The Shelley Networking Protocol

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

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

Karl Knutsson
karl.knutsson@iohk.io

Marc Fontaine
marc.fontaine@iohk.io

Armando Santos
armando@well-typed.com

Marcin Szamotulski
marcin.szamotulski@iohk.io

Alex Vieth
alex@well-typed.com

Version 1.3.0, 26th January 2023

Abstract

This document provides technical specification of the implementation of the `ouroboros-network` component of `cardano-node`. It provides specification of all mini-protocols as well 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 are 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

Version history

Version 1.0.0 Nov 2019, State machines and wire format for Ouroboros-Network-1.0.0.

Version 1.1.0 Apr 2021, Connection Manager for Ouroboros-Network-Framework.

Version 1.2.0 Apr 2021, tx-submission version 2, local-state-query, keep-alive mini-protocols

Version 1.3.0 Jul 2021, Review of the multiplexer documentation

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 desiminate transactions across the network.

Figure 1.1 illustrates 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 communicate by means of a 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 to run node-to-node or 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, its framing is 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 of the rate at which transactions are submitted must not cause a decrease of the block chain quality.

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

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

Of cause it makes no sense if the system design is robust, but so defensive that it fails to meet performance goals. An example would be a protocol that never transmits a message unless it has received an explicit ACK for the previous

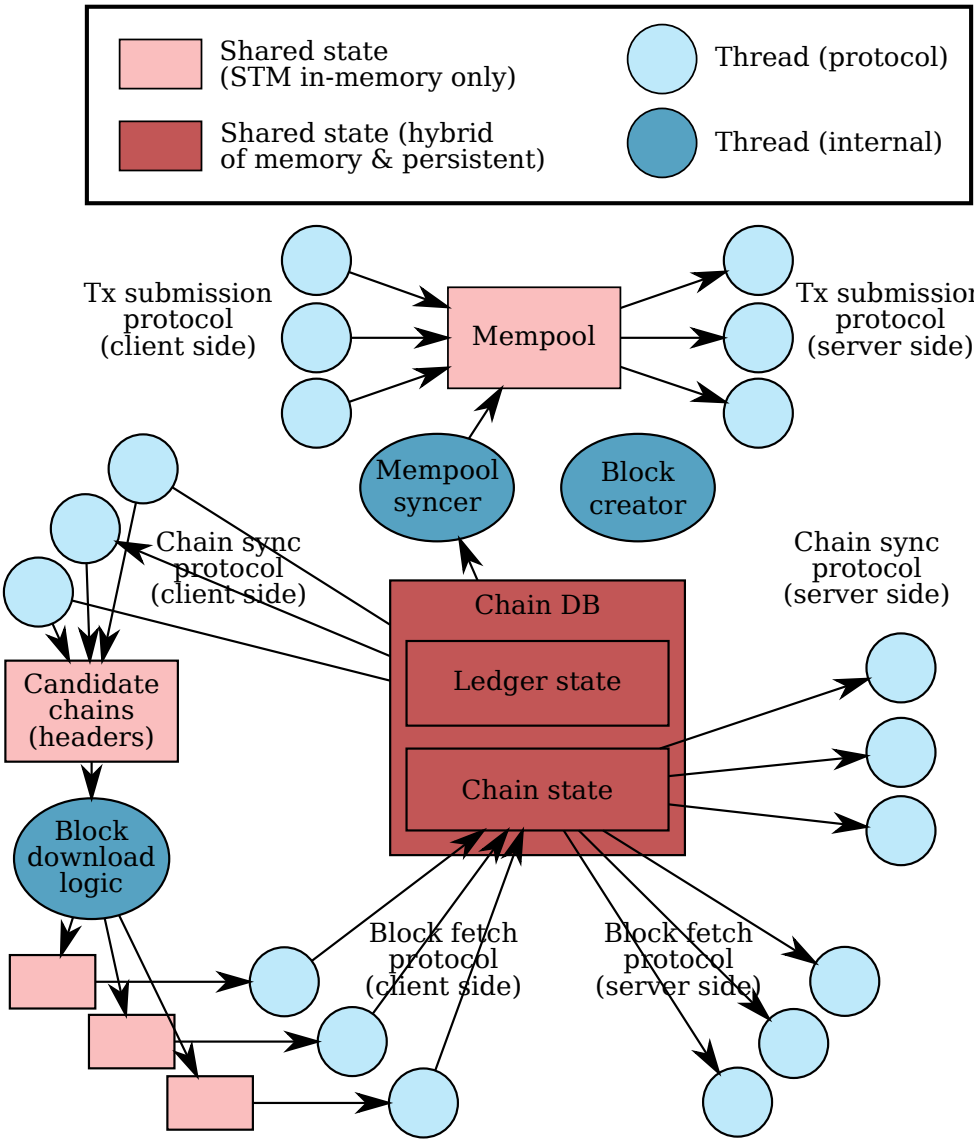


Figure 1.1: Cardano Node

message. This approach might avoid overloading the network, but would waste most of the potential bandwidth. To avoid such performance problems, our implementation is heavily using 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 block chain and transactions that are the basis for the new block, then it creates the new block and sends the new block to its peers. When the new block reaches the next block leader before the beginning of next time slot, the next block leader can extend the block chain upon this block (if the block did not arrive on time the next leader will create a new block anyway).

There are some trade-offs when choosing the slot time that is used for the protocol but basically the slot length should be long enough such that a new block has a good chance to reach the next slot leader in time. 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 behavior of an other node.

Chapter 2

Multiplexing mini-protocols

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.

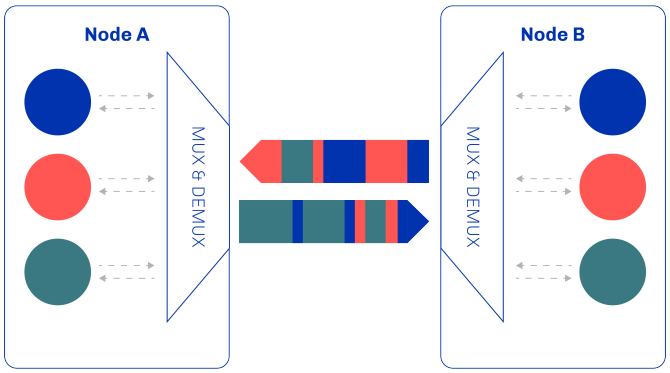


Figure 2.1: Data flow though the multiplexer and de-multiplexer

The multiplexer is agnostic to the bearer it runs over, however it assume that the bearer guarantees an ordered 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 specifies its own framing / binary serialisation format, which is described in section 2.1.1. The multiplexer allows to use each mini-protocol in either direction.

The multiplexer exposes interface which hides all the multiplexer details, a single mini-protocol communication can be written as if it would only communicate with its instance on the remote end. When the muliplexer 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 the bearer. When reading data from the network, segment's headers are used reassemble mini-protocol byte streams.

[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).

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 binary encoding, see Network.Mux.Codec.

2.1.1 Wire Format

Table 2.1 shows the layout of the data segments 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 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 2.2 and 2.3.

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

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

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 implementation of multiplexer uses a round-robin-schedule of the mini protocols to choose the next data segment to transmit. If a mini protocol does not have new data available when it is scheduled, it is skipped. A mini-protocol can transmit at most one segment of data every time it is scheduled and it will only be rescheduled immediately if no other mini protocol 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.

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 3.13.). 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.

Specify ingress buffer sizes for each mini-protocol

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 a inter process communication, usedy by clients, e.g. a wallet, db-sync, etc. Each of them consists of a bundle of mini-protocols (see chapter 3) consists of a bundle of mini-protocols. The protocol numbers of both protocols are specified in tables 2.2 and 2.3.

mini-protocol	mini-protocol number
Handshake	0
Chain-Sync (instantiated to headers)	2
Block-Fetch	3
TxSubmission	4
Keep-alive	8

Table 2.2: Node-to-node protocol numbers

mini-protocol	mini-protocol number
Handshake	0
Chain-Sync (instantiated to blocks)	5
Local TxSubmission	6
Local State Query	7

Table 2.3: Node-to-client protocol numbers

Chapter 3

Mini Protocols

3.1 Mini Protocols and Protocol Families

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

The mini protocols in this section describe both the initiator and responder of a communication. The initiator is the dual of the responder and vice versa. (The terms client/server and consumer/producer are also used sometimes.) At any time a node will typically run many instances of mini protocols, including many instances of the same mini protocol. Each mini protocol instance of the node communicates with the dual instance of exactly one peer.

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

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 the protocol and the implementation. In particular, it guarantees that there are no deadlocks. At any time, one side has agency (is expected to transmit the next message) and the other side is awaiting for the message (or both sides agree that the protocol has terminated). If either side receives a message that is not expected according to the protocol the communication is aborted.

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

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

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

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

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

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

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

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

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

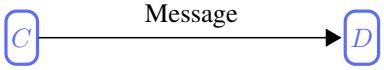
$$\{(label, data) \mid label \in Labels, data \in Data\}$$

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

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

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

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 at the server.

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

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

3.3 Overview of all implemented Mini Protocols

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 to exchanges data.
typed-protocols/src/Network/TypedProtocol/ReqResp/Type.hs

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

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

Transaction Submission Protocol v2 Section 3.9.2
A Protocol for transmitting transaction between core nodes.
ouroboros-network/src/Ouroboros/Network/Protocol/TxSubmission2/Type.hs

Keep Alive Protocol Section 3.10
A protocol for sending keep alive messages and round trip measurments
ouroboros-network/src/Ouroboros/Network/Protocol/KeepAlive/Type.hs

Locat State Query Mini Protocol Section 3.12
Protocol used by local clients to query ledger state
ouroboros-network/src/Ouroboros/Network/Protocol/LocalStateQuery/Type.hs

3.4 CBOR and CDDL

All mini-protocols are encoded using concise binary object representation (CBOR), see https://cbor.io. Each codec comes along with a specification written in CDDL, see https://cbor-wg.github.io/cddl/draft-ietf-cbor-cddl.html.

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.

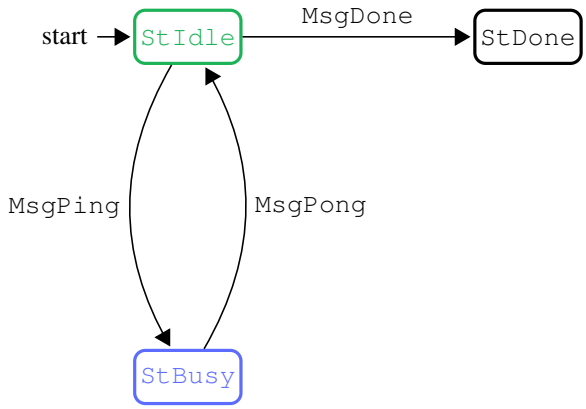
3.5.1 Ping-Pong mini-protocol

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



Agency	
Client has Agency	`StIdle`
Server has Agency	`StBusy`

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.

Transition table		
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

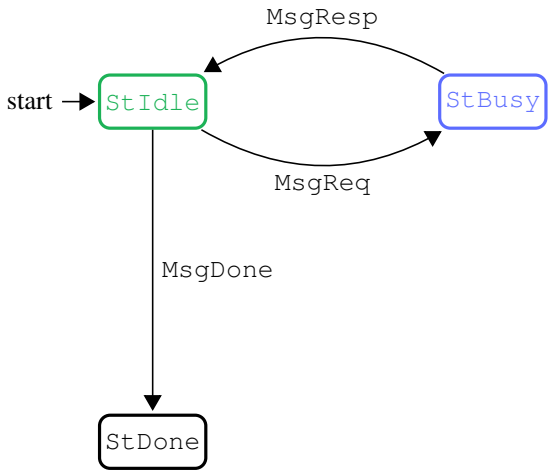
haddock documentation: `Network.TypedProtocol.ReqResp.Type`

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 the requests and responses.

State machine

Agency	
Client has Agency	`StIdle`
Server has Agency	`StBusy`



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.

Transition table			
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

haddock documentation: `Ouroboros.Network.Protocol.Handshake.Type`

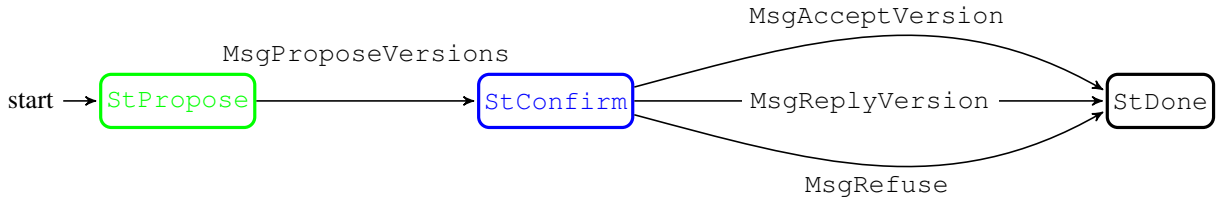
3.6.1 Description

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

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

Agency	
Client has Agency	StPropose
Server has Agency	StConfirm



3.6.2 State machine

Messages of the protocol:

MsgProposeVersions $(versionTable)$ The client proposes a number of possible versions and protocol parameters.

MsgReplyVersion $(versionTable)$ In TCP simultaneous open the client will receive MsgReplyVersion (which was sent as MsgProposeVersions) as a reply to its own MsgProposeVersions; thus both MsgProposeVersions and MsgReplyVersion have to have the same CBOR encoding.

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

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

Transition table			
from	message/event	parameters	to
StPropose	MsgProposeVersions	$versionTable$	StConfirm
StConfirm	MsgReplyVersion	$versionTable$	StDone
StConfirm	MsgAcceptVersion	$(versionNumber, extraParameters)$	StDone
StConfirm	MsgRefuse	$reason$	StDone

3.6.3 Client and Server Implementation

Section 3.6.4 contains the CDDL-specification of the binary format of the handshake messages. The version table is encoded as a CBOR table with the version number as key and the protocol parameters as value. The handshake protocol requires that the version numbers (i.e. the keys) in the version table are unique and appear in ascending order. (Note, that CDDL is not expressive enough to precisely specify that requirement on the keys of the CBOR table. Therefor the CDDL-specification uses a table with keys from 1 to 4 as an example.)

In a run of the handshake mini protocol the peers exchange only two messages: The client requests to connect with a 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 extra parameters 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 support 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: Encode the extra parameters 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 set of protocols that the server supports. The handshake protocol is designed, such 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 version.

In case of simultaneous open of a TCP connection, both handshake clients will send their `MsgProposeVersions`, 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) it can reset the connection.

The protocol does not forbid, nor could it detect a usage of `MsgReplyVersion` outside of TCP simultaneous open. The process of choosing between 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 proposed and received data, where

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

See ref. Both `acceptable proposed received` and `acceptable received proposed` must satisfy the following condition:

- if either of them accepts a version by returning `Accept`, the other one must accept the same value, i.e. in this case `acceptable proposed received == acceptable received proposed`
- if either of them refuses to accept (returns `Refuse reason`) the other one shall 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 returns 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 at the same state (once all in-flight message arrived). This is no longer true in general, however this is still true for the handshake protocol. Event though the opening message `MsgProposeVersions` of a simultaneous open, will materialise on the other side as termination message `MsgReplyVersions`, the same will happen to the `MsgProposeVersion` transmitted in the other direction. We include a special test case (`prop_channel_simultaneous_open`) to verify that simultaneous open well behaves and does not lead to protocol errors.

The handshake mini protocol runs before the MUX/DEMUX itself is initialised. Each message is transmitted within a single MUX segment, i.e. with a proper segment header, but, as the MUX/DEMUX is not yet running the messages must not be split into multiple segments. These MUX segments are using a reserved protocol id 0 (`Muxcontrol`).

3.6.4 CDDL encoding specification

There are two flavours of the mini-protocol which 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 the other by node to client protocol.

Node to node handshake mini-protocol

```
;
; NodeToNode Handshake, v4 or higher
;

handshakeMessage
    = msgProposeVersions
    / msgAcceptVersion
    / msgRefuse

msgProposeVersions = [0, versionTable]
msgAcceptVersion   = [1, versionNumber, nodeToNodeVersionData]
msgRefuse          = [2, refuseReason]

versionTable = { * versionNumber => nodeToNodeVersionData }

versionNumber = 7 / 8 / 9 / 10

nodeToNodeVersionData = [ networkMagic, initiatorAndResponderDiffusionMode ]

; range between 0 and 0xffffffff
networkMagic = 0..4294967295
initiatorAndResponderDiffusionMode = bool

refuseReason
    = refuseReasonVersionMismatch
    / refuseReasonHandshakeDecodeError
    / refuseReasonRefused

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

Node to client handshake mini-protocol

```
;
; NodeToClient Handshake
;

handshakeMessage
    = msgProposeVersions
    / msgAcceptVersion
    / msgRefuse

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

versionTable = { * versionNumber => nodeToClientVersionData }

; as of version 2 (which is no longer supported) we set 15th bit to 1
;                9     / 10    / 11    / 12    / 13    / 14
versionNumber = 32777 / 32778 / 32779 / 32780 / 32781 / 32782
```

```
19
20 nodeToClientVersionData = networkMagic
21
22 networkMagic = uint
23
24 refuseReason
25     = refuseReasonVersionMismatch
26     / refuseReasonHandshakeDecodeError
27     / refuseReasonRefused
28
29 refuseReasonVersionMismatch       = [0, [ *versionNumber ] ]
30 refuseReasonHandshakeDecodeError = [1, versionNumber, tstr]
31 refuseReasonRefused               = [2, versionNumber, tstr]
```

3.7 Chain-Sync mini-protocol

haddock documentation: `Ouroboros.Network.Protocol.ChainSync.Type`

3.7.1 Description

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

The chain synchronisation protocol is polymorphic. The (full)-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 transfer full blocks.

3.7.2 State Machine

Agency	
Client has Agency	`StIdle`
Server has Agency	`StCanAwait`, `StMustReply`, `StIntersect`

The protocol uses the following messages:

`MsgRequestNext` Request the next update from the producer.

`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.

`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 producers 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 perference (e.g. points with 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 send with `MsgFindIntersect` the server will reply with `MsgIntersectNotFound`.

`MsgIntersectFound` $(\boldsymbol{point_{intersect}}, \boldsymbol{tip})$ The producer replies with the first point of the request that 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 replay to `MsgRequestNext`) to be `MsgRollBackward` to the specified $point_{intersect}$ (which makes handling state updates on the client side easier).

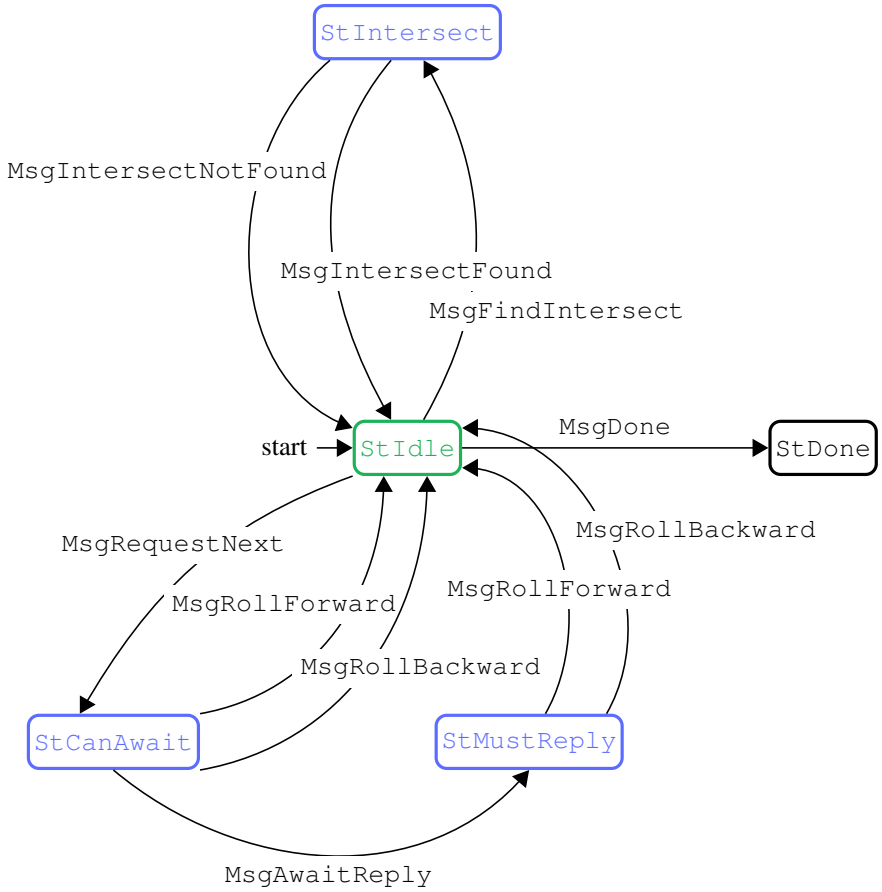


Figure 3.1: State machine of the Chain-Sync mini-protocol.

`MsgIntersectNotFound` (tip) The 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.

3.7.3 Implementation of the Chain Producer

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

Transition table			
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`	$header_{old}, tip$	`StIdle`
`StMustReply`	`MsgRollForward`	$header, tip$	`StIdle`
`StMustReply`	`MsgRollBackward`	$point_{old}, tip$	`StIdle`
`StIntersect`	`MsgIntersectFound`	$point_{intersect}, tip$	`StIdle`
`StIntersect`	`MsgIntersectNotFound`	tip	`StIdle`

Table 3.3: Chain-Sync mini-protocol messages.

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 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.

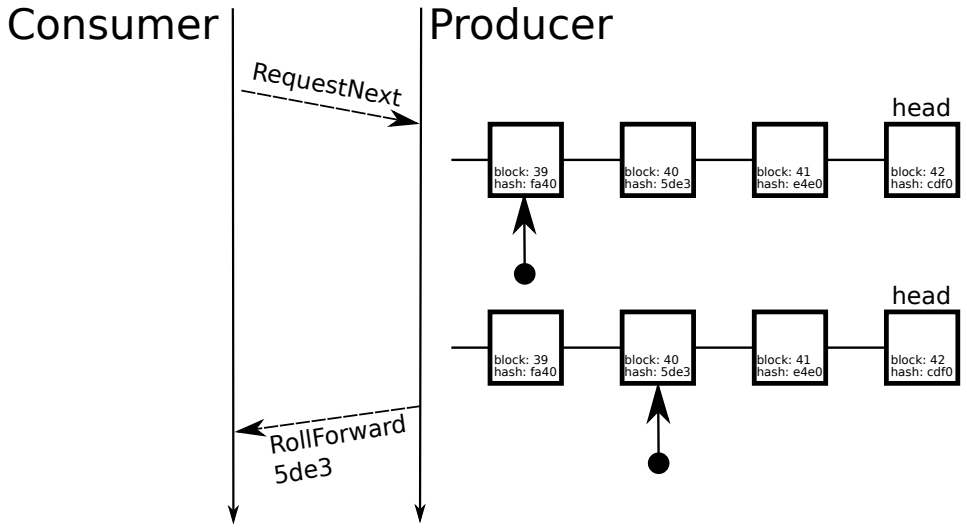


Figure 3.2: Consumer driven block download.

Producer driven updates If the *read-pointer* points to the end of the chain and the producer receives a `MsgRequestNext` the consumers 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 just 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 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.

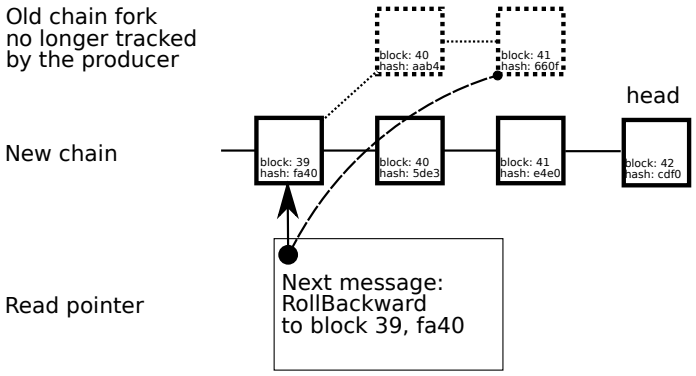


Figure 3.3: *read-pointer* update for a fork switch in case of a rollback.

Figure 3.3 illustrates a fork switch that requires an update of the *read-pointer* for one of the chain consumers, i.e. an example for case 2. Before the switch, the *read-pointer* of the consumer points to block $0x660f$. The producer switches to a new chain with the head of the chain at block $0xcdf0$. The node must update the *read-pointer* to block $0xfa40$ and the next message to the consumer will be a `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 block chain 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 seems beneficial. If the consumer does not know anything about the producer's chain, it can start the search with the following list of points: $[point(b), point(b-1), point(b-2), point(b-4), point(b-8), \ldots]$ where $point(b-i)$ is the point of the ith predecessor of block b and b is the head of the consumer fork. 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*.

3.7.4 Implementation of the Chain Consumer

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

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

Here are some cases to consider:

`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, just switch to a longer chain fork.

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

3.7.5 CDDL encoding specification

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

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

wrappedHeader = #6.24(bytes .cbor blockHeader)
tip = [point, uint]

points = [ *point ]
```

See appendix A for common definitions.

3.8 Block-Fetch mini-protocol

haddock documentation: `Ouroboros.Network.Protocol.BlockFetch.Type`

3.8.1 Description

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

Agency	
Client has Agency	`StIdle`
Server has Agency	`StBusy`, `StStreaming`

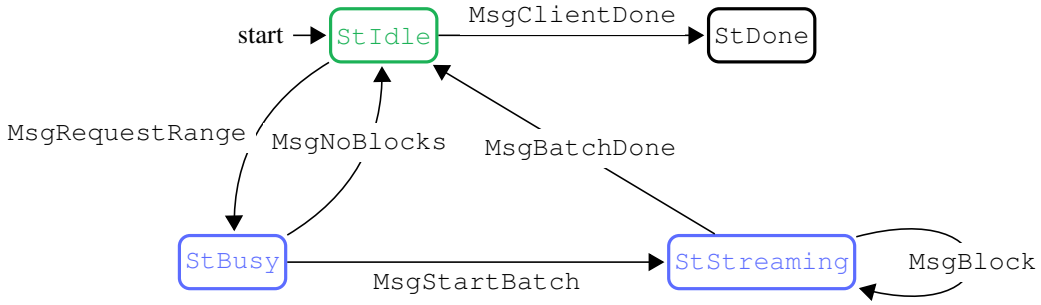


Figure 3.4: State machine of the block-fetch mini-protocol.

3.8.2 State machine

Protocol messages

`MsgRequestRange` $(range)$ The client requests a $range$ of blocks from the server.

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

`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.

Transition table is shown in table 3.4.

Transition table			
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 3.4: Block-Fetch mini-protocol messages.

3.8.3 CDDL encoding specification

```
;
; BlockFetch mini-protocol
;

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

```
7
8  blockFetchMessage
9        = msgRequestRange
10       / msgClientDone
11       / msgStartBatch
12       / msgNoBlocks
13       / msgBlock
14       / msgBatchDone
15
16 msgRequestRange = [0, point, point]
17 msgClientDone   = [1]
18 msgStartBatch   = [2]
19 msgNoBlocks     = [3]
20 msgBlock        = [4, #6.24(bytes .cbor block)]
21 msgBatchDone    = [5]
```

See appendix A for common definitions.

3.9 Tx-Submission mini-protocol

3.9.1 Version 1

Version 1 of the tx-submission protocol is no longer supported.

3.9.2 Version 2

haddock documentation: `Ouroboros.Network.Protocol.TxSubmission2.Type`

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, which is used when the server trusts a local client, is described in Section 3.11.

The version 2 is used by version 6 an higher of node-to-node protocol.

State machine

Agency	
Client has Agency	`StInit`, `StTxIdsBlocking`, `StTxIdsNonBlocking`
Server has Agency	`StIdle`, `StTxs`

Protocol messages

`MsgInit` initial message of the protocol

`MsgRequestTxIdsBlocking` (ack, req) The server asks for new transaction ids and acknowledges old ids. The client will block until new transactions are available.

`MsgRequestTxIdsNonBlocking` (ack, req) The server asks for new transaction ids and acknowledges old ids. The client immediately replies (possible with an empty list).

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

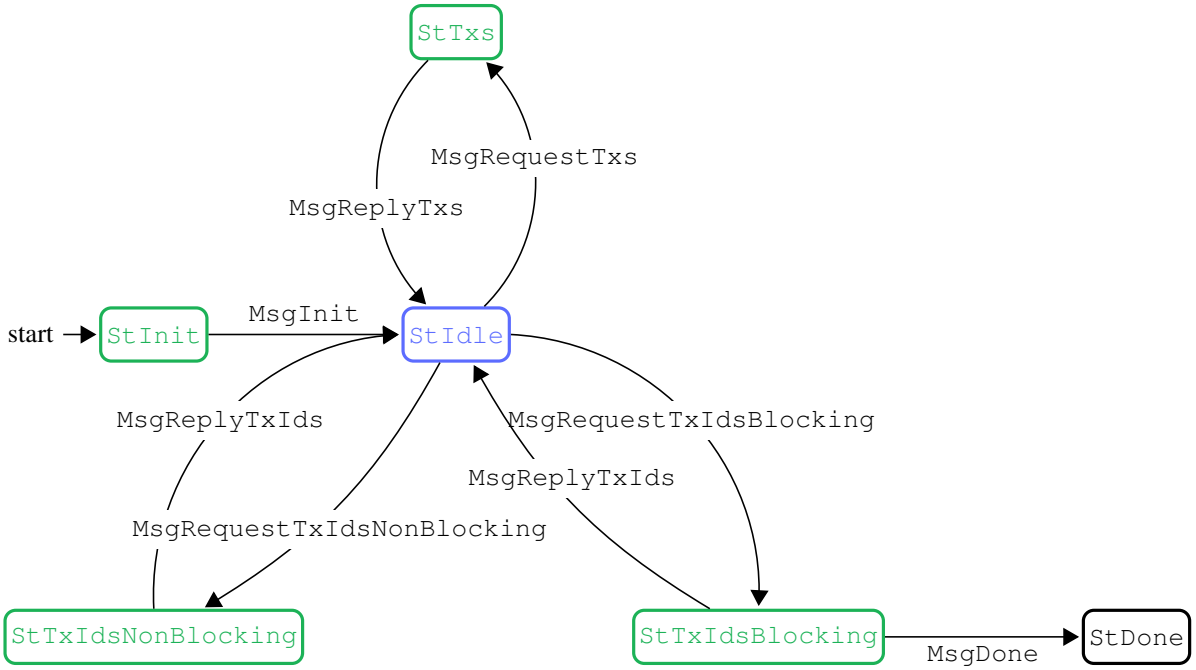


Figure 3.5: State machine of the Tx-Submission mini-protocol (version 2).

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

`MsgReplyTxs` ($[txs]$) The client replies with a list transaction.

`MsgDone` The client terminates the mini protocol.

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

Table 3.5: Tx-Submission mini-protocol (version 2) messages.

CDDL encoding specification

```
;
; TxSubmission mini-protocol v2
;

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

txSubmission2Message
```

```
9       = msgInit
10      / msgRequestTxIds
11      / msgReplyTxIds
12      / msgRequestTxs
13      / msgReplyTxs
14      / tsMsgDone
15
16
17  msgInit          = [6]
18  msgRequestTxIds  = [0, tsBlocking, txCount, txCount]
19  msgReplyTxIds    = [1, [ *txIdAndSize] ]
20  msgRequestTxs    = [2, tsIdList ]
21  msgReplyTxs      = [3, tsIdList ]
22  tsMsgDone        = [4]
23
24  tsBlocking       = false / true
25  txCount          = word16
26  ; The codec only accepts infinite-length list encoding for tsIdList !
27  tsIdList         = [ *txId ]
28  txIdAndSize      = [txId, txSizeInBytes]
29  txSizeInBytes    = word32
```

See version 1 of the mini-protocol in section **??** and appendix A for common definitions.

3.9.3 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 of a given id 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 for requesting transaction ids and transactions. One possible implementation for a consumer is to maintain a FIFO which mirrors the producers 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 still require a 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 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 then the requested number of ids (possible 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

haddock documentation: `Ouroboros.Network.Protocol.KeepAlive.Type`

3.10.1 Description

Keep alive mini-protocol is a member of 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

Agency	
Client has Agency	`StClient`
Server has Agency	`StServer`

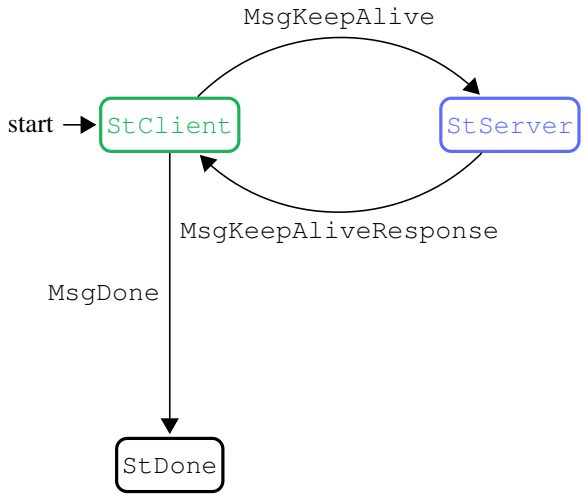


Figure 3.6: State machine of the keep alive protocol.

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`.

`MsgKeepAliveResponse` *cookie* Keep alive response message.

`MsgDone` Terminating message.

3.10.3 CDDL encoding specification

```
;
; KeepAlive Mini-Protocol
;

keepAliveMessage = msgKeepAlive
                 / msgKeepAliveResponse
                 / msgDone

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

word16 = 0..65535
```

3.11 Local Tx-Submission mini-protocol

haddock documentation: `Ouroboros.Network.Protocol.LocalTxSubmission.Type`

3.11.1 Description

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

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 is protocol is only for use between a node and local client.

3.11.2 State machine

Agency	
Client has Agency	`StIdle`
Server has Agency	`StBusy`

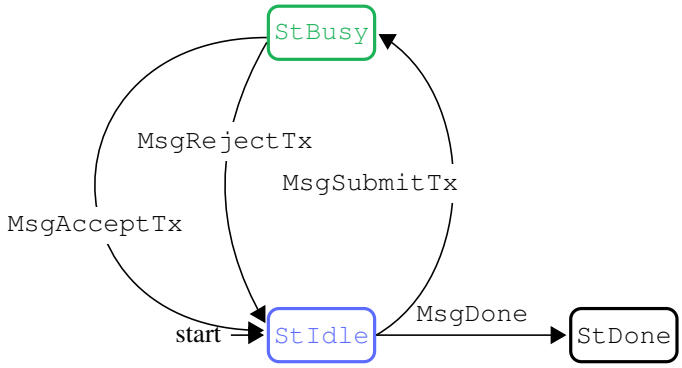


Figure 3.7: State machine of the Local Tx-Submission mini-protocol.

Protocol messages

`MsgSubmitTx` (t) The client submits a transaction.

`MsgAcceptTx` The server accepts the transaction.

`MsgRejectTx` $(reason)$ The server rejects the transactions and replies with the $reason$.

`MsgDone` The client terminates the mini protocol.

3.12 Local State Query mini-protocol

haddock documentation: `Ouroboros.Network.Protocol.LocalStateQuery.Type`

3.12.1 Description

Local State Query mini-protocol allows to query 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.

3.12.2 State machine

Agency	
Server has Agency	`StIdle`, `Acquired`
Client has Agency	`Acquiring`, `Querying`

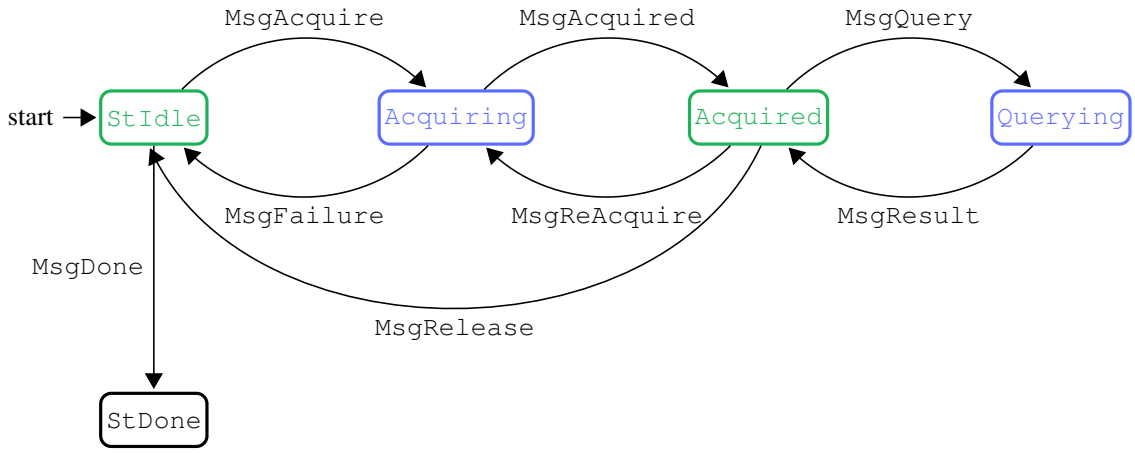


Figure 3.8: State machine of the Local State Query mini-protocol.

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

Figure 3.9: Local State Query mini-protocol messages.

Protocol messages where $AcquireFailure$ is either $AcquireFailurePointTooOld$ or $AcquireFailurePointNotOnChain$.

3.12.3 CDDL encoding specification

```
;
; LocalStateQuery mini-protocol.
;
```

```
4
5  localStateQueryMessage
6    = msgAcquire
7    / msgAcquired
8    / msgFailure
9    / msgQuery
10   / msgResult
11   / msgRelease
12   / msgReAcquire
13   / lsqMsgDone
14
15 acquireFailurePointTooOld      = 0
16 acquireFailurePointNotOnChain = 1
17
18 failure       = acquireFailurePointTooOld
19               / acquireFailurePointNotOnChain
20
21 ; 'query' and 'result' encodings are not specified; The values are only used
22 ; for compatibility with
23 ; 'Ouroboros.Network.Protocol.LocalStateQuery.Test.codec'
24 query         = null
25 result        = []
26               / [point]
27
28 msgAcquire    = [0, point]
29               / [8]
30 msgAcquired   = [1]
31 msgFailure    = [2, failure]
32 msgQuery      = [3, query]
33 msgResult     = [4, result]
34 msgRelease    = [5]
35 msgReAcquire = [6, point]
36               / [9]
37 lsqMsgDone    = [7]
```

See appendix A for common definitions.

3.13 Pipelining of Mini Protocols

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

As an example, let's consider the Block-Fetch mini protocol. When a client follows the protocol and sends a sequence of `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 whether the client uses pipelining or not. The MUX/DEMUX layer (Section 2.1) guarantees that messages of the same mini protocol are delivered in transmission order, and therefore the client can determine which response belongs to which request.

The MUX/DEMUX layer also provides a fixed size buffer between the egress of DEMUX and the ingress of mini protocol thread. The size of this buffer is a protocol parameter that determines how many messages a client can send before waiting for a reply from the server (see Section 2.1.3). The protocol requires that a client must never cause a overrun of these buffers on a server node. If a message arrives at the server that would cause the buffer to overrun, the server treats this case as a protocol violation of the peer (and closes the connection to the peer).

3.14 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 **??**)
- keep alive mini-protocol; from $NodeToNodeV\_3$ (section 3.10)

Currently supported versions of the *node-to-node protocol* are listed in table 3.10.

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

Figure 3.10: Node-to-node protocol versions

3.15 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 3.11)
- *local-state-query mini-protocol*; from version $NodeToClient\_2$ (section 3.12)

Supported versions of *node-to-client protocol* are listed in table 3.11.

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
$NodeToClientV\_6$	mary
$NodeToClientV\_7$	new queries added to local state query mini-protocol
$NodeToClientV\_8$	codec changed for local state query mini-protocol

Figure 3.11: Node-to-client protocol versions

Chapter 4

Connection Manager State Machine Specification

4.1 Introduction

As described in the Network Design document, the goal is to transition to a more decentralized network. To make that happen, a plan was designed to come up with a P2P network that is capable to achieve desired network properties. One key component of such design is the *p2p governor*, which is responsible for managing the *cold*/*warm*/*hot* peer selection and 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 2 instances (client and server). Which 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.

4.2 Components

Figure 4.1 illustrates the 3 main components of the decentralization 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 to detect if its local peer was added as a *warm*/*hot* peer in some other remote node, starting/restarting the required mini-protocols. Another role of the `Inbound Protocol Governor` is to setup timers in some cases, e.g. if the remote end opened a connection, and did not sent any message, the `Inbound Protocol Governor` will timeout after some time and close the connection. The arrows in Figure 4.1 represent dependencies between components: 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).

One simple illustration of how these 3 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 $\texttt{InboundIdleState}^{\tau}$ `Duplex`;
- The remote end transitions the local node to warm (using the connection) within expected timeout;

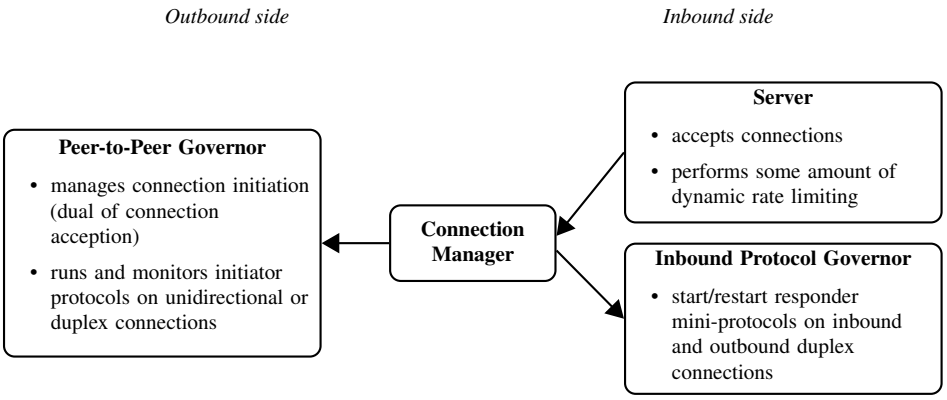


Figure 4.1: 3 main components

- 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 section 4.3.3.

Figure 4.2 shows the high-level architecture of how the 3 components mentioned interact with each other. A single *Connection manager* is shared between the *Server* and *p2p governor*, where, in case of an *Outbound Duplex* connection is negotiated, the *Server* is notified via a control channel. Although in this document we will use Server and IPG interchangeably, it is worth to keep them separate concepts for possible future developments.

4.3 Connection Manager

4.3.1 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).

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 outcome of the handshake negotiation is:

- the negotiated version of the protocol

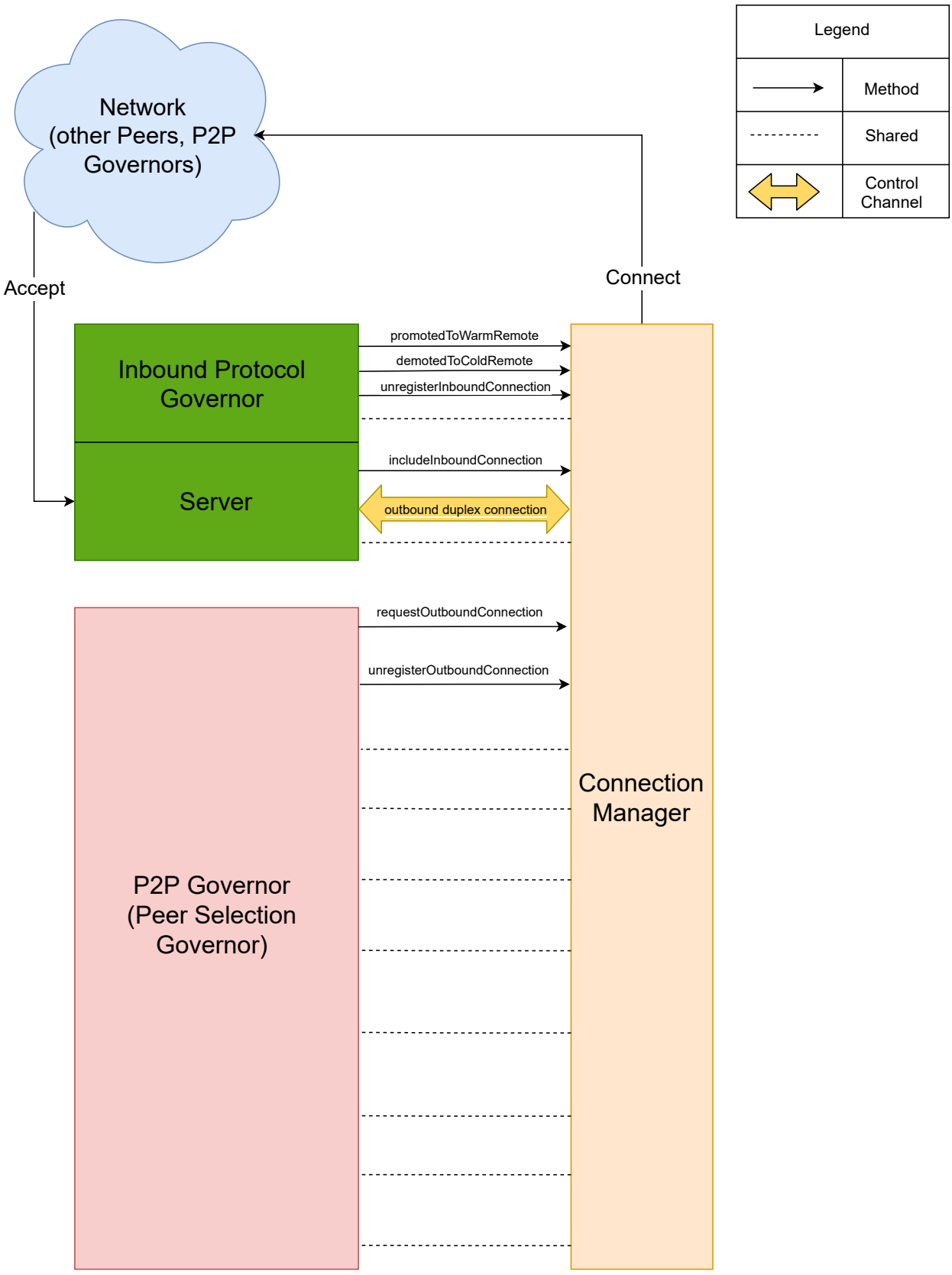


Figure 4.2: High-level architecture of how the 3 components interact

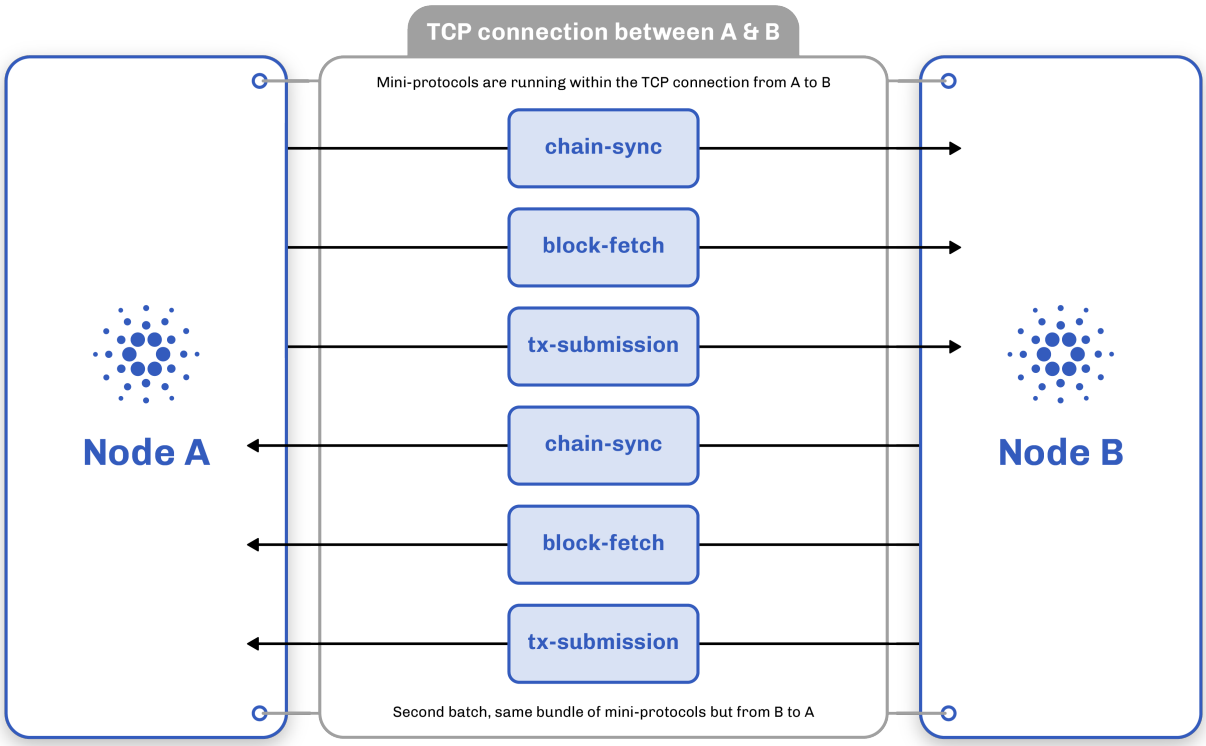


Figure 4.3: Duplex connection running severall mini-protocols

- negotiated parameters, which includes 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 4.3 shows 6 mini protocols running, 3 in each direction. If we negotiated only a unidirectional connection, then we'd only be running 3 (with the direction being 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 which run either the initiator or responder side of mini-protocols, exclusively, 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.

4.3.2 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 handshake negotiation .
  | Disconnected !(ConnectionId peerAddr) !(Maybe handleError)

-- | Include outbound connection into '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 to register an inbound connection, which was `accepted`. Both methods are blocking 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 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 successul.
--
-- 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
```

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

```
data ConnectionState
  -- Connection manger is about to connect to 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
  -- 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 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 4.4 shows all the transitions between `ConnectionStates`. Blue and Violet states represent states of an *Outbound* connection, 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.

Note that the vertical symmetry in the graph corresponds to local vs remote state of the connection, see table 4.1. 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 start initiator-protocols, and the implementation is supposed to do that promptly. This however, cannot be assumed about the inbound side.

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

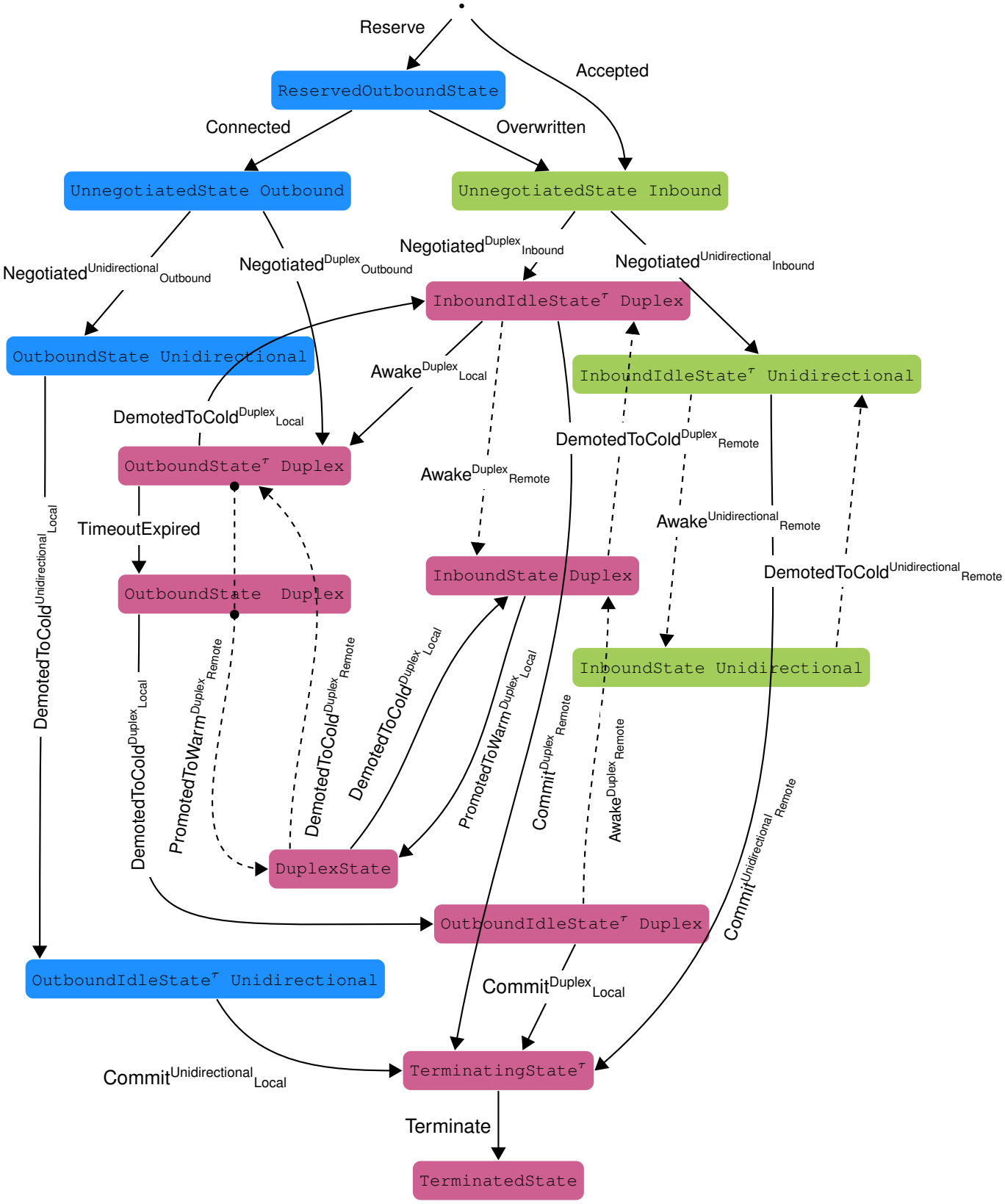


Figure 4.4: *Outbound* (blue & violet) and *inbound* (green & violet) connection states and allowed transitions.

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

Table 4.1: Symmetry between local and remote states

(in the sense that inbound connection 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 4.5. 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.

4.3.4 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 performed 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 latter is captured by the Overwritten transition.

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 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.

the exact version number might change

If 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`.

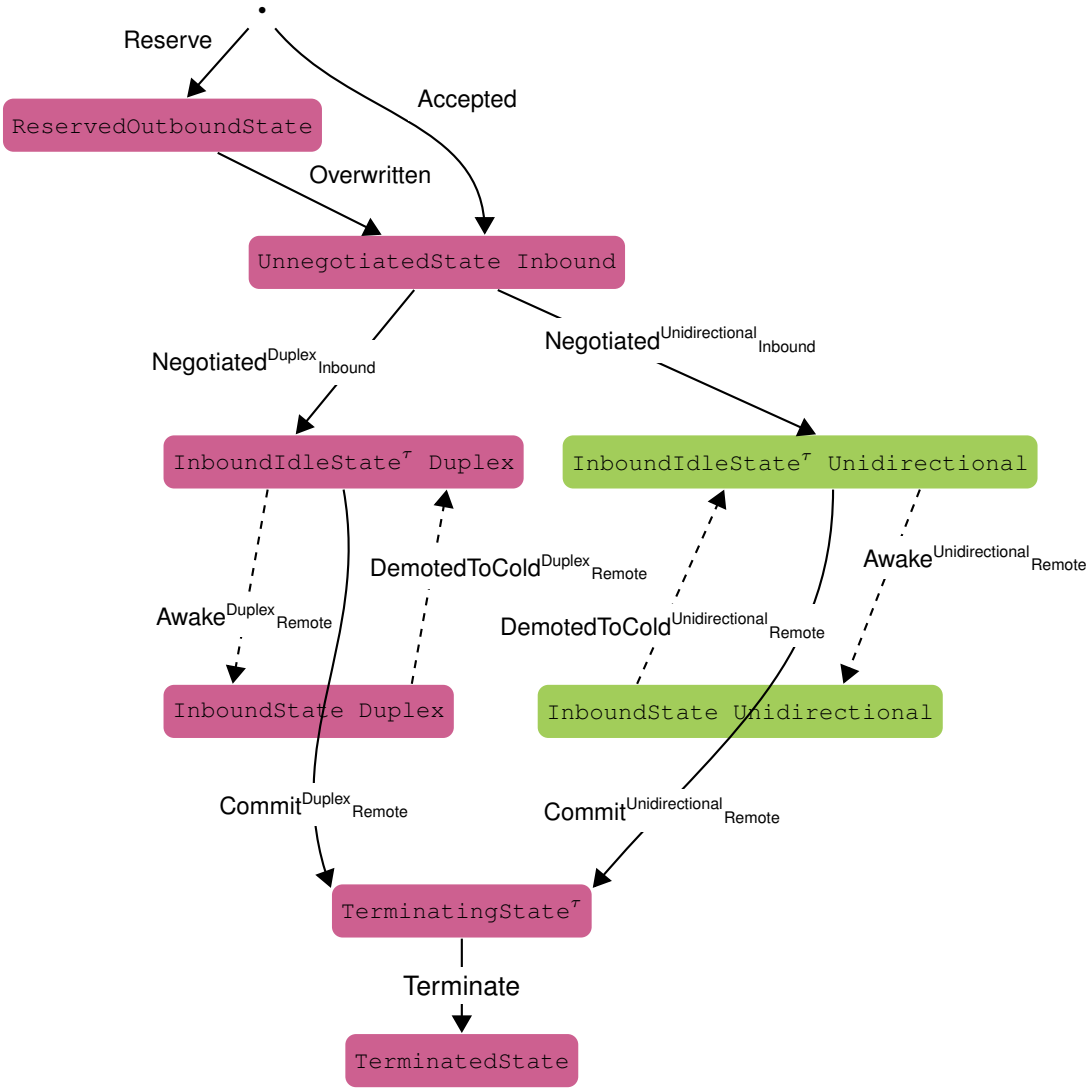


Figure 4.5: Sub-graph of inbound states.

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

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

For $\text{Negotiated}^{\text{Unidirectional}}_{\text{Inbound}}$, $\text{Negotiated}^{\text{Duplex}}_{\text{Inbound}}$, $\text{Negotiated}^{\text{Duplex}}_{\text{Outbound}}$ 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.*

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

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

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

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

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

These transitions are triggered by 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:

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

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 $\text{Awake}^{\text{Duplex}}_{\text{Remote}}$ or $\text{Awake}^{\text{Unidirectional}}_{\text{Remote}}$ transition is performed. The idleness detection might also be interrupted by the local $\text{Awake}^{\text{Duplex}}_{\text{Local}}$ 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 is relaying 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 do that also 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 starts mini-protocols, as they are already running (we restart responders as soon as the stop, using the on-demand strategy).*

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

This transitions 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 connection manager will do Commit$^{dataFlow}_{Local}$ transition, which will reset the connection.

Implementation detail

This transition is done by `unregisterOutboundConnection`.

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

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 4.3.4 for details.

Implementation detail

Both transitions are done by `demotedToColdRemote`.

PromotedToWarm$^{Duplex}_{Local}$

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 DemotedToCold$^{dataFlow}_{Remote}$ transition. The connection manager tracks the state of this timeout so we can decide if a connection in outbound state can terminate or it needs to await for that timeout to expire.

Implementation detail

This transition is done by `unregisterInboundConnection`.

PromotedToWarm$^{Duplex}_{Remote}$

This asynchronous transition is triggered by the remote peer. The *inbound protocol governor* can notice it by observing 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 server expects an initial message;*
> - *all mini-protocols are started using 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 have not 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:

$$\text{Connected},$$

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

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

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

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 end up exceeding server hard limit.

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

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

To solve this problem, in any of the above transitions the connection manager will check if the server hard limit was exceeded. If that happened, 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, but 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 open 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 a 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 and there might not be any other connection available to prune that can make space for this one.

The *inbound protocol governor* is in 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[1] 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 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 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 inbound protocol governor. The implementation is relying 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 do that also 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 starts mini-protocols, as they are already running (we restart responders as soon as the 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 trigged after *protocol idle timeout*, but this time are triggered on the outbound side. These transition will reset the connection, and the timeout make sure that the remote end will be able to clear its ingress queue before the TCP reset arrives. For a more detailed analysis see 4.3.6 section.

[1] 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.

Terminate

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

Add a haddock link to `daTimeWaitTimeout`

4.3.5 Protocol errors

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

4.3.6 Closing connection

By default when 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 power to the remote end to keep resources for 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 to close each side of the connection separately. With 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 is able to read from ingress buffers before the `RST` packet arrives. Excluding protocol errors and prune transitions, which uncooperatively reset the connection.

Add haddock link to `daProtocol`

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 responsibility of the client to not re-connect too early. For example, *p2p governor* enforces 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 important 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.

4.3.7 *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 which is in any other state, e.g. `UnnegotiatedState Outbound`, `OutboundState dataFlow`, `DuplexState`, *connection manager* signals the caller with an error, see section 4.2.

Figure 4.6 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 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`. Server is using this channel for incoming duplex outbound and all inbound connections.*

Termination

When *p2p governor* demotes a peer to *cold* state, an outbound 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:

unregisterOutboundConnection :: peerAddr $\rightarrow$ m ()

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

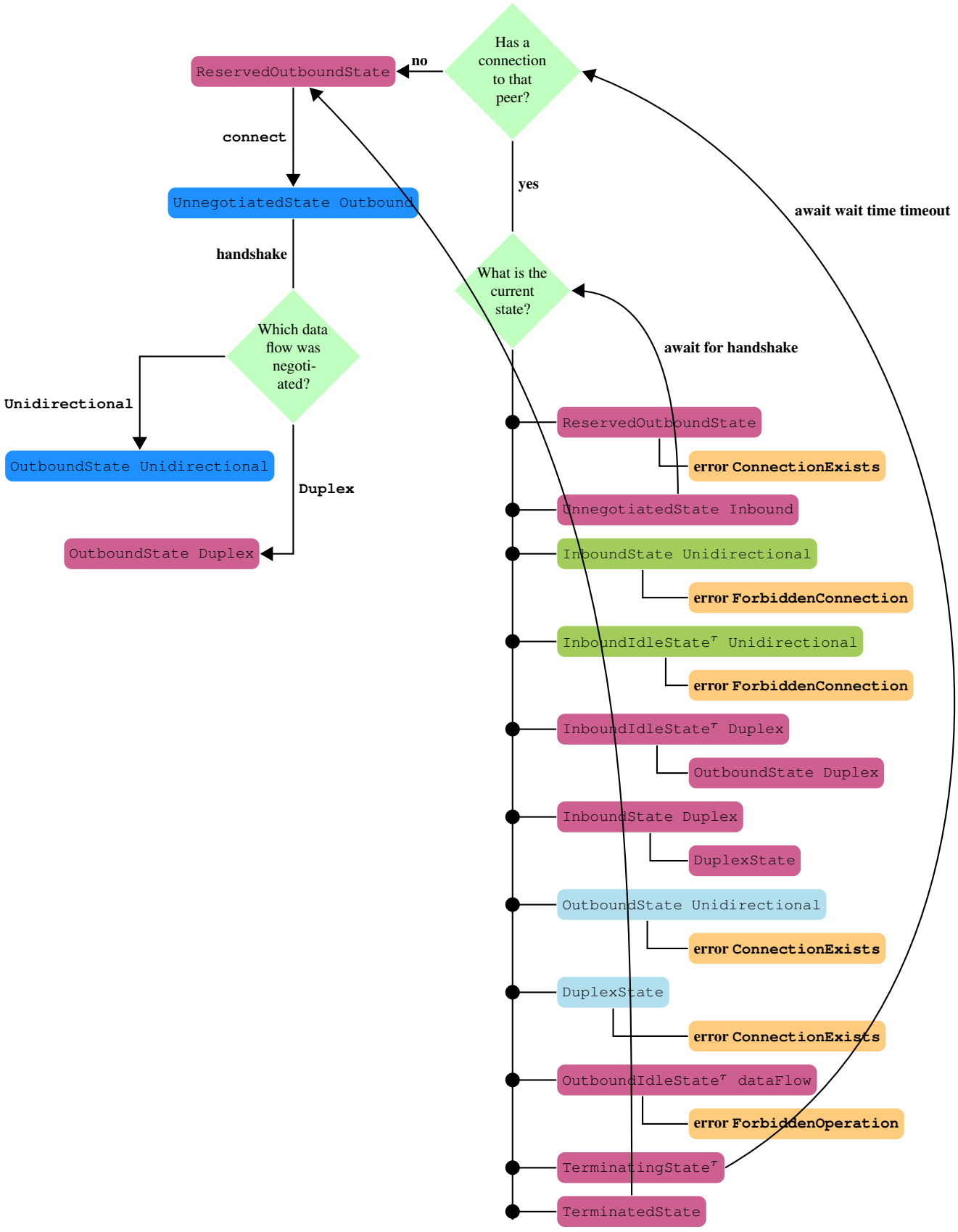


Figure 4.6: *Outbound* connection flow graph

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`
$\texttt{OutboundState}^{\tau}$ `Duplex`	error `ConnectionExists`
$\texttt{OutboundIdleState}^{\tau}$ `dataFlow`	error `ForbiddenOperation`
$\texttt{InboundIdleState}^{\tau}$ `Unidirectional`	error `ForbiddenConnection`
$\texttt{InboundIdleState}^{\tau}$ `Duplex`	transition to `OutboundState Duplex`
`InboundState Unidirectional`	error `ForbiddenConnection`
`InboundState Duplex`	transition to `DuplexState`
`DuplexState`	error `ConnectionExists`
$\texttt{TerminatingState}^{\tau}$	await for `TerminatedState`
`TerminatedState`	can be treated as initial state

Table 4.2: `requestOutboundConnection`; states indicated with a † are forbidden by TCP.

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

Table 4.3: `unregisterOutboundConnection`

Connection manager methods

The tables 4.2 and 4.3 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*), is able to 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 its actually impossible that the state would be inconsistent.

4.3.8 *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 there is no thread that could be blocked on it: if there was another thread that 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.

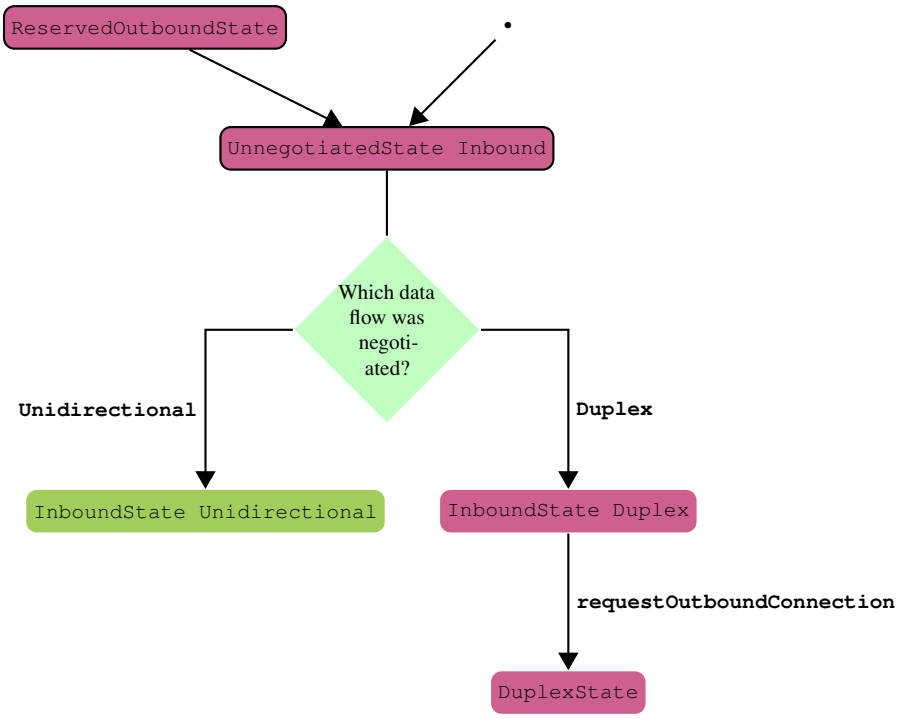


Figure 4.7: *Inbound* connection flow graph, where both bordered states: `ReservedOutboundState` and `UnnegotiatedState Inbound` are initial states.

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

Table 4.4: `includeInboundConnection`

Connection manager methods

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

- `includeInboundConnection`: table 4.4
- `promotedToWarmRemote`: table 4.5
- `demotedToColdRemote`: table 4.6
- `unregisterInboundConnection`: table 4.7

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 $\mathsf{DemotedToCold}^{\mathsf{Duplex}}_{\mathsf{Remote}}$[2]. Lets consider the following sequence of transitions:

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

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

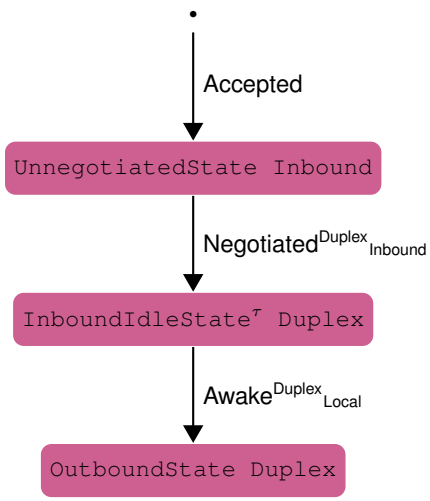
Table 4.5: `promotedToWarmRemote`

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 4.6: `demotedToColdRemote`

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

Table 4.7: `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`.

4.4 Server

The server consists of two components: 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 responder side of all mini-protocols, and it is responsible for starting and restarting mini-protocols, as well as detecting if they are used, in order to support:

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

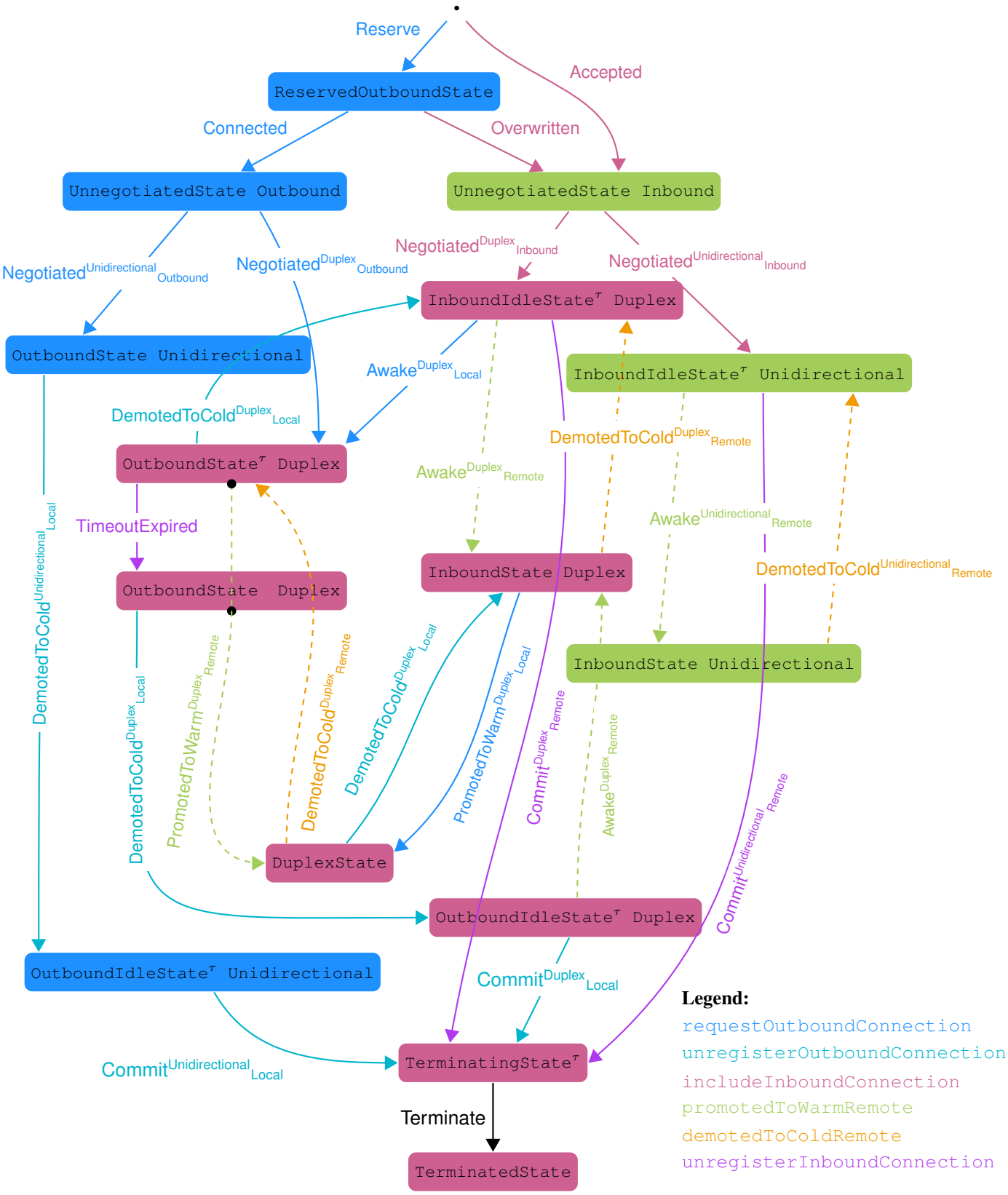


Figure 4.8: Transitions classified by connection manager method.

- DemotedToCold$^{Unidirectional}_{Remote}$,
- Commit$^{Unidirectional}_{Remote}$ and Commit$^{Duplex}_{Remote}$ 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 PromotedToWarm$^{Duplex}_{Remote}$ transition using `promotedToWarmRemote` method.

Once all responder mini-protocols become idle, i.e. they all stopped, were re-started (on-demand) but are not yet running, a DemotedToCold$^{dataFlow}_{Remote}$ 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:

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

```
-- | Return 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 side of all protocols ready.
  | KeepTr
  deriving Show

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

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

4.5 Inbound Protocol Governor

Inbound protocol governor keeps track of 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 to notify it about new connections once they are negotiated. Figure 4.9 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, which represents mux or mini-protocol failures.

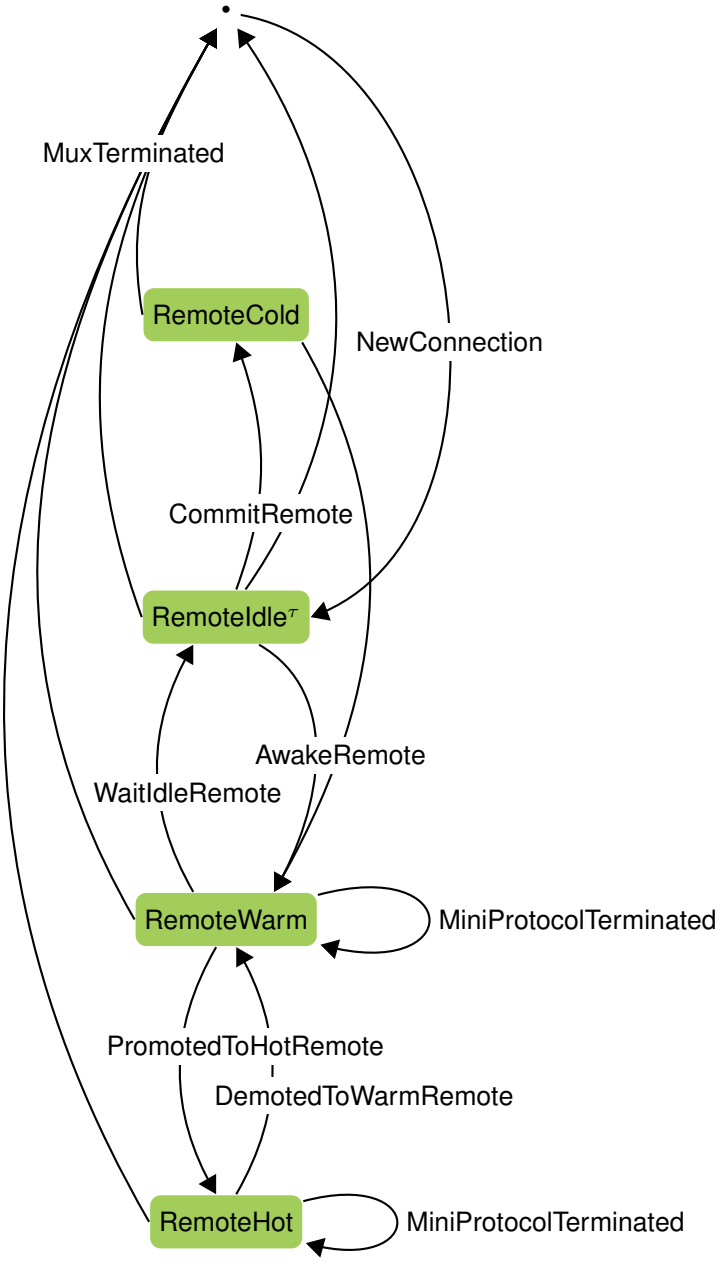


Figure 4.9: Inbound protocol governor state machine

4.5.1 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 shutdown (MuxTerminated).

RemoteIdle$^{\tau}$

The RemoteIdle$^{\tau}$ state is the initial state of each new connection (NewConnection). An active connection will become 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 whether the connection is used (*warm* or *hot*) or not (*cold*) by the outbound side.

RemoteWarm

A connection enters RemoteWarm state once any of the mini-protocols starts to operate. Once all hot mini-protocols started the state will transition to RemoteHot. Note that this is slightly different than the notion of a *warm* peer, for which all *established* and *warm* mini-protocols are active, but *hot* ones are idle.

RemoteHot

A connection enters RemoteHot transition once all hot protocol started, if any of them terminates the connection will be put in RemoteWarm.

4.5.2 Transitions

NewConnection

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

CommitRemote

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

AwakeRemote

While a connection was put in 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 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 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 will be able to detect this by itself.

PromotedToHotRemote

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

DemotedToWarmRemote

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

Appendix A

Common CDDL definitions

```
1
2  ; The Codecs are polymorphic in the data types for blocks, points, slot
3  ; numbers etc..
4
5  block           = [blockHeader, blockBody]
6
7  blockHeader     = [headerHash, chainHash, headerSlot, headerBlockNo, headerBodyHash]
8  headerHash      = int
9  chainHash       = genesisHash / blockHash
10 genesisHash     = []
11 blockHash       = [int]
12 blockBody       = bstr
13 headerSlot      = word64
14 headerBlockNo   = word64
15 headerBodyHash  = int
16
17 point           = origin / blockHeaderHash
18 origin          = []
19 blockHeaderHash = [slotNo, int]
20 slotNo          = word64
21
22 txId         = int
23 transaction  = int
24 rejectReason = int
25
26 word16 = 0..65535
27 word32 = 0..4294967295
28 word64 = 0..18446744073709551615
```

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.