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

Christopher R. Wicks
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
cw9887@cs.rit.edu

***Abstract*—Transactional events are used in the implementation of message broker middleware. In particular, STOMP (Simple Text Oriented Messaging Protocol) client APIs, shared libraries, and a message broker are implemented utilizing an experimental transactional events library built on Concurrent Haskell.**

***Index Terms*—Transactional Events, Concurrency, Haskell, Functional Programming, Monad, Message Broker, Message Oriented Middleware, STOMP**

I. Introduction

Transactional events are a relatively recent concurrency abstraction in the functional programming paradigm that allow inter-thread communications to be composed as all-or-nothing transactions. STOMP, the Simple (or Streaming) Text Oriented Message Protocol, is a text-based message passing protocol from the same school of design as HTTP [1]. It is designed so that applications in a distributed software system can communicate easily (from the perspective of an application developer) over a network.

In this work, the use of an experimental transactional events library is explored in the implementation and API design of a collection of software tools and applications that implement the STOMP protocol. We implement a client API, a command-line client application, a message broker (server), and shared libraries utilizing Transactional Events Haskell (TE Haskell). In particular, we identify useful idioms, patterns, and interesting use cases for transactional events as a programming model, and establish the model as an approach that is well-suited to the implementation of large-scale software projects that must leverage concurrency.

II. The STOMP Protocol

STOMP is used primarily in the domain of message-oriented middleware. In this domain, many clients communicate with one or more servers, or message brokers, via some protocol. The message brokers then contain the necessary routing and transformation logic to pass that information on to other interested clients.

STOMP is designed to be a very simple and easy to use protocol. The specification for STOMP version 1.2 specifies 11 different client frames and 4 different server frames. One particularly interesting aspect of the protocol for the purposes of this work is the support of client-initiated transactions.

The STOMP protocol is *text-based*, with the command and header portion of the frames encoded using UTF-8. The frame body, if present, may be encoded arbitrarily, and may even contain binary data. A frame consists of a command, zero or more headers followed by a blank line, and a body which may be empty. A frame must always be terminated by a null octet (^@):

```
COMMAND
header1:value1
header2:value2

Message body^@
```

In the STOMP protocol, connected clients act as either producers of messages, consumers of messages, or both (see Figure 1). A client will send a message to a specific *destination* using the SEND client frame; the message is then queued up for consumption by a *subscriber*. When a client subscribes to a given destination by sending a SUBSCRIBE frame to the STOMP broker, it registers itself with the broker as a willing consumer of messages that are sent to that destination.

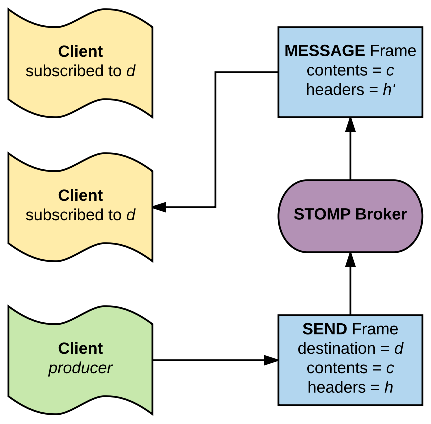


Fig. 1. Producer/Consumer of a STOMP Frame

A. Client Frame Commands

The CONNECT and STOMP frames are functionally equivalent; STOMP is preferred, as in some cases a broker may need to implement logic for differentiating between STOMP and some other protocol, but CONNECT is preserved for backwards compatibility. These frames allow the client to initiate a connection to a STOMP broker. The DISCONNECT frame allows a client to gracefully disconnect from a session with a STOMP broker. The SUBSCRIBE and UNSUBSCRIBE client frames allow a client to register and deregister their willingness to receive messages from a given destination. The ACK and NACK frames allow a client to either acknowledge of disacknowledge the processing of a received message. The BEGIN, COMMIT, and ABORT frames, respectively, allow the client to begin, commit, or abort a named transaction.

B. Server Frame Commands

The server only has 4 frames that it may send to a client. The CONNECTED frame acknowledges a successful connection request from the client and contains the results of both protocol negotiations (the highest protocol version supported by both client and server) and heartbeat negotiations. The MESSAGE frame contains message data from a registered destination. The RECEIPT frame contains a receipt for a frame sent by the client in which a receipt was requested via a special header. The ERROR frame indicates a protocol or processing error. After sending an ERROR frame, the protocol dictates the the server close the connection immediately.

C. Transactions

STOMP supports client-initated named transactions, in which it is requested that all messages sent as part of a given transaction are processed atomically by the server. STOMP brokers support the ability for a single client to have multiple ongoing transactions at any given time (the frames need not be sent atomically, or even successively), with the server keeping track of the state. Only SEND, ACK, and NACK frames may be part of a transaction, with the BEGIN, COMMIT, and ABORT frames managing the control. Whether or not a message is part of a transaction, or the specification of which transaction a control frame is referring to, is handled using the transaction frame header. An example of a SEND frame that is to be processed as part of the named transaction *tx1* is given:

```
SEND
destination:q1
content-type:text/plain
content-length:13
transaction:tx1

Hello, world!^@
```

D. Heart-beating

STOMP supports bidirectional heart-beats between the client and server. If heart-beats are expected, the sender must send either a frame or a heart-beat at least once every *n* milliseconds. A heart-beat is represented by sending an end-of-line (EOL) character to the receiver. If a longer interval transpires without having received any data, the receiver may consider the connection lost and close it. The client will send, in the STOMP or CONNECT frame, the minimum rate of heart-beats that it is able to both send and receive, if any. The server then compares those values to its own capabilities, and chooses the longest interval (or, if either client or server does not wish to send or receive heartbeats, 0 is chosen). An example of the STOMP frame is given:

```
STOMP
accept-version:1.0,1.1,1.2
host:wicks.com
heart-beat:4000,0

^@
```

The first value in the heart-beat header indicates that the client is able to send heart-beats as quickly as once every 4000 milliseconds. The second value in the heart-beat header indicates that the client does not wish to receive heart-beats from the server. If, for example, the server to which it is connecting only wishes to receive heart-beats every 10000 milliseconds, the corresponding CONNECTED acknowledgement would look as follows:

```
CONNECTED
accept-version:1.2
heart-beat:0,10000
```

Note that the value of zero indicates that the server will not be sending heart-beats, even if it did support such, as the client has requested not to receive them. The value of 10000 indicates that based on the client's capabilities and its own expectations that it expects to receive either a heartbeat every 10000 milliseconds (the maximum value between the client's send rate and its own receive rate). Also note that heart-beating capabilities are dictated by the client. A server is free to reject a client if its heart-beating capabilities are not up to its minimum requirements, but this is an implementation detail and is not dictated by the protocol.

III. Transactional Events

Transactional events are a high-level concurrency abstraction that combine the first-class synchronous message-passing events of Concurrent ML (CML) with all-or-nothing transactions [2]. Their inception draws inspiration from CML, Concurrent Haskell, and Software Transactional Memory (STM) Haskell. There currently exist implementations as language extensions for both Haskell [2] and ML [3].

Transactional events are primarily motivated by the need for better abstractions in the development of concurrent programs. The non-deterministic execution order of concurrent programs makes them inherently difficult to reason about. Transactional events help to ease some of the pain by allowing for inter-thread communications to be composed of modular events in the form of transactions.

A. Haskell Interface

The basic transactional events interface given by Fluet and Donnelly [2] is provided in Figure 2 for ease of refernece.

```
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
```

Fig. 2. The Evt Monad

This interface is used by composing events using the provided event combinators: *thenEvt, alwaysEvt, chooseEvt*, and *neverEvt*. We then use the *sync* function to synchronize on an event. Said synchronization blocks until the event is able to complete, and the result is returned wrapped in the *IO* monadic context. The synchronous channels allow us to create event channels that can be used to pass messages between concurrently executing threads. There are additional combinators, *sendEvt* and *recvEvt*, that allow the composition of events that pass messages between multiple threads. Any send on a synchronous channel must be matched with a corresponding receive in another thread on the same channel in order to be able to synchronize. A simple "Hello, world!" program using transactional events can be constructed as demonstrated in Figure 3.

```
import Control.Concurrent
import Control.Concurrent.TxEvent

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

Fig. 3. "Hello, World!" Using Transactional Events

This program first synchronizes on the creation of a new synchronous communications channel, forks off a thread that sends a String on the channel, and then blocks until it receives a String on the same channel, finally displaying it on the standard output. Note that if we were to attempt to write this program without forking off the send to take place in a new thread, the initial synchronization on *sendEvt* would block indefinitely, waiting for a receiver to receive the string.

One of the primary attributes of transactional events that makes them an attractive tool for concurrent programming, and the one that sets the idiom apart from the similar message-passing style found in CML, is the fact that they are easily composable due to their monadic structure. As seen here, in the Haskell implementation the *Evt* typeclass implements both the *Monad* and *MonadPlus* interfaces. This allows for easy sequencing of events utilizing Haskell's *do* notation, as the monadic bind is implemented as the *thenEvt* combinator. Using this combinator, one can compose an event comprised of smaller events that must be able to complete in sequence. If any individual event in the composite is unable to complete, the entire event "fizzles" without any side-effects that are observable by the rest of the system, providing a transactional guarantee for event synchronization.

The interface also provides a non-deterministic choice combinator in the form of *chooseEvt*. This allows an event to synchronize on whichever of the two events it is able to synchronize on first, effectively aborting the computations performed in the event that is not chosen. Note that *neverEvt* is the event that can never be synchronized upon. Hence, sequencing a *neverEvt* using the monadic bind is an effective abort. Likewise the *chooseEvt*, as it cannot choose an event that will not successfully complete, will never synchronize on the *neverEvt*.

Using this interface, one can easily compose sophisticated computational event structures that allow for inter-thread communication but guarantee that irrevocable side effects cannot take place during event synchronization (hence the "all-or-nothing" transactional claim). Indeed, transactional events have been shown to be strictly more expressive than either STM Haskell and CML [2]. The authors provide encodings for both idioms using transactional events. The then demonstrate that one can express computations that are not possible in either, such as *n*-way synchronization between threads.

IV. Library Design & Usage

The *tehstomp-lib* package that was developed as part of this project provides modules that export useful functions for developers that are working with STOMP-based applications in Haskell. Where appropriate (that is, where doing so does not expose unsafe operations) both *IO*- and *Evt*-based functions are exported for operations that involve side-effects. This provides the caller some flexibility in the design of their applications. If the caller does not wish to manage the details of transactional event synchronization, they can simply call the *IO*-based functionality. If they wish to incorporate library events into an application that also utilizes transactional events, the *Evt*-based functions are provided so that they may be composed as part of larger event synchronizations.

A. *Stomp.Frames*

The initial challenge in implementing the shared libraries was in determining exactly how STOMP frames should be represented, and how they would be passed back-and-forth over a TCP connection. The *Frame* data type was designed such that it can be easily parsed and constructed piecemeal while reading bytes from a socket handle (see Figure 4).

```
-- |A HeaderName is a type synonym for String, and
-- is one component of a Header
type HeaderName     =   String

-- |A HeaderValue is a type synonym for String, and
-- is one component of a Header
type HeaderValue    =   String

-- |A Header contains a name and a value for a
-- STOMP header
data Header         =   Header HeaderName HeaderValue

-- |A Headers is a recursive, list-like data structure
-- representing the set of Headers for a STOMP frame
data Headers        =   Some Header Headers | EndOfHeaders

-- |A Body represents the body of a STOMP frame. It
-- is either empty or consists of a single ByteString.
data Body           =   EmptyBody | Body ByteString

-- |A Command represents the command portion of
-- a STOMP Frame.
data Command        =   SEND |
                        SUBSCRIBE |
                        UNSUBSCRIBE |
                        BEGIN |
                        COMMIT |
                        ABORT |
                        ACK |
                        NACK |
                        DISCONNECT |
                        CONNECT |
                        STOMP |
                        CONNECTED |
                        MESSAGE |
                        RECEIPT |
                        ERROR deriving Show

-- |An abstract data type representing a STOMP Frame.
-- It consists of a Command, Headers, and Body.
data Frame          =   Frame Command Headers Body
```

Fig. 4. The Frame Data Type

The definition for the *Frame* datatype is contained within the *Stomp.Frames* module of the *tehstomp-lib* library package (see *Appendix A*). Additionally, that module exports a plethora of utility functions for constructing, manipulating, and extracting information from *Frames*, including straightforward constructions for all of the commands with their required headers. Utilizing this interface it is simple for clients of the library to construct any possible STOMP frame.

B. *Stomp.Frames.IO*

Another module provided by the *tehstomp-lib* package is called *Stomp.Frames.IO*. This module encapsulates error-handling IO operations on Handles that are expected to be receiving STOMP frames. In most cases, this will be a *Handle* to a TCP socket connection in a STOMP client or broker, although any producer or consumer of STOMP frames could be handled using the library. The *FrameHandler* datatype utilizes transactional events to provide multiple asynchronous readers and writers with synchronous access to the underlying *Handle* as frames are read and written. To achieve this, the datatype houses two synchronous channels (*SChans*). Clients obtain a *FrameHandler* by using the *initFrameHandler* function. We obtain two synchronous channels, and then fork off looping threads that handle their processing (see Figure 5).

```
-- |Given a resource Handle, initialize a FrameHandler
-- and return it in an IO context.
initFrameHandler :: Handle -> IO FrameHandler
initFrameHandler handle = do
    writeChan <- sync newSChan
    readChan  <- sync newSChan
    forkIO $ writeLoop handle 0 writeChan
    forkIO $ readLoop handle readChan
    return $ FrameHandler handle writeChan readChan

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

parseFrame :: Handle -> IO FrameEvt

frameReaderLoop :: Handle -> SChan FrameEvt -> IO ()
frameReaderLoop handle readChan = do
    evt <- parseFrame handle
    sync $ sendEvt readChan evt
    case evt of
        NewFrame _ -> frameReaderLoop handle readChan
        Heartbeat  -> frameReaderLoop handle readChan
        -- Terminate the loop in the error case
        otherwise  -> return ()

-- |Get the next FrameEvt from the FrameHandler and
-- return it in an Evt context.
getEvt :: FrameHandler -> Evt FrameEvt
getEvt (FrameHandler _ _ readChannel _ _) =
    recvEvt readChannel
```

Fig. 5. The FrameHandler

The *frameReaderLoop* function shown above loops waiting for input on a handle (the *parseFrame* function), and then attempts to synchronize on its *SChan*. When a client synchronizes on the *getEvt* function, the synchronization occurs, the *FrameEvt* is returned in an *IO* context, and the next frame is parsed. The API takes a two-tiered approach. The bare *Evt* functions are exposed, along with functions that abstract the event synchronization away from the client (an analogous *get* function is provided as well). This offers some flexibility in the construction of applications using these events.

For readers that are expecting heart-beats, we offer an additional function that times out if no data is received on the *Handle*, shown in figure 6.

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

Fig. 6. Heart-beat Management for Receivers

The *timeOutEvt* is part of the extended *TxEvents* library [2]. It is an event that becomes available for synchronization after a given number of microseconds has transpired. In this particular case, the *timeOutEvt* is sequenced with an *alwaysEvt* that will always return *TimedOut*; note that the *timeOutEvt* simply returns the unit value, and this particular function needs to return a *TimeOutEvt*. This necessitates the usage of the *thenEvt* sequencing combinator as shown above. Without the availability of this combinator, returning default values or specifying alternative control flows for timeouts in event processing would require much more complex logic. As it is, we can specify the "next step" to take after a timeout simply and logically. When a client synchronizes on the *getEvtWithTimeOut* event, if we have to wait longer than the given time allotment to receive an event on the read channel, the second choice is taken, and *TimedOut* is returned to the client.

The *FrameHandler* abstraction also provides functionality for *automatically* generating heart-beats if so needed (see Figure 7). This allows the client of this library, be it a client or a server application, to simply "set and forget" once initial negotiations have transpired.

```
data SendEvt  = SendFrame Frame |
                UpdateHeartbeat Int |
                DoHeartbeat

-- |Puts the given Frame into the FrameHandler
-- in an Evt context.
putEvt :: Frame -> FrameHandler -> Evt ()
putEvt frame (FrameHandler _ writeChan _ _ _) =
    sendEvt writeChan $ SendFrame frame

-- |Update the rate at which this FrameHandler sends
-- heart-beats. A rate of 0 or less means that no
-- heart-beats will be transmitted. Otherwise, we
-- will send one heart-beat every n microseconds.
updateHeartbeat :: FrameHandler -> Int -> IO ()
updateHeartbeat (FrameHandler _ writeChan _ _ _) n =
    sync $ sendEvt writeChan (UpdateHeartbeat n)

frameWriterLoop :: Handle -> Int -> SChan SendEvt -> IO ()
frameWriterLoop handle freq writeChan = do
    update <- if freq < 1
        then sync $ recvEvt writeChan
        else sync $ (recvEvt writeChan) `chooseEvt`
            (timeOutEvt freq `thenEvt`
              (\_ -> alwaysEvt DoHeartbeat))
    case update of
        SendFrame frame -> do
            hPut handle $ frameToBytes frame
            frameWriterLoop handle freq writeChan
        UpdateHeartbeat freq' -> do
            hPut handle $ UTF.fromString "\n"
            frameWriterLoop handle freq' writeChan
        DoHeartbeat       -> do
            hPut handle $ UTF.fromString "\n"
            frameWriterLoop handle freq writeChan
```

Fig. 7. Heart-beat Management for Senders

Since the protocol dictates that the sender delivers either a frame or a heartbeat at least once per interval, and whether or not a sender is going to send a frame at some point during that interval is non-deterministic, we construct the event using the *chooseEvt* combinator. This event will either send a frame if one is provided within the interval, or send a heartbeat after said interval has transpired without receiving any frames to transmit. We also provide a means for updating the heartbeat after the loop has been initiated. The reasoning for this is that it is likely that a client or server will want to use the *FrameHandler* in setting up heart-beating and protocol negotiations, so it will need to be initialized prior to the final frequency determination. This allows for those initial negotiations to take place and the heart-beating mechanism to be kicked off after the fact if necessary.

C. *Stomp.Frames.IO.Router*

One can envision a client application in which multiple system components are interested in the different types of frames that are received from the server. As part of the *tehstomp-lib* library, we provide a client-centric (in that it would be not suitable for use in a server implementation) event management system, wrapping a *FrameHandler*, that routes frames to registered listeners. The exported API is given in Figure 8.

```
data RequestHandler = RequestHandler (SChan Update)

initFrameRouter ::
  FrameHandler ->
  IO RequestHandler

-- |Request a dedicated channel on which to receive
-- CONNECTED and RECEIPT Frames
requestResponseEvents ::
  RequestHandler ->
  IO (SChan FrameEvt)

-- |Request a dedicated channel on which to receive
-- MESSAGE frames for the given subscription ID
requestSubscriptionEvents ::
  RequestHandler ->
  String -> IO (SChan FrameEvt)

-- |Request a dedicated channel on which to receive
-- heart-beats
requestHeartbeatEvents ::
  RequestHandler ->
  IO (SChan FrameEvt)

-- |Request a dedicated channel on which to receive
-- ERROR frames
requestErrorEvents ::
  RequestHandler ->
  IO (SChan FrameEvt)
```

Fig. 8. The FrameRouter API

Once the *RequestHandler* has been initialized with a *FrameHandler*, interested system components can use it to obtain dedicated event channels. The *FrameRouter* maintains lists of those channels, and when a relevant *FrameEvt* is received, it is broadcast to all registered listeners. Clients can register for frames that are sent to individual subscriptions (MESSAGE), frames that are sent as responses (CONNECTED, RECEIPT), or ERROR frames. They can also request to receive heart-beats. Per-frame processing for each registered channel is handled on a separate thread, so that no individual registered component is able to hold up processing for any of the other registered components.

D. *Additional Modules*

There are a few additional modules exported by the library that do not have to do with STOMP directly, but are useful nonetheless and are mentioned here in brief. The *Stomp.TLogger* module implements a transactional "logger", similar but simpler in concept and design to that of the *FrameHandler*. The *Stomp.Increment* module implements a stateful incrementer that is used in generating unique message and subscription identifiers.

V. Application Development

Two applications were developed using transactional events, along with the *tehstomp-lib* libraries developed as part of this project: a basic command-line STOMP client, and a STOMP message broker.

A. The STOMP Broker

At the core of the STOMP broker is a looping function over an open socket. The function loops as connections are received, and forks off a new thread to negotiate each incoming connection request.

```
socketLoop ::
    Socket ->
    Logger ->
    SubscriptionManager ->
    Incrementer ->
    IO ()
socketLoop sock console subManager inc = do
    (uSock, _) <- accept sock
    addr <- getPeerName uSock
    log console $ "New connection received from "
        ++ (show addr)
    handle <- socketToHandle uSock ReadWriteMode
    hSetBuffering handle NoBuffering
    frameHandler <- initFrameHandler handle
    forkIO $ negotiateConnection frameHandler
        (addTransform (appendTransform
            ("[" ++ show addr ++ "]")) console)
                subManager inc
    socketLoop sock console subManager inc
```

Fig. 9. STOMP Broker Socket Loop

If protocol negotiations are successful, the handle remains open, and the thread loops waiting for frames and heartbeats from the client until the client disconnects. The connection data for a given client is encapsulated in the *Connection* datatype:

```
-- |Encapsulation of a single client's connection data
data Connection       = Connection
                          ClientId
                          FrameHandler
                          ClientTransactionManager
                          Int
```

Fig. 10. Client Connection Data

The *ClientId* is a unique identifier for the client that is to be used in the *SubscriptionManager*. The server has a single stateful *Incrementer* that it uses to generate unique client identifiers. The *FrameHandler* is constructed using the *Handle* obtained from the socket connection. The *ClientTransactionManager* is a dedicated transaction manager for that client, and the *Int* is the frequency with which heart-beats are expected to be received from the client. The *ClientTransactionManager* contains a stateful loop that is maintained per-client, as the rest of the system doesn't need to be aware of pending client transactions. The *ClientTransactionManager* will be discussed in more detail in the subsection dealing with the *Transaction* module.

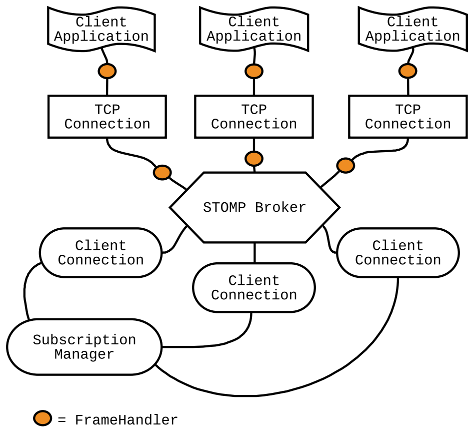


Fig. 11. STOMP Broker Architecture

The server itself maintains a separate stateful loop in the *SubscriptionManager*. The *SubscriptionManager* is a singleton that manages all of the subscriptions for all of the clients across that server. As clients subscribe to destinations, the *SubscriptionManager* is updated with that information.

B. The Subscriptions Module

The *Subscriptions* module exports a *SubscriptionManager* that the threads that manage client connections use to report updates to client subscriptions. Subscription handling is the most complex portion of the system. Initialization of the *SubscriptionManager* involves forking off two separate stateful loops: one to maintain the state of active client subscriptions, and one to maintain the state of message acknowledgements.

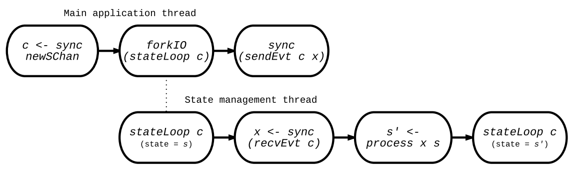


Fig. 12. State Management Workflow

The *Subscriptions* datatype houses the state that needs to be maintained in the *updateLoop* function. It contains a *SubMap*, which maps destinations to collections of client subscriptions. It also contains a datatype called *ClientDests*. This is not strictly necessary, and makes for a somewhat more complex implementation, but there is a desirable tradeoff in efficiency for unsubscribe operations. The *ClientDests* contains mappings of *ClientId* and *SubscriptionId* pairs to *Destinations*. This is to enable fast unsubscribe operations: *O(log(n))* compared to *O(d)*, where *n* is the number of clients and *d* is the number of active destinations. Using these two datastructures in conjunction, though somewhat complicated, precludes the

```
type ClientId           = Integer
type SubscriptionId     = String

type SubMap = HashMap
  Destination (HashMap ClientId ClientSub)

type ClientDests = HashMap
  ClientId (HashMap SubscriptionId Destination)

data Subscriptions = Subscriptions SubMap ClientDests

-- |Initialize a SubscriptionManager and return it in
-- an IO context.
initManager :: IO SubscriptionManager
initManager = do
    updateChan <- sync newSChan
    ackChan    <- sync newSChan
    inc        <- newIncrementer
    subs       <- return $ Subscriptions HM.empty HM.empty
    forkIO $ updateLoop updateChan ackChan subs inc
    forkIO $ ackLoop ackChan updateChan HM.empty
    return $ SubscriptionManager updateChan
```

Fig. 13. Forking State Loops in the SubscriptionManager

need to search through all destinations to see if they countain the given mapping.

The state of *Subscriptions* is managed by abstracting all subscription operations through event-generating functions that send updates on the *updateChan*. As SUBSCRIBE, UNSUBSCRIBE, and SEND frames are received, the client thread calls the appropriate update function and the *updateLoop* adjusts the state of its data accordingly as part of processing the update. This is a broadly useful pattern (see Figure 12) when developing concurrent applications, as most useful programs need some way of maintaining running state in various components. It is particularly useful when the events that affect state originate as a result of side effects, and as such the pattern is made exceptionally straightforward using transactional events.

```
-- |General Update type for main processing loop
data Update =
  Add Destination ClientSub |
  Remove ClientId SubscriptionId |
  GotMessage Destination Frame |
  ResendMessage Destination Frame [ClientId] MessageId |
  Ack ClientAckResponse |
  Disconnected ClientId

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

-- |Send a Frame to a Destination
-- (example of an exported update function)
sendMessageEvt ::
  SubscriptionManager ->
  Destination ->
  Frame ->
  Evt ()
sendMessageEvt
  manager@(SubscriptionManager updateChan)
  destination
  frame =
    sendEvt updateChan $ GotMessage destination frame
```

Fig. 14. State Management for Subscriptions

The code in Figure 14 is a slight simplification of the update code in the *Subscriptions* module (some parameters are left out for space considerations), but it is illustrative of how transactional events easily enable this pattern. It is easy to reason about and build such state management loops for any module that requires synchronous management of the state of its data. The entire operation of maintaining state is abstracted into three basic operations: (1) wait for an update, (2) process the update (3) return a modified state from the update processing. We have a loop that waits for updates on its synchronous channel. When one is received, it is processed by a function that returns a (potentially) modified subscription mapping. We then proceed to the next loop iteration with the updated state.

C. Client Selection in Message Processing

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. There are other possible implementation-specific behaviors (such as one-to-many broadcast) that are not strictly dictated by the protocol. One client produces a message and delivers it to a destination; some other client, acting as a consumer, subscribes to a destination, advertising its willingness to process messages sent to that destination. As there could potentially be multiple clients subscribed to a single destination, the broker needs some mechanism by which to choose the "most eligible" client for consumption of a given message.

We can imagine several possible approaches to this problem. We could choose a client at random, but we run the risk of that client being blocked on some other operation. We could implement a "round-robin" approach, where every client gets a turn, but we run into the same risk; the chosen client might be blocked. One could imagine that there exists a complicated suitability heuristic, but it is difficult to ascertain how that would read in code. It most certainly would not be straightforward, as it would necessitate logic for making an attempt, handling individual attempt failures, and canceling attempted sends that were in-progress. Additionally, we would somehow need to guarantee that we are able to cancel an in-progress send in a timely manner: we could very easily end up with a race condition in which two clients consume the message; such logic would need to be handled very carefully, and it is reasonably safe to assume that the resultant code would be difficult to reason about.

Transactional events can be leveraged to make this choice intuitive to implement using the non-deterministic *chooseEvt* combinator. Events can be constructed dynamically from a map containing client channels. When we synchronize on the event, the first client thread that is able to complete the transaction will do so, with all other events aborting without having sent the message to their respective clients thanks to the transactional guarantee of transactional events.

The code in Figure 15 demonstrates the dynamic event construction for client choice. There are a few implementation

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

partialClientChoiceEvt ::
  Frame ->
  MessageId ->
  [ClientId] ->
  ClientSub ->
  Evt (Maybe ClientSub) ->
  Evt (Maybe ClientSub)
partialClientChoiceEvt
  frame
  messageId
  sentClients
  sub@(ClientSub clientId _ _ frameHandler) =
    if clientId `elem` sentClients then (chooseEvt neverEvt)
    else let frame' = transformFrame frame messageId sub in
        chooseEvt $ (putEvt frame' frameHandler)
            `thenEvt` (\_ -> alwaysEvt $ Just sub)
```

Fig. 15. Nondeterministic Choice of Consumer

details to take note of here. First, notice that the event is constructed using a recursive fold over the map of client subscriptions. The base case in the fold is a *timeOutEvt* that returns *Nothing* after a half second's wait. If there are no clients available to synchronize on the message within a half second, we retry the send, as it is possible that new subscribers have been registered in that time. If *Nothing* is returned from this event, we simply fork off an update to the main processing loop and wait until subscribers are registered for the given destination. Also, notice the *sentClients* list; this list contains all those client identifiers of clients to whom the message was sent and subsequently NACK'd (disacknowledged). We do not want to continually send a message to a client that has already rejected the message, so we maintain a running list in the *AckContext* and use it here to avoid doing so.

D. The Transaction Module

The *Transaction* module exports a *ClientTransactionManager* to each client thread that is used to report updates to client transactions. It uses the same state management pattern described in the section on the *Subscriptions* module to maintain what could potentially be multiple ongoing transactions for a single client. When a BEGIN frame is received, a transaction is initiated. We do so by inserting a unit *alwaysEvt* into the mapping. As additional frames are received, the transactional event is constructed dynamically using the *thenEvt* combinator. This ensures that when we synchronize on the final event, all component events occur in sequence. This also ensures that all component events synchronize in sequence *atomically*, an important guarantee. They are not processed by the *SubscriptionManager* asynchronously; no other events are processed until all events in the transaction composite are processed sequentially. Such a composite event, in which two messages are sent as part of a transaction, would look similar to the code representation in Figure 16.

```
alwaysEvt ()
    `thenEvt` ((\_ -> sendMessageEvt subManager d1 f1)
        (`thenEvt` (\_ -> sendMessageEvt subManager d2 f2)))
```

Fig. 16. Composite Transactional Event For a STOMP Transaction

There is something subtle in the construction of this event that may not be immediately apparent. In order for this event to synchronize, the *SubscriptionManager* must support the ability to receive multiple updates in one synchronization. Recall the implementation of the *updateLoop* function in the section detailing the *SubscriptionManager*. Given that implementation, only one update is possible per event synchronization. As such, our event will compile, but will block indefinitely on synchronization, as the server will be waiting for the first update syncrhonization to complete prior to looping through and accepting the next update. The addition of transaction handling hence required an update to the *SubscriptionManager* to support such updates. Kehrt, Effinger-Dean, Schmitz, and Grossman identify this particular issue, and their solution pattern [4] is applied here successfully. We make the following change to the *updateLoop* function in the *Subscriptions* module (once again the code, in particular the number of parameters, is slightly simplified for purposes of demonstration):

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

-- |Looping event to handle the possiblility 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')
```

Fig. 17. Looping Transactional Event

Instead of synchronizing on a single update and then 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, returning an updated *Subscriptions* structure with each iteration. We achieve this through use of the sequencing monadic bind (recall that this uses the *thenEvt* combinator) and the *chooseEvt* combinator. The event must first receive an update on the channel, then handle the update, and then either receive another update or complete and return the updated *Subscriptions*. One might, upon first glance, wonder whether this functions as we would expect due to the non-deterministic nature of the *chooseEvt* combinator.

However, 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 were a sequence of sends on the *updateChan*, the composite would not be able to complete if the *alwaysEvt* were chosen, so the semantics of transactional events dictates that the *updateEvtLoop* choice *must* be taken until there are no more sends that need to be matched, at which point the *alwaysEvt* will be chosen and the final result returned.

E. Client Application

The STOMP client implemented for this project is a basic command-line client. It is not particularly representative of "real-world" usage of a STOMP client, but is rather intended for interactive demonstration of the capbabilities of the message broker.

The approach to the client design is "event-driven". There are asynchronous loops handling input from the keyboard and input from various listeners on the *FrameRouter*. As events are received on these loops, they in turn send events to the main session loop, which synchronously handles displaying the event on the terminal for the user.

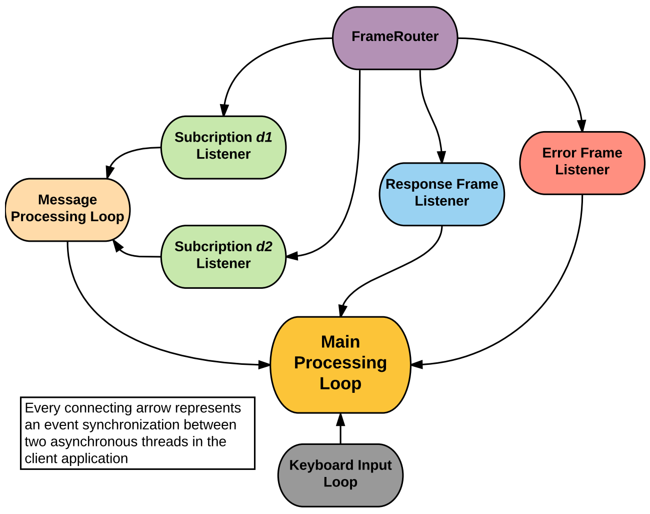


Fig. 18. Client Event Handling

The core code for the message processing loop is given in Figure 19. Note that the *subChan* is used to receive updates with respect to new subscriptions. The *subChoiceEvt* is a dynamically constructed selector that synchronizes on the first available subscription listener, or blocks if there are no new messages on any active subscription.

VI. Conclusion

We have shown that a fairly large and complex concurrent software project can be successfully built using transactional events as a foundation for inter-thread communications and library design. Additionally, we have used transactional events to their full effect in leveraging their guarantee of atomicity when committing client transactions in our message broker

```
subscriptionLoop ::
  SChan SubscriptionUpdate ->
  Subscriptions ->
  TLog.Logger ->
  IO ()
subscriptionLoop subChan subs = do
    update <- sync $ (recvEvt subChan)
        `chooseEvt` (subChoiceEvt subs)
    subs'  <- handleSubUpdate update subs
    subscriptionLoop subChan subs'

subChoiceEvt :: Subscriptions -> Evt SubscriptionUpdate
subChoiceEvt subs = HM.foldr partialSubChoiceEvt neverEvt subs

partialSubChoiceEvt ::
  Subscription ->
  Evt SubscriptionUpdate ->
   Evt SubscriptionUpdate
partialSubChoiceEvt (Subscription _ _ evtChan ackType) =
    chooseEvt $ (recvEvt evtChan) `thenEvt`
        (\frameEvt -> alwaysEvt (Received frameEvt ackType))
```

Fig. 19. Handling Subscription Events in the Client

application. We have also identified many additional useful patterns that stem from using a message-passing abstraction. Fluet and Donnelly state that the abstraction provides greater modularity and eases reasoning about concurrent programs [2]. While that may indeed be subjective and difficult to quantify, we believe that the work accomplished here validates that claim. Indeed, it is fairly clear that the solutions given here would have been extremely complex and bug-prone, if not outright impossible, were they to have been implemented given some other concurrency model.

VII. Related Work

To date there has been relatively little work around transactional events aside from the foundational work done by Fluet and Donnelly in 2008 [2]. Fluet and Amsden extend this foundation with a refined semantics for "fairness" in 2011 [5]. A detailed account of a large-scale implementation utilizing the abstraction does not appear to exist in the literature. In addition to Haskell, the semantics outlined in Fluet and Donnelly's 2008 work have been implemented in ML [3]. Kehrt, Effinger-Dean, Schmitz, and Grossman identify some basic problems that arise from some simple use cases for transactional events, and propose idiomatic approaches to these issues in their work "Programming Idioms for Transactional Events" [4].

References

[1] "Stomp 1.2 specification." [Online]. Available: https://stomp.github.io

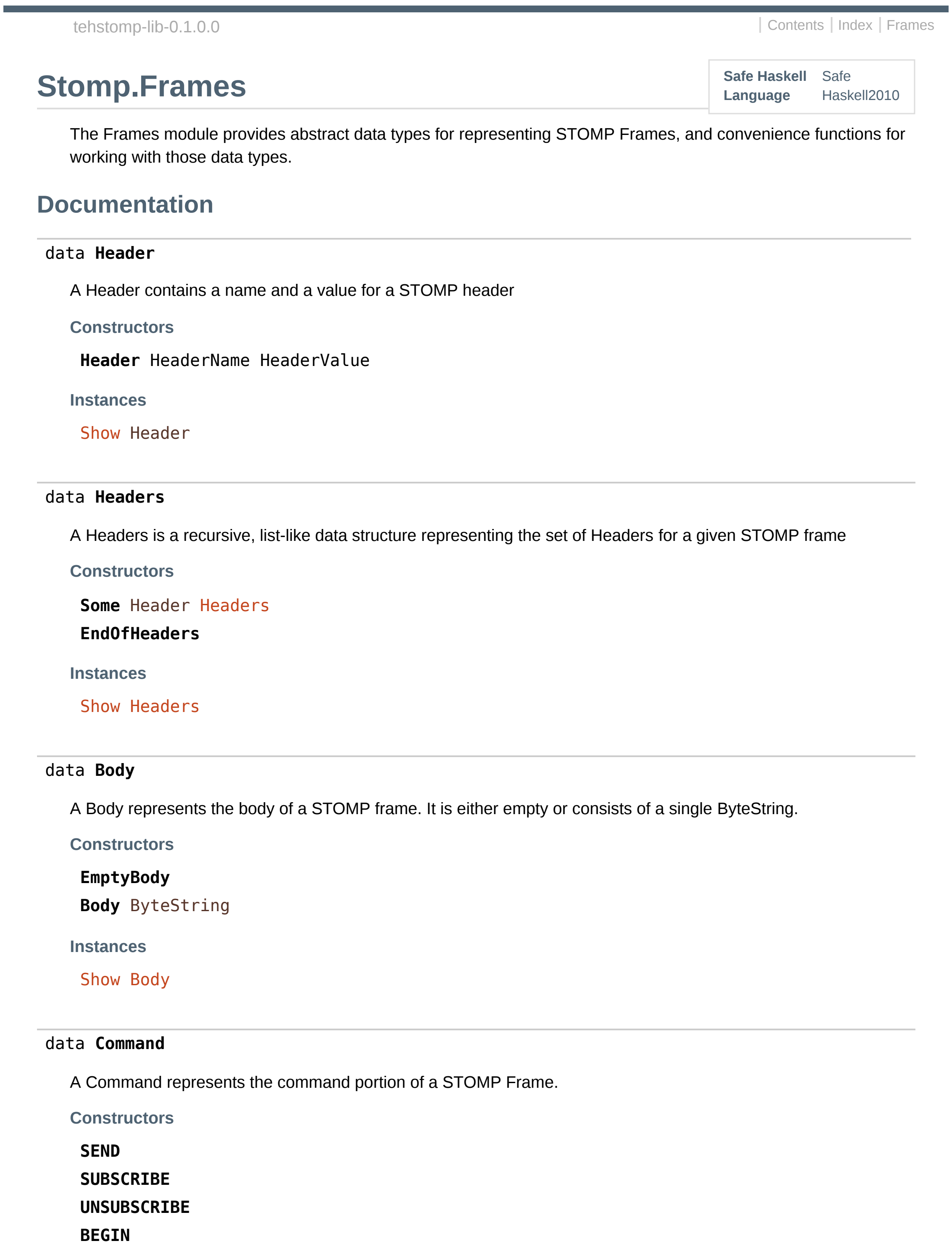
[2] K. Donnelly and M. Fluet, "Transactional events," *SIGPLAN Not.*, vol. 41, no. 9, pp. 124–135, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1160074.1159821

[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

[4] M. Kehrt, L. Effinger-Dean, M. Schmitz, and D. Grossman, "Programming idioms for transactional events," *arXiv preprint arXiv:1002.0936*, 2010.

[5] E. Amsden and M. Fluet, "Fairness for transactional events," in *Proceedings of the 23rd International Conference on Implementation and Application of Functional Languages*, ser. IFL'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 17–34. [Online]. Available: http://dx.doi.org.ezproxy.rit.edu/10.1007/978-3-642-34407-7_2

APPENDIX A
HADDOCK DOCUMENTATION FOR TEHSTOMP-LIB



tehstomp-lib-0.1.0.0 | Contents | Index | Frames

Stomp.Frames

Safe Haskell	Safe
Language	Haskell2010

The Frames module provides abstract data types for representing STOMP Frames, and convenience functions for working with those data types.

Documentation

`data` **`Header`**

A Header contains a name and a value for a STOMP header

Constructors

```
Header HeaderName HeaderValue
```

Instances

```
Show Header
```

`data` **`Headers`**

A Headers is a recursive, list-like data structure representing the set of Headers for a given STOMP frame

Constructors

```
Some Header Headers
EndOfHeaders
```

Instances

```
Show Headers
```

`data` **`Body`**

A Body represents the body of a STOMP frame. It is either empty or consists of a single ByteString.

Constructors

```
EmptyBody
Body ByteString
```

Instances

```
Show Body
```

`data` **`Command`**

A Command represents the command portion of a STOMP Frame.

Constructors

```
SEND
SUBSCRIBE
UNSUBSCRIBE
BEGIN
```

COMMIT
ABORT
ACK
NACK
DISCONNECT
CONNECT
STOMP
CONNECTED
MESSAGE
RECEIPT
ERROR

Instances

`Show Command`

`data` **`Frame`**

An abstract data type representing a STOMP Frame. It consists of a Command, Headers, and Body.

Constructors

`Frame Command Headers Body`

Instances

`Show Frame`

`data` **`AckType`**

Data type representing the possible acknowledgement types for a subscription to a STOMP broker.

Constructors

Auto
Client
ClientIndividual

Instances

`Show AckType`

`abort :: String -> Frame`

Generate an ABORT Frame given a transaction identifier as a String.

`ack :: String -> Frame`

Generate an ACK Frame given a message identifier as a String.

`ackHeader :: String -> Header`

Given a String, generate an ack header.

`addFrameHeaderFront :: Header -> Frame -> Frame`

Given a Frame, add a Header to the front of its Headers.

`addFrameHeaderEnd :: Header -> Frame -> Frame`

Given a Frame, add a Header to the end of its Headers.

`addHeaderEnd :: Header -> Headers -> Headers`

Add a Header to the end of the Headers

`addHeaderFront :: Header -> Headers -> Headers`

Add a Header to the front of the Headers

`addReceiptHeader :: String -> Frame -> Frame`

Add a receipt Header to the Frame.

`begin :: String -> Frame`

Generate a BEGIN Frame given a transaction identifier as a String.

`commit :: String -> Frame`

Generate a COMMIT Frame given a transaction identifier as a String.

`connect :: String -> Int -> Int -> Frame`

Generate a CONNECT Frame given a host identifier as a String.

`connected :: String -> Int -> Int -> Frame`

Generate a CONNECTED Frame given a version identifier as a String.

`disconnect :: String -> Frame`

Generate a DISCONNECT Frame given a receipt identifier as a String.

`errorFrame :: String -> Frame`

Generate an ERROR Frame given an error message as a String.

`getAckType :: Frame -> Maybe AckType`

Given a Frame, get the AckType if it is present

`getBody :: Frame -> Body`

Convenience function for retrieving the Body portion of a Frame.

```
getCommand :: Frame -> Command
```

Convenience function for retrieiving the Command portion of a Frame.

```
getContentLength :: Headers -> Maybe Int
```

Given Headers, get the value of the content-length Header if it is present.

```
getDestination :: Frame -> Maybe String
```

Given a Frame, get the value of the destination header if it is present.

```
getHeaders :: Frame -> Headers
```

Convenience function for retrieving the Headers portion of a Frame.

```
getHeartbeat :: Frame -> (Int, Int)
```

Given a Frame, get the (x, y) values in its 'heart-beat' header. If the header is not present, or the values are malformed, returns (0, 0).

```
getId :: Frame -> Maybe String
```

Given a Frame, get the value of the id header if it is present.

```
getReceipt :: Frame -> Maybe String
```

Given a Frame, get the value of the receipt header if it is present.

```
getReceiptId :: Frame -> Maybe String
```

Given a Frame, get the value of the receipt-id header if it is present.

```
getSupportedVersions :: Frame -> Maybe [String]
```

Given a Frame, get a list of supported STOMP versions, provided that the accept-version Header is present and well-formed.

```
getTransaction :: Frame -> Maybe String
```

Given a Frame, get the value of the transaction header if it is present.

```
getValueForHeader :: String -> Headers -> Maybe String
```

Given a header name and a Frame, get the value for that header if it is present.

```
idHeader :: String -> Header
```

Given an ID as a String, create an id Header.

```
makeHeaders :: [Header] -> Headers
```

Given a list of Header datatypes, return a Headers datatype.

```
messageIdHeader :: String -> Header
```

Given a message identifier as a String, generate a message-id Header.

```
nack :: String -> Frame
```

Generate a NACK Frame given a message identifier as a String.

```
receipt :: String -> Frame
```

Generate a RECEIPT Frame given a receipt identifier as a String.

```
sendText :: String -> String -> Frame
```

Generate a plain text SEND Frame given a message as a String and a destination as a String.

```
subscribe :: String -> String -> AckType -> Frame
```

Generate a SUBSCRIBE Frame given a subscription identifer and destination as Strings, an

```
subscriptionHeader :: String -> Header
```

Given a subscription identifier as a String, generate a subscription Header.

```
txHeader :: String -> Header
```

Given a transaction identifier as a String, generate a transaction Header.

```
unsubscribe :: String -> Frame
```

Generate an UNSUBSCRIBE Frame given a subscription identifier.

```
_getDestination :: Frame -> String
```

Given a Frame, get the value of the destination header. If it is not present, throw an error.

```
_getAck :: Frame -> String
```

Given a Frame, get the value of the ack header; throws an error if it is not present.

```
_getId :: Frame -> String
```

Given a Frame, get the value of the id header. If it is not present, throws an error.

Produced by Haddock version 2.16.1

tehstomp-lib-0.1.0.0 | Contents | Index | Frames

Stomp.Frames.IO

Safe Haskell	None
Language	Haskell2010

The IO module of the Frames package encapsulates error-handling IO operations on Handles that are expected to be receiving STOMP frames.

Documentation

```
data FrameHandler
```

A FrameHandler encapsulates the work of sending and receiving frames on a Handle. In most cases, this will be a Handle to a TCP socket connection in a STOMP client or broker.

```
data FrameEvt
```

A FrameEvt is a type of event that can be received from a FrameHandler (see the get function).

Constructors

```
NewFrame Frame
ParseError String
Heartbeat
GotEof
TimedOut
```

Instances

```
Show FrameEvt
```

```
initFrameHandler :: Handle -> IO FrameHandler
```

Given a resource Handle to which STOMP frames will be read from and written to, initializes a FrameHandler and returns it to the caller.

```
put :: FrameHandler -> Frame -> IO ()
```

Puts the given Frame into the given FrameHandler in an IO context. This function will block until the Frame has been processed.

```
putEvt :: Frame -> FrameHandler -> Evt ()
```

Puts the given Frame into the FrameHandler in an Evt context.

```
get :: FrameHandler -> IO FrameEvt
```

Get the next FrameEvt from the given FrameHandler and return it in an IO context. This function will block until a FrameEvt is available.

```
getEvt :: FrameHandler -> Evt FrameEvt
```

Get the next FrameEvt from the given FrameHandler and return it in an Evt context.

`getEvtWithTimeOut` `:: FrameHandler -> Int -> Evt FrameEvt`

Get the next FrameEvt from the given FrameHandler and return it an Evt context. If the given timeout (in microseconds) is exceeded prior to receiving activity on the channel, this will return TimedOut.

`close` `:: FrameHandler -> IO ()`

Kills all threads associated with the FrameHandler.

`frameToBytes` `:: Frame -> ByteString`

Convert a Frame to a STOMP protocol adherent ByteString suitable for transmission over a handle.

`updateHeartbeat` `:: FrameHandler -> Int -> IO ()`

Update the rate at which this FrameHandler sends heart-beats. A rate of 0 or less means that no heart-beats will be transmitted. Otherwise, we will send one heart-beat every n microseconds.

Produced by Haddock version 2.16.1

tehstomp-lib-0.1.0.0 | Contents | Index | Frames

Stomp.Frames.Router

Safe Haskell	None
Language	Haskell2010

The Router module implements an asynchronous notification system for events received on a FrameHandle. Callers can request various types of events and receive those events on a dedicated communications channel.

Documentation

```
initFrameRouter :: FrameHandler -> IO RequestHandler
```

Given a FrameHandler on which FramesEvts are being received, initalize a FrameRouter and return a RequestHandler for that FrameRouter to the caller.

```
requestResponseEvents :: RequestHandler -> IO (SChan FrameEvt)
```

Given a RequestHandler, request a new SChan on which to receive response events. Response events are defined as any Frame received other than a MESSAGE Frame.

```
requestSubscriptionEvents :: RequestHandler -> String -> IO (SChan FrameEvt)
```

Given a RequestHandler, request a new SChan on which to receive subscription events for a given subscription identifier.

```
requestHeartbeatEvents :: RequestHandler -> IO (SChan FrameEvt)
```

Given a RequestHandler, request a new SChan on which to receive heart-beats

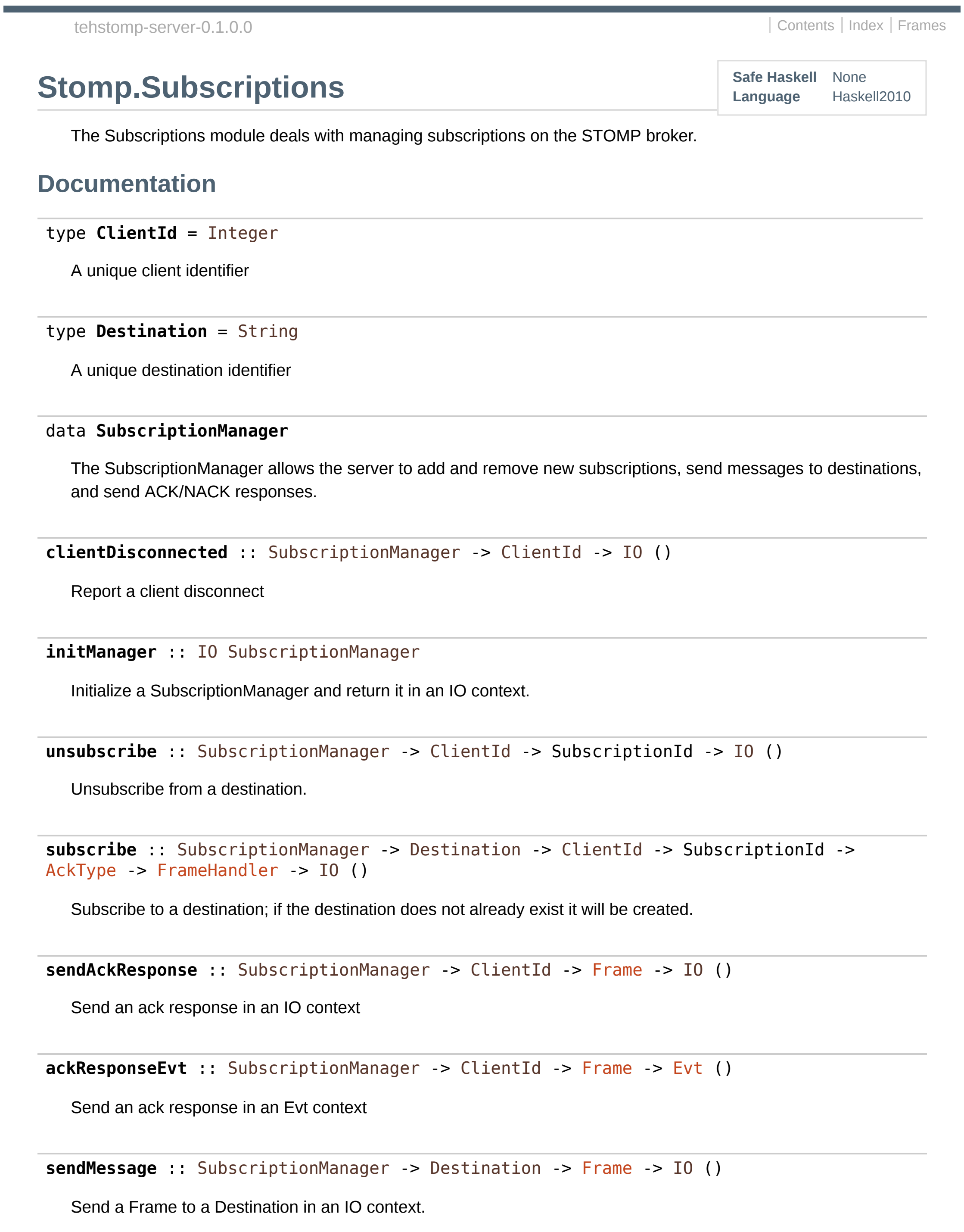
```
requestErrorEvents :: RequestHandler -> IO (SChan FrameEvt)
```

Given a RequestHandler, request a new SChan on which to receive ERROR frames

```
data RequestHandler
```

The RequestHandler is used to request various types of notifications from the FrameRouter.

Produced by Haddock version 2.16.1



tehstomp-server-0.1.0.0 | Contents | Index | Frames

Stomp.Subscriptions

Safe Haskell	None
Language	Haskell2010

The Subscriptions module deals with managing subscriptions on the STOMP broker.

Documentation

```
type ClientId = Integer
```

A unique client identifier

```
type Destination = String
```

A unique destination identifier

```
data SubscriptionManager
```

The SubscriptionManager allows the server to add and remove new subscriptions, send messages to destinations, and send ACK/NACK responses.

```
clientDisconnected :: SubscriptionManager -> ClientId -> IO ()
```

Report a client disconnect

```
initManager :: IO SubscriptionManager
```

Initialize a SubscriptionManager and return it in an IO context.

```
unsubscribe :: SubscriptionManager -> ClientId -> SubscriptionId -> IO ()
```

Unsubscribe from a destination.

```
subscribe :: SubscriptionManager -> Destination -> ClientId -> SubscriptionId ->
AckType -> FrameHandler -> IO ()
```

Subscribe to a destination; if the destination does not already exist it will be created.

```
sendAckResponse :: SubscriptionManager -> ClientId -> Frame -> IO ()
```

Send an ack response in an IO context

```
ackResponseEvt :: SubscriptionManager -> ClientId -> Frame -> Evt ()
```

Send an ack response in an Evt context

```
sendMessage :: SubscriptionManager -> Destination -> Frame -> IO ()
```

Send a Frame to a Destination in an IO context.

```
sendMessageEvt :: SubscriptionManager -> Destination -> Frame -> Evt ()
```

Send a Frame to a Destination in an Evt context.

Produced by Haddock version 2.16.1

tehstomp-server-0.1.0.0 | Contents | Index | Frames

Stomp.Transaction

Safe Haskell	None
Language	Haskell2010

The Stomp.Transaction module implements a ClientTransactionManager that can be used, in conjunction with a SubscriptionManager, to handle STOMP transactiosn for a single client.

Documentation

```
data ClientTransactionManager
```

Encapuslates the communications channels for a client transaction manager.

```
data UpdateResponse
```

An update will either be successful or generate an error message.

Constructors

```
Success
Error String
```

```
type TransactionId = String
```

A TransactionId is a unique (per client) transaction identifier

```
initTransactionManager :: SubscriptionManager -> IO ClientTransactionManager
```

Initialize a ClientTransactionManager and return it in an IO context.

```
begin :: TransactionId -> ClientTransactionManager -> IO UpdateResponse
```

Begin a Transaction with the given TransactionId

```
commit :: TransactionId -> ClientTransactionManager -> IO UpdateResponse
```

Commit the Transaction with the given TransactionId

```
abort :: TransactionId -> ClientTransactionManager -> IO UpdateResponse
```

Abort the Transaction with the given TransactionId

```
ackResponse :: TransactionId -> ClientId -> Frame -> ClientTransactionManager -> IO
UpdateResponse
```

Add an AckResponse to the Transaction with the given TransactionId

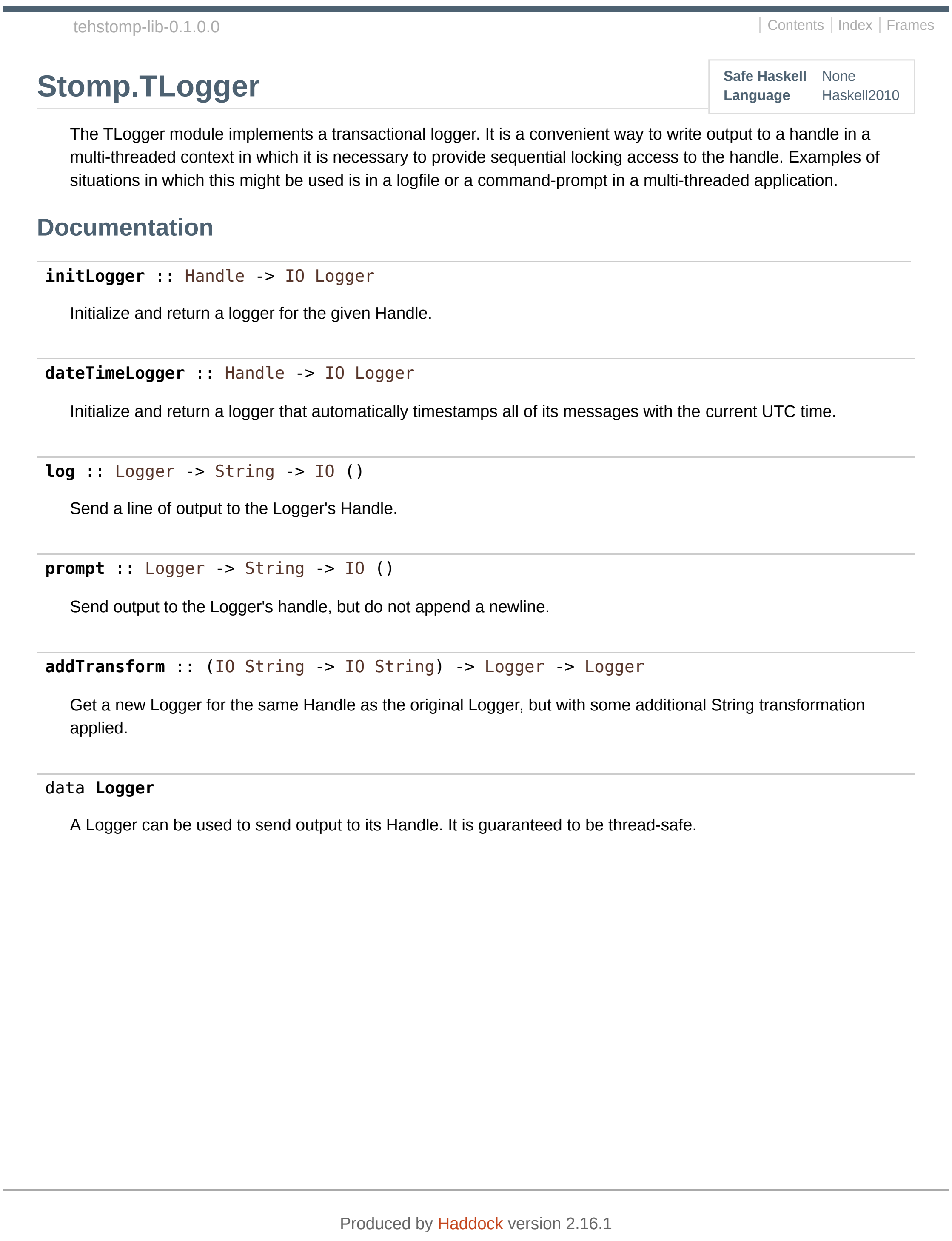
```
send :: TransactionId -> Destination -> Frame -> ClientTransactionManager -> IO
UpdateResponse
```

Add a SEND frame to the Transaction with the given TransactionId

```
disconnect :: ClientTransactionManager -> IO UpdateResponse
```

Send a "disconnect" notice; all pending transactions will be aborted. The ClientTransactionManager should not be used after calling this function.

Produced by Haddock version 2.16.1



tehstomp-lib-0.1.0.0 | Contents | Index | Frames

Stomp.TLogger

Safe Haskell	None
Language	Haskell2010

The TLogger module implements a transactional logger. It is a convenient way to write output to a handle in a multi-threaded context in which it is necessary to provide sequential locking access to the handle. Examples of situations in which this might be used is in a logfile or a command-prompt in a multi-threaded application.

Documentation

```
initLogger :: Handle -> IO Logger
```

Initialize and return a logger for the given Handle.

```
dateTimeLogger :: Handle -> IO Logger
```

Initialize and return a logger that automatically timestamps all of its messages with the current UTC time.

```
log :: Logger -> String -> IO ()
```

Send a line of output to the Logger's Handle.

```
prompt :: Logger -> String -> IO ()
```

Send output to the Logger's handle, but do not append a newline.

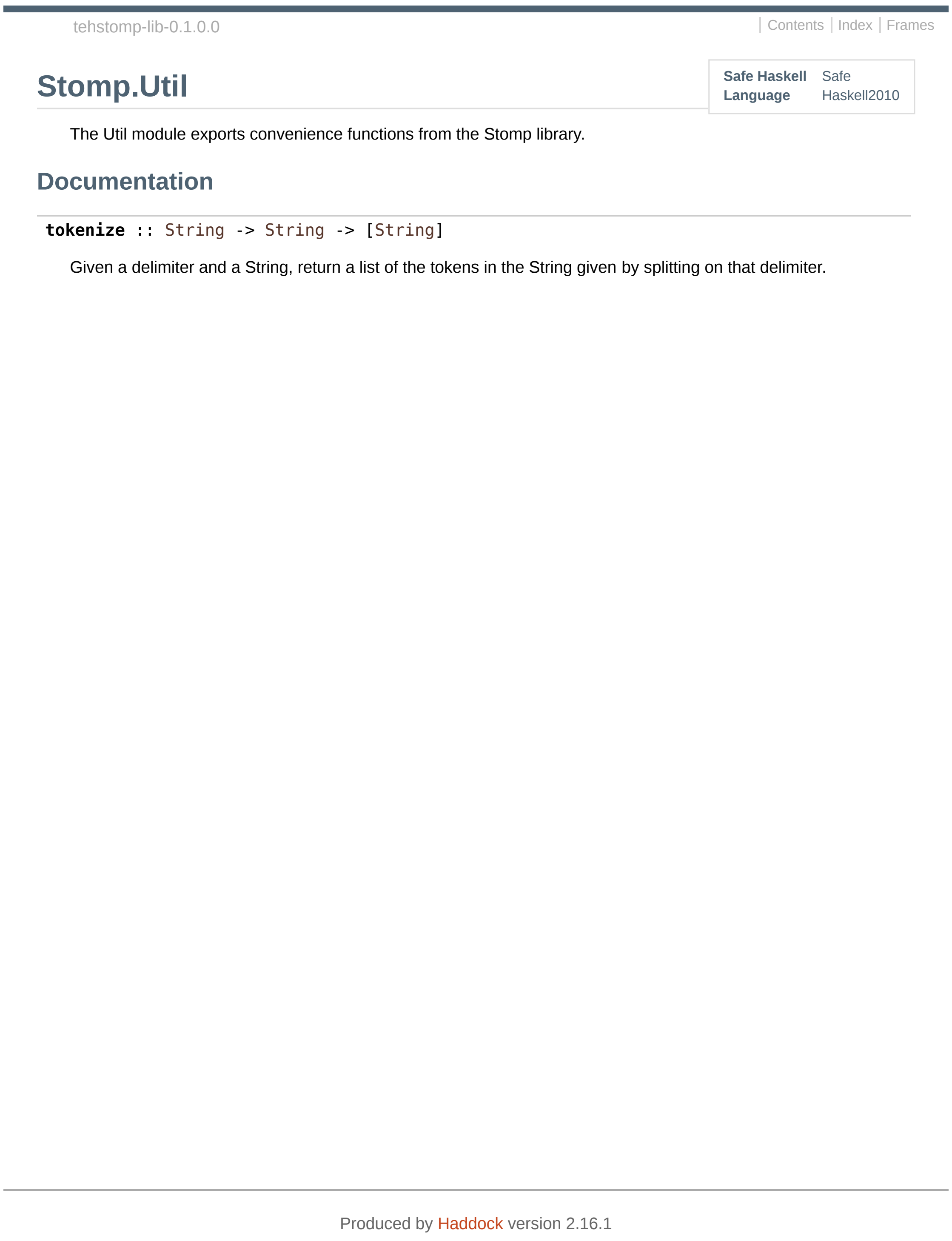
```
addTransform :: (IO String -> IO String) -> Logger -> Logger
```

Get a new Logger for the same Handle as the original Logger, but with some additional String transformation applied.

```
data Logger
```

A Logger can be used to send output to its Handle. It is guaranteed to be thread-safe.

Produced by Haddock version 2.16.1



tehstomp-lib-0.1.0.0 | Contents | Index | Frames

Stomp.Util

Safe Haskell	Safe
Language	Haskell2010

The Util module exports convenience functions from the Stomp library.

Documentation

```
tokenize :: String -> String -> [String]
```

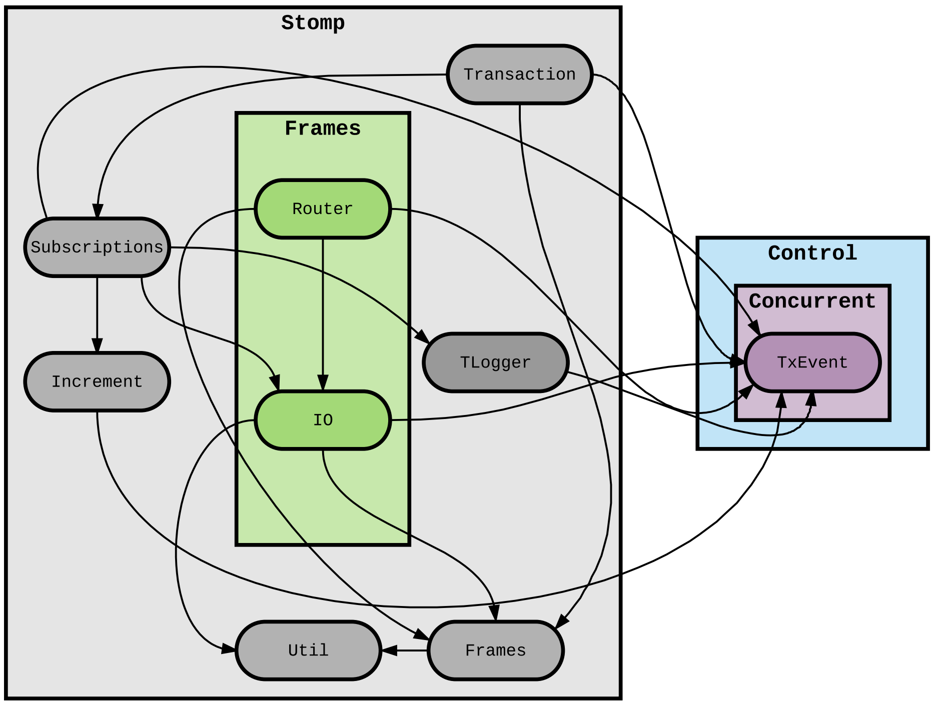
Given a delimiter and a String, return a list of the tokens in the String given by splitting on that delimiter.

Produced by Haddock version 2.16.1

APPENDIX B
MODULE DEPENDENCIES FOR TEHSTOMP-LIB

Modules of tehstomp-lib

- **Stomp.Frames** defines the *Frame* datatype and includes functions for the construction and manipulation of STOMP frames.
- **Stomp.Transaction** defines the transactionally stateful *ClientTransactionManager* datatype and functions for interacting with it.
- **Stomp.Increment** defines a transactionally stateful incrementer.
- **Stomp.Subscriptions** defines a transactionally stateful *SubscriptionManager* datatype and functions for interacting with it.
- **Stomp.TLogger** defines a transactional logger.
- **Stomp.Util** contains convenience functions for ease of sharing.
- **Stomp.Frames.Router** defines a client-centric event-handling system for routing frames to interested system components.
- **Stomp.Frames.IO** defines the *FrameHandler* datatype and implements stream-based parsing and IO operations for STOMP frames.


Stomp
Transaction
Frames
Router
Subscriptions
TLogger
Increment
IO
Util
Frames
Control
Concurrent
TxEvent