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

Week 8 Milestone Report
Christopher R. Wicks
10-21-17

Introduction

My master's project explores the use of Transactional Events, a novel concurrency abstraction in the functional paradigm, in the implementation of message broker middleware. Specifically, client and server software for the STOMP protocol are being implemented using the *tx-events* library for Haskell. The performance and convenience of the abstraction are meant to be evaluated in a "real-world" implementation scenario. This brief report outlines the work that has been done thus far for my master's project, and the work to be done over the course of the rest of the semester.

Work done to date

So far, several pieces of software have been implemented to achieve this end. The *tehstomp-lib* library contains code that can be shared between client and server applications: a library for generating and manipulating STOMP frames, an IO library for reading and parsing STOMP frames directly from a resource handle (e.g. a handle to a TCP connection), a thread-safe logger for generating output in multi-threaded applications, and an event handling system to route STOMP frames and other events to disparate components of a multi-threaded application. The *tehstomp-client* application implements a command-line client for connecting to and testing the functionality of a STOMP broker. The *tehstomp-server* application implements a STOMP broker for managing connected clients and their subscriptions.

The client and server functionality, at this point, do not yet implement the protocol completely. This functionality is being added iteratively to both the client and server components. The *STOMP/CONNECT* frames (client-side) and the *CONNECTED* frame (server-side) allow for initial protocol negotiations between the client and the server. The *ERROR* frame (server-side) allows the server to signal an error to the client and close the connection. The *SUBSCRIBE* frame enables the client to subscribe to messages on a given destination. The *SEND* enables the client to send messages to a given destination. The *RECEIPT* frame allows the server to respond to the client with an acknowledgement of a frame if so requested. A client may request a receipt for any frame (except for the *CONNECT* frame) by adding a special header to the frame. The *MESSAGE* frame allows the server to send a message to a subscriber to a given destination.

The work ahead

The behavior of the broker with respect to subscriptions is not dictated by the STOMP protocol [1]. In the current implementation, all of the clients subscribed to a destination receive all of the messages sent to that destination. The next step is to implement a queuing system so that the default behavior is that of clients acting as producers and consumers; that is, the default behavior will be that only one interested client need consume a message by default. This will involve implementing specific behavior for the *ACK* and *NACK* client frames, and the headers for the different *ack* types supported by the protocol. These frames are responses to the server indicating whether or not a client has consumed a frame that was sent by the server. If the client has not (*NACK*) consumed a frame, the server should send it to a different subscriber. In such a system, we may need an algorithm to determine which client, out of many potential subscribers, is chosen to consume a given message. This is a particular case in which transactional events could prove useful in simplifying our application logic. Instead of implementing (potentially complex) logic to choose the most suitable client to receive the message (or just choosing randomly), we can leverage the non-determinism of the *chooseEvt* combinator. In such a scenario, the first client that is able to synchronize on the event is the one chosen. This provides a nice, clean way of abstracting away what could otherwise be very complex code, and always provides us with the "best" choice: one of the less active client threads will be chosen in most scenarios, as they will be able to synchronize on the event more quickly.

A basic example of how this might be achieved in code follows:

```
constructEvt :: [SChan Frame] -> Frame -> Evt ()
constructEvt [] frame = neverEvt
constructEvt (channel:channels) frame =
    (sendEvt channel frame) `chooseEvt` (constructEvt channels frame)

listen :: SChan FrameEvt -> [SChan Frame] -> IO ()
listen frameChannel listeners = do
    frameEvt <- sync $ recvEvt frameChannel
    case frameEvt of
        (NewFrame frame) -> sync $ constructEvt listeners frame
            ...
            ...
    listen frameChannel listeners
```

Note that the event is constructed dynamically and recursively, leveraging the fact that we can use the *neverEvt* in the base case of the recursion. Recall that *neverEvt* can never be synchronized on, and hence is a left or right identity for the *chooseEvt* combinator [2]. Another possible base case would be to utilize a *timeOutEvt* (part of the extended *tx-events* library) to timeout after a certain period if none of the events can synchronize. There will additionally be a server-specific header that a client may use to indicate that a given message should be broadcast to all subscribers to its destination.

Additional work to round out the protocol will involve implementing transactions using the *BEGIN*, *COMMIT*, and *ABORT* client frames. This would likely use a similar dynamic event construction on the server to that outlined above, except utilizing the *thenEvt* combinator. Finally, clients and servers are allowed to request periodic heartbeats to affirm the health of the underlying connection. This negotiation is done through special headers on the *CONNECT* and *CONNECTED* frames.

Other possible work could involve implementing additional client applications to test certain scenarios. This opportunity could be taken to hone the API libraries, perhaps devising a subset or wrapper of the current API that is specific to clients.

References

[1] STOMP Specification. Retrieved from https://stomp.github.io/.
[2] Kevin Donnelly & Matthew Fluet (2006): Transactional Events.In: 11 th ACM International Conference on Functional Programming.