Tendermint Network Metering Reactor

Charles Dusek

April 25, 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 network metering.

Tendermint is a Byzantine Fault Tolerant consensus engine that is used primarily to build blockchain applications. Tendermint uses a gossip protocol to disseminate messages to all connected nodes as well as maintain a membership list. The consensus can 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 Application Structure

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

Most implementations of Cobra use separate the main.go and the root.go files. Tendermint combines these two files into the main.go and places it in the ./cmd/main.go

```
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)
  }
}
```

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 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.

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,
  )
}
```

1.2 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