P2P: network stack

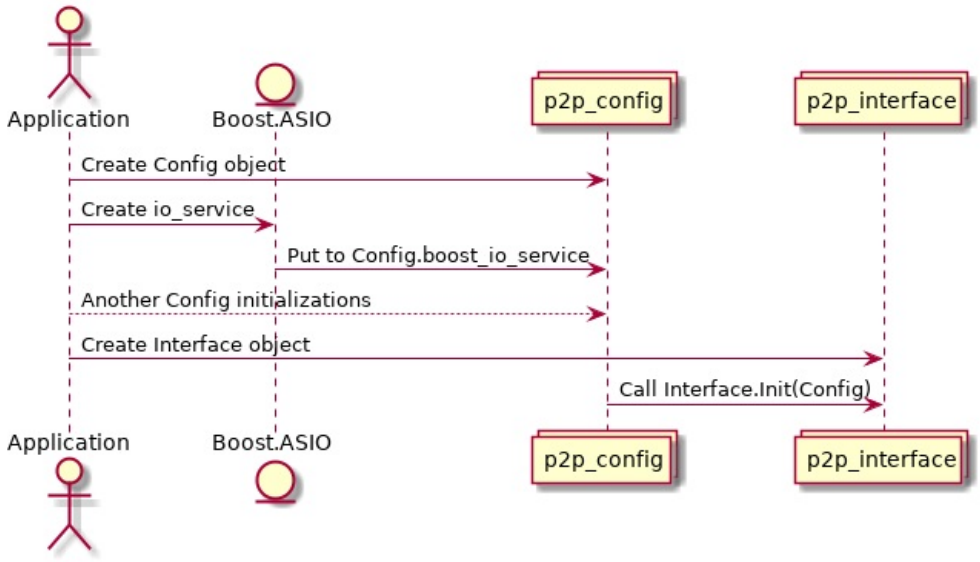
1. Overview

In this document, we consider the three layers of the network stack of the p2p subsystem. The low-level I/O in the p2p subsystem is based on an asynchronous Boost ASIO library, which acts on top of the TCP protocol. The next is the layer of messages that the p2p module inherits from the Bitcoin protocol. A few messages are left of the entire set of the Bitcoin messages. They are necessary for exchanging addresses and maintaining connections. In addition, the Propagate type of message was added to the protocol. The Propagate message consists of a header and one of the Logos protocol messages. These Logos protocol messages are in the top layer of the p2p network stack.

2. Low-level network connections

2.1. Initialization

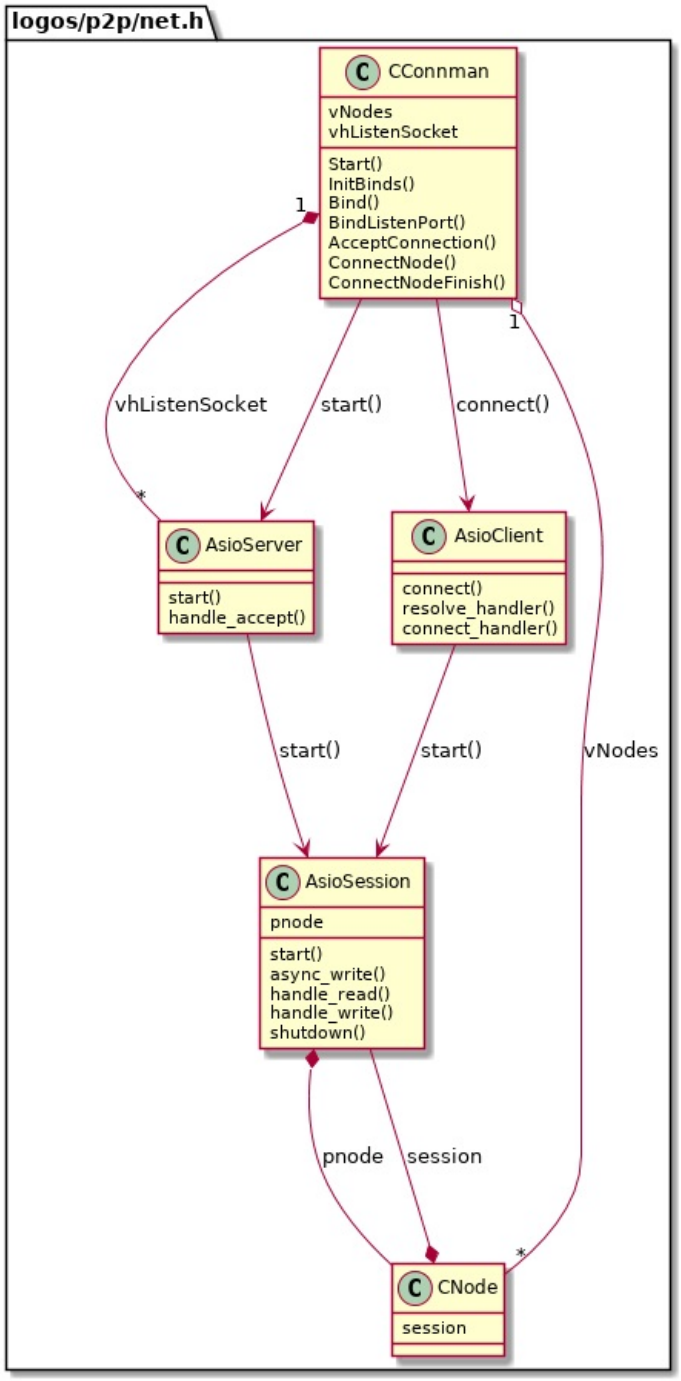
When initializing the p2p subsystem, one must first create an object of the type boost::asio::io_service and place a pointer to it in the field boost_io_service of the structure p2p_config. More information about the initialization of p2p can be found in section 2 of the document "P2P: databases". All network I/O inside the p2p subsystem is done inside the Boost ASIO library using the passed pointer to the io_service object. If the unit test does not plan to use network I/O inside p2p, the io_service field can be left blank.



Since the Boost ASIO library is used for network I/O, it requires one or more I/O threads. Note that the p2p subsystem does not independently create and does not start such ASIO threads. This should be done by the one who calls the p2p subsystem. The example of launching the ASIO thread can be found in the function io_service_run() of the test application logos/p2p/test/p2p-standalone.cpp.

Note also that the p2p subsystem creates 5 other threads on its own, which are started when the CConnman::Start() function is called. The main functions of these threads are ThreadSocketHandler(), ThreadMessageHandler(), ThreadOpenConnections(), ThreadOpenAddedConnections() and ThreadDNSAddressSeed(). The first two threads are discussed in sections 3.2 and 3.3 respectively. The remaining three threads will be discussed in another document related to the p2p network maintenance algorithms.

2.2. Classes



There are three main classes used for low-level I/O. These are AsioServer, AsioClient and AsioSession. These classes are defined in the file logos/p2p/net.h file and are implemented in the file logos/p2p/net.cpp. TCP connections through which the p2p subsystem is connected to other nodes, are divided into incoming and outgoing. For each outgoing connection, an AsioClient class object is created. For each port on which incoming connections are listened, an AsioServer class object is created. Finally, in either case, an AsioSession class object is created for each established connection.

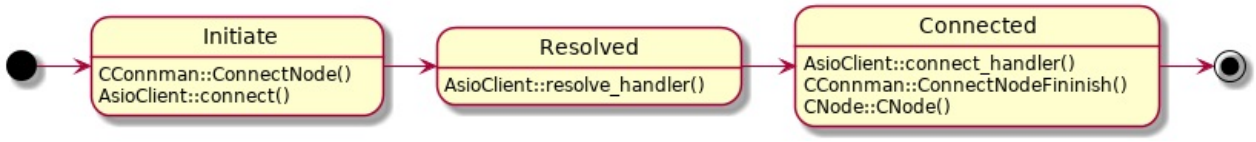
The CConnman class represented by a single object is the key class of the network subsystem of the p2p module. In particular, it contains the vector vhListenSocket of links of type shared_ptr<> to all

AsioServer class objects corresponding to listening sockets. In contrast, the CConnman class does not contain references to objects of the AsioClient class. The AsioClient class object is created only at the time of establishing a new connection, and then automatically destroyed, since all references to it are of shared_ptr<> type. At the same time, an AsioSession class object created during the connection establishment continues to exist. Also note that the AsioServer class object does not contain references to the AsioSession class objects it has created.

Each established connection corresponds to a CNode class object containing information about the node to which the connection leads. The CConnman class object contains the vector vNodes of links of type shared_ptr<> to all objects of the class CNode. The object of the class CNode is created after the establishment of either the incoming or outgoing connection and exists in parallel with the object of the class AsioSession of the given connection. At the same time, objects of the classes CNode and AsioSession have mutual references of type shared_ptr<> to each other. The mechanism for the destruction of objects of the AsioSession and CNode classes is described in the Section 2.5.

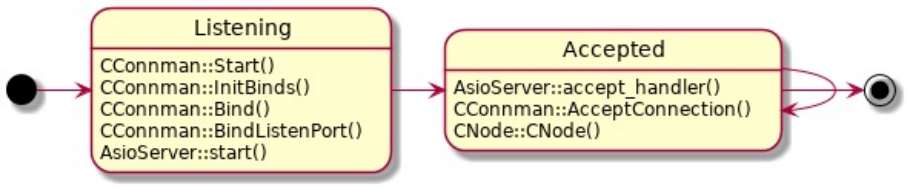
2.3. Outgoung connections

When a new outgoing connection is created, the AsioClient class object is created in the function CConnman:ConnectNode() and the method AsioClient::connect() is called, to which strings containing the name and port of the remote host are transmitted. The connect() method that uses the asynchronous I/O mechanism starts the resolving address procedure first, and then the connection procedure to the host. The callback resolve_handler() is called after a resolving finished. Next, the callback connect_handler() is called after connection is established. When the latter is called, the object of the class CNode is created inside the function CConnman::ConnectNodeFinish().



In the case of a error during resolving address, the callback resolve_handler() is also called. It does not continue further connection, but ends with an error output in the log. In case of an error of connection establishing, the callback connect_handler() is called, and it ends with an error output without continuing the further creation of the connection. For better understanding we note that according to the ideology of ASIO, each asynchronous operation contains a shared_ptr<> link to the AsioClient object. If the callback completes without opening a new asynchronous operation, then the AsioClient object is automatically destroyed, since there are no more references to it.

2.4. Incoming connections

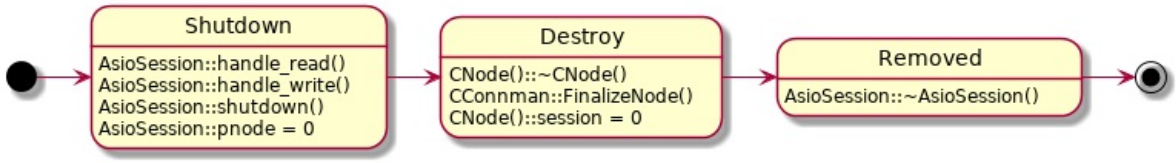


When the CConnman object starts, it binds all ports and start servers which listen incoming connections. More precise, the function CConnman::Start() calls the function InitBinds() which call the method Bind() for every listening port. The Bind() method subsequently calls the BindListenPort() function. The last function creates an AsioServer class object and calls its start() method which starts the asynchronous listening process. The callback handle_accept() is called when an incoming connection arrives. During this callback, the object of the type CNode is created inside the function CConnman::AcceptConnection(). In case of an error when accepting an incoming connection, the handle_accept() callback is also called, which outputs an error message and recreates the AsioSession

object for accepting the next connection.

2.5. Connection finalization

An AsioSession object exists as long as the connection exists. In parallel with it, there is an object of the class of CNode as described in section 2.2. The AsioSession and CNode objects refer to each other. The links are of type shared_ptr<> so the objects are not destroyed. When the connection is broken, the link leading from AsioSession to CNode is removed first. This happens in the function AsioSession::shutdown(). Then, when the last reference to the CNone is deleted, its destructor is called, which removes the reference to AsioSession. Finally, the AsioSession class object is also deleted.



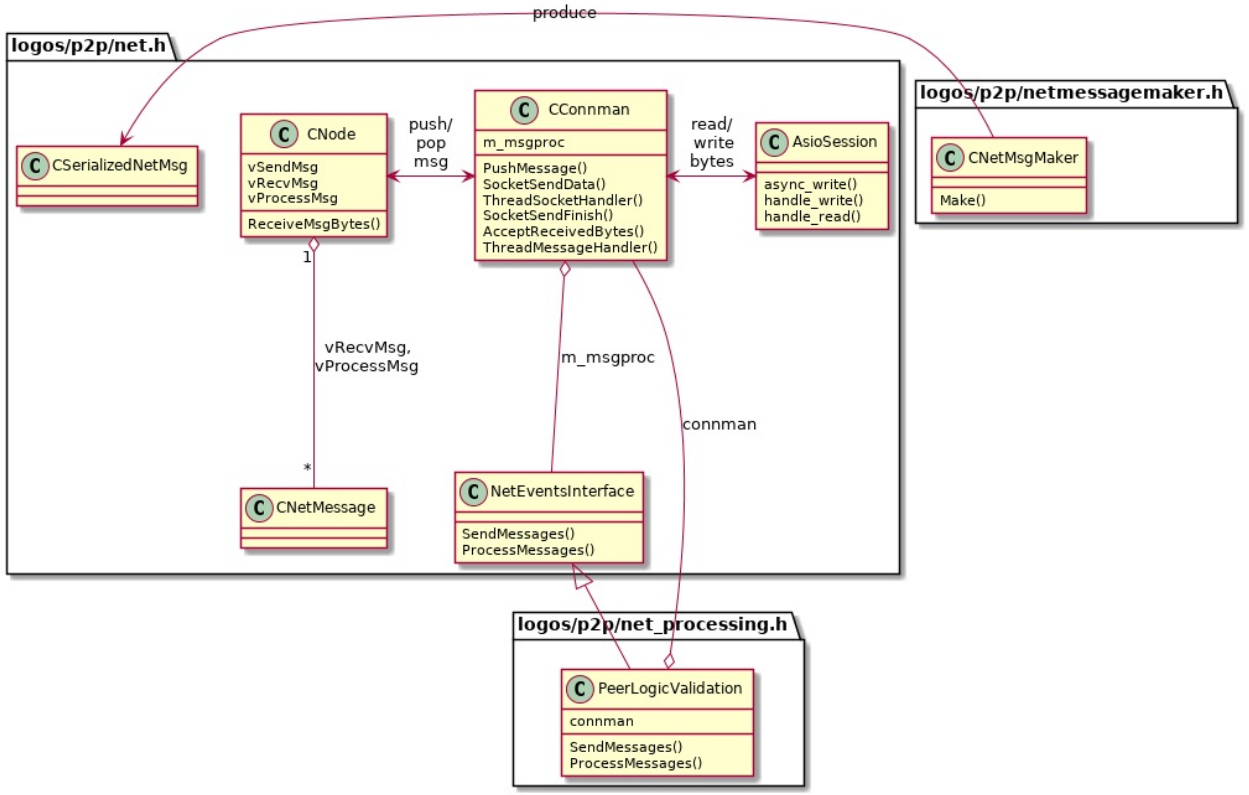
Let's talk about finalization of other objects. An AsioClient class object is automatically destroyed when the last shared_ptr<> link to it is deleted. This happens after the connection has been successfully created, or vice versa, its creation has completed with an error. An AsioServer class object is also deleted when the last shared_ptr<> link is deleted, but since it is referenced by the CConnman class object, the deletion occurs only when the CConnman object is deactivated.

Finally, there is a single object of the CConnman class, which is deleted when the entire p2p subsystem is deleted. The last deletion proceeds correctly, which can be observed when the p2p-standalone test application is exited. However, the re-creation of the p2p subsystem within the same process is apparently impossible until the planned clearing of the code when all global variables will be deleted.

3. Bitcoin type messages

3.1. Classes

After the connection between the hosts is established, messages start to be transmitted via it. These are mid-level messages, they are inherited from the Bitcoin project and will be discussed in this chapter. In addition to the already considered classes of CConnman, CNode and AsioSession, the class PeerLogicValidation described in the file logos/p2p/net_processing.h also participates in the processing of messages. This class is derived from the incomplete class NetEventsInterface described in the file logos/p2p/net.h. The class is responsible for parsing and composing of Bitcoin messages of different types. The class CNetMessage represents non-serialized message, and the class CSerializedNetMsg represents serialized one. Both classes are defined in the file logos/p2p/net.h. The class CNetMsgMaker described in the file logos/p2p/netmessagemaker.h do the serialization of messages.



3.2. Sending messages

The messages are usually sent by specialized function PeerLogicValidation::SendMessages() but may also be sent in a fly when processing incoming messages. After the message is created by the function CNetMsgMaker::Make() the CConnman::PushMessage() function is called which places the message in the queue CNode::vSendMsg, and if the queue is empty, then immediately goes on to send it. To send a message it calls the function CConnman::SocketSendData(). If the message was placed in the queue, then it is there until it is processed by the thread CConnman::ThreadSocketHandler() which again calls the method SocketSendData() to send the message. The last method calls the function AsioSession::async_write() which initializes the asynchronous mechanism for sending messages of the Boost.ASIO library.

After the message is sent, the AsioSession::handle_write() handler is called, which calls the CConnman::SocketSendFinish() function to complete the process of sending the message. If the writing ends with an error which is usually indicates a disconnection, then the callback handle_write() is also called which calls the SocketSendFinish() method with an error flag, and then calls the AsioSession::shutdown() method which starts the process of deleting the AsioSession object and its associated CNode object. More detailed information about the removal is written in the section 2.5.

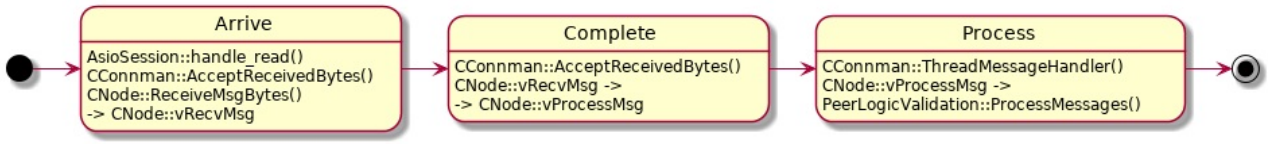
Note that the thread ThreadSocketHandler() in addition to the above function also closes connections

when a number of conditions are met. In particular, if they are inactive. These conditions will be discussed in more detail in another document on algorithms for maintaining a p2p network. The thread ThreadSocketHandler() processes all connections and then sleeps for 50 milliseconds, but wakes up earlier if new packets are added to the vSendMsg queue that require sending.

3.3. Receiving messages

When the next data bytes arrive on the connection, the AsioSession::handle_read() handler is called which sends these bytes to the function CConnman::AcceptReceivedBytes(). If the reading ends with an error which is usually indicates a disconnection, then the callback handle_read() is also called which calls the AcceptReceivedBytes() method with an error flag, and then calls the AsioSession::shutdown() method which starts the process of deleting the AsioSession object and its associated CNode object. More detailed information about the removal is written in the section 2.5. The processing of incorrect messages will be discussed in a separate document on algorithms for maintaining the p2p network.

If no error is occured then the function AcceptReceivedBytes() sends arrived bytes to the function CNode::ReceiveMsgBytes() which adds them to the previously received bytes and tries to compose a whole message from these bytes. Last incomplete message lies in the list CNode::vRecvMsg. When a whole message is received, then it is added to the list CNode::vProcessMsg. Then, the thread CConnman::ThreadMessageHandler() goes through all the objects of class CNode and for each of them calls the function PeerLogicValidation::ProcessMessages() which processes incoming messages from the list vProcessMsg. The ProcessMessages() function performs substantial work if the list vProcessMsg is non-empty. Note that this thread also calls the SendMessages() function, which initiates sending regular messages to the host. If for all hosts there is no more work for both ProcessMessages() and SendMessages() functions, then the thread sleeps for 100 milliseconds.



3.4. Bitcoin message format

Bitcoin message consists of a header and a body. Header is represented by the class CMessageHeader defined in the file logos/p2p/protocol.h. Header's magic number is defined in the class CMainParams in the file logos/p2p/chainparams.cpp. This magic number is inherited from the Bitcoin message format. Each type of Bitcoin network, namely, the main and test networks, has its own magic number. This magic number is taken from the main Bitcoin network. It does not contain version information. Perhaps this number will be changed. Message body is a sequence of bytes followed by a header. The structure of a message is shown in the following table.

Length in bytes	Field name	Field description
4	Message Start	Magic number, 0xf9beb4d9, big-endian
12	Command	String, name of the Bitcoin message type, see 3.5
4	Message Size	Size of the message body in bytes, little-endian
4	Checksum	First 4 bytes of 256-bit Blake2b hash of message body
?	Message Body	Depends on message type

3.5. Bitcoin message types

Of the entire set of Bitcoin messages, only those that are necessary to maintain the p2p network are left. In addition, there was added a new type of message named Propagate. The Propagate messages represent the top level of the network hierarchy. They are the payload for the Logos project and discussed in section 4. The following table presents the actual list of Bitcoin message types for the Logos project. This list is defined in the files logos/p2p/protocol.[h, cpp]. Some information on standard Bitcoin messages is omitted here because these message format is well known.

Name of the Type	**Description**	**Contents of the Message Body**
addr	List of addresses of known p2p nodes	Complex format
getaddr	Request for sending the addresses list	Empty body
ping	Keep-alive message	Nonce, 8 bytes
pong	Reply for keep-alive message	The same Nonce as in the ping
propagate	Logos consensus message, see section 4	Depends on the message
reject	Reject for arrived message	Rejected command, the reason
version	Hello message from the node	Complex format
verack	Acknowledge for the version message	Empty body

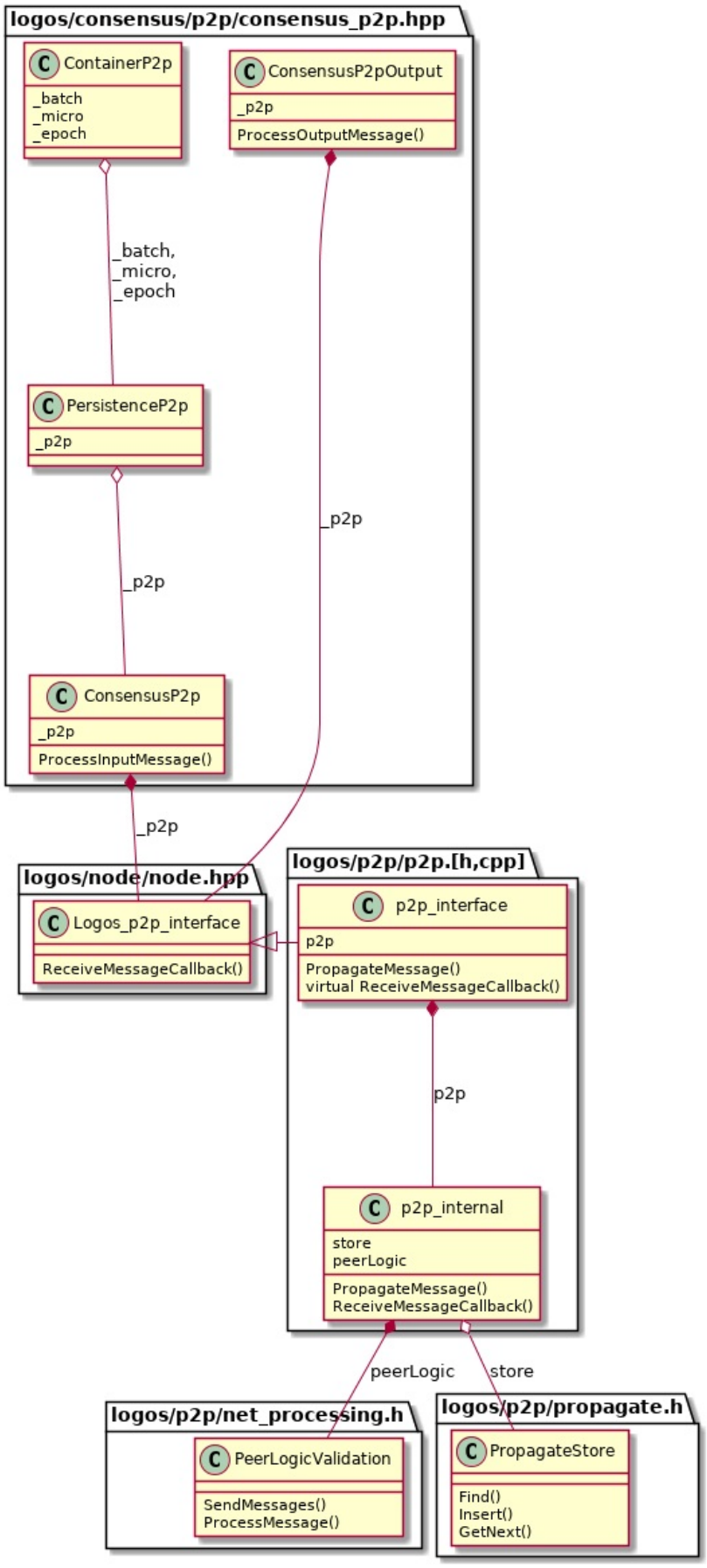
4. Logos consensus messages in p2p

4.1. Classes

Logos consensus messages are packed into Bitcoin Propagate messages and thus transmitted over the p2p network. The end point where messages are transmitted to lower levels of the network hierarchy is the already mentioned class PeerLogicValidation. The next class is PropagateStore implemented in the file logos/p2p/propagate.h. It is a repository of limited size for previous messages transmitted to or from this node. The maximum number of messages in the repository is determined by the constant DEFAULT_PROPAGATE_STORE_SIZE in the file logos/p2p/propagate.h. Currently, it is equal to 65536. When a new node connects to this, all messages contained in this repository are sent to it. Messages are sent to the new host entirely, in a natural order, without any request from that host, as if they had just come from the network. In addition, the repository helps filter out packet repetitions and not retransmit identical packets.

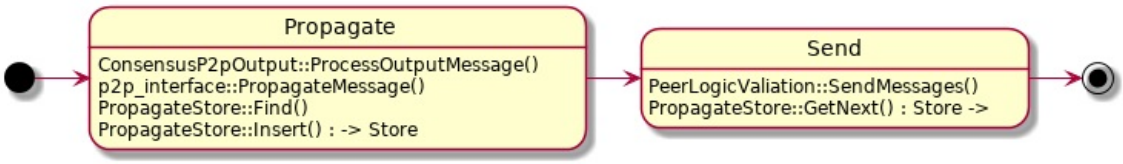
The class p2p_interface described in the file logos/p2p/p2p.h and the hidden part of this class named p2p_internal implemented in the file logos/p2p/p2p.cpp are place in the next layer of the hierarchy. They participate in the process of sending and receiving messages. The p2p_interface class is virtual and the class Logos_p2p_interface defined in the file logos/node/node.hpp is its instantiation which plays as the interface between the p2p and the rest of the Logos code.

Finally, at the top of the pyramid are the classes ContainerP2p, PersistenseP2p, ConsensusP2p and ConsensusP2pOutput defined in the logos/consensus/p2p/consensus_p2p.hpp. These classes pack the Logos messages into the Propagate containers and transmit them for sending to the p2p network and perform the reverse operation when receiving packets from p2p. Also in the class ConsensusP2p the message cache is implemented, which is necessary for situations when any link of the Logos message has not yet come from the network and one need to wait for it. Since the message cache will probably be redone, we will not consider it.


logos/consensus/p2p/consensus_p2p.hpp
ContainerP2p
_batch
_micro
_epoch
ConsensusP2pOutput
_p2p
ProcessOutputMessage()
_batch,
_micro,
_epoch
PersistenceP2p
_p2p
_p2p
_p2p
ConsensusP2p
_p2p
ProcessInputMessage()
_p2p
logos/node/node.hpp
Logos_p2p_interface
ReceiveMessageCallback()
logos/p2p/p2p.[h,cpp]
p2p_interface
p2p
PropagateMessage()
virtual ReceiveMessageCallback()
p2p
p2p_internal
store
peerLogic
PropagateMessage()
ReceiveMessageCallback()
peerLogic
store
logos/p2p/net_processing.h
PeerLogicValidation
SendMessages()
ProcessMessage()
logos/p2p/propagate.h
PropagateStore
Find()
Insert()
GetNext()

4.2. Sending Logos messages to the p2p network

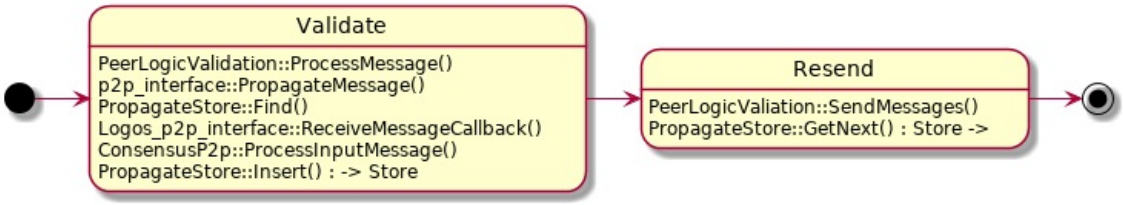
When sending a Logos consensus message to the p2p network the function ConsensusP2pOutput::ProcessOutputMessage() is called which receives the serialized representation of the message as an array of bytes. This function adds p2p header to the posting message thereby receiving the body of the Propagate Bitcoin message and passes it to the function p2p_interface::PropagateMessage() specifying the third parameter as true which means that the packet is generated locally and does not need to be checked. Further, this message is searched for in the PropagateStore repository and if it is not there then it is added there mediated by class p2p_internal. At the same time, for each connected node, the function PeerLogicValiation::SendMessages() is periodically called which sequentially sends all messages from the PropagateStore that have not yet been sent to this node.



4.3. Receiving Logos messages from the p2p network

When a Bitcoin Propagate message arrives from the p2p network, it is processed by the function PeerLogicValidation::ProcessMessage(). This function for the message of the Propagate type calls the function p2p_interface::PropagateMessage() with false as the third argument which means that the message came from the network and needs to be checked. This function first looks for a message in the PropagateStore repository and if it is not there then it calls the virtual callback redefined as Logos_p2p_interface::ReceiveMessageCallback().

The last callback through the mediation of the classes ContainerP2p and PersistenseP2p transmits the message to the method ConsensusP2p::ProcessInputMessage() which separates the header and obtain Logos message. Further, the Logos message is validated and stored in the Logos database. The cache is also used here, which we do not consider. If the message is recognized as correct then the function ProcessInputMessage() returns true and then the function PropagateMessage() adds it to the ProposalStore repository which leads to its further distribution to the p2p network. If the message goes to the cache, i.e., the question of validating the message is not fully resolved, no proof of correctness or incorrectness has been found, then the function ProcessInputMessage() also returns true and the message continues to propagate through the network.



4.4. Format of the Propagate message

The following table shows the format the body of the Propagate message. The Bitcoin message header is described earlier in section 3.4. The Propagate message body consists of a second layer header and a standard Logos message. The format of second layer header defined in the structure P2pConsensusHeader in the file logos/consensus/messages/messages.hpp. The Logos message format is described in the Logos protocol documentation.

Length in bytes	Field name	Field description
4	Epoch number	Number of the Epoch which the message belongs to
1	Source Delegate Id	Id of delegate which issued the message or 0xff for non-delegates or PostCommittedBlock messages
1	Destination Delegate Id	Id of destination delegate or 0xff if no unique destination is given
?	Logos Consensus Message	One of standard Logos messages

5. Unit tests

The functionality described in this document is mostly not covered by unit tests, since checking it requires multiple hosts and a network. To test the network functionality of the p2p subsystem, one can use the test application logos/p2p/test/p2p_standalone.cpp which will be described in a separate document.

There is the unit test VerifyStore for testing the message repository PropagateStore. It is in the file logos/p2p/test/propagate_test.cpp. There is also a VerifyCash unit test for checking the message cache (which may be reworked later). It is in the file logos/unit_test/p2p_test.cpp.