Utilizing Transactional Events in Haskell in the Implementation of Message Broker Middleware

Christopher R. Wicks <cw9887@rit.edu>
Under the advisement of Dr. James Heliotis

Introduction

Transactional events are a relatively recent concurrency abstraction in the functional programming paradigm that combine the first-class synchronous message-passing events of Concurrent ML with all-or-nothing transactions [1]. STOMP, the Simple (or Streaming) Text Oriented Message Protocol is a text-based message passing protocol from the same school of design as HTTP [2], used primarily in the domain of message-oriented middleware.

In this project, an experimental transactional events library [1] built on top of Concurrent Haskell is utilized in implementing a STOMP message broker, client applications, and shared libraries for dealing with the STOMP protocol.

STOMP Frames [2]

Client Frames
CONNECT / STOMP
DISCONNECT
SEND
SUBSCRIBE
UNSUBSCRIBE
BEGIN
COMMIT
ABORT
ACK
NACK

Server Frames
CONNECTED
MESSAGE
RECEIPT
ERROR

https://stomp.github.io

Transactional Events Interface [1]

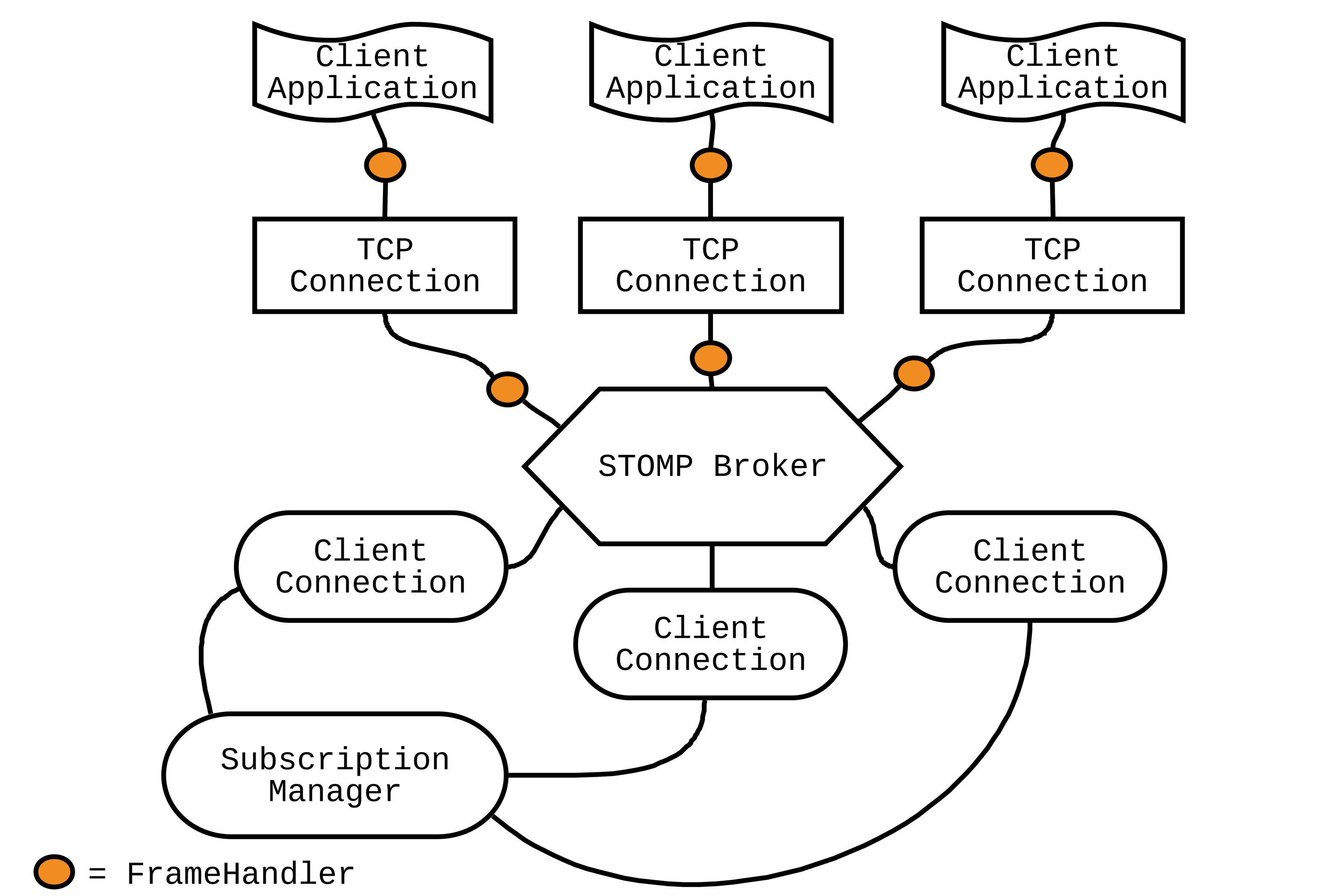
```
-- The Evt monad
data Evt a

sync       :: Evt a -> IO a
thenEvt    :: Evt a -> (a -> Evt b) -> Evt b
alwaysEvt  :: a -> Evt a
chooseEvt  :: Evt a -> Evt a -> Evt a
neverEvt   :: Evt a

instance Monad Evt where
     (>>=)  = thenEvt
     return = alwaysEvt

instance MonadPlus Evt where
     mplus = chooseEvt
     mzero = neverEvt

data SChan a -- Synchronous channels
newSChan :: Evt (SChan a)
sendEvt  :: SChan a -> a -> Evt ()
recvEvt  :: SChan a -> Evt a
```

Transactional Events

- The transactional events interface given above is used by composing events with the provided combinators: *thenEvt, alwaysEvt, chooseEvt,* and *neverEvt*
- We use the *sync* function to synchronize on an event; synchronization blocks until the event is able to complete, and the result is returned wrapped in the *IO* monadic context
- The interface also provides synchronous event channels that allow us to pass messages between concurrently executing threads using the *sendEvt* and *recvEvt* functions
- Each call to *sendEvt* in one thread must be matched to a corresponding call to *recvEvt* in a different thread, using the same *SChan*, in order to synchronize; the following program would block indefinitely if a new thread were not spawned for the send, or if there were no receive:

```
  -- "Hello, world!" program using transactional events
main :: IO ()
main = do
    channel <- sync newSChan
    forkIO $ sync (sendEvt channel "Hello, world!")
    s <- sync $ recvEvt channel
    putStrLn s
```

STOMP Broker Architecture



Stomp.Frames.IO and the FrameHandler

The Stomp.Frames.IO module provides STOMP client and server applications with an error-handling, stream-based frame parser coupled with transactional events-based I/O abstraction that guarantess synchronous access to the underlying TCP connection

```
data FrameEvt     = NewFrame Frame |
                    ParseError String |
                    Heartbeat |
                    GotEof |
                    TimedOut

data FrameHandler = FrameHandler Handle (SChan SendEvt) (SChan FrameEvt) ThreadId ThreadId

initFrameHandler :: Handle -> IO FrameHandler
initFrameHandler handle = do
    writeChannel <- sync newSChan
    readChannel  <- sync newSChan
    wTid <- forkIO $ frameWriterLoop handle 0 writeChannel
    rTid <- forkIO $ frameReaderLoop handle readChannel
    return $ FrameHandler handle writeChannel readChannel wTid rTid

getEvt :: FrameHandler -> Evt FrameEvt
getEvt (FrameHandler _ _ readChannel _ _) = recvEvt readChannel

putEvt :: Frame -> FrameHandler -> Evt ()
putEvt frame (FrameHandler _ writeChannel _ _ _) =
    sendEvt writeChannel $ SendFrame frame

getEvtWithTimeOut :: FrameHandler -> Int -> Evt FrameEvt
getEvtWithTimeOut (FrameHandler _ _ readChannel _ _) timeOut =
    if timeOut < 1 then
        recvEvt readChannel
    else
        (recvEvt readChannel) `chooseEvt`
            (timeOutEvt timeOut `thenEvt` (\_ -> alwaysEvt TimedOut))
```

References

[1] K. Donnelly and M. Fluet, “Transactional events,” SIGPLAN Not., vol. 41, no. 9, pp. 124–135, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1160074.1159821

[2] “Stomp 1.2 specification.” [Online]. Available: https://stomp.github.io

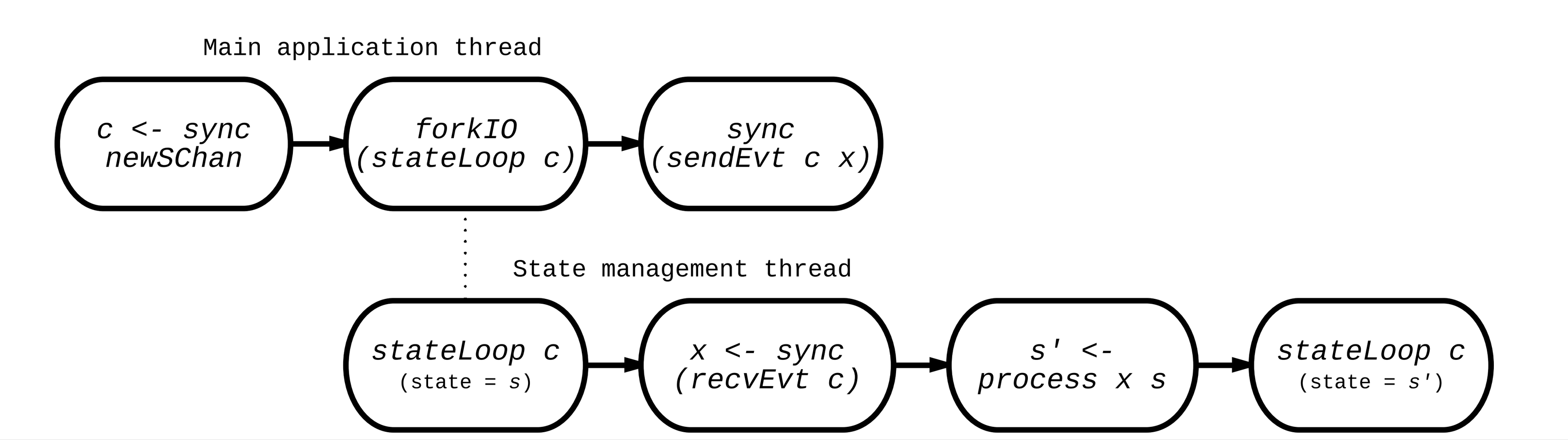
[3] L. Effinger-Dean, M. Kehrt, and D. Grossman, “Transactional events for ml,” SIGPLAN Not., vol. 43, no. 9, pp. 103–114, Sep. 2008. [Online]. Available: http://doi.acm.org.ezproxy.rit.edu/10.1145/1411203.1411222

Non-deterministic Choice of Consumer

- The distributed model enabled by the STOMP broker is that of many clients acting as both producers and consumers
- An individual message is produced by one client and consumed by another
- How do we choose which client consumes a message on a given destination?
- Transactional events make the "right" choice straightforward to implement in the Subscription Manager: simply take the first client that is able to synchronize on a dynamically constructed transactional event
- The event is recursively constructed with all active subscribers for a given destination using a *timeOutEvt* as the base case

```
clientChoiceEvt :: Frame -> HashMap ClientId ClientSub -> Evt (Maybe ClientSub)
clientChoiceEvt frame = HM.foldr (partialClientChoiceEvt frame)
    $ (timeOutEvt 500000) `thenEvt` (\_ -> alwaysEvt Nothing)

partialClientChoiceEvt :: Frame -> ClientSub -> Evt (Maybe ClientSub) -> Evt (Maybe ClientSub)
partialClientChoiceEvt frame sub@(ClientSub clientId _ _ frameHandler) =
    chooseEvt $ (putEvt frame frameHandler) `thenEvt` (\_ -> alwaysEvt $ Just sub)
```

Application State Management



Stomp.Transaction and the ClientTransactionManager

- STOMP supports client-initated named transactions; every client connection to the STOMP broker maintains a *ClientTransactionManager* instance
- When a **BEGIN** frame is received, we initialize the named transaction by inserting a unit-valued *alwaysEvt* into the mapping
- As the frames that make up a transaction are received, a transactional event is dynamically constructed using the *thenEvt* combinator
- This ensures that when we synchronize on the final event upon receipt of a **COMMIT** frame from the client, all component events occur in sequence

```
-- Example of dynamically constructed event in which two messages
-- are sent as part of a transaction
alwaysEvt ()
   ‘thenEvt‘ ((\_ -> sendMessageEvt subManager d1 f1)
       (‘thenEvt‘ (\_ -> sendMessageEvt subManager d2 f2)))
```

- This requires multiple sequential send/receive matches with the SubscriptionManager *(sendMessageEvt),* so it is essential that the state management loop in that module supports this type of synchronization.
- We adopt a pattern for looping receive given in [3] to achieve this
- Instead of synchronizing on a single update and using that to update the state of the Subscriptions, we create a composite event that has the ability to synchronize on arbitrarily many sequential updates
- We achieve this through use of the sequencing monadic bind (which uses the *thenEvt* combinator) and the *chooseEvt* combinator
- Since we are sequencing the event using Haskell's *do* notation, every event in the sequence must be able to complete; if the initiating event is a sequence of sends on the *updateChan,* the composite would not be able to complete if the *alwaysEvt* were chosen, so the *updateEvtLoop* choice *must* be taken until there are no more sends that need to be matched

```
-- |State management loop for Subscriptions
updateLoop :: SChan Update -> Subscriptions -> IO ()
updateLoop updateChan subs = do
    subs' <- sync $ updateEvtLoop updateChan subs
    updateLoop updateChan subs'

-- |Looping event to handle the possiblity of multiple subsequent transactional
-- synchronizations in the updateLoop
updateEvtLoop :: SChan Update -> Subscriptions -> Evt Subscriptions
updateEvtLoop updateChan subs = do
    update <- recvEvt updateChan
    subs'  <- handleUpdate update subs updateChan
    (alwaysEvt subs') `chooseEvt` (updateEvtLoop updateChan subs')
```