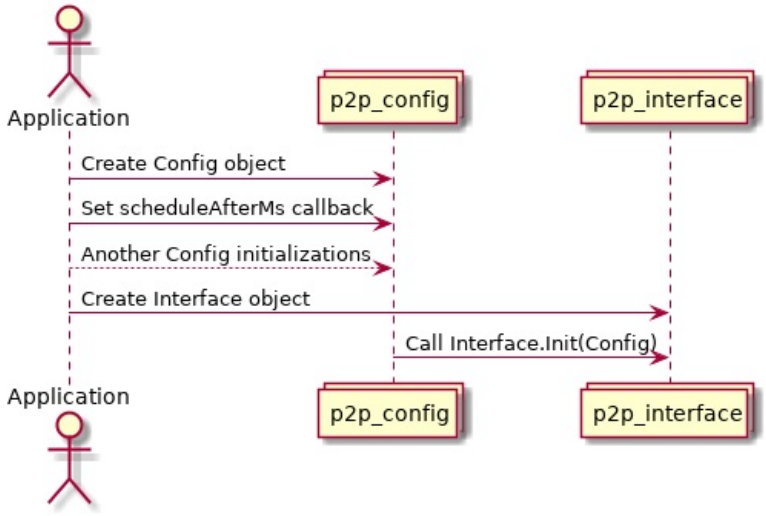
P2P: network maintenance

1. Overview

This document focuses on algorithms that support the p2p network. In particular, we will consider the following questions:
1) Under what conditions is a new connection created and when an existing one is manually broken.
2) What is the messaging procedure through the connection.
3) What happens if incorrect data comes through the connection.
4) In what case the newly discovered host included to the database and is excluded from it.
5) When the host will be banned and when will be excluded from ban list.
6) How is the exchange of contacts between hosts.
7) How the DNS service works, which is built into the p2p network.

2. Initialization

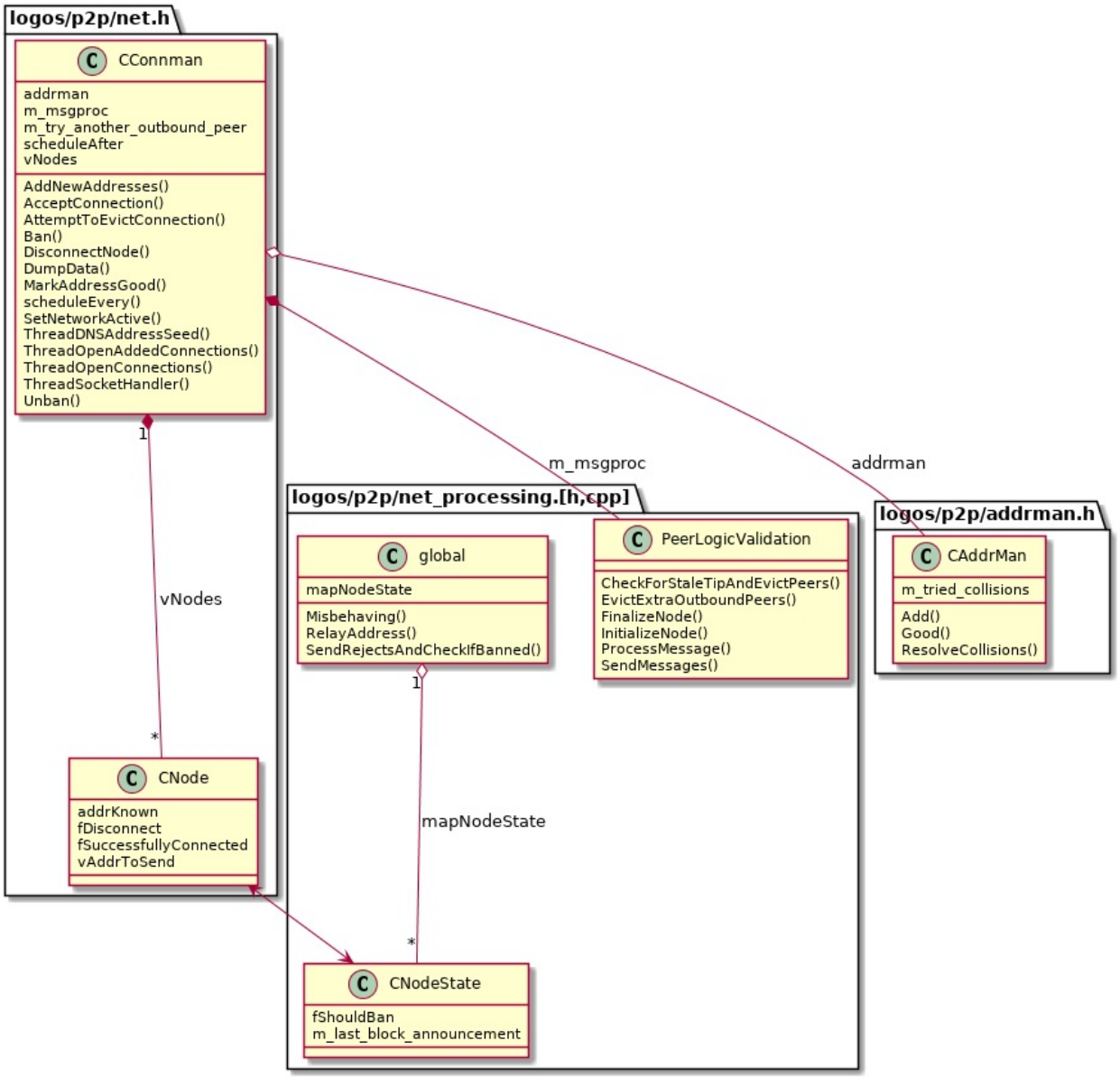
In this section, we describe what needs to be done to initialize the p2p subsystem in the part related to this document. Additionally, one need to perform the actions that are described in the sections on initialization in documents p2p-databases.pdf, p2p-network-stack.pdf and p2p-options-feedback.pdf. In the p2p_config structure, one must fill in the scheduleAfterMs field. In this field, one need to put a function that will be called as a callback if the p2p subsystem needs to set a task for execution after a certain number of milliseconds from the current moment. Callback arguments are a function that will be called one time after a given number of milliseconds and this number of milliseconds. It is assumed that the one who initializes the p2p subsystem has one or several threads that play the role of a scheduler, that is, they perform the given functions on a schedule. For example, such a scheduler is inside the Logos code. In order not to create unnecessary threads, the p2p subsystem uses an external scheduler.



During the initialization of the p2p subsystem the scheduleAfterMs callback is copied into the field CConnman::scheduleAfter and the function CConnman::scheduleEvery() is created based on it. The function scheduleEvery() execute the given task periodically with a given period. Currently the p2p subsystem has two such periodic tasks. They are the CConnman::DumpData() function which is called with a period of DUMP_ADDRESSES_INTERVAL seconds (the current value is 900) and saves the databases of all and banned hosts to disk, and the PeerLogicValidation::CheckForStaleTipAndEvictPeers() function which is called with the period EXTRA_PEER_CHECK_INTERVAL (the current value is 45) and searches for staled connections that can

be broken. Databases are discussed in the p2p-databases.pdf document, and the CheckForStaleTipAndEvictPeers() function will be discussed in Section 4 about the connection management.

3. Classes



Next, we provide a class diagram, in which the functions mentioned in subsequent chapters are marked. The classes referred to in this document have basically already been described in the p2p-network-stack.pdf document. An exception is the CNodeState class, which is described inside the logos/p2p/net_processing.cpp file and is used only in this file. This class contains the internal data structure for each host with which there is an active connection. Many of fields in this class are obsolete and will apparently be deleted. In addition, the mapping that associates each node with information about it is specified as the global variable mapNodeState is also subject to reorganization.

Currently, when a new connection is created, the PeerLogicValidation::InitializeNode() function is called, which adds an instance of the CNodeState class to the mapNodeState. Note that the mapping does not contain a reference to an object of the CNodeState class, but the object itself. When the

connection is deleted, the PeerLogicValidation::FinalizeNode() function is called, which removes the object from the map.

4. Connections Management

4.1. Types of outgoing connections

Outbound connections are of several types:

1) Manually performed connections. These include connections to hosts that are specified on the command line using either --addode or --connect option. The difference between these options is that if there is at least one --connect option, then outgoing connections will be made only with those hosts that are listed in the --connect options. At the same time, connections to the hosts listed in the --addnode options are not exceptional i.e. such connections are always keeped established, but at the same time connections to other hosts are allowed.

2) One shot connections. These connections are performed with the hosts specified on the command line using the --seed option. These hosts are used to request the addresses of other hosts on the network. Therefore, connection to such hosts occurs only once.

3) Feeler connections. The feeler connections mechanism allows testing of new hosts whose addresses have become known to this host, but with which a connection has not been established before. A feeler connection is established every few minutes with a random such a host. After a full handshake over the connection it is broken, and the host with which this connection was established is marked as tested. No more than one feeler connection is allowed at a time.

4) Regular connections. All other outgoing connections. The number of regular outgoing connections is limited by the MAX_OUTBOUND_CONNECTIONS constant specified in the file logos/p2p/net.h. Its current value is 8.

4.2. Establishing outgoing connections

The establishment of outgoing connections involved in two threads. The thread CConnman::ThreadOpenAddedConnections() establishes only the manual connections specified on the command line with the --addnode option. This thread runs in an infinite loop and checks if all such hosts have active connections. If not, it attempts to connect with those of them with which there is no connection. After attempting to connect to one host, there is a break of half a second before attempting to connect to the next host. After go through all the hosts, the thread falls asleep for 2 seconds if there were no attempts to connect in this round, and for 60 seconds if at least one attempt was made.

The second thread, CConnman::ThreadOpenConnections(), is responsible for establishing all other types of connections. Namely, if at least one host is specified on the command line using the --connect option, then this thread in the infinite loop establishes connections only with those hosts that are specified with the --connect options, making a half-second break between connection attempts. In this case, the thread also creates one shot connections with seed hosts. Otherwise, if the --connect options are not in the command line, then the ThreadOpenConnections() thread every 5 seconds creates one of the one shot connections that have not yet been established, as well as either a feeler or regular connection. In this case, the logic of the work of the ThreadOpenConnections() thread is described in the following section.

4.3. Establishing regular and feeler connections

A regular connection is created in one of the following cases: 1) if the number of outgoing regular connections is less than the allowed limit; 2) if it is time to create a new regular connection and, at the same time, delete one of the existing ones. In case 2) the occurrence of the desired time is governed

by the variable CConnman::m_try_another_outbound_peer. The CheckForStaleTipAndEvictPeers() function which is periodically started by the scheduler (see section 2) sets this variable to the value true with a periodicity of STALE_CHECK_INTERVAL seconds. The value of this constant is 600 seconds, that is, 10 minutes. If a regular connection will not be created, but it is time to create a feeler connection, then a feeler connection will be created. The creation time of the next feeler connection is defined as the last time a feeler connection was created plus a random number of seconds where the random variable is distributed according to the Poisson law and its average value is equal to the constant FEELER_INTERVAL, namely 120 seconds, i.e. 2 minutes.

After the decision to create a connection is made, and its type is determined, the process of selecting the host with which the connection will be established begins. For this, 100 attempts are made to randomly select a host from the database. After each attempt, the necessary conditions are checked, and if the next host satisfies them, then the process of connecting to it begins. The conditions are: the host address must be correct, not local (correct and local addresses are defined in the next paragraph), and at the moment there is no connection to another host in the same /16 range of ip-addresses. The latter is done to protect against spam attacks, when a malicious host sends a lot of addresses of fake hosts. Typically, the IP addresses of the fake hosts are close to each other.

An address is considered correct if it is not an address of the type INADDR_NONE (all zeros in the case of IPv4), is not an address reserved for documentation in IPv6, and is not the internal address for the p2p subsystem. Here is the definition of internal address. For each ip-address of p2p node there is stored the address of another node from which sent the first address. But the ip address of the root seed node is hardcoded, so a special non-existent ip address is stored as the source of such a seed node address. This special ip address is called internal. Next, an address is considered local if it is assigned to some network interface on the local machine. All local interfaces are discovered in the CConnman::Discover() function. The linux function getifaddrs() is called for this purpose. Local addresses are added into the mapping CConnman::mapLocalHost.

Further, during the first 30 attempts, the hosts with which the connection was made within the last 10 minutes are not considered. During the first 50 attempts, hosts whose p2p port is different from the standard one are not considered. If a feeler connection to be created, then only "new" hosts considered which have not previously connected. If a regular connection to be created, then with a 50% probability, an already tested host is selected, and with a 50% probability, a "new" one is selected. It is also possible that some host established an incoming connection with us, but there was not enough space in the corresponding basket of checked hosts to record this host, i.e. there was a conflict. In this case the host is marked as requiring verification, and if there is such a host, then a feeler connection is established with such a host for verification. More information about the internal structure of the database of hosts is written in section 6.

4.4. Disconnection

Disconnection occurs in 2 stages. Initially, the connection is marked as intended to delete. This is done by setting the flag of CNode::fDisconnect. Then the thread CConnman::ThreadSocketHandler() goes through all connections every 50 milliseconds and for those that are marked to delete, starts the process of breaking the connection and deleting all objects. Deletion of objects occurs automatically, as they use shared_ptr<> type pointers.

There are several actors that can mark a connection for deletion. First, the ThreadSocketHandler() thread itself marks some connections during a regular processing. Namely, if the network subsystem is turned off manually, then all connections are marked for deletion. One can turn off or turn on the network subsystem with the CConnman::SetNetworkActive() function, but it is not currently used. Further, ThreadSocketHandler() marks the connection when one of the following conditions is met, if at least one minute has passed since the connection was established:
1) either not sent or not received any messages;
2) more than TIMEOUT_INTERVAL seconds passed since the last message was sent, the value of this

constant is 20 * 60, that is, 20 minutes;
3) more than TIMEOUT_INTERVAL seconds passed since the last message was received;
4) more than TIMEOUT_INTERVAL seconds passed since teh PING message was sent and no response was received to it;
5) the connection is not fully initialized; the message exchange protocol for the connection is discussed in more detail in the section 4.

The next actor that can mark connections to deletion is the PeerLogicValidation::ProcessMessage() function which parses the received messages. The connection is flagged in one of the following cases:
1) if, upon receiving the VERSION message, the host properties do not match the mask (this is a relic from Bitcoin and will be removed);
2) if the connection leads to the same host, it is determined by the parameter nonse in the VERSION message;
3) if this connection is a feeler one, then after receiving the VERSION message it is broken;
4) if this type of connection is one shot, then after receiving the ADDR message with the list of addresses, it is broken;
5) if the message header does not match the format.

Further, there are several other possibilities when a connection will be marked for deletion. The CheckForStaleTipAndEvictPeers() function which is periodically called by the scheduler, calls the PeerLogicValidation::EvictExtraOutboundPeers() function which if the number of regular outgoing connections exceeds the allowed limit, selects one of them and marks it for deletion (so it is allowed to exceed the limit for short time). For each connection, the variable CNodeState::m_last_block_announcement stores the time of the last arrival of the Propagate message, that is, the Logos consensus message. The connection for which least time is recorded in this variable will be selected for deletion. However, if the selected connection exists less than MINIMUM_CONNECT_TIME seconds, where the value of the constant is 30, then it is not deleted.

Finally, connections are removed when they break at the network stack level. In addition, the connection is deleted if the host to which it is going, is marked as should be banned (it means that the host will be banned soon). Also, there are several cases of deletion of connections inherited from Bitcoin that will be cleared from the code: the timeout exceeding when the blocks are initially loaded, and the CConnman::DisconnectNode() function that is not used. Deletion of incoming connections, except for common cases for incoming and outgoing, will be discussed in section 4.5.

4.5. Incoming connections

When an incoming connection is established, the CConnman::AcceptConnection() function is called, which creates the necessary objects for the connection. This function, in particular, verifies if the number of existing incoming connections exceeds the maximum number allowed. The maximum number of incoming connections is calculated as the maximum number of all connections minus the maximum number of outgoing connections and minutes, the maximum number of feeler connections which equal to 1. The maximum number of all connections by default is set to the constant DEFAULT_MAX_PEER_CONNECTIONS the value of which is 125. Further, the maximum number of connections can be overridden by the option - -maxconnections of the command line.

Finally, the maximum number of connections is reduced if the system gives the application an insufficient number of handles to maintain all connections. Each connection needs its own file descriptor. The number of descriptors available to the application is limited by the system. The p2p module tries to increase the number of available descriptors by calling the system function setrlimit() in the function RaiseFileDescriptorLimit() in the file util.cpp, and if this fails, it reduces the maximum number of connections. Another limit for the number of connections is the FD_SETSIZE constant — the maximum number of descriptors that can be passed to the select() function.

If the current number of incoming connections including the new one is more than the maximum

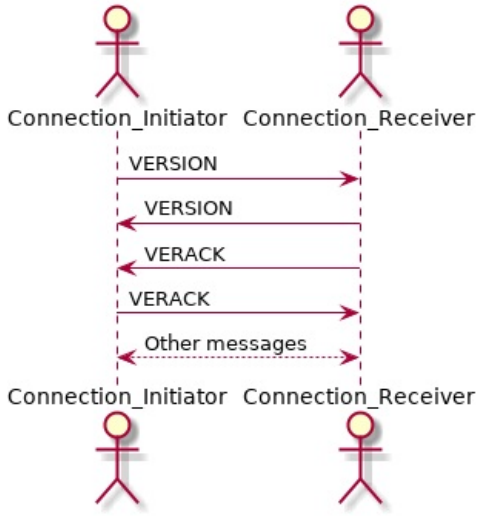
allowed, then the function CConnman::AttemptToEvictConnection() is called which tries to delete one extra incoming connection. If this fails, the new connection will not be established. To determine the connection that will be deleted, the following algorithm is used. First, non-incoming connections are eliminated, as well as connections to the whitelist hosts and connections already marked for deletion. Further, a deterministic algorithm with an unpredictable result for an attacker selects some hosts, connections to which are protected from deletion. Namely it selects:
1) hosts which included in 4 determined groups (see the next paragraph) of network addresses /16;
2) 8 hosts with minimal ping time;
3) 4 hosts who recently sent transactions (apparently this remained from Bitcoin and requires adjustments);
4) 4 hosts that recently sent blocks (likewise);
5) half of the remaining hosts that were connected to ours as long as possible (i.e. connected via long-lived connections).

We describe what the deterministic algorithm in the previous section means. When creating a connection, it is assigned a number CNode::nKeyedNetGroup. This is the hash of the address group /16 of the remote host ip address. To obtain this hash, the CConnman::CalculateKeyedNetGroup() function is called, which in turn uses a deterministic randomizer. The latter is a special hashing algorithm that uses two pseudo-random constants CConnman::nSeed0 and CConnman::nSeed1 to generate a hash. These constants are generated once at the start of the program using the system tools for generating pseudo-random numbers. So, when defining 4 protected address groups, those groups are selected whose codes nKeyedNetGroup are minimal.

Finally, the network address group /16 is selected with which there is the largest number of unprotected connections remaining. If there are several such groups, then the one is choosed in which there is the youngest connection (which was established as late as possible), and this connection is marked for deletion.

5. Messaging

5.1. Messaging protocol



The message format is described in the p2p-network-stack.pdf document. The first message sent from each side when opening a connection should be the VERSION message. In this case, the peer initiating the connection sends the message VERSION first. This is done in the PeerLogicValidation::InitializeNode() function. Further, the peer for which the connection is incoming receives the message VERSION and sends its VERSION message, and then the response message VERACK. Finally, the initiator of the connection receives the VERSION message from the passive side

and sends the VERACK message in response. After receiving the VERACK message, each side decides that the connection is fully established and sets the variable CNode::fSuccessfullyConnected to the value true. Recall that if this variable is not set within 60 seconds after the connection is established, then the connection will be deleted. Further messages can be sent in custom order with some exceptions. Messages in response to other messages are sent from the PeerLogicValidation::ProcessMessage() function, and messages initiated by the host such as PING, ADDR, and PROPAGATE are sent from the PeerLogicValidation::SendMessages() function.

The protocol rules are:
1) the message VERSION is sent only once at the very beginning;
2) the message VERACK sent only once as the second message;
3) the PING message is sent every PING_INTERVAL seconds, the value of this constant is 120;
4) the PONG message is sent in response to the ping;
5) the message GETADDR is sent once and only by the initiator of the connection;
6) the ADDR message with the address list is sent in response to GETADDR, as well as spontaneously in some cases (spontaneously means that the ADDR message is not sent as a response to GETADDR, but on the host's own initiative, in accordance with the algorithm discussed in the section 5.2). How often an ADDR message is sent and what addresses will be included in it is discussed in section 5.2.
7) A REJECT message is sent in response to any message if an abnormal situation occurs during its processing. The places in the code from where the REJECT is sent are not indicated explicitly. This message is sent if an exception is thrown during the processing of an incoming message in the ProcessMessage() function.

If an invalid message arrives, there may be different reactions. If the magic number in the header is wrong, then the connection is immediately broken. If another error in parsing the header occurs, in particular, the checksum mismatches or header includes an unknown command, the message is ignored without consequences for the connection. If the peer violate the order of messages then the penalties are imposed to the peer. For example, if the first message is not the VERSION or the second is not VERACK, or such a message came more than once, then 1 penalty point is imposed. Upon reaching the threshold of penalty points, the peer will be banned. The threshold for penalty points is set by default to the constant DEFAULT_BANSCORE_THRESHOLD which is 100, but it can be changed with the --banscore command line option.

5.2. Address exchange

Each host for each connection periodically sends an ADDR message containing some addresses of other hosts. This message is sent from the PeerLogicValidation::SendMessages() function if the time for sending the next message is reached. The send time is calculated as the time of sending the previous ADDR message plus a random number of seconds. The random number of seconds is distributed according to the Poisson law with an average value AVG_ADDRESS_BROADCAST_INTERVAL which is equal to 30. When this time is reached one or more ADDR messages are generated containing all the addresses accumulated to the current time for transmission to this host. If the number of addresses is more than 1000, then they are divided into several ADDR messages, the size of each of which does not exceed 1000 addresses. These messages are sent immediately one after another. If the host receives an ADDR message with more than 1000 addresses, this is considered a protocol violation and 20 penalty points are imposed on the peer that sent the message.

Addresses destined for sending to this peer are stored in the array CNode::vAddrToSend. When sending addresses, they are removed from this array and added to the array CNode::addrKnown. Only addresses that are not contained in the addrKnown array are sent. Thus, the same address cannot be sent twice to the same peer. Please also note that if node A sends all known addresses to node B and later the two nodes disconnects and reconnects, then node A will send all known addresses to node B again.

Addresses are added to the vAddrToSend array in the following cases:

1) when establishing an outbount connection, host self address (i.e. address of our host) is added;
2) upon receipt of the request GETADR addresses of all known hosts added;
3) when an ADDR message is received, if its size does not exceed 10 addresses, then the addresses of hosts from this message that were active in the previous 10 minutes are added to the vAddrToSend array of one or two hosts selected by a deterministic algorithm. Please note that hen an address is sent to another host, information about it is sent along with it, for example, about the time of the last activity of this host. Thus, chain distribution of addresses occurs. The parameters in this distribution are selected in such a way that each address is sent to each host on average once a day. One or two hosts to which this address will be passed for further distribution are determined by a deterministic algorithm in the RelayAddress() function. As a pseudo-random number, this algorithm uses the constants nSeed0 and nSeed1 randomly chosen at the beginning of the program.

6. Database management

6.1. Internal structure of hosts database

The database of all known hosts is divided into two parts. Each of these parts is organized as a hash table divided into fixed-size buckets. The size of the bucket in each hash table is 64. The first hash table contains the addresses of the new hosts. These addresses were obtained from other nodes, but the connection with them has not yet been established. The second hash table contains the addresses of already tested hosts with which the connection was established. All parameters of the tables are specified in the logos/p2p/addrman.h file as named constants. The choice of buckets or bucket groups is produced by a deterministic algorithm that uses a 256-bit key generated by the host and unknown to other network peers. When a database is saved to disk, its entire structure is saved.

The table of new hosts contains 1024 buckets. The bucket number is calculated based on the address of the parent host and the address of the host itself. The parent host is the one that sent us the address of this host. First, 64 out of 1024 buckets are selected. This choice is based on the subnet /16 of the parent host, that is, it takes into account only the higher 2 bytes of the ip address. Further, from these 64 buckets, one is selected depending on the subnet /16 of the host itself. If there is no space in this bucket, then a randomly selected host is removed from it. Preference when deleting is given to hosts whose last appearance on the network was earlier. The same address can be contained several times in this table if it was sent to us by several parent hosts, but the number of repetitions should not exceed 8.

The table of tried hosts contains 256 buckets. Eight of these buckets are selected based on the subnet /16 of the host. From the selected 8 buckets, one is selected based on the full address of the host. If, when moving a new host to the tried table, there is no free space in the bucket, then the new host is placed in the array CAddrMan::m_tried_collisions. The size of this array is limited by the constant ADDRMAN_SET_TRIED_COLLISION_SIZE, equal to 10. If the array is full, then moving to the table of tried hosts does not occur. If this array is not empty and a new feeler connection is created, then a host from this array is taken as a candidate for new connection, as described in the end of section 4.3.

In addition, the CAddrMan::ResolveCollisions() function tries to resolve collisions placed in the array m_tried_collisions by analyse the old host which is a candidate to remove from the tried table. This function is called before selecting the host for the feeler connection. The function ResolveCollisions() replaces the old host with a new one in the tried hash table if during the last ADDRMAN_REPLACEMENT_HOURS hours, i.e. 4 hours, there were no successful attempts to connect to the old host, but at least one such failed attempt was made not earlier than ADDRMAN_REPLACEMENT_HOURS hours ago and no later than a minute ago. If the replacement is took place, then the old host is moved from the table of tried hosts to the table of new hosts. If during the last ADDRMAN_REPLACEMENT_HOURS hours there was a successful attempt to connect to the old host, then the new host is excluded from the m_tried_collisions array and thus it does not move from the new hosts table to the tried table.

Here is pseudocode of ResolveCollisions() logic:

```
if last successfull connection to the old host was during last 4 hours
  do not replace old host with a new one
else if last failed connection to the old host was during last minute
  do not replace old host with a new one
else if last failed connection to the old host was during last 4 hours
  replace old host with new one
```

6.2. Inclusion / exclusion from databases

In this section, we describe in which cases hosts are included or excluded from the databases, in particular, moved between different hash tables, and also added or removed from the database of banned hosts. Inclusion in the hash table of new hosts occurs in the following cases:
1) If the address of the new host arrives in the ADDR message. In this case the function CConnman::AddNewAddresses() is called, which in turn calls the the function CAddrMan::Add().
2) In the case of an explicit addition of a host to the database through the interface which is described in the p2p-databases.pdf document. The AddNewAddresses() function is also called here.
3) When retrieving the address from the DNS host message. This mechanism will be discussed in section 6.3. In this case, the function CAddrMan::Add() is called directly.

Moving a host from the hash table of new hosts to the hash table of checked hosts is initiated in a single place, namely, in the ProcessMessage() function, after the connection is fully established. In this case the CConnman::MarkAddressGood() function is called which in turn calls the function CAddrMan::Good(). How the process of moving to another hash table is arranged is described in section 6.1. Moving back from a hash table of checked hosts to a hash table of new ones, as well as removing a host from the hash table of new hosts, occurs only by wiping out when the buckets are full.

In order to ban a host, the CConnman::Ban() function is called. This happens in the following two cases:
1) When reaching the threshold of penalty points. Penalty points are discussed in the section 5. When charging penalty points, the global function Misbehaving() in the logos/p2p/p2p_processing.cpp file is called which sets the CNodeState::fShouldBan flag when the threshold is reached. Next, when sending or receiving messages, the SendRejectsAndCheckIfBanned() global function is called which if the flag is set, calls the Ban() function.
2) When an explicit blocking of the node through the interface which is described in the document p2p-databases.pdf.
The reverse mechanism of host unbanning is provided, there is an CConnban::Unban() function, but it is not currently used.

6.3. DNS (seed) hosts

Hosts DNS in terms of the messaging protocol is the usual hosts network p2p. The difference from other hosts is that the domain names of the DNS hosts are well known and spelled out in the code. Also, DNS hosts work only in the address provider mode, i.e. the connection with them is broken immediately after receiving the address list inside the ADDR message. The initial connection to the DNS hosts is handled by the thread CConnman::ThreadDNSAddressSeed(). This thread performs the work once and terminates. Initially, the stream determines whether the DNS hosts will be used. They are used only if at least one of the following conditions is met:
1) there is no hosts in the database;
2) the command line option --forcednsseed is explicitly specified;
3) 11 seconds after starting work, the number of fully established regular outgoing connections is less than 2.

After the decision is made to use DNS hosts, the list of such hosts is requested from the protocol

parameters which is hardcoded in the file logos/p2p/chainparams.cpp. At the present time, the list of seed hosts is empty. For each host, it first checks for the availability of the domain name xHH.seed_domain where HH is the hexadecimal bitmask of the required host parameters (this is inherited from Bitcoin and probably is outdated). If such a domain exists, then it is added to the database of known hosts. Otherwise, a one shot connection is established with the original seed_domain. After going through all the DNS hosts, the thread terminates.