocks

It must do this using poorly-specified and/or poorly-implemented sub-systems outside Cardano's control[47], offering highly variable performance, in particular the global public IP network[48].

6.2.1 Timeliness constraint

The time-slotted nature of the Ouroboros algorithm introduces a strong timeliness constraint on data diffusion to all nodes. Since *any* (stake-holding) node could be the next leader, *every* node needs to have a copy of the last block before the end of the slot. Failing to meet this constraint reduces the security guarantees provided by the algorithm, damages the quality of the chain and hence strengthens any adversary.

Note that it is *not sufficient* to distribute blocks in a timely way 'on average' – every new block relies on knowledge of the preceding block, whichever node this was generated by. The data diffusion function therefore needs to deliver tight bounds on the probability distribution of delivery of every individual block, effectively making Cardano a globally-distributed stochastic hard-real-time system.[49]

Note that the 'hard' does not refer to the level of difficulty – increasing the slot length would make the problem easier – but to the consequences of failing to meet the deadline. It is rarely catastrophic if, e.g., a web page takes a long time to load, whereas failing to deliver blocks on time (whatever that time is chosen to be) has the potential to arm an adversary to disrupt Cardano.

The table 2 gives a per-hop budget for "data diffusion", based on the business requirement of a 20s inter-block interval[50]. Note that each 'hop' includes both reception and validation of the block. The next slot leader needs to receive the block in time to fully verify it and update its local mempool/UTxO before creating the next block.

The budget also needs to accommodate the potential clock skew between nodes (as discussed in Section 11.6), which can be 100ms or more. These figures should be compared with the time-to-complete for delivering a block in Section 11.2.1.

The budget also needs to cover worst-case and not just typical conditions. Factors making meeting the constraint harder to meet include the threats enumerated in Section 6.3 and the

[47] Any blockchain protocol has to deal with potentially corrupt and adversarial players but at least we can set the rules they play by when we design the system. The network component has no such guarantee: we can not modify the rules of the global IP network's game.

[48] Which also suffers from the legacy of many sub-optimal design decisions [Day08].

[49] By contrast: BitTorrent is globally distributed and scalable, but not time-critical; Skype or WhatsApp group is global and (moderately) time-critical, but not distributed, in the sense that the number of participants is small.

[50] This is the exact interval for Ouroboros Classic, and the targeted average interval for Praos.

extent and scale of the network (which will not be under any control once the system is fully decentralised).

6.2.2 Comparison with previous network implementations

The immediate question is: is this timeliness constraint hard to meet, or not? We consider some evidence from Bitcoin, Ethereum, and Cardano.

Bitcoin There is some evidence available from Bitcoin, in the form of two research papers and publicly available data following those papers.

http://bitcoinstats.com/network/propagation/ shows daily data on block and tx propagation time from 2013 to 2017, following a paper from 2013 [DW13]. The relevant data is the column "Block 90th percentile", showing the average time that it took a block to reach 90% of the nodes (averaged over a day). There are very few days where this has been under 10 seconds, and several days where it took more than 100 seconds.

Note that these are only averages over a day, so there are periods within a day where the time to get to 90% of the nodes is larger. Also notice the long tail, zooming in to a particular day (http://bitcoinstats.com/network/propagation/2016/02/25): 50% of the nodes took 8.3s, 75% 20.3s, but 99% 1205s.

The data shows an enormous spread. Of course, this is not a problem for Bitcoin, due to its large inter-block interval. What we need for Cardano, however, is consistently delivering blocks to (at least) the next block producer in less than 20s, minus whatever time it takes to create another block. Any period of time where we do not reach that will lead to random forks, and an opportunity for an attacker to mount an attack.

There is another paper [Neu19] that shows a large improvement in Bitcoin block propagation time in 2018. With those numbers, it seems possible to run Cardano for a couple of months before forks would appear and the system becomes easily attackable. The authors attribute this decrease in block propagation time to two things:

1. The appearance of relay networks such as FIBRE[51].
2. Changes in the bitcoin network have allowed a faster transmission of blocks.

Keep in mind that for Bitcoin it suffices to check just the hash of the block before passing it on to peers, but for Cardano, more expensive block validation is needed before relaying, as not doing so would open up an asymmetric resource DoS attack on the network, as discussed in Section 5.1.1.

Ethereum Etherscan shows data for the average block time in Ethereum. The graph suggests that Cardano's requirement of consistently achieving block delivery in 20 seconds can be realistically solved by existing technology. If it works for Ethereum, why not for us?

The graph shows daily averages, not individual blocks; the 95th percentile is inevitably higher! Also, there are extended periods (2.5 months in 2017, when the system was under heavy load) where the *average* block time was consistently larger than 20 seconds.

The further crucial point is that comparing the block time in Ethereum and the slot length in Cardano is not straightforward. In Ethereum (or any PoW system), block time is determined by how long it takes a mining node to: process transactions; solve the cryptographic puzzle; and construct the next block. Since *any* node can produce this block, there is no requirement on the propagation of the previous block through the network; if only a subset of nodes receives

[51] Note that this is essentially the approach that we are taking with Cardano for the time being until we have demonstrated in simulation that our peer selection algorithm results are good, and have determined suitable parameters.

the block in this time, one of *those* nodes will win the race. But in Cardano, the node that can produce the next block, and the time by which this has to happen, are fixed beforehand. Failure to deliver the previous block to this specific node[52] on time will result in a fork, a reduction in chain growth, and an advantage for the adversary.

The question then is:

> *if the block propagation times (the time it takes a block to propagate to the vast majority of nodes) in Ethereum were not consistently below the block time (the time it takes a node to create a block), would that not lead to frequent forks?*

The answer is:

> yes, it would – or rather, yes, it *does* ...

... but instead of avoiding frequent forks by forcing the block time to be well above the expected block propagation time (as Bitcoin does), Ethereum uses a consensus algorithm and reward mechanism that accept frequent forks. So-called "uncle blocks" – blocks that are not a direct ancestor, but forked off the same chain in the recent past – can be referenced in a block, to increase the chance of the block being accepted, and miners of uncle blocks get (reduced) rewards for them. More on this in

- https://ethereum.stackexchange.com/questions/10166/why-doesnt-ethereum-have-a-fast-relay-network-like-bitcoin and also
- https://medium.facilelogin.com/the-mystery-behind-block-time-63351e35603a).

Besides, the block size for Ethereum is typically 20-30kb (kilobytes) – with ~35kb during the 2017/18 winter rush. This is mainly in the ballistic throughput range of TCP[53], so a hop in Ethereum is fast. For Cardano, we want up to 50 transactions per second[54], which for transaction sizes of 1kb and a slot length of 20s, implies 1Mb (megabyte) blocks, which take significantly longer to send (compare with Section 11.2).

So the short answer to the question

> *"Why can't we use what Ethereum uses to guarantee block delivery within a slot length?"*

is:

> Ethereum's networking does not provide such delivery guarantees. Instead, it uses small blocks and adapts its consensus and rewards algorithm to reduce the impact of forks that result from a failure to deliver blocks before the next block is created.

Additionally, prior to early 2018, Ethereum was vulnerable to eclipse attacks that could be executed by attackers with very low resources. This was discovered by independent academic analysis of the code [MHG18]. There remains little positive evidence of the robustness of the P2P layer in Ethereum to this class of problem. The existing research has focused on finding and fixing Kademlia flaws, rather than proving outcomes within a threat model. Indeed, even with the adoption of the recommendations from the academic analysis of Ethereum's Kademlia, eclipse attacks using thousands of nodes are still possible.

[52] In Ouroboros Praos, the node due to be leader is known only to itself, thus blocks must be propagated to all stake-holding nodes.

[53] The ballistic part of a TCP connection is those packets that can be sent within the current available window. An idle TCP connection "closes its window" down to an initial value (between 4 and 10 packets depending on O/S). Etherum (either deliberately or not) is taking advantage of this in the block size used.

[54] 50 is the stretch target, not the minimum, but it does mean we should not have a design limitation that would prevent 50.

Cardano Byron The third data point is the first implementation of Cardano itself. Before launching, an extensive benchmark/stress test determined that we *could not* distribute blocks amongst the nodes within a slot if the "discovered" network diameter was too large, using the Kademlia-based network implementation[55].

This led to the short-term decision to choose the network layout that we used in Byron – 7 tightly connected core nodes that produce blocks, surrounded by relay nodes that facilitate communication with end users – and to the long-term decision to design a network stack that could handle the unique requirements of Cardano.

6.2.3 Stateful connections

In constructing an implementation of the consensus algorithm, it is essential to interleave aspects of validation with communication, as explained in detail in Section 5.1. This has the important implication that the *sequence of communications* from any peer needs to be preserved, which means that connections need to be stateful. This effectively rules out stateless communication stacks, such as typical RPC approaches.

6.3 High-level threat model

Fundamentally, the threat model is that of Ouroboros, with some practical real-world considerations (like having to work on the existing Internet and not trying to be safe against well-funded/state-level actors).

Whilst this is not captured in a traditional methodology of threat model development, the below are the major areas of consideration for such a model.

The Ouroboros assumption is quite weak: simply that at least 50% of the stake is held by agents that will follow the Ouroboros algorithm faithfully[56]. This obviously says very little about the trustworthiness of any particular agent/network node.

We also have the requirement that anyone can participate in the Cardano system, even if they own no or very little stake. Thus the risk of Sybil-style attacks[57] in the network layer is very high: it is very cheap to make a large number of network-level agents that can interact with honest nodes in the system.

The principal threats considered in the Shelley network design are:

1. Adversarial peers
2. Resource exhaustion attacks
3. Tier-1 actors
4. Bearer-level attacks

The response to these in the Shelley network design is summarised in Section 9.9

[55]Note that there was a Kademlia implementation in the original Cardano-SL code, and it was used in tests. It was realised that this would not work for large networks and for small networks it often resulted in a partitioned network graph. Thus it was never used in production, rather a static graph was used instead. The static graph topology was subject to K-cut analysis to ensure maximum resilience to both localised (i.e data centre) and large scale (loss of connectivity due undersea earthquake).

[56]Technically, in Ouroboros Praos, this assertion is diluted by allowing for delays in delivering messages. The honest stake proportion becomes $50\% + F(\Delta)$, where Δ is the number of slot-times that a message is allowed to be delayed.

[57]https://www.geeksforgeeks.org/sybil-attack/

6.3.1 Adversarial peers

The Cardano system consists of independently operating nodes that exchange information according to a specific protocol; the set of nodes that a particular node communicates with are called its 'peers'.

We define an *adversarial* peer to be one that does not correctly implement the protocol (so by definition both inoperative and malicious peers are adversarial)[58].

The Ouroboros algorithm works to deliver reliable consensus even when some nodes are adversarial, provided the proportion of the total stake held by the remaining 'honest' nodes is no less than 51%.[59]

Adversarial peers can engage in the protocols incorrectly in two main ways:

1. functional: by sending incorrect information; and
2. non-functional: by failing to meet timing constraints – deliberately or not.

Functional violations include high level examples such as sending invalid blocks, and low level examples such as sending unexpected or ill-formed protocol messages. Non-functional violations include examples such as failing to forward blocks or transactions.

6.3.2 Eclipse attacks

A particular case of adversarial peers is the *eclipse attack*, in which a high proportion of a node's peers are adversarial and collaborate to deliver false but consistent information to the eclipsed node.

In PoW systems the inherent uncertainty in the appearance of new blocks makes this attack effective, since it is difficult for the node to detect that it is failing to receive blocks generated by honest actors using information available within the system[60]. An adversary can effectively magnify its hash power by delaying or deleting blocks generated by honest miners. This leads to requirements on the network layer to construct connectivity graphs in which eclipse attacks are more difficult (see Section 7.2 on Kademlia).

In PoS systems such as Ouroboros, however, blocks are generated according to a pre-established stake distribution, so a node can *always* detect eclipse; either it receives few or no blocks[61], or blocks that are invalid. This means that making eclipse attacks difficult is *not* a requirement for the network component of Shelley.

6.3.3 Resource exhaustion attacks

Another way to attack a node is to exhaust some resource such as:

- CPU capacity;
- RAM;
- storage;
- network interface capacity.

[58] This is consistent with the use of the term in the Ouroboros papers, but applied to the network protocols between peers.

[59] In fact $50\% + \varepsilon$ would do, but it would take a long time to certify that the system has settled to a consistent state.

[60] Information extrinsic to the system, such as the locations of major mining pools, might be useful.

[61] Complications in eclipse detection arising from the random generation of blocks in Ouroboros Praos are discussed in Section 11.4.1

There is a further implied requirement to achieve timely information exchanges even in the presence of *adversarial behaviour at the network level*. Also implicit is that we must do this within bounded (and reasonable) resource limits; otherwise the system becomes too expensive and/or impractical to deploy. Combining these requirements we conclude that it is desirable to ensure that adversarial behaviour cannot increase the resource consumption at a node.

We cannot prevent this altogether, but we can make it expensive for the attacker, i.e. they must expend as much resource to mount the attack as it will consume on the node (preferably more). This makes it difficult for the attacker to compromise sufficient nodes simultaneously to weaken the security guarantee of Ouroboros.

Note that there is a significant difference in the nature of this threat between PoW and PoS systems. In PoW there is an inherent asymmetry between the honest nodes and adversaries: a PoW header is very cheap to check and requires no context from the chain. False headers passing the PoW check are very expensive to create. A PoW relay node does not need to keep a copy of the chain or ledger state and can still bound the number of false headers it propagates to be proportional to the hashing power of the adversary.

By contrast, in a PoS blockchain such as Cardano it is cheap for network-level adversaries to construct erroneous headers and blocks. Honest nodes validating headers require the full state of the ledger as context[62], as discussed in detail in the Section 5.1.1. Adversaries with a relatively modest amount of stake can create unbounded numbers of apparently valid headers and blocks (in the slots in which they are entitled to create blocks). For DoS prevention overall, the balance between honest vs adversarial nodes in a PoW system is much better than for honest nodes in a PoS system. This is a significant attack vector that must be addressed.

It is also necessary to consider its interaction with the timeliness constraint; if an attacker can increase the resources used by a node to the point where it becomes significantly slower to respond this could impact timely diffusion of blocks.

6.3.4 Tier-1 actors

Executing any consensus algorithm between a set of distributed nodes connected by the global IP network adds new threats from outside the system. In particular we must consider what can be done by 'actors' who can observe/modify the traffic contents and/or arrival pattern to/from a node; we call these *tier-1 actors*[63], which include ISPs, datacenter operators and nation states.

Clearly a tier-1 actor can prevent a given node within its domain from participating in the algorithm, and there is no way around this for the node other than to have access to an alternative bearer that is not under control of that actor.

However, for Cardano, provided the whole system of nodes is sufficiently distributed and diverse so that no single tier-1 actor can isolate a large proportion of the total stake, the security of the algorithm will be preserved. Ensuring the delivery of correct and timely information so that Cardano as a whole cannot be easily compromised is one of the primary goals of the network design.

6.3.5 Bearer-level attacks

In our context, the bearer is TCP over the public IP network. An attack on the node at this level is typically a denial-of-service (DoS) attack that involves overloading the host machine with IP packets. Such attacks on a node at the bearer level are beyond our control, since the decision to forward packets is made by the IP network and is unlikely to be affected by any action a

[62] If we delay validating headers until adding the whole block to the chain then we have already spent our network resources on downloading the corresponding block body. This would open up lots of resource attacks.

[63] https://en.wikipedia.org/wiki/Tier_1_network

node can take[64]. However, the impact of such attacks can be mitigated by careful design of the network architecture.

7 Analysis of alternative approaches

A variety of approaches to constructing networks have been proposed in the literature. Questions to ask about them include:

- What advantages do they offer?
- Can they natively:
  - Ensure meeting the timeliness constraint?
  - Deal with adversarial peers?
  - Resist resource exhaustion attacks?
- If not, what is the likely effort in modifying them to be able to do so?
  - And the risk of failing to do so in the Shelley delivery timescale?

In general:

- Broadcast methods are not used here for two reasons:
  a. broadcast without interleaved validation would be subject to trivial DoS,
  b. quite some time was spent looking at how to use block broadcast to implement Ouroboros but did not produce anything safe, whereas using chain syncing the security was clear and relatively simple to validate.
- Off-the-shelf P2P systems are not designed to deal with our threat model. In particular very few are designed to resist asymmetric resource consumption attacks. Such attacks are a real problem as demonstrated by recent CVEs on HTTP2 implementations[65].
- Existing P2P graph-construction systems are of limited use to us because they rely on the number of attackers in the network being a modest fraction of the overall network. Since creating network nodes in Cardano is essentially free, and our network will not be very large, this is not a strong defence and certainly would invalidate our Ouroboros claims, based only on the Ouroboros assumptions[66].

We now consider selected published related work in more detail.

7.1 Dandelion

This is an approach to improve potential anonymity for the submission of transactions in existing cryptocurrency network systems such as Bitcoin's through encrypted forwarding and randomised routing [VFV17]. The goal is to minimise the information that observers *within the distributed ledger system* can obtain about the ultimate source of a transaction. The authors admit, however, that their system provides no guarantee of anonymity against Section 6.3.4.

Such anonymity enhancement is not a requirement for Shelley, but could be considered in later iterations.

[64] An organisation or individual responsible for a node may be able to take action, but on a longer timescale.

[65] https://www.securityweek.com/http2-implementation-vulnerabilities-expose-servers-dos-attacks

[66] Our claim would essentially reduce to "assume 50% of stake is non-adversarial *and* assume that nobody bothers to create 1000 adversarial network nodes (with 0 stake each)".

7.2 Kademlia

Kademlia [MM02] is an approach to creating distributed hash tables, which helps with randomizing peer choices to minimise the risk of node eclipse in proof-of-work systems. Avoiding eclipse in PoW requires a node to associate with a majority of non-adversarial peers. Thus an adversary will attempt to bias a node's selection of peers (towards other compromised nodes); Kademlia aims to prevent this. Several variants of Kademlia exist (e.g. "secure Kademlia"); overall, however, the Kademlia literature concludes this is not a robust solution to the problem of eclipse [MHG15]. In fact there is no argument that even with various enhancements the algorithm in fact achieves anything, with any particular threat model.

Integrated into the algorithm for using the distributed hash table is functionality for constructing and managing a peer-to-peer connectivity graph. This functionality can be extracted, albeit with some necessary remaining vestiges[67]. Ethereum uses a variant of S/Kademlia.

A 2018 paper [MHG18] analysing Ethereum's Kademlia implementation nevertheless demonstrated a number of different eclipse attacks requiring minimal resources. The countermeasures suggested by the paper authors mean that instead of two machines being required to execute the attacks, thousands are required. Nevertheless, thousands of machines are only a modest expense when "borrowed" as part of a botnet.

The paper authors' commentary on the choice of Kademlia for Ethereum is instructive:

> "The Ethereum developers state that the Ethereum peer-to-peer network protocol is based on the Kademlia DHT. However, the design goals of the two are dramatically different. Kademlia provides an efficient means for storing and finding content in a distributed peer-to-peer network. Each item of content (e.g., a video) is stored at small subset *[sic]* of peers in the network. Kademlia ensures that each item of content can be discovered by querying no more than a logarithmic number of nodes in the network. By contrast, the Ethereum protocol has just one item of content that all nodes wish to discover: the Ethereum blockchain. The full Ethereum blockchain is stored at each Ethereum node. As such, Ethereum's peer-to-peer network is not needed for content discovery; it is only used to discover new peers. This means that Ethereum inherits most of the complicated artefacts of the Kademlia protocol, even though it rarely uses the key property for which Kademlia was designed."

In addition to the unnecessary complexity and the problem of eclipse attacks, Kademlia is not well suited for constructing connectivity graphs for low latency data dissemination. Kademlia constructs graphs with a bounded hop count, but the hops that it picks are completely unrelated to the network distance between nodes. This unhelpful property is not easy to change because Kademlia is fundamentally based around – and self-optimises for – a notion of distance (in a virtual metric space) that is unrelated to physical network distance. This can mean that a path from London to Dublin goes via Sydney, or worse since the typical path length will be considerably more than two. This problem could perhaps be solved by yet another overlay, but see Section 11.5 on the drawbacks.

Proof-of-work systems must rely to some extent on what peers report about other peers[68] (note that this requires nodes to have stable identities) see Section 6.3.2 on eclipse attacks. Cardano does not need this because the stake-based leader selection means that an adversary cannot spoof blocks; it can only reduce the chain growth quality. Thus the intended benefits of Kademlia (over and above the use of gossiping) are not required for Cardano.

Conversely the randomisation of the node topology imposed by Kademlia works against the timeliness of data diffusion and thus reduces the performance and security of Cardano. This is evidenced by the block distribution data from Ethereum in Section 6.2.2.

[67] The equivalent of DHT lookup is needed to maintain the routing tables

[68] We could refer to this as 'transitive trust'.

Kademlia introduces considerable complexity: a month-long study was performed early in the design cycle that concluded that validating a Kademlia implementation would be extremely hard, and debugging any issues that arose in deployment even harder. At a minimum it would require adding the peer selection via ΔQ, which is not native to Kademlia.

Kademlia is designed to work over UDP, which typically fails to reach nodes behind NAT/Firewalls (see for example https://geth.ethereum.org/doc/Connecting-to-the-network, where the instructions are clearly directed at technically savvy users only). Dealing with this in a smooth way, transparent to end-users, would require a fall-back TCP connectivity mode. Our preferred solution is to avoid this complexity by using TCP only.

There are three available Haskell implementations:

- The original one on hackage:
  - This leaves "some of the implementation details, like timeout intervals and k-bucket size, for the user."
  - It has several open issues dating back to 2015, which does not inspire confidence in the quality of the code.
- The Serokell derivative of this:
  - This was included in the cardano-sl codebase but proved unsuitable for use in production.
- A new implementation that the networking team started in 2018 but after at least a month's work decided to abandon because the peer validation and veracity mechanism was proving too complex and had significant problems.

Thus, no suitable 'off the shelf' implementation is available.
In summary, Kademlia:

- Adds huge complexity, both in coding and testing
  - No suitable off-the-shelf implementation
- Delivers no specific benefits to Cardano
  - Peer discovery 'gossiping' function very simple to implement by itself
    - Kademlia approach is providing design inspiration for this part

7.3 PolderCast

PolderCast [SvVV12] is a pub/sub system designed to support large numbers of "topic" channels, each with a few publishers and many subscribers, and aims to exploit variability in topic popularity. PolderCast is designed to support a large number of pub/sub topics while keeping the in/out-degree of each node to a reasonable number. Its experimental performance evaluation assumes that topics only cover a small fraction of nodes in the system[69]. This is not a good fit for Ouroboros which requires only two topics – blocks and transactions – but all nodes are interested in blocks and the vast majority are interested in publishing transactions.

As discussed in Section 5.1.1, pub/sub systems do not have the appropriate properties to support chain sync. However, there is another aspect of that could be of interest – that of topology creation.

[69] In the Facebook and Twitter datasets no topic ring covers more than around 5% of the 10,000 nodes.

PolderCast uses a three-level approach to creating topology: rings (which are per topic – to support deterministic distribution), with two additional topology mutators aimed at reducing path length and ring-reconstruction under node churn. Its primary use case (major motivating example) is that of social networking.

We have examined the PolderCast topology construction approach, but Ouroboros represents a pathological case from its design perspective. It can either be seen as a single topic or as a topic per potential slot leader – each of which contains the total population of all nodes (both wallets and slot leaders). PolderCast is designed to support connectivity models (i.e. strongly connected graph components – "super nodes") that occur in 'small world' [Wat99] networks, of which social networks are a good example.

It creates a ring per topic and uses the non-uniformity of the user base's participation in a topic to derive *proximity* (a measure of "interest locality") which, in turn, is used to drive one of the mutators that creates probabilistic short cuts. This non-uniformity is absent in the Ouroboros use case. It should be noted [SvVV12, Section 5.4] that whenever PolderCast uses a "gossip" list (not one of the ring) to communicate with a peer it removes that peer association[70], re-establishing it only when it receives a new message. This removes the ability of a node to sequence the messages from another peer, which is essential for Cardano in order to be able to validate them correctly.

There is no consideration in their model for optimising the overall timeliness of the distribution of messages. Their "fast dissemination latency" assertion is based on hop count, which is "typically of the order of the log of the topic subscribers". It would appear (on the basis of the description of their construction mechanism) that the base of the log used is ˜2. Note that their simulation results in [SvVV12, Section 6.4] (fig 8) appear to relate over the total population of all their topic sizes, which from [SvVV12, Section 6.2 fig 7] are for (relatively) small topic population sizes (hundreds to a few thousand) compared to the target population size for Cardano – 10k-100k+ total nodes.

There is no evaluation of effectiveness of the system in the presence of adversarial network conditions and/or behaviour – it appears their reachability completeness results are assuming a 'failure-free' operational environment (although nodes may drop out and rejoin).

Indeed, the topology construction (which is node identity aware) would imply that two cooperating nodes (especially those that have the additional cross connect path) – knowing their relative relationship – could conspire to dramatically increase the dissemination path length or even partition the network at will.

There doesn't seem to be any inherent advantage to adopting a PolderCast style topology construction over assuming a random graph and appealing to the various phase transition properties (including characteristic path length) described in [Wat99].

Furthermore, the basis of a pub/sub system is that subscribing to a channel implies a willingness to accept everything published on that channel. Thus, PolderCast has no means of managing node resources at either the network or data storage levels, which arms a range of denial-of-service attacks.

7.4 Summary of comparison

	Dandelion	**Kademlia**	**PolderCast**
Advantages	Reduces the ability to deduce the source of transactions	Resistance to eclipse (irrelevant in Cardano) Peer discovery	Automatic creation of topic rings; supports many different channels

[70]The shot-cuts are single use only.

	Dandelion	Kademlia	PolderCast
Timeliness	Makes worse	Makes worse	Renders unachievable
Adversarial peers	No mitigation impact	Some resistance to DoS	No mitigation impact
Resource exhaustion	No mitigation impact	No mitigation impact	Pub/sub creates vulnerability
Effort required to make suitable	Moderate	Substantial: one month spent already	Substantial: fundamental assumptions are misaligned

8 Operational environment and constraints

Cardano Shelley should be realisable on various platforms and scales of resources, from large stake pools processing many transactions and frequently creating blocks, to exchanges and, ultimately, individuals, either creating (occasional) blocks or simply running wallets.

Cardano Shelley should also be usable by enterprises, so it must meet corporate governance criteria, including regulatory constraints, and should operate across a DMZ.

It should support many deployment scenarios without requiring much specialist expertise, and run on standard operating systems without requiring kernel modifications or parameter changes, including:

- Windows
- Linux
- MacOS

It should support a range of connectivity types:

- Consumer broadband
  - Implying firewall and NAT (domestic and carrier) traversal[71]
- Datacenter networks
- Wide-area networks
  - IPv4 and IPv6
  - Other bearers
- Intra-machine IPC
  - Avoiding 'TLS hell'

It should be executable without consuming excessive[72] resources:

- CPU
- Memory
- Network interface capacity
- TCP ports
- Storage

[71] This essentially forces us to use TCP as the underlying transport mechanism.

[72] We take a T2 Medium AWS instance as representing 'reasonable' resources for a full node.

8.1 Data diffusion targets

The fundamental task of the data diffusion function is to distribute blocks amongst peers in a timely fashion. Failure to deliver on time should be rare, even under adverse conditions. We are assuming that Ouroboros Praos will aim for the same rate of block production as Ouroboros Classic[73] by aiming for a slot time of 1s and a slot occupancy of 5%. Allowing the Praos Δ parameter to be ~5 gives a target for block distribution of X% of peers reached within 5s. For well-connected core nodes, the business requirements of 11.1 provide the following targets:

Threshold	$> 95\%$
Target	$> 98\%$
Stretch	$> 99\%$

This should be achieved even when blocks are all 'full', as constrained by the on-chain parameter. The hazard that arises otherwise is that there will be a turning point in the performance curve, which could lead to throughput collapse and excessive forking of the chain. Note that this implies a bandwidth[74] of 100kb/s, which is roughly 1Mbit/s, even in the 'super-stretch' scenario where 2Mb blocks emerge (on average) every 20s.

The network component is also responsible for diffusing transactions so that they can be incorporated into blocks. There is no benefit in transporting more transactions than can be incorporated into the chain, so the maximum bandwidth that needs to be sustained is the same as for diffusing blocks.

8.2 Fundamental tradeoffs

The intrinsic nature of any distributed consensus process (as discussed in Section 5.6 above) creates a series of fundamental tradeoffs that need to be managed:

- Between geographic spread and minimum slot time and Δ;
  - A consistent view can be created more quickly if the participating nodes are physically close together, so that there are lower delays in exchanging information;
- Between transaction size (50 bytes - 10's kbytes) and transaction rate[75];
  - Larger transactions take more space in the block, so fewer of them can be incorporated in any given slot;
- Between node valency, path length and network capacity;
  - A higher valency (more connected peers) reduces the number of hops that information must travel, but increases the consumption of available network capacity at a node[76];
  - A larger number of hops reduces the time that any individual hop can be allowed to take and hence demands more network capacity to deliver the information quickly to the next node;
  - Both of these drive up the peak capacity requirement at a node.

[73] We assume that relaxing these constraints substantially – which would lead to Shelley having much lower transaction throughput and longer settling times – is not acceptable from a business/community acceptance perspective.

[74] Note that this is required for *any* node (even a wallet) to keep up with the maximum rate of chain growth.

[75] Transaction bytes per second might be a better measure than transaction rate.

[76] Increasing average node valency has diminishing returns, such that values beyond 20 are probably not justified. What is more significant for the characteristic path length is the variance of node valency - i.e. some nodes have substantially more chain-following subscribers, e.g. in the hundreds.

Table 6: Transaction size as a function of block size and transactions per second

	transactions per second						
Block size (MB)	2	5	10	20	50	100	200
0.064	1661	665	332	166	66	33	17
0.128	3323	1329	665	332	133	66	33
0.256	6645	2658	1329	665	266	133	66
0.512	13291	5316	2658	1329	532	266	133
0.768	19936	7974	3987	1994	797	399	199
1.024	26581	10633	5316	2658	1063	532	266
1.536	39872	15949	7974	3987	1595	797	399
2.048	53163	21265	10633	5316	2127	1063	532

Table 6 shows the effect of transaction rate and block size on the resulting average transaction size (in bytes), assuming a 20s average block interval.

It can be seen that high transaction rates limit the transaction size, even with large blocks. Also there are combinations that are infeasible for different uses. A transaction has a absolute minimum size of 305 bytes[77] (single input UTxO to single UTxO), a "useful" one (1 input to two outputs) 361 bytes. Coin selection (for privacy enhancement) uses multiple inputs and multiple outputs, and exchanges look for large numbers (50+) and transactions sizes in the 8k+ range. Envisaged future uses (Plutus / Marlow) will bring with them their own characteristic transaction size distributions.

8.3 Adversarial power and knowledge

Following established best practice in security design, we assume the adversary knows every detail of how the system works (i.e. has access to the code); this means any weaknesses may be exploited, including 'auto-immune attacks', in which the mitigation for one attack opens up a vector for another. We assume the adversary is not a Tier-1 actor and does not have access to secret keys, but may have access to resources to mount denial-of-service attacks.

In common with the design of Ouroboros, we assume that any Cardano peer may be adversarial, and so cannot be trusted. It follows that we cannot trust anything they tell us about other nodes either, i.e. there is no 'transitive trust'. The avoidance of the construction of hegemony also implies that we cannot rely on something like a standard public key infrastructure (PKI).

We assume that power outages, network connectivity failures, routing table failures etc. can occur, but can't be predicted or correlated by an adversary. An adversary might exploit them when they occur, however.

8.4 Stake distribution

Assuming that stake pools exhibit rational behavior, we expect of our delegation design[78] that roughly 100 - 1000 peers will hold at least 80% of the total stake.

[77] At least in the benchmarking environment, which is using the simplest forms of transaction (correct at time of writing).

[78] https://hydra.iohk.io/job/Cardano/cardano-ledger-specs/delegationDesignSpec/latest/download-by-type/doc-pdf/delegation_design_spec

8.5 Graceful degradation

In order to maximise the chances that Cardano/Shelley will survive a major incident, we need to implement a precedence order of failure. This means prioritising:

1. The system as a whole over any individual node;
2. Large nodes (in terms of stake) over small nodes;
3. Nodes that create blocks over those that do not;
4. Blocks over transactions.

8.6 Backward compatibility and extensibility

While Shelley aims to be more efficient than Byron, it must still be possible to communicate between the two different versions of Cardano in order to enable a smooth transition. Thus it is essential to be able to construct a proxy that can translate between the old and new protocols.

More generally, in the expectation that Cardano will continue to evolve, it is important to have a framework that is flexible and extensible to support new features without requiring a hard fork. It is also desirable to provide a clean boundary between the network component and the consensus component, so that they can be developed, tested, and evolve independently.

9 Key design decisions

Creating an efficient implementation of an algorithm[79] requires dealing with real-world complications. This adds detail to the interactions and creates *new possibilities for adversarial behaviour* to subvert the algorithm[80] (the *attack surface*).

Ideally we would prove that even with the new ways that the adversary can interact with honest nodes, that the network information exchange requirements can still be met with bounded resources.

In the Shelley iteration we will not aim to formally prove this. It is, however, a design goal to be able to construct informal arguments that the design meets its information exchange requirements in bounded resources in the presence of adversaries. Thus, throughout the implementation process we need to take care to:

- minimise the increase in the attack surface, and
- mitigate against any increase that we cannot avoid.

We call this *threat-aware design*.

Following the threat-aware principle while respecting the targets and constraints of Section 5 has led to mutually reinforcing design decisions for the data diffusion component. The most important are:

- Peer-with-peer communication;
- A DoS-resistant network architecture;
- A development approach exploiting the capabilities of Haskell;
- The use of abstract multiplexed point-to-point bearers;

[79] For Ouroboros, an example would be broadcasting blocks rather than complete chains.

[80] For instance the constraint that buffers for data need to be of a finite size creates the (frequently exploited) potential to overrun them.

- Managing the network topology with a demand-driven spanning tree;
- Using a compositional, polymorphic, typed protocol framework;
- Continual, automatic performance assessment and optimisation.

9.1 Stateful implementation

As noted in Section 5.1.3, the option to have a connection-oriented stateful protocol enables protocols that rely on an *amortised* analysis for their resource bounds. Establishing resource bounds at all is difficult, especially while balancing the other priorities, such as achieving good performance by making effective use of available network bandwidth and latency hiding. In the chosen network design, this extra flexibility has proven useful for the transaction submission protocol. Here we revisit the stateless/stateful question again from an implementation perspective.

The primary advantage of using stateless protocols is that there are more available frameworks and off-the-shelf implementations, especially in the category of HTTP and RPC implementations. The hope of course is that this can save development effort, increase flexibility and reduce maintenance costs. So it is worth reviewing the issues involved in selecting an existing implementation, stateless or stateful.

The most immediate constraint is that the implementation – not just the protocol in principle but the actual implementation – must be designed for an adversarial environment, including asymmetric resource attacks. However, the bulk of off-the-shelf protocol implementations are designed for business information processing applications within a corporate network or data centre, rather than for use on the open internet. For example ZeroMQ was designed for use in data-centres and its documentation used to warn explicitly against running it over the public internet, though it is now more sanguine[81]. Facebook's Thrift and Google's gRPC are similar in that they are derivatives of their respective internal data centre RPC frameworks. This does not exclude them, but highlights that careful scrutiny is required to ensure that they have been made sufficiently safe and robust to be open to the public internet. There are examples on the stateful side too: the widely used message broker RabbitMQ for example is not recommended for anonymous untrusted access from the internet.

The most obvious example of a stateless protocol with many reasonable implementations that are fairly resistant to adversarial behaviour on the public internet is HTTP. Over the years, HTTP implementations have faced trial by fire, and repeatedly been found wanting (e.g. the slowloris asymmetric resource attack), but there are now many reasonable implementations[82]. Using a good HTTP implementation is not enough of course; the slowloris attack, for example, is now primarily an application level attack[83].

The most obvious example of a *stateful* protocol with good implementations that are pretty resistant to adversarial behaviour on the public internet is TCP. TCP is of course extremely widely supported, in software support, tool support and in the physical infrastructure of the internet. Furthermore it is supported in every OS and receives updates for security and performance issues.

[81] It now claims not to crash with malformed packets but makes no claim about resource attacks or actually being *recommended* to use over the public internet.

[82] In 2012 the venerable Apache web server 2.4 release finally changed the default MPM on unix so it now avoids the slowloris attack

[83] Services offering slowloris protection exist, e.g. https://www.cloudflare.com/learning/ddos/ddos-attack-tools/slowloris/

9.2 Peer-with-peer

A typical 'peer-to-peer' application such as BitTorrent or PolderCast establishes a forwarding graph between nodes and then forwards data as rapidly as possible without much, if any, regard to its validity (this is regarded as the problem of the ultimate recipient). This approach would be catastrophic for Cardano, however, as it would enable an adversary to amplify a resource exhaustion attack - one adversarial node can consume resources of many honest nodes in forwarding junk data.

The key threat-aware design principles of:

1. Detect and disconnect adversarial peers
2. Control resource consumption

lead to a different model, which we call 'peer-with-peer' to distinguish it from the typical peer-to-peer approach. In order to prevent adversarial amplification we must exploit the semantic content of the information exchanged (i.e. its meaning in the context of the current state) to:

1. detect aberrant behaviour and
2. manage resource consumption exposure[84].

The assumption that no peer can be trusted leads to three fundamental design decisions:

1. Only forwarding validated information;
2. Strictly controlling the potential for resource consumption;
3. Not forwarding information about other nodes.[85]

9.2.1 Validated forwarding

Nodes are responsible for the validity of forwarded messages. In particular, messages must be:

- well-formed syntactically, and
- semantically valid in the context of previously forwarded information from that peer (as determined by the consensus component):
  - Blocks in order on their chain
  - Transactions
    - in submission order
    - compatible with UTxO and other recent transactions

If a peer fails to meet these requirements it is considered adversarial and the connection to it is dropped. This means that block and transaction relaying must be interleaved with validation. Without this the system is trivially vulnerable to adversaries broadcasting nonsense, using up everyone's bandwidth and thereby executing a DoS attack on the whole system.

[84] For an example of the consequences of failing to validate information before forwarding it, see: https://www.theregister.co.uk/2019/08/20/centurylink_outage_report_fcc/

[85] Apart from their addresses, which a node can use to attempt to form new connections.

Consequently, there is no network 'layer'[86] that sends messages (blocks or transactions) from one node indirectly to other nodes. The network component only sends messages from one node to a direct peer, and those messages go up to the application logic level (consensus, mempool etc)[87].

Note that it is a misunderstanding to think that block validation logic is 'pushed down' into the network layer. They are clearly separated, but there is an interleaving in the data flow. How this is done in the design is discussed in full detail, with justification and a diagram, in Section 5.1 and Section 5.2.

9.2.2 Demand-driven protocols

Protocols are designed in a demand-driven style, so that for each node and each peer connected to that node, the node controls the rate of data arriving, the maximum concurrency, and the amount of outstanding data. This prevents an adversarial peer from mounting a resource consumption attack: if it obeys the protocols, its ability to consume resources at the node is bounded, and if it violates them it will be disconnected. Of course, initial connection requests cannot be bounded by the protocol, but the rate at which they are accepted can be limited.

Controlling the potential offered load is also a prerequisite for managing timeliness, since it constrains the amount of queuing delay that can build up.

Note that this pull-based, data-consumer driven approach provides a degree of control that is absent from a pub/sub system, in which a subscription gives implicit permission to send an arbitrary amount of data; rate-limiting such data destroys its timeliness. Thus in a pub/sub system an individual node can arm the hazard of failing to meet delivery deadlines for the system as a whole.

The principle that no peer can be trusted implies that any information it supplies about other nodes cannot be trusted either, so there is no value in sending such information. Thus approaches such as reputation scores to assess trustworthiness of nodes cannot be employed. Each node trusts only information that it can verify itself.

9.3 Network architecture

The approach to mitigation of bearer-level DoS attacks is to defend *core* nodes, meaning the top stake pools and the transitional block-producing nodes operated by IOHK, by surrounding them with sacrificial relays.

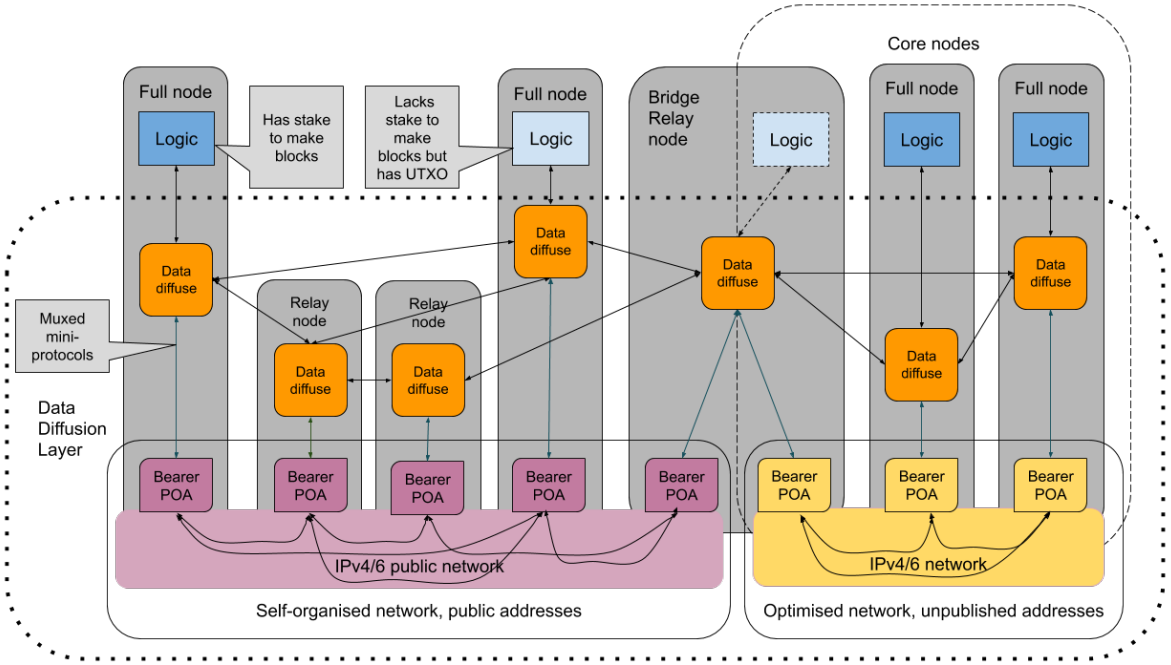
The IP address of the core node is not published[88], making it difficult to attack; and if relays are attacked they can be replaced with new ones having different IP addresses (which can be published to other nodes via mechanisms such as dynamic DNS). This is embodied in the figure below:

[86] In networking literature 'layer' is often used as a technical term that includes the idea that information can be delivered indirectly to other nodes, and that the delivery takes place within the network layer. In this document we use the term layer with a more colloquial meaning that is synonymous with the term component.

[87] Note that this necessarily rules out direct re-use of many network algorithms and implementations. At best one would have to figure out how to integrate block validation with variations of existing algorithms.

[88] It could be shared with other core nodes that are considered trustworthy so that they can communicate directly.

An alternative is not to hide the IP address, and rely on firewalls instead. However, care must be taken since some cloud services implement the firewall mechanism on the host rather than within the network, and in this case the host would still be open to bearer level attacks.



9.4 Development approach

To deal with the threat of an adversary who knows the code (which is inevitable for an open-source project), two methods were used to eliminate implementation errors:

1. Make maximum use of the Haskell type system;
   (a) Using a simple version of session types[89];
2. Build on the idea of an executable specification by being able to run the actual code in a simulated mode, which enables;
   (a) control over entropy and timing for reproducibility;
   (b) regression testing;
   (c) trying out worst-case scenarios.

9.4.1 Session Type Framework

An important feature of any protocol design is avoiding deadlock, in which both parties are either waiting for something or trying to send something to the other at the same time. The solution in Shelley is agency management, in which exactly one peer has agency at any time, enforced by the Haskell type system. This approach avoids deadlock while allowing pipelining, which overcomes the effect of bearer latency on performance-critical protocols. Pipelining can be implemented without changing protocol description or the server implementation – only the client side of the code has to be altered.

This is achieved using a Session Type framework, in which protocols are described as state machines encoded into Haskell types. The implementations that engage in these typed protocols are required to respect the protocol description, and this is enforced by the types. The state machine descriptions are constrained so that in each protocol state only one peer may send,

[89]Session types: https://groups.inf.ed.ac.uk/abcd/

while the other must receive. This guarantees that the protocols are free from race conditions or deadlocks. This eliminates a class of bugs and makes testing significantly easier.

The session type framework was also developed to allow the construction of polymorphic protocols: protocols which can be instantiated with different concrete data, e.g. the developed chain sync protocol is independent of the block type[90] which allows it to be used for different ouroboros protocols like Ouroboros-BFT or Ouroboros-Praos. Applying types to protocols rather than channels allows the channels to be polymorphic, so that there is no need to create a separate channel for each protocol.

The session type framework has been tremendously helpful in the design phase of each mini-protocol. It allowed us to communicate and discuss concise ideas, both within our own team and to other teams (e.g. the Wallet team). This is similar to the way other teams are using mathematical notation to precisely formulate and communicate ideas. The difference is that we used the strong type system of Haskell to encode protocols with mathematical precision, providing proofs of their correctness. We can also write tests for the protocols without running them on a network or using multiple threads.

9.5 Point-to-point bearers

The idea is to use a single 'bearer' between peers over which all interactions are multiplexed. This allows us to control the resources that can be consumed at a node by any connected peer, thereby minimising the scope for resource consumption attacks, including peer-related denial-of-service attacks. This has the downside of making the RTT more of a problem for performance, which we manage with pipelining (see Section 5.3.4). We apply backpressure to manage edge conditions, while minimising head-of-line blocking effects[91] by means of explicit multiplexing on the bearer. This gives us control over precedence of delivery and the emergent properties of the bearer by having the point of contention within the local node's domain of control.

The abstract notion[92] of a 'bearer' carrying a single ordered stream allows a range of connectivity types to be accommodated, from local sockets to satellite links. Although the initial wide-area bearer will be TCP, this generic approach means that other interprocess communication mechanisms can be used. For example communication between different processes (e.g. wallet backend) and the node can be done entirely within the security confines of a single machine.

This also accords with the 'minimise resource usage' principle expressed in Section 5, by using only one TCP port per connected peer. Also, using a uniform abstraction reduces the attack surface.

9.6 Demand-driven spanning tree

The set of nodes that blocks need to reach quickly is given by the stake distribution/number of stake pools. The block distribution time then depends on the topology of the peer interconnections between these nodes. As discussed in Section 5.6, Cardano nodes can detect eclipse attacks, unlike PoW systems such as Bitcoin, and so the Kademlia approach of enforcing randomness on the connectivity graph to make such attacks more difficult is not required. A more pressing requirement for Cardano is to limit the number of hops blocks must traverse[93], and so the decision was taken to use a demand-driven spanning tree approach, in which each node makes

[90] It is also similarly independent of the codec (transformation between concrete and abstract syntax).

[91] Failure to clear the queues by reading from the TCP socket will cause the receive window to fill up and the sender will stop sending more data. No data is lost and no packets are retransmitted, but no progress is made.

[92] Following the principle of compositionality,

[93] So that the per-node hop time target is achievable for a globally distributed system: this is discussed in more detail in the Annex.

independent choices of which of its peers to download new data from. Each node maintains a local list of peers, divided into three sets:

State	Definition
Cold	A 'cold' peer is one which the node is aware of but has not yet connected with
Warm	A 'warm' peer is one that the node has established a bearer connection with, and is following its chain, but is not currently using to obtain new blocks or transactions
Hot	A 'hot' peer is one that the node is actively using to obtain blocks and/or transactfions

Peers delivering data that is incorrect or late are rapidly deselected, i.e. removed or demoted from 'hot' to 'warm'. The 'cold' set is populated at start-up (as described in Section 5.7) and refreshed by lists of addresses provided on connection with peers. A node will move peers between these sets in response to receiving (or not) the number of well-formed blocks that it expects. Ultimately, if this fails (which could indicate a possible eclipse attack) it will have to return to its bootstrap procedure to obtain new 'cold' peers.

While it is important to minimise block propagation time, it is also important to avoid forming cliques (strongly-connected subgraphs) because this creates the risk that the network could be partitioned by the loss of only a few bearers. This means some 'warm' peers will need to be 'far off' (i.e. kept in the set even if they perform poorly).

As discussed in Section 5.6, one constraint is the total capacity of the network interface of a node. Each connected peer that is downloading blocks from the node will consume up to ~1Mbit/s. Thus a 'domestic' node with a 10Mbit/s uplink capacity could not accept requests from more than 10 peers. In extreme cases it might only serve blocks it generated itself!

9.7 Protocol framework

9.7.1 Compositionality

Shelley fundamentally improves on Byron by having a *compositional* design. This strong form of modularity means that separate aspects of each protocol's behaviour, resource consumption, and robustness can be designed and reasoned about independently.

The protocol framework is designed to maximise compositionality by using independent 'mini-protocols' for distinct functions (sidestepping the typical combinatorial complexity explosion) so that no single protocol becomes too complex to reason about. Key mini-protocols are:

- Bulk block sync
- Chain following – filter useful interactions to the consensus component
- Transaction submission

A separate function called 'the Mux' multiplexes these mini-protocols onto a point-to-point bearer for each connected peer node. This provides a common approach to version control, managing load and controlling resource consumption using isometric flow control.

The wireformat used by Mux is simple and compact which gives a low overhead, both in bytes over the network and in CPU cycles[94].

[94]Note that the efficiency of the encoding is important as it affects the network interface capacity required, which in turn constrains who can effectively run a Cardano node, as discussed in Section 8.2

9.7.2 Structured information exchanges

A naive implementation of Ouroboros would transmit far more data than strictly necessary, making it impossible to meet the timeliness constraint. Thus an important design approach was to analyse the information exchange requirements (IERs) and then map these onto data transfers in the most efficient way, while avoiding opening up opportunities for adversaries to attack the system. This is described above in Section 5.1

Resulting design decisions include:

- Exchanging blocks rather than broadcasting chains;
  - All practical consensus algorithms do this – it is simply a mathematical convenience to formulate the Ouroboros algorithm in terms of broadcasting chains;
- Interleaving forwarding with validation
  - To prevent adversaries consuming resources by sending invalid data;
- Sending block headers ahead of whole blocks, using the chain-sync protocol;
  - this allows a node to decide whether it needs to download a full block;
  - if it does it can choose which peer to download it from;
- Sending transaction IDs ahead of full transactions;
  - Request a number of transactions;
  - Receive a set of (short) transaction IDs and sizes;
  - Request a subset of those transactions.

Details of these protocols can be found in the second part of the design documentation.

9.7.3 Protocol polymorphism

A key point is the separation of:

- codec
- channel, and
- consensus logic

This enables early and thorough testing ahead of the integration of the parts.

The protocol driver takes the codec as a parameter, which allows for different choices of encoding. Codecs themselves can also be polymorphic, for example in the type of blocks or transactions. This aids in enabling the consensus and network layer combination to be parameterized by the choice of the ledger rules and representation. Allowing different choices of ledger rules improves the future flexibility and enables reuse of these components in other products.

The protocols are also polymorphic in the consensus algorithm, which provides a clean separation of concerns between the components and allows consensus algorithm evolution without changing the data diffusion code.

9.8 Performance assessment and optimisation

The strict timeliness constraints of block diffusion means that the data diffusion component must constantly monitor and optimise performance. Each node evaluates the performance of its peer connections by measuring their ΔQ [TD20], at both the bearer and mini-protocol levels. This enables efficient performance-optimising decisions by the mini-protocols; for instance, the block-fetch protocol can select between several different peers offering the same block(s) on the basis of which would be expected to deliver soonest based on their measured ΔQ.

Congestion between each pair of nodes is managed by the Mux function, and overall congestion at the network interface of a node is bounded by *isometric flow-control* (meaning that the potential outstanding traffic is limited)

The 'topography' is the combination of topology and performance. Performance is a function of both capacity and distance, because achievable TCP download speeds depend on the round-trip time (RTT) of the bearer (this is discussed in more detail in Section 11.2). The goal is to optimise the topography of the data diffusion network, not just its topology.

9.9 Summary response to threats

Here we summarise the responses to the threats discussed in Section 6.3 embodied in the Shelley network design.

Threat	Response
Adversarial peers	Detect adversarial behaviour (including poor performance) and drop the connection to that peer
Resource exhaustion attacks	Structure protocols so as to strictly bound the resources that any connected peer can consume (attempting to consume more than the protocol allows would be adversarial, see row above)
Tier-1 actors	Ensure nodes are sufficiently distributed and diverse so that no single tier-1 actor can isolate a large proportion of the total stake
Bearer-level attacks	Design the network architecture so that large stake-holding nodes are defended against DoS attacks

9.10 Bootstrap

Note that the problem of NAT/firewalls means that there is no way to construct a network without at least some nodes having accessible public IP addresses. A new node joining the network needs to be able to find these addresses. This can be achieved in different ways, depending on the deployment scenarios:

- The node can be provided with IP addresses in a configuration file;
  - This would be the default option for an exchange or stakepool node, or in an enterprise setting;
- The node can be provided with DNS names in a configuration file;
  - This would be the default option for smaller nodes, including wallets

Once a node has succeeded in connecting to one peer, that peer can provide addresses of other peers that it knows; note that the receiving node does not need to trust the addresses that

it is given, as it can try them out for itself and make dynamic decisions about which peers it uses as described in Section 5.7. This exchange of information about (the addresses of) other nodes is called 'gossiping', and is required for the fully decentralised version of Shelley. The implementation that will be taken here is inspired by Kademlia, but can be far less complex.

10 Outstanding and unresolved issues

10.1 Cold/black start scenarios

Cold start scenarios arise when some event has taken all block-producing nodes and other intermediate nodes offline for some time. Examples include natural disasters such as solar storms, or catastrophic software or configuration failures.

In such a scenario, restarting the system cannot be achieved by the network layer alone. These scenarios break the assumptions of Ouroboros and handling them may require Ouroboros protocol modifications, and may also require additional out-of-band communication between stake pool operators. This requires further analysis.

10.2 Resources and decentralisation

The size ratio between empty and full blocks is more than 3000, and to create full blocks a similar amount of data must be moved in transactions. Thus the minimum amount of data to be received by each node varies from ~1kb/20s (system idle) to ~2Mb/20s (leaf node) to ~4Mb/20s (full node). The amount of data transmitted by a full node will exceed this in proportion to the number of peers downloading from it.

This creates a risk (which occurs in practice in other industries) that the capacity of nodes may be dimensioned on the basis of historical 'typical' load (i.e. mostly empty blocks), so that if a burst of large transactions occurs requiring full-sized blocks to be created, such nodes will not be able to keep up. If such load is sustained, the integrity of Cardano might be at risk.

This is an example of a hazard which, although it appears distributed, has a correlate (and part of potential normal operation) criteria for arming that hazard, with an associated reputational risk. Although there are node configuration parameters to contain this problem, their correct use cannot be enforced once the system is decentralised

There are a number of non-technical risks related to third-party behaviour in Ouroboros/Cardano related to the operation of stakepools. Typical industry practice for handling this is to create a certification / conformance programme, i.e. tie this into aspects of branding. Assuring the veracity of stakepool holders claims about the resource allocation and other stakepool operation is, perhaps, something that the Cardano Foundation could help manage through a suitable conformance / certification programme.

11 Annexes

11.1 Business requirements

The high level business requirements reproduced below were signed off in late 2017.

They are expressed in informal prose, often following a "user story" style. Roughly, there are three kinds of users:

1. users who have delegated,
2. small stakeholders, and
3. large stakeholders.

11.1.1 Network connectivity

Participate as a user who has delegated As a Daedalus home user with my stake delegated to other users I would like to join the Cardano network so I can participate in the network.

1. The system must be designed to provide this user segment with the ability to catch up and keep up with the blockchain without having to do any local network configuration.
2. The system must be designed to provide this user segment with the ability to continuously find and maintain a discovery of a sufficient number of other network participants that have reasonable connectivity.
3. The system must be designed to provide this user segment with the ability to find and maintain a minimum of 3 other network participants to maintain connectivity with performance that is sufficient to catch up with the blockchain.
4. The system design will take into account that this user will probably be behind a firewall.
5. Users in the segment can be defined by having all their stake delegated to other network participants. As such they will never be selected as a slot leader (i.e required to generate a block).

Participate in network as small stakeholder As a Daedalus home user operating a node with a small stake, I would like to join the Cardano network so I can participate in the network as a node that produces blocks i.e. my stake is not delegated to someone else.

1. The system must be designed to provide this user segment with the ability to receive the transactions that will be incorporated into blocks (although sizing the operation of the distributed system to ensure that all such participants would be able to receive all transactions is not a bounding constraint).
2. The system must be designed to provide this user segment with the ability to participate in the MPC protocol[95].
3. The system will be designed to provide this user segment with the ability to catch up and keep up with the blockchain without having to do any local network configuration (this is a bounding constraint).
4. The user will have sufficient connectivity and performance to receive a block within a time slot AND they have to be able to create and broadcast a block within a time slot in which the block is received by other participating nodes.
5. The system will be designed to maximise the likelihood that 50% of home users operating a participating node are compliant with the previous requirement at any one time.
6. The system will be designed to provide this user segment with the ability to continuously find and maintain a discovery of a sufficient number of other network participants that have reasonable connectivity.
7. The system will provide a discovery mechanism that will find and maintain a minimum of 3 other network participants to maintain connectivity with performance that is sufficient to catch up with the blockchain.
8. The system design will take into account that this user may be behind a firewall (i.e being behind a firewall should not preclude a user participating in this fashion).

[95]Note that this requirement is now redundant as the MPC protocol is required only for Ouroboros Classic

9. The Delegation workstream will provide a UI feature for the user to choose to control their own stake.

10. Users in this segment will be defined as NOT

    (a) being in the top 100 users ranked by stake or

    (b) in a ranked set of users who together control 80% of the stake

11. Users in this segment will not be part of the Core Dif, but still subject to the normal incentives related to creating blocks.

Participate in network as a large stakeholder As a user running a core node on a server and with large stake in the network, I would like to join the Cardano network so I can participate in the network as a core server node that produces blocks i.e. have not delegated to someone else.

1. A large stakeholder will be defined as

    (a) being in the top 100 users ranked by stake; or

    (b) in a ranked set of users who combined control 80% of the stake

2. Assuming that this user has sufficient connectivity and performance, the system should ensure that the collective operation of the distributed system will ensure that they have a high probability of receiving a block within a time slot such that they have sufficient time to be able to create and broadcast a block within a time slot where the block is received by other core nodes.

3. It is expected that the previous requirement will be fulfilled to a high degree of reliability between nodes in this category – assuming normal network operations

Threshold	$> 95\%$
Target	$> 98\%$
Stretch	$> 99\%$

4. The system will be designed to provide this user segment with the ability to continuously find and maintain a discovery of a sufficient number of other network participants that have reasonable connectivity.

5. Discovery will find and maintain a minimum of 10 other network participants to maintain connectivity with performance that is sufficient to catch up with the blockchain.

6. Ability to receive the transactions that will be incorporated into blocks.

7. Ability to participate in the MPC protocol[96].

8. The user will catch up and keep up with the blockchain.

9. The server firewall rules will be such that it can communicate with other core nodes on the system (and vice versa) – The system will provide the necessary information to update firewall rules if the server is operating behind a firewall to ensure the server can communicate with other core nodes.

[96] Note that this requirement is now redundant as the MPC protocol is required only for Ouroboros Classic

10. The threshold which defines the group of large stakeholders may be configurable on the network layer. The configuration may include toggling between the rules a) and b) in the previous requirement and the threshold numbers within these (this is pending a decision from the Incentives workstream.

11. The rules and threshold configuration may need to be a protocol parameter that is updated by the update system.

11.1.2 Network performance

Poor network connectivity notification As a home user, I want to see a network connection status on Daedalus so that I know the state of my network connection.

- If the user receives a notification that they are in red or amber mode, Daedalus will give the user some helpful information on how to resolve common connectivity issues.

There are three (at least) the following three distinct modes that the network can be operating in: each one has a red, green, amber status.

Initial block sync	
red	receiving $<$ 1 blocks per 10s
amber	receiving $<$ 10 blocks per 10s
green	otherwise
Recovery	
red	receiving $<$ 1 block per 10s
amber	otherwise
green	(not applicable)
Block chain following	
red	it has been more than 200s since a slot indication was received.
amber	it has been more than 60s since a slot indication was received.
green	otherwise.

This assumes that the slot time remains 20 seconds[97].

Transaction Throughput The transaction per second of the system as a whole will be:

Threshold	8 tps
Target	16 tps
Stretch	50 tps

This assumes that the slot time remains 20 seconds.

Network Bearer Resource Use – end user control As a user operating on the network as a home user not behind a firewall, I would like a cap on the total amount of network capacity in terms of short-term bandwidth that other network users can request from my connection so I am assured my network resource is not eaten up by the data diffusion function.

[97] Under Praos this should be interpreted as the average time between production of new blocks is 20 seconds.

1. The cap should be based on a fraction of a typical home internet connection – it can be changed by configuration including "don't act as a super node".
2. The system will allow users syncing with the latest version of the blockchain to download blocks from more then one and up to five network peers concurrently.
3. A cap on the number of incoming subscribers.
4. A cap on number of outbound requests for block syncing from other users.
5. The cap will not be imposed on core nodes running on a server.
6. If these resources are available, a reasonable connection speed should be available to users requesting to sync the latest version of the blockchain e.g. downloading blocks from 5 peers concurrently to aggregate the bandwidth.
7. (nice to have) the actual number and capacity being used is available to users.

Participant performance measurement There may be a requirement for measuring if a large stakeholder is not meeting their network obligation Brünjes et al. (2018).

It is accepted that this requirement is a "nice to have", and it has not been established that it is possible, nor has it been incorporated into the incentives mechanism.

11.1.3 Distributed System Resilience and Security

Resilience to abuse As a user I should not be able to attack the system using an asymmetric denial of service attack that will deplete network resources from other users.

1. The system should achieve its connectivity and performance requirements even in the presence of a non-trivial proportion of bad actors on the network.
2. There is an assumption that there are not large numbers of bad actors in the network.
3. The previous assumption does not follow from the assumptions of Ouroboros which states that the users that control 50% of the stake are non-adversarial.

DDoS protection As a large stakeholder running a core node on a server, I should still be able to communicate with other users in this segment, even if the system comes under a DDoS attack.

- Users in this segment will be able to generate and broadcast blocks to each other within the usual timing constraints in this situation.

IP addresses will be hidden.

- Encrypted IP addresses will be published by 10 of the other members of the group of large stakeholder core nodes.

Assumption

- Core node operators will not publish their IP addresses publicly.
- Encrypted IP addresses will be published by the 10 of the other members of the group of large stakeholder core nodes.
- If a node operator's IP address is compromised the operator will respond and change the IP address of their node.
- The system will allow operators to change the address of their core nodes and communicate with that new IP address within a reasonable period of time.

11.1.4 Network decentralisation

No hegemony As a user I want to be assured that IOHK and its business partners are not in an especially privileged position in terms of trust, responsibility and necessity to the network so that network hegemony is avoided.

- IOHK should be in the same position on the network as any other stakeholder with an equivalent amount of stake.
- There is a more general requirement that no other actor could achieve hegemonic control of the operation of the data diffusion layer.

11.2 TCP RPC response behavior

In Shelley we are using (pre-established) TCP/IP bearers, over which information exchanges are multiplexed.

1. Given the interval between block diffusions we are assuming that the TCP/IP congestion window will be reset to its *IW* (initial window)[98].
2. There is a plethora of options for TCP/IP window scaling algorithms[99], but changing them on a particular machine requires superuser access, which violates the constraint in Section 11.1.
3. As our exemplar environment here we are considering AWS linux deployments, specifically contemporary AWS Ubuntu images.

11.2.1 Time to transmit a block of given size across given latencies

The $\Delta Q|_G$ value[100] here is for the round trip time, the $\Delta Q|_S$ is 8×10^{-8} s/o[101] (the typical value from tests), and the ΔQ per direction is taken to be $1/2$ the round trip (reasonable in an AWS environment, not so in an asymmetric networking case e.g. residential, SME)

Note that window size is always restricted in practice: the unrestricted window case is presented here to capture the causality restriction within the slow-start TCP paradigm.

The tables show the time in seconds to achieve the bulk transfer (as per the outcome description above) – their likely accuracy is ± 1 RTT (left hand column) due to various vagaries of how actual behaviour is context sensitive. Note that this does not model the initial request time, but also note that the IW size can be 10, which reduces RTT count by one or so, so these effects are cancelled out for longer transfers as the actual "bandwidth-delay product" dominates those (i.e. once the TCP window is fully open). The red 'danger' colour shows block transfer times that risk missing the targets set in Section 11.1.

The table 14 represents the "best credible case" (between AWS instances before kernel parameters have to be changed) for a large window size that can be set from within a user program.

[98] See Section 4.1 in https://tools.ietf.org/html/rfc5681 – the reset to initial conditions timer is based on the TCP retransmit timer (see https://tools.ietf.org/html/rfc6298 Section 2). For Cardano deployments within AWS this has a credible upper bound of one second (by observation and analysis), even between the most distant locations.

[99] See https://en.wikipedia.org/wiki/TCP_congestion_control – it should be noted that this work has as its goal throughput fairness under saturation of a common resource with long lived data flows, whereas our data flows are relatively short. Other flow control mechanisms that are not available in a TCP/IP environment, such as rate-based, may be more suitable for our needs.

[100] The concept of ΔQ is explained in [TD20]. Here is a link to slides on applicability to block chain.

[101] s/o - seconds per octet

Table 13: Unrestricted Window Size

$\Delta Q\|_G$ (s)	File size (ko) 2	5	10	20	50	100	200	500	1000	2000
0.002	0.001	0.001	0.003	0.005	0.007	0.010	0.012	0.017	0.025	0.042
0.005	0.003	0.003	0.008	0.013	0.018	0.023	0.028	0.035	0.044	0.061
0.010	0.005	0.005	0.015	0.025	0.035	0.046	0.056	0.068	0.080	0.097
0.020	0.010	0.010	0.030	0.050	0.070	0.091	0.111	0.133	0.155	0.180
0.050	0.025	0.025	0.075	0.125	0.175	0.226	0.276	0.328	0.380	0.435
0.100	0.050	0.050	0.150	0.250	0.350	0.451	0.551	0.653	0.755	0.860
0.150	0.075	0.075	0.225	0.375	0.525	0.676	0.826	0.978	1.130	1.285
0.200	0.100	0.100	0.300	0.500	0.700	0.901	1.101	1.303	1.505	1.710
0.250	0.125	0.125	0.375	0.625	0.875	1.126	1.376	1.628	1.880	2.135
0.300	0.150	0.150	0.450	0.750	1.050	1.351	1.651	1.953	2.255	2.560

Table 14: Large max window size (near default maximum)

$\Delta Q\|_G$ (s)	File Size (ko) 2	5	10	20	50	100	200	500	1000	2000
0.002	0.001	0.001	0.003	0.005	0.007	0.010	0.012	0.017	0.025	0.042
0.005	0.003	0.003	0.008	0.013	0.018	0.023	0.028	0.035	0.044	0.062
0.010	0.005	0.005	0.015	0.025	0.035	0.046	0.056	0.068	0.087	0.119
0.020	0.010	0.010	0.030	0.050	0.070	0.091	0.111	0.133	0.172	0.234
0.050	0.025	0.025	0.075	0.125	0.175	0.226	0.276	0.328	0.427	0.579
0.100	0.050	0.050	0.150	0.250	0.350	0.451	0.551	0.653	0.852	1.154
0.150	0.075	0.075	0.225	0.375	0.525	0.676	0.826	0.978	1.277	1.729
0.200	0.100	0.100	0.300	0.500	0.700	0.901	1.101	1.303	1.702	2.304
0.250	0.125	0.125	0.375	0.625	0.875	1.126	1.376	1.628	2.127	2.879
0.300	0.150	0.150	0.450	0.750	1.050	1.351	1.651	1.953	2.552	3.454

Table 15: Medium max window size (near default)

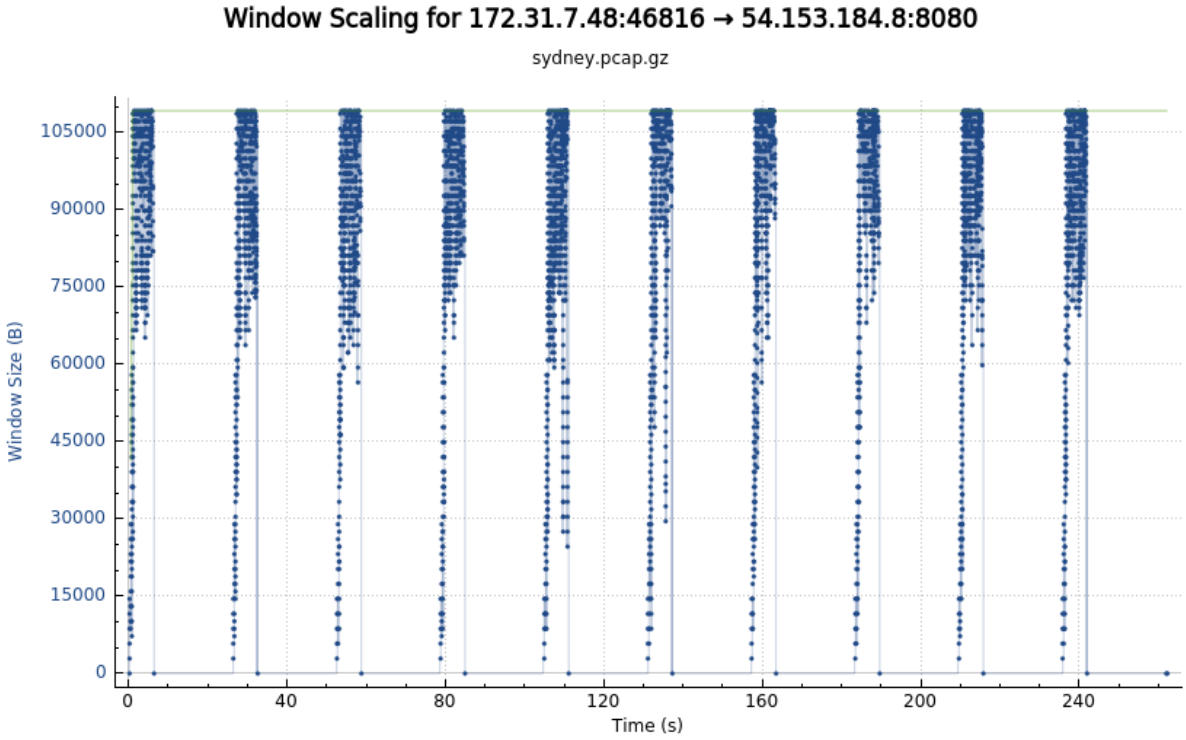
$\Delta Q\|_{G,S}$ (s)	File Size (ko) 2	5	10	20	50	100	200	500	1000	2000
0.002	0.001	0.001	0.003	0.005	0.007	0.010	0.012	0.017	0.025	0.042
0.005	0.003	0.003	0.008	0.013	0.018	0.023	0.028	0.039	0.055	0.090
0.010	0.005	0.005	0.015	0.025	0.035	0.046	0.056	0.076	0.107	0.178
0.020	0.010	0.010	0.030	0.050	0.070	0.091	0.111	0.151	0.212	0.353
0.050	0.025	0.025	0.075	0.125	0.175	0.226	0.276	0.376	0.527	0.878
0.100	0.050	0.050	0.150	0.250	0.350	0.451	0.551	0.751	1.052	1.753
0.150	0.075	0.075	0.225	0.375	0.525	0.676	0.826	1.126	1.577	2.628
0.200	0.100	0.100	0.300	0.500	0.700	0.901	1.101	1.501	2.102	3.503
0.250	0.125	0.125	0.375	0.625	0.875	1.126	1.376	1.876	2.627	4.378
0.300	0.150	0.150	0.450	0.750	1.050	1.351	1.651	2.251	3.152	5.253

Note a $\Delta Q|_{G,S}$ of (300ms, 8×10^{-8} o/s) is not the absolute worst case – the worst we've measured (in other work) is $\Delta Q|_{G,S}$ of (188ms, 6.14×10^{-5} o/s) *one-way* between San Paulo and Singapore. This would give a time-to-complete of 5.38s in the 2000 ko transfer case.

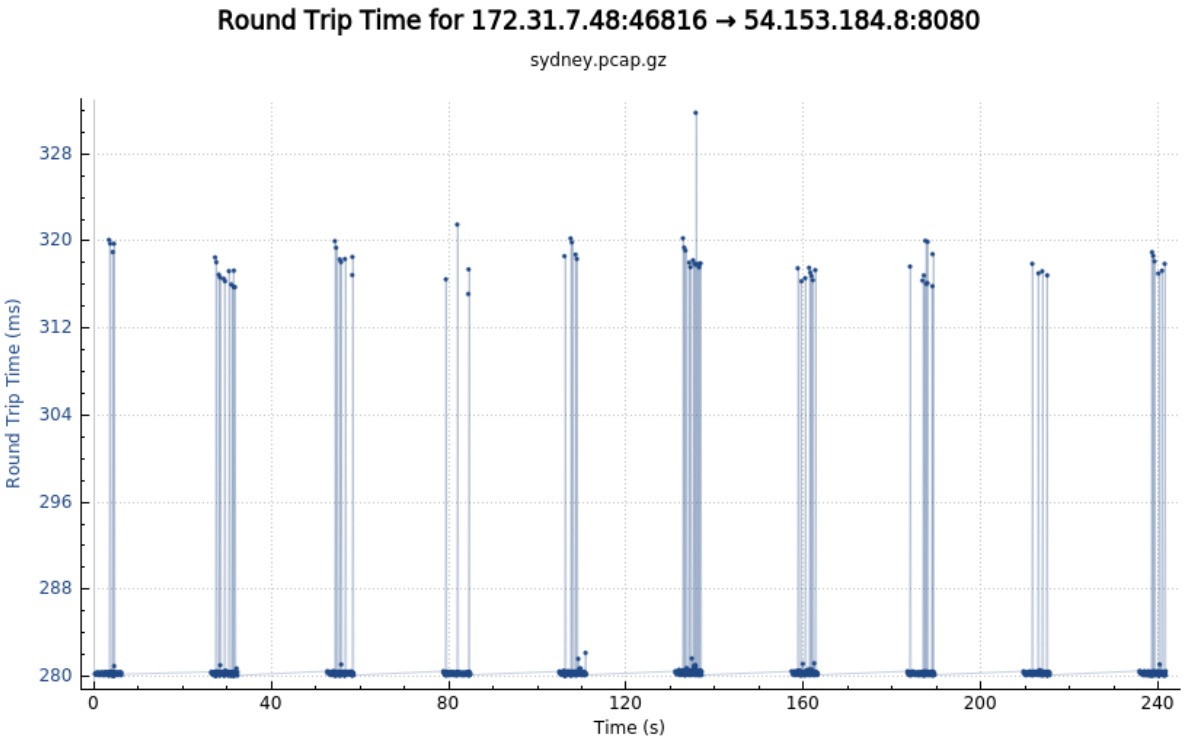
The table 15 represents the default case – i.e. no special action on the code and hence acceptance of the contemporary window size default.

11.2.2 Examples of TCP/IP window opening between London and Sydney

The figure below shows the window size over a four-minute run:



The following figure shows the contemporaneous round trip (as reported from TCP):



Note that the outliers in round trip time are (most likely) relating to those packets that were at the end of the round trip time / window closing – most ACKs would have been triggered by the "two packet rule"; if an odd number of packets arrived just before a pause, its acknowledgement would be dependent on the ACK timeout (˜40ms).

11.3 Model of network scaling

Table 16 below shows how the overall data diffusion time budget translates into a per-hop budget as a function of the depth of the spanning tree.

	Data diffusion time budget		
	Threshold	Target	Stretch
Max hops	**20s**	**15s**	**10s**
2	10.0*s*	7.5*s*	5.0*s*
3	6.6*s*	5.0*s*	3.3*s*
4	5.0*s*	3.75*s*	2.5*s*
5	4.0*s*	3.0*s*	2.0*s*

Table 16: Per-hop budget for data diffusion

These budgets should be compared with the time-to-complete for delivering a block in Section 11.2.1.

Table 17 above shows the size of a (homogeneous) spanning tree as a function of depth and valency. The shading represents increasing 'success' in reaching a large number of nodes. However, as discussed in Section 5.6, a large depth is problematic because it reduces the per-hop time budget, and a large valency is problematic because it increases the resources required by a node. The figures in bold italics represent reasonable compromises.

11.4 Performance model of Ouroboros Praos

	Valency				
Depth	2	3	4	5	6
2	7	26	63	124	215
3	15	80	255	***624***	1295
4	31	242	***1023***	***3124***	7775
5	63	728	***4095***	***15624***	46655
6	127	2186	16383	78124	279935

Table 17: Size of spanning tree as a function of depth and node valency

Parameter	Description	Notes
N	Number of active nodes	Expected to be ˜1000
T_s	Duration of a slot	Expected to be ˜ 2s
f	Active slot fraction	$0 < f \leq 1$
Δ	Maximum number of slots before a diffused message is received	$\Delta \geq 1$
k	Depth of chain for effective immutability	

11.4.1 Distribution of leadership

From [BGKR17], the probability of stakeholder U_i with relative stake α_i being chosen in any slot is:

$$\begin{aligned} p_i &= \phi_f(\alpha_i) \\ &= 1 - (1-f)^{\alpha_i} \end{aligned}$$

We assume here that each of the N active nodes has an equal amount of stake[102], $\alpha_i = \frac{1}{N}$, and hence equal probability of being a leader in any particular slot. Then:

$$p_i = 1 - (1-f)^{\frac{1}{N}}$$

The probability that stakeholder U_i is not the leader is $1 - p_i = (1-f)^{\frac{1}{N}}$, and so the probability that no stakeholder is the leader (i.e. we have an empty slot) is[103]:

$$\begin{aligned} P(\text{no leader}) &= \left((1-f)^{\frac{1}{N}}\right)^N \\ &= 1 - f \end{aligned}$$

(Hence the definition of f as the active slot fraction). If the probability of leadership was directly proportional to stake it would be $\frac{f}{N}$ for each node. The error relative to the true function above is:

[102]The incentive system is designed to encourage a uniform distribution of stake between major nodes.
[103]Each node decides independently whether it is the leader, so we just multiply probabilities.

$$
\begin{aligned}
E_{f,N} = p_i - \frac{f}{N} &= 1 - \frac{f}{N} - (1-f)^{\frac{1}{N}} \\
&\simeq 1 - \frac{f}{N} - \left(1 - \frac{f}{N} + \frac{(\frac{1}{N} - 1)}{2N} f^2\right) \\
&\simeq \frac{f^2}{2N}
\end{aligned}
$$

For reasonably large N and small f we could ignore this error term, and use the linear approximation[104]. This would imply:

$$
P(\text{one leader}) = f\left(1 - \frac{f}{N}\right)^{N-1}
$$

In the large N limit[105] this becomes:

$$
P(\text{one leader}) = fe^{-f}
$$

Conversely, the probability of a run of m successive empty slots (since these are independent trials) is:

$$
\begin{aligned}
P_m^{\text{NL}} &\equiv P(m \text{ slots no leader}) \\
&= P(\text{no leader})^m \\
&= (1-f)^m
\end{aligned}
$$

This gives the following table:

	Active slots fraction f									
m	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
1	9.00E-01	8.00E-01	7.00E-01	6.00E-01	5.00E-01	4.00E-01	3.00E-01	2.00E-01	1.00E-01	0.00E+00
3	7.29E-01	5.12E-01	3.43E-01	2.16E-01	1.25E-01	6.40E-02	2.70E-02	8.00E-03	1.00E-03	0.00E+00
10	3.49E-01	1.07E-01	2.82E-02	6.05E-03	9.77E-04	1.05E-04	5.90E-06	1.02E-07	1.00E-10	0.00E+00
31	3.82E-02	9.90E-04	1.58E-05	1.33E-07	4.66E-10	4.61E-13	6.18E-17	2.15E-22	1.00E-31	0.00E+00
100	2.66E-05	2.04E-10	3.23E-16	6.53E-23	7.89E-31	1.61E-40	5.15E-53	1.27E-70	1.00E-100	0.00E+00

We can convert slots into a time interval by multiplying by Ts. Conversely, if we want to know the probability that a certain time interval t will be leaderless, we can approximate this[106] as:

$$
\begin{aligned}
P_t^{\text{NL}} &\equiv P(t \text{ seconds no leader}) \\
&= P(\text{slot has no leader})^{\frac{t}{T_s}} \\
&= (1-f)^{\frac{t}{T_s}} \\
&= \left((1-f)^{\frac{1}{T_s}}\right)^t
\end{aligned}
$$

If we set a threshold Y (say 0.999) for the probability of having at least one leader in time t, then t is bounded by:

[104] Which is simpler than the true Binomial distribution.

[105] The intention is to encourage ~1000 stake pools, which is enough for the large N limit to be quite accurate.

[106] Ignoring the quantisation of m.

$$t \geq T_s \frac{\log(1-Y)}{\log(1-f)}$$

This is relevant for the consideration of timeouts that might be triggered by a leaderless interval.

11.5 Comparison with general overlay networks

One approach to building a large-scale decentralized blockchain applications is to take existing P2P protocols designed for other applications domains (notably P2P file sharing) and use them as the distributed application infrastructure to build the blockchain itself. It is a temptingly fast approach, but it comes with a number of security, scalability and performance hazards:

1. Protocols that were designed for a specific environment (e.g. efficient location of distributed content) are not well-suited to the requirements of a blockchain. For example, the adoption of the Kademlia protocol in Ethereum facilitates low-resource Eclipse attacks on the nodes of the blockchain [MHG15].

2. Protocols in a P2P framework are usually designed for a single environment, integrated into their original application, and hard to repurpose to different requirements. This exercise usually requires a redesign and re-implementation of the protocol from scratch, since the protocol is not designed with adaptability in mind [Sli19]

3. P2P frameworks mostly focus on the construction and maintenance of the overlay structure and on routing application requests / disseminating information. Other aspects that are also important for the communication of application instances are usually not covered, such as:

    - Quality of service: ability to handle multiple classes of application traffic within the overlay and differentially allocate loss and delay to them.
    - Security: authentication, access control, confidentiality and message integrity, usually achieved through protocols that are not part of the P2P framework such as TLS or DTLS.
    - Ability to operate over multiple IP networks (e.g. public Internet or private IP networks) or other non-IP bearers (e.g. directly over Ethernet).

4. Scalability of the overlay is achieved ad-hoc. Some P2P frameworks introduce the concept of "super-peers" [Mal15], but there is no formal means of creating hierarchies within the overlay framework.

5. It is hard to reason about the correctness of a group of independently-designed protocols working in conjunction. The properties of their joint operation in production environments are going to be emergent and hence hard or impossible to control by design. The system is always going to be suboptimal from an efficiency, security and performance standpoint compared to a solution that designs the complete "communications middleware" in a coherent and integrated fashion.

A piecemeal approach to addressing these communication challenges will not deliver a robust and sustainable solution. In the same way that following a typical industry 'low assurance' approach to software development risks sinking effort into constantly debugging poorly-specified code, continuing with a 'business as usual' approach to network issues risks creating a complex and fragile solution that similarly consumes an ever-increasing fraction of the available development and maintenance resources. Instead, we need a consistent framework in

which to define and describe issues and posit and refine solutions without piling up complexity and technical debt.

Feature	Standard P2P overlay framework
Authentication, access control, confidentiality, integrity	Such functions are usually not part of the overlay framework, which relies on protocols such as TLS and/or DTLS.
Routing and forwarding	Most literature on P2P overlay frameworks is about efficient location of distributed content, leveraging DHT (Distributed Hash Table) technology to achieve consistent access this goal [Mal15]
Support for multihoming	Rely on limited definition of multihoming (load balancing only) provided by IP-based technologies (e.g. multipath TCP or SCTP) or implement specialized protocols
Support for node mobility (e.g. wallet nodes)	Relies on home and foreign agents, tunnels, anchors, and specialized protocols to construct and maintain the overlay connectivity graph, which are use-case specific [MA13]
Efficient operation over heterogeneous physical media (e.g. fibre, satellite, cellular, WiFi, etc.)	Would require development of a specialized transport protocol within the overlay with support for congestion management. Interconnection with other IP systems dependent on widespread adoption / specialist gateways. Not addressed by P2P overlay frameworks
Support for QoS (traffic differentiation, routing)	Multiple techniques and algorithms available in the literature [RS14], but these are hard (that is, not achieved end-to-end in practice after ˜50 years since first proposed) to integrate into a coherent solution
Extensibility / adaptability	Protocols designed for a single purpose, need to be redesigned for different environments [Sli19]
Efficiency of the system specification and implementation	Each "control protocol" specifies its own encoding format, information modelling and dissemination mechanisms.
Structured / generic approach towards scalability	No, ad-hoc extensions (e.g. "super-peers" [Mal15])
Support for non-IP bearers?	No, designed to be overlaid on top of IP networks
Collaborative management of the overlay	Hard to manage a group of independently designed protocols working in conjunction: lack of commonality and of a consistent theory of operation

11.6 Time synchronisation constraints

Ouroboros Praos (in common with the original 'classic' version) assumes that the notion of a time-slot is unambiguous, which requires participating nodes to have adequately synchronised local clocks. The assumption is made in [BGKR17] that "any discrepancies between parties' local clocks are insignificant in comparison with the length of time represented by a slot". Since in

reality there will be skew between the local clocks of participating nodes, this implicitly places a lower limit on the slot time T_s, and potentially opens a new attack vector in which an adversary manipulates the time reference of honest parties.

The current implementation approach is to rely on the NTP service maintained by OS kernels. This has a number of performance and security constraints; security concerns are in principle mitigated by the 'secure' variant of the protocol, but few servers support this due to the computational cost of the cryptographic operations required. In terms of performance, there is little reliable research, but the consensus is (from [Mil12]):

> As a rule of thumb, the accuracy over the Internet is proportional to the propagation delay. For a lightly loaded 100-Mb/s Ethernet, the accuracy is in the order of 100μs. For an intercontinental Internet path, the accuracy can be up to several tens of milliseconds.
>
> On network paths with large asymmetric propagation delays, such as when one direction is via satellite and the other via landline, the errors can reach 100 ms or more. There is no way these errors can be avoided, unless there is prior knowledge of the path characteristics.

Taking 100ms as the credible worst-case, if we interpret 'insignificant' as $< 5\%$, this means the minimum slot time T_s should not be less than 20 x 100ms = 2s, unless steps are taken to mitigate sources of NTP inaccuracy. Further research is advisable into the consequences of violating the clock-synchronisation assumption of [BGKR17], since Cardano is otherwise potentially vulnerable to NTP-spoofing attacks.

11.6.1 Leap seconds

A further complication arises from the occasional insertion of 'leap seconds' into UTC. A leap second is a one-second adjustment that is occasionally applied to civil time Coordinated Universal Time (UTC) to keep it close to the mean solar time at Greenwich, in spite of the Earth's rotation slowdown and irregularities. UTC was introduced on January 1, 1972, initially with a 10 second lag behind International Atomic Time (TAI). Since that date, 27 leap seconds have been inserted, the most recent on December 31, 2016 at 23:59:60 UTC, so in 2018, UTC lags behind TAI by an offset of 37 seconds. When it occurs, a positive leap second is inserted between second 23:59:59 of a chosen UTC calendar date and second 00:00:00 of the following date. The definition of UTC states that the last day of December and June are preferred, with the last day of March or September as second preference, and the last day of any other month as third preference. All leap seconds (as of 2017) have been scheduled for either June 30 or December 31.

Leap seconds do not pose a problem to Praos by themselves; however, the widespread ambivalence towards them and varied/patchy implementation of them does. Not all clocks implement leap seconds in the same manner as UTC [GvB14]: leap seconds in Unix time are commonly implemented by repeating the last second of the day; Network Time Protocol freezes time during the leap second; other schemes such as those deployed by Google and Amazon in their public cloud infrastructure smear the lengths of seconds in the vicinity of a leap second. It would thus be prudent to avoid generating blocks in the immediate vicinity of a leap second, but this would require knowing when this is going to happen. Although the decision about insertion of a leap second is made a long time in advance, most time distribution systems (SNTP, IRIG-B, PTP) only announce leap seconds at most 12 hours in advance and sometimes only in the last minute.

It seems that at worst one validly-generated block may be rejected by other nodes depending on a temporary time difference, which represents a level of 'adversarial' behavior that Praos can easily withstand. However, inconsistent implementations of leap second insertion make the problem much worse. For example, the NTP packet includes a leap second flag, which

informs the user that a leap second is imminent. It has been reported [Mal16] that never, since the monitoring began in 2008 and whether or not a leap second should be inserted, have all NTP servers correctly set their flags on a potential leap second day. This is one reason many NTP servers broadcast the wrong time for up to a day after a leap second insertion. Detailed studies of the leap seconds of 2015 and 2016 show that, even for the Stratum-1 servers which anchor the NTP server network, errors both in leap second flags and the server clocks themselves are widespread, and can be severe [CV16].

Thus, even with all Cardano nodes using NTP, local variations in the propagation of a leap second could result in persistent differences in local clock values over many slot times. This clearly violates the assumption in the Praos paper that differences between local clocks are insignificant; deeper analysis would be required to understand the consequences of this to normal operation and what opportunities it represents for an adversary.

Another problem is that different time standards do not recognise the accumulation of leap seconds. International Atomic Time (TAI) is exactly 37 seconds ahead of UTC (the 37 seconds results from the initial difference of 10 seconds at the start of 1972, plus 27 leap seconds in UTC since 1972). GPS time remains at a constant offset with TAI (TAI − GPS = 19 seconds), and hence at a constant offset with UTC, although this offset changes whenever a leap-second is introduced. If a Cardano node happened to be synchronised with GPS time, say, and failed to account for the offset to UTC, it would be unable to participate properly in the protocol.

11.7 Ouroboros Network Components

The *Ouroboros Network* implementation consists of the following Haskell packages:

1. `typed-protocols` – a standalone session type framework, which allows communication protocols to be expressed and enforced;
2. `typed-protocols-cbor` – a small extension to provide codec support for the CBOR format;
3. `network-mux` – a standalone multiplexing library;
4. `ouroboros-network-framework` – a library which makes low level choice for *ouroboros-network* and implements low level network components / primitives, e.g. connection manager, server.
5. `ouroboros-network` – a library which implements the consensus application-level protocols to support *node-to-node* as well as *node-to-client* communication (each consists of a bundle of *mini-protocols* expressed with *typed-protocols* package), and all the required architecture to run them.

For the larger picture how all the network components fit together with the consensus components in a full node, see the diagram in Section 5.2.

All the *node-to-node* mini-protocols (*chain-sync*, *block-fetch* and *tx-submission* protocols) are multiplexed over a single bearer (e.g. TCP socket) using a network-mux package. For a *node-to-node* (similarly for *node-to-client* protocol) the networking components are schematically aligned in the following stack:

The bearer is an abstraction which allows reading / writing bytes from the underlying network (TCP socket, Unix socket, Windows named pipes, etc.). Protocols are expressed using the session type framework.

11.7.1 Typed Protocols

Typed protocols is a standalone library which implements a form of *session types*. It allows expression of both a server and a client which stay dual to each other (i.e. being in the same state,

and only one side having the agency to send a message) throughout the evolution of the protocol. In particular this also means that one cannot have a deadlock (when two nodes are waiting for the next message), or being in different states (which would result in receiving an unexpected message). This library is novel in that it allows the addition of pipelining, which gives good performance through latency hiding. This library gives us a way to write protocols in which correctness is proven by the compiler and still can provide necessary performance characteristics.. Session types have been a main research topic at some of the best academic centers, and we received interest from well-known scientists in the field (prof. Philip Wadler) as well as from the Haskell community during various events where the framework was presented[107].

The only prerequisite for using *typed-protocols* is to use bearers that guarantee order of delivery. Currently, we target TCP, but there are possible extensions of UDP that could satisfy this constraint, using other low level network protocols is also plausible. Within a local computational context we can use named pipes as the bearers, which removes a lot of complexity.

11.7.2 Network-Mux

network-mux is a general multiplexing library which allows us to run multiple protocols over a single bearer. The multiplexer avoids head-of-line blocking, guarantees fairness and is designed to use minimal resources.

On the egress end (outbound messages) mini-protocols are competing for a common resource. The important part of the technical design is to impose fair share of the resource. In this context we define (and guarantee) fairness defined as:

- no starvation;
- when presented with equal demand (from a selection of mini protocols) deliver equal service.

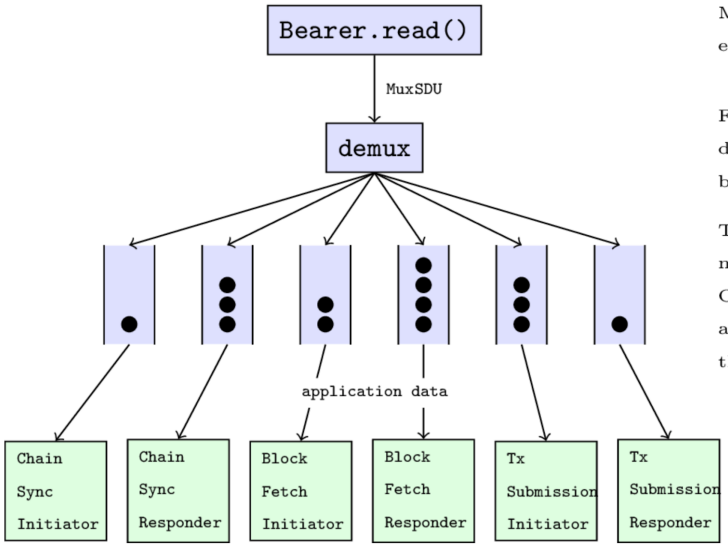
network-mux is designed to support full-duplex bearer capabilities. This means it can run initiator / responder (client / server) parts of a mini-protocol at both ends of the connection simultaneously.

This component is independent of typed-protocols (or any protocol library), and thus can be used for many different purposes.

Data Flow in Network-Mux

[107] The framework was presented by Duncan Coutts at Haskell eXchange 2019 , and Marcin Szamotulski at Monadic Party 2019.

Incoming Data path (Ingress path)

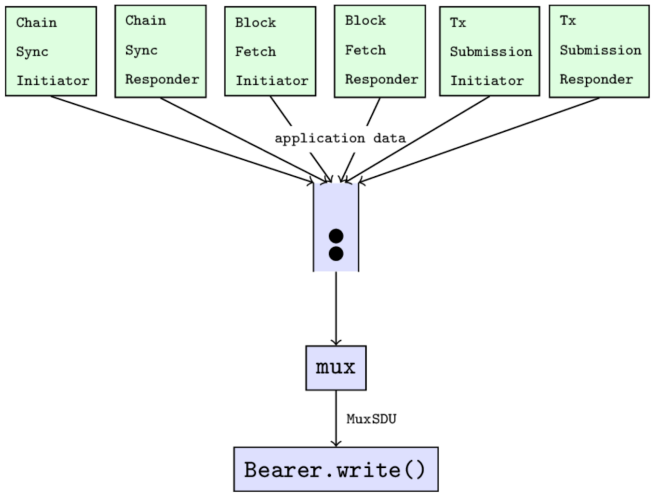


Mux Bearer implementation (Socket, Pipe, etc.)

For a given Mux Bearer there is a single demux thread reading from the underlying bearer.

There is a limited queue (in bytes) for each mode (responder/initiator) per miniprotocol. Overflowing a queue is a protocol violation and a MuxIngressQueueOverRun exception is thrown and the bearer torn down.

Outgoing Data Path (Egress Path)



Every mode per miniprotocol has a dedicated thread which will send ByteStrings of CBOR encoded data.

For a given Mux Bearer there is a single egress queue shared among all miniprotocols. To ensure fairness each miniprotocol can at most have one message in the queue.

The egress queue is served by a dedicated thread which chops up the CBOR data into MuxSDUs with at most sduSize bytes of data in them.

11.7.3 Ouroboros-Network

The *ouroboros-network* library implements *mini-protocols* for *node-to-node* and *node-to-client* communication using the typed-protocols package; it provides the means to run them over TCP sockets, unix sockets, Windows named pipes and can be easily extended to unix pipes. It also exports an API for writing nodes and clients.

A *node-to-node* protocol consists of:

- *chain-sync* mini-protocol – a protocol which allows a node to reconstruct a chain of an upstream node;
- *block-fetch* mini-protocol – a protocol which allows a node to download block bodies from various peers;
- *tx-submission* mini-protocol – a protocol which allows submission of transactions.

A client is a standalone local application which communicates with a node using a unix-socket (or named-pipe on Windows). This avoids the design flaw in the Byron era, which is using tcp-sockets and thus needs to use tls. This has been a source of frustration for many end-users).

Example local clients are: wallets, explorers, command line tools for various third-parties (e.g. exchanges).

The *node-to-client* protocol consists of:

- *chain-sync* mini-protocol
- *local-tx-submission* mini-protocol
- *local-state-query* mini-protocol

From the networking perspective it is not important which flavour of Ouroboros protocol is being used, and all the protocols reflect that (the current Haskell implementation is using polymorphism to achieve this goal). Both *node-to-client* and *node-to-node* protocols are using the same *chain-sync* protocol, but the difference is that for the former we are only sending headers through the network, while the latter is sending whole blocks.

Block Fetch Component Block fetch logic is the component responsible for downloading blocks from connected peers. It runs the client and server sides of the block fetch protocol, block fetch state per peer and a block fetch logic component which orchestrates all the downloads. The block fetch logic makes decisions from whom to download a block (as blocks may be available from various peers).

Handshake Mini Protocol On all incoming connections, before starting the multiplexer we run a handshake protocol. It allows agreement on protocol parameters. It is designed so that the whole mini-protocol conversation fits into a single round trip (at the TCP level).

Currently, in version 'V1' we only support network magic, which supports distinguishing various networks, e.g. stage-net, test-net, main-net. The protocol is extensible in the sense that any future version can support its own set of protocol parameters.

Other components of Ouroboros-Network package We implemented a peer-to-peer governor which role is to drive decisions about promotions, demotions and gossip. It is there to provide the rest of the system with possible peers to communicate with and govern their state (hot peer / worm peer / cold peer). We are implementing a connection manager which will manage active connections and threads and will provide low level primitives which are to be used by the peer-to-peer governor. The peer-to-peer governor is flexible enough to handle various deployment scenarios as well as internal usage - we plan to use it not only in a decentralised setting but also to subscribe clients (wallets, explorers) over a local bearer to a node.

Anchored Fragments This is a small part of the Ouroboros-Network package which implements an efficient chain data structure: 'AnchoredFragment'. It is hidden at the bottom of the stack since it is useful in the 'BlockFetch' component and in various *Ouroboros-Network* tests.

The implementation represents the sequence of headers using a finger tree data structure instantiated for block headers to enable efficient O(log n) operations based on slot numbers and block numbers, in addition to the normal efficient sequence operations.

11.7.4 Simulator environment IOSim

Source Lines of Code:

Component	*sloc*	*sloc of tests*
io-classes	***1148***	-
io-sim	***822***	***601***

We wrote a simulation environment for the *IO* monad (which is called *IOSim* or *sim-io*). This is similar to the system that was used to test the Bitcoin backbone protocol is [MJ15]. This, unlike the other components, is very Haskell-specific, and thus its description below will use language which might be more familiar for Haskell developers. It is a pure monad (in the same sense as the *Identity* functor or *Free* monad from the *free* package). The simulator monad supports the following interfaces:

- software transactional memory *STM*
- concurrency in the form of low level *fork* interface and the *async* package
- *ST* monad interface
- synchronous and asynchronous exception handling
- timers *API*
- monotonic time

The interface is presented as a set of type classes; its instances are written with the following principles:

- the *IO* instances are as close to the *base*, *stm* and *async* packages as possible (we have very few places where we slightly diverge).
- the *Sim* instances are faithful to *IO* instances.

This allows us to write computations that can be interpreted both in *IO* and the simulator monad *IOSim*.

Currently *IOSim* provides a single scheduling policy, but it is possible to extend it, which would improve assurance for tests expressed using *IOSim* monad.

The time in *IOSim* is simulated, which allows running tests that otherwise would take a very long time to complete. Not surprising this was very useful for testing networking code, but also turned out to be very useful testing the *ouroboros-consensus* package maintained by the consensus team.

The simulator environment allows collection of a trace of a computation (similar to Haskell's eventlog). This is valuable for constructing complex property tests of some of the components and it has proven useful in finding low-level bugs that would be extremely difficult to diagnose in a running system.

The *stm* interface is more general than that of the *stm* package. It defines strict and lazy versions of stm mutable variables, what allows tracing of memory leaks.

The hierarchy of classes is slightly different than that of standard Haskell packages, and it was designed to help in the reviewing process (both during development, and for external evaluation). Standard Haskell instances are also exported.

11.7.5 Testing Strategies

All the protocols are tested on several different levels:

- the *typed-protocol* level, the tests prove that the implementation meets the specification. The *type-protocol* package turns test errors into type errors. These tests are carried for both non-pipelined and pipelined clients.
- The wrapper API for each protocol is tested. This is an un-typed version of the previous tests. It assures that the client and server types are dual to each other. These tests are carried for both non-pipelined and pipelined clients.
- Test the protocols using example clients and servers over:
  - a channel (using *STM*'s *TMVar*'s)
  - using pipes

All the tests are done in both the *IO* monad and the *IOSim* monad.

Beside protocol tests we also maintain codecs for each protocol and a testing architecture for them. The typed protocols package exports properties which allows us to:

- test that decoding is the right inverse to encoding (e.g. decoding an encoded message brings back the original message)
- test incremental decoders, which allows guarding against boundary errors.

Some of the tests rely on the *IOSim* monad which has extensive tracing support. This allows checking invariants of the running component. This also permits us to see the evolution of state maintained by the component, and trace events that are happening in various testing scenarios. We use this technique in the *BlockFetch*, *peer-to-peer governor* as well as some other components. In all cases it allowed us to catch subtle bugs early in the development process.

References

[BGK+19] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, , and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. 2019.

[BGKR17] David Bernado, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. 2017.

[CBT+19] Nakul Chawla, Hans Walter Behrens, Darren Tapp, Dragan Boscovic, and K. Selcuk Candan. Velocity: Scalability improvements in block propagation through rateless erasure coding. 2019.

[CV16] Yi Cao and Darryl Veitch. Network timing, weathering the 2016 leap second. *IEEE Infocom*, 2016.

[Day08] John Day. *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall, 2008.

[DW13] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. 2013.

[GvB14] K. Gross and R. van Brandenburg. RFC7164: RTP and leap seconds, 2014.

[Hon93] Kohei Honda. Types for dyadic interaction. *CONCUR'93*, 1993.

[HVK98] K. Honda, V.T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. *Programming Languages and Systems. ESOP*, 1998.

[MA13] A. Mohammadi and B. Akbari. Design a peer to peer overlay protocol for mobility management of mobile nodes using bluetooth media. *International Conference on Computing, Communications and Networking Technologies*, 2013.

[Mal15] A. Malatras. State of the art survey on P2P overlay networks in pervasive computing environments. *Elsevier Journal of Network and Computer Applications*, 2015.

[Mal16] David Malone. The leap second behaviour of NTP servers, 2016.

[Mar13] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, Inc., 2013.

[MHG15] Y. Marcus, E. Heilman, and S. Goldberg. Low-resource eclipse attacks on ethereum's peer-to peer network. *24th USENIX Security Symposium*, 2015.

[MHG18] Yuval Marcus, Ethan Heilman, , and Sharon Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network, 2018.

[Mil12] David L. Mills. Executive summary: Computer network time synchronization, 2012.

[MJ15] Andrew Miller and Rob Jansen. Shadow-bitcoin: Scalable simulation via direct execution of multi-threaded applications. *8th Workshop on Cyber Security Experimentation and Test*, 2015.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. *Peer-to-Peer Systems*, 2002.

[MV11] D. Mostrous and V.T. Vasconcelos. Session typing for a featherweight erlang. *Coordination Models and Languages. COORDINATION 2011*, 2011.

[Neu19] Till Neudecker. Characterization of the bitcoin peer-to-peer network (2015-2018), 2019. associated data `https://dsn.tm.kit.edu/bitcoin/#propagation`, sampled only in Germany.

[RS14] K. Ramalakshmi and P. K. Sasikumaar. Study on security and quality of service implementations in P2P overlay network for efficient content distribution. *International Journal of Research in Engineering and Technology*, 2014.

[Sli19] Matthew Slippe. A study of libp2p and ETH2, 2019.

[SvVV12] Vinay Setty, Maarten van Steen, Roman Vitenberg, and Spyros Voulgaris. Poldercast: Fast, robust, and scalable architecture for P2P topic-based pub/sub. 2012.

[TD20] Peter Thompson and Neil Davies. Towards a rina-based architecture for performance management of large-scale distributed systems. *Computers*, 2020.

[THK94] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. *PARLE'94 Parallel Architectures and Languages Europe*, 1994.

[VFV17] Shaileshh Bojja Venkatakrishnan, Giulia Fanti, and Pramod Viswanath. Dandelion: Redesigning the bitcoin network for anonymity. *SIGMETRICS'17*, 2017.

[Wat99] Duncan J Watts. *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton University Press, 1999.