Cardano.BM - benchmarking and logging

Alexander Diemand Andreas Triantafyllos

November 2018

Abstract

This is a framework that combines logging, benchmarking and monitoring. Complex evaluations of STM or monadic actions can be observed from outside while reading operating system counters before and after, and calculating their differences, thus relating resource usage to such actions. Through interactive configuration, the runtime behaviour of logging or the measurement of resource usage can be altered. Further reduction in logging can be achieved by redirecting log messages to an aggregation function which will output the running statistics with less frequency than the original message.

Contents

Chapter 1

Cardano BM

1.1 Overview

In figure 1.1 we display the relationships among modules in *Cardano.BM*. The arrows indicate import of a module. The arrows with a triangle at one end would signify "inheritance" in object-oriented programming, but we use it to show that one module replaces the other in the namespace, thus refines its interface.

1.2 Introduction

1.2.1 Logging with Trace

1.2.2 Setup procedure

Hierarchy of Traces

1.2.3 Measuring *Observable*s

1.2.4 Information reduction in Aggregation

1.2.5 Output selection

1.2.6 Monitoring

1.3 Examples

1.3.1 Observing evaluation of a STM action

1.3.2 Observing evaluation of a monad action

1.3.3 Simple example showing plain logging

```
{-# LANGUAGE OverloadedStrings #-}
module Main
  (main)
  where
import Control.Concurrent (threadDelay)
import Cardano.BM.Configuration.Static (defaultConfigStdout)
```

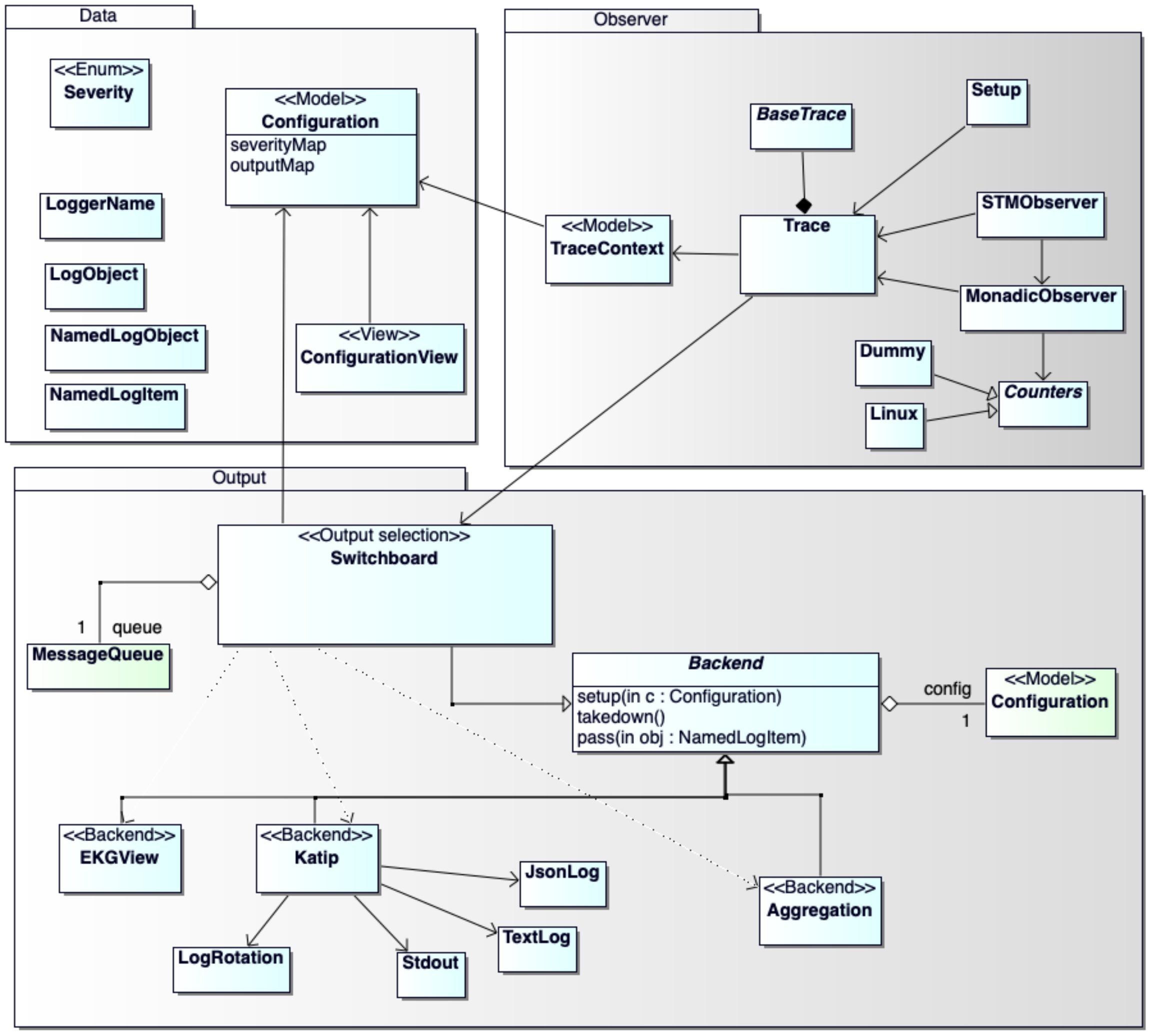

Figure 1.1: Overview of module relationships

```haskell
import Cardano.BM.Setup (setupTrace)
import Cardano.BM.Trace (logDebug, logError, logInfo, logNotice,
        logWarning)
main :: IO ()
main = do
  c ← defaultConfigStdout
  tr ← setupTrace (Right c) "simple"
  logDebug tr "this is a debug message"
  logInfo   tr "this is an information."
  logNotice tr "this is a notice!"
  logWarning tr "this is a warning!"
  logError  tr "this is an error!"
  threadDelay 80000
```

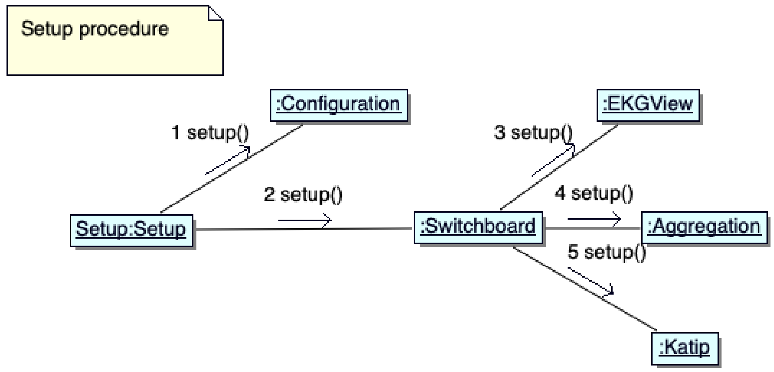

Figure 1.2: Setup procedure

```
    return ()
```

1.3.4 Complex example showing logging, aggregation of log items, and observing *IO* actions

Module header and import directives

```
{-# LANGUAGE OverloadedStrings #-}
module Main
  (main)
  where
import qualified Control.Concurrent.Async as Async
import Control.Concurrent (threadDelay)
import qualified Cardano.BM.Configuration.Model as CM
import Cardano.BM.Data.Aggregated (Measurable (..))
import Cardano.BM.Data.BackendKind
import Cardano.BM.Data.LogItem
import Cardano.BM.Data.Observable
import Cardano.BM.Data.Output
import Cardano.BM.Data.Severity
import Cardano.BM.Data.SubTrace
import Cardano.BM.Observer.Monadic (bracketObserveIO)
import Cardano.BM.Setup
import Cardano.BM.Trace
import System.Random
```

Define configuration

The output can be viewed in EKG on `http://localhost:12789`.

```
config :: IO CM.Configuration
config = do
  c ← CM.empty
  CM.setMinSeverity c Debug
  CM.setSetupBackends c [KatipBK, AggregationBK, EKGViewBK]
   {-per default each messages is sent to the logs, if not otherwise defined (see below: 'CM.setBackend') -}
  CM.setDefaultBackends c [KatipBK]
  CM.setSetupScribes c [ScribeDefinition {
      scName = "stdout"
      , scKind = StdoutSK
      , scRotation = Nothing
      }
    , ScribeDefinition {
      scName = "out.json"
      , scKind = FileJsonSK
      , scRotation = Nothing
      }
    , ScribeDefinition {
      scName = "out.txt"
      , scKind = FileTextSK
      , scRotation = Nothing
      }
    ]
  -- per default each messages is sent to the logs, if not otherwise defined (see below:
  CM.setDefaultScribes c ["StdoutSK::stdout", "FileJsonSK::out.json"]
  CM.setScribes c "complex.random" (Just ["StdoutSK::stdout", "FileTextSK::out.txt"])
  CM.setScribes c "complex.random.aggregated" (Just ["StdoutSK::stdout"])
   {-define a subtrace whose behaviour is to copy all log items, and pass them up with a name added to th
  CM.setSubTrace c "complex.random" (Just $ TeeTrace "copy")
  -- not every aggregated value needs to be displayed
  CM.setSubTrace c "#ekgview"
    (Just $ FilterTrace [Drop (StartsWith "#ekgview.#aggregation.complex.random"),
      Unhide (Named "count"),
      Unhide (Named "mean")
      ])
   {-define a subtrace whose behaviour is to copy all log items, and pass them up with a name added to th
  CM.setSubTrace c "complex.observeIO" (Just $ ObservableTrace [GhcRtsStats, MemoryStats])
  -- forward the random number to aggregation:
  CM.setBackends c "complex.random" (Just [AggregationBK, KatipBK])
  CM.setBackends c "complex.random.copy" (Just [AggregationBK])
  -- forward the observed values to aggregation:
  CM.setBackends c "complex.observeIO" (Just [KatipBK])
  -- forward the aggregated output to the EKG view:
  CM.setBackends c "#aggregation.complex.random" (Just [EKGViewBK])
  CM.setBackends c "#aggregation.complex.random.copy" (Just [EKGViewBK])
  CM.setBackends c "#aggregation.complex.observeIO" (Just [EKGViewBK])
  -- start EKG on http://localhost:12789
```

```haskell
    CM.setEKGport c 12789
    return c
```

Thread that outputs a random number to a Trace

```haskell
randomThr :: Trace IO → IO (Async.Async ())
randomThr trace = do
    logInfo trace "starting random generator"
    trace' ← subTrace "random" trace
    proc ← Async.async (loop trace')
    return proc
  where
    loop tr = do
        threadDelay 800000
        num ← randomRIO (42 - 42, 42 + 42) :: IO Double
        traceNamedObject tr (LP (LogValue "rr" (PureD num)))
        loop tr
```

Thread that observes an *IO* action

```haskell
observeIO :: Trace IO → IO (Async.Async ())
observeIO trace = do
    logInfo trace "starting observer"
    proc ← Async.async (loop trace)
    return proc
  where
    loop tr = do
        threadDelay 1000000-- 1 second
        bracketObserveIO trace "observeIO" $ do
            num ← randomRIO (100000, 2000000) :: IO Int
            _ ← return $ reverse $ reverse $ 42 : [1 .. num]
            threadDelay 50000-- .05 second
            pure ()
        loop tr
```

Main entry point

```haskell
main :: IO ()
main = do
    -- create configuration
    c ← config
    -- create initial top-level Trace
    tr ← setupTrace (Right c) "complex"
    logNotice tr "starting program; hit CTRL-C to terminate"
```

```
logInfo tr "watch its progress on http://localhost:12789"
 {-start thread sending unbounded sequence of random numbers to a trace which aggregates them into a
procRandom ← randomThr tr
-- start thread endlessly reversing lists of random length
procObsvIO ← observeIO tr
-- wait for observer thread to finish, ignoring any exception
_ ← Async.waitCatch procObsvIO
-- wait for random thread to finish, ignoring any exception
_ ← Async.waitCatch procRandom
return ()
```

1.4 Code listings

1.4.1 Cardano.BM.Observer.STM

```
stmWithLog :: STM.STM (t, [LogObject]) → STM.STM (t, [LogObject])
stmWithLog action = action
```

Observe *STM* action in a named context

With given name, create a SubTrace according to Configuration and run the passed *STM* action on it.

```
bracketObserveIO :: Trace IO → Text → STM.STM t → IO t
bracketObserveIO logTrace0 name action = do
    logTrace ← subTrace name logTrace0
    let subtrace = typeofTrace logTrace
    bracketObserveIO' subtrace logTrace action
  where
    bracketObserveIO' :: SubTrace → Trace IO → STM.STM t → IO t
    bracketObserveIO' NoTrace _ act =
      STM.atomically act
    bracketObserveIO' subtrace logTrace act = do
      mCountersid ← observeOpen subtrace logTrace
      -- run action; if an exception is caught will be logged and rethrown.
      t ← (STM.atomically act) `catch` (λ(e :: SomeException) → (logError logTrace (pack (show e)) ≫ throwM e)
      case mCountersid of
        Left openException →
          -- since observeOpen faced an exception there is no reason to call observeClose
          -- however the result of the action is returned
          logNotice logTrace ("ObserveOpen: " <> pack (show openException))
        Right countersid → do
          res ← observeClose subtrace logTrace countersid [ ]
          case res of
            Left ex → logNotice logTrace ("ObserveClose: " <> pack (show ex))
```

```
      _ → pure ()
  pure t
```

Observe *STM* action in a named context and output captured log items

The *STM* action might output messages, which after ”success” will be forwarded to the logging trace. Otherwise, this function behaves the same as Observe *STM* action in a named context.

```
bracketObserveLogIO :: Trace IO → Text → STM.STM (t, [LogObject]) → IO t
bracketObserveLogIO logTrace0 name action = do
    logTrace ← subTrace name logTrace0
    let subtrace = typeofTrace logTrace
    bracketObserveLogIO' subtrace logTrace action
  where
    bracketObserveLogIO' :: SubTrace → Trace IO → STM.STM (t, [LogObject]) → IO t
    bracketObserveLogIO' NoTrace _ act = do
      (t, _) ← STM.atomically $ stmWithLog act
      pure t
    bracketObserveLogIO' subtrace logTrace act = do
      mCountersid ← observeOpen subtrace logTrace
      -- run action, return result and log items; if an exception is
      -- caught will be logged and rethrown.
      (t, as) ← (STM.atomically $ stmWithLog act) `catch`
          (λ(e :: SomeException) → (logError logTrace (pack (show e)) ≫ throwM e))
      case mCountersid of
        Left openException →
          -- since observeOpen faced an exception there is no reason to call observeClose
          -- however the result of the action is returned
          logNotice logTrace ("ObserveOpen: " <> pack (show openException))
        Right countersid → do
          res ← observeClose subtrace logTrace countersid as
          case res of
            Left ex → logNotice logTrace ("ObserveClose: " <> pack (show ex))
            _ → pure ()
      pure t
```

1.4.2 Cardano.BM.Observer.Monadic

Monadic.bracketObserverIO

Observes an *IO* action and adds a name to the logger name of the passed in Trace. An empty *Text* leaves the logger name untouched.

Microbenchmarking steps:

1. Create a *trace* which will have been configured to observe things besides logging.

```haskell
import qualified Cardano.BM.Configuration.Model as CM
 ◦ ◦ ◦
c ← config
trace@(ctx, _) ← setupTrace (Right c) "demo-playground"
  where
    config :: IO CM.Configuration
    config = do
      c ← CM.empty
      CM.setMinSeverity c Debug
      CM.setSetupBackends c [KatipBK, AggregationBK]
      CM.setDefaultBackends c [KatipBK, AggregationBK]
      CM.setSetupScribes c [ScribeDefinition {
        scName = "stdout"
        , scKind = StdoutSK
        , scRotation = Nothing
        }
        ]
      CM.setDefaultScribes c ["StdoutSK::stdout"]
      return c
```

2. *c* is the Configuration of *trace*. In order to enable the collection and processing of measurements (min, max, mean, std-dev) *AggregationBK* is needed.

```haskell
CM.setDefaultBackends c [KatipBK, AggregationBK]
```

in a configuration file (YAML) means

```yaml
defaultBackends:
  - KatipBK
  - AggregationBK
```

3. Set the measurements that you want to take by changing the configuration of the *trace* using *setSubTrace*, in order to declare the namespace where we want to enable the particular measurements and the list with the kind of measurements.

```haskell
CM.setSubTrace
    (configuration ctx)
    "demo-playground.submit-tx"
    (Just $ ObservableTrace observablesSet)
  where
    observablesSet = [MonotonicClock, MemoryStats]
```

4. Find an action to measure. e.g.:

```haskell
runProtocolWithPipe x hdl proto `catch` (λProtocolStopped → return ())
```

and use bracketObserveIO. e.g.:

```haskell
bracketObserveIO trace "submit-tx" $
  runProtocolWithPipe x hdl proto `catch` (λProtocolStopped → return ())
```

```
bracketObserveIO :: Trace IO → Text → IO t → IO t
bracketObserveIO logTrace0 name action = do
    logTrace ← subTrace name logTrace0
    bracketObserveIO' (typeofTrace logTrace) logTrace action
  where
    bracketObserveIO' :: SubTrace → Trace IO → IO t → IO t
    bracketObserveIO' NoTrace _ act = act
    bracketObserveIO' subtrace logTrace act = do
      mCountersid ← observeOpen subtrace logTrace
      -- run action; if an exception is caught will be logged and rethrown.
      t ← act `catch` (λ(e :: SomeException) → (logError logTrace (pack (show e)) ≫ throwM e))
      case mCountersid of
        Left openException →
          -- since observeOpen faced an exception there is no reason to call observeClose
          -- however the result of the action is returned
          logNotice logTrace ("ObserveOpen: " <> pack (show openException))
        Right countersid → do
          res ← observeClose subtrace logTrace countersid [ ]
          case res of
            Left ex → logNotice logTrace ("ObserveClose: " <> pack (show ex))
            _ → pure ()
      pure t
```

Monadic.bracketObserverM

Observes a *MonadIO m* ⇒ *m* action and adds a name to the logger name of the passed in Trace. An empty *Text* leaves the logger name untouched.

```
bracketObserveM :: (MonadCatch m, MonadIO m) ⇒ Trace IO → Text → m t → m t
bracketObserveM logTrace0 name action = do
    logTrace ← liftIO $ subTrace name logTrace0
    bracketObserveM' (typeofTrace logTrace) logTrace action
  where
    bracketObserveM' :: (MonadCatch m, MonadIO m) ⇒ SubTrace → Trace IO → m t → m t
    bracketObserveM' NoTrace _ act = act
    bracketObserveM' subtrace logTrace act = do
      mCountersid ← liftIO $ observeOpen subtrace logTrace
      -- run action; if an exception is caught will be logged and rethrown.
      t ← act `catch`
        (λ(e :: SomeException) → (liftIO (logError logTrace (pack (show e)) ≫ throwM e)))
      case mCountersid of
        Left openException →
          -- since observeOpen faced an exception there is no reason to call observeClose
          -- however the result of the action is returned
          liftIO $ logNotice logTrace ("ObserveOpen: " <> pack (show openException))
```

```
        Right countersid → do
          res ← liftIO $ observeClose subtrace logTrace countersid [ ]
          case res of
            Left ex → liftIO (logNotice logTrace ("ObserveClose: " <> pack (show ex)))
            _ → pure ()
      pure t
```

observerOpen

```
observeOpen :: SubTrace → Trace IO → IO (Either SomeException CounterState)
observeOpen subtrace logTrace = (do
  identifier ← newUnique
  -- take measurement
  counters ← readCounters subtrace
  let state = CounterState identifier counters
  -- send opening message to Trace
  traceNamedObject logTrace $ ObserveOpen state
  return (Right state)) ‘catch‘ (return ∘ Left)
```

observeClose

```
observeClose :: SubTrace → Trace IO → CounterState → [LogObject] → IO (Either SomeException ())
observeClose subtrace logTrace initState logObjects = (do
  let identifier = csIdentifier initState
      initialCounters = csCounters initState
  -- take measurement
  counters ← readCounters subtrace
  -- send closing message to Trace
  traceNamedObject logTrace $ ObserveClose (CounterState identifier counters)
  -- send diff message to Trace
  traceNamedObject logTrace $
    ObserveDiff (CounterState identifier (diffCounters initialCounters counters))
  -- trace the messages gathered from inside the action
  forM_ logObjects $ traceNamedObject logTrace
  return (Right ())) ‘catch‘ (return ∘ Left)
```

1.4.3 BaseTrace

Contravariant

A covariant is a functor: F A → F B
A contravariant is a functor: F B → F A

Op a b implements the inverse to ’arrow’ ”*getOp* :: $b \to a$”, which when applied to a BaseTrace of type ”*Op* (*m* ()) *s*”, yields ”$s \to m$ ()”. In our case, *Op* accepts an action in a monad *m* with input type LogNamed LogObject (see ’Trace’).

$$\textbf{newtype}\ \mathrm{BaseTrace}\ m\ s = \mathrm{BaseTrace}\ \{runTrace :: Op\ (m\ ())\ s\}$$

contramap

A covariant functor defines the function "$fmap :: (a \to b) \to f\ a \to f\ b$". In case of a contravariant functor, it is the dual function "$contramap :: (a \to b) \to f\ b \to f\ a$" which is defined.

In the following instance, $runTrace$ extracts type "$Op\ (m\ ())\ s$" to which contramap applies f, thus "$f\ s \to m\ ()$". The constructor BaseTrace restores "$Op\ (m\ ())\ (f\ s)$".

```
instance Contravariant (BaseTrace m) where
  contramap f = BaseTrace ∘ contramap f ∘ runTrace
```

traceWith

Accepts a Trace and some payload s. First it gets the contravariant from the Trace as type "$Op\ (m\ ())\ s$" and, after "$getOp::b \to a$" which translates to "$s \to m\ ()$", calls the action on the LogNamed LogObject.

```
traceWith :: BaseTrace m s → s → m ()
traceWith = getOp ∘ runTrace
```

natTrace

Natural transformation from monad m to monad n.

```
natTrace :: (forall x ∘ m x → n x) → BaseTrace m s → BaseTrace n s
natTrace nat (BaseTrace (Op tr)) = BaseTrace $ Op $ nat ∘ tr
```

noTrace

A Trace that discards all inputs.

```
noTrace :: Applicative m ⇒ BaseTrace m a
noTrace = BaseTrace $ Op $ const (pure ())
```

1.4.4 Cardano.BM.Trace

Utilities

Natural transformation from monad m to monad n.

```
natTrace :: (forall x ∘ m x → n x) → Trace m → Trace n
natTrace nat (ctx, trace) = (ctx, BaseTrace.natTrace nat trace)
```

Access type of Trace.

```
typeofTrace :: Trace m → SubTrace
typeofTrace (ctx, _) = tracetype ctx
```

Update type of Trace.

```
updateTracetype :: SubTrace → Trace m → Trace m
updateTracetype subtr (ctx, tr) = (ctx {tracetype = subtr}, tr)
```

Enter new named context

The context name is created and checked that its size is below a limit (currently 80 chars). The minimum severity that a log message must be labelled with is looked up in the configuration and recalculated.

```haskell
appendName :: MonadIO m ⇒ LoggerName → Trace m → m (Trace m)
appendName name (ctx, trace) = do
  let prevLoggerName = loggerName ctx
      prevMinSeverity = minSeverity ctx
      newLoggerName   = appendWithDot prevLoggerName name
  globMinSeverity ← liftIO $ Config.minSeverity (configuration ctx)
  namedSeverity ← liftIO $ Config.inspectSeverity (configuration ctx) newLoggerName
  case namedSeverity of
    Nothing → return (ctx {loggerName = newLoggerName}, trace)
    Just sev → return (ctx {loggerName = newLoggerName
      , minSeverity = max (max sev prevMinSeverity) globMinSeverity}
      , trace)
appendWithDot :: LoggerName → LoggerName → LoggerName
appendWithDot "" newName = T.take 80 newName
appendWithDot xs "" = xs
appendWithDot xs newName = T.take 80 $ xs <> "." <> newName
```

Contramap a trace and produce the naming context

```haskell
named :: BaseTrace.BaseTrace m (LogNamed i) → LoggerName → BaseTrace.BaseTrace m i
named trace name = contramap (LogNamed name) trace
```

Trace a LogObject through

```haskell
traceNamedObject
  :: MonadIO m
  ⇒ Trace m
  → LogObject
  → m ()
traceNamedObject trace@(ctx, logTrace) lo = do
  let lname = loggerName ctx
  doOutput ← case (typeofTrace trace) of
    FilterTrace filters → do
      return $ evalFilters filters lname lo
    TeeTrace secName → do
      -- create a newly named copy of the LogObject
      BaseTrace.traceWith (named logTrace (lname <> "." <> secName)) lo
      return True
    _ → return True
  if doOutput
```

```
    then BaseTrace.traceWith (named logTrace lname) lo
    else return ()
```

Evaluation of *FilterTrace*

```
evalFilters :: [NameOperator] → LoggerName → LogObject → Bool
evalFilters nos nm lo =
    any (evalFilter nm lo) nos
  where
    evalFilter :: LoggerName → LogObject → NameOperator → Bool
    evalFilter name item (Drop sel) = ¬ (matchName name sel) ∧ ¬ (matchItem item sel)
    evalFilter name item (Unhide sel) = matchName name sel ∨ matchItem item sel
    matchName :: LoggerName → NameSelector → Bool
    matchName name (Exact name') = name ≡ name'
    matchName name (StartsWith prefix) = T.isPrefixOf prefix name
    matchName name (EndsWith postfix) = T.isSuffixOf postfix name
    matchName name (Contains name') = T.isInfixOf name' name
    matchName _ _ = False
    matchItem :: LogObject → NameSelector → Bool
    matchItem (LP (LogValue name _)) (Named name') = name ≡ name'
    matchItem _ _ = False
```

Concrete Trace on stdout

This function returns a trace with an action of type "(LogNamed LogObject) → IO ()" which will output a text message as text and all others as JSON encoded representation to the console.

TODO remove *locallock*

```
locallock :: MVar ()
locallock = unsafePerformIO $ newMVar ()
```

```
stdoutTrace :: TraceNamed IO
stdoutTrace = BaseTrace.BaseTrace $ Op $ λlognamed →
    withMVar locallock $ \_ →
      case lnItem lognamed of
        LP (LogMessage logItem) →
          output (lnName lognamed) $ liPayload logItem
        obj →
          output (lnName lognamed) $ toStrict (encodeToLazyText obj)
  where
    output nm msg = TIO.putStrLn $ nm <> " :: " <> msg
```

Concrete Trace into a *TVar*

```
traceInTVar :: STM.TVar [a] → BaseTrace.BaseTrace STM.STM a
traceInTVar tvar = BaseTrace.BaseTrace $ Op $ λa → STM.modifyTVar tvar ((:) a)
traceInTVarIO :: STM.TVar [LogObject] → TraceNamed IO
traceInTVarIO tvar = BaseTrace.BaseTrace $ Op $ λln →
  STM.atomically $ STM.modifyTVar tvar ((:) (lnItem ln))
traceNamedInTVarIO :: STM.TVar [LogNamed LogObject] → TraceNamed IO
traceNamedInTVarIO tvar = BaseTrace.BaseTrace $ Op $ λln →
  STM.atomically $ STM.modifyTVar tvar ((:) ln)
```

Check a log item's severity against the Trace's minimum severity

do we need three different minSeverity defined?

We do a lookup of the global minSeverity in the configuration. And, a lookup of the minSeverity for the current named context. These values might have changed in the meanwhile.
A third filter is the minSeverity defined in the current context.

```
traceConditionally
  :: MonadIO m
  ⇒ Trace m → LogObject
  → m ()
traceConditionally logTrace@(ctx, _) msg@(LP (LogMessage item)) = do
  globminsev ← liftIO $ Config.minSeverity (configuration ctx)
  globnamesev ← liftIO $ Config.inspectSeverity (configuration ctx) (loggerName ctx)
  let minsev = max (minSeverity ctx) $ max globminsev (fromMaybe Debug globnamesev)
    flag = (liSeverity item) ⩾ minsev
  when flag $ traceNamedObject logTrace msg
traceConditionally logTrace logObject =
  traceNamedObject logTrace logObject
```

Enter message into a trace

The function traceNamedItem creates a LogObject and threads this through the action defined in the Trace.

```
traceNamedItem
  :: (MonadIO m)
  ⇒ Trace m
  → LogSelection
  → Severity
  → T.Text
  → m ()
traceNamedItem trace p s m =
  let logmsg = LP $ LogMessage $ LogItem {liSelection = p
    , liSeverity = s
```

```haskell
        , liPayload = m
        }
  in
  traceConditionally trace $ logmsg
```

Logging functions

```haskell
logDebug, logInfo, logNotice, logWarning, logError, logCritical, logAlert, logEmergency
  :: (MonadIO m) => Trace m -> T.Text -> m ()
logDebug     logTrace = traceNamedItem logTrace Both Debug
logInfo      logTrace = traceNamedItem logTrace Both Info
logNotice    logTrace = traceNamedItem logTrace Both Notice
logWarning   logTrace = traceNamedItem logTrace Both Warning
logError     logTrace = traceNamedItem logTrace Both Error
logCritical  logTrace = traceNamedItem logTrace Both Critical
logAlert     logTrace = traceNamedItem logTrace Both Alert
logEmergency logTrace = traceNamedItem logTrace Both Emergency
logDebugS, logInfoS, logNoticeS, logWarningS, logErrorS, logCriticalS, logAlertS, logEmergencyS
  :: (MonadIO m) => Trace m -> T.Text -> m ()
logDebugS     logTrace = traceNamedItem logTrace Private Debug
logInfoS      logTrace = traceNamedItem logTrace Private Info
logNoticeS    logTrace = traceNamedItem logTrace Private Notice
logWarningS   logTrace = traceNamedItem logTrace Private Warning
logErrorS     logTrace = traceNamedItem logTrace Private Error
logCriticalS  logTrace = traceNamedItem logTrace Private Critical
logAlertS     logTrace = traceNamedItem logTrace Private Alert
logEmergencyS logTrace = traceNamedItem logTrace Private Emergency
logDebugP, logInfoP, logNoticeP, logWarningP, logErrorP, logCriticalP, logAlertP, logEmergencyP
  :: (MonadIO m) => Trace m -> T.Text -> m ()
logDebugP     logTrace = traceNamedItem logTrace Public Debug
logInfoP      logTrace = traceNamedItem logTrace Public Info
logNoticeP    logTrace = traceNamedItem logTrace Public Notice
logWarningP   logTrace = traceNamedItem logTrace Public Warning
logErrorP     logTrace = traceNamedItem logTrace Public Error
logCriticalP  logTrace = traceNamedItem logTrace Public Critical
logAlertP     logTrace = traceNamedItem logTrace Public Alert
logEmergencyP logTrace = traceNamedItem logTrace Public Emergency
logDebugUnsafeP, logInfoUnsafeP, logNoticeUnsafeP, logWarningUnsafeP, logErrorUnsafeP,
  logCriticalUnsafeP, logAlertUnsafeP, logEmergencyUnsafeP
  :: (MonadIO m) => Trace m -> T.Text -> m ()
logDebugUnsafeP     logTrace = traceNamedItem logTrace PublicUnsafe Debug
logInfoUnsafeP      logTrace = traceNamedItem logTrace PublicUnsafe Info
logNoticeUnsafeP    logTrace = traceNamedItem logTrace PublicUnsafe Notice
logWarningUnsafeP   logTrace = traceNamedItem logTrace PublicUnsafe Warning
logErrorUnsafeP     logTrace = traceNamedItem logTrace PublicUnsafe Error
logCriticalUnsafeP  logTrace = traceNamedItem logTrace PublicUnsafe Critical
```

```
logAlertUnsafeP     logTrace = traceNamedItem logTrace PublicUnsafe Alert
logEmergencyUnsafeP logTrace = traceNamedItem logTrace PublicUnsafe Emergency
```

subTrace

Transforms the input Trace according to the Configuration using the logger name of the current Trace appended with the new name. If the empty *Text* is passed, then the logger name remains untouched.

```
subTrace :: MonadIO m ⇒ T.Text → Trace m → m (Trace m)
subTrace name tr@(ctx, _) = do
  let newName = appendWithDot (loggerName ctx) name
  subtrace0 ← liftIO $ Config.findSubTrace (configuration ctx) newName
  let subtrace = case subtrace0 of Nothing → Neutral; Just str → str
  case subtrace of
    Neutral      → do
                   tr′ ← appendName name tr
                   return $ updateTracetype subtrace tr′
    UntimedTrace → do
                   tr′ ← appendName name tr
                   return $ updateTracetype subtrace tr′
    TeeTrace _   → do
                   tr′ ← appendName name tr
                   return $ updateTracetype subtrace tr′
    FilterTrace _ → do
                   tr′ ← appendName name tr
                   return $ updateTracetype subtrace tr′
    NoTrace      → return $ updateTracetype subtrace (ctx, BaseTrace.BaseTrace $ Op $ \_ → pure ())
    DropOpening  → return $ updateTracetype subtrace (ctx, BaseTrace.BaseTrace $ Op $ λlognamed → do
      case lnItem lognamed of
        ObserveOpen _ → return ()
        obj → traceNamedObject tr obj)
    ObservableTrace _ → do
                   tr′ ← appendName name tr
                   return $ updateTracetype subtrace tr′
```

1.4.5 Cardano.BM.Setup

setupTrace

Setup a new Trace (Trace) with either a given Configuration (Configuration.Model) or a *FilePath* to a configuration file.

```
setupTrace :: MonadIO m ⇒ Either FilePath Config.Configuration → Text → m (Trace m)
setupTrace (Left cfgFile) name = do
  c ← liftIO $ Config.setup cfgFile
  setupTrace_ c name
setupTrace (Right c) name = setupTrace_ c name
```

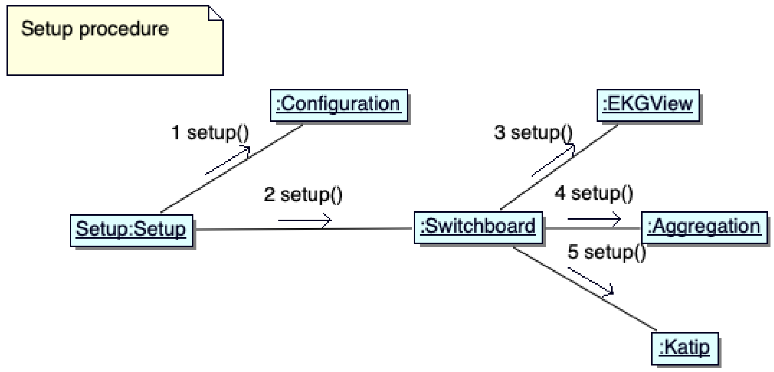

Figure 1.3: Setup procedure

$$
\begin{aligned}
&\mathit{setupTrace\_} :: \mathit{MonadIO}\ m \Rightarrow \mathit{Config}.\mathrm{Configuration} \rightarrow \mathit{Text} \rightarrow m\ (\mathrm{Trace}\ m)\\
&\mathit{setupTrace\_}\ c\ \mathit{name} = \mathbf{do}\\
&\quad \mathit{sb} \leftarrow \mathit{liftIO}\ \$\ \mathrm{Switchboard}.\mathit{realize}\ c\\
&\quad \mathit{sev} \leftarrow \mathit{liftIO}\ \$\ \mathit{Config}.\mathrm{minSeverity}\ c\\
&\quad \mathit{ctx} \leftarrow \mathit{liftIO}\ \$\ \mathrm{newContext}\ \mathit{name}\ c\ \mathit{sev}\ \mathit{sb}\\
&\quad \mathbf{let}\ \mathit{logTrace} = \mathrm{natTrace}\ \mathit{liftIO}\ (\mathit{ctx}, \mathrm{Switchboard}.\mathrm{mainTrace}\ \mathit{sb})\\
&\quad \mathit{logTrace}' \leftarrow \mathrm{subTrace}\ \texttt{""}\ \mathit{logTrace}\\
&\quad \mathit{return}\ \mathit{logTrace}'
\end{aligned}
$$

withTrace

$$
\begin{aligned}
&\mathrm{withTrace} :: \mathit{MonadIO}\ m \Rightarrow \mathit{Config}.\mathrm{Configuration} \rightarrow \mathit{Text} \rightarrow (\mathrm{Trace}\ m \rightarrow m\ t) \rightarrow m\ t\\
&\mathrm{withTrace}\ \mathit{cfg}\ \mathit{name}\ \mathit{action} = \mathbf{do}\\
&\quad \mathit{logTrace} \leftarrow \mathrm{setupTrace}\ (\mathit{Right}\ \mathit{cfg})\ \mathit{name}\\
&\quad \mathit{action}\ \mathit{logTrace}
\end{aligned}
$$

newContext

$$
\begin{aligned}
&\mathrm{newContext} :: \mathrm{LoggerName}\\
&\quad\ \rightarrow \mathit{Config}.\mathrm{Configuration}\\
&\quad\ \rightarrow \mathrm{Severity}\\
&\quad\ \rightarrow \mathrm{Switchboard}.\mathrm{Switchboard}\\
&\quad\ \rightarrow \mathit{IO}\ \mathrm{TraceContext}\\
&\mathrm{newContext}\ \mathit{name}\ \mathit{cfg}\ \mathit{sev}\ \mathit{sb} = \mathbf{do}\\
&\quad \mathit{return}\ \$\ \mathrm{TraceContext}\ \{\\
&\qquad \mathrm{loggerName} = \mathit{name}\\
&\qquad , \mathrm{configuration} = \mathit{cfg}\\
&\qquad , \mathrm{minSeverity} = \mathit{sev}\\
&\qquad , \mathrm{tracetype} = \mathrm{Neutral}
\end{aligned}
$$

```
    ,shutdown = unrealize sb
    }
```

1.4.6 Cardano.BM.Counters

Here the platform is chosen on which we compile this program.

Currently, we mainly support *Linux* with its 'proc' filesystem.

```
{-# LANGUAGE CPP #-}
#if defined (linux_HOST_OS)
#define LINUX
#endif
module Cardano.BM.Counters
  (
    Platform.readCounters
  , diffTimeObserved
  , getMonoClock
  ) where
#ifdef LINUX
import qualified Cardano.BM.Counters.Linux as Platform
#else
import qualified Cardano.BM.Counters.Dummy as Platform
#endif
import Cardano.BM.Counters.Common (getMonoClock)
import Cardano.BM.Data.Aggregated (Measurable (..))
import Cardano.BM.Data.Counter
import Data.Time.Units (Microsecond)
```

Calculate difference between clocks

```
diffTimeObserved :: CounterState → CounterState → Microsecond
diffTimeObserved (CounterState id0 startCounters) (CounterState id1 endCounters) =
    let
      startTime = getMonotonicTime startCounters
      endTime = getMonotonicTime endCounters
    in
    if (id0 ≡ id1)
      then endTime − startTime
      else error "these clocks are not from the same experiment"
  where
    getMonotonicTime counters = case (filter isMonotonicClockCounter counters) of
      [(Counter MonotonicClockTime _ (Microseconds micros))] → fromInteger micros
      _ → error "A time measurement is missing!"
    isMonotonicClockCounter :: Counter → Bool
    isMonotonicClockCounter = (MonotonicClockTime ≡) ◦ cType
```

1.4.7 Cardano.BM.Counters.Common

Common functions that serve *readCounters* on all platforms.

```haskell
nominalTimeToMicroseconds :: Word64 → Microsecond
nominalTimeToMicroseconds = fromMicroseconds ∘ toInteger ∘ (`div` 1000)
```

Read monotonic clock

```haskell
getMonoClock :: IO [Counter]
getMonoClock = do
  t ← getMonotonicTimeNSec
  return [Counter MonotonicClockTime "monoclock" $ Microseconds (toInteger $ nominalTimeToMicrosecond
```

Read GHC RTS statistics

Read counters from GHC's *RTS* (runtime system). The values returned are as per the last GC (garbage collection) run.

```haskell
readRTSStats :: IO [Counter]
readRTSStats = do
    iscollected ← GhcStats.getRTSStatsEnabled
    if iscollected
      then ghcstats
      else return [ ]
  where
    ghcstats :: IO [Counter]
    ghcstats = do
      -- need to run GC?
      rts ← GhcStats.getRTSStats
      let getrts = ghcval rts
      return [getrts (toInteger ∘ GhcStats.allocated_bytes, "bytesAllocated")
        ,getrts (toInteger ∘ GhcStats.max_live_bytes, "liveBytes")
        ,getrts (toInteger ∘ GhcStats.max_large_objects_bytes, "largeBytes")
        ,getrts (toInteger ∘ GhcStats.max_compact_bytes, "compactBytes")
        ,getrts (toInteger ∘ GhcStats.max_slop_bytes, "slopBytes")
        ,getrts (toInteger ∘ GhcStats.max_mem_in_use_bytes, "usedMemBytes")
        ,getrts (toInteger ∘ GhcStats.gc_cpu_ns, "gcCpuNs")
        ,getrts (toInteger ∘ GhcStats.gc_elapsed_ns, "gcElapsedNs")
        ,getrts (toInteger ∘ GhcStats.cpu_ns, "cpuNs")
        ,getrts (toInteger ∘ GhcStats.elapsed_ns, "elapsedNs")
        ,getrts (toInteger ∘ GhcStats.gcs, "gcNum")
        ,getrts (toInteger ∘ GhcStats.major_gcs, "gcMajorNum")
        ]
    ghcval :: GhcStats.RTSStats → ((GhcStats.RTSStats → Integer), Text) → Counter
    ghcval s (f, n) = Counter RTSStats n $ PureI (f s)
```

1.4.8 Cardano.BM.Counters.Dummy

This is a dummy definition of *readCounters* on platforms that do not support the 'proc' filesystem from which we would read the counters.

The only supported measurements are monotonic clock time and RTS statistics for now.

```haskell
readCounters :: SubTrace → IO [Counter]
readCounters NoTrace          = return [ ]
readCounters Neutral          = return [ ]
readCounters (TeeTrace _)     = return [ ]
readCounters (FilterTrace _)  = return [ ]
readCounters UntimedTrace     = return [ ]
readCounters DropOpening      = return [ ]
readCounters (ObservableTrace tts) = foldrM (λ(sel, fun) a →
      if any (≡ sel) tts
      then (fun ≫= λxs → return $ a ++ xs)
      else return a) [ ] selectors
   where
      selectors = [(MonotonicClock, getMonoClock)
        -- , (MemoryStats, readProcStatM)
        -- , (ProcessStats, readProcStats)
            -- , (IOStats, readProcIO)
          , (GhcRtsStats, readRTSStats)
                           ]
```

1.4.9 Cardano.BM.Counters.Linux

we have to expand the *readMemStats* function
to read full data from *proc*

```haskell
readCounters :: SubTrace → IO [Counter]
readCounters NoTrace          = return [ ]
readCounters Neutral          = return [ ]
readCounters (TeeTrace _)     = return [ ]
readCounters (FilterTrace _)  = return [ ]
readCounters UntimedTrace     = return [ ]
readCounters DropOpening      = return [ ]
readCounters (ObservableTrace tts) = foldrM (λ(sel, fun) a →
      if any (≡ sel) tts
      then (fun ≫= λxs → return $ a ++ xs)
      else return a) [ ] selectors
   where
      selectors = [(MonotonicClock, getMonoClock)
        , (MemoryStats, readProcStatM)
        , (ProcessStats, readProcStats)
        , (IOStats, readProcIO)
        ]
```

```
pathProc :: FilePath
pathProc = "/proc/"
pathProcStat :: ProcessID → FilePath
pathProcStat pid = pathProc </> (show pid) </> "stat"
pathProcStatM :: ProcessID → FilePath
pathProcStatM pid = pathProc </> (show pid) </> "statm"
pathProcIO :: ProcessID → FilePath
pathProcIO pid = pathProc </> (show pid) </> "io"
```

Reading from a file in /proc/<pid >

```
readProcList :: FilePath → IO [Integer]
readProcList fp = do
  cs ← readFile fp
  return $ map (λs → maybe 0 id $ (readMaybe s :: Maybe Integer)) (words cs)
```

readProcStatM - /proc/<pid >/statm

```
/proc/[pid]/statm
       Provides information about memory usage, measured in pages.  The columns are:
              size       (1) total program size
                            (same as VmSize in /proc/[pid]/status)
              resident   (2) resident set size
                            (same as VmRSS in /proc/[pid]/status)
              shared     (3) number of resident shared pages (i.e., backed by a file)
                            (same as RssFile+RssShmem in /proc/[pid]/status)
              text       (4) text (code)
              lib        (5) library (unused since Linux 2.6; always 0)
              data       (6) data + stack
              dt         (7) dirty pages (unused since Linux 2.6; always 0)
```

```
readProcStatM :: IO [Counter]
readProcStatM = do
    pid ← getProcessID
    ps0 ← readProcList (pathProcStatM pid)
    let ps = zip colnames ps0
      psUseful = filter (("unused" ≢) ∘ fst) ps
    return $ map (λ(n,i) → Counter MemoryCounter n (PureI i)) psUseful
  where
    colnames :: [Text]
    colnames = ["size","resident","shared","text","unused","data","unused"]
```

readProcStats - //proc//<pid >//stat

```
/proc/[pid]/stat
       Status  information about the process.  This is used by ps(1).  It is defined in the kernel source file
       fs/proc/array.c.

       The fields, in order, with their proper scanf(3) format specifiers, are listed below.  Whether  or  not
       certain   of   these   fields   display   valid  information  is  governed  by  a  ptrace  access  mode
       PTRACE_MODE_READ_FSCREDS | PTRACE_MODE_NOAUDIT check (refer to ptrace(2)).  If the check denies access,
```

then the field value is displayed as 0. The affected fields are indicated with the marking [PT].

(1) pid %d
    The process ID.

(2) comm %s
    The filename of the executable, in parentheses. This is visible whether or not the executable is swapped out.

(3) state %c
    One of the following characters, indicating process state:

    R Running

    S Sleeping in an interruptible wait

    D Waiting in uninterruptible disk sleep

    Z Zombie

    T Stopped (on a signal) or (before Linux 2.6.33) trace stopped

    t Tracing stop (Linux 2.6.33 onward)

    W Paging (only before Linux 2.6.0)

    X Dead (from Linux 2.6.0 onward)

    x Dead (Linux 2.6.33 to 3.13 only)

    K Wakekill (Linux 2.6.33 to 3.13 only)

    W Waking (Linux 2.6.33 to 3.13 only)

    P Parked (Linux 3.9 to 3.13 only)

(4) ppid %d
    The PID of the parent of this process.

(5) pgrp %d
    The process group ID of the process.

(6) session %d
    The session ID of the process.

(7) tty_nr %d
    The controlling terminal of the process. (The minor device number is contained in the combination of bits 31 to 20 and 7 to 0; the major device number is in bits 15 to 8.)

(8) tpgid %d
    The ID of the foreground process group of the controlling terminal of the process.

(9) flags %u
    The kernel flags word of the process. For bit meanings, see the PF_* defines in the Linux kernel source file include/linux/sched.h. Details depend on the kernel version.

    The format for this field was %lu before Linux 2.6.

(10) minflt %lu
    The number of minor faults the process has made which have not required loading a memory page from disk.

(11) cminflt %lu
    The number of minor faults that the process's waited-for children have made.

(12) majflt %lu
    The number of major faults the process has made which have required loading a memory page from disk.

```
(13) cmajflt  %lu
            The number of major faults that the process's waited-for children have made.

(14) utime  %lu
            Amount of time that this process has been scheduled in user mode,  measured  in  clock  ticks
            (divide by sysconf(_SC_CLK_TCK)).  This includes guest time, guest_time (time spent running a
            virtual CPU, see below), so that applications that are not aware of the guest time  field  do
            not lose that time from their calculations.

(15) stime  %lu
            Amount  of  time that this process has been scheduled in kernel mode, measured in clock ticks
            (divide by sysconf(_SC_CLK_TCK)).

(16) cutime  %ld
            Amount of time that this process's waited-for children have been scheduled in user mode, mea-
            sured  in  clock ticks (divide by sysconf(_SC_CLK_TCK)).  (See also times(2).)  This includes
            guest time, cguest_time (time spent running a virtual CPU, see below).

(17) cstime  %ld
            Amount of time that this process's waited-for children have been scheduled  in  kernel  mode,
            measured in clock ticks (divide by sysconf(_SC_CLK_TCK)).

(18) priority  %ld
            (Explanation  for  Linux  2.6)  For  processes  running a real-time scheduling policy (policy
            below; see sched_setscheduler(2)), this is the negated scheduling priority, minus  one;  that
            is,  a  number  in  the range -2 to -100, corresponding to real-time priorities 1 to 99.  For
            processes running under a non-real-time scheduling policy, this is the raw nice  value  (set-
            priority(2))  as  represented in the kernel.  The kernel stores nice values as numbers in the
            range 0 (high) to 39 (low), corresponding to the user-visible nice range of -20 to 19.

(19) nice  %ld
            The nice value (see setpriority(2)), a value in the range 19 (low priority) to -20 (high pri-
            ority).

(20) num_threads  %ld
            Number of threads in this process (since Linux 2.6).  Before kernel 2.6, this field was  hard
            coded to 0 as a placeholder for an earlier removed field.

(21) itrealvalue  %ld
            The  time in jiffies before the next SIGALRM is sent to the process due to an interval timer.
            Since kernel 2.6.17, this field is no longer maintained, and is hard coded as 0.

(22) starttime  %llu
            The time the process started after system boot.  In kernels before Linux 2.6, this value  was
            expressed  in  jiffies.   Since  Linux  2.6, the value is expressed in clock ticks (divide by
            sysconf(_SC_CLK_TCK)).

            The format for this field was %lu before Linux 2.6.

(23) vsize  %lu
            Virtual memory size in bytes.

(24) rss  %ld
            Resident Set Size: number of pages the process has in real memory.  This is  just  the  pages
            which  count  toward  text, data, or stack space.  This does not include pages which have not
            been demand-loaded in, or which are swapped out.

(25) rsslim  %lu
            Current soft limit in bytes on the rss of the process; see the description of  RLIMIT_RSS  in
            getrlimit(2).

(26) startcode  %lu  [PT]
            The address above which program text can run.

(27) endcode  %lu  [PT]
            The address below which program text can run.
```

```
(28) startstack  %lu  [PT]
          The address of the start (i.e., bottom) of the stack.

(29) kstkesp  %lu  [PT]
          The current value of ESP (stack pointer), as found in the kernel stack page for the process.

(30) kstkeip  %lu  [PT]
          The current EIP (instruction pointer).

(31) signal  %lu
          The  bitmap of pending signals, displayed as a decimal number.  Obsolete, because it does not
          provide information on real-time signals; use /proc/[pid]/status instead.

(32) blocked  %lu
          The bitmap of blocked signals, displayed as a decimal number.  Obsolete, because it does  not
          provide information on real-time signals; use /proc/[pid]/status instead.

(33) sigignore  %lu
          The  bitmap of ignored signals, displayed as a decimal number.  Obsolete, because it does not
          provide information on real-time signals; use /proc/[pid]/status instead.

(34) sigcatch  %lu
          The bitmap of caught signals, displayed as a decimal number.  Obsolete, because it  does  not
          provide information on real-time signals; use /proc/[pid]/status instead.

(35) wchan  %lu  [PT]
          This  is  the  "channel" in which the process is waiting.  It is the address of a location in
          the kernel where the process is sleeping.  The corresponding symbolic name can  be  found  in
          /proc/[pid]/wchan.

(36) nswap  %lu
          Number of pages swapped (not maintained).

(37) cnswap  %lu
          Cumulative nswap for child processes (not maintained).

(38) exit_signal  %d  (since Linux 2.1.22)
          Signal to be sent to parent when we die.

(39) processor  %d  (since Linux 2.2.8)
          CPU number last executed on.

(40) rt_priority  %u  (since Linux 2.5.19)
          Real-time  scheduling priority, a number in the range 1 to 99 for processes scheduled under a
          real-time policy, or 0, for non-real-time processes (see sched_setscheduler(2)).

(41) policy  %u  (since Linux 2.5.19)
          Scheduling policy  (see  sched_setscheduler(2)).   Decode  using  the  SCHED_*  constants  in
          linux/sched.h.

          The format for this field was %lu before Linux 2.6.22.

(42) delayacct_blkio_ticks  %llu  (since Linux 2.6.18)
          Aggregated block I/O delays, measured in clock ticks (centiseconds).

(43) guest_time  %lu  (since Linux 2.6.24)
          Guest  time  of  the process (time spent running a virtual CPU for a guest operating system),
          measured in clock ticks (divide by sysconf(_SC_CLK_TCK)).

(44) cguest_time  %ld  (since Linux 2.6.24)
          Guest  time  of  the  process's  children,  measured  in  clock  ticks  (divide  by
          sysconf(_SC_CLK_TCK)).

(45) start_data  %lu  (since Linux 3.3)  [PT]
          Address above which program initialized and uninitialized (BSS) data are placed.

(46) end_data  %lu  (since Linux 3.3)  [PT]
          Address below which program initialized and uninitialized (BSS) data are placed.
```

```
(47) start_brk  %lu  (since Linux 3.3)  [PT]
          Address above which program heap can be expanded with brk(2).

(48) arg_start  %lu  (since Linux 3.5)  [PT]
          Address above which program command-line arguments (argv) are placed.

(49) arg_end  %lu  (since Linux 3.5)  [PT]
          Address below program command-line arguments (argv) are placed.

(50) env_start  %lu  (since Linux 3.5)  [PT]
          Address above which program environment is placed.

(51) env_end  %lu  (since Linux 3.5)  [PT]
          Address below which program environment is placed.

(52) exit_code  %d  (since Linux 3.5)  [PT]
          The thread's exit status in the form reported by waitpid(2).
```

```
readProcStats :: IO [Counter]
readProcStats = do
    pid ← getProcessID
    ps0 ← readProcList (pathProcStat pid)
    let ps = zip colnames ps0
      psUseful = filter (("unused" ≢) ∘ fst) ps
    return $ map (λ(n,i) → Counter StatInfo n (PureI i)) psUseful
  where
    colnames :: [Text]
    colnames = ["pid","unused","unused","ppid","pgrp","session","ttynr","tpgid","flags","minfl
      ,"cminflt","majflt","cmajflt","utime","stime","cutime","cstime","priority","nice","num
      ,"itrealvalue","starttime","vsize","rss","rsslim","startcode","endcode","startstack","
      ,"signal","blocked","sigignore","sigcatch","wchan","nswap","cnswap","exitsignal","proc
      ,"policy","blkio","guesttime","cguesttime","startdata","enddata","startbrk","argstart"
      ,"envend","exitcode"
      ]
```

readProcIO - //proc//<pid >//io

```
/proc/[pid]/io (since kernel 2.6.20)
       This file contains I/O statistics for the process, for example:

              # cat /proc/3828/io
              rchar: 323934931
              wchar: 323929600
              syscr: 632687
              syscw: 632675
              read_bytes: 0
              write_bytes: 323932160
              cancelled_write_bytes: 0

       The fields are as follows:

       rchar: characters read
              The number of bytes which this task has caused to be read from storage.  This is simply the  sum
              of bytes which this process passed to read(2) and similar system calls.  It includes things such
              as terminal I/O and is unaffected by whether or not actual physical disk I/O was  required  (the
              read might have been satisfied from pagecache).
```

```
wchar: characters written
       The  number  of bytes which this task has caused, or shall cause to be written to disk.  Similar
       caveats apply here as with rchar.

syscr: read syscalls
       Attempt to count the number of read I/O operations-that is, system calls  such  as  read(2)  and
       pread(2).

syscw: write syscalls
       Attempt  to  count the number of write I/O operations-that is, system calls such as write(2) and
       pwrite(2).

read_bytes: bytes read
       Attempt to count the number of bytes which this process really did cause to be fetched from  the
       storage layer.  This is accurate for block-backed filesystems.

write_bytes: bytes written
       Attempt to count the number of bytes which this process caused to be sent to the storage layer.

cancelled_write_bytes:
       The  big  inaccuracy  here  is truncate.  If a process writes 1MB to a file and then deletes the
       file, it will in fact perform no writeout.  But it will have been accounted as having caused 1MB
       of  write.   In other words: this field represents the number of bytes which this process caused
       to not happen, by truncating pagecache.  A task can cause "negative"  I/O  too.   If  this  task
       truncates  some  dirty  pagecache,  some  I/O  which another task has been accounted for (in its
       write\_bytes) will not be happening.

Note: In the current implementation, things are a bit racy  on  32-bit  systems:  if  process  A  reads
process  B's  /proc/[pid]/io  while process B is updating one of these 64-bit counters, process A could
see an intermediate result.

Permission to access this file is governed by a ptrace access mode PTRACE\_MODE\_READ\_FSCREDS check;  see
ptrace(2).
```

```haskell
readProcIO :: IO [Counter]
readProcIO = do
    pid ← getProcessID
    ps0 ← readProcList (pathProcIO pid)
    let ps = zip3 colnames ps0 units
    return $ map (λ(n,i,u) → Counter IOCounter n (u i)) ps
  where
    colnames :: [Text]
    colnames = ["rchar","wchar","syscr","syscw","rbytes","wbytes","cxwbytes"]
    units = [Bytes,Bytes,PureI,PureI,Bytes,Bytes,Bytes]
```

1.4.10 Cardano.BM.Data.Aggregated

Measurable

A Measurable may consist of different types of values.

```haskell
data Measurable = Microseconds Integer
   | Seconds Integer
   | Bytes Integer
   | PureI Integer
   | PureD Double
  deriving (Eq, Ord, Generic, ToJSON)
```

Measurable can be transformed to an integral value.

```haskell
getInteger :: Measurable -> Integer
getInteger (Microseconds a) = a
getInteger (Seconds a) = a
getInteger (Bytes a)   = a
getInteger (PureI a)   = a
getInteger (PureD a)   = round a
```

Measurable can be transformed to a rational value.

```haskell
getDouble :: Measurable -> Double
getDouble (Microseconds a) = fromInteger a
getDouble (Seconds a) = fromInteger a
getDouble (Bytes a)   = fromInteger a
getDouble (PureI a)   = fromInteger a
getDouble (PureD a)   = a
```

It is a numerical value, thus supports functions to operate on numbers.

```haskell
instance Num Measurable where
  (+) (Microseconds a) (Microseconds b) = Microseconds (a + b)
  (+) (Seconds a) (Seconds b) = Seconds (a + b)
  (+) (Bytes a)   (Bytes b)   = Bytes   (a + b)
  (+) (PureI a)   (PureI b)   = PureI   (a + b)
  (+) (PureD a)   (PureD b)   = PureD   (a + b)
  (+) _ _                     = error "Trying to add values with different units"
  (*) (Microseconds a) (Microseconds b) = Microseconds (a * b)
  (*) (Seconds a) (Seconds b) = Seconds (a * b)
  (*) (Bytes a)   (Bytes b)   = Bytes   (a * b)
  (*) (PureI a)   (PureI b)   = PureI   (a * b)
  (*) (PureD a)   (PureD b)   = PureD   (a * b)
  (*) _ _                     = error "Trying to multiply values with different units"
  abs (Microseconds a) = Microseconds (abs a)
  abs (Seconds a) = Seconds (abs a)
  abs (Bytes a)   = Bytes   (abs a)
  abs (PureI a)   = PureI   (abs a)
  abs (PureD a)   = PureD   (abs a)
  signum (Microseconds a) = Microseconds (signum a)
  signum (Seconds a) = Seconds (signum a)
  signum (Bytes a)   = Bytes   (signum a)
  signum (PureI a)   = PureI   (signum a)
  signum (PureD a)   = PureD   (signum a)
  negate (Microseconds a) = Microseconds (negate a)
  negate (Seconds a) = Seconds (negate a)
  negate (Bytes a)   = Bytes   (negate a)
  negate (PureI a)   = PureI   (negate a)
  negate (PureD a)   = PureD   (negate a)
```

```
    fromInteger = PureI
```

Pretty printing of Measurable.

```
instance Show Measurable where
    show = showSI
showUnits :: Measurable → String
showUnits (Microseconds _) = " s"
showUnits (Seconds _) = " s"
showUnits (Bytes _)   = " B"
showUnits (PureI _)   = ""
showUnits (PureD _)   = ""
-- show in S.I. units
showSI :: Measurable → String
showSI (Microseconds a) = show (fromFloatDigits ((fromInteger a) / (1000000 :: Float))) ++
    showUnits (Seconds a)
showSI v@(Seconds a) = show a ++ showUnits v
showSI v@(Bytes a)   = show a ++ showUnits v
showSI v@(PureI a)   = show a ++ showUnits v
showSI v@(PureD a)   = show a ++ showUnits v
```

Stats

```
data Stats = Stats {
    flast  :: Measurable,
    fmin   :: Measurable,
    fmax   :: Measurable,
    fcount :: Integer,
    fsum_A :: Double,
    fsum_B :: Double
    } deriving (Eq, Generic, ToJSON, Show)

meanOfStats :: Stats → Double
meanOfStats s = fsum_A s

stdevOfStats :: Stats → Double
stdevOfStats s =
    if fcount s < 2
    then 0
    else sqrt $ (fsum_B s) / (fromInteger $ (fcount s) − 1)
```

instance *Semigroup* Stats disabled for the moment, because not needed.

We use a parallel algorithm to update the estimation of mean and variance from two sample statistics. (see https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Parallel_algorit

```haskell
instance Semigroup Stats where
  (<>) a b = let counta = fcount a
    countb = fcount b
    newcount = counta + countb
    delta = fsum_A b - fsum_A a
    in
    Stats {flast = flast b -- right associative
      ,fmin   = min (fmin a) (fmin b)
      ,fmax   = max (fmax a) (fmax b)
      ,fcount = newcount
      ,fsum_A = fsum_A a + (delta / fromInteger newcount)
      ,fsum_B = fsum_B a + fsum_B b + (delta * delta) * (fromInteger (counta * countb) / fromInteger newcount)
      }
```

```haskell
stats2Text :: Stats -> Text
stats2Text s@(Stats slast smin smax scount _ _) =
  pack $
    "{ last = " ++ show slast ++
    ", min = " ++ show smin ++
    ", max = " ++ show smax ++
    ", mean = " ++ show (meanOfStats s) ++ showUnits slast ++
    ", std-dev = " ++ show (stdevOfStats s) ++
    ", count = " ++ show scount ++
    " }"
```

Exponentially Weighted Moving Average (EWMA)

```haskell
data EWMA = EmptyEWMA {alpha :: Double}
  | EWMA {alpha :: Double
    ,avg :: Measurable
    } deriving (Show, Eq, Generic, ToJSON)
```

Aggregated

```haskell
data Aggregated = AggregatedStats Stats
  | AggregatedEWMA EWMA
  deriving (Eq, Generic, ToJSON)
```

instance *Semigroup* Aggregated disabled for the moment, because not needed.

```haskell
instance Semigroup Aggregated where
  (<>) (AggregatedStats a) (AggregatedStats b) =
    AggregatedStats (a <> b)
  (<>) _ _ = error "Cannot combine different objects"
```

```
singletonStats :: Measurable → Aggregated
singletonStats a =
  let stats = Stats {flast = a
    ,fmin       = a
    ,fmax       = a
    ,fcount = 1
    ,fsum_A = getDouble a
    ,fsum_B = 0
    }
  in
  AggregatedStats stats

instance Show Aggregated where
  show (AggregatedStats astats) =
    "{ stats = " ++ show astats ++ " }"
  show (AggregatedEWMA a) = show a
```

1.4.11 Cardano.BM.Data.Backend

Accepts a NamedLogItem

Instances of this type class accept a NamedLogItem and deal with it.

```
class IsEffectuator t where
  effectuate :: t → NamedLogItem → IO ()
  effectuatefrom :: forall s ∘ (IsEffectuator s) ⇒ t → NamedLogItem → s → IO ()
  default effectuatefrom :: forall s ∘ (IsEffectuator s) ⇒ t → NamedLogItem → s → IO ()
  effectuatefrom t nli _ = effectuate t nli
```

Declaration of a Backend

A backend is life-cycle managed, thus can be *realize*d and *unrealize*d.

```
class (IsEffectuator t) ⇒ IsBackend t where
  typeof     :: t → BackendKind
  realize    :: Configuration → IO t
  realizefrom :: forall s ∘ (IsEffectuator s) ⇒ Trace IO → s → IO t
  default realizefrom :: forall s ∘ (IsEffectuator s) ⇒ Trace IO → s → IO t
  realizefrom (ctx, _) _ = realize (configuration ctx)
  unrealize :: t → IO ()
```

Backend

This data structure for a backend defines its behaviour as an IsEffectuator when processing an incoming message, and as an IsBackend for unrealizing the backend.

```
data Backend = MkBackend
  {bEffectuate :: NamedLogItem → IO ()
```

```
  , bUnrealize :: IO ()
  }
```

1.4.12 Cardano.BM.Data.Configuration

Data structure to help parsing configuration files.

Representation

```
type Port = Int
data Representation = Representation
  {minSeverity     :: Severity
  , rotation       :: RotationParameters
  , setupScribes   :: [ScribeDefinition]
  , defaultScribes :: [(ScribeKind, Text)]
  , setupBackends  :: [BackendKind]
  , defaultBackends :: [BackendKind]
  , hasEKG         :: Maybe Port
  , hasGUI         :: Maybe Port
  , options        :: HM.HashMap Text Object
  }
  deriving (Generic, Show, ToJSON, FromJSON)
```

parseRepresentation

```
parseRepresentation :: FilePath → IO Representation
parseRepresentation fp = do
  repr :: Representation ← decodeFileThrow fp
  return $ implicit_fill_representation repr
```

after parsing the configuration representation we implicitly correct it.

```
implicit_fill_representation :: Representation → Representation
implicit_fill_representation =
    remove_ekgview_if_not_defined ∘
    filter_duplicates_from_backends ∘
    filter_duplicates_from_scribes ∘
    union_setup_and_usage_backends ∘
    add_ekgview_if_port_defined ∘
    add_katip_if_any_scribes
  where
    filter_duplicates_from_backends r =
      r {setupBackends = mkUniq $ setupBackends r}
    filter_duplicates_from_scribes r =
      r {setupScribes = mkUniq $ setupScribes r}
    union_setup_and_usage_backends r =
```

```
      r {setupBackends = setupBackends r <> defaultBackends r}
    remove_ekgview_if_not_defined r =
      case hasEKG r of
      Nothing → r {defaultBackends = filter (λbk → bk ≢ EKGViewBK) (defaultBackends r)
        ,setupBackends = filter (λbk → bk ≢ EKGViewBK) (setupBackends r)
        }
      Just _ → r
    add_ekgview_if_port_defined r =
      case hasEKG r of
      Nothing → r
      Just _ → r {setupBackends = setupBackends r <> [EKGViewBK]}
    add_katip_if_any_scribes r =
      if (any ¬ [null $ setupScribes r, null $ defaultScribes r])
      then r {setupBackends = setupBackends r <> [KatipBK]}
      else r
    mkUniq :: Ord a ⇒ [a] → [a]
    mkUniq = Set.toList ∘ Set.fromList
```

1.4.13 Cardano.BM.Data.Counter

Counter

```
data Counter = Counter
        {cType  :: CounterType
        ,cName  :: Text
        ,cValue :: Measurable
        }
        deriving (Eq, Show, Generic, ToJSON)
data CounterType = MonotonicClockTime
   | MemoryCounter
   | StatInfo
   | IOCounter
   | CpuCounter
   | RTSStats
     deriving (Eq, Show, Generic, ToJSON)
instance ToJSON Microsecond where
  toJSON = toJSON ∘ toMicroseconds
  toEncoding = toEncoding ∘ toMicroseconds
```

Names of counters

```
nameCounter :: Counter → Text
nameCounter (Counter MonotonicClockTime _ _) = "Time-interval"
nameCounter (Counter MemoryCounter _ _) = "Mem"
nameCounter (Counter StatInfo           _ _) = "Stat"
```

```haskell
nameCounter (Counter IOCounter     _ _) = "IO"
nameCounter (Counter CpuCounter    _ _) = "Cpu"
nameCounter (Counter RTSStats      _ _) = "RTS"
```

CounterState

```haskell
data CounterState = CounterState {
  csIdentifier :: Unique
  , csCounters :: [Counter]
  }
  deriving (Generic, ToJSON)
instance ToJSON Unique where
  toJSON = toJSON ∘ hashUnique
  toEncoding = toEncoding ∘ hashUnique
instance Show CounterState where
  show cs = (show ∘ hashUnique) (csIdentifier cs)
     <> "  =>  " <> (show $ csCounters cs)
```

Difference between counters

```haskell
diffCounters :: [Counter] → [Counter] → [Counter]
diffCounters openings closings =
    getCountersDiff openings closings
  where
    getCountersDiff :: [Counter]
                → [Counter]
                → [Counter]
    getCountersDiff as bs =
      let
        getName counter = nameCounter counter <> cName counter

        asNames = map getName as
        aPairs = zip asNames as

        bsNames = map getName bs
        bs' = zip bsNames bs
        bPairs = HM.fromList bs'
      in
        catMaybes $ (flip map) aPairs $ λ(name, Counter _ _ startValue) →
          case HM.lookup name bPairs of
            Nothing      → Nothing
            Just counter → let endValue = cValue counter
                           in Just counter {cValue = endValue − startValue}
```

1.4.14 Cardano.BM.Data.LogItem

LoggerName

```haskell
type LoggerName = Text
```

NamedLogItem

```haskell
type NamedLogItem = LogNamed LogObject
```

LogItem

TODO liPayload :: ToObject

```haskell
data LogItem = LogItem
  {liSelection :: LogSelection
  ,liSeverity :: Severity
  ,liPayload :: Text-- TODO should become ToObject
  } deriving (Show, Generic, ToJSON)
```

```haskell
data LogSelection =
  Public  -- only to public logs.
  | PublicUnsafe-- only to public logs, not console.
  | Private-- only to private logs.
  | Both  -- to public and private logs.
  deriving (Show, Generic, ToJSON, FromJSON)
```

LogObject

```haskell
data LogPrims = LogMessage LogItem
  | LogValue Text Measurable
    deriving (Generic, Show, ToJSON)
data LogObject = LP LogPrims
  | ObserveOpen CounterState
  | ObserveDiff CounterState
  | ObserveClose CounterState
  | AggregatedMessage [(Text, Aggregated)]
  | KillPill
    deriving (Generic, Show, ToJSON)
```

LogNamed

A LogNamed contains of a context name and some log item.

```
data LogNamed item = LogNamed
  {lnName :: LoggerName
  ,lnItem :: item
  } deriving (Show)
deriving instance Generic item ⇒ Generic (LogNamed item)
deriving instance (ToJSON item, Generic item) ⇒ ToJSON (LogNamed item)
```

1.4.15 Cardano.BM.Data.Observable

ObservableInstance

```
data ObservableInstance = MonotonicClock
   | MemoryStats
   | ProcessStats
   | IOStats
   | GhcRtsStats
     deriving (Generic, Eq, Ord, Show, FromJSON, ToJSON, Read)
```

1.4.16 Cardano.BM.Data.Output

OutputKind

```
data OutputKind = TVarList (STM.TVar [LogObject])
   | TVarListNamed (STM.TVar [LogNamed LogObject])
  deriving (Eq)
```

ScribeKind

This identifies katip's scribes by type.

```
data ScribeKind = FileTextSK
   | FileJsonSK
   | StdoutSK
   | StderrSK
  deriving (Generic, Eq, Ord, Show, FromJSON, ToJSON)
```

ScribeId

A scribe is identified by ScribeKind *x Filename*

```
type ScribeId = Text-- (ScribeKind :: Filename)
```

ScribeDefinition

This identifies katip's scribes by type.

```
data ScribeDefinition = ScribeDefinition
  {scKind  :: ScribeKind
  ,scName :: Text
  ,scRotation :: Maybe RotationParameters
  }
  deriving (Generic, Eq, Ord, Show, FromJSON, ToJSON)
```

1.4.17 Cardano.BM.Data.Severity

Severity

The intended meaning of severity codes:

Debug *detailled information about values and decision flow* Info general information of events; progressing properly Notice *needs attention;something ¬ progressing properly* Warning may continue into an error condition if continued Error *unexpected set* **of** *event or condition occured* Critical error condition causing degrade of operation Alert *a subsystem is no longer operating correctly,likely requires manual* at this point, the system can never progress without additional intervention

We were informed by the *Syslog* taxonomy: https://en.wikipedia.org/wiki/Syslog#Severity_level

```
data Severity = Debug
   | Info
   | Notice
   | Warning
   | Error
   | Critical
   | Alert
   | Emergency
     deriving (Show, Eq, Ord, Generic, ToJSON, Read)
instance FromJSON Severity where
  parseJSON = withText "severity" $ λcase
    "Debug"     → pure Debug
    "Info"      → pure Info
    "Notice"    → pure Notice
    "Warning"   → pure Warning
    "Error"     → pure Error
    "Critical"  → pure Critical
    "Alert"     → pure Alert
    "Emergency" → pure Emergency
    _           → pure Info-- catch all
```

1.4.18 Cardano.BM.Data.SubTrace

SubTrace

```
data NameSelector = Exact Text | StartsWith Text | EndsWith Text | Contains Text
   | Named Text-- LP (LogValue (name value))
```

```
    deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
data NameOperator = Drop NameSelector | Unhide NameSelector
    deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
data SubTrace = Neutral
  | UntimedTrace
  | NoTrace
  | TeeTrace LoggerName
  | FilterTrace [NameOperator]
  | DropOpening
  | ObservableTrace [ObservableInstance]
    deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
```

1.4.19 Cardano.BM.Data.Trace

Trace

A Trace consists of a TraceContext and a TraceNamed in *m*.

```
type Trace m = (TraceContext, TraceNamed m)
```

TraceNamed

A TraceNamed is a specialized Contravariant of type NamedLogItem, a LogNamed with payload LogObject.

```
type TraceNamed m = BaseTrace m (NamedLogItem)
```

TraceContext

We keep the context's name and a reference to the Configuration in the TraceContext.

```
data TraceContext = TraceContext
  {loggerName :: LoggerName
  ,configuration :: Configuration
  ,tracetype     :: SubTrace
  ,minSeverity :: Severity
  ,shutdown      :: IO ()
  }
```

1.4.20 Cardano.BM.Configuration

see Cardano.BM.Configuration.Model for the implementation.

```
getOptionOrDefault :: CM.Configuration → Text → Text → IO (Text)
getOptionOrDefault cg name def = do
  opt ← CM.getOption cg name
  case opt of
    Nothing → return def
    Just o → return o
```

1.4.21 Cardano.BM.Configuration.Model

Configuration.Model

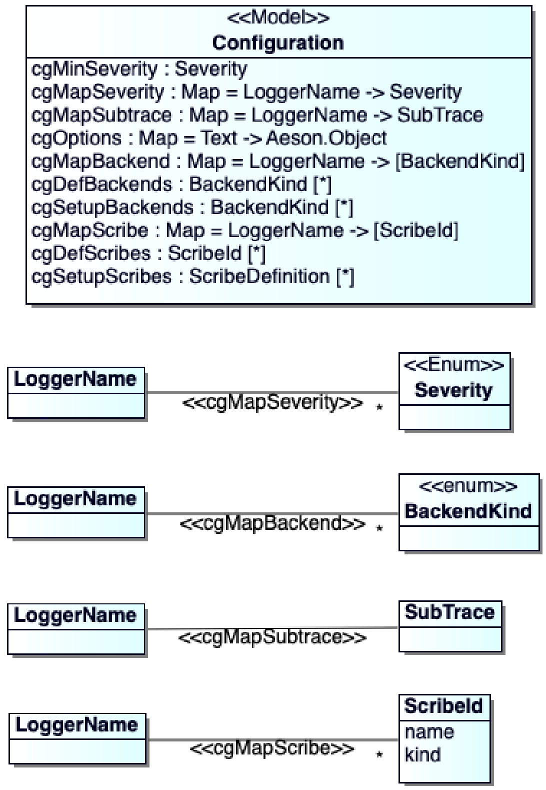

Figure 1.4: Configuration model

```
type ConfigurationMVar = MVar ConfigurationInternal
newtype Configuration = Configuration
  {getCG :: ConfigurationMVar}
-- Our internal state; see -"Configuration model"-
data ConfigurationInternal = ConfigurationInternal
  {cgMinSeverity   :: Severity
  -- minimum severity level of every object that will be output
  ,cgMapSeverity   :: HM.HashMap LoggerName Severity
  -- severity filter per loggername
  ,cgMapSubtrace   :: HM.HashMap LoggerName SubTrace
  -- type of trace per loggername
  ,cgOptions       :: HM.HashMap Text Object
  -- options needed for tracing, logging and monitoring
  ,cgMapBackend    :: HM.HashMap LoggerName [BackendKind]
  -- backends that will be used for the specific loggername
  ,cgDefBackendKs :: [BackendKind]
```

```
-- backends that will be used if a set of backends for the
-- specific loggername is not set
,cgSetupBackends :: [BackendKind]
-- backends to setup; every backend to be used must have
-- been declared here
,cgMapScribe      :: HM.HashMap LoggerName [ScribeId]
-- katip scribes that will be used for the specific loggername
,cgDefScribes     :: [ScribeId]
-- katip scribes that will be used if a set of scribes for the
-- specific loggername is not set
,cgSetupScribes   :: [ScribeDefinition]
-- katip scribes to setup; every scribe to be used must have
-- been declared here
,cgMapAggregatedKind :: HM.HashMap LoggerName AggregatedKind
-- kind of Aggregated that will be used for the specific loggername
,cgDefAggregatedKind :: AggregatedKind
-- kind of Aggregated that will be used if a set of scribes for the
-- specific loggername is not set
,cgPortEKG        :: Int
-- port for EKG server
,cgPortGUI        :: Int
-- port for changes at runtime (NOT IMPLEMENTED YET)
} deriving (Show, Eq)
```

Backends configured in the Switchboard

For a given context name return the list of backends configured, or, in case no such configuration exists, return the default backends.

```
getBackends :: Configuration → LoggerName → IO [BackendKind]
getBackends configuration name =
  withMVar (getCG configuration) $ λcg → do
    let outs = HM.lookup name (cgMapBackend cg)
    case outs of
      Nothing → do
        return (cgDefBackendKs cg)
      Just os → return os
getDefaultBackends :: Configuration → IO [BackendKind]
getDefaultBackends configuration =
  withMVar (getCG configuration) $ λcg → do
    return (cgDefBackendKs cg)
setDefaultBackends :: Configuration → [BackendKind] → IO ()
setDefaultBackends configuration bes = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgDefBackendKs = bes}
setBackends :: Configuration → LoggerName → Maybe [BackendKind] → IO ()
setBackends configuration name be = do
```

```
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $ cg {cgMapBackend = HM.alter (\_ → be) name (cgMapBackend cg)}
```

Backends to be setup by the Switchboard

Defines the list of Backends that need to be setup by the Switchboard.

```
setSetupBackends :: Configuration → [BackendKind] → IO ()
setSetupBackends configuration bes = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $ cg {cgSetupBackends = bes}
getSetupBackends :: Configuration → IO [BackendKind]
getSetupBackends configuration =
    withMVar (getCG configuration) $ λcg →
        return $ cgSetupBackends cg
```

Scribes configured in the Log backend

For a given context name return the list of scribes to output to, or, in case no such configuration exists, return the default scribes to use.

```
getScribes :: Configuration → LoggerName → IO [ScribeId]
getScribes configuration name =
    withMVar (getCG configuration) $ λcg → do
        let outs = HM.lookup name (cgMapScribe cg)
        case outs of
            Nothing → do
                return (cgDefScribes cg)
            Just os → return $ os
setScribes :: Configuration → LoggerName → Maybe [ScribeId] → IO ()
setScribes configuration name scribes = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $ cg {cgMapScribe = HM.alter (\_ → scribes) name (cgMapScribe cg)}
setDefaultScribes :: Configuration → [ScribeId] → IO ()
setDefaultScribes configuration scs = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $ cg {cgDefScribes = scs}
```

Scribes to be setup in the Log backend

Defines the list of *Scribe*s that need to be setup in the Log backend.

```
setSetupScribes :: Configuration → [ScribeDefinition] → IO ()
setSetupScribes configuration sds = do
    cg ← takeMVar (getCG configuration)
    putMVar (getCG configuration) $ cg {cgSetupScribes = sds}
```

```
getSetupScribes :: Configuration → IO [ScribeDefinition]
getSetupScribes configuration =
  withMVar (getCG configuration) $ λcg → do
    return $ cgSetupScribes cg
```

AggregatedKind to define the type of measurement

For a given context name return its *AggregatedKind* or in case no such configuration exists, return the default *AggregatedKind* to use.

```
getAggregatedKind :: Configuration → LoggerName → IO AggregatedKind
getAggregatedKind configuration name =
  withMVar (getCG configuration) $ λcg → do
    let outs = HM.lookup name (cgMapAggregatedKind cg)
    case outs of
      Nothing → do
        return (cgDefAggregatedKind cg)
      Just os → return $ os
setDefaultAggregatedKind :: Configuration → AggregatedKind → IO ()
setDefaultAggregatedKind configuration defAK = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgDefAggregatedKind = defAK}
setAggregatedKind :: Configuration → LoggerName → Maybe AggregatedKind → IO ()
setAggregatedKind configuration name ak = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgMapAggregatedKind = HM.alter (\_ → ak) name (cgMapAggregatedKi
```

Access port numbers of EKG, GUI

```
getEKGport :: Configuration → IO Int
getEKGport configuration =
  withMVar (getCG configuration) $ λcg → do
    return $ cgPortEKG cg
setEKGport :: Configuration → Int → IO ()
setEKGport configuration port = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgPortEKG = port}
getGUIport :: Configuration → IO Int
getGUIport configuration =
  withMVar (getCG configuration) $ λcg → do
    return $ cgPortGUI cg
setGUIport :: Configuration → Int → IO ()
setGUIport configuration port = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgPortGUI = port}
```

Options

```
getOption :: Configuration → Text → IO (Maybe Text)
getOption configuration name = do
  withMVar (getCG configuration) $ λcg →
    case HM.lookup name (cgOptions cg) of
      Nothing → return Nothing
      Just o → return $ Just $ pack $ show o
```

Global setting of minimum severity

```
minSeverity :: Configuration → IO Severity
minSeverity configuration = withMVar (getCG configuration) $ λcg →
  return $ cgMinSeverity cg
setMinSeverity :: Configuration → Severity → IO ()
setMinSeverity configuration sev = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgMinSeverity = sev}
```

Relation of context name to minimum severity

```
inspectSeverity :: Configuration → Text → IO (Maybe Severity)
inspectSeverity configuration name = do
  withMVar (getCG configuration) $ λcg →
    return $ HM.lookup name (cgMapSeverity cg)
setSeverity :: Configuration → Text → Maybe Severity → IO ()
setSeverity configuration name sev = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgMapSeverity = HM.alter (\_ → sev) name (cgMapSeverity cg)}
```

Relation of context name to SubTrace

A new context may contain a different type of Trace. The function appendName (Enter new named context) will look up the SubTrace for the context's name.

```
findSubTrace :: Configuration → Text → IO (Maybe SubTrace)
findSubTrace configuration name = do
  withMVar (getCG configuration) $ λcg →
    return $ HM.lookup name (cgMapSubtrace cg)
setSubTrace :: Configuration → Text → Maybe SubTrace → IO ()
setSubTrace configuration name trafo = do
  cg ← takeMVar (getCG configuration)
  putMVar (getCG configuration) $ cg {cgMapSubtrace = HM.alter (\_ → trafo) name (cgMapSubtrace cg)}
```

Parse configuration from file

Parse the configuration into an internal representation first. Then, fill in Configuration after refinement.

```
setup :: FilePath → IO Configuration
setup fp = do
    r ← R.parseRepresentation fp
    cgref ← newEmptyMVar
    let mapseverity = HM.lookup "mapSeverity" (R.options r)
        mapbackends = HM.lookup "mapBackends" (R.options r)
        mapsubtrace  = HM.lookup "mapSubtrace" (R.options r)
        mapscribes    = HM.lookup "mapScribes"  (R.options r)
        mapAggregatedKinds = HM.lookup "mapAggregatedkinds" (R.options r)
    putMVar cgref $ ConfigurationInternal
        {cgMinSeverity = R.minSeverity r
        ,cgMapSeverity = parseSeverityMap mapseverity
        ,cgMapSubtrace = parseSubtraceMap mapsubtrace
        ,cgOptions = R.options r
        ,cgMapBackend = parseBackendMap mapbackends
        ,cgDefBackendKs = R.defaultBackends r
        ,cgSetupBackends = R.setupBackends r
        ,cgMapScribe = parseScribeMap mapscribes
        ,cgDefScribes = r_defaultScribes r
        ,cgSetupScribes = R.setupScribes r
        ,cgMapAggregatedKind = parseAggregatedKindMap mapAggregatedKinds
        ,cgDefAggregatedKind = StatsAK
        ,cgPortEKG = r_hasEKG r
        ,cgPortGUI = r_hasGUI r
        }
    return $ Configuration cgref
  where
    parseSeverityMap :: Maybe (HM.HashMap Text Value) → HM.HashMap Text Severity
    parseSeverityMap Nothing = HM.empty
    parseSeverityMap (Just hmv) = HM.mapMaybe mkSeverity hmv
    mkSeverity (String s) = Just (read (unpack s) :: Severity)
    mkSeverity _ = Nothing

    parseBackendMap Nothing = HM.empty
    parseBackendMap (Just hmv) = HM.map mkBackends hmv
    mkBackends (Array bes) = catMaybes $ map mkBackend $ Vector.toList bes
    mkBackends _ = [ ]
    mkBackend (String s) = Just (read (unpack s) :: BackendKind)
    mkBackend _ = Nothing

    parseScribeMap Nothing = HM.empty
    parseScribeMap (Just hmv) = HM.map mkScribes hmv
    mkScribes (Array scs) = catMaybes $ map mkScribe $ Vector.toList scs
    mkScribes (String s) = [(s :: ScribeId)]
    mkScribes _ = [ ]
```

```
mkScribe (String s) = Just (s :: ScribeId)
mkScribe _ = Nothing

parseSubtraceMap :: Maybe (HM.HashMap Text Value) → HM.HashMap Text SubTrace
parseSubtraceMap Nothing = HM.empty
parseSubtraceMap (Just hmv) = HM.mapMaybe mkSubtrace hmv
mkSubtrace (String s) = Just (read (unpack s) :: SubTrace)
mkSubtrace (Object hm) = mkSubtrace′ (HM.lookup "tag" hm) (HM.lookup "contents" hm)
mkSubtrace _ = Nothing
mkSubtrace′ Nothing _ = Nothing
mkSubtrace′ _ Nothing = Nothing
mkSubtrace′ (Just (String tag)) (Just (Array cs)) =
  if tag ≡ "ObservableTrace"
  then Just $ ObservableTrace $ map (λ(String s) → (read (unpack s) :: ObservableInstance)) $ Vector.toLis
  else Nothing
mkSubtrace′ _ _ = Nothing

r_hasEKG r = case (R.hasEKG r) of
  Nothing → 0
  Just p → p
r_hasGUI r = case (R.hasGUI r) of
  Nothing → 0
  Just p → p
r_defaultScribes r = map (λ(k, n) → pack (show k) <> "::" <> n) (R.defaultScribes r)

parseAggregatedKindMap Nothing = HM.empty
parseAggregatedKindMap (Just hmv) =
  let
    listv = HM.toList hmv
    mapAggregatedKind = HM.fromList $ catMaybes $ map mkAggregatedKind listv
  in
  mapAggregatedKind
mkAggregatedKind (name, String s) = Just (name, read (unpack s) :: AggregatedKind)
mkAggregatedKind _ = Nothing
```

Setup empty configuration

```
empty :: IO Configuration
empty = do
  cgref ← newEmptyMVar
  putMVar cgref $ ConfigurationInternal Debug HM.empty HM.empty HM.empty HM.empty [ ] [ ] HM.empty [ ]
  return $ Configuration cgref
```

1.4.22 Cardano.BM.Output.Switchboard

Switchboard

```
type SwitchboardMVar = MVar SwitchboardInternal
newtype Switchboard = Switchboard
```

```
    {getSB :: SwitchboardMVar}
data SwitchboardInternal = SwitchboardInternal
    {sbQueue :: TBQ.TBQueue NamedLogItem
    ,sbDispatch :: Async.Async ()
    }
```

Trace that forwards to the Switchboard

Every Trace ends in the Switchboard which then takes care of dispatching the messages to outputs

```
mainTrace :: Switchboard → TraceNamed IO
mainTrace sb = BaseTrace.BaseTrace $ Op $ λlognamed → do
    effectuate sb lognamed
```

Process incoming messages

Incoming messages are put into the queue, and then processed by the dispatcher.
The queue is initialized and the message dispatcher launched.

```
instance IsEffectuator Switchboard where
    effectuate switchboard item = do
        let writequeue :: TBQ.TBQueue NamedLogItem → NamedLogItem → IO ()
            writequeue q i = do
                nocapacity ← atomically $ TBQ.isFullTBQueue q
                if nocapacity
                then return ()
                else atomically $ TBQ.writeTBQueue q i
        withMVar (getSB switchboard) $ λsb →
            writequeue (sbQueue sb) item
```

Switchboard implements Backend functions

Switchboard is an Declaration of a Backend

```
instance IsBackend Switchboard where
    typeof _ = SwitchboardBK
    realize cfg =
        let spawnDispatcher :: Configuration → [(BackendKind, Backend)] → TBQ.TBQueue NamedLogItem → I
            spawnDispatcher config backends queue =
                let sendMessage nli befilter = do
                        selectedBackends ← getBackends config (lnName nli)
                        let selBEs = befilter selectedBackends
                        forM_ backends $ λ(bek, be) →
                            when (bek ∈ selBEs) (bEffectuate be $ nli)
                    qProc = do
                        nli ← atomically $ TBQ.readTBQueue queue
```

```
                case lnItem nli of
                  KillPill →
                    forM_ backends (λ(_, be) → bUnrealize be)
                  AggregatedMessage _ → do
                    sendMessage nli (filter (≢ AggregationBK))
                    qProc
                  _ → sendMessage nli id ≫ qProc
            in
            Async.async qProc
    in do
    q ← atomically $ TBQ.newTBQueue 2048
    sbref ← newEmptyMVar
    putMVar sbref $ SwitchboardInternal q $ error "unitialized dispatcher"
    let sb :: Switchboard = Switchboard sbref
    backends ← getSetupBackends cfg
    bs ← setupBackends backends cfg sb [ ]
    dispatcher ← spawnDispatcher cfg bs q
    modifyMVar_ sbref $ λsbInternal → return $ sbInternal {sbDispatch = dispatcher}
    return sb
  unrealize switchboard = do
    let clearMVar :: MVar a → IO ()
        clearMVar = void ∘ tryTakeMVar
    (dispatcher, queue) ← withMVar (getSB switchboard) (λsb → return (sbDispatch sb, sbQueue sb))
    -- send terminating item to the queue
    atomically $ TBQ.writeTBQueue queue $ LogNamed "kill.switchboard" KillPill
    -- wait for the dispatcher to exit
    res ← Async.waitCatch dispatcher
    either throwM return res
    (clearMVar ∘ getSB) switchboard
```

Realizing the backends according to configuration

```
setupBackends :: [BackendKind]
          → Configuration
          → Switchboard
          → [(BackendKind, Backend)]
          → IO [(BackendKind, Backend)]
setupBackends [ ] _ _ acc = return acc
setupBackends (bk : bes) c sb acc = do
  be' ← setupBackend' bk c sb
  setupBackends bes c sb ((bk, be') : acc)
setupBackend' :: BackendKind → Configuration → Switchboard → IO Backend
setupBackend' SwitchboardBK _ _ = error "cannot instantiate a further Switchboard"
setupBackend' EKGViewBK c _ = do
  be :: Cardano.BM.Output ∘ EKGView.EKGView ← Cardano.BM.Output ∘ EKGView.realize c
  return MkBackend
```

```
    {bEffectuate = Cardano.BM.Output ∘ EKGView.effectuate be
    ,bUnrealize = Cardano.BM.Output ∘ EKGView.unrealize be
    }
setupBackend' AggregationBK c sb = do
  let trace = mainTrace sb
    ctx = TraceContext {loggerName = " "
         ,configuration = c
         ,minSeverity = Debug
         ,tracetype = Neutral
         ,shutdown = pure ()
         }
  be :: Cardano.BM.Output ∘ Aggregation.Aggregation ← Cardano.BM.Output ∘ Aggregation.realizefrom (ctx,
  return MkBackend
    {bEffectuate = Cardano.BM.Output ∘ Aggregation.effectuate be
    ,bUnrealize = Cardano.BM.Output ∘ Aggregation.unrealize be
    }
setupBackend' KatipBK c _ = do
  be :: Cardano.BM.Output ∘ Log.Log ← Cardano.BM.Output ∘ Log.realize c
  return MkBackend
    {bEffectuate = Cardano.BM.Output ∘ Log.effectuate be
    ,bUnrealize = Cardano.BM.Output ∘ Log.unrealize be
    }
```

1.4.23 Cardano.BM.Output.Log

Internal representation

```
type LogMVar = MVar LogInternal
newtype Log = Log
  {getK :: LogMVar}
data LogInternal = LogInternal
  {kLogEnv :: K.LogEnv
  ,configuration :: Config.Configuration}
```

Log implements *effectuate*

```
instance IsEffectuator Log where
  effectuate katip item = do
    c ← withMVar (getK katip) $ λk → return (configuration k)
    selscribes ← getScribes c (lnName item)
    forM_ selscribes $ λsc → passN sc katip item
```

Log implements backend functions

```haskell
instance IsBackend Log where
  typeof _ = KatipBK
  realize config = do
    let updateEnv :: K.LogEnv -> IO UTCTime -> K.LogEnv
        updateEnv le timer =
          le {K._logEnvTimer = timer, K._logEnvHost = "hostname"}
        register :: [ScribeDefinition] -> K.LogEnv -> IO K.LogEnv
        register [ ] le = return le
        register (defsc : dscs) le = do
          let kind = scKind defsc
              name = scName defsc
              name' = pack (show kind) <> "::" <> name
          scr <- createScribe kind name
          register dscs =<< K.registerScribe name' scr scribeSettings le
        mockVersion :: Version
        mockVersion = Version [0, 1, 0, 0] [ ]
        scribeSettings :: KC.ScribeSettings
        scribeSettings =
          let bufferSize = 5000-- size of the queue (in log items)
          in
          KC.ScribeSettings bufferSize
        createScribe FileTextSK name = mkTextFileScribe (FileDescription $ unpack name) False
        createScribe FileJsonSK name = mkJsonFileScribe (FileDescription $ unpack name) False
        createScribe StdoutSK _ = mkStdoutScribe
        createScribe StderrSK _ = mkStderrScribe
    cfoKey <- Config.getOptionOrDefault config (pack "cfokey") (pack "<unknown>")
    le0 <- K.initLogEnv
            (K.Namespace ["iohk"])
            (fromString $ (unpack cfoKey) <> ":" <> showVersion mockVersion)
    -- request a new time 'getCurrentTime' at most 100 times a second
    timer <- mkAutoUpdate defaultUpdateSettings {updateAction = getCurrentTime, updateFreq = 10000}
    let le1 = updateEnv le0 timer
    scribes <- getSetupScribes config
    le <- register scribes le1
    kref <- newEmptyMVar
    putMVar kref $ LogInternal le config
    return $ Log kref
  unrealize katip = do
    le <- withMVar (getK katip) $ \k -> return (kLogEnv k)
    void $ K.closeScribes le

example :: IO ()
example = do
  config <- Config.setup "from_some_path.yaml"
```

```
  k ← setup config
  passN (pack (show StdoutSK)) k $ LogNamed
    {lnName = "test"
    ,lnItem = LP $ LogMessage $ LogItem
      {liSelection = Both
      ,liSeverity = Info
      ,liPayload = "Hello!"
      }
    }
  passN (pack (show StdoutSK)) k $ LogNamed
    {lnName = "test"
    ,lnItem = LP $ LogValue "cpu-no" 1
    }
```

```
-- useful instances for katip
deriving instance K.ToObject LogObject
deriving instance K.ToObject LogItem
deriving instance K.ToObject (Maybe LogObject)
instance KC.LogItem LogObject where
  payloadKeys _ _ = KC.AllKeys
instance KC.LogItem LogItem where
  payloadKeys _ _ = KC.AllKeys
instance KC.LogItem (Maybe LogObject) where
  payloadKeys _ _ = KC.AllKeys
```

Log.passN

The following function copies the NamedLogItem to the queues of all scribes that match on their name. Compare start of name of scribe to (*show backend* <> "::"). This function is non-blocking.

```
passN :: Text → Log → NamedLogItem → IO ()
passN backend katip namedLogItem = do
  env ← withMVar (getK katip) $ λk → return (kLogEnv k)
  forM_ (Map.toList $ K._logEnvScribes env) $
    λ(scName, (KC.ScribeHandle _ shChan)) →
      -- check start of name to match ScribeKind
        if backend ‘isPrefixOf‘ scName
        then do
          let item = lnItem namedLogItem
          let (sev, msg, payload) = case item of
                (LP (LogMessage logItem)) →
                  (liSeverity logItem, liPayload logItem, Nothing)
                (ObserveDiff counters) →
                  let text = toStrict (encodeToLazyText counters)
                  in
                  (Info, text, Just item)
                (ObserveOpen counters) →
```

```
                let text = toStrict (encodeToLazyText counters)
                in
                (Info, text, Just item)
              (ObserveClose counters) →
                let text = toStrict (encodeToLazyText counters)
                in
                (Info, text, Just item)
              (AggregatedMessage aggregated) →
                let text = T.concat $ (flip map) aggregated $ λ(name, agg) →
                  "\n" <> name <> ":  " <> pack (show agg)
                in
                (Info, text, Nothing)
              (LP (LogValue name value)) →
                (Debug, name <> " = " <> pack (show value), Nothing)
              KillPill →
                (Info, "Kill pill received!", Nothing)
          if (msg ≡ "") ∧ (isNothing payload)
          then return ()
          else do
            threadIdText ← KC.mkThreadIdText < $ > myThreadId
            let ns = lnName namedLogItem
            itemTime ← env ^. KC.logEnvTimer
            let itemKatip = K.Item {
              _itemApp = env ^. KC.logEnvApp
              , _itemEnv       = env ^. KC.logEnvEnv
              , _itemSeverity = sev2klog sev
              , _itemThread   = threadIdText
              , _itemHost     = env ^. KC.logEnvHost
              , _itemProcess  = env ^. KC.logEnvPid
              , _itemPayload  = payload
              , _itemMessage = K.logStr msg
              , _itemTime     = itemTime
              , _itemNamespace = (env ^. KC.logEnvApp) <> (K.Namespace [ns])
              , _itemLoc      = Nothing
              }
            void $ atomically $ KC.tryWriteTBQueue shChan (KC.NewItem itemKatip)
        else return ()
```

Scribes

```
mkStdoutScribe :: IO K.Scribe
mkStdoutScribe = mkTextFileScribeH stdout True
mkStderrScribe :: IO K.Scribe
mkStderrScribe = mkTextFileScribeH stderr True
mkTextFileScribeH :: Handle → Bool → IO K.Scribe
mkTextFileScribeH handler color = do
```

```
    mkFileScribeH handler formatter color
  where
    formatter h colorize verbosity item =
      TIO.hPutStrLn h $! toLazyText $ formatItem colorize verbosity item
mkFileScribeH
    :: Handle
    → (forall a ∘ K.LogItem a ⇒ Handle → Bool → K.Verbosity → K.Item a → IO ())
    → Bool
    → IO K.Scribe
mkFileScribeH h formatter colorize = do
    hSetBuffering h LineBuffering
    locklocal ← newMVar ()
    let logger :: forall a ∘ K.LogItem a ⇒ K.Item a → IO ()
      logger item = withMVar locklocal $ \_ →
        formatter h colorize K.V0 item
    pure $ K.Scribe logger (hClose h)
mkTextFileScribe :: FileDescription → Bool → IO K.Scribe
mkTextFileScribe fdesc colorize = do
    mkFileScribe fdesc formatter colorize
  where
    formatter :: Handle → Bool → K.Verbosity → K.Item a → IO ()
    formatter hdl colorize' v' item =
      case KC._itemMessage item of
        K.LogStr "" →
          -- if message is empty do not output it
          return ()
        _ → do
          let tmsg = toLazyText $ formatItem colorize' v' item
          TIO.hPutStrLn hdl tmsg
mkJsonFileScribe :: FileDescription → Bool → IO K.Scribe
mkJsonFileScribe fdesc colorize = do
    mkFileScribe fdesc formatter colorize
  where
    formatter :: (K.LogItem a) ⇒ Handle → Bool → K.Verbosity → K.Item a → IO ()
    formatter h _ verbosity item = do
      let tmsg = case KC._itemMessage item of
        -- if a message is contained in item then only the
        -- message is printed and not the data
        K.LogStr "" → K.itemJson verbosity item
        K.LogStr msg → K.itemJson verbosity $
          item {KC._itemMessage = K.logStr ("" :: Text)
            ,KC._itemPayload = LogItem Both Info $ toStrict $ toLazyText msg
            }
      TIO.hPutStrLn h (encodeToLazyText tmsg)
mkFileScribe
    :: FileDescription
    → (forall a ∘ K.LogItem a ⇒ Handle → Bool → K.Verbosity → K.Item a → IO ())
```

```
    → Bool
    → IO K.Scribe
mkFileScribe fdesc formatter colorize = do
    let prefixDir = prefixPath fdesc
    (createDirectoryIfMissing True prefixDir)
      `catchIO` (prtoutException ("cannot log prefix directory: " ++ prefixDir))
    let fpath = filePath fdesc
    h ← catchIO (openFile fpath WriteMode) $
        λe → do
          prtoutException ("error while opening log: " ++ fpath) e
          -- fallback to standard output in case of exception
          return stdout
    hSetBuffering h LineBuffering
    scribestate ← newMVar h
    let finalizer :: IO ()
      finalizer = withMVar scribestate hClose
    let logger :: forall a ∘ K.LogItem a ⇒ K.Item a → IO ()
      logger item =
        withMVar scribestate $ λhandler →
          formatter handler colorize K.V0 item
    return $ K.Scribe logger finalizer

formatItem :: Bool → K.Verbosity → K.Item a → Builder
formatItem withColor _verb K.Item {..} =
    fromText header <>
    fromText " " <>
    brackets (fromText timestamp) <>
    fromText " " <>
    KC.unLogStr _itemMessage
  where
    header = colorBySeverity _itemSeverity $
      "[" <> mconcat namedcontext <> ":" <> severity <> ":" <> threadid <> "]"
    namedcontext = KC.intercalateNs _itemNamespace
    severity = KC.renderSeverity _itemSeverity
    threadid = KC.getThreadIdText _itemThread
    timestamp = pack $ formatTime defaultTimeLocale tsformat _itemTime
    tsformat :: String
    tsformat = "%F %T%2Q %Z"
    colorBySeverity s m = case s of
      K.EmergencyS → red m
      K.AlertS     → red m
      K.CriticalS  → red m
      K.ErrorS     → red m
      K.NoticeS    → magenta m
      K.WarningS   → yellow m
      K.InfoS      → blue m
      _ → m
    red = colorize "31"
```

```
    yellow = colorize "33"
    magenta = colorize "35"
    blue = colorize "34"
    colorize c m
      | withColor = "\ESC[" <> c <> "m" <> m <> "\ESC[0m"
      | otherwise = m
-- translate Severity to Log.Severity
sev2klog :: Severity → K.Severity
sev2klog = λcase
    Debug     → K.DebugS
    Info      → K.InfoS
    Notice    → K.NoticeS
    Warning   → K.WarningS
    Error     → K.ErrorS
    Critical  → K.CriticalS
    Alert     → K.AlertS
    Emergency → K.EmergencyS

data FileDescription = FileDescription {
  filePath :: !FilePath}
  deriving (Show)
prefixPath :: FileDescription → FilePath
prefixPath = takeDirectory ∘ filePath
-- display message and stack trace of exception on stdout
prtoutException :: Exception e ⇒ String → e → IO ()
prtoutException msg e = do
  putStrLn msg
  putStrLn ("exception: " ++ displayException e)
```

1.4.24 Cardano.BM.Output.EKGView

Structure of EKGView

```
type EKGViewMVar = MVar EKGViewInternal
newtype EKGView = EKGView
  {getEV :: EKGViewMVar}
data EKGViewInternal = EKGViewInternal
  {evLabels :: HM.HashMap Text Label.Label
  ,evServer :: Server
  ,evTrace  :: Trace IO
  }

ekgTrace :: EKGView → Configuration → IO (Trace IO)
ekgTrace ekg c = do
    let trace = ekgTrace' ekg
```

```
      ctx = TraceContext {loggerName = ""
            , configuration = c
            , minSeverity = Debug
            , tracetype = Neutral
            , shutdown = pure ()
            }
    Trace.subTrace "#ekgview" (ctx, trace)
  where
    ekgTrace' :: EKGView → TraceNamed IO
    ekgTrace' ekgview = BaseTrace.BaseTrace $ Op $ λlognamed → do
      let setlabel :: Text → Text → EKGViewInternal → IO (Maybe EKGViewInternal)
        setlabel name label ekg_i@(EKGViewInternal labels server _) =
          case HM.lookup name labels of
            Nothing → do
              ekghdl ← getLabel name server
              Label.set ekghdl label
              return $ Just $ ekg_i {evLabels = HM.insert name ekghdl labels}
            Just ekghdl → do
              Label.set ekghdl label
              return Nothing
        update :: LogObject → LoggerName → EKGViewInternal → IO (Maybe EKGViewInternal)
        update (LP (LogMessage logitem)) logname ekg_i =
          setlabel logname (liPayload logitem) ekg_i
        update (LP (LogValue iname value)) logname ekg_i =
          let logname' = logname <> "." <> iname
          in
          setlabel logname' (pack $ show value) ekg_i
        update _ _ _ = return Nothing
      ekgup ← takeMVar (getEV ekgview)
      let lognam0 = (lnName lognamed)
        lognam1 = case stripPrefix "#ekgview.#aggregation." lognam0 of
          Nothing → lognam0
          Just ln' → ln'
        logname = case stripPrefix "#ekgview." lognam1 of
          Nothing → lognam1
          Just ln' → ln'
      upd ← update (lnItem lognamed) logname ekgup
      case upd of
        Nothing → putMVar (getEV ekgview) ekgup
        Just ekgup' → putMVar (getEV ekgview) ekgup'
```

EKG view is an effectuator

```
instance IsEffectuator EKGView where
  effectuate ekgview item = do
    ekg ← readMVar (getEV ekgview)
```

```
let trace0 = evTrace ekg
trace ← Trace.appendName (lnName item) trace0
case (lnItem item) of
  AggregatedMessage ags → liftIO $ do
    let traceAgg :: [(Text, Aggregated)] → IO ()
      traceAgg [ ] = return ()
      traceAgg ((n, AggregatedEWMA ewma) : r) = do
        trace′ ← Trace.appendName n trace
        Trace.traceNamedObject trace′ (LP (LogValue "avg" $ avg ewma))
        traceAgg r
      traceAgg ((n, AggregatedStats stats) : r) = do
        trace′ ← Trace.appendName n trace
        Trace.traceNamedObject trace′ (LP (LogValue "mean" (PureD $ meanOfStats stats)))
        Trace.traceNamedObject trace′ (LP (LogValue "min" $ fmin stats))
        Trace.traceNamedObject trace′ (LP (LogValue "max" $ fmax stats))
        Trace.traceNamedObject trace′ (LP (LogValue "count" $ PureI $ fcount stats))
        Trace.traceNamedObject trace′ (LP (LogValue "last" $ flast stats))
        Trace.traceNamedObject trace′ (LP (LogValue "stdev" (PureD $ stdevOfStats stats)))
        traceAgg r
    traceAgg ags
  _ → liftIO $ Trace.traceNamedObject trace (lnItem item)
```

EKGView implements Backend functions

EKGView is an IsBackend

```
instance IsBackend EKGView where
  typeof _ = EKGViewBK
  realize config = do
    evref ← newEmptyMVar
    let ekgview = EKGView evref
    evport ← getEKGport config
    ehdl ← forkServer "127.0.0.1" evport
    ekghdl ← getLabel "iohk-monitoring version" ehdl
    Label.set ekghdl $ pack (showVersion version)
    ekgtrace ← ekgTrace ekgview config
    putMVar evref $ EKGViewInternal
      {evLabels = HM.empty
      ,evServer = ehdl
      ,evTrace = ekgtrace
      }
    return ekgview
  unrealize ekgview = do
    ekg ← takeMVar $ getEV ekgview
    killThread $ serverThreadId $ evServer ekg
```

Interactive testing EKGView

```
test :: IO ()
test = do
  c ← Cardano.BM.Configuration.setup "test/config.yaml"
  ev ← Cardano.BM.Output ∘ EKGView.realize c
  effectuate ev $ LogNamed "test.questions" (LP (LogValue "answer" 42))
  effectuate ev $ LogNamed "test.monitor023" (LP (LogMessage (LogItem Public Warning "!!!! ALARM !
```

1.4.25 Cardano.BM.Output.Aggregation

Internal representation

```
type AggregationMVar = MVar AggregationInternal
newtype Aggregation = Aggregation
  {getAg :: AggregationMVar}
data AggregationInternal = AggregationInternal
  {agQueue :: TBQ.TBQueue (Maybe NamedLogItem)
  ,agDispatch :: Async.Async ()
  }
```

Relation from context name to aggregated statistics

We keep the aggregated values (Aggregated) for a named context in a *HashMap*.

```
type AggregationMap = HM.HashMap Text AggregatedExpanded
```

Info for Aggregated operations

Apart from the Aggregated we keep some valuable info regarding to them; such as when was the last time it was sent.

```
type Timestamp = Word64
data AggregatedExpanded = AggregatedExpanded
  {aeAggregated :: Aggregated
  ,aeResetAfter :: Maybe Integer
  ,aeLastSent :: Timestamp
  }
```

Aggregation implements *effectuate*

Aggregation is an Accepts a NamedLogItem Enter the log item into the Aggregation queue.

```
instance IsEffectuator Aggregation where
  effectuate agg item = do
    ag ← readMVar (getAg agg)
    atomically $ TBQ.writeTBQueue (agQueue ag) $ Just item
```

Aggregation implements Backend functions

Aggregation is an Declaration of a Backend

```
instance IsBackend Aggregation where
  typeof _ = AggregationBK
  realize _ = error "Aggregation cannot be instantiated by 'realize'"
  realizefrom trace0@(ctx, _) _ = do
    trace ← Trace.subTrace "#aggregation" trace0
    aggref ← newEmptyMVar
    aggregationQueue ← atomically $ TBQ.newTBQueue 2048
    dispatcher ← spawnDispatcher (configuration ctx) HM.empty aggregationQueue trace
    putMVar aggref $ AggregationInternal aggregationQueue dispatcher
    return $ Aggregation aggref
  unrealize aggregation = do
    let clearMVar :: MVar a → IO ()
        clearMVar = void ∘ tryTakeMVar
    (dispatcher, queue) ← withMVar (getAg aggregation) (λag →
        return (agDispatch ag, agQueue ag))
    -- send terminating item to the queue
    atomically $ TBQ.writeTBQueue queue Nothing
    -- wait for the dispatcher to exit
    res ← Async.waitCatch dispatcher
    either throwM return res
    (clearMVar ∘ getAg) aggregation
```

Asynchrouniously reading log items from the queue and their processing

```
spawnDispatcher :: Configuration
                → AggregationMap
                → TBQ.TBQueue (Maybe NamedLogItem)
                → Trace.Trace IO
                → IO (Async.Async ())
spawnDispatcher conf aggMap aggregationQueue trace = Async.async $ qProc aggMap
  where
    qProc aggregatedMap = do
      maybeItem ← atomically $ TBQ.readTBQueue aggregationQueue
      case maybeItem of
        Just item → do
          (updatedMap, aggregations) ← update (lnItem item) (lnName item) aggregatedMap
          unless (null aggregations) $
            sendAggregated (AggregatedMessage aggregations) (lnName item)
          qProc updatedMap
        Nothing → return ()
    update :: LogObject
      → LoggerName
```

```
    → HM.HashMap Text AggregatedExpanded
    → IO (HM.HashMap Text AggregatedExpanded, [(Text, Aggregated)])
update (LP (LogValue iname value)) logname agmap = do
  let name = logname <> "." <> iname
  aggregated ←
    case HM.lookup name agmap of
      Nothing → do
        -- if Aggregated does not exist; initialize it.
        aggregatedKind ← getAggregatedKind conf name
        case aggregatedKind of
          StatsAK → return $ singletonStats value
          EwmaAK aEWMA → do
            let initEWMA = EmptyEWMA aEWMA
            return $ AggregatedEWMA $ ewma initEWMA value
      Just a → return $ updateAggregation value (aeAggregated a) (aeResetAfter a)
  now ← getMonotonicTimeNSec
  let aggregatedX = AggregatedExpanded {
            aeAggregated = aggregated
      , aeResetAfter = Nothing
      , aeLastSent = now
      }
      namedAggregated = [(iname, aeAggregated aggregatedX)]
      updatedMap = HM.alter (const $ Just $ aggregatedX) name agmap
  -- use of HM.alter so that in future we can clear the Agrregated
  -- by using as alter's arg a function which returns Nothing.
  return (updatedMap, namedAggregated)
update (ObserveDiff counterState) logname agmap = do
  let counters = csCounters counterState
  (mapNew, aggs) ← updateCounters counters logname agmap [ ]
  return (mapNew, reverse aggs)
-- TODO for text messages aggregate on delta of timestamps
update _ _ agmap = return (agmap, [ ])
updateCounters :: [Counter]
    → LoggerName
    → HM.HashMap Text AggregatedExpanded
    → [(Text, Aggregated)]
    → IO (HM.HashMap Text AggregatedExpanded, [(Text, Aggregated)])
updateCounters [ ] _ aggrMap aggs = return $ (aggrMap, aggs)
updateCounters (counter : cs) logname aggrMap aggs = do
  let name = cName counter
      fullname = logname <> "." <> name
      value = cValue counter
  aggregated ←
    case HM.lookup fullname aggrMap of
        -- if Aggregated does not exist; initialize it.
        Nothing → do
          aggregatedKind ← getAggregatedKind conf fullname
```

```
                case aggregatedKind of
                  StatsAK → return $ singletonStats value
                  EwmaAK aEWMA → do
                    let initEWMA = EmptyEWMA aEWMA
                    return $ AggregatedEWMA $ ewma initEWMA value
            Just a → return $ updateAggregation value (aeAggregated a) (aeResetAfter a)
    now ← getMonotonicTimeNSec
    let aggregatedX = AggregatedExpanded {
                  aeAggregated = aggregated
      , aeResetAfter = Nothing
      , aeLastSent = now
      }
      namedAggregated = (((nameCounter counter) <> "." <> name), aggregated)
      updatedMap = HM.alter (const $ Just $ aggregatedX) fullname aggrMap
    updateCounters cs logname updatedMap (namedAggregated : aggs)
  sendAggregated :: LogObject → Text → IO ()
  sendAggregated aggregatedMsg@(AggregatedMessage _) logname = do
    -- enter the aggregated message into the Trace
    trace' ← Trace.appendName logname trace
    liftIO $ Trace.traceNamedObject trace' aggregatedMsg
  -- ingnore every other message that is not of type AggregatedMessage
  sendAggregated _ _ = return ()
```

Update aggregation

We distinguish an unitialized from an already initialized aggregation. The latter is properly initialized.
We use Welford's online algorithm to update the estimation of mean and variance of the sample statistics. (see https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_Online

```
updateAggregation :: Measurable → Aggregated → Maybe Integer → Aggregated
updateAggregation v (AggregatedStats s) resetAfter =
  let count = fcount s
    reset = maybe False (count ⩾) resetAfter
  in
  if reset
  then
    singletonStats v
  else
    let newcount = count + 1
      newvalue = getDouble v
      delta = newvalue − fsum_A s
      dincr = (delta / fromInteger newcount)
      delta2 = newvalue − fsum_A s − dincr
    in
    AggregatedStats Stats {flast = v
      ,fmin = min (fmin s) v
```

```
        ,fmax = max (fmax s) v
        ,fcount = newcount
        ,fsum_A = fsum_A s + dincr
        ,fsum_B = fsum_B s + (delta * delta2)
        }
updateAggregation v (AggregatedEWMA e) _ =
  AggregatedEWMA $ ewma e v
```

Calculation of EWMA

Following https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average we calculate the exponential moving average for a series of values Y_t according to:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

The pattern matching below ensures that the EWMA will start with the first value passed in, and will not change type, once determined.

```
ewma :: EWMA → Measurable → EWMA
ewma (EmptyEWMA a) v = EWMA a v
ewma (EWMA a (Microseconds s)) (Microseconds y) =
  EWMA a $ Microseconds $ round $ a * (fromInteger y) + (1 - a) * (fromInteger s)
ewma (EWMA a (Seconds s)) (Seconds y) =
  EWMA a $ Seconds $ round $ a * (fromInteger y) + (1 - a) * (fromInteger s)
ewma (EWMA a (Bytes s)) (Bytes y) =
  EWMA a $ Bytes $ round $ a * (fromInteger y) + (1 - a) * (fromInteger s)
ewma (EWMA a (PureI s)) (PureI y) =
  EWMA a $ PureI $ round $ a * (fromInteger y) + (1 - a) * (fromInteger s)
ewma (EWMA a (PureD s)) (PureD y) =
  EWMA a $ PureD $ a * y + (1 - a) * s
ewma _ _ = error "Cannot average on values of different type"
```

Index