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", 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 Measuring *Observable*s

1.2.3 Monitoring

1.2.4 Information reduction in *Aggregation*

1.2.5 Output selection

1.2.6 Setup procedure

1.3 Examples

1.3.1 Observing evaluation of a STM action

1.3.2 Observing evaluation of a monad action

1.4 Code listings

1.4.1 Cardano.BM.Observer.STM

$$stmWithLog :: STM.STM\ (t, [LogObject]) \rightarrow STM.STM\ (t, [LogObject])$$
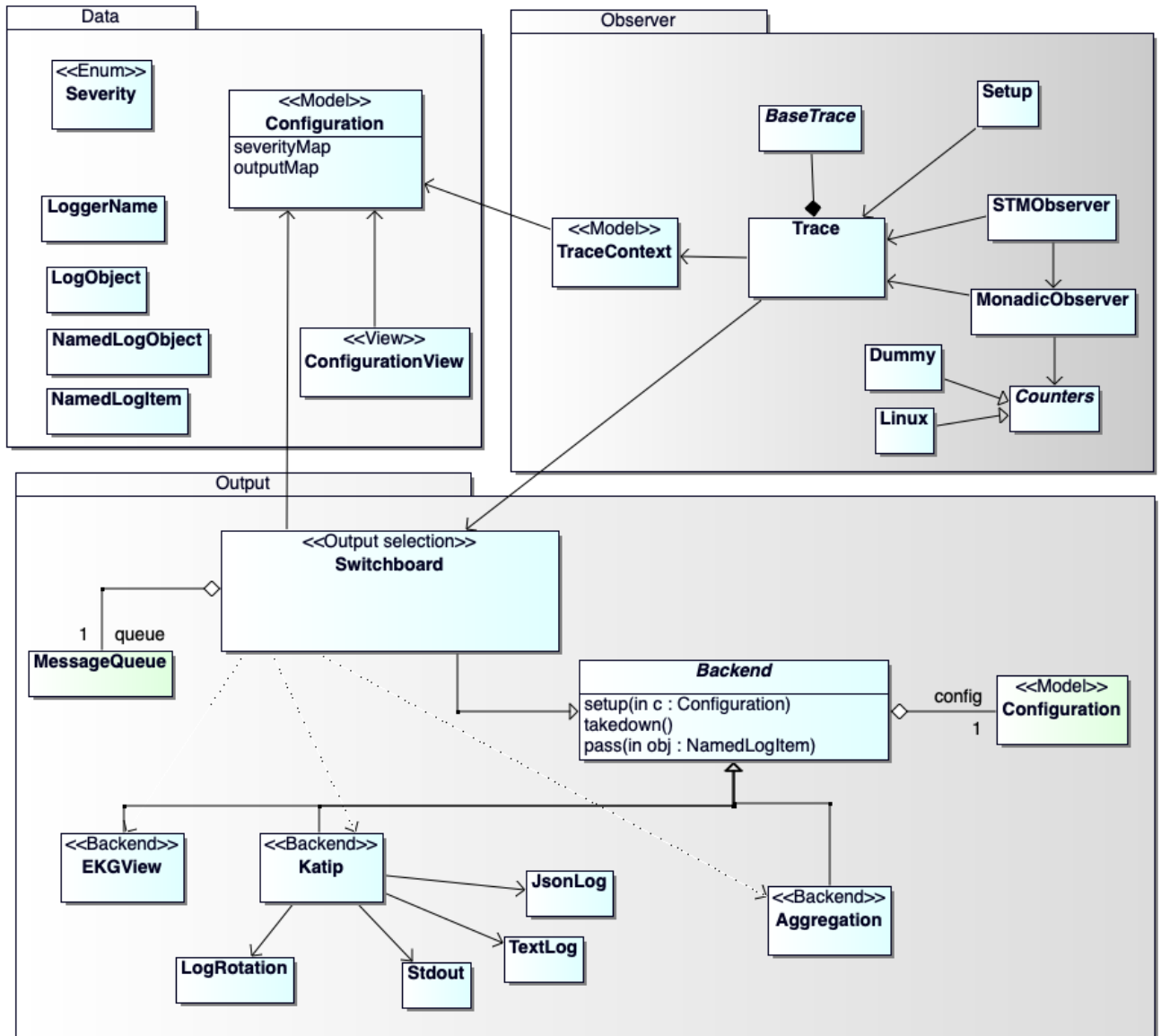$$stmWithLog\ action = action$$

Figure 1.1: Overview of module relationships

Figure 1.2: Setup procedure

Observe *STM* action in a named context

bracketObserveIO :: *Trace IO → Text → STM.STM t → IO t*
bracketObserveIO logTrace0 name action = **do**
  (subtrace, logTrace) ← subTrace name logTrace0
  bracketObserveIO′ subtrace logTrace action

bracketObserveIO′ :: *SubTrace → Trace IO → STM.STM t → IO t*
bracketObserveIO′ NoTrace _ action =
  STM.atomically action
bracketObserveIO′ subtrace logTrace action = **do**
  countersid ← observeOpen subtrace logTrace
  `-- run action, returns result only`
  t ← STM.atomically action
  observeClose subtrace logTrace countersid [ ]
  pure t

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

bracketObserveLogIO :: *Trace IO → Text → STM.STM (t, [LogObject]) → IO t*
bracketObserveLogIO logTrace0 name action = **do**
  (subtrace, logTrace) ← subTrace name logTrace0
  bracketObserveLogIO′ subtrace logTrace action

bracketObserveLogIO′ :: *SubTrace → Trace IO → STM.STM (t, [LogObject]) → IO t*
bracketObserveLogIO′ NoTrace _ action = **do**
  (t, _) ← STM.atomically $ stmWithLog action
  pure t
bracketObserveLogIO′ subtrace logTrace action = **do**
  countersid ← observeOpen subtrace logTrace
  `-- run action, return result and log items`
  (t, as) ← STM.atomically $ stmWithLog action

observeClose subtrace logTrace countersid as
pure t

1.4.2   Cardano.BM.Observer.Monadic

```
-- Observes an action and adds name given in the logger
-- name of the given Trace.  If the empty Text is
-- given as name then the logger name remains untouched.
```
bracketObserveIO :: Trace IO → Text → IO t → IO t
bracketObserveIO logTrace0 name action = **do**
  (subtrace, logTrace) ← subTrace name logTrace0
  bracketObserveIO′ subtrace logTrace action

bracketObserveIO′ :: SubTrace → Trace IO → IO t → IO t
bracketObserveIO′ NoTrace _ action = action
bracketObserveIO′ subtrace logTrace action = **do**
  countersid ← observeOpen subtrace logTrace
  `-- run action`
  t ← action
  observeClose subtrace logTrace countersid [ ]
  pure t

observeOpen :: SubTrace → Trace IO → IO CounterState
observeOpen subtrace logTrace = **do**
  identifier ← newUnique
  logInfo logTrace $ `"Opening: "` *<> pack* (*show $ hashUnique identifier*)
  `-- take measurement`
  counters ← readCounters subtrace
  let *state = CounterState identifier counters*
  `-- send opening message to Trace`
  traceNamedObject logTrace $ ObserveOpen state
  return state

observeClose :: SubTrace → Trace IO → CounterState → [LogObject] → IO ()
observeClose subtrace logTrace (*CounterState identifier _*) *logObjects =* **do**
  logInfo logTrace $ `"Closing: "` *<> pack* (*show $ hashUnique identifier*)
  `-- take measurement`
  counters ← readCounters subtrace
  `-- send closing message to Trace`
  traceNamedObject logTrace $ ObserveClose (*CounterState identifier counters*)
  `-- trace the messages gathered from inside the action`
  forM_ logObjects $ traceNamedObject logTrace

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 → a"*, which when applied to a *BaseTrace* of type *"Op (m ()) s"*, yields *"s → m ()"*. In our case, *Op* accepts an action in a monad *m* with input type *LogNamed LogObject* (see 'Trace').

> **newtype** *BaseTrace m s = BaseTrace {runTrace :: Op (m ()) s}*

contramap

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

In the following instance, *runTrace* extracts type *"Op (m ()) s"* to which contramap applies *f*, thus *"f s → 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 → a"* which translates to *"s → 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

Enter new named context

The context name is created and checked that its size is below a limit (currently 50 chars). The minimum severity that a log message must be labelled with is looked up in the configuration and recalculated.

```
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 50 newName
appendWithDot xs "" = xs
appendWithDot xs newName = T.take 50 $ xs <> "." <> newName


-- return a BaseTrace from a TraceNamed
named :: BaseTrace m (LogNamed i) → LoggerName → BaseTrace m i
named trace name = contramap (LogNamed name) trace
```

> TODO remove *locallock*

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

Trace that forwards to the Switchboard

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

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

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.

```
stdoutTrace :: TraceNamed IO
stdoutTrace = BaseTrace $ Op $ λlognamed →
```

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

Concrete Trace into a *TVar*

```
traceInTVar :: STM.TVar [a] → BaseTrace STM.STM a
traceInTVar tvar = BaseTrace $ Op $ λa → STM.modifyTVar tvar ((:) a)

traceInTVarIO :: STM.TVar [LogObject] → TraceNamed IO
traceInTVarIO tvar = BaseTrace $ Op $ λln →
  STM.atomically $ STM.modifyTVar tvar ((:) (lnItem ln))

traceNamedInTVarIO :: STM.TVar [LogNamed LogObject] → TraceNamed IO
traceNamedInTVarIO tvar = 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)
  ⇒ TraceContext → BaseTrace m LogObject → LogObject
  → m ()
traceConditionally ctx logTrace 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 $ traceWith logTrace msg
traceConditionally _ logTrace logObject = traceWith 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 (ctx, logTrace) p s m =
    let logmsg = LP $ LogMessage $ LogItem {liSelection = p
        , liSeverity = s
        , liPayload = m
        }
    in
    traceConditionally ctx (named logTrace (loggerName ctx)) $ logmsg

logDebug, logInfo, logNotice, logWarning, logError
    :: (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

logDebugS, logInfoS, logNoticeS, logWarningS, logErrorS
    :: (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

logDebugP, logInfoP, logNoticeP, logWarningP, logErrorP
    :: (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

logDebugUnsafeP, logInfoUnsafeP, logNoticeUnsafeP, logWarningUnsafeP, logErrorUnsafeP
    :: (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


traceNamedObject
    :: Trace m
    → LogObject
```

$\rightarrow m$ ()
traceNamedObject (*ctx*, *logTrace*) = *traceWith* (*named logTrace* (*loggerName ctx*))

subTrace

Transforms the *Trace* according to the *Configuration* using the logger name of the current *Trace* appended with the given name. If the empty *Text* is given as name then the logger name remains untouched.

subTrace :: *MonadIO m* ⇒ *T.Text* → *Trace m* → *m* (*SubTrace*, *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 $ (*subtrace*, *tr'*)
     UntimedTrace → **do**
          tr' ← *appendName name tr*
          return $ (*subtrace*, *tr'*)
     NoTrace → *return* (*subtrace*, (*ctx*, *BaseTrace* $ *Op* $ _ → *pure* ()))
     DropOpening → *return* (*subtrace*, (*ctx*, *BaseTrace* $ *Op* $ λ*lognamed* → **do**
       case *lnItem lognamed* **of**
         ObserveOpen _ → *return* ()
         obj → *traceNamedObject tr obj*))
     ObservableTrace _ → **do**
          tr' ← *appendName name tr*
          return $ (*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*

*setupTrace_* :: *MonadIO m* ⇒ *Config.Configuration* → *Text* → *m* (*Trace m*)
setupTrace_ c name = **do**
  sb ← *liftIO* $ *Cardano.BM.Output* ∘ *Switchboard.setup c*
  sev ← *liftIO* $ *Config.minSeverity c*
  ctx ← *liftIO* $ *newContext name c sev*
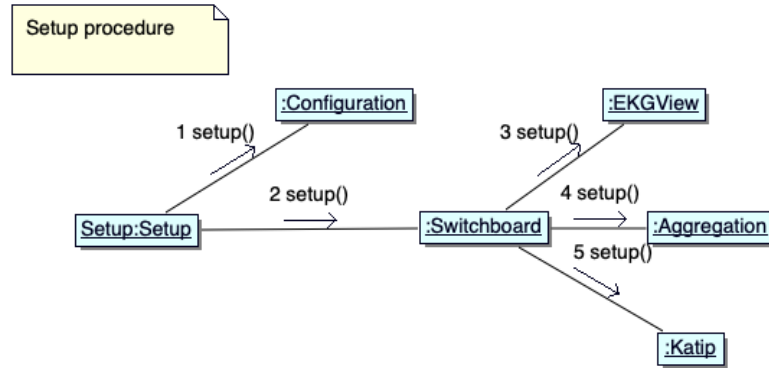
Figure 1.3: Setup procedure

> **let** *logTrace* = (*ctx*, *natTrace liftIO* (*mainTrace sb*))
> (_, *logTrace'*) ← *subTrace* " " *logTrace*
> *return logTrace'*

withTrace

> *withTrace* :: *MonadIO m* ⇒ *Config.Configuration* → *Text* → (*Trace m* → *m t*) → *m t*
> *withTrace cfg name action* = **do**
>   *logTrace* ← *setupTrace* (*Right cfg*) *name*
>   *action logTrace*

newContext

> *newContext* :: *LoggerName* → *Config.Configuration* → *Severity* → *IO TraceContext*
> *newContext name cfg sev* = **do**
>   *return* $ *TraceContext* {
>   *loggerName* = *name*
>   , *configuration* = *cfg*
>   , *minSeverity* = *sev*
>   }

1.4.6   Cardano.BM.Counters

Here the platform is chosen on which we compile this program.
   Currently, we only 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.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
      [(MonotonicClockTime _ micros)] → micros
      _ → error "A time measurement is missing!"

    isMonotonicClockCounter :: Counter → Bool
    isMonotonicClockCounter (MonotonicClockTime _ _) = True
    isMonotonicClockCounter _ = False
```

1.4.7 Cardano.BM.Counters.Common

Common functions that serve *readCounters* on all platforms.

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

getMonoClock :: IO [Counter]
getMonoClock = do
  t ← getMonotonicTimeNSec
  return [MonotonicClockTime "monoclock" $ nominalTimeToMicroseconds t]
```

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 measurement is monotonic clock time for now.

we could well imagine that some day we support all platforms

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

1.4.9   Cardano.BM.Counters.Linux

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

```
readCounters :: SubTrace → IO [Counter]
readCounters NoTrace        = return [ ]
readCounters Neutral        = return [ ]
readCounters UntimedTrace = return [ ]
readCounters DropOpening  = return [ ]
readCounters (ObservableTrace tts) = foldrM (λ(sel, fun) a →
    if sel 'member' 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
```

$pathProcStat\ pid = pathProc </\!/> (show\ pid) </\!/> \texttt{"stat"}$
$pathProcStatM :: ProcessID \rightarrow FilePath$
$pathProcStatM\ pid = pathProc </\!/> (show\ pid) </\!/> \texttt{"statm"}$
$pathProcIO :: ProcessID \rightarrow FilePath$
$pathProcIO\ pid = pathProc </\!/> (show\ pid) </\!/> \texttt{"io"}$

Reading from a file in /proc/<pid >

$readProcList :: FilePath \rightarrow IO\ [Integer]$
$readProcList\ fp = \textbf{do}$
   $cs \leftarrow readFile\ fp$
   $return\ \$\ map\ (\lambda s \rightarrow maybe\ 0\ id\ \$\ (readMaybe\ s :: Maybe\ Integer))\ (words\ cs)$

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)
```

$readProcStatM :: IO\ [Counter]$
$readProcStatM = \textbf{do}$
    $pid \leftarrow getProcessID$
    $ps0 \leftarrow readProcList\ (pathProcStatM\ pid)$
    $ps \leftarrow return\ \$\ zip\ colnames\ ps0$
    $forM\ ps\ (\lambda(n,i) \rightarrow return\ \$\ MemoryCounter\ n\ i)$
  where
    $colnames :: [Text]$
    $colnames = [\texttt{"size"},\texttt{"resident"},\texttt{"shared"},\texttt{"text"},\texttt{"unused"},\texttt{"data"},\texttt{"unused"}]$

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, measured 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 (setpriority(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 priority).

(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*)
    ps ← *return* $ *zip colnames ps0*
    forM ps (λ(*n*, *i*) → *return* $ *StatInfo n i*)
  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).
```

$readProcIO :: IO\,[\,Counter\,]$
$readProcIO = \mathbf{do}$
    $pid \leftarrow getProcessID$
    $ps0 \leftarrow readProcList\;(pathProcIO\;pid)$
    $ps \leftarrow return\;\$\;zip\;colnames\;ps0$
    $forM\;ps\;(\lambda(n,i) \rightarrow return\;\$\;IOCounter\;n\;i)$
  $\mathbf{where}$
    $colnames :: [\,Text\,]$
    $colnames = [\,$"rchar"$,\,$"wchar"$,\,$"syscr"$,\,$"syscw"$,\,$"rbytes"$,\,$"wbytes"$,\,$"cxwbytes"$\,]$

1.4.10   Cardano.BM.Data.Backend

BackendKind

This identifies the backends that can be attached to the Switchboard.

$\mathbf{data}\;BackendKind = AggregationBK$
    $|\;EKGViewBK$
    $|\;KatipBK$
    $|\;DevNullBK$
    $\mathbf{deriving}\;(Generic, Eq, Show, ToJSON, FromJSON)$

Accepts a Type of a logged item

$\mathbf{class}\;HasPass\;t\;\mathbf{where}$
    $pass :: t \rightarrow NamedLogItem \rightarrow IO\,()$

Backend

A backend is referenced through the function *pass'* which accepts a Type of a logged item.

> **data** *Backend = MkBackend {pass' :: NamedLogItem → IO ()}*

1.4.11 Cardano.BM.Data.Configuration

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 representation we implicitly correct it.

> *implicit_fill_representation :: Representation → Representation*
> *implicit_fill_representation =*
>   *union_setup_and_usage_scribes ∘*
>   *union_setup_and_usage_backends ∘*
>   *filter_duplicates_from_backends ∘*
>   *filter_duplicates_from_scribes ∘*
>   *add_katip_if_any_scribes*
>   **where**
>   *union_setup_and_usage_scribes r = r*
>   *union_setup_and_usage_backends r = r*
>   *filter_duplicates_from_backends r = r*
>   *filter_duplicates_from_scribes r = r*
>   *add_katip_if_any_scribes r = r*

```
test_log_backend_1 :: IO ()
test_log_backend_1 =
  BS.putStrLn $
  encode $ ScribeDefinition {scKind = StdoutSK
        ,scName = "testlog"
        ,scRotation = Nothing
        }
test_log_backend_2 :: IO ()
test_log_backend_2 =
  BS.putStrLn $
  encode $ ScribeDefinition {scKind = StdoutSK
        ,scName = "testlog"
        ,scRotation = Just $ RotationParameters 5000000 24 10
        }
test_conf_representation_1 :: IO ()
test_conf_representation_1 =
  BS.putStrLn $
  encode $ Representation
    {minSeverity = Info
    ,rotation = RotationParameters 5000000 24 10
    ,setupScribes =
      [ScribeDefinition {scName = "stdout"
        ,scKind = StdoutSK
        ,scRotation = Nothing}
      ]
    ,defaultScribes = [(StdoutSK, "stdout")]
    ,setupBackends = [EKGViewBK, KatipBK]
    ,defaultBackends = [KatipBK]
    ,hasGUI = Just 12789
    ,hasEKG = Just 18321
    ,options =
      HM.fromList [("test1",(HM.singleton "value" "object1"))
        ,("test2",(HM.singleton "value" "object2"))]
  }
test_conf_representation_2 :: FilePath → IO ()
test_conf_representation_2 fp = do
  repr :: Representation ← decodeFileThrow fp
  BS.putStrLn $ encode repr
```

1.4.12   Cardano.BM.Data.Counter

Counter

```
data Counter = MonotonicClockTime Text Microsecond
  | MemoryCounter Text Integer
  | StatInfo Text Integer
```

```
                | IOCounter Text Integer
                | CpuCounter Text Integer
                        deriving (Eq, Show, Generic, ToJSON)
        instance ToJSON Microsecond where
            toJSON = toJSON ∘ toMicroseconds
            toEncoding = toEncoding ∘ toMicroseconds
```

CounterState

```
        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)
```

1.4.13   Cardano.BM.Data.LogItem

```
        type LoggerName = Text
```

Type of a logged item

```
        type NamedLogItem = LogNamed LogObject
```

LogItem

> TODO *liPayload :: ToObject*

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

        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

```
data LogPrims = LogMessage LogItem
    | LogValue Text Integer
       deriving (Generic, Show, ToJSON)
data LogObject = LP LogPrims
    | ObserveOpen CounterState
    | ObserveClose CounterState
       deriving (Generic, Show, ToJSON)
```

LogNamed

A *LogNamed* contains of a list of context names 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)
```

1.4.14   Cardano.BM.Data.Observable

ObservableInstance

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

1.4.15   Cardano.BM.Data.Output

OutputKind

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

ScribeKind

This identifies katip's scribes by type.

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

1.4.16 Cardano.BM.Data.Severity

Severity

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

1.4.17   Cardano.BM.Data.SubTrace

SubTrace

> **data** *SubTrace = Neutral*
> | *UntimedTrace*
> | *NoTrace*
> | *DropOpening*
> | *ObservableTrace* (*Set ObservableInstance*)
>   **deriving** (*Generic, Show, FromJSON, ToJSON*)

1.4.18   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 LogNamed with payload LogObject.

> **type** *TraceNamed m = BaseTrace m (LogNamed LogObject)*

TraceContext

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

> **data** *TraceContext = TraceContext* {
>   *loggerName :: LoggerName*
>   *, configuration :: Configuration*
>   *, minSeverity :: Severity*
>   }

1.4.19   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*
>
> *parseFrom :: FilePath → IO (CM.Configuration)*
> *parseFrom fp = CM.setup fp*

1.4.20 Cardano.BM.Configuration.Model
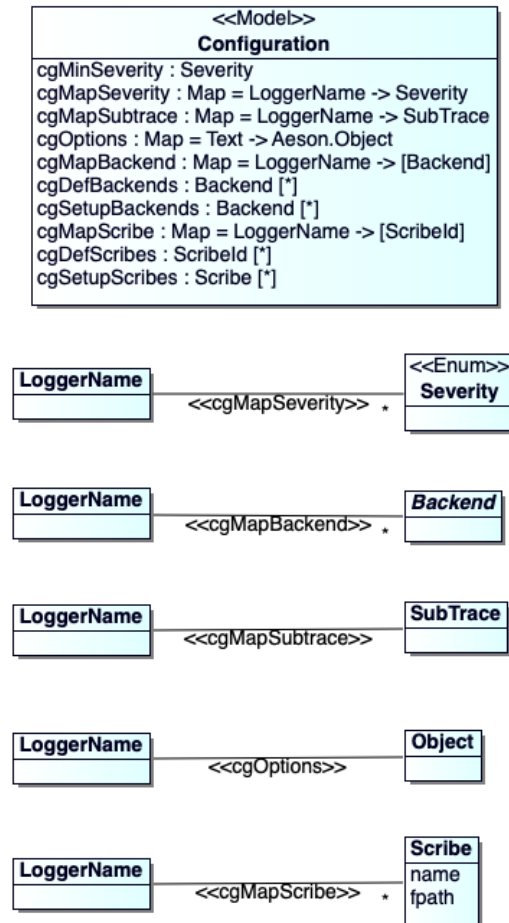
Configuration.Model

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
  ,cgMapSeverity  :: HM.HashMap LoggerName Severity
  ,cgMapSubtrace  :: HM.HashMap LoggerName SubTrace
  ,cgOptions      :: HM.HashMap LoggerName Object
  ,cgMapBackend   :: HM.HashMap LoggerName [Backend]
  ,cgDefBackends  :: [Backend]
  ,cgSetupBackends :: [BackendKind]
```

```
,cgMapScribe    :: HM.HashMap LoggerName [ScribeId]
,cgDefScribes   :: [ScribeId]
,cgSetupScribes :: [ScribeDefinition]
}
```

TODO *listOutput* ← *o* . :? `"output_map"` .! = [ ]

TODO *defaultBackendKinds* ← *o* . :? `"default_backends"` .! = [ ]

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.

$getBackends :: Configuration \rightarrow LoggerName \rightarrow IO\,[Backend]$
$getBackends\ configuration\ name =$
   $withMVar\,(getCG\ configuration)\,\$\,\lambda cg \rightarrow$ **do**
     let $outs = HM.lookup\ name\,(cgMapBackend\ cg)$
     case $outs$ **of**
       $Nothing \rightarrow$ **do**
         $return\,(cgDefBackends\ cg)$
       $Just\ os \rightarrow return\,\$\,os$
$setDefaultBackends :: Configuration \rightarrow [Backend] \rightarrow IO\,()$
$setDefaultBackends\ configuration\ bes =$ **do**
  $cg \leftarrow takeMVar\,(getCG\ configuration)$
  $putMVar\,(getCG\ configuration)\,\$\,cg\,\{cgDefBackends = bes\}$
$registerBackend :: Configuration \rightarrow Text \rightarrow Maybe\ Backend \rightarrow IO\,()$
$registerBackend\ \_\ \_kn\ \_f = pure\,()$-- `TODO`

```
-- registerBackend "some" (Just Backend  pass' = Katip.pass (show StdoutSK) )
-- registerBackend "severe.error" (Just Backend  pass' = Katip.pass "StdoutSK::severe.l
```

Backends to be setup by the *Switchboard*

Defines the list of *Backend*s that need to be setup by the *Switchboard*.

$setSetupBackends :: Configuration \rightarrow [BackendKind] \rightarrow IO\,()$
$setSetupBackends\ configuration\ bes =$ **do**
  $cg \leftarrow takeMVar\,(getCG\ configuration)$
  $putMVar\,(getCG\ configuration)\,\$\,cg\,\{cgSetupBackends = bes\}$

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.

$getScribes :: Configuration \rightarrow LoggerName \rightarrow IO\,[ScribeId]$
$getScribes\ configuration\ name =$

```
    withMVar (getCG configuration) $ λcg → do
      let outs = HM.lookup name (cgMapScribe cg)
      case outs of
        Nothing → do
          return (cgDefScribes cg)
        Just os → return $ os
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}
```

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*)

```
-- if Maybe Severity given is Nothing then the entry for this name is deleted.
```

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*)}

Configuration.Model.setup

setup :: *FilePath* → *IO Configuration*
setup _fp = **do**
  c ← *empty*
  `-- r <- parseRepresentation`
  return c

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

1.4.21   Cardano.BM.Aggregated

module *Cardano.BM.Aggregated*
  (
    Aggregated (..)
  , *Stats* (..)
  , *updateAggregation*
) **where**

```
data Stats = Stats {
  fmin :: Integer,
  fmax :: Integer,
  fcount :: Integer,
  fsum_A :: Integer,
  fsum_B :: Integer
  } deriving (Show, Eq)
data Aggregated = Aggregated {
  fstats :: Stats,
  flast  :: Integer,
  fdelta :: Stats
  } deriving (Show, Eq)
```

Update aggregation

We distinguish an unitialized from an already initialized aggregation:

```
updateAggregation :: Integer → Maybe Aggregated → Maybe Aggregated
updateAggregation v Nothing =
  Just $
    Aggregated {fstats = Stats {
        fmin = v, fmax = v, fcount = 1
        , fsum_A = v, fsum_B = v * v}
      , flast = v
      , fdelta = Stats {
        fmin = 0, fmax = 0, fcount = 0
        , fsum_A = 0, fsum_B = 0}
      }
updateAggregation v (Just (Aggregated (Stats _min _max _count _sumA _sumB)
  _last
  (Stats _dmin _dmax _dcount _dsumA _dsumB)
  )) =
  let delta = v − _last
  in
  Just $
    Aggregated {fstats = Stats {
        fmin = (min _min v)
        , fmax = (max _max v)
        , fcount = (_count + 1)
        , fsum_A = (_sumA + v)
        , fsum_B = (_sumB + v * v)
        }
      , flast = v
      , fdelta = Stats {
        fmin = (min _dmin delta)
        , fmax = (max _dmax delta)
        , fcount = (_dcount + 1)
```

```
            ,fsum_A = (_dsumA + delta)
            ,fsum_B = (_dsumB + delta * delta)
            }
        }
```

1.4.22   Cardano.BM.Output.Switchboard

Switchboard

The switchboard is a singleton.

```
    type SwitchboardMVar = MVar SwitchboardInternal
    newtype Switchboard = Switchboard
      {getSB :: SwitchboardMVar}
    -- Our internal state
    data SwitchboardInternal = SwitchboardInternal
      {sbQueue :: TBQ.TBQueue (Maybe NamedLogItem)
      ,sbDispatch :: Async.Async ()
      ,sbBackends :: [Backend]
      }
```

Starting the switchboard from configuration

The queue is initialized and the message dispatcher launched. TODO: the backends should be connected according to configuration.

```
    setup :: Configuration → IO Switchboard
    setup cfg = do
        ekgv ← Cardano.BM.Output ∘ EKGView.setup cfg
        aggr ← Cardano.BM.Output ∘ Aggregation.setup cfg
        logs ← Cardano.BM.Output ∘ Log.setup cfg
        -- TODO connect backends according to configuration
        let bs = [MkBackend {pass' = Cardano.BM.Output ∘ Log.passN "StdoutSK" logs}
          ,MkBackend {pass' = Cardano.BM.Output ∘ EKGView.pass ekgv}
          ,MkBackend {pass' = Cardano.BM.Output ∘ Aggregation.pass aggr}]

        sbref ← newEmptyMVar
        q ← atomically $ TBQ.newTBQueue 2048
        d ← spawnDispatcher sbref q
        putMVar sbref $ SwitchboardInternal q d bs
        return $ Switchboard sbref
      where
        spawnDispatcher :: SwitchboardMVar → TBQ.TBQueue (Maybe NamedLogItem) → IO (Async.Async ())
        spawnDispatcher switchboard queue = Async.async qProc
          where
            qProc = do
                nli' ← atomically $ TBQ.readTBQueue queue
                case nli' of
```

```
                    Just nli → do
                      putStrLn $ "dispatcher read: " ++ (show nli)
                      withMVar switchboard $ λsb →
                        forM_ (sbBackends sb) (dispatch nli)
                      qProc
                    Nothing → return ()-- end dispatcher
              dispatch :: NamedLogItem → Backend → IO ()
              dispatch nli backend = (pass' backend) nli
```

Process incoming messages

Incoming messages are put into the queue, and then processed by the dispatcher.

```
instance HasPass Switchboard where
  pass switchboard item = do
    let writequeue :: TBQ.TBQueue (Maybe NamedLogItem) → NamedLogItem → STM ()
        writequeue q i = do
          nocapacity ← TBQ.isFullTBQueue q
          if ¬ nocapacity
          then TBQ.writeTBQueue q (Just i)
          else return ()
    putStrLn $ "Cardano.BM.Output.Switchboard.pass " ++ (show item)
    withMVar (getSB switchboard) $ λsb →
      atomically $ writequeue (sbQueue sb) item
```

Halting the switchboard

The queue is flushed before the dispatcher terminates.

```
takedown :: Switchboard → IO ()
takedown switchboard = do
  (q, d) ← withMVar (getSB switchboard) $ λsb →
    return (sbQueue sb, sbDispatch sb)
  -- send terminating item to the queue
  atomically $ TBQ.writeTBQueue q Nothing
  -- wait for the dispatcher to exit
  _ ← Async.waitCatch d
  return ()
```

1.4.23 Cardano.BM.Output.Log

Log is a singleton.

```
type KatipMVar = MVar KatipInternal
newtype Log = Log
  {getK :: KatipMVar}
-- Our internal state
```

```
data KatipInternal = KatipInternal
  {kLogEnv :: K.LogEnv}


instance HasPass Log where
  pass _ _ = pure ()-- error "use passN"
```

Setup *katip* and its scribes according to the configuration

```
setup :: Config.Configuration → IO Log
setup config = do
    cfoKey ← Config.getOptionOrDefault config (pack "cfokey") (pack "<unknown>")
    -- TODO setup katip
    le0 ← K.initLogEnv
      (K.Namespace ["ouroboros-bm"])
      (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
    stdoutScribe ← mkStdoutScribeJson K.V0
    le ← register [(StdoutSK, "stdout", stdoutScribe)] le1

    kref ← newEmptyMVar
    putMVar kref $ KatipInternal le
    let katipref = Log kref
    setDefaultBackends config
        [MkBackend {pass' = Cardano.BM.Output ∘ Log.passN (pack (show StdoutSK)) katipref}
        ,MkBackend {pass' = Cardano.BM.Output ∘ Log.passN (pack (show FileTextSK)) katipref}
        ,MkBackend {pass' = Cardano.BM.Output ∘ Log.passN (pack (show FileJsonSK)) katipref}
        ]
    return katipref
  where
    updateEnv :: K.LogEnv → IO UTCTime → K.LogEnv
    updateEnv le timer =
        le {K._logEnvTimer = timer, K._logEnvHost = "hostname"}
    register :: [(ScribeKind, Text, K.Scribe)] → K.LogEnv → IO K.LogEnv
    register [] le = return le
    register ((kind, name, scribe) : scs) le =
        let name' = pack (show kind) <> "::" <> name in
        register scs ≪ K.registerScribe name' scribe scribeSettings le
    mockVersion :: Version
    mockVersion = Version [0, 1, 0, 0] []
    scribeSettings :: KC.ScribeSettings
    scribeSettings = KC.ScribeSettings bufferSize
      where
        bufferSize = 5000-- size of the queue (in log items)


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


passN :: Text → Log → NamedLogItem → IO ()
passN backend katip namedLogItem = withMVar (getK katip) $ λk → do
  -- TODO go through list of registered scribes
  -- and put into queue of scribe if backend kind matches
  -- compare start of name of scribe to (show backend <> "::")
  let env = 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)
            _ → (Info, "", Just item)
          threadIdText ← KC.mkThreadIdText < $ > myThreadId
          let ns = lnName namedLogItem
          itemTime ← env ^. KC.logEnvTimer
          let itemKatip = K.Item {
            _itemApp     = env ^. KC.logEnvApp
```

```
              , _itemEnv       = env ˆ. KC.logEnvEnv
              , _itemSeverity = sev2klog sev
              , _itemThread   = threadIdText
              , _itemHost      = env ˆ. KC.logEnvHost
              , _itemProcess   = env ˆ. KC.logEnvPid
              , _itemPayload  = payload
              , _itemMessage = K.logStr msg
              , _itemTime      = itemTime
              , _itemNamespace = (env ˆ. KC.logEnvApp) <> (K.Namespace [ns])
              , _itemLoc        = Nothing
              }
        atomically $ KC.tryWriteTBQueue shChan (KC.NewItem itemKatip)
     else return False
```

Scribes

```
mkStdoutScribe :: K.Verbosity → IO K.Scribe
mkStdoutScribe = mkTextFileScribeH stdout True

mkStdoutScribeJson :: K.Verbosity → IO K.Scribe
mkStdoutScribeJson = mkJsonFileScribeH stdout True

mkStderrScribe :: K.Verbosity → IO K.Scribe
mkStderrScribe = mkTextFileScribeH stderr True

mkJsonFileScribeH :: Handle → Bool → K.Verbosity → IO K.Scribe
mkJsonFileScribeH handler color verb = do
    mkFileScribeH handler formatter color verb
  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
          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)

mkTextFileScribeH :: Handle → Bool → K.Verbosity → IO K.Scribe
mkTextFileScribeH handler color verb = do
    mkFileScribeH handler formatter color verb
  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 ())-- format and output f
    → Bool-- whether the output is colourized
    → K.Verbosity
```

```haskell
    → IO K.Scribe
mkFileScribeH h formatter colorize verbosity = do
    hSetBuffering h LineBuffering
    locklocal ← newMVar ()
    let logger :: forall a ∘ K.LogItem a ⇒ K.Item a → IO ()
        logger item = withMVar locklocal $ \_ →
          formatter h colorize verbosity item
    pure $ K.Scribe logger (hClose h)

mkTextFileScribe :: FileDescription → Bool → Severity → K.Verbosity → IO K.Scribe
mkTextFileScribe fdesc colorize s v = do
    mkFileScribe fdesc formatter colorize s v
  where
    formatter :: Handle → Bool → K.Verbosity → K.Item a → IO ()
    formatter hdl colorize' v' item = do
        let tmsg = toLazyText $ formatItem colorize' v' item
        TIO.hPutStrLn hdl tmsg

mkFileScribe
    :: FileDescription
    → (forall a ∘ K.LogItem a ⇒ Handle → Bool → K.Verbosity → K.Item a → IO ())-- format and output f
    → Bool-- whether the output is colourized
    → Severity
    → K.Verbosity
    → IO K.Scribe
mkFileScribe fdesc formatter colorize _ v = 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 v 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


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

The ekgview is a singleton.

```
type EKGViewMVar = MVar EKGViewInternal
newtype EKGView = EKGView
  {getEV :: EKGViewMVar}
-- Our internal state
data EKGViewInternal = EKGViewInternal
  {evGauges :: HM.HashMap Text Gauge.Gauge
  ,evLabels  :: HM.HashMap Text Label.Label
  , _ekgServer :: Server
  }


setup :: Configuration → IO EKGView
setup _ = do
  evref ← newEmptyMVar
  ehdl ← forkServer "127.0.0.1" 16543
  putMVar evref $ EKGViewInternal HM.empty HM.empty ehdl
  return $ EKGView evref


instance HasPass EKGView where
  pass ekgview item =
    let update :: LogObject → LoggerName → EKGViewInternal → IO (Maybe EKGViewInternal)
        update (LP (LogMessage logitem)) logname ekg@(EKGViewInternal _ labels server) =
          case HM.lookup logname labels of
            Nothing → do
              ekghdl ← getLabel logname server
              Label.set ekghdl (liPayload logitem)
              return $ Just $ ekg {evLabels = HM.insert logname ekghdl labels}
            Just ekghdl → do
              Label.set ekghdl (liPayload logitem)
              return Nothing
        update (LP (LogValue iname value)) logname ekg@(EKGViewInternal gauges _ server) =
          let name = logname <> "." <> iname
          in
          case HM.lookup name gauges of
            Nothing → do
              ekghdl ← getGauge name server
              Gauge.set ekghdl (fromInteger value)
              return $ Just $ ekg {evGauges = HM.insert name ekghdl gauges}
            Just ekghdl → do
              Gauge.set ekghdl (fromInteger value)
              return Nothing

        update _ _ _ = return Nothing
    in do
    ekg ← takeMVar (getEV ekgview)
```

```
upd ← update (lnItem item) (lnName item) ekg
case upd of
   Nothing → putMVar (getEV ekgview) ekg
   Just ekg' → putMVar (getEV ekgview) ekg'
```

1.4.25   Cardano.BM.Output.Aggregation

The aggregation is a singleton.

```
type AggregationMVar = MVar AggregationInternal
newtype Aggregation = Aggregation
   {getAg :: AggregationMVar}
-- Our internal state
data AggregationInternal = AggregationInternal
   {agMap  :: HM.HashMap Text Aggregated
   ,agSome :: [Int]-- TODO
   }


inspect :: Aggregation → Text → IO (Maybe Aggregated)
inspect agg name =
   withMVar (getAg agg) $ λag →
      return $ HM.lookup name (agMap ag)


setup :: Configuration → IO Aggregation
setup _ = do
   aggref ← newEmptyMVar
   -- TODO create thread which will periodically output
   -- aggregated values to the switchboard
   putMVar aggref $ AggregationInternal HM.empty [ ]
   return $ Aggregation aggref


pass :: Aggregation → NamedLogItem → IO ()
pass agg item = do
      ag ← takeMVar (getAg agg)
      putMVar (getAg agg) $ AggregationInternal (updated $ agMap ag) (agSome ag)
   where
      updated agmap = pass' (lnItem item) (lnName item) agmap
      pass' :: LogObject → LoggerName → HM.HashMap Text Aggregated → HM.HashMap Text Aggregated
      pass' (LP (LogValue iname value)) logname agmap =
         let name = logname <> "." <> iname
         in
         HM.alter (λm → updateAggregation value m) name agmap
      -- TODO for text messages aggregate on delta of timestamps
      pass' _ _ agmap = agmap
```