Ouroboros Network Specification

Duncan Coutts[1] Neil Davies[2] Marc Fontaine[3] Karl Knutsson[4]
Armando Santos[5] Marcin Szamotulski[6] Alex Vieth[7]

9th January 2026

[1]duncan@well-typed.com, duncan.coutts@iohk.io
[2]neil.davies@pnsol.com, neil.davies@iohk.io
[3]marc.fontaine@iohk.io
[4]karl.knutsson-ext@cardanofoundation.org
[5]armando@well-typed.com
[6]marcin.szamotulski@iohk.io
[7]alex@well-typed.com

Abstract

This document provides technical specifications for the implementation of the `ouroboros-network` component of `cardano-node`. It provides specifications of all mini-protocols, multiplexing, and low-level wire encoding. It provides necessary information about both node-to-node and node-to-client protocols.

The primary audience for this document is engineers wishing to build clients interacting with a node via node-to-client or node-to-node protocols or independent implementations of a node. Although the original implementation of `ouroboros-network` is done `Haskell`, this specification is made language agnostic. We may provide some implementation details which are `Haskell` specific.

Contents

Chapter 1

System Architecture

1.1 Protocols and node design

There are two protocols which support different sets of mini-protocols:

- node-to-node protocol for communication between different nodes usually run by different entities across the globe. It consists of chain-sync, block-fetch, tx-submission and keep-alive mini-protocols.
- node-to-client protocol for intra-process communication, which allows to build applications that need access to the blockchain, ledger, e.g. a wallet, an explorer, etc. It consists of chain-sync, local-tx-submission and local-state-query mini-protocols.

Chain-sync mini-protocol (the node-to-node version) is used to replicate a remote chain of headers; block-fetch mini-protocol to download blocks and tx-submission to disseminate transactions across the network.

Figure 1.1 illustrates the design of a node. Circles represents threads that run one of the mini-protocols. Each mini-protocols communicate with a remote node over the network. Threads communicate by means of shared mutable variables, which are represented by boxes in Figure 1.1. We heavily use Software transactional memory (STM), which is a mechanism for safe and lock-free concurrent access to mutable state (see Harris and Peyton Jones (2006)).

The ouroboros-network supports multiplexing mini-protocols, which allows us to run the node-to-node or the node-to-client protocol on a single bearer, e.g. a TCP connection; other bearers are also supported. This means that chain-sync, block-fetch and tx-submission mini-protocols will share a single TCP connection. The multiplexer and its framing are described in Chapter 2.

1.2 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 in the rate at which transactions are submitted must not cause a decrease in the blockchain 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 into 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 from transient conditions. In any condition, a node must not exceed its memory limits, that is there must be defined limits, breaches of which are treated like protocol violations.

Of course, 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

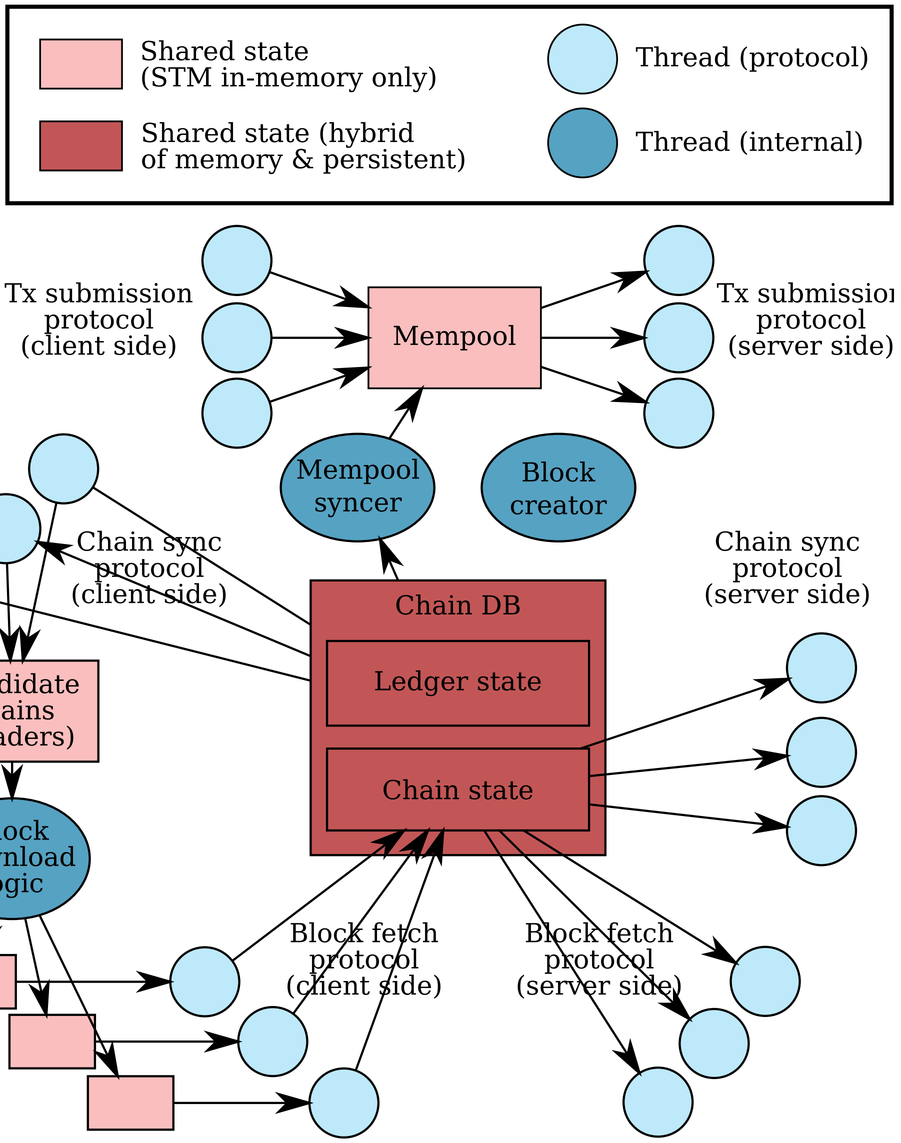

Shared state (STM in-memory only)
Shared state (hybrid of memory & persistent)
Thread (protocol)
Thread (internal)
Tx submission protocol (client side)
Mempool
Tx submission protocol (server side)
Mempool syncer
Block creator
Chain sync protocol (client side)
Chain sync protocol (server side)
Chain DB
Ledger state
Chain state
Candidate chains (headers)
Block download logic
Block fetch protocol (client side)
Block fetch protocol (server side)

Figure 1.1: Cardano Node

message. This approach might avoid overloading the network but would waste most of the potential bandwidth. To avoid such performance problems our implementation relies upon protocol pipelining.

1.3 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 blockchain 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 the next time slot, the next block leader can extend the blockchain 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 of reaching the next slot leader in time. 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 another node.

1.4 Nodes behind NAT firewalls

Many end-user systems, as well as enterprise systems, operate behind firewalls, in particular NATs. Such environments require specific support from the software to operate a P2P distributed system on equal terms. One of the requirements for `cardano-node` was for nodes operating behind a NAT to be able to contribute to the network. This includes full-node wallets (e.g. Daedalus) or enterprise deployments of relays/BPs to supply blocks, instead of only submitting transactions and consuming the blockchain. This requires a way for the outside world to traverse through the NAT. `ouroboros-network` uses a hole-punching method for this purpose.

1.4.1 Implementation details

`ouroboros-network` allows for promoting an inbound (or outbound) connection to be used as an outbound (inbound) connection. This way, an outgoing connection from behind a NAT can be reused as an inbound one, allowing access to blocks produced (or relayed) behind the firewall. Collected data shows that such nodes (primarily coming from Daedalus users) meaningfully contribute to the network.

`ouroboros-network` goes one step forward and binds all outbound connections to the same port as inbound connections (with some exceptions, controlled by a topology file). This allows us to reduce the number of open file-descriptors used by the node (at most by the number of all established outbound connections, e.g. 70 by default, and much more for some specific configurations).

Chapter 2

Multiplexing mini-protocols

The role of the multiplexing layer is to take an established underlying point-to-point bearer (e.g. a TCP connection, a UNIX socket or similar) and offer a multiplexed, sequenced-record delivery service for a fixed collection of services (fixed after negotiation).

Carrying all the related services between two peers has several advantages over multiple TCP connections: It reduces overheads (in the kernel and network path), improves congestion window management by minimising network capacity over-allocation during periods of congestion while, at the same time, gaining more dynamic responsiveness to changing end-to-end transport conditions.

Finally, it helps with performance exception detection and mitigation by creating a logical unit-of-failure - if a single component should 'fail' all the associated services on that peer can be failed together.

2.1 The Multiplexing Layer

Multiplexing is used to run several mini-protocols in parallel over a bidirectional bearer (for example, a TCP connection). Figure 2.1 illustrates multiplexing of three mini-protocols over a single duplex bearer. The multiplexer guarantees a fixed pairing of mini-protocol instances, each mini-protocol only communicates with its counter part on the remote end.

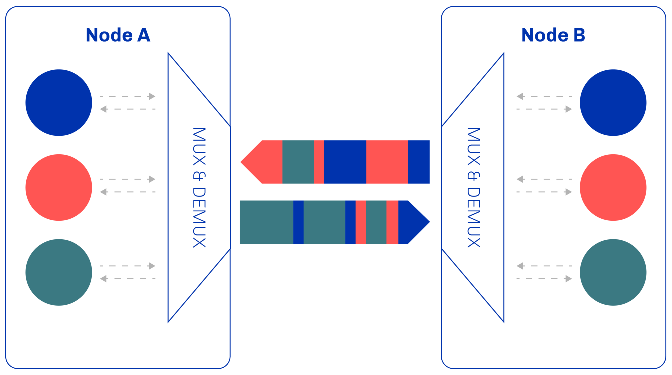


Figure 2.1: Data flow through the multiplexer and demultiplexer

The multiplexer is agnostic to the bearer it runs over. However, it assumes that the bearer guarantees an ordered

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 2.1: Multiplexer's segment data unit (SDU) encoding, see Network.Mux.Codec.

and reliable transport layer[1] and it requires the bearer to be full-duplex to allow simultaneous reads and writes[2]. The multiplexer is agnostic to the serialisation used by a mini-protocol (which we specify in section 3). Multiplexer has its own framing / binary serialisation format, described in section 2.1.1. The multiplexer allows the use of each mini-protocol in either direction.

The multiplexer exposes an interface that hides all the multiplexer details, and a single mini-protocol communication can be written as if it would only communicate with its instance on the remote end. When the multiplexer is instructed to send bytes of some mini-protocol, it splits the data into segments, adds a segment header, encodes it and transmits the segments over to the bearer. When reading data from the network, the segment's headers are used to reassemble mini-protocol byte streams.

2.1.1 Wire Format

Table 2.1 shows the layout of the service data unit (SDU) of the multiplexing protocol in big-endian bit order. The segment header contains the following data:

Transmission Time The transmission time is a time stamp based on the lower 32 bits of the sender's monotonic clock with a resolution of one microsecond.

Mini Protocol ID The unique ID of the mini-protocol as in tables 3.14 and 3.16.

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 distinguish 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.

2.1.2 Fairness and Flow-Control in the Multiplexer

The Shelley network protocol requires that the multiplexer uses a fair scheduling of the mini-protocols. Haskell's implementation of the 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 is ready to send data.

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 splitting 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.

[1] Slightly more relaxed property is required: in order delivery of multiplexer segments which belong to the same mini-protocol.

[2] Note that one can always pair two unidirectional bearers to form a duplex bearer; we use this to define a duplex bearer out of unix pipes or queues (for intra-process communication only).

2.1.3 Flow-control and Buffering in the Demultiplexer

The demultiplexer eagerly reads data from the bearer. 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 guarantees that this buffer never overflows (see Section 3.15.). 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.

For ingress buffer limits for each mini-protocol see 3.15.

For Cardano Node, each SDU for the *node-to-node mini-protocol* has the size at most of 12 288 bytes. This is not a protocol limit, Cardano Node can handle larger SDUs. In general this is implementation dependent.

Each SDU for the *node-to-client mini-protocol* has the size at most 12 288 bytes (24 576 bytes on Windows).

When receiving SDU we place a timeout. For the handshake mini-protocol we use a *10s* timeout, for all other node-to-node and node-to-client mini-protocols a *30s* timeout is used.

2.2 Node-to-node and node-to-client protocol numbers

haddock documentation: `Network.Mux.Types`
haddock documentation: `Ouroboros.Network.NodeToNode`
haddock documentation: `Ouroboros.Network.NodeToClient`

Ouroboros network defines two protocols: *node-to-node* and *node-to-client* protocols. *Node-to-node* is used for inter-node communication across the Internet, while *node-to-client* is an inter-process communication used by clients, e.g. a wallet, db-sync, etc. Each of them consists of a bundle of mini-protocols (see chapter 3). The protocol numbers of both protocols are specified in tables 3.14 and 3.16.

Chapter 3

Mini Protocols

3.1 Mini Protocols and Protocol Families

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

The mini-protocols in this section describe the initiator and responder of a communication. The initiator is the dual of the responder and vice versa. (The terms client/server, consumer/producer or initiator/responder 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.

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 blockchain consumer, such as a wallet.

3.2 Protocols as State Machines

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

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

- An informal description of the mini-protocol.
- States of the state machine.
- The messages (transitions) of the mini-protocol.
- A transition graph of the global view of the state machine.
- The client implementation of the mini-protocol.
- The server implementation of the mini-protocol.

State Machine Each mini-protocol is described as a state machine. This document uses 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 mini-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 a whole, e.g. that a client is responsible for sending a particular type of message and the server is waiting on it. This, in particular, means that if the state machine is in a given state, then both the client and server are in this state. An additional piece of information that differentiates the roles of peers in a given state is the 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 the value held 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 the 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.

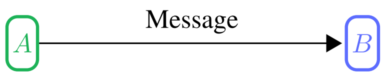
Messages Messages exchanged by peers form edges of a state machine diagram; in other words, they are transitions between states. They are elements from the set

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

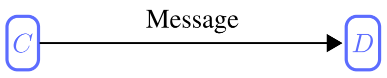
Protocols use a small set of $Labels$ typically $|Labels| \leq 10$. The state machine framework requires that messages can be serialised, transferred over the network and de-serialised by the receiver.

Agency A node has agency if it is expected to send the next message. The client or server has agency in every state, except a termination state in which nor the client, nor the server can send any message. All our mini-protocols have a single terminating state `StDone`.

State machine diagrams States are drawn as circles in state machine diagrams. States with the agency on the client side are drawn in green, states with the agency on the server side are drawn in blue, and the termination states are in black. By construction, the system is always in exactly one state, i.e. the client's state is always the same state as the server's, and the colour indicates who the agent is. 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, the arrow means: "send a message to the other peer and move to the next state". For a non-agent, an arrow in the diagram can be interpreted as: "receive an incoming message and move to the next state". This may not be very clear because the arrows are labelled with the messages, and many arrows go from a green state (the client has the agency) to a blue state (the server has the agency) or vice versa.



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 to server.



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 on 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 the mode of operation of the mini protocols.

3.3 Overview of all implemented Mini Protocols

3.3.1 Dummy mini-protocols

Dummy mini-protocols are not used by ‘cardano-node‘; however, they might be helpful when writing demos, testing purposes or getting familiar with the framework.

Ping Pong Protocol Section 3.5.1
A simple ping-pong protocol for testing.
`typed-protocols/src/Network/TypedProtocol/PingPong/Type.hs`

Request Response Protocol Section 3.5.2
A ping-pong-like protocol which allows the exchange of data.
`typed-protocols/src/Network/TypedProtocol/ReqResp/Type.hs`

3.3.2 Handshake

Handshake mini-protocol is shared by the node-to-node and node-to-client protocols (it is polymorphic to allow that).

Handshake Mini Protocol Section 3.6
This protocol is used for version negotiation.
`ouroboros-network/framework/lib/Ouroboros/Network/Protocol/Handshake/Type.hs`

3.3.3 Node-to-node mini-protocols

In this section, we list all the mini-protocols that constitute the node-to-node protocol.

Chain Synchronisation Protocol Section 3.7
The protocol by which a downstream chain consumer follows an upstream chain producer.
`ouroboros-network/protocols/lib/Ouroboros/Network/Protocol/ChainSync/Type.hs`

Block Fetch Protocol Section 3.8
The block fetching mechanism enables a node to download ranges of blocks.
`ouroboros-network/protocols/lib/Ouroboros/Network/Protocol/BlockFetch/Type.hs`

Transaction Submission Protocol v2 Section 3.9
A Protocol for transmitting transactions between core nodes.
`ouroboros-network/protocols/lib/Ouroboros/Network/Protocol/TxSubmission2/Type.hs`

Keep Alive Protocol Section 3.10
A protocol for sending keep alive messages and doing round trip measurements
`ouroboros-network/protocols/lib/Ouroboros/Network/Protocol/KeepAlive/Type.hs`

Peer Sharing Protocol Section 3.11
A mini-protocol which allows to share peer addresses
`ouroboros-network/protocols/lib/Ouroboros/Network/Protocol/PeerSharing/Type.hs`

3.3.4 Node-to-client mini-protocols

Mini-protocols used by node-to-client protocol. The chain-sync mini-protocol is shared between node-to-node and node-to-client protocols, but it is instantiated differently. In node-to-client protocol, it is used with full blocks rather than just headers.

Chain Synchronisation Protocol Section 3.7
The protocol by which a downstream chain consumer follows an upstream chain producer.
`ouroboros-network/protocols/lib/Ouroboros/Network/Protocol/ChainSync/Type.hs`

Local State Query Mini Protocol Section 3.13
Protocol used by local clients to query ledger state
`ouroboros-network/protocols/lib/Ouroboros/Network/Protocol/LocalStateQuery/Type.hs`

Local Tx Submission Mini Protocol Section 3.12
Protocol used by local clients to submit transactions
`ouroboros-network/protocols/lib/Ouroboros/Network/Protocol/LocalTxSubmission/Type.hs`

Local Tx Monitor Mini Protocol Section 3.14
Protocol used by local clients to monitor transactions
`ouroboros-network/protocols/lib/Ouroboros/Network/Protocol/LocalTxMonitor/Type.hs`

3.4 CBOR and CDDL

All mini-protocols are encoded using the concise binary object representation (CBOR), see `https://cbor.io`. Each codec comes along with a specification written in CDDL, see 'Coincise data definition language (CDDL)'.

The networking layer knows little about blocks, transactions or their identifiers. In `ouroboros-network` we use parametric polymorphism for blocks, tx, txids, etc, and we only assume these data types have their own valid CDDL encoding (and CDDL specifications). For testing against the `ouroboros-network` CDDL, we need concrete values; for this reason, we use `any` in our CDDL specification. This describes very closely what the `ouroboros-network` implementation does. It doesn't mean the payloads are not validated, the full codecs of messages transferred on the wire are composed from network, consensus & ledger codecs. There is an ongoing effort to capture combined CDDLs. If you want to find concrete instantiations of these types by 'Cardano', you will need to consult cardano-ledger and ouroboros-consensus (in particular ouroboros-consensus#1422). Each ledger era has its own CDDL spec, which you can find here. Note that the hard fork combinator (HFC) also allows us to combine multiple eras into a single blockchain. It affects how many of the data types are encoded across different eras.

We want to retain the ability to decode messages incrementally, which for the Praos protocol might allow us to improve performance.

3.5 Dummy Protocols

Dummy protocols are only used for testing and are not needed either for Node-to-Node nor for the Node-to-Client protocols.

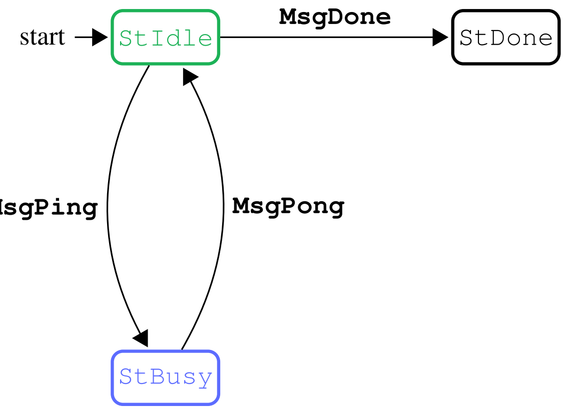
3.5.1 Ping-Pong mini-protocol

haddock documentation: `Network.TypedProtocol.PingPong.Type`

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.

State Machine



state	**agency**
`StIdle`	**Client**
`StBusy`	**Server**

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

`MsgPing` The client sends a Ping request to the server.

`MsgPong` The server replies to a Ping with a Pong.

`MsgDone` Terminate the protocol.

from state	**message**	**to state**
`StIdle`	**`MsgPing`**	`StBusy`
`StBusy`	**`MsgPong`**	`StIdle`
`StIdle`	**`MsgDone`**	`StDone`

Table 3.1: Ping-Pong mini-protocol messages.

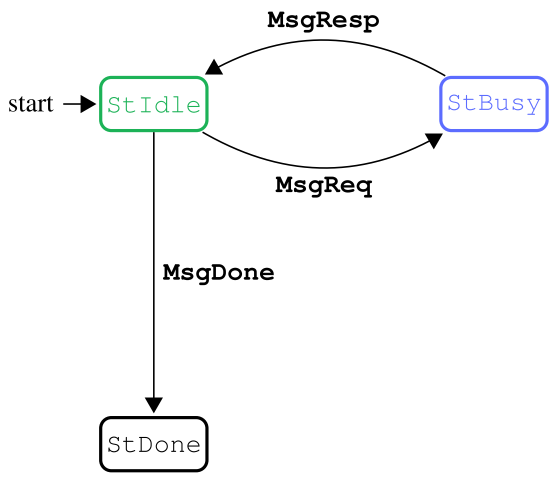
3.5.2 Request-Response mini-protocol

haddock documentation: `Network.TypedProtocol.ReqResp.Type`

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 requests and responses.

State machine



state	agency
StIdle	**Client**
StBusy	**Server**

The protocol uses the following messages.

MsgReq (*request*) The client sends a request to the server.

MsgResp (*response*) The server replies with a response.

MsgDone (*done*) Terminate the protocol.

from	message	parameters	to
StIdle	**MsgReq**	*request*	StBusy
StBusy	**MsgResp**	*response*	StIdle
StIdle	**MsgDone**		StDone

Table 3.2: Request-Response mini-protocol messages.

3.6 Handshake mini-protocol

protocol haddocks: `Ouroboros.Network.Protocol.Handshake.Type`
codec haddocks: `Ouroboros.Network.Protocol.Handshake.Codec`
node-to-node mini-protocol number: 0
node-to-client mini-protocol number: 0
node-to-client handshake CDDL spec

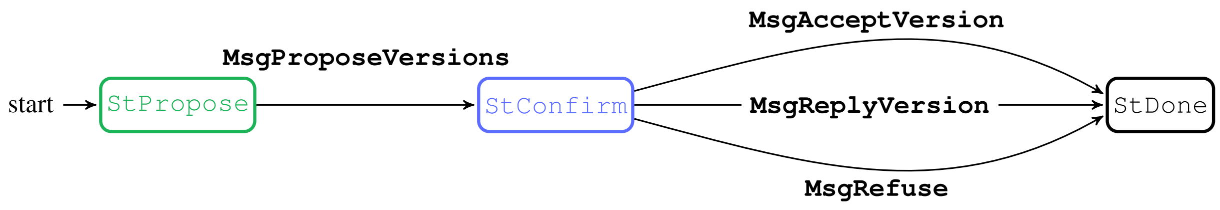
3.6.1 Description

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 initialised and consists of a single request from the client and a single reply from the server.

The handshake mini protocol is a generic protocol that can negotiate version number and protocol parameters (these my depend on the version number). 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 to handle the protocol parameters. These callback functions are specific to the supported protocol versions.

The handshake mini protocol is designed to handle simultaneous TCP open.

3.6.2 State machine



state	agency
`StPropose`	**Client**
`StConfirm`	**Server**

Messages of the protocol:

`MsgProposeVersions` $(versionTable)$ The client proposes a number of possible versions and protocol parameters. $versionTable$ is a map from version numbers to their associated version data. Note that different version numbers might use different version data (e.g. supporting a different set of parameters).

`MsgReplyVersion` $(versionTable)$ This message must not be explicitly sent, it's only to support TCP simultaneous open scenario in which both sides sent **`MsgProposeVersions`**. In this case, the received **`MsgProposeVersions`** is interpreted as **`MsgReplyVersion`** and thus it MUST have the same CBOR decoding as **`MsgProposeVersions`**.

`MsgAcceptVersion` $(versionNumber, extraParameters)$ The server accepts $versionNumber$ and returns possible extra protocol parameters.

`MsgRefuse` $(reason)$ The server refuses the proposed versions.

from	message	parameters	to
`StPropose`	**`MsgProposeVersions`**	$versionTable$	`StConfirm`
`StConfirm`	**`MsgReplyVersion`**	$versionTable$	`StDone`
`StConfirm`	**`MsgAcceptVersion`**	$(versionNumber, versionData)$	`StDone`
`StConfirm`	**`MsgRefuse`**	$reason$	`StDone`

3.6.3 Size limits per state

These bounds limit how many bytes can be sent in a given state; indirectly, this limits the payload size of each message. If a space limit is violated, the connection SHOULD be torn down.

state	size limit in bytes
`StPropose`	5760
`StConfirm`	5760

3.6.4 Timeouts per state

These limits bound how much time the receiver side can wait for the arrival of a message. If a timeout is violated, the connection SHOULD be torn down.

state	**timeout**
`StPropose`	`10s`
`StConfirm`	`10s`

Table 3.3: timeouts per state

3.6.5 Node-to-node handshake

codec haddocks: `Cardano.Network.NodeToNode`

The node-to-node handshake instantiates version data[1] to a record which consists of

network magic a `Word32` value;

diffusion mode a boolean value: `True` value indicates initiator only mode, `False` - initiator and responder mode;

peer sharing either 0 or 1: `1` indicates that the node will engage in peer sharing (and thus it will run the PeerSharing mini-protocol);

query a boolean value: `True` will send back all supported versions & version data and terminate the connection.

When negotiating a connection, each side will have access to local and remote version data associated with the negotiated version number. The result of negotiation is a new version data record which consists of:

- if the network magic agrees, then it is inherited by the negotiated version data, otherwise the negotiation fails;
- diffusion mode SHOULD be initiator only if and only if any side proposes the initiator-only mode (i.e. the logical disjunction operator);
- peer sharing SHOULD be inherited from the remote side;
- query SHOULD be inherited from the client (the side that sent **`MsgProposeVersions`**).

If the negotiation is successful, the negotiated version data is sent back using **`MsgAcceptVersion`**, otherwise **`MsgRefuse`** SHOULD be sent.

Size limits per state

These bounds limit how many bytes can be sent in a given state; indirectly, this limits the payload size of each message. If a space limit is violated, the connection SHOULD be torn down.

state	**size limit in bytes**
`StPropose`	`5760`
`StConfirm`	`5760`

Timeouts per state

These limits bound how much time the receiver side can wait for the arrival of a message. If a timeout is violated, the connection SHOULD be torn down.

[1]To be precise, in ouroboros-network, we instantiate version data to CBOR terms and do encoding / decoding of version data lazily (as required) rather than as part of the protocol codec (the protocol codec only decodes bytes to CBOR terms, and thus fails only if received bytes are not a valid CBOR encoding). This is important in order to support receiving a mixture of known and unknown versions. The same the remark applies to the node-to-client protocol as well.

state	timeout
`StPropose`	`10s`
`StConfirm`	`10s`

3.6.6 Node-to-client handshake

codec haddocks: `Cardano.Network.NodeToClient`

The node-to-node handshake instantiates version data to a record which consists of

network magic a `Word32` value;

query a boolean value: `True` will send back all supported; versions & version data and terminate the connection.

The negotiated version data is computed similarly as in the node-to-node protocol:

- if the network magic agrees, then it is inherited by the negotiated version data, otherwise the negotiation fails;
- query SHOULD be inherited from the client (the side that sent **`MsgProposeVersions`**).

If the negotiation is successful, the negotiated version data is sent back using **`MsgAcceptVersion`**, otherwise **`MsgRefuse`** SHOULD be sent.

Size limits per state

These bounds limit how many bytes can be sent in a given state; indirectly, this limits the payload size of each message. If a space limit is violated, the connection SHOULD be torn down.

state	size limit in bytes
`StPropose`	`5760`
`StConfirm`	`5760`

Timeouts per state

No timeouts are used for node-to-client handshake.

3.6.7 Client and Server Implementation

Section 3.6.9 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 the key and the protocol parameters as a 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. Therefore, 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 initiates the protocol with a **`MsgProposeVersions`** message that contains information about all protocol versions it wants to support. The server replies either with an **`MsgAcceptVersion`** message containing the negotiated version number and version data or a **`MsgRefuse`** message. The **`MsgRefuse`** message contains one of three alternative refuse reasons: **`VersionMismatch`**, **`HandshakeDecodeError`** or just **`Refused`**.

When a server receives a **`MsgProposeVersions`** message, it uses the following algorithm to compute the response:

1. Compute the intersection of the set of protocol version numbers that the server supports and the version numbers requested by the client.

2. If the intersection is empty: Reply with **`MsgRefuse`**(**`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 **`MsgRefuse`**(**`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 **`MsgRefuse`**(**`Refused`**), the selected version number and an error message.

8. Otherwise, compute negotiation parameters according to the algorithm 3.6.5 or 3.6.6, encode them with the corresponding CBOR codec and reply with **`MsgAcceptVersion`**, 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 a set of protocols that the server supports. The handshake protocol is designed so that a server can always handle requests for protocol versions that it does not support. The server simply ignores the CBOR terms that represent the protocol parameters of unsupported versions.

In case of simultaneous open of a TCP connection, both handshake clients will send their **`MsgProposeVersions`**, and both will interpret the incoming message as **`MsgReplyVersion`** (thus, both must have the same encoding; the implementation can distinguish them by the protocol state). Both clients should choose the highest version of the protocol available. If any side does not accept any version (or its parameters), the connection can be reset.

The protocol does not forbid, nor could it detect a usage of **`MsgReplyVersion`** outside of TCP simultaneous open. The process of choosing between the proposed and received version must be symmetric in the following sense.

We use `acceptable :: vData -> vData -> Accept vData` function to compute accepted version data from local and remote data, where

```
data Accept vData = Accept vData
                  | Refuse Text
                  deriving Eq
```

See ref. Both `acceptable local remote` and `acceptable remote local` must satisfy the following conditions:

- if either of them accepts a version by returning `Accept`, the other one must accept the same value, i.e. in this case `acceptable local remote == acceptable remote local`
- if either of them refuses to accept (returns `Refuse reason`) the other one SHOULD return `Refuse` as well.

Note that the above condition guarantees that if either side returns `Accept`, then the connection will not be closed by the remote end. A weaker condition, in which the return values are equal if they both return `Accept` does not guarantee this property. We also verify that the whole Handshake protocol, not just the `acceptable` satisfies the above property, see Ouroboros-Network test suite.

The fact that we are using non-injective encoding in the handshake protocol side steps typed-protocols strong typed-checked properties. For injective codecs (i.e. codecs for which each message has a distinguished encoding), both sides of typed-protocols are always in the same state (once all in-flight the message arrived). This is no longer true in general; however, this is still true for the handshake protocol. Even though the opening message **`MsgProposeVersions`** of a simultaneous open will materialise on the other side as a termination message **`MsgReplyVersion`**, and the same will happen to the **`MsgProposeVersions`** transmitted in the other direction. We include a special test case (`prop_channel_simultaneous_open`) to verify that simultaneous open behaves well and does not lead to protocol errors.

3.6.8 Handshake and the multiplexer

The handshake mini protocol runs before the multiplexer is initialised. Each message is transmitted within a single MUX segment, i.e. with a proper segment header, but as the multiplexer is not yet running, the messages MUST not be split into multiple segments. The Handshake protocol uses the mini-protocol number 0 in both node-to-node and node-to-client cases.

3.6.9 CDDL encoding specification

There are two flavours of the mini-protocol that only differ with type instantiations, e.g., different protocol versions and version data carried in messages. First, one is used by the node-to-node protocol, and the other is used by the node-to-client protocol.

Node-to-node handshake mini-protocol

```
;
; NodeToNode Handshake (>=v14)
;
handshakeMessage
    = msgProposeVersions
    / msgAcceptVersion
    / msgRefuse
    / msgQueryReply

msgProposeVersions = [0, versionTable]
msgAcceptVersion   = [1, versionNumber_v14, v14.nodeToNodeVersionData]
msgRefuse          = [2, refuseReason]
msgQueryReply      = [3, versionTable]

; The codec only accepts definite-length maps.
versionTable = { * versionNumber_v14 => v14.nodeToNodeVersionData }

versionNumber_v14 = 14 / 15

; All version numbers
versionNumbers = versionNumber_v14

refuseReason
    = refuseReasonVersionMismatch
    / refuseReasonHandshakeDecodeError
    / refuseReasonRefused

refuseReasonVersionMismatch      = [0, [ *versionNumbers ] ]
refuseReasonHandshakeDecodeError = [1, versionNumbers, tstr]
refuseReasonRefused              = [2, versionNumbers, tstr]

;# import node-to-node-version-data-v14 as v14
;# import network.base as base
```

Node-to-client handshake mini-protocol

```
;
; NodeToClient Handshake
;

handshakeMessage
    = msgProposeVersions
    / msgAcceptVersion
```

```
      / msgRefuse
      / msgQueryReply

msgProposeVersions = [0, versionTable]
msgAcceptVersion   = [1, versionNumber, nodeToClientVersionData]
msgRefuse          = [2, refuseReason]
msgQueryReply      = [3, versionTable]

; Entries must be sorted by version number. For testing, this is handled in `handshakeFix`.
; The codec only accepts definite-length maps.
versionTable = { * versionNumber => nodeToClientVersionData }


; as of version 2 (which is no longer supported) we set 16th bit to 1
;               16    / 17    / 18    / 19    / 20    / 21    / 22    / 23
versionNumber = 32784 / 32785 / 32786 / 32787 / 32788 / 32789 / 32790 / 32791

; As of version 15 and higher
nodeToClientVersionData = [networkMagic, query]

networkMagic = uint
query        = bool

refuseReason
    = refuseReasonVersionMismatch
    / refuseReasonHandshakeDecodeError
    / refuseReasonRefused

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

3.7 Chain-Sync mini-protocol

protocol haddocks: `Ouroboros.Network.Protocol.ChainSync.Type`
codec haddocks: `Ouroboros.Network.Protocol.ChainSync.Codec`
node-to-node mini-protocol number: `2`
node-to-client mini-protocol number: `5`

3.7.1 Description

The chain synchronisation protocol is used by a blockchain consumer to replicate the producer's blockchain locally. A node communicates with several upstream and downstream nodes and runs an independent client instance and an independent server instance for every other node it communicates with. (See Figure 1.1.)

The chain synchronisation protocol is polymorphic. The node-to-client protocol uses an instance of the chain synchronisation protocol that transfers full blocks, while the node-to-node instance only transfers block headers. In the node-to-node case, the block fetch protocol (Section 3.8) is used to diffuse full blocks.

3.7.2 State Machine

The protocol uses the following messages:

`MsgRequestNext` Request the next update from the producer. The response can be a roll forward, a roll back or wait.

`MsgAwaitReply` 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.

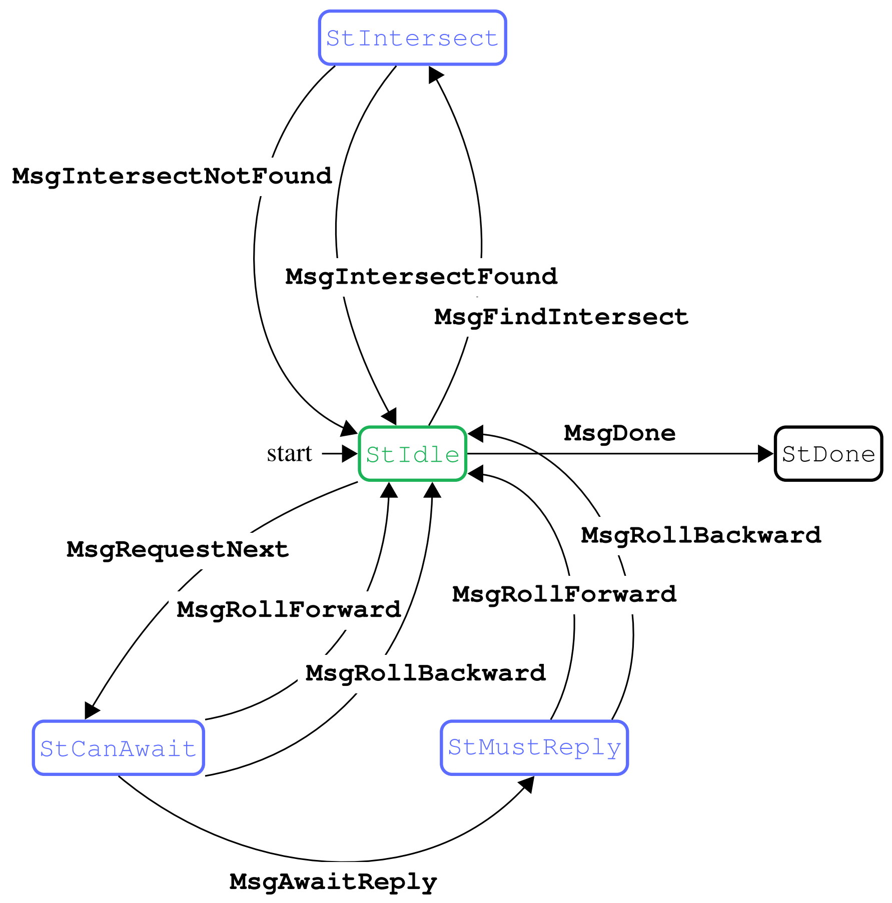


Figure 3.1: State machine of the Chain-Sync mini-protocol

MsgRollForward $(\boldsymbol{header}, \boldsymbol{tip})$ Tell the consumer to extend their chain with the given $header$. The message also tells the consumer about the tip of the producer's chain.

MsgRollBackward $(\boldsymbol{point_{old}}, \boldsymbol{tip}$ Tell the consumer to roll back to a given $point_{old}$ on their chain. The message also tells the consumer about the current tip of the chain the producer is following.

MsgFindIntersect $[\boldsymbol{point_{head}}]$ Ask the producer to try to find an improved intersection point between the consumer and producer's chains. The consumer sends a sequence $[\boldsymbol{point}]$, which shall be ordered by preference (e.g. points with the highest slot number first), and it is up to the producer to find the first intersection point on its chain and send it back to the consumer. If an empty list of points is sent with **MsgFindIntersect**, the server will reply with **MsgIntersectNotFound**.

MsgIntersectFound $(\boldsymbol{point_{intersect}}, \boldsymbol{tip})$ The producer replies with the first point of the request, which is on his current chain. The consumer can decide whether to send more points. The message also tells the consumer about the tip of the producer. Whenever the server replies with **MsgIntersectFound**, the client can expect the next update (i.e. a reply to **MsgRequestNext**) to be **MsgRollBackward**, either to the specified $point_{intersect}$ or an earlier point if the producer switched to a different fork in the meantime. This makes handling state updates on the client side easier.

MsgIntersectNotFound $(\boldsymbol{tip})$ Reply to the consumer that no intersection was found: none of the points the consumer supplied are on the producer chain. The message only contains the tip of the producer chain.

MsgDone Terminate the protocol.

state	agency
StIdle	Client
StIntersect	Server
StCanAwait	Server
StMustReply	Server

Figure 3.2: Chain-Sync state agencies

from state	message	parameters	to state
StIdle	**MsgRequestNext**		StCanAwait
StIdle	**MsgFindIntersect**	$[point]$	StIntersect
StIdle	**MsgDone**		StDone
StCanAwait	**MsgAwaitReply**		StMustReply
StCanAwait	**MsgRollForward**	$header, tip$	StIdle
StCanAwait	**MsgRollBackward**	$point_{old}, tip$	StIdle
StMustReply	**MsgRollForward**	$header, tip$	StIdle
StMustReply	**MsgRollBackward**	$point_{old}, tip$	StIdle
StIntersect	**MsgIntersectFound**	$point_{intersect}, tip$	StIdle
StIntersect	**MsgIntersectNotFound**	tip	StIdle

Table 3.4: Chain-Sync mini-protocol messages.

3.7.3 Node-to-node size limits per state

Table 3.5 specifies how many bytes can be sent in a given state in the chain-sync mini-protocol of the node-to-node protocol; indirectly, this limits the payload size of each message. If a space limit is violated, the connection SHOULD be torn down.

state	size limit in bytes
StIdle	65535
StCanAwait	65535
StMustReply	65535
StIntersect	65535

Table 3.5: size limits per state

3.7.4 Node-to-node timeouts per state

The table 3.6 specifies message timeouts in a given state. If a timeout is violated, the connection SHOULD be torn down.

state	timeout
StIdle	3673s
StCanAwait	10s
StMustReply	random between 135s and 269s
StIntersect	10s

Table 3.6: timeouts per state

3.7.5 Node-to-client size limits and timeouts

There are no size-limits nor timeouts for the chain-sync mini-protocol of the node-to-client protocol.

3.7.6 Implementation of the Chain Producer

This section describes a stateful 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 the consumer. By running many instances of this protocol against different peers, a node can reproduce chains in the network and make chain selection, which by design is not part of this protocol. Note that when we refer to the consumer's chain in this section, we mean the chain that is reproduced by the consumer with the instance of the chain-sync protocol and not the result of the chain selection algorithm.

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-pointer*s are part of the shared state of the node (Figure 1.1), 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.

Initializing the *read-pointer*. The chain producer assumes that a consumer which has just connected, only knows the genesis block and initialises 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 the genesis). In this case, the protocol follows a simple, consumer-driven, request-response pattern. The consumer sends **`MsgRequestNext`** 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 **`MsgRollForward`** and advances the *read-pointer* to the next block (optimistically assuming that the client will update its chain accordingly). The **`MsgRollForward`** 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.

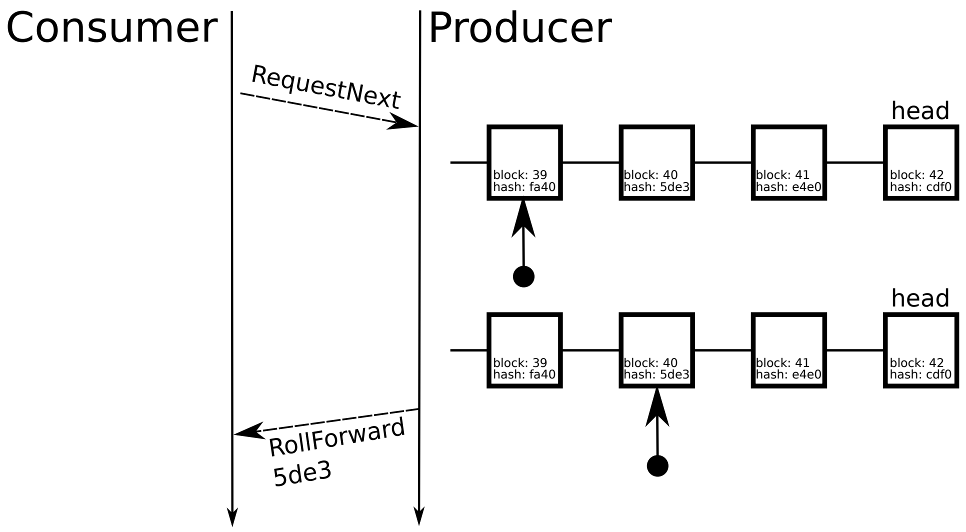


Figure 3.3: Consumer-driven block download.

Producer driven updates If the *read-pointer* points to the end of the chain and the producer receives a **`MsgRequestNext`** the consumer's chain is already up to date. The producer informs the consumer with an **`MsgAwaitReply`** that no new data is available. After receiving a **`MsgAwaitReply`**, the consumer waits for a new message, and the producer keeps agency. The **`MsgAwaitReply`** 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 **`MsgRollForward`** 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 the 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 **`MsgRollBackward`** instead of a **`MsgRollForward`**. Finally, the producer thread must unblock if it is in the `StMustReply` state.

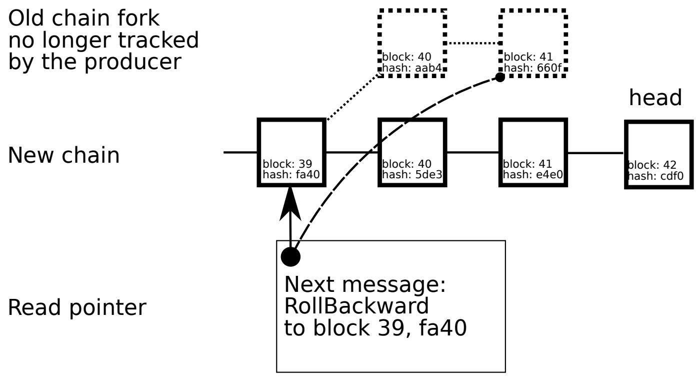


Figure 3.4: *read-pointer* update for a fork switch in case of a rollback.

Figure 3.4 illustrates a fork switch that requires an update of the *read-pointer* for one of the chain consumers. 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 **`MsgRollBackward`**.

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 blockchain when it starts to track the producer. The protocol provides an efficient method to search for the longest 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 **`MsgFindIntersect`** message with a list of chain points on the chain known to the consumer. If the producer does not know any of the points, it replies with **`MsgIntersectNotFound`**. Otherwise, it replies with **`MsgIntersectFound`** 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 **`MsgFindIntersect`** in a fresh connection and it is free to use **`MsgFindIntersect`** at any time later as it is beneficial. If the consumer does not know anything about the producer's chain, it can start the search with the following list of points: $[point(b), point(b-1), point(b-2), point(b-4), point(b-8), \ldots]$ where $point(b-i)$ is the point of the ith predecessor of block b and b is the head of the consumer fork. The maximum depth of a fork in Ouroboros is bounded, and the intersection will always be found with a small number of iterations of this algorithm.

Additional remarks Note that by sending **`MsgFindIntersect`**, the server will not modify its *read-pointer*.

3.7.7 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 a role in favour of the consumer here.

- The protocol is consumer-driven, i.e., the producer cannot send unsolicited data to the consumer (within the protocol).
- The consumer can verify the response data itself.

Here are some cases to consider:

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

The producer replies with a `MsgRollForward` 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 `MsgRollBackward` The consumer tracks several producers, so if the producer sends false **`MsgRollBackward`** messages, the consumer's node will, at some point, 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.

3.7.8 CDDL encoding specification

```
chainSyncMessage
       = msgRequestNext
       / msgAwaitReply
       / msgRollForward
       / msgRollBackward
       / msgFindIntersect
       / msgIntersectFound
       / msgIntersectNotFound
       / chainSyncMsgDone

msgRequestNext            = [0]
msgAwaitReply             = [1]
msgRollForward            = [2, base.header, base.tip]
msgRollBackward           = [3, base.point, base.tip]
msgFindIntersect          = [4, base.points]
msgIntersectFound         = [5, base.point, base.tip]
msgIntersectNotFound      = [6, base.tip]
chainSyncMsgDone          = [7]

;# import network.base as base
```

See appendix A for common definitions.

3.8 Block-Fetch mini-protocol

protocol haddocks: `Ouroboros.Network.Protocol.BlockFetch.Type`
codec haddocks: `Ouroboros.Network.Protocol.BlockFetch.Codec`

node-to-node mini-protocol number: `3`

3.8.1 Description

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

3.8.2 State machine

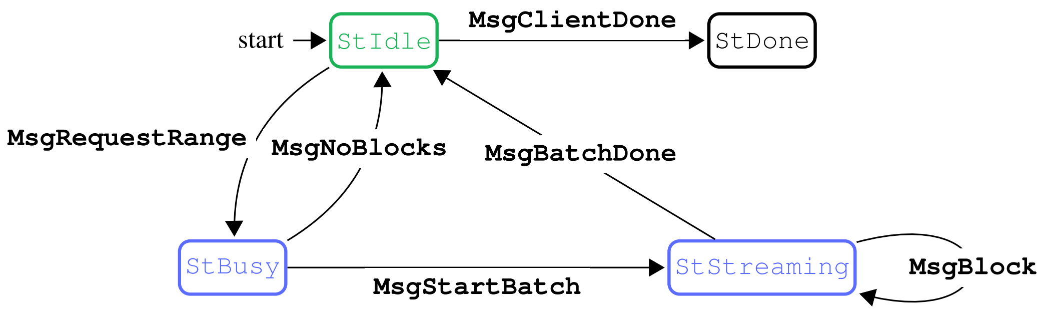


Figure 3.5: State machine of the block-fetch mini-protocol

state	agency
`StIdle`	**Client**
`StBusy`	**Server**
`StStreaming`	**Server**

Figure 3.6: Block-Fetch state agencies

Protocol messages

`MsgRequestRange` $(range)$ The client requests a $range$ of blocks from the server. The range is inclusive on both sides.

`MsgNoBlocks` The server tells the client that it does not have all of the blocks in the requested $range$.

`MsgStartBatch` The server starts block streaming.

`MsgBlock` $(body)$ Stream a single block's body.

`MsgBatchDone` The server ends block streaming.

`MsgClientDone` The client terminates the protocol.

The transitions are shown in table 3.7.

3.8.3 Size limits per state

These bounds limit how many bytes can be sent in a given state; indirectly, this limits the payload size of each message. If a space limit is violated, the connection SHOULD be torn down.

from state	message	parameters	to state
`StIdle`	**`MsgClientDone`**		`StDone`
`StIdle`	**`MsgRequestRange`**	*range*	`StBusy`
`StBusy`	**`MsgNoBlocks`**		`StIdle`
`StBusy`	**`MsgStartBatch`**		`StStreaming`
`StStreaming`	**`MsgBlock`**	*body*	`StStreaming`
`StStreaming`	**`MsgBatchDone`**		`StIdle`

Table 3.7: Block-Fetch mini-protocol messages.

state	size limit in bytes
`StIdle`	65535
`StBusy`	65535
`StStreaming`	2500000

state	timeout
`StIdle`	-
`StBusy`	60s
`StStreaming`	60s

Table 3.8: timeouts per state

3.8.4 Timeouts per state

These limits bound how much time the receiver side can wait for the arrival of a message. If a timeout is violated, the connection SHOULD be torn down.

3.8.5 CDDL encoding specification

```
;
; BlockFetch mini-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, base.point, base.point]
msgClientDone   = [1]
msgStartBatch   = [2]
msgNoBlocks     = [3]
msgBlock        = [4, base.block]
msgBatchDone    = [5]

;# import network.base as base
```

See appendix A for common definitions.

3.9 Tx-Submission mini-protocol

protocol haddocks: `Ouroboros.Network.Protocol.TxSubmission2.Type`
codec haddocks: `Ouroboros.Network.Protocol.TxSubmission2.Codec`
node-to-node mini-protocol number: `4`

Description

The node-to-node transaction submission protocol is used to transfer transactions between full nodes. The protocol follows a pull-based strategy where the initiator asks for new transactions, and the responder sends them back. It is suitable for a trustless setting where both sides need to guard against resource consumption attacks from the other side. The local transaction submission protocol, is a simpler which is used when the server trusts a local client, is described in Section 3.12.

The *tx-submission* mini-protocol is designed in a way that the information (e.g. transactions) flows across the system in the other direction than in the *chain-sync* or *block-fetch* protocols. Transactions must flow toward the block producer, while headers and blocks disseminate from it to the rest of the system. This is reflected in the protocol graphs, transactions are sent from a client to a server. However, to preserve that all mini-protocols start on the client, the `StInit` state was added in version 2 of the protocol.

Note that Version 1 of the tx-submission protocol is no longer supported. Version 2 is used since `NodeToNode_V6` of the node-to-node protocol.

3.9.1 State machine

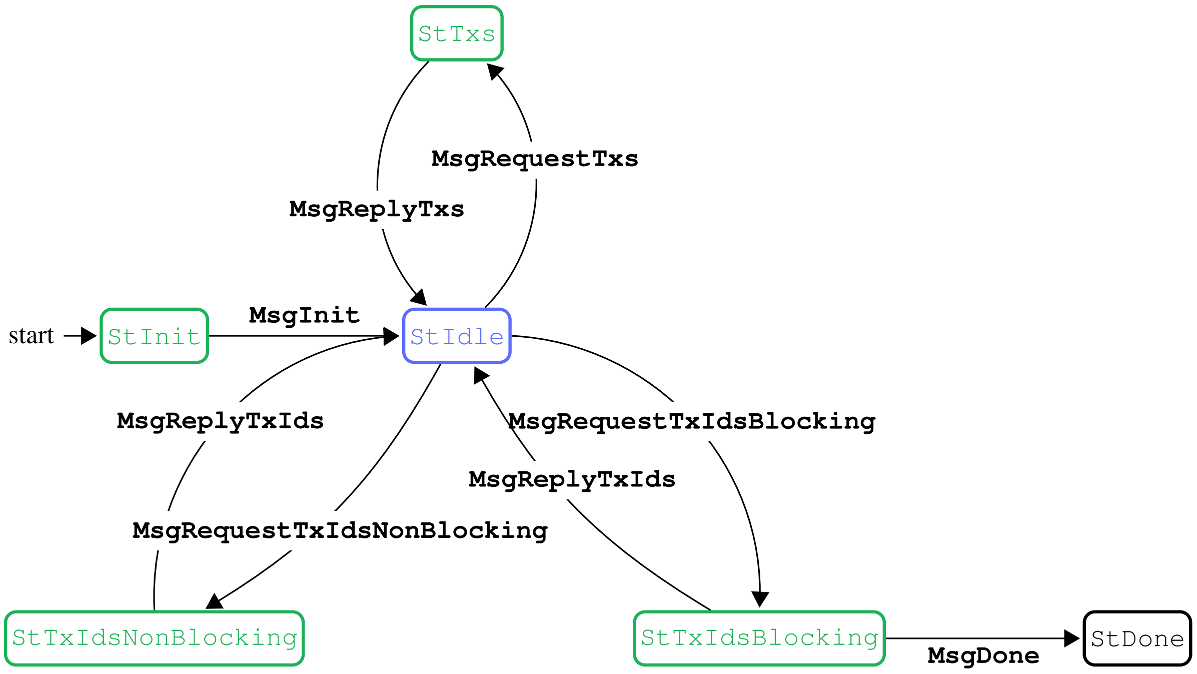


Figure 3.7: State machine of the Tx-Submission mini-protocol (version 2).

Protocol messages

`MsgInit` initial message of the protocol

state	agency
`StInit`	Client
`StIdle`	Server
`StTxIdsBlocking`	Client
`StTxIdsNonBlocking`	Client
`StTxs`	Client

Figure 3.8: Tx-Submission state agencies

`MsgRequestTxIdsNonBlocking` (ack, req) Request a non-empty list of transaction identifiers from the client, and confirm a number of outstanding transaction identifiers.

This is a non-blocking operation: the response may be an empty list and this does expect a prompt response. This covers high throughput use cases where we wish to pipeline, by interleaving requests for additional transaction identifiers with requests for transactions, which requires these requests not block.

The request gives the maximum number of transaction identifiers that can be accepted in the response. Either the numbers acknowledged or the number requested MUST be non-zero. In either case, the number requested MUST not put the total outstanding over the fixed protocol limit (see below in section 3.9.2).

The request also gives the number of outstanding transaction identifiers that can now be acknowledged. The actual transactions to acknowledge are known to the peer based on the FIFO order in which they were provided.

The request MUST be made (over **`MsgRequestTxIdsBlocking`**) if there are non-zer remaining unacknowledged transactions.

`MsgRequestTxIdsBlocking` (ack, req) The server asks for new transaction ids and acknowledges old ids. The client will block until new transactions are available, thus the respond will always have at least one transaction identifier.

This is a blocking operation: the response will always have at least one transaction identifier, and it does not expect a prompt response: there is no timeout. This covers the case when there is nothing else to do but wait. For example this covers leaf nodes that rarely, if ever, create and submit a transaction.

The request gives the maximum number of transaction identifiers that can be accepted in the response. This must be greater than zero. The number requested ids MUST not put the total outstanding over the fixed protocol limit (see below in section 3.9.2).

The request also gives the number of outstanding transaction identifiers that can now be acknowledged. The actual transactions to acknowledge are known to the peer based on the FIFO order in which they were provided.

The request MUST be made (over **`MsgRequestTxIdsNonBlocking`**) if there are zero remaining unacknowledged transactions.

`MsgReplyTxIds` $([(id, size)])$ The client replies with a list of available transactions. The list contains pairs of transaction ids and the corresponding size of the transaction in bytes. In the blocking case, the reply MUST contain at least one transaction identifier. In the non-blocking case, the reply may contain an empty list.

These transactions are added to the notional FIFO of outstanding transaction identifiers for the protocol.

The order in which these transaction identifiers are returned must be the order in which they are submitted to the mempool, to preserve dependent transactions.

`MsgRequestTxs` $([ids])$ The server requests transactions by sending a non-empty list of transaction-ids.

While it is the responsibility of the replying peer to keep within pipelining in-flight limits, the sender must also cooperate by keeping the total requested across all in-flight requests within the limits.

It is an error to ask for transaction identifiers that were not previously announced (via **`MsgReplyTxIds`**).

It is an error to ask for transaction identifiers that are not outstanding or that were already asked for.

`MsgReplyTxs` ($[txs]$) The client replies with a list of transactions. It may implicitly discard transaction-ids which were requested.

Transactions can become invalid between the time the transaction identifier was sent and the transaction being requested. Invalid (including committed) transactions do not need to be sent.

Any transaction identifiers requested but not provided in this reply should be considered as if this peer had never announced them. (Note that this is no guarantee that the transaction is invalid, it may still be valid and available from another peer).

`MsgDone` Termination message, initiated by the client when the server is making a blocking call for more transaction identifiers.

from state	message	parameters	to state
`StInit`	**`MsgInit`**		`StIdle`
`StIdle`	**`MsgRequestTxIdsNonBlocking`**	ack,req	`StTxIdsNonBlocking`
`StIdle`	**`MsgRequestTxIdsBlocking`**	ack,req	`StTxIdsBlocking`
`StTxIdsNonBlocking`	**`MsgReplyTxIds`**	$[(id, size)]$	`StIdle`
`StTxIdsBlocking`	**`MsgReplyTxIds`**	$[(id, size)]$	`StIdle`
`StIdle`	**`MsgRequestTxs`**	$[ids]$	`StTxs`
`StTxs`	**`MsgReplyTxs`**	$[txs]$	`StIdle`
`StTxIdsBlocking`	**`MsgDone`**		`StDone`

Table 3.9: Tx-Submission mini-protocol (version 2) messages.

3.9.2 Size limits per state

Table 3.10 specifies how many bytes can be sent in a given state; indirectly, this limits the payload size of each message. If a space limit is violated, the connection SHOULD be torn down.

state	size limit in bytes
`StInit`	`5760`
`StIdle`	`5760`
`StTxIdsBlocking`	`2500000`
`StTxIdsNonBlocking`	`2500000`
`StTxs`	`2500000`

Table 3.10: size limits per state

Maximum number of unacknowledged transaction identifiers

The maximal number of unacknowledged transactions ids is `10`. It is a protocol error to exceed it.

3.9.3 Timeouts per state

The table 3.11 specifies message timeouts in a given state. If a timeout is violated, the connection SHOULD be torn down.

3.9.4 CDDL encoding specification

```
;
; TxSubmission mini-protocol v2
;
```

state	timeout
StInit	-
StIdle	-
StTxIdsBlocking	-
StTxIdsNonBlocking	10s
StTxs	10s

Table 3.11: timeouts per state

```

; reference implementation of the codec in :
; ouroboros-network/src/Ouroboros/Network/Protocol/TxSubmission2/Codec.hs

txSubmission2Message
    = msgInit
    ; corresponds to either MsgRequestTxIdsBlocking or
    ; MsgRequestTxIdsNonBlocking in the spec
    / msgRequestTxIds
    / msgReplyTxIds
    / msgRequestTxs
    / msgReplyTxs
    / tsMsgDone


msgInit         = [6]
msgRequestTxIds = [0, tsBlocking, txCount, txCount]
msgReplyTxIds   = [1, txIdsAndSizes ]
msgRequestTxs   = [2, txIdList ]
msgReplyTxs     = [3, txList ]
tsMsgDone       = [4]

tsBlocking      = false / true
txCount         = base.word16
; The codec only accepts indefinite-length lists.
txIdList        = [ *base.txId ]
txList          = [ *base.tx ]
txIdAndSize     = [base.txId, txSizeInBytes]
; The codec only accepts indefinite-length lists.
txIdsAndSizes   = [ *txIdAndSize ]
txSizeInBytes   = base.word32

;# import network.base as base
```

3.9.5 Client and Server Implementation

The protocol has two design goals: It must diffuse transactions with high efficiency and, at the same time, it must rule out asymmetric resource attacks from the transaction consumer against the transaction provider.

The protocol is based on two pull-based operations. The transaction consumer can ask for a number of transaction ids, and it can use these transaction ids to request a batch of transactions. The transaction consumer has flexibility in the number of transaction ids it requests, whether to actually download the transaction body and flexibility in how it batches the download of transactions. The transaction consumer can also switch between requesting transaction ids and downloading transaction bodies at any time. It must, however, observe several constraints that are necessary for a memory-efficient implementation of the transaction provider.

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

After the consumer requests new transaction ids, the provider replies with a list of transaction ids and puts these transactions in its FIFO. As part of a request, a consumer also acknowledges the number of old transactions, which are removed from the FIFO at the same time. The provider checks that the size of the FIFO, i.e. the number of outstanding transactions, never exceeds the protocol limit and aborts the connection if a request violates the limits. The consumer can request any batch of transactions from the current FIFO in any order. Note, however, that the reply will omit any transactions that have become invalid in the meantime. (More precisely, the server will omit invalid transactions from the reply, but they will still be counted in the FIFO size, and they will still require an acknowledgement from the consumer).

The protocol supports blocking and non-blocking requests for new transactions ids. If the FIFO is empty, the consumer must use a blocking request; otherwise, it must be a non-blocking request. The producer must reply immediately (i.e. within a small timeout) to a non-blocking request. It replies with not more than the requested number of ids (possibly with an empty list). A blocking request, on the other side, waits until at least one transaction is available.

3.10 Keep Alive Mini Protocol

protocol haddocks: `Ouroboros.Network.Protocol.KeepAlive.Type`
codec haddocks: `Ouroboros.Network.Protocol.KeepAlive.Codec`
node-to-node mini-protocol number: `8`

3.10.1 Description

Keep-alive mini-protocol is a member of the node-to-node protocol. It is used for two purposes: to provide keep alive messages and do round trip time measurements.

3.10.2 State machine

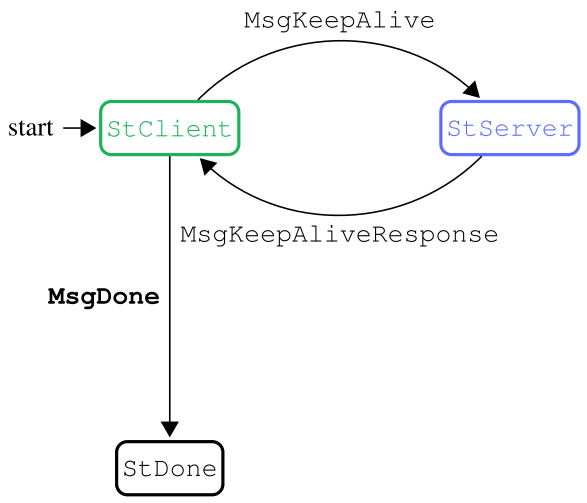


Figure 3.9: State machine of the keep-alive protocol.

Protocol messages

`MsgKeepAlive` *cookie* Keep alive message. The *cookie* value is a `Word16` value, which allows to match requests with responses. It is a protocol error if the cookie received back with `MsgKeepAliveResponse` does not match the value sent with `MsgKeepAlive`.

state	agency
`StClient`	**Client**
`StServer`	**Server**

Figure 3.10: Keep-Alive state agencies

`MsgKeepAliveResponse` *cookie* Keep alive response message.

`MsgDone` Terminating message.

3.10.3 Size limits per state

These bounds limit how many bytes can be sent in a given state; indirectly, this limits the payload size of each message. If a space limit is violated, the connection SHOULD be torn down.

state	size limit in bytes
`StClient`	65535
`StServer`	65535

3.10.4 Timeouts per state

These limits bound how much time the receiver side can wait for the arrival of a message. If a timeout is violated, the connection SHOULD be torn down.

state	timeout
`StClient`	97s
`StServer`	60s

Table 3.12: timeouts per state

3.10.5 CDDL encoding specification

```
;
; KeepAlive Mini-Protocol
;

keepAliveMessage = msgKeepAlive
                 / msgKeepAliveResponse
                 / msgDone

msgKeepAlive         = [0, base.word16]
msgKeepAliveResponse = [1, base.word16]
msgDone              = [2]

;# import network.base as base
```

3.11 Peer Sharing mini-protocol

protocol haddocks: `Ouroboros.Network.Protocol.PeerSharing.Type`
codec haddocks: `Ouroboros.Network.Protocol.PeerSharing.Codec`
node-to-node mini-protocol number: `10`

3.11.1 Description

The Peer-Sharing mini-protocol is a simple Request-Reply mini-protocol. The mini-protocol is used by nodes to share their upstream peers (a subset of their Known Peers).

3.11.2 State machine

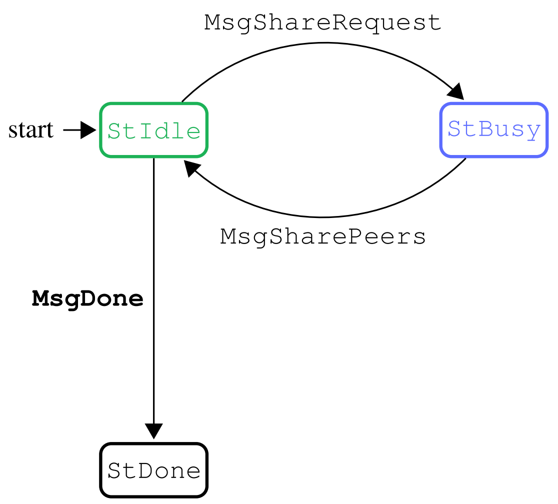


Figure 3.11: State machine of the peer sharing protocol.

state	agency
`StIdle`	**Client**
`StBusy`	**Server**

Figure 3.12: Peer-Sharing state agencies

Protocol messages

`MsgShareRequest` *amount* The client requests a maximum number of peers to be shared (*amount*). Ideally, this amount should limited by a protocol level constant to disallow a bad actor from requesting too many peers.

`MsgSharePeers` [*peerAddress*] The server replies with a set of peers. The amount of information send is limited by message size limit (see below).

It is a protocol error to send more peers than it was requested.

The server should only share peers with which it has (or recently had) an successful inbound or outbound session.

`MsgDone` Terminating message.

3.11.3 Size limits per state

These bounds limit how many bytes can be sent in a given state; indirectly, this limits the payload size of each message. If a space limit is violated, the connection SHOULD be torn down.

3.11.4 Timeouts per state

These limits bound how much time the receiver side can wait for the arrival of a message. If a timeout is violated, the connection SHOULD be torn down.

state	size limit in bytes
`StIdle`	`5760`
`StBusy`	`5760`

state	timeout
`StIdle`	`-`
`StBusy`	`60s`

Table 3.13: timeouts per state

3.11.5 Client Implementation Details

The initiator side will have to be running indefinitely since protocol termination means either an error or peer demotion. Because of this, the protocol won't be able to be run as a simple request-response protocol. To overcome this, the client-side implementation will use a registry so that each connected peer gets registered and assigned a controller with a request mailbox. This controller will be used to issue requests to the client implementation, which will be waiting for the queue to be filled up to send a `MsgShareRequest`. After sending a request, the result is put into a local result mailbox.

If a peer gets disconnected, it should get unregistered.

Deciding from whom to request peers (and how many)

First of all, peer-sharing requests should only be issued if:

- The current number of known peers is less than the target for known peers;
- The rate limit value for peer sharing requests isn't exceeded;
- There are available peers to issue requests to;

If these conditions hold, then we can pick a set of peers to issue requests to. Ideally, this set respects the rate limit value for peer-sharing requests.

If a peer has `PeerSharingDisabled`, flag value, do not ask it for peers. This peer won't even have the Peer-Sharing miniprotocol server running.

The number of peers to request from each upstream peer should aim to fulfil the target for known peers. This number should be split for the current peer target objective across all peer-sharing candidates for efficiency and diversity reasons.

Picking peers for the response

Apart from managing the Outbound Governor state correctly, the final result set should be a random distribution of the original set.

This selection should be done in such a way that when the same initial PRNG state is used, the selected set does not significantly vary with small perturbations in the set of published peers.

The intention of this selection method is that the selection should give approximately the same replies to the same peers over the course of multiple requests from the same peer. This is to deliberately slow the rate at which peers can discover and map out the entire network.

3.11.6 Server Implementation Details

As soon as the server receives a share request, it needs to pick a subset not bigger than the value specified in the request's parameter. The reply set needs to be sampled randomly from the Known Peer set according to the following constraints:

- Only pick peers that we managed to connect to at some point
- Don't pick known-to-be-ledger peers

- Pick peers that have public willingness information (e.g. `DoAdvertisePeer`).
- Pick peers that haven't behaved badly (e.g. `PeerFailCount == 0`)

Computing the result (i.e. random sampling of available peers) needs access to the `PeerSelectionState`, which is specific to the `peerSelectionGovernorLoop`. However, when initialising the server side of the mini-protocol, we have to provide the result computing function early on the consensus side. This means we will have to find a way to delay the function application all the way to diffusion and share the relevant parts of `PeerSelectionState` with this function via a `TVar`.

3.11.7 CDDL encoding specification (≥ 14)

```
;
; Peer Sharing MiniProtocol
;

peerSharingMessage = msgShareRequest
                   / msgSharePeers
                   / msgDone

msgShareRequest = [0, base.word8]
msgSharePeers   = [1, peerAddresses]
msgDone         = [2]

peerAddresses = [* peerAddress]

peerAddress = [0, base.word32, portNumber] ; ipv4 + portNumber
            / [1, base.word32, base.word32, base.word32, base.word32, portNumber] ; ipv6 + portNumbe

portNumber = base.word16

;# import network.base as base
```

3.12 Local Tx-Submission mini-protocol

protocol haddocks: `Ouroboros.Network.Protocol.LocalTxSubmission.Type`
codec haddocks: `Ouroboros.Network.Protocol.LocalTxSubmission.Codec`
node-to-client mini-protocol number: `6`

3.12.1 Description

The local transaction submission mini protocol is used by local clients, For example, wallets or CLI tools are used to submit transactions to a local node. The protocol is **not** used to forward transactions from one core node to another. The protocol for the transfer of transactions between full nodes is described in Section 3.9.

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-based protocol where the client creates a workload for the server. This is acceptable because this mini-protocol is only to be used between a node and a local client.

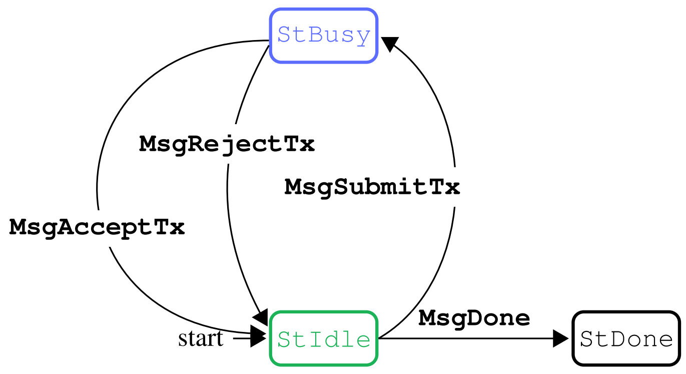


Figure 3.13: State machine of the Local Tx-Submission mini-protocol.

state	**agency**
`StIdle`	**Client**
`StBusy`	**Server**

Figure 3.14: Local Tx-Submission state agencies

3.12.2 State machine

Protocol messages

`MsgSubmitTx` (t) The client submits a single transaction. It MUST wait for a reply.

`MsgAcceptTx` The server confirms that it accepted the transaction.

`MsgRejectTx` $(reason)$ The server informs the client that it rejected the transaction and provides a $reason$.

`MsgDone` The client terminates the mini protocol.

3.12.3 Size limits per state

No size limits.

3.12.4 Timeouts per state

No timeouts.

3.12.5 CDDL encoding specification

```
;
; LocalTxSubmission mini-protocol
;


; Reference implementation of the codec in:
; ouroboros-network/src/Ouroboros/Network/Protocol/LocalTxSubmission/Codec.hs

localTxSubmissionMessage
    = msgSubmitTx
    / msgAcceptTx
    / msgRejectTx
    / ltMsgDone
```

```

msgSubmitTx = [0, base.tx ]
msgAcceptTx = [1]
msgRejectTx = [2, rejectReason ]
ltMsgDone   = [3]

rejectReason = int

;# import network.base as base
```

See appendix A for common definitions.

3.13 Local State Query mini-protocol

protocol haddocks: `Cardano.Network.Protocol.LocalStateQuery.Type`
codec haddocks: `Cardano.Network.Protocol.LocalStateQuery.Codec`
node-to-client mini-protocol number: 7

3.13.1 Description

Local State Query mini-protocol allows querying of the consensus/ledger state. This mini protocol is part of the node-to-client protocol; hence, it is only used by local (and thus trusted) clients. Possible queries depend on the era (Byron, Shelly, etc.) and are not specified in this document. The protocol specifies basic operations like acquiring/releasing the consensus/ledger state, which is done by the server, or running queries against the acquired ledger state.

3.13.2 State machine

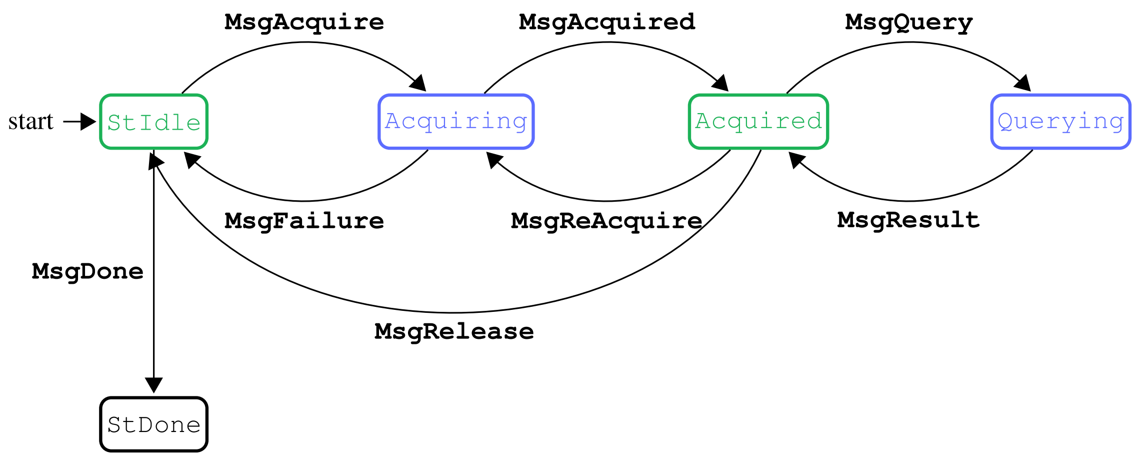


Figure 3.15: State machine of the Local State Query mini-protocol.

state	**agency**
`StIdle`	**Client**
`Acquiring`	**Server**
`Acquired`	**Client**
`Querying`	**Server**

Protocol messages See Figure 3.16, where $AcquireFailure$ is either:

- $AcquireFailurePointTooOld$, or
- $AcquireFailurePointNotOnChain$

$Target$ is either $ImmutableTip$, $VolatileTip$, or $SpecificPoint pt$.

The primary motivation for being able to acquire the $ImmutableTip$ is that it's the most recent ledger state that the node will never abandon: the node will never rollback to a prefix of that immutable chain (unless the on-disk ChainDB is corrupted/manipulated). Therefore, answers to queries against the $ImmutableTip$ is necessarily not subject to rollback.

`MsgAcquire` The client requests that the $Target$ ledger state on the server's be made available to query, and waits for confirmation or failure.

`MsgAcquired` The server can confirm that it has the state at the requested point.

`MsgFailure` The server can report that it cannot obtain the state for the requested point.

`MsgQuery` The client can perform queries on the current acquired state.

`MsgResult` The server must reply with the queries.

`MsgRelease` The client can instruct the server to release the state. This lets the server free resources.

`MsgReAcquire` This is like **`MsgAcquire`** but for when the client already has a state. By moving to another state directly without a **`MsgRelease`** it enables optimisations on the server side (e.g. moving to the state for the immediate next block).

Note that failure to re-acquire is equivalent to **`MsgRelease`**, rather than keeping the exiting acquired state.

`MsgDone` The client can terminate the protocol.

from state	**message**	**parameters**	**to state**
`StIdle`	**`MsgAcquire`**	$Target\ point$	`Acquiring`
`Acquiring`	**`MsgFailure`**	$AcquireFailure$	`StIdle`
`Acquiring`	**`MsgAcquired`**		`Acquired`
`Acquired`	**`MsgQuery`**	$query$	`Querying`
`Querying`	**`MsgResult`**	$result$	`Acquired`
`Acquired`	**`MsgReAcquire`**	$Target\ point$	`Acquiring`
`Acquired`	**`MsgRelease`**		`StIdle`
`StIdle`	**`MsgDone`**		`StDone`

Figure 3.16: Local State Query mini-protocol messages.

3.13.3 Size limits per state

No size limits.

3.13.4 Timeouts per state

No timeouts.

3.13.5 CDDL encoding specification

```
;
; LocalStateQuery mini-protocol.
;

localStateQueryMessage
  = msgAcquire
  / msgAcquired
  / msgFailure
  / msgQuery
  / msgResult
  / msgRelease
  / msgReAcquire
  / lsqMsgDone

acquireFailurePointTooOld     = 0
acquireFailurePointNotOnChain = 1

failure       = acquireFailurePointTooOld
              / acquireFailurePointNotOnChain

query         = any
result        = any

msgAcquire    = [0, base.point]
              / [8]
              / [10]
msgAcquired   = [1]
msgFailure    = [2, failure]
msgQuery      = [3, query]
msgResult     = [4, result]
msgRelease    = [5]
msgReAcquire  = [6, base.point]
              / [9]
              / [11]
lsqMsgDone    = [7]

;# import network.base as base
```

See appendix A for common definitions.

3.14 Local Tx-Monitor mini-protocol

protocol haddocks: `Cardano.Network.Protocol.LocalTxMonitor.Type`
codec haddocks: `Cardano.Network.Protocol.LocalTxMonitor.Codec`
node-to-client mini-protocol number: `9`

3.14.1 Description

A mini-protocol which allows the monitoring of transactions in the local mempool. This mini-protocol is stateful; the server side tracks transactions already sent to the client.

3.14.2 State machine

Protocol messages

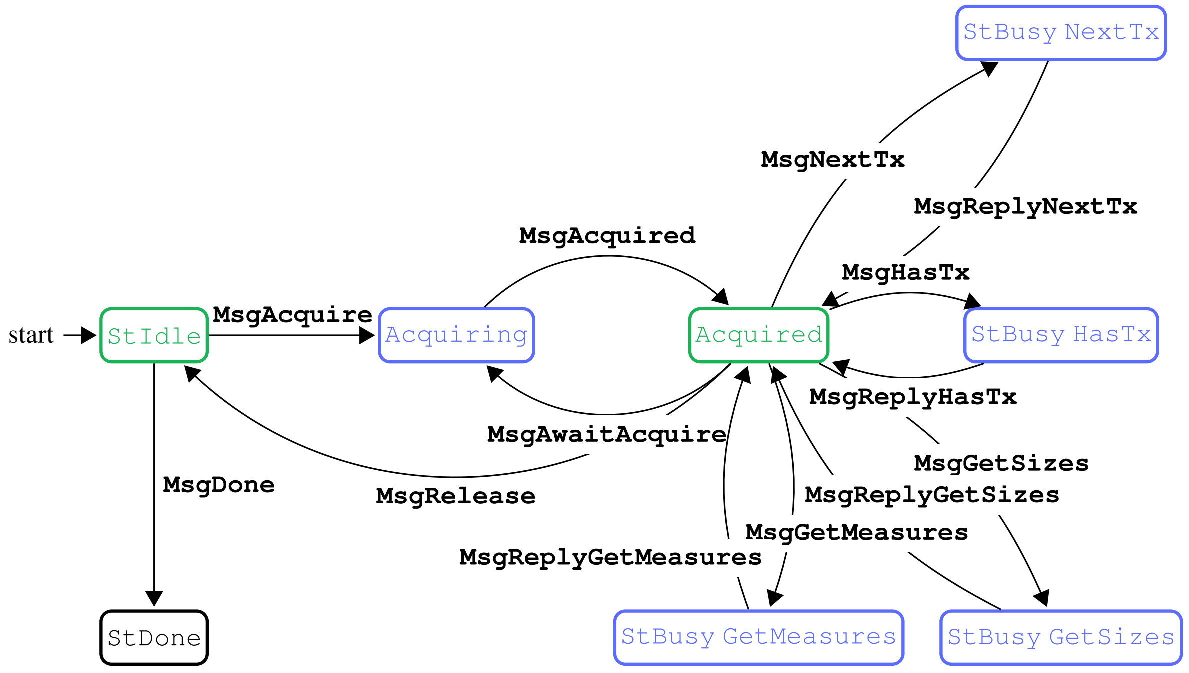


Figure 3.17: State machine of the Local Tx-Monitor mini-protocol.

state	**agency**
`StIdle`	**Client**
`Acquiring`	**Server**
`Acquired`	**Client**
`StBusy`	**Server**

Figure 3.18: Local Tx-Monitor state agencies

`MsgAcquire` Acquire the latest snapshot. This enables subsequent queries to be made against a consistent view of the mempool.

`MsgAcquired` (SlotNo) The server side is now locked to a particular mempool snapshot. It returns the slot number of the 'virtual block' under construction.

`MsgAwaitAcquire` Like 'MsgAcquire' but await a new snapshot different from the one currently acquired.

`MsgRelease` Release the acquired snapshot in order to loop back to the idle state.

`MsgNextTx` The client requests a single transaction and waits for a reply.

`MsgReplyNextTx` (Nothing | Just tx) The server responds with a single transaction if one is available in the mempool. This must be a transaction that was not previously sent to the client for this particular snapshot.

`MsgHasTx` The client checks whether the server knows of a particular transaction identified by its id.

`MsgReplyHasTx` (Bool) The server responds `True` when the given tx is present in the snapshot, `False` otherwise.

`MsgGetSizes` The client asks the server about the mempool current size and max capacity.

`MsgReplyGetSizes` (Word32,Word32,Word32) The server responds with three sizes. The meaning of them are:

capacity in bytes the maximum capacity of the mempool (note that this may dynamically change when the ledger state is updated);

size in bytes the summed byte size of all the transactions in the mempool;

number of transactions the number of transactions in the mempool.

`MsgGetMeasures` The client asks the server for information on the mempool's measures.

`MsgReplyGetMeasures` (Word32, Map Text (Integer, Integer)) The server responds with the total number of transactions currently in the mempool, and a map of the measures known to the mempool. The keys of this map are textual labels of the measure names, which should typically be considered stable for a given node version, and the values are a pair of integers representing the current size and maximum capacity respectively for that measure. The maximum capacity should not be considered fixed and is likely to change due to mempool conditions. The size should always be less than or equal to the capacity.

from state	message	parameters	to state
`StIdle`	**`MsgAcquire`**		`Acquiring`
`Acquiring`	**`MsgAcquired`**	SlotNo	`Acquired`
`Acquired`	**`MsgAwaitAcquire`**		`Acquiring`
`Acquired`	**`MsgRelease`**		`StIdle`
`Acquired`	**`MsgNextTx`**		`StBusy NextTx`
`StBusy NextTx`	**`MsgReplyNextTx`**	(**Nothing** | **Just** tx)	`Acquired`
`Acquired`	**`MsgHasTx`**		`StBusy HasTx`
`StBusy HasTx`	**`MsgReplyNextTx`**	Bool	`Acquired`
`Acquired`	**`MsgGetSizes`**		`StBusy GetSizes`
`StBusy GetSizes`	**`MsgReplyGetSizes`**	Word32,Word32,Word32	`Acquired`
`Acquired`	**`MsgGetMeasures`**		`StBusy GetMeasures`
`StBusy GetMeasures`	**`MsgReplyGetMeasures`**	Word32,Map Text (Integer,Integer)	`Acquired`
`StIdle`	**`MsgDone`**		`StDone`

Figure 3.19: Local Transaction Monitor mini-protocol messages.

3.14.3 Size limits per state

No size limits.

3.14.4 Timeouts per state

No timeouts.

3.14.5 CDDL encoding specification

```
;
; LocalTxMonitor mini-protocol.
;
; reference implementation of the codec in :
; ouroboros-network/src/Ouroboros/Network/Protocol/LocalTxMonitor/Codec.hs

localTxMonitorMessage
  = msgDone
  / msgAcquire
  / msgAcquired
  / msgNextTx
  / msgReplyNextTx
  / msgHasTx
  / msgReplyHasTx
  / msgGetSizes
```

```
    / msgReplyGetSizes
    / msgGetMeasures
    / msgReplyGetMeasures
    / msgRelease

msgDone              = [0]

msgAcquire           = [1]
msgAcquired          = [2, base.slotNo]

msgAwaitAcquire      = msgAcquire
msgRelease           = [3]
msgNextTx            = [5]
msgReplyNextTx       = [6] / [6, base.tx]
msgHasTx             = [7, base.txId]
msgReplyHasTx        = [8, bool]
msgGetSizes          = [9]
msgReplyGetSizes     = [10, [base.word32, base.word32, base.word32]]
msgGetMeasures       = [11]
msgReplyGetMeasures = [12, base.word32, {* text => [integer, integer]}]

;# import network.base as base
```

See appendix A for common definitions.

3.15 Pipelining of Mini Protocols

Protocol pipelining is a technique that improves the performance of some protocols. The underlying idea is that a client that 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-protocols 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 and does not require any modifications to 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 **`MsgRequestRange`** messages to the server, the data stream from the client to the server will only consist of **`MsgRequestRange`** messages (and a final **`MsgClientDone`** message) and no other message types. The server can simply follow the state machine of the protocol and process the messages in turn, regardless of whether the client uses pipelining or not. The MUX/DEMUX layer (Chapter 2) guarantees that messages of the same mini protocol are delivered in transmission order. 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 2.1.3). The protocol requires that a client must never cause an 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).

3.16 Node-to-node protocol

haddock documentation: `Ouroboros.Network.NodeToNode`
haddock documentation: `Ouroboros.Network.NodeToNode.Version`

The *node-to-node protocol* consists of the following protocols:

- *chain-sync mini-protocol* for headers (section 3.7)
- *block-fetch mini-protocol* (section 3.8)
- *tx-submission mini-protocol*; from `NodeToNodeV_6` the version 2 is used (section 3.9)
- *keep alive mini-protocol*; from `NodeToNodeV_3` (section 3.10)

- *peer-sharing mini-protocol*; from `NodeToNodeV_11` (section 3.11)

Currently supported versions of the *node-to-node protocol* are listed in table 3.20.

version	description
`NodeToNodeV_14`	No changes, identifies Plomin HF nodes mandatory on mainnet as of 2025.01.29
`NodeToNodeV_15`	No changes, identifies nodes which support SRV records

Figure 3.20: Node-to-node protocol versions

Previously supported node-to-node versions are listed in table B.1.

3.16.1 Node-to-node mux mini-protocol numbers

The following table 3.14 shows mux mini-protocol numbers assigned to each node-to-node mini-protocol.

mini-protocol	mini-protocol number
Handshake	0
Chain-Sync	2
Block-Fetch	3
Tx-Submission	4
Keep-Alive	8
Peer-Sharing (optional)	10

Table 3.14: Node-to-node protocol numbers

3.16.2 Node-to-node mux ingress buffer size limits

Ingress buffer is the buffer which holds received data for a given mini-protocol. It is an internal detail of the multiplexer. Each implementation should define its ingress buffer size limits. Here we specify the default choices we made for Cardano Node. These limits depend on how much pipelining depth a given mini-protocol can do. This is an internal implementation detail since the amount of pipelining is controlled by the peer who owns its ingress buffer.

mini-protocol	ingress size limit in bytes
Handshake	-
Chain-Sync	462 000
Block-Fetch	230 686 940
Tx-Submission	721 424
Keep-Alive	1 408
Peer-Sharing	5 760

Table 3.15: Mux ingress buffer sizes for each mini-protocol

3.17 Node-to-client protocol

haddock documentation: `Ouroboros.Network.NodeToClient`
haddock documentation: `Ouroboros.Network.NodeToClient.Version`

The *node-to-client protocol* consists of the following protocols:

- *chain-sync mini-protocol* for blocks (section 3.7)

- *local-tx-submission mini-protocol* (section 3.12)
- *local-state-query mini-protocol*; from version `NodeToClientV_2` (section 3.13)
- *local tx-monitor mini-protocol*; from version `NodeToClientV_12` (section 3.14)

Supported versions of *node-to-client protocol* are listed in table 3.21.

version	**description**
`NodeToClientV_16`	Conway era, `ImmutableTip` and `GetStakeDelegDeposits` queries
`NodeToClientV_17`	`GetProposals`, `GetRatifyState` queries
`NodeToClientV_18`	`GetFuturePParams` query
`NodeToClientV_19`	`GetBigLedgerPeerSnapshot` query
`NodeToClientV_20`	`QueryStakePoolDefaultVote` query;
	added `MsgGetMeasures` and `MsgReplyGetMeasures` queries
`NodeToClientV_21`	new codecs for `PParams` and `CompactGenesis`

Figure 3.21: Node-to-client protocol versions

Previously supported node-to-client versions are listed in table B.2.

3.17.1 Node-to-client mux mini-protocol numbers

The following table 3.16 show mux mini-protocol numbers assigned to each node-to-client mini-protocol.

mini-protocol	**mini-protocol number**
Handshake	0
Chain-Sync	5
Local Tx-Submission	6
Local State Query	7
Local Tx-Monitor	9

Table 3.16: Node-to-client protocol numbers

3.17.2 Node-to-client mux ingress buffer size limits

All *node-to-client protocols* are using very large ingress buffer size limits of 4 294 967 295 bytes, effectively there are no size limits.

Chapter 4

Time and size limits

4.1 Timeouts

There are several layers, where timeouts play a crucial way in making the system secure. At the lowest layer is a mux timeout which we explain next. After establishing a connection (either a Node-to-Node or Node-to-Client one), the handshake is using a bearer with `10s` timeout on receiving each mux SDU. Note, this is a timeout which bounds how long it takes to receive a single mux SDU, e.g. from receiving the leading edge of the mux SDU until receiving its trailing edge, not how long we wait to receive a next SDU. Handshake protocol is then imposing its own timeouts, see table 3.3.

After handshake negotiation is done, mux is using a bearer with `30s` timeout on receiving a mux SDU (the previous note applies as well). Once a mini-protocol is in execution it must enforce it's own set of timeouts which we included in the previous chapter and for convenience we referenced them in the table 4.1 below.

mini-protocol	**timeouts**
Handshake	table 3.3
Chain-Sync	table 3.6
Block-Fetch	table 3.8
Tx-Submission	table 3.11
Keep-Alive	table 3.12
Peer-Share	table 3.13

Figure 4.1: Node-To-Node mini-protocol timeouts

On the inbound side of the Node-to-Node protocol, we also include a `5s` idleness timeout. It starts either when a connection is accepted or when all responder mini-protocols terminated. If this timeout expires, without receiving any message from a remote end, the connection must be closed unless it is a duplex connection which is used by the outbound side.

Once all outbound and inbound mini-protocols have terminated and the idleness timeout expired, the connection is reset and put on a `60s` timeout. See section 5.8.6 why this timeout is required.

4.2 Space limits

All per mini-protocol size limits are referenced in table 4.2:

mini-protocol	**space limits**
Handshake	table 3.6.3
Chain-Sync	table 3.5
Block-Fetch	table 3.8.3
Tx-Submission	table 3.10
Keep-Alive	table 3.10.3
Peer-Share	table 3.11.3

Figure 4.2: Node-To-Node mini-protocol size limits

Chapter 5

Connection Manager State Machine Specification

5.1 Introduction

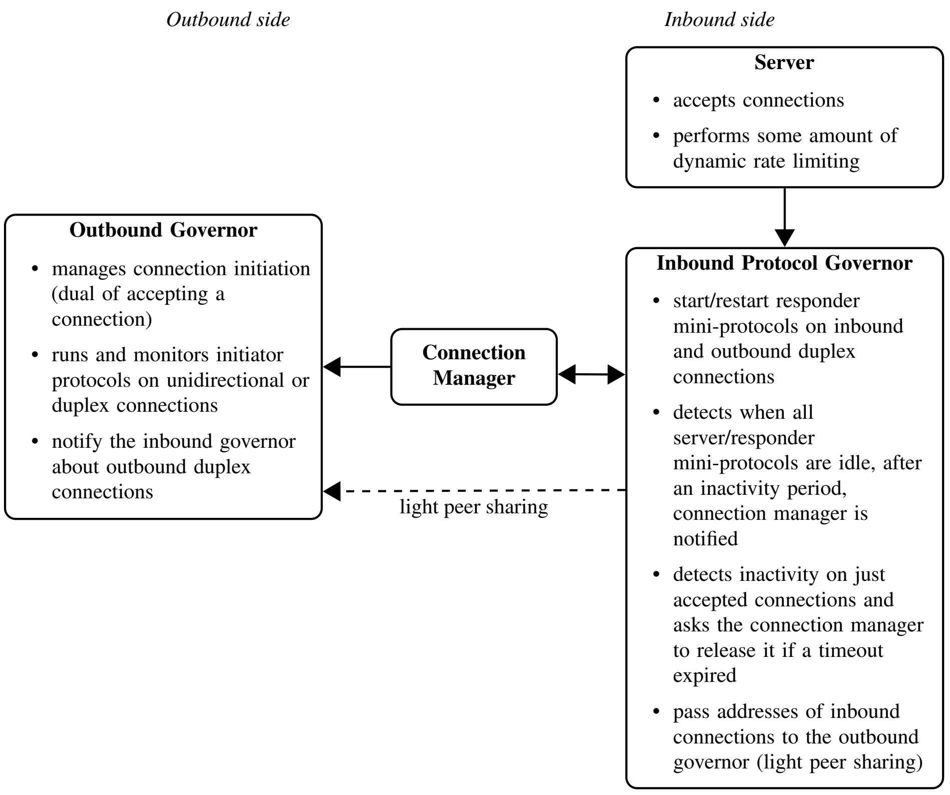
As described in the Network Design document, the goal is to transition to a more decentralised network. To make that happen, a plan was designed to come up with a P2P network that is capable of achieving desired network properties. One key component of such design is the *p2p governor*, which is responsible for managing the *cold*/*warm*/*hot* peer selection, managing the churn of these groups, and adjusting the targets in order for the network to reach the desired properties. However, having *warm* and *hot* peers implies establishing a bearer connection; *hot* peers need to run several mini-protocols, and each mini-protocol runs two instances (client and server). This means that with a large enough warm/hot peer target, there's going to be a lot of resource waste when it comes to file descriptor usage. There's also the problem of firewalls, where it matters who tries to start a communication with whom (if it's the client or the server).

Knowing this, it would be good to make the most of each connection and, in order to do so, the *Connection manager* was designed.

5.2 Components

Figure 5.1 illustrates the three main components of the decentralisation process from the perspective of a local node. In the `Outbound` side, the *p2p governor*, as said previously, takes care of all connection initiation (outbound connections) and decides which mini-protocols to run (*established*, *warm* or *hot*). In the `Inbound` side, the `Server` is just a simple loop, responsible for accepting incoming connections; and the `Inbound Protocol Governor` role is starting/restarting the required mini-protocols, to detect if its local peer was added as a *warm*/*hot* peer in some other remote node and to set timers in some cases, e.g. if the remote end opened a connection and did not send any message; the `Inbound Protocol Governor` will timeout after some time and close the connection. The arrows in Figure 5.1 represent dependencies between components: The server accepts a connection, which is then given to *Connection manager*. *Connection manager* exposes methods to update its state whenever the `Inbound Protocol Governor` notices that the connection was used (could be used due to *warm*
hot transitions). If peer sharing is enabled, the incoming address will eventually be added to the known set of the outbound governor.

Using a TCP connection in both directions rather than two independent TCP connections is suitable for efficient use of network resources, but more importantly, it is crucial to support certain essential scenarios where one node is behind a firewall that blocks incoming TCP connections. For example, it is good practice to have a block-producing node behind a firewall while deploying relay nodes outside of it. If the node behind the firewall can establish an outbound TCP connection to its relays but still has those relays select the block-producing node as an upstream peer, which means that node operators do not need to configure any holes and/or port forwarding in the firewall. If we were only to support running mini-protocols in one direction, then this scenario would require a hole in the firewall to allow the relays to establish incoming connections to the block-producing node. That would be both less secure and also require additional configuration.



Outbound side
Inbound side
Server
• accepts connections
• performs some amount of dynamic rate limiting
Outbound Governor
• manages connection initiation (dual of accepting a connection)
• runs and monitors initiator protocols on unidirectional or duplex connections
• notify the inbound governor about outbound duplex connections
Connection Manager
Inbound Protocol Governor
• start/restart responder mini-protocols on inbound and outbound duplex connections
• detects when all server/responder mini-protocols are idle, after an inactivity period, connection manager is notified
• detects inactivity on just accepted connections and asks the connection manager to release it if a timeout expired
• pass addresses of inbound connections to the outbound governor (light peer sharing)
light peer sharing

Figure 5.1: Main components

Consider, however, what is required to make this scenario work.

1. We must start with an outbound connection being established from the block-producing node to a relay.
2. The block-producing node wants the relay as an upstream peer – to receive blocks from the rest of the network – so the normal mini-protocols need to be run with the block-producing node in the client role and relay in the server role. So initially, at least, the relay had to act as a server to accept the connection and to run the server side of the mini-protocols.
3. Next, however, we want the relay to be able to select the block-producing node as an upstream peer, and we want it to do so by reusing the existing connection since we know the firewall makes it impossible to establish a new outbound connection to the block-producing node. Thus, we must be able to have the relay start the client side of the usual mini-protocols and The block-producer must be running on their server side.
4. So, notice that this means we have started with just running the mini-protocols in one direction and transitioned to running them in both directions, what we call full *duplex*.
5. Furthermore, such transitions are not a one-off event. It is entirely possible for a node to select another peer as an upstream peer and later change its mind. This means we could transition from duplex back to unidirectional – and that unidirectional direction need not even be the same as the initial direction!

This leads to a couple of observations:

1. that, in the general case, we need to support any number of transitions between unidirectional and duplex use of a connection and
2. that once a bearer has been established, the relationship between the two ends is symmetric: the original direction hardly matters.

A consequence of all this is that we cannot use a classic client/server design. We are decoupling the ongoing role of the connection from who initiated it. That is, we cannot just run a server component that manages all the connections and threads for the server (inbound) side of things and a separate component that manages the connections and threads for the client (outbound) side of things. The connections have to be a shared resource between the inbound and outbound sides so that we can use connections in either or both directions over the lifetime of a connection.

Although actual TCP connections must be a shared resource, we do not wish to Intermingle the code to handle the inbound and outbound directions. As noted above, the selection of upstream (outbound) peers is quite complicated, and we would not want to add to that complexity by mixing it with a lot of other concerns, and vice versa. To minimise complexity, it would be preferable if the code that manages the outbound side would be completely unaware of the inbound side and vice versa. Yet, we still want the inbound and outbound sides to opportunistically share TCP connections where possible. This appears to be eminently achievable given that we are using multiplexing to run mini-protocols in either direction and concurrency for mini-protocol handlers to achieve a degree of modularity.

The use of a single TCP connection helps simplify exception processing and mitigate poor peer performance in a timely manner (whether connection-related or otherwise). This is covered in more detail in Section 5.3.

These ideas lead to the design illustrated in Figure 5.1. In this design, there is an outbound and inbound side – which are completely unaware of each other – mediated by a shared *connection manager* component.

The connection manager is there to manage the underlying TCP connection resources. It has to provide an interface to the outbound side to enable the use of connections in an outbound direction. Correspondingly, it must provide an interface to the inbound side to enable the use of connections in an inbound direction. Internally, it must deal with connections being used in a unidirectional or duplex way, as well as the transitions between them. Of course, it can be the case that connections are no longer required in either direction, and such connections should be closed in an orderly manner. This must be the responsibility of the connection manager since it is the only component that can see both inbound and outbound sides to be able to see that a connection is no longer needed in either direction and, hence, not needed at all.

In the next couple of sections, we will review the inbound and outbound sides need to be able to do, and what service does the connection manager need to provide?

5.3 Exception Processing

We maintain a one-to-one correspondence between peers and connections, which simplifies exception handling since if there's a single mini-protocol violation, we need to shut down the thread that handles that particular connection. Although multiple threads handle a single connection: two threads per a pipelined mini-protocol, one thread per a non-pipelined one, plus two multiplexer threads (muxer & demuxer threads). However, all these threads are spawned and managed by the multiplexer, which has the property that if any of the threads throws an exception, all of the threads will be killed. This property allows us to have a single error handling policy (called `RethrowPolicy`) per connection handler thread. A `RethrowPolicy` classifies exceptions into two categories, depending on whether an exception should terminate the connection or be propagated to terminate the whole process. `RethrowPolicy`-ies can be composed in terms of a semi-group. Network code only makes `IOManagerErrors` fatal. On top of that, consensus introduces its own `consensusRethrowPolicy` for the Node-To-Node protocol.

5.4 Mini-protocol return values

Handling of mini-protocol return values is a complementary feature to exception processing, hence it's described here, although it is done at the Outbound-Governor level rather than Connection-Manager level, which is primarily described in this part of the documentation.

We classify mini-protocol return values for initiator/client mini-protocols (this feature is only needed for the *chain-sync mini-protocol*). For a given return value, we compute the re-promotion delay used by the Outbound-Governor. Here is the `returnPolicy`. introduced in Ouroboros-Consensus for the Node-To-Node protocol. Cardano-Node is not managing outbound node-to-client connection; hence, a policy for the node-to-client protocol is not needed.

The outbound governor is also given a policy which controls how long to wait until re-promote a peer after an exception (for now, we use a fixed delay).

5.5 Outbound side: the outbound governor

A key component of the design for decentralisation is the outbound governor. It is responsible for:

- managing the selection of upstream peers;
- managing the transitions of upstream peers between cold/warm/hot states;
- continuously making progress towards the target number of peers in each state; and
- adjusting these targets over time to achieve a degree of 'churn'.

Taken together, and with appropriate policies, a network of nodes should be able to self-organise and achieve the desired properties. We have simulation results that give us a good degree of confidence that this is indeed the case at a large scale.

Fortunately, while the outbound governor's decision-making procedures are relatively complex, the use of connections is quite simple. The governor needs only two interactions.

Acquire a connection. The governor decides when to promote a peer from cold to warm. To perform the promotion, it needs to acquire access to a connection – either fresh or pre-existing. To complete the promotion, the client side of warm mini-protocols will be started.

Release a connection. The governor also decides when to demote a peer to cold. As part of the demotion, the client-side mini-protocols are terminated. The connection is then no longer needed by the governor and is released.

It is worth noting again that the outbound governor does not require exclusive access to the TCP bearer. It has no special TCP-specific needs during setup or shutdown. It needs access to the multiplexer to be able to run a set of mini-protocols in one direction. So, in a sense, it needs exclusive access to 'half' of a multiplexer for a connection, but it does not need to coordinate with or even be aware of any use of the other 'half' of the multiplexer. It is this separation of concerns that enables a modular design and implementation.

5.6 Inbound side: the server

The inbound side has a less complex task than the outbound governor, but its interactions with the connection manager are slightly more complicated.

The inbound side is split into two components: the server and the inbound governor.

The server is responsible for accepting new TCP connections on the listening socket. It is responsible for not exceeding resource limits by accepting too many new connections. It is also responsible for a little bit of DoS protection: limiting the rate of accepting new connections.

The server component is much simpler than in most network server applications because it does not need to manage the connection resources once created. The server hands new connections over to the connection manager as soon as they are accepted. The server's responsibilities end there. The server needs only two interactions with the connection manager.

Query number of connections The server component needs to query the connection manager to find the current number of connections. It uses this information to decide if any new connections can be accepted or if we are at the resource limits. Below the hard limits, the current number can be used as part of rate-limiting decisions.

Hand over a new connection Once the server component has successfully accepted a new connection, it needs to hand over responsibility for it to the connection manager.

5.7 Inbound side: the inbound governor

The inbound governor is responsible for starting, restarting and monitoring the the server side of the mini-protocols.

One of the high-level design choices is that when a server-side mini-protocol terminates cleanly (usually because the client chose to terminate it), then the the server side of the mini-protocol should be restarted in its initial state in case the client wishes to use the protocol again later. It is the inbound governor that is responsible for doing this.

The mux component provides a mechanism to start mini-protocol handlers on a connection for a specific mini-protocol number in a particular direction. These handlers can then be monitored to see when they terminate. The inbound governor relies on this mechanism to monitor when the protocol handler terminates cleanly. When it does terminate cleanly, the governor restarts the mini-protocol handler.

All the mini-protocols have the property that agency starts with the client/initiator side[1]. This allows all of the server/responder side protocols to be started in the mux 'on-demand' mode. In the on-demand mode, the protocol handler thread is not started until the client's first message arrives.

The inbound governor gets informed of new connections that should be monitored either via the server or by the connection manager. The server informs the governor about fresh inbound connections. The connection manager informs the governor about connections that started due to a request for an outbound connection – at least for those connections that are to be available to use in duplex mode.

As illustrated in Figure 5.1, both the connection manager and server components communicate with the inbound governor directly. They do this to inform the inbound governor about new connections so that it can start to run and monitor the server-side protocols. The server notifies about new connections established inbound, while the connection manager acquires new connections established outbound (at least the duplex ones) through the connection manager API. A slight simplification would be to have only one of these routes of notification.

The inbound governor

One simple illustration of how these three components interact together:

- Server accepts a connection;
- Server registers that connection to the connection manager (which puts the connection in `UnnegotiatedState Inbound`);
- Assuming the handshake was successful, the connection is put in `InboundIdleState`$^{\tau}$ `Duplex`;
- The remote end transitions the local node to warm (using the connection) within the expected timeout;
- IPG (Inbound Protocol Governor) notifies the *Connection manager* about this state change, via `promotedToWarmRemote`. Now the connection is in `InboundState Duplex`;
- *Connection manager* is asked for an outbound connection to that peer (by the *p2p governor*), it notices that it already has a connection with that peer in `InboundState Duplex`, so it gives that connection to *p2p governor* and updates its state to `DuplexState`.

You can find more information about the possible different connection states in the section 5.8.3.

5.8 Connection Manager

5.8.1 Overview

Connection manager is a lower-level component responsible for managing connections and its resources. Its responsibilities consist of:

- Tracking each connection, in order to keep an eye on the bounded resources;
- Starting new connections, negotiating if the connection should be *full-duplex* or *half-duplex*, through the *Connection Handler*;
- Be aware of *warm/hot* transitions, in order to try and reuse already established connections;
- Negotiating which direction, which mini-protocol is going to run (Client $\rightarrow$ Server, Server$\rightarrow$Client, or both);
- Taking care of a particularity of TCP connection termination (lingering connections).

[1] Originally transaction submission protocol had agency start with the responder/server side. A later protocol update reversed the initial agency so that they are now all consistent.

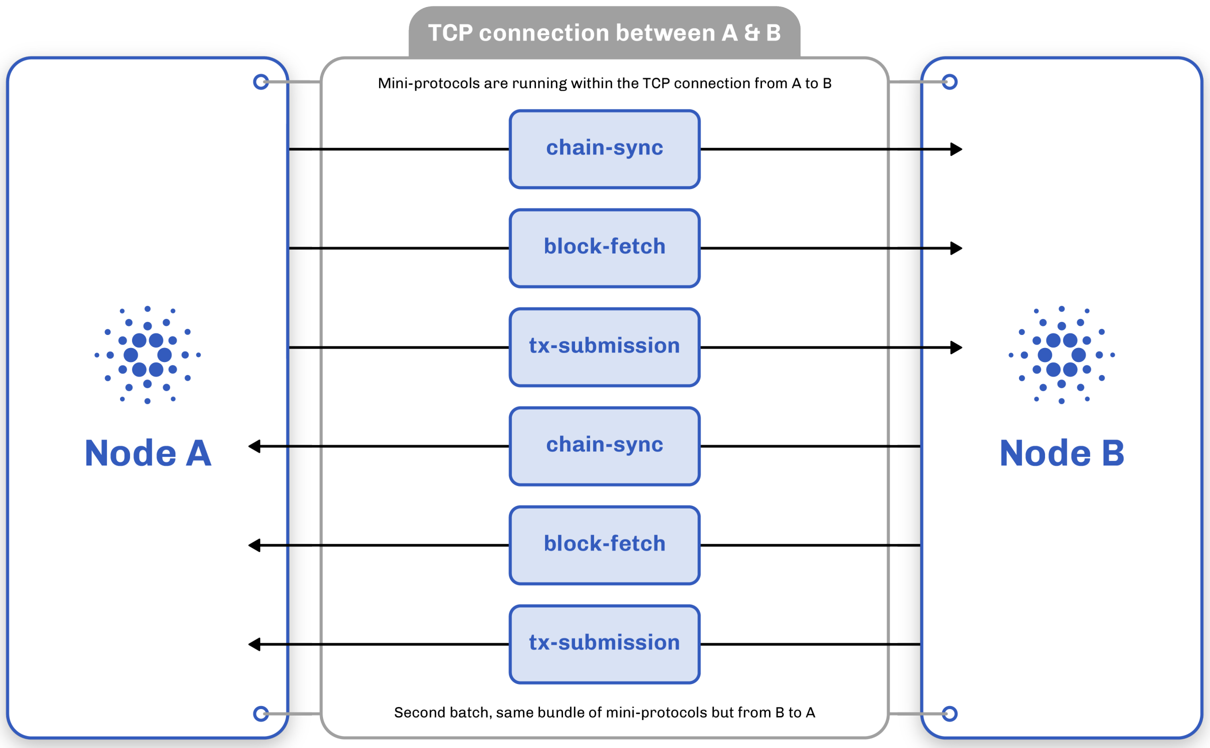

TCP connection between A & B
Mini-protocols are running within the TCP connection from A to B
chain-sync
block-fetch
tx-submission
Node A
Node B
chain-sync
block-fetch
tx-submission
Second batch, same bundle of mini-protocols but from B to A

Figure 5.2: Duplex connection running several mini-protocols

The *Connection manager* creates and records accepted connections and keeps track of their state as negotiations for the connection and start/stop mini-protocols are made. There's an *internal state machine* that helps the *Connection manager* keep track of the state of each connection, and help it make decisions when it comes to resource management and connection reusing.

The *Connection Handler* drives through handshake negotiation and starts the multiplexer. The the outcome of the handshake negotiation is:

- the negotiated version of the protocol
- negotiated parameters, which include the mode in which the connection will be run (`InitiatorOnlyMode`, `ResponderOnlyMode`, `InitiatorAndResponderMode` - the first two are *half-duplex*, the last one is *full-duplex* mode)
- Handshake might error

The *Connection Handler* notifies the *Connection manager* about the result of a negotiation, which triggers a state transition. If we can run the connection in full-duplex mode, then it is possible to run the bundles of mini-protocols in both directions and otherwise only in one direction. So, Figure 5.2 shows 6 mini protocols running, 3 in each direction. If we negotiated only a unidirectional connection, then we'd only be running 3 (The direction is based on which peer established the connection).

From the point of view of the *connection manager*, it only matters whether an *unidirectional* or *duplex* connection was negotiated. Unidirectional connections are the ones that run exclusively on either the initiator or responder side of mini-protocols, while duplex connections can run either or both initiator and responder protocols. Note that in the outbound direction (initiator side), it is the *p2p governor* responsibility to decide which set of mini-protocols: *established*, *warm* or *hot*, are running. On the inbound side (responder mini-protocols), we have no choice but to run all of them.

The *connection manager* should only be run in two `MuxModes`:

- `ResponderMode` or
- `InitiatorAndResponderMode`

, the `InitiatorMode` is not allowed, since that mode is reserved for special leaf nodes in the network (such as the blockchain explorer, for example), and it doesn't make sense to run a node-to-client client side.

The duplex mode: `InitiatorAndResponderMode` is useful for managing connection with external nodes (*node-to-node protocol*), while `ResponderMode` is useful for running a server which responds to local connections (server side of *node-to-client protocol*).

Connection manager can use at most one ipv4 and at most one ipv6 address. It will bind to the correct address depending on the remote address type (ipv4/ipv6).

In this specification, we will often need to speak about two nodes communicating via a TCP connection. We will often call them local and remote ends of the connection or local / remote nodes; we will usually take the perspective of the local node.

5.8.2 Types

Connection manager exposes two methods to register a connection:

```
data Connected peerAddr handle handleError
  -- | We are connected, and mux is running.
  = Connected    !(ConnectionId peerAddr) !handle

  -- | There was an error during the handshake negotiation.
  | Disconnected !(ConnectionId peerAddr) !(Maybe handleError)

-- | Include the outbound connection in 'ConnectionManager'.

--  This executes:
--
-- * \(Reserve\) to \(Negotiated^{*}_{Outbound}\) transitions
-- * \(PromotedToWarm^{Duplex}_{Local}\) transition
-- * \(Awake^{Duplex}_{Local}\) transition
requestOutboundConnection
  :: HasInitiator muxMode ~ True
  ⇒ ConnectionManager muxMode socket peerAddr handle handleError m
  → peerAddr → m (Connected peerAddr handle handleError)

-- | Include an inbound connection into 'ConnectionManager'.

--  This executes:
--
-- * \(Accepted\) ∨ \(Overwritten\) to \(Negotiated^{*}_{Inbound}\) transitions
includeInboundConnection
  :: HasResponder muxMode ~ True
  ⇒ ConnectionManager muxMode socket peerAddr handle handleError m
  → socket → peerAddr → m (Connected peerAddr handle handleError)
```

The first one asks the *connection manager* to either connect to an outbound peer or, if possible, reuse a duplex connection. The other one allows registering an inbound connection, which was `accepted`. Both methods block operations and return either an error (handshake negotiation error or a multiplexer error) or a handle to a *negotiated* connection.

Other methods which are discussed in this specification:

```
-- | Custom Either type for the result of various methods.
data OperationResult a
    = UnsupportedState !InState
    | OperationSuccess a
```

```
-- | Enumeration of states, used for reporting; constructors elided from this
-- specification.
data InState

-- | Unregister an outbound connection.
--
-- This executes:
--
-- * \(DemotedToCold^{*}_{Local}\) transitions
unregisterOutboundConnection
  :: HasInitiator muxMode ~ True
  ⇒ ConnectionManager muxMode socket peerAddr handle handleError m
  → peerAddr → m (OperationResult ())

-- | Notify the 'ConnectionManager' that a remote end promoted us to a
-- /warm peer/.
--
-- This executes:
--
-- * \(PromotedToWarm^{Duplex}_{Remote}\) transition,
-- * \(Awake^{*}_{Remote}\) transition.
promotedToWarmRemote
  :: HasInitiator muxMode ~ True
  ⇒ ConnectionManager muxMode socket peerAddr handle handleError m
  → peerAddr → m (OperationResult InState)

-- | Notify the 'ConnectionManager' that a remote end demoted us to a /cold
-- peer/.
--
-- This executes:
--
-- * \(DemotedToCold^{*}_{Remote}\) transition.
demotedToColdRemote
  :: HasResponder muxMode ~ True
  ⇒ ConnectionManager muxMode socket peerAddr handle handleError m
  → peerAddr -> m (OperationResult InState)

-- | Unregister outbound connection. Returns if the operation was successful.
--
-- This executes:
--
-- * \(Commit*{*}\) transition
-- * \(TimeoutExpired\) transition
unregisterInboundConnection
  :: HasResponder muxMode ~ True
  ⇒ ConnectionManager muxMode socket peerAddr handle handleError m
  → peerAddr → m (OperationResult DemotedToColdRemoteTr)

-- | Number of connections tracked by the server.
numberOfConnections
  :: HasResponder muxMode ~ True
  ⇒ ConnectionManager muxMode socket peerAddr handle handleError m
  → STM m Int
```

5.8.3 Connection states

Each connection is either initiated by `Inbound` or `Outbound` side.

```
data Provenance
```

```
  = Inbound
  | Outbound
```

Each connection negotiates `dataFlow`:

```
data DataFlow
  = Unidirectional
  | Duplex
```

In `Unidirectional` data flow, the connection is only used in one direction: The outbound side runs the initiator side of mini-protocols, and the inbound side runs responders; in `Duplex` mode, both the inbound and outbound side runs the initiator and responder side of each mini-protocol. Negotiation of `DataFlow` is done by the handshake protocol; the final result depends on two factors: the negotiated version and `InitiatorOnly` flag, which is announced through a handshake. Each connection can be in one of the following states:

```
data ConnectionState
  -- The connection manager is about to connect with a peer.
  = ReservedOutboundState

  -- Connected to a peer, handshake negotiation is ongoing.
  | UnnegotiatedState Provenance

  -- Outbound connection, inbound idle timeout is ticking .
  | OutboundState^τ DataFlow

  -- Outbound connection, inbound idle timeout expired.
  | OutboundState DataFlow

  -- Inbound connection, but not yet used.
  | InboundIdleState^τ DataFlow

  -- Active inbound connection.
  | InboundState DataFlow

  -- Connection runs in duplex mode: either outbound connection negotiated
  -- 'Duplex' data flow, or 'InboundState Duplex' was reused.
  | DuplexState

  -- Connection manager is about to close ( reset ) the connection, before it
  -- will do that it will put the connection in 'OutboundIdleState' and start
  -- a timeout.
  | OutboundIdleState^τ

  -- Connection has terminated; socket is closed, thread running the
  -- the connection is killed. For some delay (`TIME_WAIT`) the connection is kept
  -- in this state until the kernel releases all the resources.
  | TerminatingState

  -- Connection is forgotten .
  | TerminatedState
```

The above type is a simplified version of what is implemented. The real implementation tracks more detail, e.g. connection id (the quadruple of IP addresses and ports), multiplexer handle, thread id, etc., which we do not need to take care of in this specification. The rule of thumb is that all states that have some kind of timeout should be annotated with a τ. In these cases, we are waiting for any message that would indicate a *warm* or *hot* transition. If that does not happen within a timeout, we will close the connection.

In this specification, we represent `OutboundState`$^\tau$ `Unidirectional`, which is not used, the implementation avoids this constructor, for the same reasons that were given above, regarding `InitiatorMode`.

Figure 5.3 shows all the transitions between `ConnectionStates`. Blue and Violet states represent states of an *Outbound* connection, and Green and Violet states represent states of an *Inbound* connection. Dashed arrows indicate asynchronous transitions that are triggered, either by a remote node or by the connection manager itself.

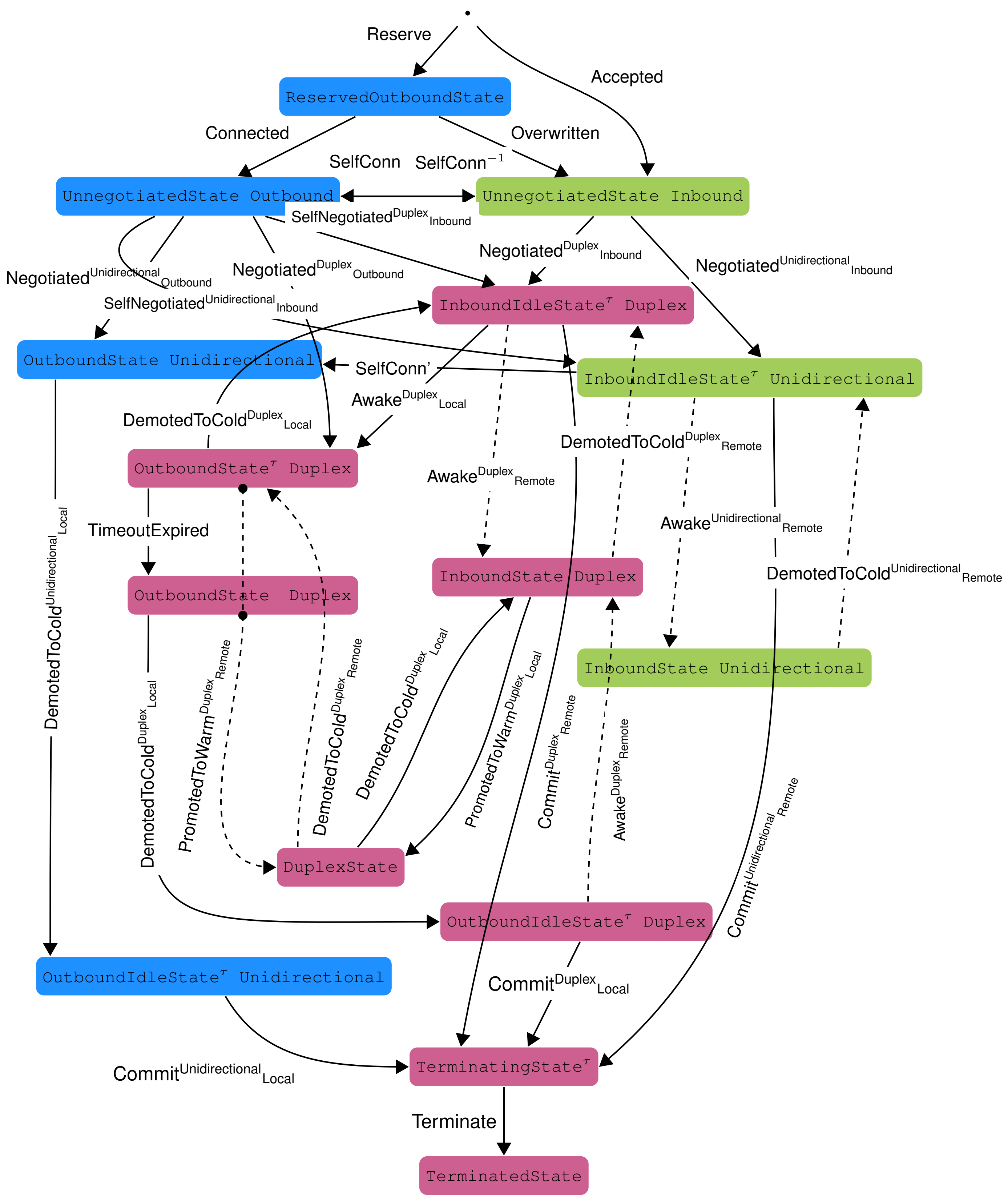

Reserve
Accepted
ReservedOutboundState
Connected
Overwritten
SelfConn
SelfConn⁻¹
UnnegotiatedState Outbound
UnnegotiatedState Inbound
SelfNegotiated Duplex Inbound
Negotiated Duplex Inbound
Negotiated Unidirectional Inbound
Negotiated Unidirectional Outbound
Negotiated Duplex Outbound
SelfNegotiated Unidirectional Inbound
InboundIdleState τ Duplex
OutboundState Unidirectional
SelfConn'
InboundIdleState τ Unidirectional
Awake Duplex Local
DemotedToCold Duplex Local
OutboundState τ Duplex
DemotedToCold Duplex Remote
Awake Duplex Remote
Awake Unidirectional Remote
TimeoutExpired
InboundState Duplex
DemotedToCold Unidirectional Remote
OutboundState Duplex
InboundState Unidirectional
DemotedToCold Unidirectional Local
DemotedToCold Duplex Local
PromotedToWarm Duplex Remote
DemotedToCold Duplex Remote
DemotedToCold Duplex Local
PromotedToWarm Duplex Local
Commit Duplex Remote
Awake Duplex Remote
Commit Unidirectional Remote
DuplexState
OutboundIdleState τ Duplex
OutboundIdleState τ Unidirectional
Commit Duplex Local
Commit Unidirectional Local
TerminatingState τ
Terminate
TerminatedState

Figure 5.3: *Outbound* (blue & violet) and *inbound* (green & violet) connection states and allowed transitions.

Note that the vertical symmetry in the graph corresponds to the local vs remote state of the connection, see table 5.1. The symmetry is only broken by `InboundIdleState`$^{\tau}$ `dataFlow`, which does not have a corresponding local equivalent. This is simply because, locally, we immediately know when we will start initiator protocols, and the implementation is supposed to do that promptly. This, however, cannot be assumed to be the case on the inbound side.

local connection state	*remote connection state*
`UnegotiatedState Outbound`	`UnegotiatedState Inbound`
`OutboundIdleState`$^{\tau}$ `dataFlow`	`InboundIdleState`$^{\tau}$ `dataFlow`
`OutboundState dataFlow`	`InboundState dataFlow`
`OutboundState`$^{\tau}$ `dataFlow`	`InboundState dataFlow`
`InboundState dataFlow`	`OutboundState dataFlow`
`DuplexState`	`DuplexState`

Table 5.1: Symmetry between local and remote states

Another symmetry that we tried to preserve is between `Unidirectional` and `Duplex` connections. The `Duplex` side is considerably more complex as it includes interaction between `Inbound` and `Outbound` connections (in the sense that inbound connections can migrate to outbound only and vice versa). However, the state machine for an inbound-only connection is the same whether it is `Duplex` or `Unidirectional`, see Figure 5.4. A *connection manager* running in `ResponderMode` will use this state machine.

For *node-to-client* server, it will be even simpler, as there we only allow for unidirectional connections. Nevertheless, this symmetry simplifies the implementation.

5.8.4 Transitions

Reserve

When *connection manager* is asked for an outbound connection, it reserves a slot in its state for that connection. If any other thread asks for the same outbound connection, the *connection manager* will raise an exception in that thread. Reservation is done to guarantee exclusiveness for state transitions to a single outbound thread.

Connected

This transition is executed once an outbound connection successfully performs the `connect` system call.

Accepted and Overwritten

Transition driven by the `accept` system call. Once it returns, the *connection manager* might either not know about such connection or, there might be one in `ReservedOutboundState`. The Accepted transition represents the former situation, while the Overwritten transition captures the latter.

Let us note that if Overwritten transition happened, then on the outbound side, the scheduled `connect` call will fail. In this case, the *p2p governor* will recover, putting the peer in a queue of failed peers and will either try to connect to another peer or reconnect to that peer after some time, in which case it would re-use the accepted connection (assuming that a duplex connection was negotiated).

Negotiated$^{\text{Unidirectional}}_{\text{Outbound}}$ and Negotiated$^{\text{Duplex}}_{\text{Outbound}}$

Once an outbound connection has been negotiated, one of Negotiated$^{\text{Unidirectional}}_{\text{Outbound}}$ or Negotiated$^{\text{Duplex}}_{\text{Outbound}}$ transition is performed, depending on the result of a handshake negotiation. Duplex connections are negotiated only for node-to-node protocol versions higher than `NodeToNodeV_7`, and neither side declared that it is an *initiator* only.

If a duplex outbound connection was negotiated, the *connection manager* needs to ask the *inbound protocol governor* to start and monitor responder mini-protocols on the outbound connection.

Implementation detail

This transition is done by the `requestOutboundConnection`.

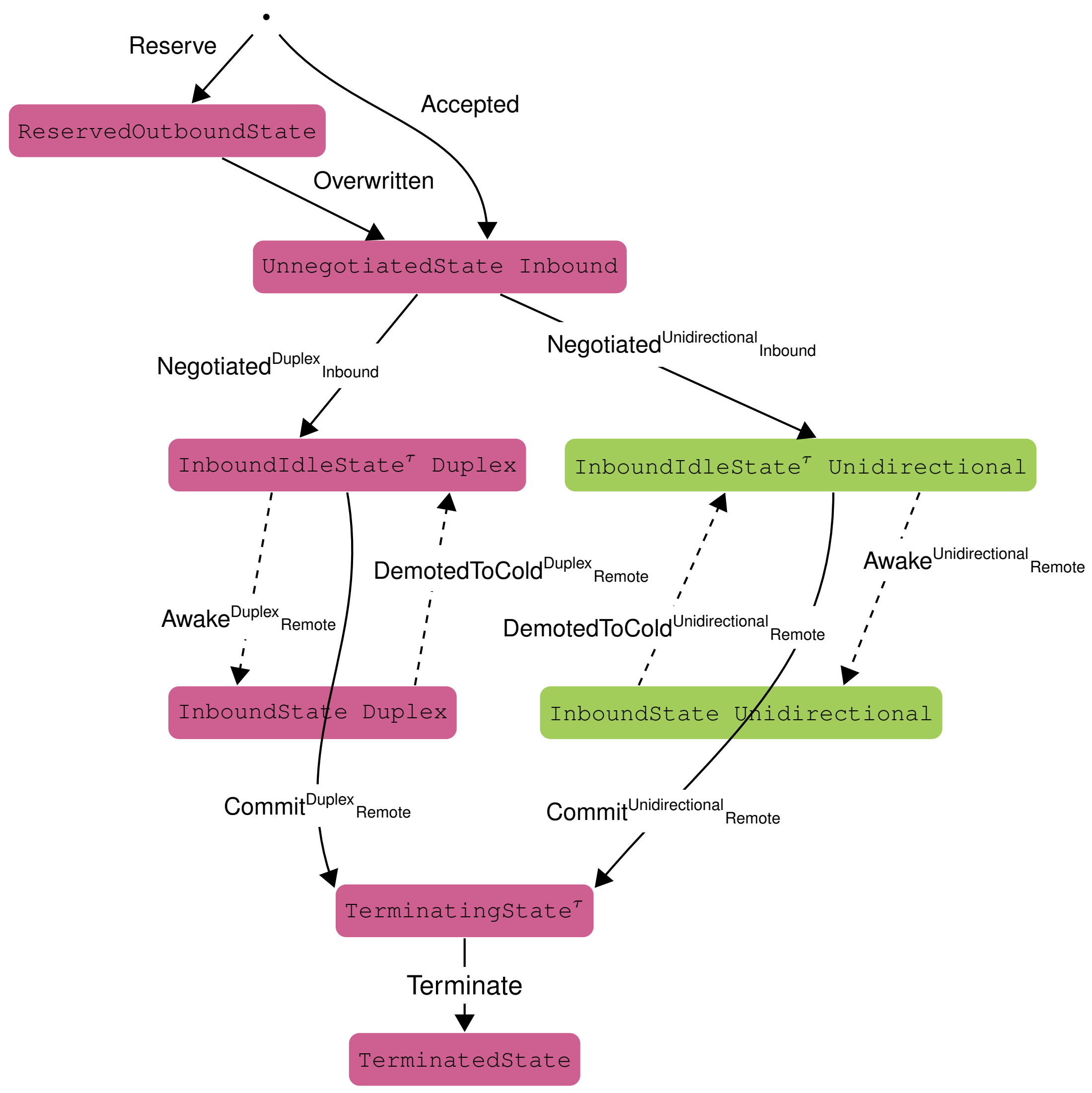

Reserve
Accepted
ReservedOutboundState
Overwritten
UnnegotiatedState Inbound
Negotiated^Duplex_Inbound
Negotiated^Unidirectional_Inbound
InboundIdleState^τ Duplex
InboundIdleState^τ Unidirectional
DemotedToCold^Duplex_Remote
Awake^Unidirectional_Remote
Awake^Duplex_Remote
DemotedToCold^Unidirectional_Remote
InboundState Duplex
InboundState Unidirectional
Commit^Duplex_Remote
Commit^Unidirectional_Remote
TerminatingState^τ
Terminate
TerminatedState

Figure 5.4: Sub-graph of inbound states.

$\mathsf{Negotiated}^{\mathsf{Unidirectional}}{}_{\mathsf{Inbound}}$ and $\mathsf{Negotiated}^{\mathsf{Duplex}}{}_{\mathsf{Inbound}}$

This transition is performed once the handshake negotiated an unidirectional or duplex connection on an inbound connection.

For $\mathsf{Negotiated}^{\mathsf{Unidirectional}}{}_{\mathsf{Inbound}}$, $\mathsf{Negotiated}^{\mathsf{Duplex}}{}_{\mathsf{Inbound}}$, $\mathsf{Negotiated}^{\mathsf{Duplex}}{}_{\mathsf{Outbound}}$ transitions, the *inbound protocol governor* will restart all responder mini-protocols (for all *established*, *warm* and *hot* groups of mini-protocols) and keep monitoring them.

Implementation detail

This transition is done by the `includeInboundConnection`.

Implementation detail

Whenever a mini-protocol terminates, it is immediately restarted using an on-demand strategy. All *node-to-node* protocols have initial agency on the client side; hence, restarting them on-demand does not send any message.

$\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$, $\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$ and $\mathsf{Awake}^{\mathsf{Unidirectional}}{}_{\mathsf{Remote}}$

All the awake transitions start either at `InboundIdleState`$^{\tau}$ `dataFlow`, the $\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$ can also be triggered on `OutboundIdleState`$^{\tau}$ `Duplex`.

Implementation detail

$\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$ transition is done by `requestOutboundConnection` on the request of *p2p governor*, while $\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$ and $\mathsf{Awake}^{\mathsf{Unidirectional}}{}_{\mathsf{Remote}}$ are triggered by incoming traffic on any of the responder mini-protocols (asynchronously if detected any *warm*/*hot* transition).

$\mathsf{Commit}^{\mathsf{Unidirectional}}{}_{\mathsf{Remote}}$, $\mathsf{Commit}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$

Both commit transitions happen after *protocol idle timeout* of inactivity (as the $\mathsf{TimeoutExpired}$ transition does). They transition to `TerminatingState`$^{\tau}$ (closing the bearer). For duplex connections, a normal shutdown procedure goes through `InboundIdleState`$^{\tau}$ `Duplex` via $\mathsf{Commit}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$ - which gave the name to this transition.

The inactivity of responder mini-protocols triggers these transitions. They both protect against a client that connects but never sends any data through the bearer; also, as part of a termination sequence, it is protecting us from shutting down a connection which is transitioning between *warm* and *hot* states.

Both commit transitions:

- $\mathsf{Commit}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$
- $\mathsf{Commit}^{\mathsf{Unidirectional}}{}_{\mathsf{Remote}}$

need to detect idleness during a time interval (which we call: protocol idle timeout). If, during this time frame, inbound traffic on any responder mini-protocol is detected, one of the $\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$ or $\mathsf{Awake}^{\mathsf{Unidirectional}}{}_{\mathsf{Remote}}$ transition is performed. The idleness detection might also be interrupted by the local $\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$ transition.

Implementation detail

These transitions can be triggered by `unregisterInboundConnection` and `unregisterOutboundConnection` (both are non-blocking), but the stateful idleness detection during *protocol idle timeout* is implemented by the server.
The implementation relies on two properties:

- the multiplexer being able to start mini-protocols on-demand, which allows us to restart a mini-protocol as soon as it returns without disturbing idleness detection;
- the initial agency for any mini-protocol is on the client.

Implementation detail

Whenever an outbound connection is requested, we notify the server about a new connection. We also do that when the connection manager hands over an existing connection. If *inbound protocol governor* is already tracking that connection, we need to make sure that

- *inbound protocol governor* preserves its internal state of that connection;
- *inbound protocol governor* does not start mini-protocols, as they are already running (we restart responders as soon as they stop, using the on-demand strategy).

$\mathsf{DemotedToCold}^{\mathsf{Unidirectional}}{}_{\mathsf{Local}}$, $\mathsf{DemotedToCold}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$

This transition is driven by the *p2p governor* when it decides to demote the peer to *cold* state; its domain is `OutboundState dataFlow` or `OutboundState`$^{\tau}$ `Duplex`. The target state is `OutboundIdleState`$^{\tau}$ `dataFlow` in which the connection manager sets up a timeout. When the timeout expires, the connection manager will do $\mathsf{Commit}^{\mathsf{dataFlow}}{}_{\mathsf{Local}}$ transition, which will reset the connection.

Implementation detail

This transition is done by `unregisterOutboundConnection`.

$\mathsf{DemotedToCold}^{\mathsf{Unidirectional}}{}_{\mathsf{Remote}}$, $\mathsf{DemotedToCold}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$

Both transitions are edge-triggered, the connection manager is notified by the *inbound protocol governor* once it notices that all responders became idle. Detection of idleness during *protocol idle timeout* is done in a separate step which is triggered immediately, see section 5.8.4 for details.

Implementation detail

Both transitions are done by `demotedToColdRemote`.

$\mathsf{PromotedToWarm}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$

This transition is driven by the local *p2p governor* when it promotes a *cold* peer to *warm* state. *connection manager* will provide a handle to an existing connection, so that *p2p governor* can drive its state.

Implementation detail

This transition is done by `requestOutboundConnection`.

TimeoutExpired

This transition is triggered when the protocol idleness timeout expires while the connection is in `OutboundState`$^{\tau}$ `Duplex`. The server starts this timeout when it triggers $\mathsf{DemotedToCold}^{\mathsf{dataFlow}}{}_{\mathsf{Remote}}$ transition. The connection manager tracks the state of this timeout so we can decide if a connection in the outbound state can terminate or if it needs to wait for that timeout to expire.

Implementation detail

This transition is done by `unregisterInboundConnection`.

$\mathsf{PromotedToWarm}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$

The remote peer triggers this asynchronous transition. The *inbound protocol governor* can notice it by observing the multiplexer ingress side of running mini-protocols. It then should notify the *connection manager*.

> *Implementation detail*
> This transition is done by `promotedToWarmRemote`.
> The implementation relies on two properties:
>
> - all initial states of node-to-node mini-protocols have client agency, i.e. the the server expects an initial message;
> - all mini-protocols are started using an on-demand strategy, which allows to detect when a mini-protocol is brought to life by the multiplexer.

Prune transitions

First, let us note that a connection in `InboundState Duplex` could have been initiated by either side (Outbound or Inbound). This means that even though a node might not have accepted any connection, it could end up serving peers and possibly go beyond server hard limit, thus exceeding the number of allowed file descriptors. This is possible via the following path:

$\mathsf{Connected}$,

$\mathsf{Negotiated}^{\mathsf{Duplex}}{}_{\mathsf{Outbound}}$,

$\mathsf{PromotedToWarm}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$,

$\mathsf{DemotedToCold}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$

which leads from the initial state • to `InboundState Duplex`, the same state in which accepted duplex connections end up. Even though the server rate limits connections based on how many connections are in this state, we could exceed the server hard limit.

These are all transitions that potentially could lead to exceeding the server hard limit, all of them are transitions from some outbound/duplex state into an inbound/duplex state:

- `DuplexState` to `InboundState Duplex` (via $\mathsf{DemotedToCold}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$)
- `OutboundState`$^{\tau}$ `Duplex` to `InboundState Duplex` (via $\mathsf{DemotedToCold}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$)
- `OutboundIdleState`$^{\tau}$ `Duplex` to `InboundState Duplex` (via $\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$)
- `OutboundState`$^{\tau}$ `Duplex` to `DuplexState` (via $\mathsf{PromotedToWarm}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$)
- `OutboundState Duplex` to `DuplexState` (via $\mathsf{PromotedToWarm}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$)

To solve this problem, the connection manager will check to see if the server hard limit was exceeded in any of the above transitions. If that happens, the *connection manager* will reset an arbitrary connection (with some preference).

The reason why going from `OutboundState`$^{\tau}$ `Duplex` (or `OutboundState Duplex`, or `OutboundIdleState`$^{\tau}$ `Duplex`) to `InboundState Duplex` might exceed the server hard limit is exacty the same as the `DuplexState` to `InboundState Duplex` one. However, the reason why going from `OutboundState`$^{\tau}$ `Duplex` to `DuplexState` might exceed the limit is more tricky. To reach a `DuplexState`, one assumes there must have been an incoming *accepted* connection. However, there's another way that two end-points can establish a connection without a node accepting it. If two nodes try to request an outbound connection simultaneously, it is possible for two applications to both perform an active opening to each other at the same time. This is called a *simultaneous open*. In a simultaneous TCP open, we can have 2 nodes establishing a connection without any of them having explicitly accepted a connection, which can make a server violate its file descriptor limit.

Given this, we prefer to reset an inbound connection rather than close an outbound connection because, from a systemic point of view, outbound connections are more valuable than inbound ones. If we keep the number of *established* peers to be smaller than the server hard limit; with the right policy, we should never need to reset a connection in `DuplexState`. However, when dealing with a connection that transitions from `OutboundState`$^{\tau}$ `Duplex` to

`DuplexState`, we actually need to make sure this connection is closed, because we have no way to know for sure if this connection is the result of a TCP simultaneous open there might not be any other connection available to prune that can make space for this one.

The *inbound protocol governor* is in a position to make an educated decision about which connection to reset. Initially, we aim for a decision driven by randomness, but other choices are possible[2] and the implementation should allow to easily extend the initial choice.

$\mathsf{Commit}^{\mathsf{Unidirectional}}{}_{\mathsf{Remote}}$, $\mathsf{Commit}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$

Both commit transitions happen after *protocol idle timeout* of inactivity (as the TimeoutExpired transition does). They transition to `TerminatingState`$^{\tau}$ (closing the bearer). For duplex connections, a normal shutdown procedure goes through `InboundIdleState`$^{\tau}$ `Duplex` via $\mathsf{Commit}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$ - which gave the name to this transition, or through `OutboundIdleState`$^{\tau}$ `Duplex` via $\mathsf{Commit}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$ transition.

These transitions are triggered by the inactivity of responder mini-protocols. They both protect against a client that connects but never sends any data through the bearer; also, as part of a termination sequence, it is protecting us from shutting down a connection which is transitioning between *warm* and *hot* states.

Both commit transitions:

- $\mathsf{Commit}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$
- $\mathsf{Commit}^{\mathsf{Unidirectional}}{}_{\mathsf{Remote}}$

Need to detect idleness during time interval (which we call: protocol idle timeout). If during this time frame, inbound traffic on any responder mini-protocol is detected, one of the $\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$ or $\mathsf{Awake}^{\mathsf{Unidirectional}}{}_{\mathsf{Remote}}$ transition is performed. The local $\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$ transition might also interrupt the idleness detection.

Implementation detail

These transitions can be triggered by `unregisterInboundConnection` and `unregisterOutboundConnection` (both are non-blocking), but the stateful idleness detection during *protocol idle timeout* is implemented by the *inbound protocol governor*. The implementation relies on two properties:

- the multiplexer being able to start mini-protocols on-demand, which allows us to restart a mini-protocol as soon as it returns without disturbing idleness detection;
- the initial agency for any mini-protocol is on the client.

Implementation detail

Whenever an outbound connection is requested, we notify the server about a new connection. We also do that when the connection manager hands over an existing connection. If *inbound protocol governor* is already tracking that connection, we need to make sure that

- *inbound protocol governor* preserves its internal state of that connection;
- *inbound protocol governor* does not start mini-protocols, as they are already running (we restart responders as soon as they stop, using the on-demand strategy).

$\mathsf{Commit}^{\mathsf{Unidirectional}}{}_{\mathsf{Local}}$, $\mathsf{Commit}^{\mathsf{Duplex}}{}_{\mathsf{Local}}$

As previous two transitions, these also are triggered after *protocol idle timeout*, but this time, they are triggered on the outbound side. This transition will reset the connection, and the timeout ensures that the remote end can clear its ingress queue before the TCP reset arrives. For a more detailed analysis, see 5.8.6 section.

Terminate

After a connection is closed, we keep it in `TerminatingState`$^{\tau}$ for the duration of *wait time timeout*. When the timeout expires, the connection is forgotten.

[2] We can take into account whether we are *hot* to the remote end, or for how long we have been *hot* to to the remote node.

Connecting to oneself

The transitions described in this section can only happen when the connection the manager was requested to connect to its own listening socket and the address wasn't translated by the OS or a NAT. This could happen only in particular situations:

1. misconfiguration a system;
2. running a node on multiple interfaces;
3. in some cases, it could also happen when learning about oneself from the ledger;
4. or due to peer sharing.

In some of these cases, the external IP address would need to agree with the internal one, which is true for some cloud service providers.

Let us note that these connections effectively only add delay, and thus they will be replaced by the outbound governor (by its churn mechanism).

These transitions are not indicated in the figure 5.3, instead they are shown bellow in figure 5.5.

SelfConn and SelfConn^{-1} We allow transitioning between

- `UnnegotiatedState Outbound` and
- `UnnegotiatedState Inbound`

or the other way. This transition is not guaranteed as on some systems in such case, the outbound and inbound addresses (as returned by the `accept` call) can be different. Whether SelfConn or SelfConn^{-1} will happen depending on the race between the inbound and outbound sides.

SelfConn' and SelfConn'$^{-1}$ We also allow transitioning between

- `InboundIdleState`$^{\tau}$ `dataFlow` and
- `OutboundState dataFlow`

After the handshake is negotiated, there is a race between inbound and outbound threads, which need to be resolved consistently.

5.8.5 Protocol errors

If a mini-protocol errors, on either side, the connection will be reset and put in `TerminatedState`. This can happen in any connection state.

5.8.6 Closing connection

By default, when the operating system is closing a socket, it is done in the background, but when `SO_LINGER` option is set, the `close` system call blocks until either all messages are sent or the specified linger timeout fires. Unfortunately, our experiments showed that if the remote side (not the one that called `close`), delays reading the packets, then even with `SO_LINGER` option set, the socket is kept in the background by the OS. On `FreeBSD` it is eventually closed cleanly, on `Linux` and `OSX` it is reset. This behaviour gives the remote end the power to keep resources for an extended amount of time, which we want to avoid. We thus decided to always use `SO_LINGER` option with timeout set to `0`, which always resets the connection (i.e. it sets the `RST` TCP flag). This has the following consequences:

- Four-way handshake used by TCP termination will not be used. The four-way handshake allows one to close each side of the connection separately. With the reset, the OS is instructed to forget the state of the connection immediately (including freeing unread ingress buffer).
- the system will not keep the socket in `TIME_WAIT` state, which was designed to:

– provide enough time for final `ACK` to be received;

– protect the connection from packets that arrive late. Such packets could interfere with a new connection (see Stevens et al. (2003)).

The connection state machine makes sure that we close a connection only when both sides are not using the connection for some time: for outbound connections this is configured by the timeout on the `OutboundIdleState`$^{\tau}$ `dataFlow`, while for inbound connections by the timeout on the `InboundIdleState`$^{\tau}$ `dataFlow`. This ensures that the application can read from ingress buffers before the `RST` packet arrives. Excluding protocol errors and prune transitions, which uncooperatively reset the connection.

We also provide application-level `TIME_WAIT` state: `TerminatingState`$^{\tau}$, in which we keep a connection, which should also protect us from late packets from a previous connection. However, the connection manager does allow to accept new connections during `TerminatingState`$^{\tau}$ - it is the client's responsibility not to reconnect too early. For example, *p2p governor* enforces a 60s idle period before it can reconnect to the same peer, after either a protocol error or a connection failure.

From an operational point of view, it's essential that connections are not held in `TIME_WAIT` state for too long. This would be problematic when restarting a node (without rebooting the system) (e.g. when adjusting configuration). Since we reset connections, this is not a concern.

5.8.7 *Outbound* connection

If the connection state is in either `ReservedOutboundState`, `UnnegotiatedState Inbound` or `InboundState Duplex` then, when calling `requestOutboundConnection` the state of a connection leads to either `OutboundState Unidirectional` or `DuplexState`.

If `Unidirectional` connection was negotiated, `requestOutboundConnection` must error. If `Duplex` connection was negotiated, it can use the egress side of this connection leading to `DuplexState`.

initial state (•): the *connection manager* does not have a connection with that peer. The connection is put in `ReservedOutboundState` before *connection manager* connects to that peer;

`UnnegotiatedState Inbound`: if the *connection manager* accepted a connection from that peer, handshake is ongoing; `requestOutboundConnection` will await until the connection state changes to `InboundState dataFlow`.

`InboundState Unidirectional`: if `requestOutboundConnection` finds a connection in this state it will error.

`InboundState Duplex`: if *connection manager* accepted connection from that peer and handshake negotiated a `Duplex` data flow; `requestOutboundConnection` transitions to `DuplexState`.

`TerminatingState`$^{\tau}$: block until `TerminatedState` and start from the initial state.

Otherwise: if *connection manager* is asked to connect to peer, and there exists a connection in any other state, e.g. `UnnegotiatedState Outbound`, `OutboundState dataFlow`, `DuplexState`, *connection manager* signals the caller with an error, see section 5.2.

Figure 5.6 shows outbound connection state evolution, e.g. the flow graph of `requestOutboundConnection`.

`OutboundState Duplex` and `DuplexState`

Once an outbound connection negotiates `Duplex` data flow, it transfers to `OutboundState Duplex`. At this point, we need to start responder protocols. This means that the *connection manager* needs a way to inform the server (which accepts and monitors inbound connections) to start the protocols and monitor that connection. This connection will transition to `DuplexState` only once we notice incoming traffic on any of *established* protocols. Since this connection might have been established via TCP simultaneous open, this transition to `DuplexState` can also trigger Prune transitions if the number of inbound connections becomes above the limit.

Implementation detail

The implementation is using a `TBQueue`. The server uses this channel for incoming duplex outbound and inbound connections.

Termination

When *p2p governor* demotes a peer to *cold* state, an outbound the connection needs to transition from either:

- `OutboundState dataFlow` to `OutboundIdleState`$^{\tau}$ `dataFlow`
- `OutboundState`$^{\tau}$ `Duplex` to `InboundIdleState`$^{\tau}$ `Duplex`
- `DuplexState` to `InboundState Duplex`

To support that the *connection manager* exposes a method:

$\mathsf{unregisterOutboundConnection} :: \mathsf{peerAddr} \to \mathsf{m}\ ()$

This method performs $\mathsf{DemotedToCold^{Unidirectional}}_{\mathsf{Local}}$ or $\mathsf{DemotedToCold^{Duplex}}_{\mathsf{Local}}$ transition. In the former case, it will shut down the multiplexer and close the TCP connection; in the latter case, besides changing the connection state, it will also trigger Prune transitions if the number of inbound connections is above the limit.

Connection manager methods

The tables 5.2 and 5.3 show transitions performed by

- `requestOutboundConnection` and
- `unregisterOutboundConnection`

respectively.

The choice between `no-op` and error is solved by the following rule: if the calling component (e.g. *p2p governor*), can keep its state in a consistent state with *connection manager* then use `no-op`, otherwise error. Since both *inbound protocol governor* and *p2p governor* are using *mux* to track the state of the connection, the state can't be inconsistent.

5.8.8 *Inbound* connection

Initial states for inbound connection are either:

- initial state •;
- `ReservedOutboundState`: this can happen when `requestOutboundConnection` reserves a connection with `ReservedOutboundState`, but before it calls `connect` the `accept` call returned. In this case, the `connect` call will fail and, as a consequence, `requestOutboundConnection` will fail too. Any mutable variables used by it can be disposed since no thread can be blocked on it: if another thread asked for an outbound connection with that peer, it would see `ReservedOutboundState` and throw `ConnectionExists` exception.

  To make sure that this case is uncommon, we need to guarantee that the *connection manager* does not block between putting the connection in the `ReservedOutboundState` and calling the `connect` system call.

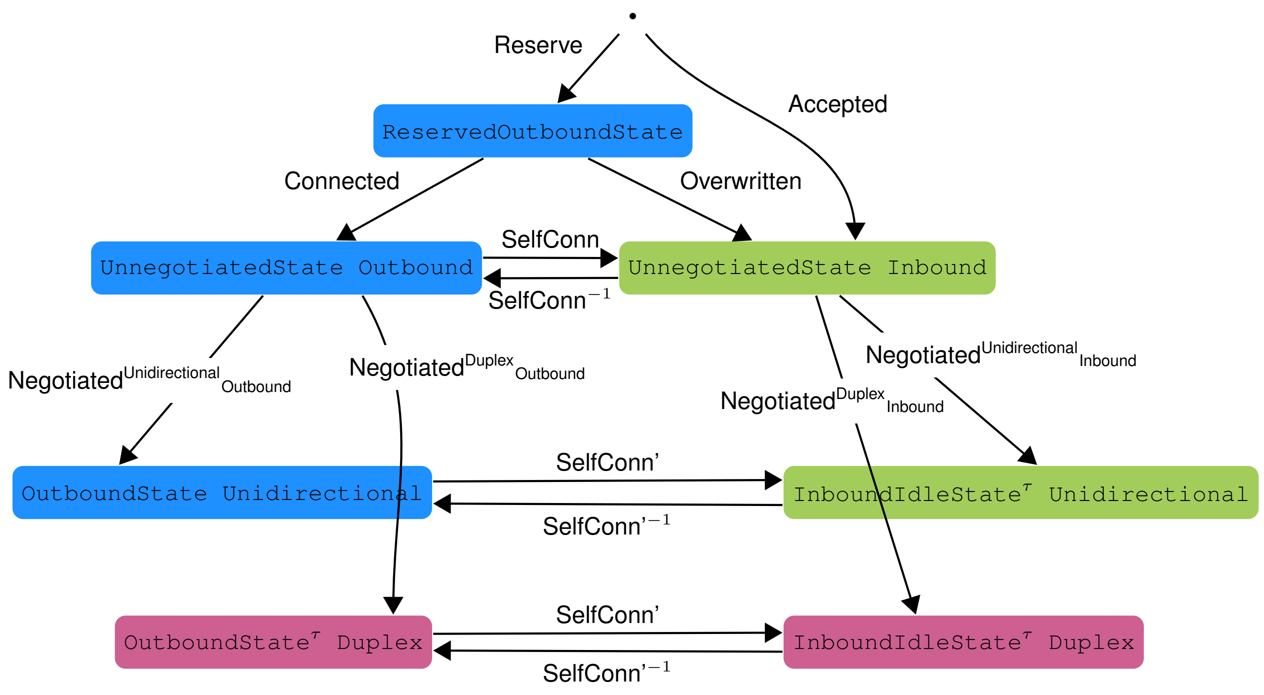
Connection manager methods

The following tables show transitions of the following connection manager methods:

- `includeInboundConnection`: table 5.4
- `promotedToWarmRemote`: table 5.5
- `demotedToColdRemote`: table 5.6

State	*Action*
•	• ReservedOutboundState, • Connected, • start connection thread (handshake, *mux*) • $\text{Negotiated}^{\text{Unidirectional}}{}_{\text{Outbound}}$ or $\text{Negotiated}^{\text{Duplex}}{}_{\text{Outbound}}$
ReservedOutboundState	error ConnectionExists
UnnegotiatedState Outbound	error ConnectionExists
UnnegotiatedState Inbound	await for InboundState dataFlow, if negotiated duplex connection transition to DuplexState, otherwise error ForbiddenConnection
OutboundState dataFlow	error ConnectionExists
OutboundState$^{\tau}$ Duplex	error ConnectionExists
OutboundIdleState$^{\tau}$ dataFlow	error ForbiddenOperation
InboundIdleState$^{\tau}$ Unidirectional	error ForbiddenConnection
InboundIdleState$^{\tau}$ Duplex	transition to OutboundState Duplex
InboundState Unidirectional	error ForbiddenConnection
InboundState Duplex	transition to DuplexState
DuplexState	error ConnectionExists
TerminatingState$^{\tau}$	await for TerminatedState
TerminatedState	can be treated as initial state

Table 5.2: requestOutboundConnection; states indicated with a † are forbidden by TCP.

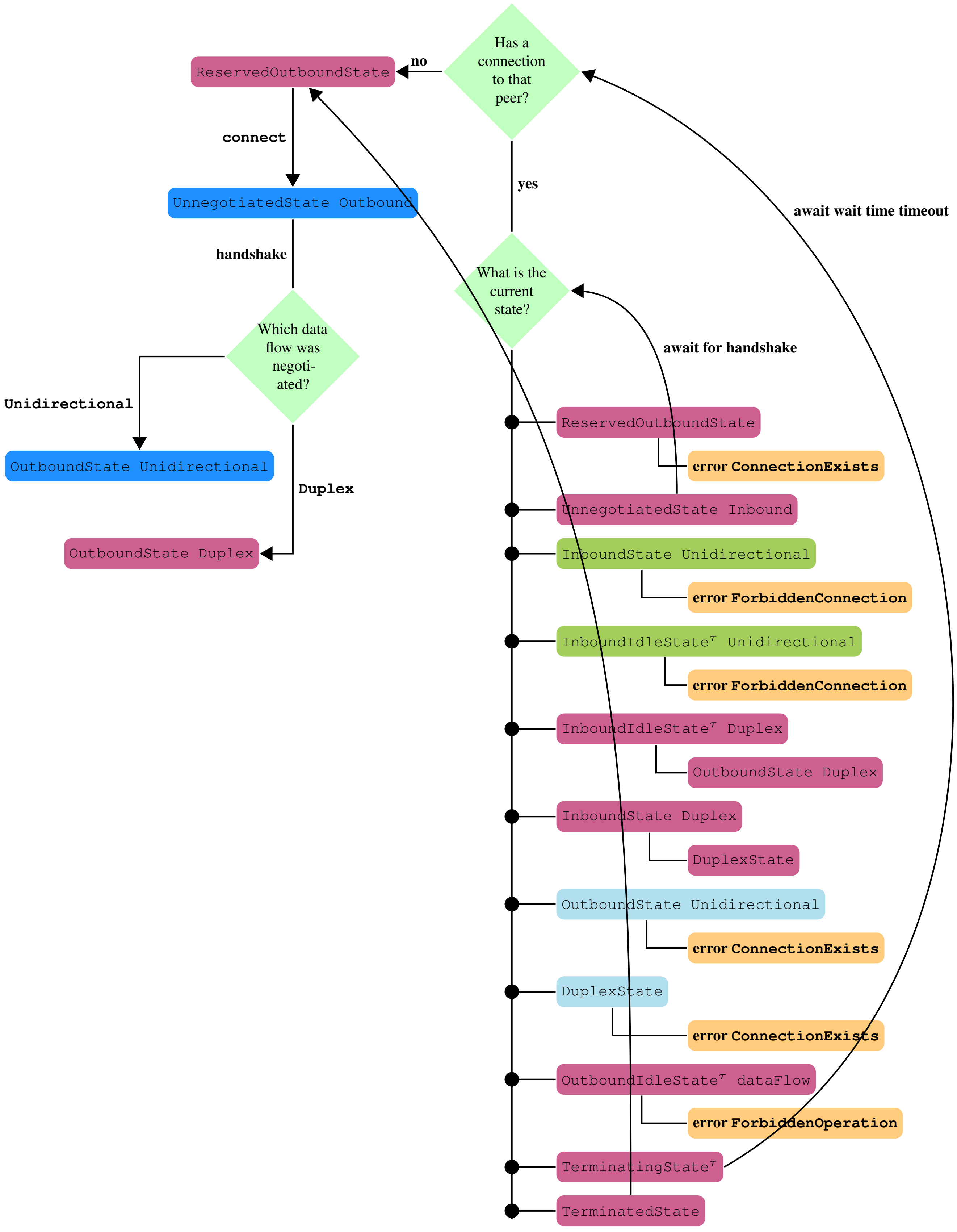


•
Reserve
Accepted
ReservedOutboundState
Connected
Overwritten
UnnegotiatedState Outbound
SelfConn
SelfConn^{-1}
UnnegotiatedState Inbound
Negotiated$^{Unidirectional}_{Outbound}$
Negotiated$^{Duplex}_{Outbound}$
Negotiated$^{Unidirectional}_{Inbound}$
Negotiated$^{Duplex}_{Inbound}$
OutboundState Unidirectional
SelfConn'
SelfConn'$^{-1}$
InboundIdleState$^{\tau}$ Unidirectional
OutboundState$^{\tau}$ Duplex
SelfConn'
SelfConn'$^{-1}$
InboundIdleState$^{\tau}$ Duplex

Figure 5.5: Extra transitions when connecting to onself

State	*Action*
•	`no-op`
`ReservedOutboundState`	error `ForbiddenOperation`
`UnnegotiatedState Outbound`	error `ForbiddenOperation`
`UnnegotiatedState Inbound`	error `ForbiddenOperation`
`OutboundState dataFlow`	DemotedToCold$^{dataFlow}_{Local}$
`OutboundState`$^{\tau}$ `Duplex`	Prune or DemotedToCold$^{Duplex}_{Local}$
`OutboundIdleState`$^{\tau}$ `dataFlow`	`no-op`
`InboundIdleState`$^{\tau}$ `Unidirectional`	assertion error
`InboundIdleState`$^{\tau}$ `Duplex`	`no-op`
`InboundState Unidirectional`	assertion error
`InboundState Duplex`	`no-op`
`DuplexState`	Prune or DemotedToCold$^{Duplex}_{Local}$
`TerminatingState`$^{\tau}$	`no-op`
`TerminatedState`	`no-op`

Table 5.3: `unregisterOutboundConnection`



Has a connection to that peer?
no
ReservedOutboundState
connect
UnnegotiatedState Outbound
handshake
Which data flow was negoti-ated?
Unidirectional
OutboundState Unidirectional
Duplex
OutboundState Duplex
yes
What is the current state?
await for handshake
await wait time timeout
ReservedOutboundState
error ConnectionExists
UnnegotiatedState Inbound
InboundState Unidirectional
error ForbiddenConnection
InboundIdleState^τ Unidirectional
error ForbiddenConnection
InboundIdleState^τ Duplex
OutboundState Duplex
InboundState Duplex
DuplexState
OutboundState Unidirectional
error ConnectionExists
DuplexState
error ConnectionExists
OutboundIdleState^τ dataFlow
error ForbiddenOperation
TerminatingState^τ
TerminatedState

Figure 5.6: *Outbound* connection flow graph

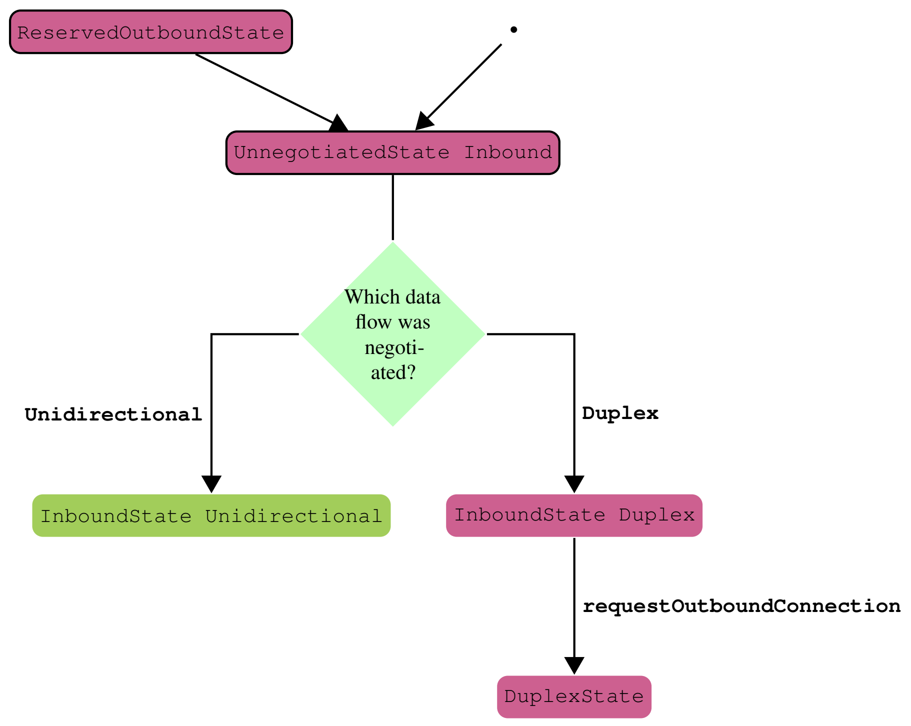

ReservedOutboundState
UnnegotiatedState Inbound
Which data flow was negotiated?
Unidirectional
Duplex
InboundState Unidirectional
InboundState Duplex
requestOutboundConnection
DuplexState

Figure 5.7: *Inbound* connection flow graph, where both bordered states: `ReservedOutboundState` and `UnnegotiatedState Inbound` are initial states.

- `unregisterInboundConnection`: table 5.7

States indicated by '-' are preserved, though unexpected; `promotedToWarmRemote` will use `UnsupportedState :: OperationResult a` to indicate that to the caller.

States indicated with a † are forbidden by TCP.

Transitions denoted by † should not happen. The implementation is using assertion, and the production system will trust that the server side calls `unregisterInboundConnection` only after all responder mini-protocols where idle for *protocol idle timeout*.

`unregisterInboundConnection` might be called when the connection is in `OutboundState Duplex`. This can, though very rarely, happen as a race between $\mathsf{Awake}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$ and $\mathsf{DemotedToCold}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$[3]. Let's consider the following sequence of transitions:

[3]race is not the right term, these transitions are concurrent and independent

State	*Action*
•	• start connection thread (handshake, *mux*) • transition to `UnnegotiatedState Inbound`. • await for handshake result • transition to $\texttt{InboundIdleState}^{\tau}$ `dataFlow`.
`ReservedOutboundState`	the same as •
`UnnegotiatedState prov`	`impossible state`†
$\texttt{InboundIdleState}^{\tau}$ `dataFlow`	`impossible state`†
`InboundState dataFlow`	`impossible state`†
`OutboundState dataFlow`	`impossible state`†
`DuplexState`	`impossible state`†
$\texttt{TerminatingState}^{\tau}$	the same as •
`TerminatedState`	the same as •

Table 5.4: `includeInboundConnection`

StateIn	*StateOut*	*Transition*
•	-	
`ReservedOutboundState`	-	
`UnnegotiatedState prov`	-	
`OutboundState Unidirectional`	-	
`OutboundState Duplex`	Prune or (`DuplexState`	$\text{PromotedToWarm}^{\text{Duplex}}{}_{\text{Remote}}$)
$\texttt{InboundIdleState}^{\tau}$ `Unidirectional`	`InboundState Unidirectional`	$\text{Awake}^{\text{Unidirectional}}{}_{\text{Remote}}$
$\texttt{InboundIdleState}^{\tau}$ `Duplex`	`InboundState Duplex`	$\text{Awake}^{\text{Duplex}}{}_{\text{Remote}}$
`InboundState Unidirectional`	-	
`InboundState Duplex`	-	
`DuplexState`	-	
$\texttt{TerminatingState}^{\tau}$	-	
`TerminatedState`	-	

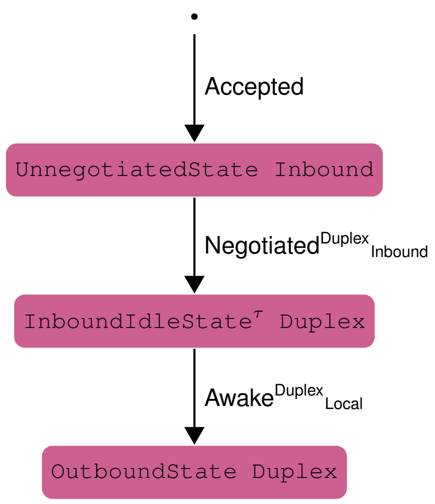
Table 5.5: `promotedToWarmRemote`

StateIn	*StateOut*	*Transition*
`ReservedOutboundState`	-	-
`UnnegotiatedState prov`	-	-
`OutboundState dataFlow`	-	-
`InboundIdleState`$^\tau$ `dataFlow`	-	-
`InboundState dataFlow`	`InboundIdleState`$^\tau$ `dataFlow`	$\mathsf{DemotedToCold}^{\mathsf{dataFlow}}{}_{\mathsf{Remote}}$
`DuplexState`	`OutboundState`$^\tau$ `Duplex`	$\mathsf{DemotedToCold}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$
`TerminatingState`$^\tau$	-	-
`TerminatedState`	-	-

Table 5.6: `demotedToColdRemote`

StateIn	*StateOut*	*Returned Value*	*Transition(s)*
•	-	-	
`ReservedOutboundState`	-	-	
`UnnegotiatedState prov`	-	-	
`OutboundState`$^\tau$ `Unidirectional`	†	-	
`OutboundState Unidirectional`	†	-	
`OutboundState`$^\tau$ `Duplex`	`OutboundState Duplex`	-	
`OutboundState Duplex`	†	-	
`InboundIdleState`$^\tau$ `dataFlow`	`TerminatingState`$^\tau$	`True`	
`InboundState dataFlow`	`TerminatingState`$^{\tau\dagger}$	`True`	$\mathsf{DemotedToCold}^{\mathsf{dataFlow}}{}_{\mathsf{Remote}}$ $\mathsf{Commit}^{\mathsf{dataFlow}}{}_{\mathsf{Remote}}$
`DuplexState`	`OutboundState Duplex`	`False`	$\mathsf{DemotedToCold}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$
`TerminatingState`$^\tau$	-	-	
`TerminatedState`	-	-	

Table 5.7: `unregisterInboundConnection`



If the *protocol idle timeout* on the `InboundIdleState`$^\tau$ `Duplex` expires the $\mathsf{Awake}^{\mathsf{Duplex}}_{\mathsf{Remote}}$ transition is triggered and the *inbound protocol governor* calls `unregisterInboundConnection`.

5.9 Server

The server consists of an accept loop and an *inbound protocol governor*. The accept loop is using `includeInboundConnnection` on incoming connections, while the *inbound protocol governor* tracks the state of the responder side of all mini-protocols and it is responsible for starting and restarting mini-protocols, as well as detecting if they are used to support:

- $\mathsf{PromotedToWarm}^{\mathsf{Duplex}}_{\mathsf{Remote}}$,
- $\mathsf{DemotedToCold}^{\mathsf{Unidirectional}}_{\mathsf{Remote}}$,
- $\mathsf{Commit}^{\mathsf{Unidirectional}}_{\mathsf{Remote}}$ and $\mathsf{Commit}^{\mathsf{Duplex}}_{\mathsf{Remote}}$ transitions.

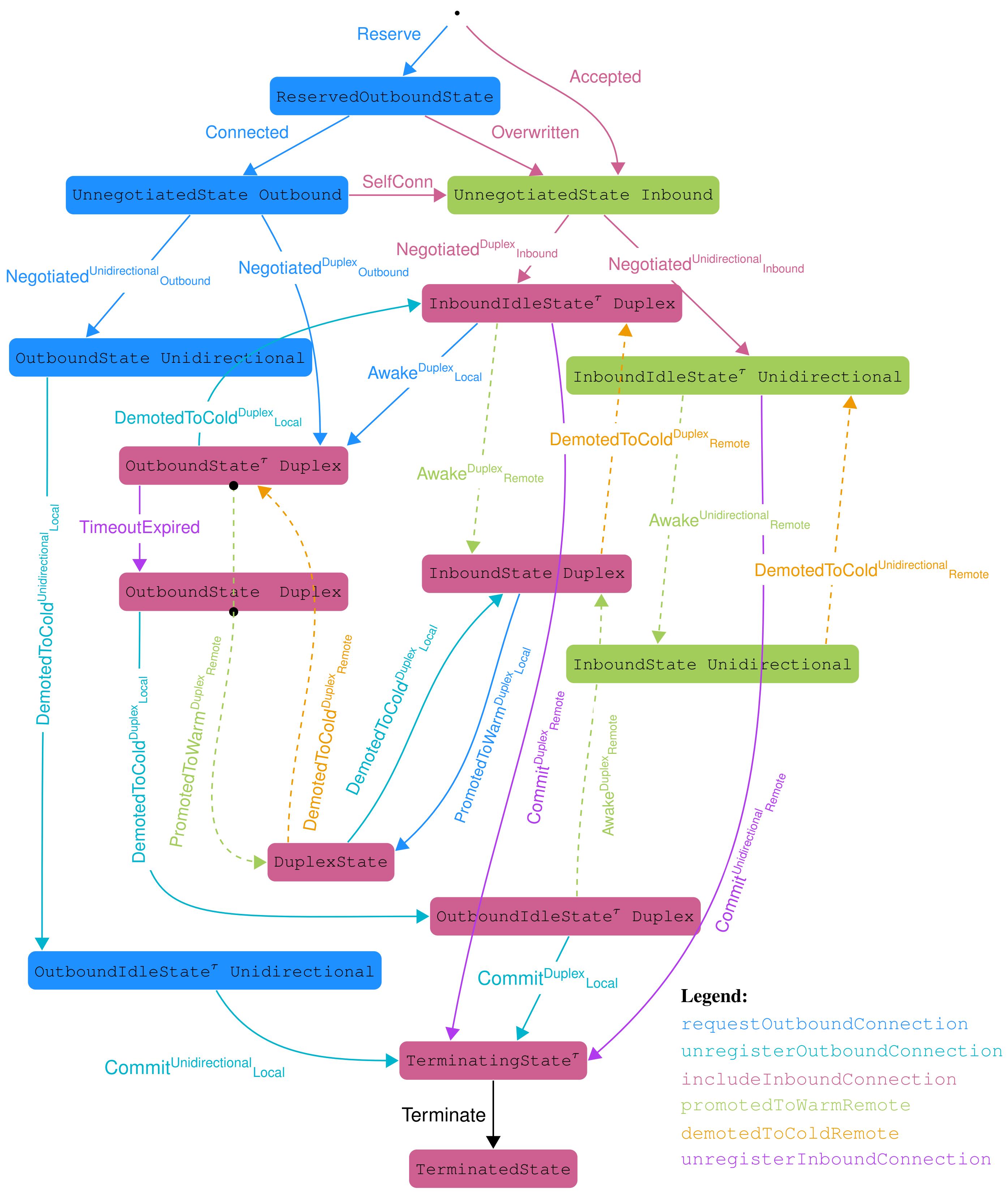
The *inbound protocol governor* will always start/restart all the mini-protocols using `StartOnDemand` strategy. When the multiplexer detects any traffic on its ingress queues, corresponding to responder protocols, it will do the $\mathsf{PromotedToWarm}^{\mathsf{Duplex}}_{\mathsf{Remote}}$ transition using `promotedToWarmRemote` method.

Once all responder mini-protocols become idle, i.e. they all stopped, were restarted (on-demand) but are not yet running, a $\mathsf{DemotedToCold}^{\mathsf{dataFlow}}_{\mathsf{Remote}}$ transition is run: the *inbound protocol governor* will notify the *connection manager* using:

```
-- | Notify the 'ConnectionManager' that a remote end demoted us to a /cold
-- peer/.
--
-- This executes:
--
-- * \(DemotedToCold^{*}_{Remote}\) transition.
demotedToColdRemote
    :: HasResponder muxMode ~ True
   => ConnectionManager muxMode socket peerAddr handle handleError m
   -> peerAddr -> m (OperationResult InState)
```

When all responder mini-protocols are idle for *protocol idle timeout*, the *inbound protocol governor* will execute `unregisterInboundConnection` which will trigger:

- $\mathsf{Commit}^{\mathsf{Unidirectional}}_{\mathsf{Remote}}$ or $\mathsf{Commit}^{\mathsf{Duplex}}_{\mathsf{Remote}}$ if the initial state is `InboundIdleState`$^\tau$ `Duplex`;
- $\mathsf{TimeoutExpired}$ if the initial state is `OutboundState`$^\tau$ `Duplex`;
- `no-op` if the initial state is `OutboundState Duplex` or `OutboundIdleState`$^\tau$ `dataFlow`.



Reserve
Accepted
ReservedOutboundState
Connected
Overwritten
SelfConn
UnnegotiatedState Outbound
UnnegotiatedState Inbound
Negotiated^Unidirectional_Outbound
Negotiated^Duplex_Outbound
Negotiated^Duplex_Inbound
Negotiated^Unidirectional_Inbound
InboundIdleState^τ Duplex
OutboundState Unidirectional
InboundIdleState^τ Unidirectional
Awake^Duplex_Local
DemotedToCold^Duplex_Local
DemotedToCold^Duplex_Remote
OutboundState^τ Duplex
Awake^Duplex_Remote
Awake^Unidirectional_Remote
TimeoutExpired
DemotedToCold^Unidirectional_Local
OutboundState Duplex
InboundState Duplex
DemotedToCold^Unidirectional_Remote
InboundState Unidirectional
DemotedToCold^Duplex_Local
PromotedToWarm^Duplex_Remote
DemotedToCold^Duplex_Remote
DemotedToCold^Duplex_Local
PromotedToWarm^Duplex_Local
Commit^Duplex_Remote
Awake^Duplex_Remote
Commit^Unidirectional_Remote
DuplexState
OutboundIdleState^τ Duplex
OutboundIdleState^τ Unidirectional
Commit^Duplex_Local
Commit^Unidirectional_Local
TerminatingState^τ
Terminate
TerminatedState
Legend:
requestOutboundConnection
unregisterOutboundConnection
includeInboundConnection
promotedToWarmRemote
demotedToColdRemote
unregisterInboundConnection

Figure 5.8: Transitions classified by connection manager method.

```
-- | Return the value of 'unregisterInboundConnection' to inform the caller about
-- the transition .
--
data DemotedToColdRemoteTr =
    -- | @Commit^{dataFlow}@ transition from @'InboundIdleState' dataFlow@.
    --
    CommitTr

    -- | @DemotedToCold^{Remote}@ transition from @'InboundState' dataFlow@
    --
  | DemotedToColdRemoteTr

    -- | Either @DemotedToCold^{Remote}@ transition from @'DuplexState'@, or
    -- a level triggered @Awake^{Duplex}_{Local}@ transition. In both cases
    -- the server must keep the responder's side of all protocols ready.
  | KeepTr
  deriving Show

unregisterInboundConnection :: peerAddr ⇒ m (OperationResult DemotedToColdRemoteTr)
```

Both $\mathsf{Commit}^{\mathsf{Unidirectional}}{}_{\mathsf{Remote}}$ and $\mathsf{Commit}^{\mathsf{Duplex}}{}_{\mathsf{Remote}}$ will free resources (terminate the connection thread, close the socket).

5.10 Inbound Protocol Governor

Inbound protocol governor keeps track of the responder side of the protocol for both inbound and outbound duplex connections. Unidirectional outbound connections are not tracked by *inbound protocol governor*. The server and connection manager are responsible for notifying it about new connections once negotiated. Figure 5.9 presents the state machine that drives changes to connection states tracked by *inbound protocol governor*. As in the connection manager case, there is an implicit transition from every state to the terminating state, representing mux or mini-protocol failures.

5.10.1 States

States of the inbound governor are similar to the outbound governor, but there are crucial differences.

RemoteCold

The remote cold state signifies that the remote peer is not using the connection, however the only reason why the inbound governor needs to track that connection is because the outbound side of this connection is used. The inbound governor will wait until any of the responder mini-protocols wakes up (AwakeRemote) or the mux will be shut down (MuxTerminated).

$\mathsf{RemoteIdle}^{\tau}$

The $\mathsf{RemoteIdle}^{\tau}$ state is the initial state of each new connection (NewConnection). An active connection will become $\mathsf{RemoteIdle}^{\tau}$ once the inbound governor detects that all responder mini-protocols terminated (WaitIdleRemote). When a connection enters this state, an idle timeout is started. If no activity is detected on the responders, the connection will either be closed by the connection manager and forgotten by the inbound governor or progress to the RemoteCold state. This depends on whether the connection is used (*warm* or *hot*) or not (*cold*) by the outbound side.

RemoteWarm

A connection dwells in RemoteWarm if there are strictly only any warm or established responder protocols running. Note also that an established protocol is one that may run in both hot and warm states, but cannot be the only type running to maintain hot state once all proper hot protocols have terminated. In other words, the connection must be demoted in that case.

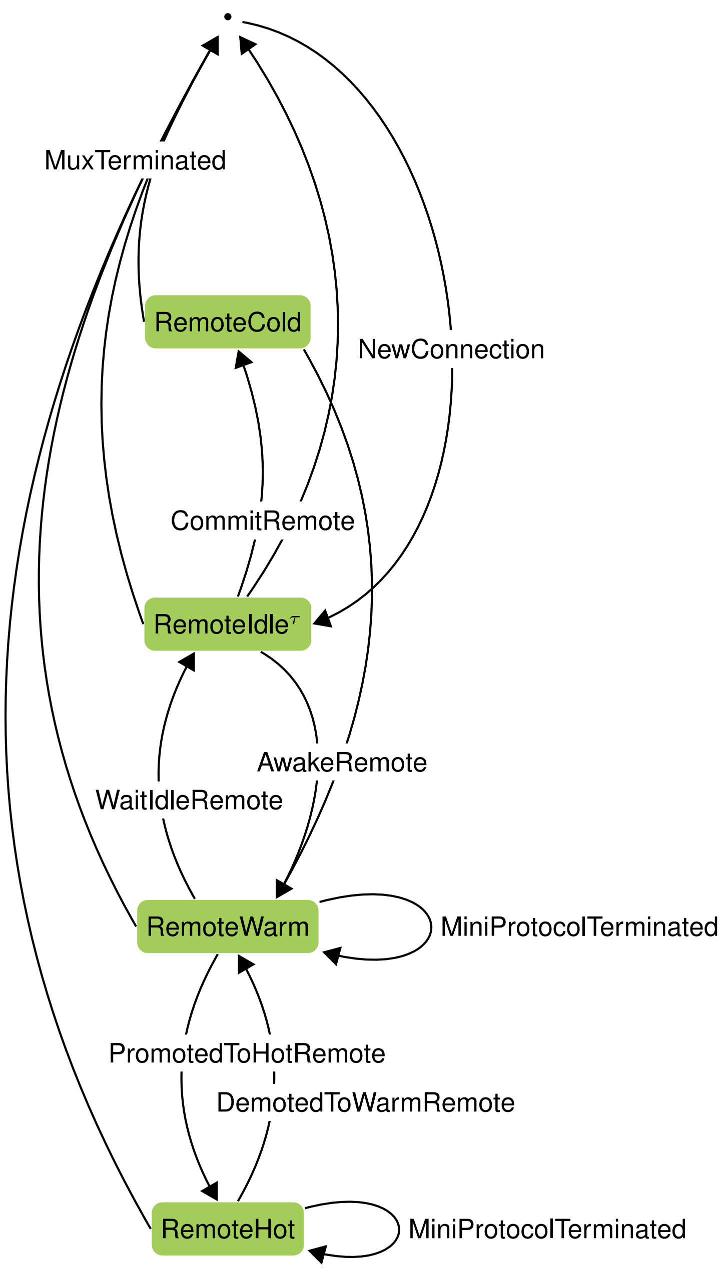


Figure 5.9: Inbound protocol governor state machine

RemoteHot

A connection enters RemoteHot state once any hot responder protocol has started. In particular, if a hot responder is the first to start, the state cycles through RemoteWarm first. Once all hot responders terminate, the connection will be put in RemoteWarm regardless of whether there are any warm or established responders left. In the latter case, if there aren't any other protocols running, the connection will then follow up with further demotion to RemoteIdle$^\tau$.

5.10.2 Transitions

NewConnection

Inbound and outbound duplex connections are passed to the inbound governor. They are then put in $\mathsf{RemoteIdle}^{\tau}$ state.

CommitRemote

Once the $\mathsf{RemoteIdle}^{\tau}$ timeout expires, the inbound governor will call `unregisterInboundConnection`. The connection will either be forgotten or kept in RemoteCold state depending on the returned value.

AwakeRemote

While a connection was put in $\mathsf{RemoteIdle}^{\tau}$ state, it is possible that the remote end will start using it. When the inbound governor detects that any of the responders is active, it will put that connection in RemoteWarm state.

Implementation detail

The inbound governor calls `promotedToWarmRemote` to notify the connection manager about the state change.

WaitIdleRemote

WaitIdleRemote transition happens once all mini-protocol is terminated.

Implementation detail

The inbound governor calls `demotedToColdRemote`. If it returns `TerminatedConnection` the connection will be forgotten (as in MuxTerminated transition), if it returns `OperationSuccess` it will register a idle timeout.

MiniProtocolTerminated

When any of the mini-protocols terminates, the inbound governor will restart the responder and update the internal state of the connection (e.g. update the stm transaction, which tracks the state of the mini-protocol).

Implementation detail

The implementation distinguishes two situations: whether the mini-protocol terminated or errored. The multiplexer guarantees that if it errors, the multiplexer will be closed (and thus, the connection thread will exit, and the associated socket will be closed). Hence, the inbound governor can forget about the connection (perform MuxTerminated).
The inbound governor does not notify the connection manager about a terminating responder mini-protocol.

MuxTerminated

The inbound governor monitors the multiplexer. As soon as it exists, the connection will be forgotten.

The inbound governor does not notify the connection manager about the termination of the connection, as it can detect this by itself.

PromotedToHotRemote

The inbound governor detects when any *hot* mini-protocols have started. In such case a RemoteWarm connection is put in RemoteHot state.

DemotedToWarmRemote

Dually to PromotedToHotRemote state transition, as soon as all of the *hot* mini-protocols terminate, the connection will transition to RemoteWarm state.

Appendix A

Common CDDL definitions

```

; Mini-protocol codecs are polymorphic in various data types, e.g. blocks, points,
; transactions, transaction ids, etc. In CDDL we need concrete values so we
; instantiate them using `any`.  See `CBOR and CDDL` in the network
; technical report
; https://ouroboros-network.cardano.intersectmbo.org/pdfs/network-spec
; if you need further advise how to find concrete encoding of `Cardano` data
; types.

block  = any
header = any
tip    = any
point  = any
; The codec only accepts definite-length list.
points = [ *point ]
txId   = any
tx     = any

; although some of our protocols are polymorphic over slots, e.g.
; `local-tx-monitor`, slots are always encoded as `word64`.
slotNo = word64

word8  = uint .size 1; 1 byte
word16 = uint .size 2; 2 bytes
word32 = uint .size 4; 4 bytes
word64 = uint .size 8; 8 bytes
```

Appendix B

Historical protocol versions

B.1 Node-to-node protocol

Previously supported versions of the *node-to-node protocol* are listed in table B.1.

version	**description**
`NodeToNodeV_1`	initial version
`NodeToNodeV_2`	block size hints
`NodeToNodeV_3`	introduction of keep-alive mini-protocol
`NodeToNodeV_4`	introduction of diffusion mode in handshake mini-protocol
`NodeToNodeV_5`	
`NodeToNodeV_6`	transaction submission version 2
`NodeToNodeV_7`	new keep-alive, Alonzo ledger era
`NodeToNodeV_8`	chain-sync & block-fetch pipelining
`NodeToNodeV_9`	Babbage ledger era
`NodeToNodeV_10`	Full duplex connections
`NodeToNodeV_11`	Peer sharing willingness
`NodeToNodeV_12`	No observable changes
`NodeToNodeV_13`	Disabled peer sharing for buggy V11 & V12 and for InitiatorOnly nodes

Figure B.1: Node-to-node protocol versions

B.2 Node-to-client protocol

Previously supported versions of the *node-to-client protocol* are listed in table B.2.

version	**description**
`NodeToClientV_1`	initial version
`NodeToClientV_2`	added local-query mini-protocol
`NodeToClientV_3`	
`NodeToClientV_4`	new queries added to local state query mini-protocol
`NodeToClientV_5`	Allegra era
`NodeToClientV_6`	Mary era
`NodeToClientV_7`	new queries added to local state query mini-protocol
`NodeToClientV_8`	codec changed for local state query mini-protocol
`NodeToClientV_9`	Alonzo era
`NodeToClientV_10`	GetChainBlock & GetChainPoint queries
`NodeToClientV_11`	GetRewardInfoPools query
`NodeToClientV_12`	Added LocalTxMonitor mini-protocol
`NodeToClientV_13`	Babbage era
`NodeToClientV_14`	GetPoolDistr, GetPoolState, GetSnapshots queries
`NodeToClientV_15`	internal changes

Figure B.2: Node-to-client protocol versions

Bibliography

Harris, T. and Peyton Jones, S. (2006). Transactional memory with data invariants. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*.

Stevens, W., Fenner, B., and Rudoff, A. (2003). *UNIX Network Programming*, volume 1 of *Addison-Wesley Professional Computing Series*. Addison-Wesley Professional. https://learning.oreilly.com/library/view/the-sockets-networking/0131411551.