```
using Neo.Core;
using Neo.Cryptography;
using Neo.IO;
using Neo.Network;
using Neo.Network.Payloads;
using Neo.Plugins;
using Neo.SmartContract;
using Neo.Wallets;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using DbgViewTR;

namespace Neo.Consensus
{
    public class ConsensusService : IDisposable
    {
        private ConsensusContext context = new ConsensusContext();
        private LocalNode localNode; //network.LocalNode.cs
        private Wallet wallet; //Neo.Wallets.Wallet.cs
        private Timer timer;
        private uint timer_height;
        private byte timer_view;
        private DateTime block_received_time;
        private bool started = false;

        public ConsensusService(LocalNode localNode, Wallet wallet)
        {
            TR.Enter();
            this.localNode = localNode;
            this.wallet = wallet;
            this.timer = new Timer(OnTimeout, null, Timeout.Infinite,
              Timeout.Infinite);
            TR.Exit();
        }

        private bool AddTransaction(Transaction tx, bool verify)
        {
            TR.Enter();
            if (Blockchain.Default.ContainsTransaction(tx.Hash) ||
                (verify && !tx.Verify(context.Transactions.Values)) ||
                !CheckPolicy(tx))
            {
                Log($"reject tx: {tx.Hash}{Environment.NewLine}{tx.ToArray
                  ().ToHexString()}");
                RequestChangeView();
                return TR.Exit(false);
            }
            context.Transactions[tx.Hash] = tx;
            if (context.TransactionHashes.Length == context.Transactions.Count)
            {
                if (Blockchain.GetConsensusAddress(Blockchain.Default.GetValidators
                  (context.Transactions.Values).ToArray()).Equals
                  (context.NextConsensus))
                {
                    Log($"send perpare response");
                    context.State |= ConsensusState.SignatureSent;
                    context.Signatures[context.MyIndex] = context.MakeHeader().Sign(context.KeyPair);
                    SignAndRelay(context.MakePrepareResponse(context.Signatures[context.MyIndex]));
                    CheckSignatures();
                }
                else
                {
                    RequestChangeView();
                    return TR.Exit(false);
                }
            }
            return TR.Exit(true);
        }

        private void Blockchain_PersistUnlocked(object sender, Block block)
        {
            TR.Enter();
            Log($"persist block: {block.Hash}");
            block_received_time = DateTime.Now;
            InitializeConsensus(0);
            TR.Exit();
        }

        private void CheckExpectedView(byte view_number)
        {
            TR.Enter();
            if (context.ViewNumber == view_number) return;
            if (context.ExpectedView.Count(p => p == view_number) >= context.M)
            {
                InitializeConsensus(view_number);
            }
            TR.Exit();
        }

        private bool CheckPolicy(Transaction tx)
        {
            TR.Enter();
            foreach (PolicyPlugin plugin in PolicyPlugin.Instances)
                if (!plugin.CheckPolicy(tx))
                    return TR.Exit(false);
            return TR.Exit(true);
        }

        private void CheckSignatures()
        {
            TR.Enter();
            if (context.Signatures.Count(p => p != null) >= context.M && context.TransactionHashes.All(p => context.Transactions.ContainsKey(p)))
            {
                Contract contract = Contract.CreateMultiSigContract(context.M, context.Validators);
                Block block = context.MakeHeader();
                ContractParametersContext sc = new ContractParametersContext(block);
                for (int i = 0, j = 0; i < context.Validators.Length && j < context.M; i++)
                    if (context.Signatures[i] != null)
                    {
                        sc.AddSignature(contract, context.Validators[i], context.Signatures[i]);
                        j++;
                    }
                sc.Verifiable.Scripts = sc.GetScripts();
                block.Transactions = context.TransactionHashes.Select(p => context.Transactions[p]).ToArray();
                Log($"relay block: {block.Hash}");
                if (!localNode.Relay(block))
                    Log($"reject block: {block.Hash}");
                context.State |= ConsensusState.BlockSent;
            }
            TR.Exit();
        }

        public void Dispose()
        {
            TR.Enter();
            Log("OnStop");
            if (timer != null) timer.Dispose();
            if (started)
            {
                Blockchain.PersistUnlocked -= Blockchain_PersistUnlocked;
                LocalNode.InventoryReceiving -= LocalNode_InventoryReceiving;
                LocalNode.InventoryReceived -= LocalNode_InventoryReceived;
            }
            TR.Exit();
        }

        private void FillContext()
        {
            TR.Enter();
            IEnumerable<Transaction> mem_pool = LocalNode.GetMemoryPool().Where(p => CheckPolicy(p));
            foreach (PolicyPlugin plugin in PolicyPlugin.Instances)
                mem_pool = plugin.Filter(mem_pool);
            List<Transaction> transactions = mem_pool.ToList();
            Fixed8 amount_netfee = Block.CalculateNetFee(transactions);
            TransactionOutput[] outputs = amount_netfee == Fixed8.Zero ? new TransactionOutput[0] : new[] { new TransactionOutput
            {
                AssetId = Blockchain.UtilityToken.Hash,
                Value = amount_netfee,
                ScriptHash = wallet.GetChangeAddress()
            } };
            while (true)
            {
                ulong nonce = GetNonce();
                MinerTransaction tx = new MinerTransaction
                {
                    Nonce = (uint)(nonce % (uint.MaxValue + 1ul)),
                    Attributes = new TransactionAttribute[0],
                    Inputs = new CoinReference[0],
                    Outputs = outputs,
                    Scripts = new Witness[0]
                };
                if (Blockchain.Default.GetTransaction(tx.Hash) == null)
                {
                    context.Nonce = nonce;
                    transactions.Insert(0, tx);
                    break;
                }
            }
            context.TransactionHashes = transactions.Select(p => p.Hash).ToArray();
            context.Transactions = transactions.ToDictionary(p => p.Hash);
            context.NextConsensus = Blockchain.GetConsensusAddress(Blockchain.Default.GetValidators(transactions).ToArray());
            TR.Exit();
        }

        private static ulong GetNonce()
        {
            TR.Enter();
            byte[] nonce = new byte[sizeof(ulong)];
            Random rand = new Random();
            rand.NextBytes(nonce);
            return TR.Exit(nonce.ToUInt64(0));
        }

        private void InitializeConsensus(byte view_number)
        {
            TR.Enter();
            lock (context)
            {
                if (view_number == 0)
                    context.Reset(wallet);
                else
                    context.ChangeView(view_number);
                if (context.MyIndex < 0) return;
                Log($"initialize: height={context.BlockIndex} view={view_number} index={context.MyIndex} role={(context.MyIndex == context.PrimaryIndex ? ConsensusState.Primary : ConsensusState.Backup)}");
                if (context.MyIndex == context.PrimaryIndex)
                {
                    context.State |= ConsensusState.Primary;
                    if (!context.State.HasFlag(ConsensusState.SignatureSent))
                    {
                        FillContext();
                    }
                    if (context.TransactionHashes.Length > 1)
                    {
                        InvPayload invPayload = InvPayload.Create(InventoryType.TX, context.TransactionHashes.Skip(1).ToArray());
                        foreach (RemoteNode node in localNode.GetRemoteNodes())
                            node.EnqueueMessage("inv", invPayload);
                    }
                    timer_height = context.BlockIndex;
                    timer_view = view_number;
                    TimeSpan span = DateTime.Now - block_received_time;
                    if (span >= Blockchain.TimePerBlock)
                        timer.Change(0, Timeout.Infinite);
                    else
                        timer.Change(Blockchain.TimePerBlock - span, Timeout.InfiniteTimeSpan);
                }
                else
                {
                    context.State = ConsensusState.Backup;
                    timer_height = context.BlockIndex;
                    timer_view = view_number;
                    timer.Change(TimeSpan.FromSeconds(Blockchain.SecondsPerBlock << (view_number + 1)), Timeout.InfiniteTimeSpan);
                }
            }
            TR.Exit();
        }

        private void LocalNode_InventoryReceived(object sender, IInventory inventory)
        {
            TR.Enter();
            ConsensusPayload payload = inventory as ConsensusPayload;
            if (payload != null)
            {
                lock (context)
                {
                    if (payload.ValidatorIndex == context.MyIndex) { TR.Exit(); return; }

                    if (payload.Version != ConsensusContext.Version)
                    {
                        TR.Exit();
                        return;
                    }
                    if (payload.PrevHash != context.PrevHash || payload.BlockIndex != context.BlockIndex)
                    {
                        // Request blocks

                        if (Blockchain.Default?.Height + 1 < payload.BlockIndex)
                        {
                            Log($"chain sync: expected={payload.BlockIndex} current: {Blockchain.Default?.Height} nodes={localNode.RemoteNodeCount}");

                            localNode.RequestGetBlocks();
                        }
                        TR.Exit();
                        return;
                    }

                    if (payload.ValidatorIndex >= context.Validators.Length) { TR.Exit(); return; }
                    ConsensusMessage message;
                    try
                    {
                        message = ConsensusMessage.DeserializeFrom(payload.Data);
                    }
                    catch
                    {
                        TR.Exit();
                        return;
                    }
                    if (message.ViewNumber != context.ViewNumber && message.Type != ConsensusMessageType.ChangeView)
                    {
                        TR.Exit();
                        return;
                    }
                    switch (message.Type)
                    {
                        case ConsensusMessageType.ChangeView:
                            OnChangeViewReceived(payload, (ChangeView)message);
                            break;
                        case ConsensusMessageType.PrepareRequest:
                            OnPrepareRequestReceived(payload, (PrepareRequest)message);
                            break;
                        case ConsensusMessageType.PrepareResponse:
                            OnPrepareResponseReceived(payload, (PrepareResponse)message);
                            break;
                    }
                }
            }
            TR.Exit();
        }

        private void LocalNode_InventoryReceiving(object sender, InventoryReceivingEventArgs e)
        {
            TR.Enter();
            Transaction tx = e.Inventory as Transaction;
            if (tx != null)
            {
                lock (context)
                {
                    if (!context.State.HasFlag(ConsensusState.Backup) || !context.State.HasFlag(ConsensusState.RequestReceived) || context.State.HasFlag(ConsensusState.SignatureSent) || context.State.HasFlag(ConsensusState.ViewChanging))
                        return;
                    if (context.Transactions.ContainsKey(tx.Hash)) return;
                    if (!context.TransactionHashes.Contains(tx.Hash)) return;
                    AddTransaction(tx, true);
                    e.Cancel = true;
                }
            }
            TR.Exit();
        }

        protected virtual void Log(string message)
        {
            // something should be here.
            TR.Enter();
            TR.Exit();
        }

        private void OnChangeViewReceived(ConsensusPayload payload, ChangeView message)
        {
            TR.Enter();
            Log($"{nameof(OnChangeViewReceived)}: height={payload.BlockIndex} view={message.ViewNumber} index={payload.ValidatorIndex} nv={message.NewViewNumber}");
            if (message.NewViewNumber <= context.ExpectedView[payload.ValidatorIndex])
            {
                TR.Exit();
                return;
            }
            context.ExpectedView[payload.ValidatorIndex] = message.NewViewNumber;
            CheckExpectedView(message.NewViewNumber);
            TR.Exit();
        }

        private void OnPrepareRequestReceived(ConsensusPayload payload, PrepareRequest message)
        {
            TR.Enter();
            Log($"{nameof(OnPrepareRequestReceived)}: height={payload.BlockIndex} view={message.ViewNumber} index={payload.ValidatorIndex} tx={message.TransactionHashes.Length}");
            if (!context.State.HasFlag(ConsensusState.Backup) || context.State.HasFlag(ConsensusState.RequestReceived))
            {
                TR.Exit();
                return;
            }
            if (payload.ValidatorIndex != context.PrimaryIndex) return;
            if (payload.Timestamp <= Blockchain.Default.GetHeader(context.PrevHash).Timestamp || payload.Timestamp > DateTime.Now.AddMinutes(10).ToTimestamp())
            {
                Log($"Timestamp incorrect: {payload.Timestamp}");
                TR.Exit();
                return;
            }
            context.State |= ConsensusState.RequestReceived;
            context.Timestamp = payload.Timestamp;
            context.Nonce = message.Nonce;
            context.NextConsensus = message.NextConsensus;
            context.TransactionHashes = message.TransactionHashes;
            context.Transactions = new Dictionary<UInt256, Transaction>();
            if (!Crypto.Default.VerifySignature(context.MakeHeader().GetHashData(), message.Signature, context.Validators[payload.ValidatorIndex].EncodePoint(false))) { TR.Exit(); return; }
            context.Signatures = new byte[context.Validators.Length][];
            context.Signatures[payload.ValidatorIndex] = message.Signature;
            Dictionary<UInt256, Transaction> mempool = LocalNode.GetMemoryPool().ToDictionary(p => p.Hash);
            foreach (UInt256 hash in context.TransactionHashes.Skip(1))
            {
                if (mempool.TryGetValue(hash, out Transaction tx))
                    if (!AddTransaction(tx, false))
                    {
                        TR.Exit();
                        return;
                    }
            }
            if (!AddTransaction(message.MinerTransaction, true)) { TR.Exit(); return; }
            if (context.Transactions.Count < context.TransactionHashes.Length)
            {
                UInt256[] hashes = context.TransactionHashes.Where(i => !context.Transactions.ContainsKey(i)).ToArray();
                LocalNode.AllowHashes(hashes);
                InvPayload msg = InvPayload.Create(InventoryType.TX, hashes);
                foreach (RemoteNode node in localNode.GetRemoteNodes())
                    node.EnqueueMessage("getdata", msg);
            }
            TR.Exit();
        }

        private void OnPrepareResponseReceived(ConsensusPayload payload, PrepareResponse message)
        {
            TR.Enter();
            Log($"{nameof(OnPrepareResponseReceived)}: height={payload.BlockIndex} view={message.ViewNumber} index={payload.ValidatorIndex}");
            if (context.State.HasFlag(ConsensusState.BlockSent)) { TR.Exit(); return; }
            if (context.Signatures[payload.ValidatorIndex] != null) { TR.Exit(); return; }
            Block header = context.MakeHeader();
            if (header == null || !Crypto.Default.VerifySignature(header.GetHashData(), message.Signature, context.Validators[payload.ValidatorIndex].EncodePoint(false))) { TR.Exit(); return; }
            context.Signatures[payload.ValidatorIndex] = message.Signature;
            CheckSignatures();
            TR.Exit();
        }

        private void OnTimeout(object state)
        {
            TR.Enter();
            lock (context)
            {
                if (timer_height != context.BlockIndex || timer_view != context.ViewNumber) { TR.Exit(); return; }
                Log($"timeout: height={timer_height} view={timer_view} state={context.State}");
                if (context.State.HasFlag(ConsensusState.Primary) && !
                context.State.HasFlag(ConsensusState.RequestSent))
                {
                    Log($"send perpare request: height={timer_height} view={timer_view}");
                    context.State |= ConsensusState.RequestSent;
                    if (!context.State.HasFlag(ConsensusState.SignatureSent))
                    {
                        context.Timestamp = Math.Max(DateTime.Now.ToTimestamp(), Blockchain.Default.GetHeader(context.PrevHash).Timestamp + 1);
                        context.Signatures[context.MyIndex] = context.MakeHeader().Sign(context.KeyPair);
                    }
                    SignAndRelay(context.MakePrepareRequest());
                    timer.Change(TimeSpan.FromSeconds(Blockchain.SecondsPerBlock << (timer_view + 1)), Timeout.InfiniteTimeSpan);
                }
                else if ((context.State.HasFlag(ConsensusState.Primary) && context.State.HasFlag(ConsensusState.RequestSent)) || context.State.HasFlag(ConsensusState.Backup))
                {
                    RequestChangeView();
                }
            }
            TR.Exit();
        }

        private void RequestChangeView()
        {
            TR.Enter();
            context.State |= ConsensusState.ViewChanging;
            context.ExpectedView[context.MyIndex]++;
            Log($"request change view: height={context.BlockIndex} view={context.ViewNumber} nv={context.ExpectedView[context.MyIndex]} state={context.State}");
            timer.Change(TimeSpan.FromSeconds(Blockchain.SecondsPerBlock << (context.ExpectedView[context.MyIndex] + 1)), Timeout.InfiniteTimeSpan);
            SignAndRelay(context.MakeChangeView());
            CheckExpectedView(context.ExpectedView[context.MyIndex]);
            TR.Exit();
        }

        private void SignAndRelay(ConsensusPayload payload)
        {
            TR.Enter();
            ContractParametersContext sc;
            try
            {
                sc = new ContractParametersContext(payload);
                wallet.Sign(sc);
            }
            catch (InvalidOperationException)
            {
                TR.Exit();
                return;
            }
            sc.Verifiable.Scripts = sc.GetScripts();
            localNode.RelayDirectly(payload);
            TR.Exit();
        }

        public void Start()
        {
            TR.Enter();
            Log("OnStart");
            started = true;
            Blockchain.PersistUnlocked += Blockchain_PersistUnlocked;
            LocalNode.InventoryReceiving += LocalNode_InventoryReceiving;
            LocalNode.InventoryReceived += LocalNode_InventoryReceived;
            InitializeConsensus(0);
            TR.Exit();
        }
    }
}
```