Robust Distributed Software Transactions for Haskell

Application Manual

Frank Kupke

July 28, 2010

Contents

1 Application Programming using DSTM

The Distributed Software Transactional Memory (DSTM) library enables the designer of robust distributed applications to focus on the application logic itself rather than on complex synchronization techniques. DSTM builds on the STM API known from Concurrent Haskell and extends it for the use in distributed systems. This manual explains the additional requirements to use the distributed library version. The application programmer needs to:

- Initialize the distributed functionality
- Run a name server application on one node
- Register with the name server `TVar`s to be shared with other nodes
- Lookup from the name server `TVar`s shared from other nodes
- Catch transactional exception to add robustness

The application programmer should encapsulate each of the node main programs in a `startDist` call to properly initialize the distributed system.

2 Name Server

The name server application maintains a dictionary of distributed link `TVar`s (line 1). It accepts messages of type `NameServerMsg` to register, unregister, and lookup `TVar`s (line 2).

```
1 type TVarDict = [(String, VarLink)]

2 data NameServerMsg = Reg    String VarLink
3                    | UnReg  String
4                    | Lookup String
```

The application programmer starts a name server application on an arbitrary node within the distributed system. We show a straightforward example name server application (lines 5–26). It creates a socket listening on a TCP port randomly chosen from the private port section (`www.iana.org/assignments/port-numbers`), continuously checks for messages, and maintains the `TVar` dictionary depending on the received messages.

Note that we create a new connection for every name server message (lines 11, 15) unlike the optimized communication. Typically, in a distributed system, name server communication is a relatively rare event. Also, note that

we unregister a possibly registered `TVar` before registering it thus remapping the dictionary entry (line 21).

```
 5 main :: IO ()
 6 main = do
 7   s <- listenOn (PortNumber 60000)
 8   readMsg [] s

 9 readMsg :: TVarDict -> Socket -> IO ()
10 readMsg tVarDict s = do
11   (h, _, _) <- accept s
12   str <- hGetLine h
13   newTable <- case reads str of
14     ((msg,_):_) -> handleMsg h msg tVarDict
15   hClose h
16   readMsg newTable s

17 handleMsg :: Handle -> NameServerMsg -> TVarDict
18              -> IO TVarDict
19 handleMsg h msg tVarDict = case msg of
20   Reg name tVar -> return $
21     (name, tVar) : filter ((name/=) . fst) tVarDict
22   UnReg name -> return $
23     filter ((name/=) . fst) tVarDict
24   Lookup name -> do
25     hPutStrLn h (show (lookup name tVarDict))
26     return tVarDict
```

The DSTM library interface provides functions for a proper communication with the name server. The application should use `registerTVar` to register a `TVar` with the name server. Note that the function also registers the `TVar` actions within the library (line 31) which is necessary for a proper DSTM communication. The `deregisterTVar` function is added for completeness. It is not required in a DSTM system.

```
27 registerTVar :: Dist a => String -> TVar a -> String
28                            -> IO ()
29 registerTVar server tVar name = do
30   putServerLn server (Reg name (tVarToLink tVar))
31   regTVars gMyEnv tVar

32 deregisterTVar :: String -> String -> IO ()
33 deregisterTVar server name =
34   putServerLn server (UnReg name)
```

The `lookupTVar` function provides the interface to reveal a `TVar` from the name server. It also properly handles finalizing the internal library communication (line 42). Note that we annotate the `TVar` type when converting the generic link `TVar` (line 41) thus enabling the type system to select the type correct `finTVars` instance. In order to use the type variable `a` inside the function body we have to declare it `forall` quantified.

```
35 lookupTVar :: forall a . Dist a => String -> String
36                -> IO (Maybe (TVar a))
37 lookupTVar server name = do
38   answer <- getServerLn server (Lookup name)
39   case reads answer of
40     ((Just link,_):_) -> do
41          let tVar::TVar a = LinkTVar link
42          finTVars tVar
43          return (Just tVar)
44     _ -> return Nothing
```

The name server message exchange routines form the low-level socket communication.

```
45 putServerLn :: String -> a -> IO ()
46 putServerLn nameServer msg = do
47   h <- connectTo nameServer (PortNumber 60000)
48   hPutStrLn h (show msg)
49   hClose h

50 getServerLn :: String -> a -> IO String
51 getServerLn nameServer msg = do
52   h <- connectTo nameServer (PortNumber 60000)
53   hPutStrLn h (show msg)
54   hFlush h
55   answer <- hGetLine h
56   hClose h
57   return answer
```

3 Robustness

The DSTM library provides one exception indicating a failure of one or more `TVars`. The library throws an exception of type `SomeDistTVarException` whenever it detects such a failure during any arbitrary `atomic` transaction.

The library ensures that any transaction which executes while a `TVar` failure occurs continues and finishes consistently without further precautions from the application. However, in general, the application itself may not continue properly as intended.

For the application programmer a failing `TVar` is identical to a `TVar` that is not accessible any more and never will be accessible again. The `TVar` symbolizes a specific service of the distributed program. The application designer should catch any `SomeDistTVarException` exception and react on it appropriately by replacing the lost service or by ensuring that the service is no longer needed.

The exception signals that at least one `TVar` accessed in an `atomic` transaction is not accessible any more. While the application may chose to continuously access failed `TVar`s without risking any inconsistencies, the library will throw further `SomeDistTVarException` exceptions which is not very efficient. Therefore, after catching a `SomeDistTVarException` exception, the application should identify which `TVar`s are no longer accessible by testing each `TVar` used in the failing transaction with the `isDistErrTVar` predicate. The predicate requires both the exception and the to be tested `TVar` and returns `True` if the `TVar` is unavailable and `False` otherwise. Note that any thrown exception of type `SomeDistTVarException` is referentially transparent. It may be used at any time to test `TVar` accessibility. The test result will always be identical.

A design pattern for adding robustness to a DSTM application may look like this:

```
58 foo tVar = Control.Exception.catch (do
59   ...
60   atomic $ do
61     ... tVar
62   )(\e -> if (isDistErrTVar e tVar) ...)
```

We give an example of implementing a robust application using robust software transactions in Subsection 5.2.

4 DSTM Library API

This is the complete DSTM library application programmer interface (API) for reference.

```
63  data STM a  -- abstract
64  instance Monad STM

65  -- Running STM computations
66  atomic :: STM a -> IO a
67  retry  :: STM a
68  orElse :: STM a -> STM a -> STM a

69  -- Transactional variables
70  data TVar a  -- abstract

71  newTVar   :: Dist a => a -> STM (TVar a)
72  readTVar  :: Dist a => TVar a -> STM a
73  writeTVar :: Dist a => TVar a -> a -> STM ()

74  -- Exceptions
75  throw :: SomeException -> STM a
76  catch :: STM a -> (SomeException -> STM a) -> STM a

77  -- Additional distributed interface
78  class (Show a, Read a) => Dist a where
79    regTVars :: EnvAddr -> a -> IO ()
80    finTVars :: a -> IO ()

81  startDist      :: IO a -> IO a
82  registerTVar   :: Dist a => String -> TVar a -> String
83                              -> IO ()
84  deregisterTVar :: String -> String -> IO ()
85  lookupTVar     :: Dist a => String -> String
86                              -> IO (Maybe (TVar a))

87  -- Additional robustness interface
88  data SomeDistTVarException  -- abstract

89  isDistErrTVar :: SomeDistTVarException -> TVar a
90                   -> Bool
```

5 Sample Applications

We present three example DSTM applications, each focusing on a special aspect of using the DSTM library. The first application is a distributed version of the classic *Dining Philosophers* problem used to demonstrate problems of and solutions for concurrent and distributed programming. We use it as an example to introduce the basic idea of designing an application with distributed `TVar`s.

The next application is a simple internet *Chat* program. With this example we show how to use a custom data type for `TVar` values and thus how to define class `Dist` instance functions that unwrap the custom `TVar` type constructors. We also introduce the usage of the library robustness functions to make the application itself robust against unexpected faults like suddenly unavailable chat participants.

Our final example is a distributed *Bomberman* game implementation. Naturally, the focus is on the application being a useful example of utilizing our library rather than the game being a breathtaking entertainment. We use this example, however, to show that the DSTM library can be used also in a soft real-time environment like a distributed game and scales well with a larger amount of `TVar`s. Also, we make a more elaborate approach to application robustness in case of disappearing game participants.

5.1 Dining Philosophers

The hallmark example of a concurrent and distributed application is the *Dining Philosophers* problem formulated by Edsger Wybe Dijkstra in 1971. We show a simple program running each one of a total of three philosopher processes when started with its ordinary number as argument.

We import `Control.Distributed.STM.DSTM` (line 2) for the library and `NameService` (line 3) for the nameserver. We synchronize solely on the sticks between the philosophers modeled as `TVar`s of type `Bool` (line 7). For simplicity we run the name server on the same process node as the philosopher processes (line 8) which in reality would be some specific domain name. Note that we call `getLine` (line 33) just to allow a somewhat synchronized start of all processes. Figure 1 symbolizes the output of three philosopher processes just started, each on a separate terminal shell.

```
1 module Main where

2 import Control.Distributed.STM.DSTM
3 import Control.Distributed.STM.NameService
```

```
4  import Prelude
5  import System
6  import System.IO

7  type Stick = TVar Bool

8  gNameServer = "localhost"

9  takeStick :: Stick -> STM ()
10 takeStick s = do
11   b <- readTVar s
12   if b
13     then writeTVar s False
14     else retry

15 putStick :: Stick -> STM ()
16 putStick s = writeTVar s True

17 phil :: Int ->Int -> Stick -> Stick -> IO ()
18 phil i n l r = do
19   atomic $ do
20     takeStick l
21     takeStick r
22   putStrLn (show n ++ ". Phil is eating "++show i)
23   atomic $ do
24     putStick l
25     putStick r
26   phil (i+1) n l r

27 main :: IO ()
28 main = startDist $ do
29   (arg:_) <- getArgs
30   let n= read arg
31   l <- atomic $ newTVar True
32   registerTVar gNameServer l arg
33   getLine
34   (Just r) <- lookupTVar gNameServer
35                          $ show ((n `mod` 3) + 1)
36   phil 1 n l r
```

```
> main 1                 > main 2                 > main 3

1. Phil is eating 1      2. Phil is eating 1      3. Phil is eating 1
1. Phil is eating 2      2. Phil is eating 2      3. Phil is eating 2
1. Phil is eating 3      2. Phil is eating 3      3. Phil is eating 3
1. Phil is eating 4      2. Phil is eating 4      3. Phil is eating 4
1. Phil is eating 5      ...                      3. Phil is eating 5
1. Phil is eating 6                               ...
...
```

Figure 1: Dining Philosopher Sample Output

5.2 Chat

The *Chat* application is a classic example for a distributed program. An arbitrary number of users, each at a computer network connected to the internet, communicates with each other. There is one dedicated host server. The client users register with the server and subsequently send messages to the server which broadcasts them to all registered clients.

We show both, a simple chat server and a simple chat client communicating with each other using `TVar`s to synchronize. Therefore, we design a custom data type `ServerCmd` providing commands to join (line 40) and leave (line 42) a chat and to distribute messages (line 41). The `ServerCmd` alternative *join* contains a mutually recursive defined type `CmdTVar` (line 44). This `TVar` may contain a command generated by a chat client and interpreted by the chat server.

In order for the DSTM library to properly communicate `TVar`s, we make the custom `TVar` type an instance of type class `Dist` and therefore import the `Dist` module (line 39). Both, `regTVars` (line 46) and `finTVars` (line 48) methods unwrap the application defined constructor from the `TVar`. Alternatives containing no `TVar`s simply return the unit value.

```
37 module ChatData where

38 import Control.Distributed.STM.DSTM
39 import Control.Distributed.STM.Dist

40 data ServerCmd = Join String CmdTVar
41                | Msg String String
42                | Leave String
43   deriving (Show,Read)
```

```
44 type CmdTVar = TVar (Maybe ServerCmd)

45 instance Dist ServerCmd where
46   regTVars env (Join _ cmd) = regTVars env cmd
47   regTVars _ _ = return ()

48   finTVars (Join _ cmd) = finTVars cmd
49   finTVars _ = return ()
```

We designate one `CmdTVar` to each participating process discriminating the chat server `TVar`. The chat server application registers its own `TVar` with the name server (line 61).

```
50 -- Chat Server
51 module Main where

52 import ChatData
53 import Control.Distributed.STM.DSTM
54 import Control.Distributed.STM.NameService
55 import Control.Exception as CE
56 import DebugTrans
57 import Maybe

58 main :: IO ()
59 main = startDist $ do
60   inVar <- atomic $ newTVar Nothing
61   registerTVar "localhost" inVar "Chat"
62   chatServer inVar []
```

The `chatServer` function (line 63) loops forever watching its `TVar` for client messages. It dynamically builds and updates a dictionary of all participating client `CmdTVar`s with the client name as key. The server realizes the watch mechanism by reading the `TVar` (line 66) and suspending itself calling `retry` if it contains `Nothing`. Note that it also reads all dictionary client `TVar`s without using their values (line 69) and suspends itself calling `retry`. We use this construct to perform a simple failure recovery. If some client becomes unavailable and no other client is sending a chat message the transparent DSTM link mechanism still detects the fault and throws a library exception.

If the server `TVar` contains a chat command (line 71), it resets the message, broadcasts a corresponding message to all dictionary clients, and maintains the dictionary accordingly.

```
63 chatServer :: CmdTVar -> [(String, CmdTVar)] -> IO ()
64 chatServer inCmd dict = CE.catch (do
65   newDict <- atomic $ do
66     cmd <- readTVar inCmd
67     case cmd of
68       Nothing -> do
69                  mapM_ (readTVar.snd) dict
70                  retry
71       Just serverCmd -> do
72         writeTVar inCmd Nothing
73         case serverCmd of
74           Join name msgVar -> do
75             mapM_ (flip writeTVar msg.snd) dict
76             return ((name,msgVar): dict)
77             where msg = Just (Msg name " joint")
78           Msg _ _ -> do
79             mapM_ (flip writeTVar cmd.snd) dict
80             return dict
81           Leave name -> do
82             mapM_ (flip writeTVar msg.snd) dic
83             return dic
84             where msg = Just (Msg name " left")
85                   dic = filter ((/=name).fst) dict
86   chatServer inCmd newDict
87   )(\e -> chatServer inCmd (removeErrDict e dict))
```

We catch any `SomeDistTVarException` arising from unavailable `TVar`s. Hence, we detect unexpectedly disappearing chat clients and continue the server loop with a dictionary cleaned from any disappeared client (line 87). The DSTM predicate `isDistErrTVar` facilitates the erroneous `TVar` detection (line 92).

```
88 removeErrDict :: SomeDistTVarException
89                  -> [(String, CmdTVar)]
90                  -> [(String, CmdTVar)]
91 removeErrDict e dict =
92   [d | d <- dict, not (isDistErrTVar e (snd d))]
```

The chat client application first looks up the chat server represented by its `CmdTVar` (line 105) from the name server. If found, it joins the chat submitting its new empty client `TVar` (line 111). Then the client starts the threads `stdinClient` to manage the user input and `serverClient` to handle chat server messages.

```
 93 -- Chat Client
 94 module Main where

 95 import ChatData
 96 import Control.Concurrent
 97 import Control.Distributed.STM.DSTM
 98 import Maybe
 99 import Control.Distributed.STM.NameService
100 import System.IO

101 main :: IO ()
102 main = startDist $ do
103   putStrLn "Your Name: "
104   name <- getLine
105   serverTVar <- lookupTVar "localhost" "Chat"
106   case serverTVar of
107     Nothing -> putStrLn "Chat server not reachable"
108     Just cmdTVar -> do
109       myTVar <- atomic $ do
110         new <- newTVar Nothing
111         writeTVar cmdTVar (Just (Join name new))
112         return new
113       forkIO (serverClient myTVar)
114       stdinClient name cmdTVar
```

We simply encode any user message into a client `TVar` command (line 124). If the user message is to terminate the chat, we encode the according command (line 121) and terminate the client itself.

```
115 stdinClient :: String -> CmdTVar -> IO ()
116 stdinClient name cmdTVar = do
117   putStrLn (name ++ " >")
118   msg <- getLine
119   if msg == "bye"
120     then atomic $
121            writeTVar cmdTVar (Just (Leave name))
122     else do
123          atomic $
124            writeTVar cmdTVar (Just (Msg name msg))
125          stdinClient name cmdTVar
```

The `serverClient` thread watches its own client `CmdTVar` (line 129), suspends using `retry` when there is no message, and prints and resets any received message.

```
126 serverClient :: CmdTVar -> IO ()
127 serverClient myTVar = do
128   s <- atomic $ do
129     cmd <- readTVar myTVar
130     case cmd of
131       Nothing -> retry
132       Just (Msg name msg) -> do
133         writeTVar myTVar Nothing
134         return (name ++ ": " ++ msg)
135       _ -> return ""
136   putStrLn s
137   serverClient myTVar
```

Note that the shown program is very elementary. A more useable chat application would probably synchronize using a transactional channel thus prohibiting the loss of single messages if the server is not able to respond to all requests in time.

```
> main
Your Name:
Curry
Curry >
Haskell: Hello Curry
Oh, hi Haskell
Curry >
Curry: Oh, hi Haskell
Haskell: Good to see you
You Too. Got to go
Curry >
Curry: You Too. Got to go
bye
>
```

```
> main
Your Name:
Haskell
Haskell >
Curry:  joint
Hello Curry
Haskell >
Haskell: Hello Curry
Curry: Oh, hi Haskell
Good to see you
Haskell >
Haskell: Good to see you
Curry: You Too. Got to go
Curry:  left
...
```

Figure 2: Chat Sample Output

Figure 2 shows the output of a sample chat session with two clients. Note that we show each output in a self-contained sequence side by side with the other. There is no common time line among the two terminal transcripts.

5.3 Bomberman

With the *Bomberman* game application we provide a somewhat more complex distributed program probing the performance of the DSTM library in a soft real-time environment. The idea of the game is that all participating players move around in a shared game field. The players can walk in four directions. There is empty space allowing to walk around. There are arbitrary walls in the field to block the player from passing.

The goal is to eliminate the opponents by dropping bombs, hence the name of the game. A dropped bomb explodes delayed to allow the player to leave the area. An exploding bomb destroys the field position it is on itself plus the four surrounding positions. Exploding bombs destroy wall elements and opponents and immediately ignite dropped but not yet exploded bombs. Figure 3 shows a screen shot of the terminal of one player while gaming with two opponents.

```
W W W W W W W W W W
W     @           W
W       W W W
W     W W W W
W       W W W
W   X X W W               o: player
W X X X X      .@         @: opponent
W   X X        .          .: bomb
W              o          X: explosion
W W               W       W: wall element
```

Figure 3: Bomberman Game Screen Shot

In this section we describe the major design ideas behind our *Bomberman* adaptation rather than every concrete implementation detail for the sake of clarity.

The *Bomberman* main data structure is the `GameState` record of system states (lines 146–161) represented by various `TVar`s properly synchronizing the process state view of each *Bomberman* thread with other threads and with the other player processes. The game field (line 140) consists of rows and columns of possible field elements as shown in Figure 3.

```
138 data Element = Empty | Wall | Player | Opponent
139              | Bomb | XPlosion
140 type Field   = [[Element]]
```

We design player positions as points, bombs as a list of points, and explosions surrounding each bomb as a list of lists of points. Each *Bomberman* instance records its user commands as moves. A `Dead` move symbolizes a killed player.

```
141 data Point = Point Int Int
142 type Bombs = [Point]
143 type Xplos = [Bombs]

144 data Move = MoveLeft | MoveRight | MoveUp | MoveDown
145           | DropBomb | Dead
```

The `GameState` record consists of `TVar`s like `repaint` (line 148) designed for *intra*-process synchronization[1] while others are designed for an additional *inter*-process synchronization like the `repaints TVar` (line 149).

```
146 data GameState = GameState {
147   move       :: TVar (Maybe Move),
148   repaint    :: TVar Bool,
149   repaints   :: TVar [TVar Bool],
150   field      :: TVar Field,
151   player     :: TVar Point,
152   opponents  :: TVar [TVar Point],
153   plBombs    :: TVar Bombs,
154   plXplosion :: TVar Xplos,
155   plBCount   :: TVar Int,
156   bombs      :: TVar [TVar Bombs],
157   xplosion   :: TVar [TVar Xplos],
158   bCounts    :: TVar [TVar Int],
159   quit       :: TVar Bool,
160   quits      :: TVar [TVar Bool]
161   }
```

Each *Bomberman* instance runs in either autonomous, master, or slave mode. The first mode is a concurrent one process game. The last two are used in a distributed game with exactly one master player and an arbitrary number of slave players. The master player hosts all unique status elements like the playing field. Each player hosts individual status elements like its next move.

[1] We distinguish here between intra-process (thread to thread) and inter-process (node to node) synchronization for explanatory reasons, only. Its application is fully transparent.

After initialization, each player node starts the game calling `launchGame`. The function starts threads to concurrently display any change of the field elements (line 164), to control the player (line 165), and to read the user input (line 166). The player thread, when dropping bombs, in turn forks a new thread for each bomb. Any bomb thread autonomously manages the behavior of its bomb including the delayed explosion.

```
162 launchGame :: GameState -> IO Int
163 launchGame gameState = do
164   forkIO (view gameState) >> return ()
165   forkIO (player gameState) >> return ()
166   input gameState
```

Table 1 gives an overview of the synchronization task each `TVar` manages. We explain the idea behind it with the *repaint* example. A similar mechanism operates on the other `TVar` combinations.

Intra-Process TVars	Inter-Process TVars	View-Threads	Input-Threads	Player-Threads	Bomb-Threads
	field	✗		✗	✗
move			✗	✗	
player		✗		✗	✗
	opponents	✗		(✗)	(✗)
repaint		✗			
	repaints	(✗)		✗	✗
plBombs					✗
plBlasts					✗
plBCount					✗
	bombs	✗		(✗)	(✗)
	blasts	✗		✗	✗
	bCounts	(✗)	✗	✗	✗
quit			✗		
	quits		✗		

Table 1: Regular ✗ and Recovery (✗) `TVar` Synchronization

Each player creates a `repaint TVar` predicate initialized to `False`. The view thread checks `repaint` and redraws the field if the predicate holds. Otherwise, the view retries thus suspending itself. The other threads set `repaint` to `True` whenever they change a field element and hence schedule a redraw. This mechanism is sufficient for a concurrent scenario with a single

player. In order to manage a distributed game we include a `repaints TVar` containing a list of `repaint TVar`s. The master player hosts the `repaints TVar`. Each slave player hosts its own `repaint TVar` and inserts it into `repaints`. In this design, scheduling a redraw simply requires to set all `repaint` predicates in the `repaints` list. The view thread design is identical to the concurrent scenario.

We also provide system recovery in case of unavailable slave players. The other players, including the master, properly remove all references to the faulty player and continue with the game. The master itself is essential to our implementation of the game. However, one could implement recovery means to replace a faulty master player as well. Providing a backup master player or designing a system that enables clients to take over the master player functionality, however, results in a significantly higher system complexity.

The DSTM library requires all data types of `TVar` values to be made a `Dist` type class instance. Other than the types used in the *Chat* sample program (see Section 5.2), the *Bomberman* types are standard compound Haskell types like `Maybe` and `[]` for which we have already defined the necessary instance functions within the library. Therefore, we make the instance functions simply returning `()`. None of the custom data types include any `TVar`s themselves (lines 167–175).

```
167 instance Dist Move where
168   finTVars _   = return ()
169   regTVars _ _ = return ()

170 instance Dist Point where
171   finTVars _   = return ()
172   regTVars _ _ = return ()

173 instance Dist Element where
174   finTVars _   = return ()
175   regTVars _ _ = return ()
```