Data Diffusion and Peer Networking in Shelley

(Version 0.10)

An IOHK technical report

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

Alex Vieth
alex@well-typed.com

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

Marcin Szamotulski
marcin.szamotulski@iohk.io

Karl Knutsson
karl.knutsson@iohk.io

Marc Fontaine
marc.fontaine@iohk.io

July 25, 2019

Abstract

This document describes .....

Contents

Version history

Version 0.1, Dec 20, 2018 Draft of the table of contents.

Version 0.2, Jan 08, 2019 Structure and outline.

Version 0.9, Apr 03, 2019 Clean-up todos and WIP remarks

Version 0.10, Jun 07, 2019 Add handshake protocol

Chapter 1

Overview

The Cardano blockchain system is based on the Ouroboros family of protocols for a proof-of-stake cryptocurrency. The operation of these protocols on a node, working in collaboration with the consensus and ledger aspects of Cardano, create the overall distributed system. It is that distributed system that defines the ”single source of truth” that is the distributed ledger. The Ouroboros papers describe these protocols in a high-level mathematical formalism that allows for a concise presentation and is appropriate for peer review by cryptography experts. For separation of concerns, the papers do not describe a concrete implementation of the Ouroboros protocols.

This document has a broader scope. It is addressed to system designers and engineers who have an IT background but are not necessarily crypto experts, but want to implement Cardano or to understand the reference implementation. The description of the protocol in this document contains all the information needed to implement the data diffusion functionality of a compatibleCardano network node. It covers:

- How nodes join the network.
- The general semantics of the messages that nodes exchange.
- The binary format of the messages.
- The order in which nodes send and receive the messages of the protocol.
- The permissible protocol states for each participant in the protocol.

This information is typically found in the description of network protocols.

However, the Ouroboros proof-of-stake cryptocurrency has additional requirements, of a sort that are not typically covered in a protocol description itself. While these underlying requirements are essential for understanding the design of the protocol, it also makes sense to also discuss these aspects and requirements in this document. Typical network protocols describe simple information exchanges; the distributed nature of blockchain computation means that additional contextual information is available. Use of this context allows, for example, for validated store and forward which is an essential feature to contain the effect of potential malicious actions against the distributed system.

Shelley is a first fully decentralised release of Cardano system, implementing the Ouroboros protocol.

This Chapter contains an overview of the content and scope of this document and aspects that are being discussed.

Software assurance

Software assurance

Layered Protocols

Traditionally network protocols are presented as a stack of layers where upper layers build on services provided by lower layers and lower layers are independent of the implementation of the upper layers. This concept of layers is misleading when discussing Ouroboros. For example, it is not the case that the consensus (layer) is built on top of the network (layer). It is more appropriate to talk about a network component than a network layer. The network component provides services to the consensus component and vice versa; both components rely on each other. The network component uses the consensus component to validate information that it distributes to other nodes, which is essential to guard against certain kinds of DoS attacks. Existing peer-to-peer systems focus on a slightly different problem domain. For example, they do not consider the information validation issue or are concerned with issues such as 'eclipse attacks' that to not apply to the Ouroboros family of protocols.

Performance of the Ouroboros Network

In computer science, Byzantine Fault Tolerance is a property of a distributed algorithm, which states that it works for the honest participants under the assumption that a certain proportion of the participants are indeed honest. A similar, but more informal property applies to performance of the Ouroboros network as well.

The network provides a service to its participants while at the same time the participants provide a service to the network. The performance of the Ouroboros network depends (among other things) on the performance of the nodes, while the performance of a node also depends on the performance of the network. Not only are there honest and adversarial participant, but there is also a huge variety of possible network topographies, bandwidths and latencies of network connections and other factors that determine the performance of the network.

This document discusses the high level functional and performance requirements for Ourobors and the assumptions made about the structure of the underlying P2P network.

Protocol vs Implementation

Network Protocols are written at different levels of abstraction. To be useful, a protocol description must be precise enough to be implemented. A protocol description should also be abstract enough to allow alternative implementations of the protocol and to facilitate developing and improving it. Furthermore, it should be possible to both implement an abstract version of a protocol and interpret it in a real-world scenario. For example, it must be possible to implement a protocol such that the real-world software runs on a machine with a typical size of memory and a typical speed network connection.

The Shelley network protocol design has been developed in parallel with a reference implementation in Haskell. Haskell (more precisely GHC) has built-in support for high level concurrency abstractions such as light-weight threads and software transactional memory. While the protocol itself is completely language agnostic, it still makes sense to discuss some aspects of the Haskell reference implementation in this document. In particular, the document describes how to achieve good resource bounds for a protocol implementation.

Threats Reference the Threats section. 'eclipse' can be deterred
TODO:extended abstract, scope of the document

1.1 High level requirements and User Stories

These are the high level business requirements for the networking that were gathered and signed off in late 2017. As such they are expressed in informal prose, often following a "user story" style. Roughly, there are three different kinds of users:

- Users who have delegated.
- Small stakeholders.
- Large stakeholders.

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.

- 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.
- 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.
- 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.
- The system design will take into account that this user will probably be behind a firewall.
- 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.

- 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).
- The system must be designed to provide this user segment with the ability to participate in the MPC protocol[1].
- 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).
- 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.
- 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.
- 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.
- 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.
- 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).
- The Delegation workstream will provide a UI feature for the user to choose to control their own stake.
- 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
- Users in this segment will not be part of the Core Dif, but still subject to the normal incentives related to creating blocks.

[1]This requirement is now redundant because the MPC protocol is specific to Ouroboros Classic.

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.

- 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
- 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.
- 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	$> 99\%$

- 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.
- 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.
- Ability to receive the transactions that will be incorporated into blocks.
- Ability to participate in the MPC protocol[2].
- The user will catch up and keep up with the blockchain.
- 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.
- 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.
- The rules and threshold configuration may need to be a protocol parameter that is updated by the update system.

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.

[2] This requirement is now redundant because the MPC protocol is specific to Ouroboros Classic.

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 that 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, or at least that the average time between production of new blocks is 20 seconds.

Transaction Latency As a user I want my transaction to be submitted to the blockchain and received by the target user within the following time period:

Threshold	100 seconds
Target	60 seconds
Stretch	30 seconds

The above time-frames will be achieved for > 95% of all transactions.

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.

- 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".
- 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.
- A cap on number of incoming subscribers.
- A cap on number of outbound requests for block syncing from other users.
- The cap will not be imposed on core nodes running on a server.
- 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.
- (nice to have) the actual number and capacity being used is available to user.

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.

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.

- The system should achieve its connectivity and performance requirements even in the presence of a non-trivial proportion of bad actors on the network.
- There is an assumption that there are not a large numbers of bad actors in the network.
- 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 user 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.

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.

1.2 Context and Introduction

How this work relates (in general terms) to the rest of the Shelley development and the Ouroboros papers.

1.2.1 Data Diffusion assumptions in Ouroboros

This is the "telling them what you are going to tell them" part - outline of the data diffusion and overlay network bringing out the key functional and non-functional relationships - aim to serve as a general executive overview as well as a framing of PoS issues.

- Assumptions of the mathematical model(s)
- Goal of the data diffusion functionality
- Strong requirement on collective performance
  - Chain growth quality
  - Adversarial actor assumption

1.2.2 Functional Layering

inline]pictures as to how the various functional layers relate; How the Data diffusion layer relates to Ledger etc; How data diffusion relates to point-to-point overlay network.

Non-function aspects

Performance; trustworthiness; Forwarding as an expression of confirmed "trust" (and corollary - forwarding of clearly incorrect information seen as prima-facie evidence of adversarial action.

1.2.3 Protocol Roles

Peer relationship between various nodes; Limited trust and verification; (Brief) description of the expectations and assumptions across the functionality boundaries;

Mini Protocols

Chain Sync

Block Fetch

Transaction Submission

ΔQ Measurement (not really a mini protocol, in that it is point-to-point and not part of the diffusion process itself, placed here because it "sits" on top of the overlay network. Role to generate active endpoint performance data to help optimise time-to-diffuse critical information exchanges (e.g. newly minted block diffusion)

Point-to-point Overlay Network

inline]need a picture Note that long term eclipse attacks are not an issue here (cover why a bit later); tie in chain growth requirements with need for "better" communications performance between (major) stake pools (notion of Core DIF); association explain need for

- fixed configuration (with/without others from this list)
- Core DIF
- Distributed endpoint discovery (subset of notion of peer in other approaches)

1.3 Layout of the Document

- What goes in which section ?
- In which order to read ?
- Which sections can be skipped ?

1.4 Notation

Chapter 2

Requirements

2.1 Performance Requirements and User Stories

2.1.1 Classes of Participants

todo: make a nice table

Stake pool

Small stakeholder

User who has delegated

Requirements for Participants

Requirements for Stake Pools

Services that the System should provide

There are two kinds of Requirements:

1. System capabilities for a node to take a blockchain slot creation role in the protocol.
2. What services that the system provides to the user.

Chapter 3

System Architecture

3.1 Congestion Control

A central design goal of the system is robust operation at high workloads. For example, it is a normal working condition of the networking design that transactions arrive at a higher rate than the number that can be included in blockchain. An increase of the rate at which transactions are submitted must not cause a decrease of the block chain quality.

Point-to-point TCP bearers do not deal well with overloading. A TCP connection has a certain maximal bandwidth, i.e. a certain maximum load that it can handle relatively reliably under normal conditions. If the connection is ever overloaded, the performance characteristics will degrade rapidly unless the load presented to the TCP connection is appropriately managed.

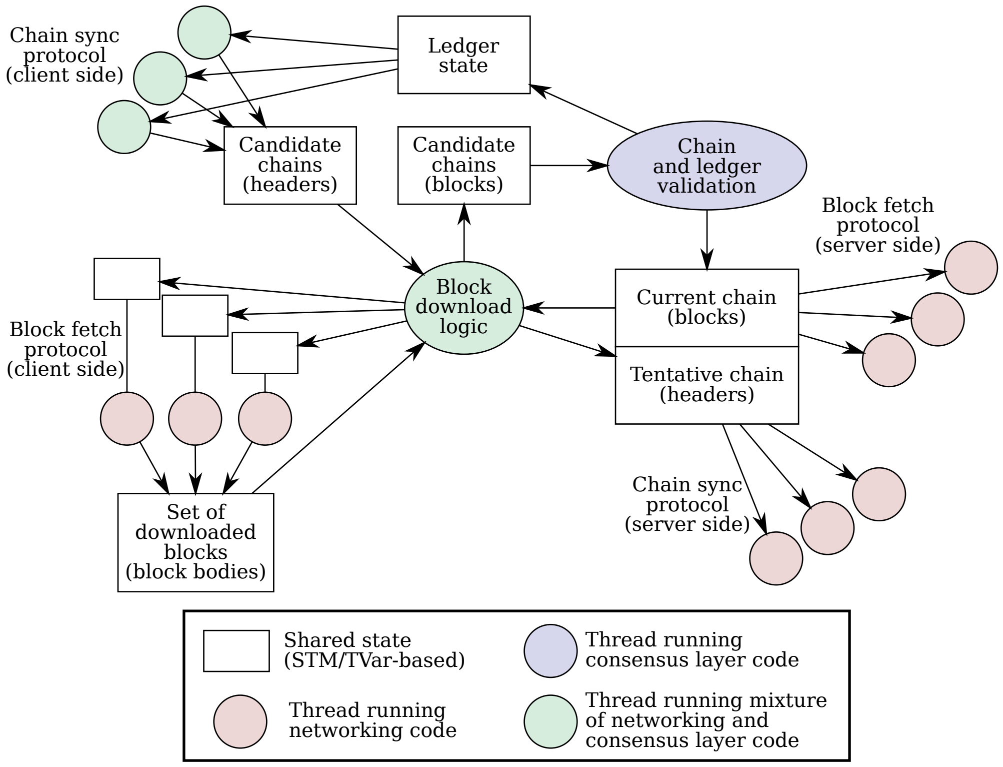
At the same time, the node itself has a limit on the rate at which it can process data. In particular, a node may have to share its processing power with other processes that run on the same machine/operation system instance, which means that a node may get slowed down for some reason, and the system may get in a situation where there is more data available from the network than the node can process. The design must operate appropriately in this situation and recover form transient conditions. In any condition, a node must not exceed its memory limits, that is there must be defined limits, breaches of which being treated like protocol violations.

Of cause it makes no sense if the system design is robust, but so defensive that it fails to meet performance goals. An example would be a protocol that never transmits a message unless it has received an explicit ACK for the previous message. This approach might avoid overloading the network, but would waste most of the potential bandwidth.

3.2 Data Flow in a Node

Nodes maintain connections with the peers that have been chosen with help of the peer selection process. Suppose node A is connected to node B. The Ouroboros protocol schedules a node N to generate a new block in a given time slot. Depending on the location of nodes A, B and N in the network topology and whether the new block arrives first at A or B, A can be either up-stream or down-stream of B. Therefore, node A runs an instance of the client side of the chain-sync mini protocol that talks with a server instance of chain-sync at node B and also a server instance of chain sync that talks with a client instance at B. The situation is similar for the other mini protocols (block fetch, transaction submission, etc). The set of mini protocols that runs over a connection is determined by the version of the network protocol, i.e. Node-to-Node, Node-to-Wallet and Node-to-Chain-Consumer connections use different sets of mini protocols (e.g. different protocol versions). The version is negotiated when a new connection is established using protocol which is described in Chapter 5.

Figure 3.1 illustrates parts of the data flow in a node. Circles represents a thread that runs one of the mini protocols (the mini protocols are explained in Chapter 4). There are two kinds of data flows: mini protocols communicate with mini protocols of other nodes by sending and receiving messages; and, within a



Chain sync protocol (client side)
Ledger state
Candidate chains (headers)
Candidate chains (blocks)
Chain and ledger validation
Block fetch protocol (server side)
Block download logic
Current chain (blocks)
Tentative chain (headers)
Block fetch protocol (client side)
Set of downloaded blocks (block bodies)
Chain sync protocol (server side)
Shared state (STM/TVar-based)
Thread running consensus layer code
Thread running networking code
Thread running mixture of networking and consensus layer code

Figure 3.1: Data flow inside a Node

node, they communicate by reading from- and writing to- a shared mutable state (represented by boxes in Figure 3.1). Software transactional memory (STM) is a mechanism for safe and lock-free concurrent access to mutable state and the reference implementation makes intensive use of this abstraction.

3.3 Real-time Constraints and Coordinated Universal Time

Ouroboros models the passage of physical time as an infinite sequence of time slots, i.e. contiguous, equal-length intervals of time, and assigns slot leaders (nodes that are eligible to create a new block) to those time slots. At the beginning of a time slot, the slot leader selects the block chain and transactions that are the basis for the new block, then it creates the new block and sends the new block to its peers. When the new block reaches the next block leader before the beginning of next time slot, the next block leader can extend the block chain upon this block (if the block did not arrive on time the next leader will create a new block anyway).

There are some trade-offs when choosing the slot time that is used for the protocol but basically the slot length should be long enough such that a new block has a good chance to reach the next slot leader in time. A chosen value for the slot length is 20 seconds. It is assumed that the clock skews between the local clocks of the nodes is small with respect to the slot length.

However, no matter how accurate the local clocks of the nodes are with respect to the time slots the effects of a possible clock skew must still be carefully considered. For example, when a node time-stamps incoming blocks with its local clock time, it may encounter blocks that are created in the future with respect to the local clock of the node. The node must then decide whether this is because of a clock skew or whether the node considers this as adversarial behaviour of an other node.

TODO :: get feedback from the researchers on this. Tentative policy: allow 200ms to 1s explain the problem in detail. A node cannot forward a block from the future. This is complicated !

Chapter 4

Mini Protocols

4.1 Mini Protocols and Protocol Families

A mini protocol is a well defined and modular building block of the network protocol. Structuring the protocol around mini protocols helps to manage the overall complexity of the design and adds useful flexibility. The design turns into a family of protocols that can be specialised to particular requirements by choosing a particular set of mini protocols.

The mini protocols in this section describe both the initiator and responder of a communication. The initiator is the dual of the responder and vice versa. (The terms client/server and consumer/producer are also used sometimes.) At any time a node will typically run many instances of mini protocols, including many instances of the same mini protocol. Each mini protocol instance of the node communicates with the dual instance of exactly one peer. All mini protocols that communicate with the same peer share a single communication channel (pipe or socket) and a multiplexer/de-multiplexer is used to multiplex the protocols over that channel. Section 5.1 describes the multiplexing layer.

The set of mini protocols that run on a connection between two participants of the system depends on the role of the participants, i.e. whether the node acts as a full node or just a block chain consumer, for example a wallet. Section 5.2 describes how a connection between two nodes that run a set of mini protocols is set up.

4.2 Protocols as State Machines

The reference implementation of several mini protocols uses a generic framework for state machines. This framework uses correct-by-construction techniques to guarantee several properties of the protocol and the implementation. In particular, it guarantees that there are no deadlocks, i.e., at any time, one side has agency (is expected to transmit the next message) and the other side is awaiting for the message (or both sides agree that the protocol has terminated). If either side receives a message that is not expected according to the protocol the communication is aborted.

For each mini protocol that is based on this underlying framework the description provides the following pieces of information:

- An informal description of the protocol.
- States of the state machine.
- The messages that are exchanged.
- A transition graph of the global view of the state machine.
- The client implementation of the protocol.
- The server implementation of the protocol.

State Machine Each mini protocol is described as a state machine. This document uses a simple diagram representations for state machines, and also includes corresponding transition tables. Descriptions of state machines in this section are directly derived from specifications of mini protocols using the state machine framework.

The state machine framework that is used to specify the protocol can be instantiated with different implementations that work at different levels of abstraction (for example implementations used for simulation, implementations that run over virtual connections and implementations that actually transmit messages over the real network).

States States are abstract: they are not a value of some variables in a node, but rather describe the state of the two-party communication as whole, e.g. that a client is responsible for sending a particular type of message and the server is awaiting on it. This, in particular, means that if the state machine is in a given state, both client and server are in this state. An additional piece of information that differentiates the roles of peers in a given state is agency, which describes which side is responsible for sending the next message.

In the state machine framework, abstract states of a state machine are modelled as promoted types, so they do not correspond to any particular value hold by one of the peers.

The document presents this abstract view of mini protocols and the state machines where the client and server are always in identical states, which also means that client and server simultaneously transit to new states. For this description network delays are not important.

An interpretation, which is closer to the real-world implementation but less concise, is that there are independent client and server states and that transitions on either side happen independently when a message is sent or received.

Messages Messages exchanged by peers form edges of a state machine diagram, in other words they are transitions between states. They are elements from the set

$$\{(label, data) \mid label \in Labels, data \in Data\}$$

Protocols use a small set of *Labels* typically $|Labels| \leq 10$. The state machine framework requires that messages can be serialised, transferred over the network and de-serialised by the receiver. The binary format for messages is described in Section A.

Agency A node has agency if it is expected to send the next message. In every state (except the Done-state) either the client or server has agency. In the Done-state the protocol has terminated and neither side is expected to send any more messages.

State machine diagrams States are drawn as circles in state machine diagrams. States with agency at the client are drawn in green, states with agency at the server in blue and the Done-state in black. By construction, the system is always in exactly one state, i.e. the client's state is always the same state as server's, and the colour indicates who is the agent. It is also important to understand that the arrows in the state transition diagram denote state transitions and not the direction of the message that is being transmitted. For the agent of the particular state it will mean send a message to the other peer, for a non-agent, listen for incoming message. This may be confusing because the arrows are labeled with the messages and many arrows go from a green state (client has the agency) to a blue state (server has the agency) or vice versa.

$$\begin{tikzcd} \boxed{A} \arrow[r, "\text{Message}"] & \boxed{B} \end{tikzcd}$$

A is green, i.e in state A the client has agency. Therefore the client sends a message to the server and both client and server transition to state B. As B is blue the agency also changes from client so server.

$$\begin{tikzcd} \boxed{C} \arrow[r, "\text{Message}"] & \boxed{D} \end{tikzcd}$$

C is blue, i.e in state C the server has agency. Therefore the server sends a message to the client and both client and server transition to state D. As D is also blue the agency remains at the server.

Client and server implementation The state machine describes which messages are sent and received and in which order. This is the external view of the protocol that every compatible implementation MUST follow. In addition to the external view of the protocol, this part of the specification describes how the client and server actually process the transmitted messages, i.e. how the client and server update their internal mutable state upon the exchange of messages.

Strictly speaking, the representation of the node-local mutable state and the updates to the node-local state are implementation details that are not part of the communication protocol between the nodes, and will depend on an application that is built on top of the network service (wallet, core node, explorer, etc.). The corresponding sections were added to clarify mode of operation of the mini protocols.

4.3 Overview of all implemented Mini Protocols

Ping Pong Protocol — Section 4.4
A simple ping-pong protocol for testing.
Haskell source: typed-protocols/src/Network/TypedProtocol/PingPong/Type.hs

Request Response Protocol — Section 4.5
A protocol similar to ping pong but which exchanges data.
Haskell source:

Single Phase Chain Synchronisation Protocol — Section 4.6

Haskell source: ouroboros-network/src/Ouroboros/Network/Protocol/ChainSync/Type.hs

Block Fetch Protocol — Section 4.7
The block fetching mechanism enables a node to download a range of blocks.
Haskell source: ouroboros-network/src/Ouroboros/Network/Protocol/BlockFetch/Type.hs

Local Transaction Submission Mini Protocol — Section 4.8
Transmitting Transactions for a wallet to a core node
Haskell source: src/Ouroboros/Network/Protocol/LocalTxSubmission/Type.hs

Handshake Mini Protocol — Section 4.9
This protocol is used for version negotiation.
Haskell source: src/Ouroboros/Network/Protocol/Handshake/Type.hs

4.4 Ping-Pong Protocol

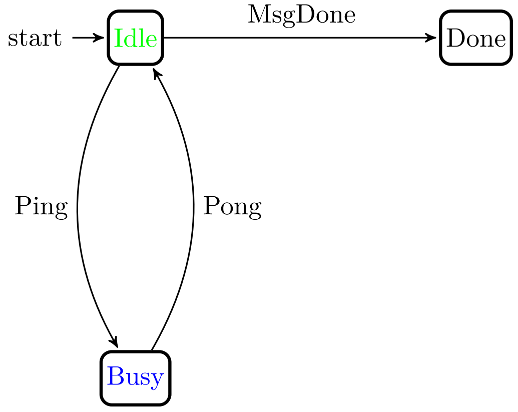
Haskell source: typed-protocols/src/Network/TypedProtocol/PingPong/Type.hs

4.4.1 Description

A client can use the Ping-Pong protocol to check that the server is responsive. The Ping-Pong protocol is very simple because the messages do not carry any data and because the Ping-Pong client and the Ping-Pong server do not access the internal state of the node. It uses the same framework for state machines as the other protocols, but because the protocol is so simple, the description of the protocol is also very simple and slightly different from descriptions of other protocols.

4.4.2 State Machine

$$\begin{tikzcd}
\text{start} \arrow[r] & \text{Idle} \arrow[rr, "\text{MsgDone}"] \arrow[d, bend right, "\text{Ping}"'] & & \text{Done} \\
& \text{Busy} \arrow[u, bend right, "\text{Pong}"'] & &
\end{tikzcd}$$



Agency	
Client has Agency	Idle
Server has Agency	Busy

The protocol uses the following messages. The messages of the Ping-Pong protocol do not carry any data.

Ping The client sends a Ping request to the server.

Pong The server replies to a Ping with a Pong.

MsgDone Terminate the protocol.

Transition table		
from state	message	to state
Idle	Ping	Busy
Busy	Pong	Idle
Idle	MsgDone	Done

4.5 Request Response Protocol

4.5.1 Description

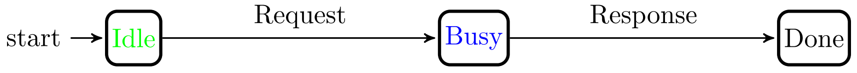
The request response protocol is polymorphic in the request and response data that is being transmitted. This means that there are different possible applications of this protocol and the application of the protocol determines the types of the requests and responses.

4.5.2 State machine

Haskell source: ouroboros-network/src/Ouroboros/Network/Protocol/ReqResp/Type.hs

Agency	
Client has Agency	Idle
Server has Agency	Busy

$$\begin{tikzcd}
\text{start} \arrow[r] & \text{Idle} \arrow[r, "\text{Request}"] & \text{Busy} \arrow[r, "\text{Response}"] & \text{Done}
\end{tikzcd}$$



Transition table			
from	message	parameters	to
Idle	Request	*request*	Busy
Busy	Response	*response*	Done

4.6 Single Phase Chain Synchronisation Protocol

Haskell source: ouroboros-network/src/Ouroboros/Network/Protocol/ChainSync/Type.hs

4.6.1 Description

The chain synchronisation protocol is used by the block chain consumer to replicate the producer's block chain locally. As it is polymorphic in blocks, it supports a family of Ouroboros protocols. The chain synchronisation protocol is used by clients that synchronise just headers (node-to-node), and also clients which synchronize actual block chains (e.g., node-to-wallet). (See Figure 3.1.)

A node communicates with several upstream and downstream nodes. A node runs an independent client instance and a independent server instance for every other node it communicates with.

4.6.2 State Machine

Agency	
Client has Agency	Idle
Server has Agency	CanAwait, MustReply, Intersect

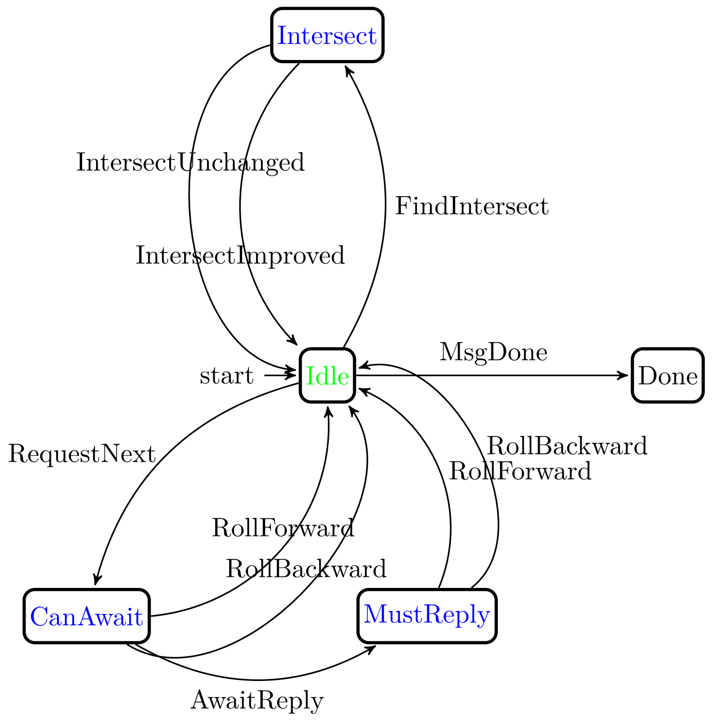


Figure 4.1: State machine of the chain sync protocol.

The protocol uses the following messages:

RequestNext Request the next update from the producer.

AwaitReply Acknowledge the request but require the consumer to wait for the next update. This means that the consumer is synced with the producer, and the producer is waiting for its own chain state to change.

RollForward $(header, point)$ Tell the consumer to extend their chain with the given *header*. The message also tells the consumer about the *head* point of the producer.

RollBackward $(point_{old}, point_{head})$ Tell the consumer to roll back to a given $point_{old}$ on their chain. The message also tells the consumer about the current head $point_{head}$ of the producer.

FindIntersect $[point_{head}]$ Ask the producer to try to find an improved intersection point between the consumer and producer's chains. The consumer sends a sequence $[point]$ and it is up to the producer to find the first intersection point on its chain and send it back to the consumer.

IntersectImproved $(point_{intersect}, point_{head})$ The reply to the consumer about an intersection found, but only if this is an improvement over the previously established intersection point. The consumer can decide whether to send more points. The message also tells the consumer about the head point of the producer.

IntersectUnchanged $(point_{head})$ The reply to the consumer that no intersection was found: none of the points the consumer supplied are on the producer chain. The message also tells the consumer about the head point of the producer.

MsgDone Terminate the protocol.

Transition table			
from state	message	parameters	to state
Idle	RequestNext		CanAwait
Idle	FindIntersect	$[point]$	Intersect
Idle	MsgDone		Done
CanAwait	AwaitReply		MustReply
CanAwait	RollForward	$header$, $point_{head}$	Idle
CanAwait	RollBackward	$header$, $point_{head}$	Idle
MustReply	RollForward	$header$, $point_{head}$	Idle
MustReply	RollBackward	$point_{old}$, $point_{head}$	Idle
Intersect	IntersectImproved	$point_{intersect}$, $point_{head}$	Idle
Intersect	IntersectUnchanged	$point_{head}$	Idle

4.6.3 Implementation of the Chain Producer

This section describes a state-full implementation of a chain producer that is suitable for a setting where the producer cannot trust the chain consumer. An important requirement in this setting is that a chain consumer must never be able to cause excessive resource use on the producer side. The presented implementation meets this requirement. It uses a constant amount of memory to store the state that the producer maintains per chain consumer. This protocol is only used to reproduce the producer chain locally by consumer. By running many instances of this protocol against different peers, a node can reproduce chains in the network and do chain selection which by design is not part of this protocol. An important note is that the consumer's chain described in this section is the chain reproduced by the client of chain-sync protocol, rather than the chain of that node.

We call the state which the producer maintains about the consumer the *read pointer*. The *read pointer* basically tracks what the producer knows about the head of the consumer's chain without storing it locally. It points to a block on the current chain of the chain producer. The *read pointers* are part of the shared state of the node (Figure 3.1) and *read pointer*s are concurrently updated by the thread that runs the chain-sync mini-protocol and the chain tracking logic of the node itself.

We first describe how the mini-protocol updates a *read pointer* and later address what happens in case of a fork. The chain producer assumes that a consumer, which has just connected, only knows the genesis block and initializes the *read pointer* of that consumer with a pointer to the genesis block on its chain.

Downloading a chain of blocks A typical situation is when the consumer follows the chain of the producer but is not yet at the head of the chain (this also covers a consumer booting from genesis). In this case, the protocol follows a simple, consumer-driven, request-response pattern. The consumer sends RequestNext messages to ask for the next block. If the *read pointer* is not yet at the head of the chain, the producer replies with a RollForward and advances the *read pointer* to the next block (optimistically assuming that the client will update its chain accordingly). The RollForward message contains the next block and also the head-point of the producer. The protocol follows this pattern until the *read pointer* reaches the end of its chain.

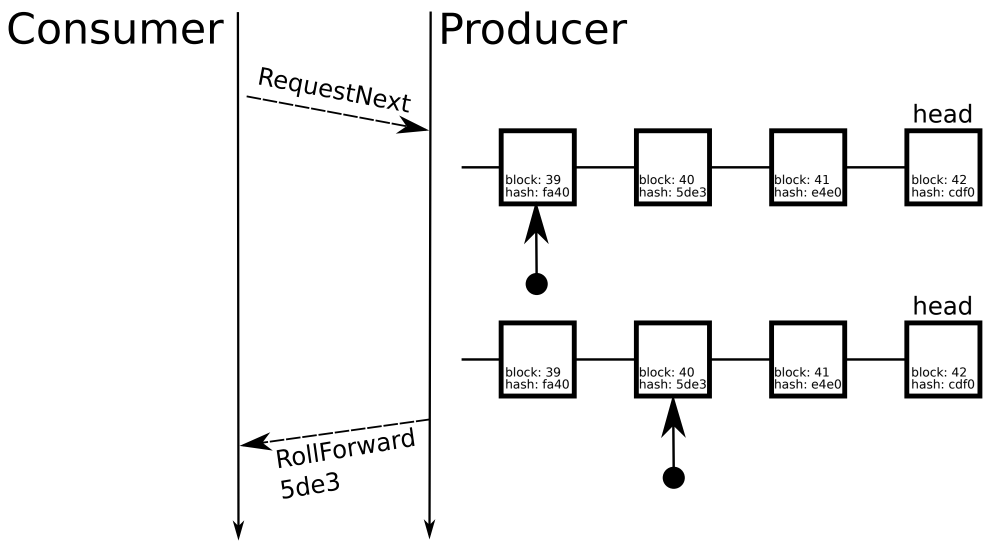


Figure 4.2: Consumer driven block download.

Producer driven updates If the *read pointer* points to the end of the chain and the producer receives a RequestNext the consumers chain is already up to date. The producer informs the consumer with an AwaitReply that no new data is available. After receiving a AwaitReply, the consumer just waits for a new message and the producer keeps agency. The AwaitReply switches from a consumer driven phase to a producer driven phase.

The producer waits until new data becomes available. When a new block is available, the producer will send a RollForward message and give agency back to the consumer. The producer can also get unblocked when its node switches to a new chain fork.

Producer switches to a new fork The node of the chain producer can switch to a new fork at any time, independent of the state machine. A chain switch can cause an update of the *read pointer*, which is part of the mutable state that is shared between the thread that runs the chain sync protocol and the thread that implements the chain following logic of the node. There are two cases:

1) If the *read pointer* points to a block that is on the common prefix of the new fork and the old fork, no update of the *read pointer* is needed.

2) If the *read pointer* points to a block that is no longer part of the chain that is followed by the node, the *read pointer* is set to the last block that is common between the new and the old chain. The node also sets a flag that signals the chain-sync thread to send a RollBackward instead of a RollForward. Finally the producer thread must unblock if it is in the MustReply state.

Figure 4.3 illustrates a fork switch that requires an update of the *read pointer* for one of the chain consumers, i.e. an example for case 2. Before the switch, the *read pointer* of the consumer points to block $0x660f$. The producer switches to a new chain with the head of the chain at block $0xcdf0$. The node must update the *read pointer* to block $0xfa40$ and the next message to the consumer will be a RollBackward.

Note, that a node typically communicates with several consumers. For each consumer it runs an independent version of the chain-sync-protocol state machine in an independent thread and with its own *read pointer*. Each of those *read pointer*s has to be updated independently and for each consumer either case 1) or case 2) can apply.

Consumer starts with an arbitrary fork Typically, the consumer already knows some fork of the block chain when it starts to track the producer. The protocol provides an efficient method to search for the longest

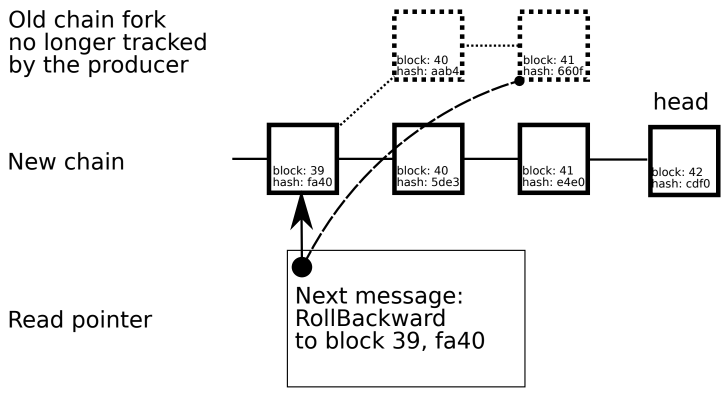


Figure 4.3: *read pointer* update for a fork switch in case of a rollback.

common prefix (here called intersection) between the fork of the producer and the fork that is known to the consumer.

To do so, the consumer sends a FindIntersect message with a list of chain points which belong to its node chain. If the producer does not know any of the points it replies with IntersectUnchanged. Otherwise it replies with IntersectImproved and the best (i.e. the newest) of the points that it knows and also updates the *read pointer* accordingly. For efficiency, the consumer should use a binary search scheme to search for the longest common prefix.

It is advised that the consumer always starts with FindIntersect in a fresh connection and it is free to use FindIntersect at any time later as seems beneficial. If the consumer does not know anything about the producer's chain, it can start the search with the following list of points: $[point(b), point(b-1), point(b-2), point(b-4), point(p-8), \ldots]$ where $point(b-i)$ is the point of the ith predecessor of block b and b is the head of the consumer fork. (Note, that the maximum depth of a fork in Ouroboros is bounded).

4.6.4 Implementation of the Chain Consumer

In principle, the chain consumer has to guard against a malicious chain producer as much as the other way around. However, two aspects of the protocol play in favour of the consumer here.

- The protocol is basically consumer driven, i.e. the producer has no way to send unsolicited data to the consumer (within the protocol).
- The consumer can verify the response data itself.

Here are some cases to consider:

FindIntersect Phase The consumer and the producer play a number guessing game, so the consumer can easily detect inconsistent behaviour.

The producer replies with a RollForward The consumer can verify the block itself with the help of the ledger layer. (The consumer may need to download the block first, if the protocol only sends block headers.)

The producer replies with a RollBackward The consumer tracks several producers, so if the producer sends false RollBackward messages the consumer's node will, at some point, just switch to a longer chain fork.

The Producer is just passive/slow The consumer's node will switch to a longer chain coming from another producer via another instance of chain-sync protocol.

This should be explained in detail

4.7 Block Fetch Protocol

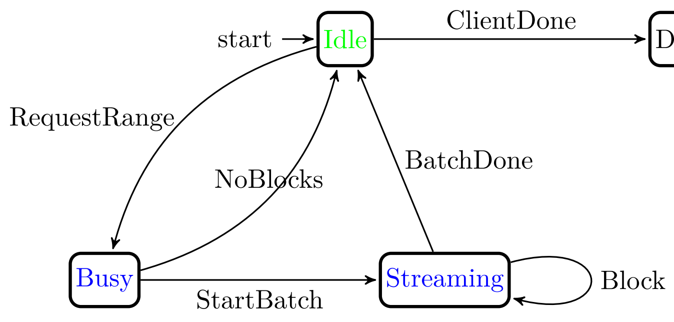
Haskell source: ouroboros-network/src/Ouroboros/Network/Protocol/BlockFetch/Type.hs

4.7.1 Description

The block fetching mechanism enables a node to download a range of blocks.

4.7.2 State machine

Agency	
Client has Agency	Idle
Server has Agency	Busy, Streaming



RequestRange ***(range)*** The client requests a ***range*** of blocks from the server.

NoBlocks The server tells the client that it does not have blocks.

StartBatch The server starts block streaming.

Block ***(body)*** Stream a single block's body.

BatchDone The server ends block streaming.

ClientDone The client terminates the protocol.

Transition table:

from state	message	parameters	to state
Idle	ClientDone		Done
Idle	RequestRange	*range*	Busy
Busy	NoBlocks		Idle
Busy	StartBatch		Streaming
Streaming	Block	*body*	Streaming
Streaming	BatchDone		Idle

4.8 Local Transaction Submission Mini Protocol

Haskell source: src/Ouroboros/Network/Protocol/LocalTxSubmission/Type.hs

4.8.1 Description

The local transaction submission mini protocol is used by local clients, for example wallets or CLI tools, to submit transactions to a local node. The protocol is not used to forward transactions from one core node to an other.

The protocol follows a simple request-response pattern:

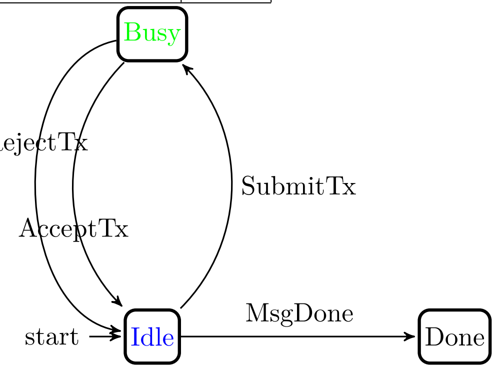
1. The client sends a request with a single transaction.

2. The Server either accepts the transaction (returning a confirmation) or rejects it (returning the reason).

Note, that the local transaction submission protocol is a push bases protocol where the client creates a workload for the server. This is acceptable because is protocol is only for use between a node and local client.

4.8.2 State machine

Agency	
Client has Agency	Idle
Server has Agency	Busy



Messages of the protocol:

SubmitTx (t) The client submits a transaction.

AcceptTx The server accepts the transaction.

RejectTx $(reason)$ The server rejects the transactions and replies with the *reason*.

MsgDone The client terminates the mini protocol.

4.9 Handshake Mini Protocol

Haskell source: src/Ouroboros/Network/Protocol/Handshake/Type.hs

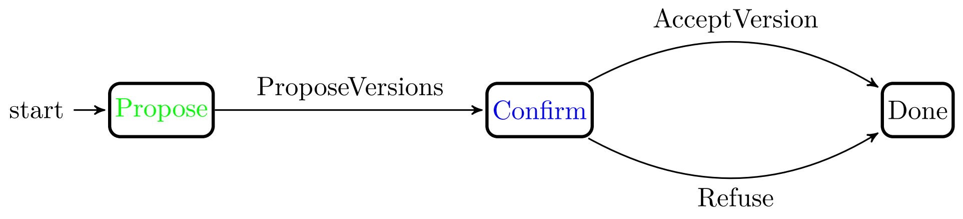
4.9.1 Description

The handshake mini protocol is used to negotiate the protocol version and the protocol parameters that are used by the client and the server. It is run exactly once when a new connection is initialized and consists of a single request from the client and a single reply from the server. Section 5.2 explains the live cycle of a connection and the role of the handshake mini protocol in more detail.

The handshake mini protocol is a generic protocol that can negotiate any kind protocol parameters. It only assumes that protocol parameters can be encoded to, and decoded from, CBOR terms. A node, that runs the handshake protocol, must instantiate it with the set of supported protocol versions and callback functions for handling the protocol parameters. These callback functions are specific for the supported protocol versions.

4.9.2 State machine

Agency	
Client has Agency	Propose
Server has Agency	Confirm



Messages of the protocol:

ProposeVersions **(*versionTable*)** The client proposes a number of possible versions and protocol parameters.

AcceptVersion **(*versionNumber, extraParameters*)** The server accepts *versionNumber* and returns possible extra protocol parameters.

Refuse **(*reason*)** The server refuses the proposed versions.

Transition table			
from	message/event	parameters	to
Propose	ProposeVersions	*versionTable*	Confirm
Confirm	AcceptVersion	(*versionNumber, extraParameters*)	Done
Confirm	Refuse	*reason*	Done

4.9.3 Client and Server Implementation

Section A contains the CDDL-specification of the binary format of the handshake messages. The version table is encoded as a CBOR table with the version number as key and the protocol parameters as value. The handshake protocol requires that the version numbers (i.e. the keys) in the version table are unique and appear in ascending order. (Note, that CDDL is not expressive enough to precisely specify that requirement on the keys of the CBOR table. Therefor the CDDL-specification uses a table with keys from 1 to 4 as an example.)

In a run of the handshake mini protocol the peers exchange only two messages: The client requests to connect with a ProposeVersions message that contains information about all protocol versions it wants to support. The server replies either with an AcceptVersion message containing the negotiated version number and extra parameters or a Refuse message. The Refuse message contains one of three alternative refuse reasons: VersionMismatch, HandshakeDecodeError or just Refused.

When a server receives a ProposeVersions message it uses the following algorithm to compute the response:

1. Compute the intersection of the set of protocol version numbers that the server support and the version numbers requested by the client.
2. If the intersection is empty: Reply with Refuse(VersionMismatch) and the list of protocol numbers the server supports.
3. Otherwise: Select the protocol with the highest version number in the intersection.
4. Run the protocol specific decoder on the CBOR term that contains the protocol parameters.
5. If the decoder fails: Reply with Refuse(HandshakeDecodeError), the selected version number and an error message.
6. Otherwise: Test the proposed protocol parameters of the selected protocol version
7. If the test refuses the parameters: Reply with Refuse(Refused), the selected version number and an error message.
8. Otherwise: Encode the extra parameters and reply with AcceptVersion, the selected version number and the extra parameters.

Note, that in step 4), 6) and 8) the handshake protocol uses the callback functions that are specific for set of protocols that the server supports. The handshake protocol is designed, such that a server can allways handle requests for protocol versions that it does not support. The server simply ignores the CBOR terms that represent the protocol parameters of unsupported version.

see in the code if this is still true: The handshake mini protocol runs before the MUX/DEMUX itself is initialized. Each message is transmitted within a single MUX segment, i.e. with a proper segment header, but, as the MUX/DEMUX is not yet running the messages must not be split into multiple segments.

4.10 Pipelining of Mini Protocols

Protocol pipelining is a technique that improves the performance of some protocols. The underlying idea is that a client, which wants to perform several requests, just transmits those requests in sequence without blocking and waiting for the reply from the server. In the reference implementation, pipelining is used by the clients of all mini protocol except Chain-Sync. Those mini protocols follow a request-response pattern that is amenable to pipelining such that pipelining becomes a feature of the client implementation that does not require any modifications of the server implementation.

As an example, let's consider the Block-Fetch mini protocol. When a client follows the protocol and sends a sequence of RequestRange messages to the server the data stream from the client to the server will only consist of RequestRange messages (and a final ClientDone message) and no other message types. The server can simply follow the state machine of the protocol and process the messages in turn, regardless whether the client uses pipelining or not. The MUX/DEMUX layer (Section 5.1) guarantees that messages of the same mini protocol are delivered in transmission order, and therefore the client can determine which response belongs to which request.

The MUX/DEMUX layer also provides a fixed size buffer between the egress of DEMUX and the ingress of mini protocol thread. The size of this buffer is a protocol parameter that determines how many messages a client can send before waiting for a reply from the server (see Section 5.1.3). The protocol requires that a client must never cause a overrun of these buffers on a server node. If a message arrives at the server that would cause the buffer to overrun, the server treats this case as a protocol violation of the peer (and closes the connection to the peer).

4.11 DeltaQ Mini Protocol

WIP : Explain DeltaQ measurement back pressure and how we deal with slow connection. See Section ??. The DeltaQ mini protocol does not transmit is own messages. Instead it relies on the time stamps that the multiplexing layer (Section ??) adds to the messages of other mini protocols.

Chapter 5

Connection Management

5.1 The Multiplexing Layer

Multiplexing is used to run several mini protocols in parallel over a single channel (for example a single TCP connection). Figure 5.1 shows an example of two nodes, each running three mini protocols and a multiplexer/de-multiplexer. All the data that is transmitted between the nodes passes through the MUX/DEMUX of the nodes. There is a fixed pairing of the mini protocol instances, i.e. each mini protocol instance only communicates with its dual instance.

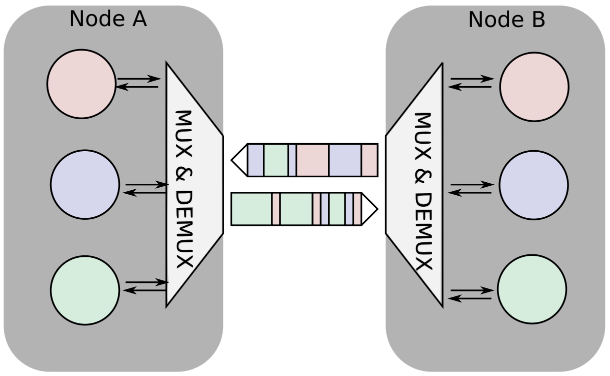


Figure 5.1: Data flow though the multiplexer and de-multiplexer

The implementation of the mini protocol also handles the serialisation and de-serialisation of its messages. The mini protocols write chunks of bytes to the MUX and read chunks of bytes from the DEMUX. The MUX reads the data from the mini protocols, splits the data into segments, adds a segment header and transmits the segments to the DEMUX of its peer. The DEMUX uses the segment's headers to reassemble the byte streams for the mini protocols on its side. The multiplexing protocol itself is completely agnostic to the structure of the multiplexed data.

5.1.1 Wire Format

Haskell source: ouroboros-network/src/Ouroboros/Network/Mux/Egress.hs

Table 5.1 shows the layout of the data segments of the multiplexing protocol (big-endian bit order). The segment header contains the following data:

Transmission Time The transmission time is a time stamp based the wall clock of the peer with a resolution of one microsecond.

Mini Protocol ID The unique ID of the mini protocol as in Table 5.2.

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Transmission Time																															
M	Mini Protocol ID															Payload-length n															
Payload of n Bytes																															

Table 5.1: Multiplexing-segments

Payload Length The payload length is the size of the segment payload in Bytes. The maximum payload length that is supported by the multiplexing wire format is $2^{16} - 1$. Note, that an instance of the protocol can choose a smaller limit for the size of segments it transmits.

Mode The single bit M (the mode) is used to distinct the dual instances of a mini protocol. The mode is set to 0 in segments from the initiator, i.e. the side that initially has agency and 1 in segments from the responder.

5.1.2 Fairness and Flow-Control in the Multiplexer

The Shelley network protocol requires that the multiplexer uses a fair scheduling of the mini protocols.

The reference Haskell implementation of multiplexer uses a round-robin-schedule of the mini protocols to choose the next data segment to transmit. If a mini protocol does not have new data available when it is scheduled, it is skipped. A mini protocol can transmit at most one segment of data every time it is scheduled and it will only be rescheduled immediately if no other mini protocol has data available. Each mini protocol is implemented as a separate Haskell thread. These threads can signal the multiplexer at any time that they have new data available.

From the point of view of the mini protocols, there is a one-message buffer between the egress of the mini protocol and the ingress of the multiplexer. The mini protocol will block when it sends a message and the buffer is full.

A concrete implementation of a multiplexer may use a variety of data structures and heuristics to yield the overall best efficiency. For example, although the multiplexing protocol itself is agnostic to the underlying structure of the data, the multiplexer may try to avoid spliting small mini protocol messages into two segments. The multiplexer may also try to merge multiple messages from one mini protocol into a single segment. Note that, the messages within a segment must all belong to the same mini protocol.

5.1.3 Flow-control and Buffering in the Demultiplexer

The demultiplexer eagerly reads data from its ingress. There is a fixed size buffer between the egress of the demultiplexer and the ingress of the mini protocols. Each mini protocol implements its own mechanism for flow control which guaranties that this buffer never overflows (See Section 4.10.). If the demultiplexer detects an overflow of the buffer, it means that the peer violated the protocol and the MUX/DEMUX layer shuts down the connection to the peer.

5.2 Setup, Shutdown and Management of Connections

In addition to the exchange of blocks and transactions, as required by Ouroboros, the network layer also handles several administrative tasks. This section describes the parts of the protocol that deal with setting up, shutting down and managing a connection between two peers. In this section, we use the term connection or bearer for the multiplexing-layer object that manages the mini protocol threads, the buffers and the OS-level connection (for example TCP socket) that deals with one peer of the node.

The multiplexing layer (Section 5.1) is the central crossing between the mini protocols and the network channel. Therefore, the reference implementation takes the approach of implementing the functions for connection management in the same part of the source code that also implements the multiplexing layer itself.

This section describes the protocol and sketches a possible implementation. Roughly the implementation performs the following tasks:

- Open a socket/ acquire resources from the OS.
- Negotiate the protocol version with the handshake mini protocol (Section ??.
- Spawn the threads that run the mini protocols.
- Measure transmission times and amount of in-flight data.
- Catch exceptions that are thrown by the mini protocols.
- Shutdown the connection in case of an error.
- Handle a shutdown request from the peer.
- Shutdown the threads that run the mini protocols.
- Close Socket/ free resources.

5.3 Life Cycle of a Connection

$$\begin{tikzcd}
\text{start} \arrow[r] & \boxed{\text{Larval}} \arrow[d, "\text{OS connect}"] \\
& \boxed{\text{Connected}} \arrow[d, "\text{agree protocol versions, start mini protocols}"] \\
& \boxed{\text{Mature}} \arrow[d, "\text{termination of any mini protocol}"] \\
& \boxed{\text{Dying}} \arrow[d, "\text{transmission of messages that are already buffered,OS disconnect}"] \\
& \boxed{\text{Dead}}
\end{tikzcd}$$

A connection passes through several stages during its life cycle.

Larval The connection exists but nothing has been initialised yet.

Connected The OS-level primitives (sockets or pipes) are connected.

Mature The mini protocols are running.

Dying One of the mini protocols has terminated.

Dead The connection has been terminated.

Haskell source: ouroboros-network/src/Ouroboros/Network/Mux/Types.hs
Haskell source: ouroboros-network/src/Ouroboros/Network/NodeToNode.hs
Haskell source: ouroboros-network/src/Ouroboros/Network/NodeToClient.hs

ID	Mini Protocol	NtN	NtC
0	MUX-Control	Yes	Yes
1	DeltaQ	Yes	Yes
2	Chain-Sync instantiated to headers	Yes	No
3	Block-Fetch	Yes	No
4	Transaction-Submission	No	Yes
5	Chain-Sync instantiated to blocks	No	Yes

Table 5.2: Mini Protocol IDs

5.3.1 Default Mini Protocol Sets for Node-to-Node and Node-to-Client

Table 5.2 show which mini protocols are enabled for node-to-node and node-to-client communication. Mux-Control and DeltaQ are enabled for all connections. The communication between two full nodes (NtN) is fully symmetric. Both nodes run initiator and responder instances of the Chain-Sync, the Block-Fetch and the Transaction-Submission protocol. Node-to-Client (NtC) is a connection between a full node and a client that does not take part in Ouroboros protocol itself and only consumes data, for example a wallet or a block chain explorer. In a NtC setup, the node only runs the producer side of the Chain-Sync protocol and the client only the consumer side. The Chain-Sync protocol is polymorphic in the type of blocks that are transmitted. NtN uses a Chain-Sync instance which only transmits block headers, while the NtC instance transmits full blocks. The two variants of Chain-Sync use different protocol IDs.

5.3.2 Time Measurement

5.3.3 Error Handling

When a mini protocol thread detects that a peer violates the mini protocol it throws an exception. The MUX-layer catches the exceptions from the mini protocol threads and shuts down the connection.

The correctness of distributed and concurrent systems has been studied intensively for decades.

Safety properties Prove that a bad thing will never happen.

- Coins cannot be stolen
- Preservation of Money
- Nodes will not run out of Memory
- (Property: Current state is valid) will always hold / never fail

Liveness properties Prove that a desired event will happen.

- Message will be delivered
- Consensus will be reached
- Transaction will be confirmed
- Fairness : the desired event will happen in time. One does not have to wait forever
- Starvation
- Deadlocks

Temporal logic Tailor made logic for analysing concurrent systems.

- Argue about the temporal order of events in transition systems.
- Express safety properties.
- Express liveness properties.
- Express Fairness.
- Prove with model checkers.

- Refinement properties.
- CTL computation tree logic (safety)
- LTL linear time logic (fairness)

Time How does a concurrent system deal with time ?

- Physical clocks / Wall clock time
- Logical clocks / Vector clocks / order of events
- Order of events : Before , Concurrent, After
- Hybrid approaches, Ouroboros, slot-times

Session Types Model protocols and transition systems in a type system.

Pi-calculus

Process algebras

5.4 Design Discussion

Why distinguish between node to node and node-to-consumer IPC

We use two different sets of protocols for these two use cases.

node-to-node IPC between nodes that are engaged in the high level Ouroboros blockchain consensus protocol.

node-to-consumer IPC between a Cardano node and a 'chain consumer' component such as a wallet, explorer or other custom application.

This section describes the differences between those two variants of IPC and why they use different protocols.

The node-to-node protocol is conducted in a P2P environment with very limited trust between peers. The node-to-node protocol utilises store-and-forward over selected bearers which form the underlying connectivity graph. A concern in this setting is asymmetric resource consumption attacks. Ease of implementation is desirable, but is subordinate to the other hard constraints.

A node-to-consumer protocol is intended to support blockchain applications like wallets and explorers, or Cardano-specific caches or proxies. The setting here is that a consumer trusts a node (a 'chain producer') and just wants to catch up and keep up with the blockchain of that producer. It is assumed that a consumer only consumes from one producer (or one of a related set of producers), so unlike in the node-to-node protocol there is no need to choose between different available chains. The producer may still not fully trust the consumer and does not want to be subject to asymmetric resource consumption attacks. In this use case, because of the wider range of applications that wish to consume the blockchain, having some options that are easy to implement is more important, even if this involves a trade-off with performance. That said, there are also use cases where tight integration is possible and making the most efficient use of resources is more desirable.

There are a number of applications that simply want to consume the blockchain, but are able to rely on an upstream trusted or semi-trusted Cardano consensus node. These applications do not need to engage in the full consensus protocol, and may be happy to delegate the necessary chain validation.

Examples include 3rd party applications that want to observe the blockchain, examples being business processes triggered by transactions or analytics. It may also include certain kinds of light client that wish to follow the blockchain but not do full validation.

Once one considers a node-to-consumer protocol as a first class citizen then it opens up opportunities for different system architecture choices. The architecture of the original Cardano Mainnet release was entirely homogeneous: every node behaved the same, each trusted nothing but itself and paid the full networking and processing cost of engaging in the consensus protocol. In particular everything was integrated into a single process: the consensus algorithm itself, serving data to other peers and components such as the wallet or explorer. If we were to have a robust and efficient node-to-consumer protocol then we can make many other choices.

With an efficient local IPC protocol we can have applications like wallets and explorers as separate processes. Even for tightly integrated components it can make sense to run them in separate OS processes and using associated OS management tools. Not only are the timing constraints for a consensus node much easier to manage when it does not have to share CPU resources with chain consumers, but it enables sophisticated end-users to use operating system features to have finer control over resource consumption. There have been cases in production where a highly loaded wallet component takes more than its allowed allocation of CPU resources and causes the local node to miss its deadlines. By giving a consensus node a dedicated CPU core it becomes easier to provide the necessary hard real time guarantees. In addition, scaling on multi-core machines is significantly easier with multiple OS processes than with a multi-threaded OS process with a shared-heap. This could allow larger capacity Cardano relay deployments where there are multiple network facing proxy processes that all get their chain from a single local consensus node.

With an efficient network IPC protocol we can do similar things but extend it across multiple machines. This permits: large organisations to achieve better alignment with their security policies; clusters of relays operated by a single organisation to use the more efficient (less resource costly) node-to-consumer protocol instead of the node-to-node protocol; and wallet or explorer-like applications that need to scale out, and are able to make use of a trusted node.

Appendix A

CDDL Specification of the Protocol Messages

Haskell source: ouroboros-network/src/Ouroboros/Network/Protocol/PingPong/Codec.hs This Sections contains the CDDLH. Birkholz and Bormann (2018) specification of the binary serialisation format of the network protocol messages.

To keep this Section in close sync with the actual Haskell implementation the names of the Haskell identifiers have been reused for the corresponding CBOR types (with the first letter converted to lower case). Note, that, for readability, the previous Sections used simplified message identifiers, for example RequestNext instead of msgRequestNext, etc. Both identifiers refer to the same message format.

All transmitted messages satisfy the shown CDDL specification. However, CDDL, by design, also permits variants in the encoding that are not valid in the protocol. In particular, the notation [...] in CDDL can be used for both fixed-length and variable-length CBOR-list, while only one of the two encodings is valid in the protocol. We add comments in specification to make clear which encoding must be used.

Note that, in the case of the request-response mini protocol (Section refrequest-response-protocol) there in only ever one possible kind of message in each state. This means that there is no need to tag messages at all and the protocol can directly transmit the plain request and response data.

TODO: test that haskell(message) => cddl(message)

```
; The only purpose of allMessages is to have an entry point for testing all parts of the specificatio
; ! allMessages is NOT a valid wire format !

allMessages =
        [0,chainSyncMessage]
     /  [1,reqRespMessage]
     /  [2,pingPongMessage]
     /  [3,blockFetchMessage]
     /  [4,txSubmissionMessage]
     /  [5, handshakeMessage]

; Chain Sync Protocol
; reference implementation of the codec in
; ouroboros-network/src/Ouroboros/Network/Protocol/ChainSync/Codec/Cbor.hs

chainSyncMessage
     = msgRequestNext
     / msgAwaitReply
     / msgRollForward
     / msgRollBackward
     / msgFindIntersect
     / msgIntersectImproved
     / msgIntersectUnchanged
     / chainSyncMsgDone

; these records use fixed length list encodings TODO: lookup CDDL syntax for that
```

```
msgRequestNext          = [0]
msgAwaitReply           = [1]
msgRollForward          = [2, header, point]
msgRollBackward         = [3, header, point]
msgFindIntersect        = [4, points]
msgIntersectImproved    = [5, point, point]
msgIntersectUnchanged   = [6, point]
chainSyncMsgDone    = [7]

points = [point]

header = bytes .cbor any ; todo include definitions for header and point
point = bytes .cbor any

; Request Response Protocol
; reference implementation of the codec in
; ouroboros-network/src/Ouroboros/Network/Protocol/ReqResp/Codec.hs

reqRespMessage
    = msgRequest
    / msgResponse
    / reqRespMsgDone

msgRequest  = [0,  requestData]
msgResponse = [1, responseData]
reqRespMsgDone = [2]

requestData  = bytes .cbor any
responseData = bytes .cbor any

; Ping-Pong Protocol
; reference implementation of the codec in
; ouroboros-network/src/Ouroboros/Network/Protocol/PingPong/Codec.hs
pingPongMessage
    = msgPing
    / msgPong
    / pingPongMsgDone
msgPing = 0
msgPong = 1
pingPongMsgDone = 2

; BlockFetch Protocol
; reference implementation of the codec in
; ouroboros-network/src/Ouroboros/Network/Protocol/BlockFetch/Codec.hs

blockFetchMessage
     = msgRequestRange
     / msgClientDone
     / msgStartBatch
     / msgNoBlocks
     / msgBlock
     / msgBatchDone

msgRequestRange = [0, bfPoint, bfPoint]
msgClientDone   = [1]
msgStartBatch   = [2]
msgNoBlocks     = [3]
msgBlock        = [4, bfBody]
msgBatchDone    = [5]
```

```
bfPoint         = origin / [slotNo, dummyBlockHash]
origin          = []
slotNo          = uint ; word64
dummyBlockHash  = null
bfBody          = bytes .cbor any


; Transaction Submission Protocol
; reference implementation of the codec in
; ouroboros-network/src/Ouroboros/Network/Protocol/TxSubmission/Codec.hs

txSubmissionMessage
    = msgGetHashes
    / msgSendHashes
    / msgGetTx
    / msgTx
    / txSubmissionMsgDone

msgGetHashes  = [0,int]
msgSendHashes = [1,[txHash]]
msgGetTx      = [2,txHash]
msgTx         = [3,transaction]
txSubmissionMsgDone = [4]

txHash        = bytes .cbor any
transaction   = bytes .cbor any

; ToDo
; MuxControl Protocol
; reference implementation of the codec in
; ouroboros-network/src/Ouroboros/Network/Mux/Control.hs
; muxControlMessage = msgInitReq / msgInitRsp / msgInitFail
; msgInitReq  = [0]
; msgInitRsp  = [1]
; msgInitFail = [2]
; Handshake Protocol
; reference implementation of the codec in
; ouroboros-network/src/Ouroboros/Network/Protocol/Handshake/Codec.hs

versionNumber = uint .size 4
handshakeMessage =
       msgProposeVersions
     / msgAcceptVersion
     / msgRefuse

msgProposeVersions = [0 , versionTable]

; CDDL is not expressive enough to describe the all possible values of proposeVersions.
; proposeVersions is a tables that maps version numbers to version parameters.
; The codec requires that the keys are unique and in ascending order.
; This specification only enumerates version numbers from 1..4.

params = any
extraParams = any
versionTable =
    { ? 1 => params
    , ? 2 => params
    , ? 3 => params
    , ? 4 => params
    }

msgAcceptVersion    = [1, versionNumber, extraParams]
msgRefuse           = [2, refuseReason  ]

refuseReason
    = refuseReasonVersionMismatch
    / refuseReasonHandshakeDecodeError
    / refuseReasonRefused

refuseReasonVersionMismatch      = [0  , [ versionNumber] ]
refuseReasonHandshakeDecodeError = [1  , versionNumber, tstr]
refuseReasonRefused              = [2  , versionNumber, tstr]
```

Bibliography

Brünjes, L., Kiayias, A., Koutsoupias, E., and Stouka, A. (2018). Reward sharing schemes for stake pools. CoRR, abs/1807.11218.

H. Birkholz, C. V. and Bormann, C. (2018). Concise data definition language (cddl): a notational convention to express cbor data structures. https://tools.ietf.org/id/draft-ietf-cbor-cddl-00.txt.