Introduction:

Originally, our software would not initiate consensus (and failed to start), if the delegates could not connect directly via TCP. However, this requirement is not necessary, as consensus can proceed via p2p. This problem is described in LOGOS-166.

However, refactoring the code to initiate p2p consensus even without a direct connection was non-trivial. ConsensusNetIOManager would only create BackupDelegates if/when a TCP connection existed. If the TCP connection could not be established, no BackupDelegate’s were ever created, and if a TCP connection dies, the BackupDelegate’s were destroyed. Furthermore, ConsensusNetIO objects relied on an established TCP connection, and when the connection failed, they were destroyed.

The fix:

The fix to this problem was to create the BackupDelegate objects and ConsensusNetIO objects in ConsensusNetIOManager::Start(), and then bind the TCP connections to ConsensusNetIO later, when the connection is made. In addition, a timer is started, which upon expiration checks if the delegates are connected. If they are not connected, p2p consensus is enabled and consensus is initiated.

The handling of connection errors was changed significantly. ConsensusNetIO::OnNetIOError used to call ConsensusNetIOManager::OnNetIOError, which would destroy the old ConsensusNetIO object and create a new one. Now, ConsensusNetIO::OnNetIOError handles the error itself, with no help from ConsensusNetIOManager. ConsensusNetIO initiates a reconnect sequence, and upon connection, binds the socket.

During the reconnection sequence, a flag is set (\_connecting). If this flag is set to true, and ConsensusNetIO::OnNetIOError is called again, that function simply returns, instead of starting another reconnection sequence. A caveat is, epoch transition calls ConsensuNetIO::OnNetIOError, to close the connection permanently. To faciliate this, an additional flag (\_epoch\_over) is added, which signals whether the connection needs to die permanently. If \_connecting is false when OnNetIOError is called for epoch transition, the socket is closed. However, if \_connecting is true when OnNetIOError is called for epoch transition, meaning a reconnect sequence is currently happening, \_epoch\_over is set to true, and the function simply returns. When one of the callbacks in the reconnect sequence is invoked, it checks if \_epoch\_over is true; if \_epoch\_over is true, the reconnect is aborted and the socket is closed.

The handling of heartbeats has been moved to ConsensusNetIO::CheckHeartbeat(). This function locks the \_connection\_mutex and does nothing if \_connecting is true. The reason for this, is the heartbeat handling can cause a reconnect sequence. But if the software is already reconnecting, there is no sense checking heartbeats.

The destruction during epoch transition occurs in ConsensusNetIOManager::CleanUp(). This function clears the \_connections vector, which is a vector of ConsensusNetIO objects. However, these objects will not be destroyed if there are pending callbacks due to a reconnect sequence. This is because the callbacks capture “this” as a shared\_ptr. However, as stated above, every callback checks if the epoch has ended before proceeding.

The sychronization is handled by a mutex and two boolean flags (connected and connecting). Every function involved in a connect or reconnect sequence first locks the \_connection\_mutex before checking the state and proceeding.

ConsensusNetIOManager::OnNetIOError is now deleted. This function used to destroy ConsensusNetIO objects and create new ones, which is no longer being done. Epoch transition calls CleanUp(), which calls ConsensusNetIO::OnNetIOError() to close the socket, and then resets the shared\_ptrs to ConsensusNetIO objects.