Tendermint Network Metering Reactor

Charles Dusek

May 3, 2020

Abstract

Decentralized network load balancing requires secure metering. In order to accomplish this a network metering reactor will be designed for the Tendermint consensus engine. This reactor will gather network measurements such as bandwidth, latency and packet loss from a set of neighbors and share those on a BFT CRDT.

1 Introduction

As the number of users increases on an autonomous Byzantine Fault Tolerant decentralized system, load balancing is necessary to meet critical time constraints that dictate user experience. The first step towards load balancing of a decentralized system is generalized network metering. This includes bandwidth, latency, loss, as well as RTT for various signed computational challenges to determine the nodes load.

Decentralized network coordinate methodologies are used to generate estimated maps of networks in a way that minimizes communication, increases fault tolerance and shares the load of computing by co-operative machine learning. The particular method that is used as part of this protocol may be generalized for any network measurement and is node-churn stable by using a self-stabilizing distributed maximum margin matrix factorization algorithm.

Within an adverserial environment, Byzantine node are incentivized to report false network measurements in order receive more rewards by reporting lower lower latency. Byzantine Fault Tolerant consensus allows for verification of tests and agreement on violations of the network metering protocol. Pair-wise network metering relies on nodes interacting and reporting network measurements in a truthful manner. In order to verify the measurements are taken and shared according to the protocol, an added verification step is needed that statistically samples the nodes on the system for faulty protocol implementations.

Each node entrant to the system will be accepted based on their network coordinate error or skew. Any node that is exhibiting faulty behavior will not accepted into the network and will be reported. Additionally, during each round, nodes will sample from their neighbor set to compare their neighbors coordinates with actual network measurements that are taken during the round. Any node exihibiting out of bounds error between coordinate estimated network measurements and actual mesaurement will be reported. Finally, reports of accepted nodes exhibiting faulty behavior will be verified through a consensus mechanism that verifies measurements between the the reporting node and the accused node

using third party nodes. If either the faulty node or a byzantine reporter are confirmed by consensus agreement agreement amonst accepted nodes that they are guilty of the accusation the convicted node will be blacklisted.

Algorithm 1: Network Metering Protocol - PLUG To Be Completed

Input: Your Input
Output: Your output
Data: Testing set x

```
1  ∑_{i=1}^{∞} := 0                                   // this is a comment
2  /* Now this is an if...else conditional loop                    */
3  if Condition 1 then
4      Do something                              // this is another comment
5      if sub-Condition then
6          Do a lot
7  else if Condition 2 then
8      Do Otherwise
9      /* Now this is a for loop                                   */
10     for sequence do
11         loop instructions
12 else
13     Do the rest
14 /* Now this is a While loop                                    */
15 while Condition do
16     Do something
```

Consensus within the network is assured through the Tendermint protocol. Tendermint is a Byzantine Fault Tolerant consensus engine that is used primarily to build blockchain applications. Tendermint uses a gossip protocol to maintain a membership list as well as disseminate messages. Tendermint consensus may tolerate only f faulty or adverserial nodes and proposals are committed by a 2f+1 vote.

The program structure of Tendermint follows the reactor pattern. Early version of the reactor pattern may be found within the book "Pattern Languages of Program Design" by Jim Coplien and Douglas Schmidt in 1995. An updated version was written by Douglas Schmidt called Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events.

1.1 Command Line Interface

Tendermint uses the Cobra command line interface commander. Cobra provides the following features:

- Easy subcommand-based CLIs: app server, app fetch, etc.
- Fully POSIX-compliant flags (including short and long versions)
- Nested subcommands
- Global, local and cascading flags

- Easy generation of applications and commands with cobra init appname and cobra add cmdname
- Intelligent suggestions (app srver... did you mean app server?)
- Automatic help generation for commands and flags
- Automatic help flag recognition of -h, –help, etc.
- Automatically generated bash autocomplete for your application
- Automatically generated man pages for your application
- Command aliases so you can change things without breaking them
- The flexibility to define your own help, usage, etc.
- Optional tight integration with viper for 12-factor apps

1.1.1 Main

While you are welcome to provide your own organization, typically a Cobra-based application will follow the following organizational structure:

- appName
  - cmd
    - add.go
    - your.go
    - commands.go
    - here.go
  - main.go

Tendermint uses two main files main.go and a root.go to initialize the commands. The root command for the Tendermint core is found within the ./cmd/-tendermint/commands/root.go file:

1.1.2 Root

The RootCmd is found within the ./cmd/tendermint/commands/root.go file:

```
// RootCmd is the root command for Tendermint core.
var RootCmd = &cobra.Command{
  Use:   "tendermint",
  Short: "Tendermint Core (BFT Consensus) in Go",
  PersistentPreRunE: func(cmd *cobra.Command, args []string)
      (err error) {
    if cmd.Name() == VersionCmd.Name() {
      return nil
    }
    config, err = ParseConfig()
    if err != nil {
      return err
    }
    if config.LogFormat == cfg.LogFormatJSON {
      logger = log.NewTMJSONLogger(log.NewSyncWriter(os.
          Stdout))
    }
    logger, err = tmflags.ParseLogLevel(config.LogLevel,
        logger, cfg.DefaultLogLevel())
    if err != nil {
      return err
    }
    if viper.GetBool(cli.TraceFlag) {
      logger = log.NewTracingLogger(logger)
    }
    logger = logger.With("module", "main")
    return nil
  },
}
```

1.1.3 Main

The main.go file is where all of the commands and node functionality are integrated with the root command. It is located in the ./cmd/tendermint/main.go file:

```
package main

import (
  "os"
  "path/filepath"

  cmd "github.com/tendermint/tendermint/cmd/tendermint/
      commands"
  "github.com/tendermint/tendermint/cmd/tendermint/commands/
      debug"
  cfg "github.com/tendermint/tendermint/config"
  "github.com/tendermint/tendermint/libs/cli"
  nm "github.com/tendermint/tendermint/node"
)

func main() {
```

```
  rootCmd := cmd.RootCmd
  rootCmd.AddCommand(
    cmd.GenValidatorCmd,
    cmd.InitFilesCmd,
    cmd.ProbeUpnpCmd,
    cmd.LiteCmd,
    cmd.ReplayCmd,
    cmd.ReplayConsoleCmd,
    cmd.ResetAllCmd,
    cmd.ResetPrivValidatorCmd,
    cmd.ShowValidatorCmd,
    cmd.TestnetFilesCmd,
    cmd.ShowNodeIDCmd,
    cmd.GenNodeKeyCmd,
    cmd.VersionCmd,
    debug.DebugCmd,
    cli.NewCompletionCmd(rootCmd, true),
  )

  // NOTE:
  // Users wishing to:
  //  * Use an external signer for their validators
  //  * Supply an in-proc abci app
  //  * Supply a genesis doc file from another source
  //  * Provide their own DB implementation
  // can copy this file and use something other than the
  // DefaultNewNode function
  nodeFunc := nm.DefaultNewNode

  // Create and start node
  rootCmd.AddCommand(cmd.NewRunNodeCmd(nodeFunc))

  cmd := cli.PrepareBaseCmd(rootCmd, "TM", os.ExpandEnv(
      filepath.Join("$HOME", cfg.DefaultTendermintDir)))
  if err := cmd.Execute(); err != nil {
    panic(err)
  }
}
```

1.2 tendermint init

The "tendermint init" command calls the InitFilesCmd function. InitFilesCmd initialises a fresh Tendermint Core instance. The InitFileCmd cobra command variable uses the handle "init" and is found within the ./cmd/tendermint/commands/init.go file:

```
var InitFilesCmd = &cobra.Command{
  Use:   "init",
  Short: "Initialize Tendermint",
  RunE:  initFiles,
}

func initFiles(cmd *cobra.Command, args []string) error {
  return initFilesWithConfig(config)
}

func initFilesWithConfig(config *cfg.Config) error {
  // private validator
  privValKeyFile := config.PrivValidatorKeyFile()
  privValStateFile := config.PrivValidatorStateFile()
  var pv *privval.FilePV
  if tmos.FileExists(privValKeyFile) {
    pv = privval.LoadFilePV(privValKeyFile, privValStateFile
        )
    logger.Info("Found private validator", "keyFile",
        privValKeyFile,
      "stateFile", privValStateFile)
  } else {
    pv = privval.GenFilePV(privValKeyFile, privValStateFile)
    pv.Save()
    logger.Info("Generated private validator", "keyFile",
        privValKeyFile,
      "stateFile", privValStateFile)
  }

  nodeKeyFile := config.NodeKeyFile()
  if tmos.FileExists(nodeKeyFile) {
    logger.Info("Found node key", "path", nodeKeyFile)
  } else {
    if _, err := p2p.LoadOrGenNodeKey(nodeKeyFile); err !=
        nil {
      return err
    }
    logger.Info("Generated node key", "path", nodeKeyFile)
  }

  // genesis file
  genFile := config.GenesisFile()
  if tmos.FileExists(genFile) {
    logger.Info("Found genesis file", "path", genFile)
  } else {
    genDoc := types.GenesisDoc{
      ChainID:         fmt.Sprintf("test-chain-%v", tmrand.
          Str(6)),
      GenesisTime:     tmtime.Now(),
      ConsensusParams: types.DefaultConsensusParams(),
    }
    pubKey, err := pv.GetPubKey()
    if err != nil {
      return errors.Wrap(err, "can't get pubkey")
```

```
    }
    genDoc.Validators = []types.GenesisValidator{{
      Address: pubKey.Address(),
      PubKey:  pubKey,
      Power:   10,
    }}

    if err := genDoc.SaveAs(genFile); err != nil {
      return err
    }
    logger.Info("Generated genesis file", "path", genFile)
  }

  return nil
}
```

1.3 tendermint node

NewRunNodeCmd returns the command that allows the CLI to start a node. It can be used with a custom PrivValidator and in-process ABCI application. NewRunNodeCmd is found in ./cmd/tendermint/commands/run_node.go:

```
func NewRunNodeCmd(nodeProvider nm.Provider) *cobra.Command
    {
  cmd := &cobra.Command{
    Use:   "node",
    Short: "Run the tendermint node",
    RunE: func(cmd *cobra.Command, args []string) error {
      if err := checkGenesisHash(config); err != nil {
        return err
      }

      n, err := nodeProvider(config, logger)
      if err != nil {
        return fmt.Errorf("failed to create node: %w", err)
      }

      if err := n.Start(); err != nil {
        return fmt.Errorf("failed to start node: %w", err)
      }

      logger.Info("Started node", "nodeInfo", n.Switch().
          NodeInfo())

      // Stop upon receiving SIGTERM or CTRL-C.
      tmos.TrapSignal(logger, func() {
        if n.IsRunning() {
          n.Stop()
        }
      })

      // Run forever.
      select {}
    },
  }

  AddNodeFlags(cmd)
  return cmd
}
```

The NewRunNodeCmd function takes the nodeProvider as parameter with type nm.Provider. The nm.Provider type is found in the "node" go package which is the main entry point, where the Node struct, which represents a full node, is defined. The Provider is imported into the ./cmd/main.go file as a function whose default is DefaultNewNode.

Provider takes a config and a logger and returns a ready to go Node. The nm.Provider type is found in the ./node/node.go file:

```
type Provider func(*cfg.Config, log.Logger) (*Node, error)
```

DefaultNewNode, which is also found in the ./node/node.go file, returns a Tendermint node with default settings for the PrivValidator, ClientCreator, GenesisDoc, and DBProvider. It implements NodeProvider.

```
func DefaultNewNode(config *cfg.Config, logger log.Logger)
    (*Node, error) {
  // Generate node PrivKey
  nodeKey, err := p2p.LoadOrGenNodeKey(config.NodeKeyFile())
  if err != nil {
    return nil, err
  }

  // Convert old PrivValidator if it exists.
  oldPrivVal := config.OldPrivValidatorFile()
  newPrivValKey := config.PrivValidatorKeyFile()
  newPrivValState := config.PrivValidatorStateFile()
  if _, err := os.Stat(oldPrivVal); !os.IsNotExist(err) {
    oldPV, err := privval.LoadOldFilePV(oldPrivVal)
    if err != nil {
      return nil, fmt.Errorf("error reading OldPrivValidator
          from %v: %v", oldPrivVal, err)
    }
    logger.Info("Upgrading PrivValidator file",
      "old", oldPrivVal,
      "newKey", newPrivValKey,
      "newState", newPrivValState,
    )
    oldPV.Upgrade(newPrivValKey, newPrivValState)
  }

  return NewNode(config,
    privval.LoadOrGenFilePV(newPrivValKey, newPrivValState),
    nodeKey,
    proxy.DefaultClientCreator(config.ProxyApp, config.ABCI,
        config.DBDir()),
    DefaultGenesisDocProviderFunc(config),
    DefaultDBProvider,
    DefaultMetricsProvider(config.Instrumentation),
    logger,
  )
}
```

The Node type defines the highest level interface to a full Tendermint node. It includes all configuration and running services.

```
type Node struct {
  service.BaseService

  // config
```

```
  config        *cfg.Config
  genesisDoc    *types.GenesisDoc   // initial validator set
  privValidator types.PrivValidator // local node's
      validator key

  // network
  transport   *p2p.MultiplexTransport
  sw          *p2p.Switch  // p2p connections
  addrBook    pex.AddrBook // known peers
  nodeInfo    p2p.NodeInfo
  nodeKey     *p2p.NodeKey // our node privkey
  isListening bool

  // services
  eventBus          *types.EventBus // pub/sub for services
  stateDB           dbm.DB
  blockStore        *store.BlockStore // store the blockchain
       to disk
  bcReactor         p2p.Reactor       // for fast-syncing
  mempoolReactor    *mempl.Reactor    // for gossipping
      transactions
  mempool           mempl.Mempool
  consensusState    *cs.State      // latest consensus state
  consensusReactor *cs.Reactor    // for participating in
      the consensus
  pexReactor        *pex.Reactor   // for exchanging peer
      addresses
  evidencePool      *evidence.Pool // tracking evidence
  proxyApp          proxy.AppConns // connection to the
      application
  rpcListeners      []net.Listener // rpc servers
  txIndexer         txindex.TxIndexer
  indexerService    *txindex.IndexerService
  prometheusSrv     *http.Server
}
```

At the heart of Tendermint is the NewNode function. The NewNode function returns an object of the type Node that represents the Tendermint Node. The NewNode function is rather lengthy so it will be broken down into sections by function beginning with parameters to end. The NewNode function is found within the ./node/node.go file:

1.3.1 Parameters

```
func NewNode(
  config *cfg.Config,
  privValidator types.PrivValidator,
  nodeKey *p2p.NodeKey,
  clientCreator proxy.ClientCreator,
  genesisDocProvider GenesisDocProvider,
  dbProvider DBProvider,
  metricsProvider MetricsProvider,
  logger log.Logger,
  options ...Option
) (*Node, error) { ...
```

- config *cfg.Config
  Config is imported into the ./cmd/main.go file ...

- privValidator types.PrivValidator
  The privValidator is generated in NewNode with the function privval.LoadOrGenFilePV(newPrivValKey newPrivValState)

- nodeKey *p2p.NodeKey
  Pointer to the node's private key

- clientCreator proxy.ClientCreator
  Proxy is imported into the ./node/node.go and clientCreator is generated with the function proxy.DefaultClientCreator(config.ProxyApp, config.ABCI, config.DBDir())

- genesisDocProvider GenesisDocProvider
  Generated by the DefaultGenesisDocProviderFunc(config) from config. GenesisDocProvider returns a GenesisDoc. It allows the GenesisDoc to be pulled from sources other than the filesystem, for instance from a distributed key-value store cluster. DefaultGenesisDocProviderFunc returns a GenesisDocProvider that loads the GenesisDoc from the config.GenesisFile() on the filesystem.

  ```
  type GenesisDocProvider func() (*types.GenesisDoc, error
      )

  func DefaultGenesisDocProviderFunc(config *cfg.Config)
      GenesisDocProvider {
    return func() (*types.GenesisDoc, error) {
      return types.GenesisDocFromFile(config.GenesisFile()
          )
    }
  }
  ```

- dbProvider DBProvider
  DBProvider takes a DBContext and returns instantiated DBs.

  ```
  type DBProvider func(*DBContext) (dbm.DB, error)
  ```

- metricsProvider MetricsProvider

```
// MetricsProvider returns a consensus, p2p and mempool
    Metrics.
type MetricsProvider func(chainID string) (*cs.Metrics,
    *p2p.Metrics, *mempl.Metrics, *sm.Metrics)

// DefaultMetricsProvider returns Metrics build using
    Prometheus client library
// if Prometheus is enabled. Otherwise, it returns no-op
     Metrics.
func DefaultMetricsProvider(config *cfg.
    InstrumentationConfig) MetricsProvider {
  return func(chainID string) (*cs.Metrics, *p2p.Metrics
      , *mempl.Metrics, *sm.Metrics) {
    if config.Prometheus {
      return cs.PrometheusMetrics(config.Namespace, "
          chain_id", chainID),
        p2p.PrometheusMetrics(config.Namespace, "
            chain_id", chainID),
        mempl.PrometheusMetrics(config.Namespace, "
            chain_id", chainID),
        sm.PrometheusMetrics(config.Namespace, "chain_id
            ", chainID)
    }
    return cs.NopMetrics(), p2p.NopMetrics(), mempl.
        NopMetrics(), sm.NopMetrics()
  }
}
```

- logger log.Logger

- options ...Option

1.3.2 Initialize Databases

The blockStore and stateDB are generated in the following function:

```
blockStore, stateDB, err := initDBs(config, dbProvider)
if err != nil {
  return nil, err
}
```

The initDBs(config, dbProvider) function is also found in the ./node/node.go file:

```
func initDBs(config *cfg.Config, dbProvider DBProvider) (
    blockStore *store.BlockStore, stateDB dbm.DB, err error)
    {
  var blockStoreDB dbm.DB
  blockStoreDB, err = dbProvider(&DBContext{"blockstore",
      config})
  if err != nil {
    return
  }
  blockStore = store.NewBlockStore(blockStoreDB)

  stateDB, err = dbProvider(&DBContext{"state", config})
```

```
  if err != nil {
    return
  }

  return
}
```

The DBContext struct type, dbProvider func type, and the DefaultDBProvider function are also declared in the ./node/node.go file:

```
type DBContext struct {
  ID     string
  Config *cfg.Config
}

type DBProvider func(*DBContext) (dbm.DB, error)

func DefaultDBProvider(ctx *DBContext) (dbm.DB, error) {
  dbType := dbm.BackendType(ctx.Config.DBBackend)
  return dbm.NewDB(ctx.ID, dbType, ctx.Config.DBDir()), nil
}
```

Todo: Fill out the DB portion here starting with dbm.BackendTye()

1.3.3 Initialize State

The state and genDoc are generated with the following function:

```
state, genDoc, err := LoadStateFromDBOrGenesisDocProvider(
    stateDB, genesisDocProvider)
if err != nil {
  return nil, err
}
```

LoadStateFromDBOrGenesisDocProvider attempts to load the state from the database, or creates one using the given genesisDocProvider and persists the result to the database. On success this also returns the genesis doc loaded through the given provider. It is also found within the ./node/node.go file

```
func LoadStateFromDBOrGenesisDocProvider(
  stateDB dbm.DB,
  genesisDocProvider GenesisDocProvider,
) (sm.State, *types.GenesisDoc, error) {
  // Get genesis doc
  genDoc, err := loadGenesisDoc(stateDB)
  if err != nil {
    genDoc, err = genesisDocProvider()
    if err != nil {
      return sm.State{}, nil, err
    }
    // save genesis doc to prevent a certain class of user
        errors (e.g. when it
    // was changed, accidentally or not). Also good for
        audit trail.
    saveGenesisDoc(stateDB, genDoc)
  }
  state, err := sm.LoadStateFromDBOrGenesisDoc(stateDB,
      genDoc)
  if err != nil {
    return sm.State{}, nil, err
  }
  return state, genDoc, nil
}
```

If state already exists with the stateDB then the following function will load it:

```
// panics if failed to unmarshal bytes
func loadGenesisDoc(db dbm.DB) (*types.GenesisDoc, error) {
  b, err := db.Get(genesisDocKey)
  if err != nil {
    panic(err)
  }
  if len(b) == 0 {
    return nil, errors.New("genesis doc not found")
  }
  var genDoc *types.GenesisDoc
  err = cdc.UnmarshalJSON(b, &genDoc)
  if err != nil {
    panic(fmt.Sprintf("Failed to load genesis doc due to
        unmarshaling error: %v (bytes: %X)", err, b))
  }
  return genDoc, nil
}
```

This function saves the genesis doc to prevent a certain class of user errors (e.g. when it was changed, accidentally or not). Also good for audit trail.

```
// panics if failed to marshal the given genesis document
func saveGenesisDoc(db dbm.DB, genDoc *types.GenesisDoc) {
  b, err := cdc.MarshalJSON(genDoc)
  if err != nil {
    panic(fmt.Sprintf("Failed to save genesis doc due to
        marshaling error: %v", err))
  }
  db.SetSync(genesisDocKey, b)
}
```

1.3.4 Create Proxy and ABCI Connections

Create the proxyApp and establish connections to the ABCI app (consensus, mempool, query).

```
proxyApp, err := createAndStartProxyAppConns(clientCreator,
    logger)
if err != nil {
    return nil, err
}
```

Add parameters in Config file here

The helper function for Proxy creation is below:

```
func createAndStartProxyAppConns(clientCreator proxy.
    ClientCreator, logger log.Logger) (proxy.AppConns, error)
     {
  proxyApp := proxy.NewAppConns(clientCreator)
  proxyApp.SetLogger(logger.With("module", "proxy"))
  if err := proxyApp.Start(); err != nil {
    return nil, fmt.Errorf("error starting proxy app
        connections: %v", err)
  }
  return proxyApp, nil
}
```

NewAppConns is located in the ./proxy/multi_app_conn.go file:

```
func NewAppConns(clientCreator ClientCreator) AppConns {
  return NewMultiAppConn(clientCreator)
}

//------------------------------
// multiAppConn implements AppConns

// a multiAppConn is made of a few appConns (mempool,
    consensus, query)
// and manages their underlying abci clients
// TODO: on app restart, clients must reboot together
type multiAppConn struct {
  service.BaseService

  mempoolConn   AppConnMempool
  consensusConn AppConnConsensus
  queryConn     AppConnQuery

  clientCreator ClientCreator
}

// Make all necessary abci connections to the application
func NewMultiAppConn(clientCreator ClientCreator) AppConns {
  multiAppConn := &multiAppConn{
    clientCreator: clientCreator,
  }
  multiAppConn.BaseService = *service.NewBaseService(nil, "
      multiAppConn", multiAppConn)
  return multiAppConn
}
```

The ClientCreator type is found within the ./proxy/client.go file:

```
type ClientCreator interface {
  NewABCIClient() (abcicli.Client, error)
}
```

The DefaultNewNode function inputs the proxy.DefaultClientCreator as the clientCreator parameter. The DefaultClientCreator is found within the ./proxy/client.go file:

```
func DefaultClientCreator(addr, transport, dbDir string)
    ClientCreator {
  switch addr {
  case "counter":
    return NewLocalClientCreator(counter.NewApplication(
        false))
  case "counter_serial":
    return NewLocalClientCreator(counter.NewApplication(true
        ))
  case "kvstore":
    return NewLocalClientCreator(kvstore.NewApplication())
  case "persistent_kvstore":
    return NewLocalClientCreator(kvstore.
        NewPersistentKVStoreApplication(dbDir))
  case "noop":
    return NewLocalClientCreator(types.NewBaseApplication())
  default:
    mustConnect := false // loop retrying
    return NewRemoteClientCreator(addr, transport,
        mustConnect)
  }
}
```

NewABCIClient returns newly connected client that is also found within the ./proxy/client.go file:

```
func (r *remoteClientCreator) NewABCIClient() (abcicli.
    Client, error) {
  remoteApp, err := abcicli.NewClient(r.addr, r.transport, r
      .mustConnect)
  if err != nil {
    return nil, errors.Wrap(err, "Failed to connect to proxy
        ")
  }
  return remoteApp, nil
}
```

NewClient returns a new ABCI client of the specified transport type. It returns an error if the transport is not "socket" or "grpc". The NewClient function is found in the ./abci/client.go file.

```
func NewClient(addr, transport string, mustConnect bool) (
    client Client, err error) {
  switch transport {
  case "socket":
    client = NewSocketClient(addr, mustConnect)
  case "grpc":
    client = NewGRPCClient(addr, mustConnect)
  default:
    err = fmt.Errorf("unknown abci transport %s", transport)
  }
  return
}
```

1.3.5 Event Bus and Indexer Service

EventBus and IndexerService must be started before the handshake because we might need to index the txs of the replayed block as this might not have happened when the node stopped last time (i.e. the node stopped after it saved the block but before it indexed the txs, or, endblocker panicked)

```
eventBus, err := createAndStartEventBus(logger)
if err != nil {
  return nil, err
}

// Transaction indexing
indexerService, txIndexer, err :=
    createAndStartIndexerService(config, dbProvider, eventBus
    , logger)
if err != nil {
  return nil, err
}
```

The createAndStartEventBus(logger) function is found within the ./node/node.go file:

```
func createAndStartEventBus(logger log.Logger) (*types.
    EventBus, error) {
  eventBus := types.NewEventBus()
  eventBus.SetLogger(logger.With("module", "events"))
  if err := eventBus.Start(); err != nil {
    return nil, err
  }
```

```
  return eventBus, nil
}
```

EventBus is a common bus for all events going through the system. All calls are proxied to underlying pubsub server. All events must be published using EventBus to ensure correct data types. The EventBus functionality below is found within the ./types/event_bus.go file:

```
type EventBus struct {
  service.BaseService
  pubsub *tmpubsub.Server
}

// NewEventBus returns a new event bus.
func NewEventBus() *EventBus {
  return NewEventBusWithBufferCapacity(defaultCapacity)
}

// NewEventBusWithBufferCapacity returns a new event bus
    with the given buffer capacity.
func NewEventBusWithBufferCapacity(cap int) *EventBus {
  // capacity could be exposed later if needed
  pubsub := tmpubsub.NewServer(tmpubsub.BufferCapacity(cap))
  b := &EventBus{pubsub: pubsub}
  b.BaseService = *service.NewBaseService(nil, "EventBus", b
      )
  return b
}

func (b *EventBus) SetLogger(l log.Logger) {
  b.BaseService.SetLogger(l)
  b.pubsub.SetLogger(l.With("module", "pubsub"))
}
```

Key value indexer is default. It is the simplest possible indexer backed by key-value storage which defaults to LevelDB.

```
// TxIndexConfig
// Remember that Event has the following structure:
// type: [
//  key: value,
//  ...
// ]
//
// CompositeKeys are constructed by `type.key`
// TxIndexConfig defines the configuration for the
    transaction indexer,
// including composite keys to index.
type TxIndexConfig struct {
  // What indexer to use for transactions
  //
  // Options:
  //   1) "null"
  //   2) "kv" (default) - the simplest possible indexer,
  //      backed by key-value storage (defaults to levelDB;
      see DBBackend).
  Indexer string `mapstructure:"indexer"`

  // Comma-separated list of compositeKeys to index (by
      default the only key is "tx.hash")
  //
```

```
  // You can also index transactions by height by adding "tx
      .height" key here.
  //
  // It's recommended to index only a subset of keys due to
      possible memory
  // bloat. This is, of course, depends on the indexer's DB
      and the volume of
  // transactions.
  IndexKeys string `mapstructure:"index_keys"`

  // When set to true, tells indexer to index all
      compositeKeys (predefined keys:
  // "tx.hash", "tx.height" and all keys from DeliverTx
      responses).
  //
  // Note this may be not desirable (see the comment above).
       IndexKeys has a
  // precedence over IndexAllKeys (i.e. when given both,
      IndexKeys will be
  // indexed).
  IndexAllKeys bool `mapstructure:"index_all_keys"`
}

// DefaultTxIndexConfig returns a default configuration for
    the transaction indexer.
func DefaultTxIndexConfig() *TxIndexConfig {
  return &TxIndexConfig{
    Indexer:      "kv",
    IndexKeys:    "",
    IndexAllKeys: false,
  }
}

// TestTxIndexConfig returns a default configuration for the
     transaction indexer.
func TestTxIndexConfig() *TxIndexConfig {
  return DefaultTxIndexConfig()
}
```

1.3.6 Handshaker

Handshaker calls RequestInfo, sets the AppVersion on the state, and replays any blocks as necessary to sync tendermint with the app.

```
consensusLogger := logger.With("module", "consensus")
if err := doHandshake(stateDB, state, blockStore, genDoc,
    eventBus, proxyApp, consensusLogger); err != nil {
  return nil, err
}
```

The doHandshake function is also found within the ./node/node.go file:

```
func doHandshake(
  stateDB dbm.DB,
  state sm.State,
  blockStore sm.BlockStore,
  genDoc *types.GenesisDoc,
  eventBus types.BlockEventPublisher,
  proxyApp proxy.AppConns,
  consensusLogger log.Logger) error {

  handshaker := cs.NewHandshaker(stateDB, state, blockStore,
      genDoc)
```

```
  handshaker.SetLogger(consensusLogger)
  handshaker.SetEventBus(eventBus)
  if err := handshaker.Handshake(proxyApp); err != nil {
    return fmt.Errorf("error during handshake: %v", err)
  }
  return nil
}
```

The NewHandShaker function handshakes with the app to figure out where it was last and uses the WAL to recover from there. NewHandShaker is locate in ./consensus/replay.go

```
type Handshaker struct {
  stateDB      dbm.DB
  initialState sm.State
  store        sm.BlockStore
  eventBus     types.BlockEventPublisher
  genDoc       *types.GenesisDoc
  logger       log.Logger

  nBlocks int // number of blocks applied to the state
}

func NewHandshaker(stateDB dbm.DB, state sm.State,
  store sm.BlockStore, genDoc *types.GenesisDoc) *Handshaker
      {

  return &Handshaker{
    stateDB:      stateDB,
    initialState: state,
    store:        store,
    eventBus:     types.NopEventBus{},
    genDoc:       genDoc,
    logger:       log.NewNopLogger(),
    nBlocks:      0,
  }
}

func (h *Handshaker) SetLogger(l log.Logger) {
  h.logger = l
}

// SetEventBus - sets the event bus for publishing block
    related events.
// If not called, it defaults to types.NopEventBus.
func (h *Handshaker) SetEventBus(eventBus types.
    BlockEventPublisher) {
  h.eventBus = eventBus
}
```

The Handshake is done via ABCI Info on the query connection. It then retrieves the last block height and hash. Next it sets the Appversion on the state. Finally is replays the blocks up to the lastest in the blockstore. The Handshake function is also found within the ./consensus/replay.go file:

```
func (h *Handshaker) Handshake(proxyApp proxy.AppConns)
    error {

  // Handshake is done via ABCI Info on the query conn.
  res, err := proxyApp.Query().InfoSync(proxy.RequestInfo)
  if err != nil {
    return fmt.Errorf("error calling Info: %v", err)
  }
```

```
  blockHeight := res.LastBlockHeight
  if blockHeight < 0 {
    return fmt.Errorf("got a negative last block height (%d)
         from the app", blockHeight)
  }
  appHash := res.LastBlockAppHash

  h.logger.Info("ABCI Handshake App Info",
    "height", blockHeight,
    "hash", fmt.Sprintf("%X", appHash),
    "software-version", res.Version,
    "protocol-version", res.AppVersion,
  )

  // Set AppVersion on the state.
  if h.initialState.Version.Consensus.App != version.
      Protocol(res.AppVersion) {
    h.initialState.Version.Consensus.App = version.Protocol(
        res.AppVersion)
    sm.SaveState(h.stateDB, h.initialState)
  }

  // Replay blocks up to the latest in the blockstore.
  _, err = h.ReplayBlocks(h.initialState, appHash,
      blockHeight, proxyApp)
  if err != nil {
    return fmt.Errorf("error on replay: %v", err)
  }

  h.logger.Info("Completed ABCI Handshake - Tendermint and
      App are synced",
    "appHeight", blockHeight, "appHash", fmt.Sprintf("%X",
        appHash))

  // TODO: (on restart) replay mempool

  return nil
}
```

Todo: add replay blocks section. Out of scope for now.

1.3.7 Reload State

Reload the state. It will have the Version.Consensus.App set by the Handshake, and may have other modifications as well (ie. depending on what happened during block replay).

```
state = sm.LoadState(stateDB)
```

The LoadState function is found within the ./state/store.go file:

```
// LoadState loads the State from the database.
func LoadState(db dbm.DB) State {
  return loadState(db, stateKey)
}

func loadState(db dbm.DB, key []byte) (state State) {
  buf, err := db.Get(key)
  if err != nil {
    panic(err)
  }
```

```
  if len(buf) == 0 {
    return state
  }

  err = cdc.UnmarshalBinaryBare(buf, &state)
  if err != nil {
    // DATA HAS BEEN CORRUPTED OR THE SPEC HAS CHANGED
    tmos.Exit(fmt.Sprintf(`LoadState: Data has been
        corrupted or its spec has changed:
                %v\n`, err))
  }
  // TODO: ensure that buf is completely read.

  return state
}
```

1.3.8 Mempool Reactor

Mempool maintains a cache of the last 10000 transactions to prevent replaying old transactions (plus transactions coming from other validators, who are continually exchanging transactions). Sending incorrectly encoded data or data exceeding maxMsgSize will result in stopping the peer.

The mempool will not send a tx back to any peer which it received it from.

The reactor assigns an uint16 number for each peer and maintains a map from p2p.ID to uint16. Each mempool transaction carries a list of all the senders([]uint16) The list is updated every time a transaction it is already seen. uint16 assumes that a node will never have over 65535 peers (0 is reserved for unknown source - e.g. RPC)

```
mempoolReactor, mempool := createMempoolAndMempoolReactor(
    config, proxyApp, state, memplMetrics, logger)

\\ Also found within the node.go file
func createMempoolAndMempoolReactor(config *cfg.Config,
    proxyApp proxy.AppConns,
  state sm.State, memplMetrics *mempl.Metrics, logger log.
      Logger) (*mempl.Reactor, *mempl.CListMempool) {

  mempool := mempl.NewCListMempool(
    config.Mempool,
    proxyApp.Mempool(),
    state.LastBlockHeight,
    mempl.WithMetrics(memplMetrics),
    mempl.WithPreCheck(sm.TxPreCheck(state)),
    mempl.WithPostCheck(sm.TxPostCheck(state)),
  )
  mempoolLogger := logger.With("module", "mempool")
  mempoolReactor := mempl.NewReactor(config.Mempool, mempool
      )
  mempoolReactor.SetLogger(mempoolLogger)

  if config.Consensus.WaitForTxs() {
    mempool.EnableTxsAvailable()
  }
  return mempoolReactor, mempool
}
```

The the mempool is created with the NewCListMempool function. It create a new CListMempool, which is an ordered in-memory pool for transactions before

they are proposed in a consensus round. Transaction validity is checked using the CheckTx abci message before the transaction is added to the pool. The mempool uses a concurrent list structure for storing transactions that can be efficiently accessed by multiple concurrent readers.

The CListMempool has the following type which may be found in the ./mempool/clist_mempool.go file:

```
type CListMempool struct {
  // Atomic integers
  height     int64 // the last block Update()'d to
  txsBytes   int64 // total size of mempool, in bytes
  rechecking int32 // for re-checking filtered txs on Update
      ()

  // notify listeners (ie. consensus) when txs are available
  notifiedTxsAvailable bool
  txsAvailable         chan struct{} // fires once for each
      height, when the mempool is not empty

  config *cfg.MempoolConfig

  proxyMtx     sync.Mutex
  proxyAppConn proxy.AppConnMempool
  txs          *clist.CList // concurrent linked-list of
      good txs
  preCheck     PreCheckFunc
  postCheck    PostCheckFunc

  // Track whether we're rechecking txs.
  // These are not protected by a mutex and are expected to
      be mutated
  // in serial (ie. by abci responses which are called in
      serial).
  recheckCursor *clist.CElement // next expected response
  recheckEnd    *clist.CElement // re-checking stops here

  // Map for quick access to txs to record sender in CheckTx
      .
  // txsMap: txKey -> CElement
  txsMap sync.Map

  // Keep a cache of already-seen txs.
  // This reduces the pressure on the proxyApp.
  cache txCache

  // A log of mempool txs
  wal *auto.AutoFile

  logger log.Logger

  metrics *Metrics
}
```

The NewClistMempool function returns a new mempool with the given configuration and connection to an application. It may be found within the ./mempool/clist_mempool.go file:

```
func NewCListMempool(
  config *cfg.MempoolConfig,
  proxyAppConn proxy.AppConnMempool,
  height int64,
  options ...CListMempoolOption,
) *CListMempool {
```

```
  mempool := &CListMempool{
    config:        config,
    proxyAppConn:  proxyAppConn,
    txs:           clist.New(),
    height:        height,
    rechecking:    0,
    recheckCursor: nil,
    recheckEnd:    nil,
    logger:        log.NewNopLogger(),
    metrics:       NopMetrics(),
  }
  if config.CacheSize > 0 {
    mempool.cache = newMapTxCache(config.CacheSize)
  } else {
    mempool.cache = nopTxCache{}
  }
  proxyAppConn.SetResponseCallback(mempool.globalCb)
  for _, option := range options {
    option(mempool)
  }
  return mempool
}
```

First the mempool is created using the referencing the CListMempool type

1.4 Communication

There are three forms of communication (e.g., requests, reponses, connections) that can happen in Tendermint Core

- Internode: communication between a node and other peers. This kind of communication happens over TCP or HTTP.
- Intranode: communication within the node itself (i.e., between reactors or other components). These are typically function or method calls, or occasionally happen through an event bus.
- Client: communication between a client (like a wallet or a browser) and a node on the network.

2 Tendermint Switch Structure

The Tendermint switch handles the incoming traffic from the P2P network. It is the interface between the node and the rest of the network and routes messages.

- Responsible for routing connections between peers
- Notably: only handles TCP connections; RPC/HTTP is separate
- Is a dependency for every reactor; all reactors expose a function 'setSwitch'
- Holds onto channels (channels on the TCP connection–NOT Go channels) and uses them to route
- Is a global object, with a global namespace for messages
- Similar functionality to libp2p

2.1 NewSwitch

The NewSwitch function can be found in p2p/switch.go with the code as follows:

```
// NewSwitch creates a new Switch with the given config.
func NewSwitch(
  cfg *config.P2PConfig,
  transport Transport,
  options ...SwitchOption,
) *Switch {
  sw := &Switch{
    config:               cfg,
    reactors:             make(map[string]Reactor),
    chDescs:              make([]*conn.ChannelDescriptor, 0)
        ,
    reactorsByCh:         make(map[byte]Reactor),
    peers:                NewPeerSet(),
    dialing:              cmap.NewCMap(),
    reconnecting:         cmap.NewCMap(),
    metrics:              NopMetrics(),
    transport:            transport,
    filterTimeout:        defaultFilterTimeout,
    persistentPeersAddrs: make([]*NetAddress, 0),
    unconditionalPeerIDs: make(map[ID]struct{}),
  }

  // Ensure we have a completely undeterministic PRNG.
  sw.rng = rand.NewRand()

  sw.BaseService = *service.NewBaseService(nil, "P2P Switch
      ", sw)

  for _, option := range options {
    option(sw)
  }

  return sw
}
```

2.2 createSwitch

Core Tendermint reactors are added to the switch in node/node.go

```
func createSwitch(config *cfg.Config,
  transport p2p.Transport,
  p2pMetrics *p2p.Metrics,
  peerFilters []p2p.PeerFilterFunc,
  mempoolReactor *mempl.Reactor,
  bcReactor p2p.Reactor,
  consensusReactor *consensus.Reactor,
  evidenceReactor *evidence.Reactor,
  nodeInfo p2p.NodeInfo,
  nodeKey *p2p.NodeKey,
  p2pLogger log.Logger) *p2p.Switch {

  sw := p2p.NewSwitch(
    config.P2P,
    transport,
    p2p.WithMetrics(p2pMetrics),
    p2p.SwitchPeerFilters(peerFilters...),
  )
  sw.SetLogger(p2pLogger)
  sw.AddReactor("MEMPOOL", mempoolReactor)
  sw.AddReactor("BLOCKCHAIN", bcReactor)
  sw.AddReactor("CONSENSUS", consensusReactor)
  sw.AddReactor("EVIDENCE", evidenceReactor)

  sw.SetNodeInfo(nodeInfo)
  sw.SetNodeKey(nodeKey)

  p2pLogger.Info("P2P Node ID", "ID", nodeKey.ID(), "file",
      config.NodeKeyFile())
  return sw
}
```

2.3 addReactors

2.4 CustomReactors

CustomReactors allows you to add custom reactors (name -¿ p2p.Reactor) to the node's Switch.

WARNING: using any name from the below list of the existing reactors will result in replacing it with the custom one. - MEMPOOL - BLOCKCHAIN - CONSENSUS - EVIDENCE - PEX

```
func CustomReactors(reactors map[string]p2p.Reactor) Option
    {
  return func(n *Node) {
    for name, reactor := range reactors {
      if existingReactor := n.sw.Reactor(name);
          existingReactor != nil {
        n.sw.Logger.Info("Replacing existing reactor with a
            custom one",
          "name", name, "existing", existingReactor, "custom
              ", reactor)
        n.sw.RemoveReactor(name, existingReactor)
      }
      n.sw.AddReactor(name, reactor)
    }
  }
}
```

The addReactor function may be found in the p2p directory inside the switch.go file as follows:

```
// AddReactor adds the given reactor to the switch.
// NOTE: Not goroutine safe.
func (sw *Switch) AddReactor(name string, reactor Reactor)
    Reactor {
  for _, chDesc := range reactor.GetChannels() {
    chID := chDesc.ID
    // No two reactors can share the same channel.
    if sw.reactorsByCh[chID] != nil {
      panic(
        fmt.Sprintf(
          "Channel %X has multiple reactors %v & %v",
          chID,
          sw.reactorsByCh[chID],
          reactor
        )
      )
    }
    sw.chDescs = append(sw.chDescs, chDesc)
    sw.reactorsByCh[chID] = reactor
  }
  sw.reactors[name] = reactor
  reactor.SetSwitch(sw)
  return reactor
}
```

3 Tendermint Reactor Structure

The base Reactor type may be found in the p2p directory inside the base_reactor.go file.

"Reactor is responsible for handling incoming messages on one or more channels. Switch calls GetChannels when reactor is added to it. When a new peer joins our node, InitPeer and AddPeer are called. RemovePeer is called when the peer is stopped. Receive is called when a message is received on a channel associated with this reactor. Peer Send or Peer TrySend should be used to send the message to a peer."

```
type Reactor interface {
  service.Service // Start, Stop

  SetSwitch(*Switch)

  GetChannels() []*conn.ChannelDescriptor

  InitPeer(peer Peer) Peer

  AddPeer(peer Peer)

  RemovePeer(peer Peer, reason interface{})

  Receive(chID byte, peer Peer, msgBytes []byte)
}

//--------------------------------------
```

- SetSwitch(*Switch)

  SetSwitch allows setting a switch. Every reactor holds a pointer to the global switch and the switch holds a pointer to every reactor.

- GetChannels() []*conn.ChannelDescriptor

  GetChannels returns the list of MConnection.ChannelDescriptor. Make sure that each ID is unique across all the reactors added to the switch.

- InitPeer(peer Peer) Peer

  InitPeer is called by the switch before the peer is started. Use it to initialize data for the peer (e.g. peer state).

  NOTE: The switch won't call AddPeer nor RemovePeer if it fails to start the peer. Do not store any data associated with the peer in the reactor itself unless you don't want to have a state, which is never cleaned up.

- AddPeer(peer Peer)

  AddPeer is called by the switch after the peer is added and successfully started. Use it to start goroutines communicating with the peer.

- RemovePeer(peer Peer, reason interface)

  RemovePeer is called by the switch when the peer is stopped (due to error or other reason).

- Receive(chID byte, peer Peer, msgBytes [ ]byte)

  Receive is called by the switch when msgBytes is received from the peer.

  NOTE: reactor can not keep msgBytes around after Receive completes without copying.

  CONTRACT: msgBytes are not nil.

3.1 Mock Reactor

The mock reactor file is located in the p2p/mock/reactor.go file

```
package mock

import (
  "github.com/tendermint/tendermint/libs/log"
  "github.com/tendermint/tendermint/p2p"
  "github.com/tendermint/tendermint/p2p/conn"
)

type Reactor struct {
  p2p.BaseReactor
}

func NewReactor() *Reactor {
  r := &Reactor{}
  r.BaseReactor = *p2p.NewBaseReactor("Mock-PEX", r)
  r.SetLogger(log.TestingLogger())
  return r
}

func (r *Reactor) GetChannels() []*conn.ChannelDescriptor
              { return []*conn.ChannelDescriptor{} }
func (r *Reactor) AddPeer(peer p2p.Peer)
                                {}
func (r *Reactor) RemovePeer(peer p2p.Peer, reason interface
    {})      {}
func (r *Reactor) Receive(chID byte, peer p2p.Peer, msgBytes
     []byte) {}
```

3.2 Existing Tendermint Reactors

Tendermint currently contains five reactors in the