Pontarius XMPP 0.1 Manual (Third Draft)

The Pontarius Project

The 6th of July, 2011

Contents

1 Introduction

Pontarius XMPP aims to be a convenient-to-use, powerful, correct, secure, and extendable XMPP client library for Haskell. It is written by Jon Kristensen and Mahdi Abdinejadi. Being licensed under the GNU Lesser General Public License, Pontarius XMPP is free and open source software.

2 Features and Implementation Specifics

Pontarius XMPP 0.1 implements the client capabilities of the XMPP Core specification (RFC 6120)[1]. Below are the specifics of our implementation.

- The client is always the initiating entity
- A client-of-server connection is always exactly one TCP connection
- TLS is supported for client-to-server confidentiality
- Only the SCRAM authentication method is supported

[1] http://tools.ietf.org/html/rfc6120

- ...

Later versions will add supports for different XMPP extensions, such as RFC 6121 (XMPP IM), XEP-0004: Data Forms, and XEP-0077: In-Band Registration.[2]

3 Usage

Working with Pontarius XMPP is mostly done asynchronously; Pontarius XMPP "owns" the XMPP thread, and calls different StateT s m a callback functions in the client. StateT is a monad transformer which allows the functions to be stateful (being able to access and modify the arbitrary client-defined state of type s) and to be executed on top of a MonadIO m monad (typically IO).

3.1 Creating the session

Setting up an XMPP session is done through the (blocking) session function:

```
session :: (MonadIO m, ClientState s m) => s ->
        [ClientHandler s m] -> (StateT s m ()) -> m ()
```

The first parameter (of type s) is an arbitrary state that is defined by the client. This is the initial state, and it will be passed to the stateful client callbacks. It will typically be modified by the client.

The second parameter is the list of client handlers to deal with XMPP callbacks. The reason why we have a list is because we want to provide a "layered" system of XMPP event handlers. For example, XMPP client developers may want to have a dedicated handler to manage messages, implement a spam protection system, and so on. Messages are piped through these handlers one by one, and any handler may block the message from being sent to the next handler(s) above in the stack.

```
data MonadIO m => ClientHandler s m = ClientHandler {
        messageReceived :: Maybe (Message ->
        StateT s m Bool), presenceReceived :: Maybe
        (Presence -> StateT s m Bool), iqReceived ::
        Maybe (IQ -> StateT s m Bool),
        sessionTerminated :: Maybe (TerminationReason ->
        StateT s m ()) }
```

ClientHandler is a record which specifies four callback functions. The first three deals with the three XMPP stanzas, and are called once an XMPP stanza is received. These functions take the stanza in question, and are stateful with the current client state. The boolean value returned signals whether or not the message should be blocked to clients further down the stack. For example, a

[2] XMPP RFCs can be found at http://xmpp.org/xmpp-protocols/rfcs/, and the so-called XEPs at http://xmpp.org/xmpp-protocols/xmpp-extensions/.

XEP-0030: Service Discovery handler may choose to hide disco#info requests handlers above it in the stack. The last function is the callback that is used when the XMPP session is terminated. All callbacks are optional.

The third argument to session is a callback function that will be called when the session has been initialized.

Any function with access to the Session object can operate with the XMPP session, such as connecting the XMPP client or sending stanzas. More on this below.

3.2 Connecting the client

Different clients connect to XMPP in different ways. Some secure the stream with TLS, and some authenticate with the server. Pontarius XMPP provides a flexible function to help out with this in a convenient way:

```
connect :: MonadIO m => Session s m -> HostName ->
          PortNumber -> Maybe (Certificate, (Certificate ->
          Bool)) -> Maybe (UserName, Password, Maybe
          Resource) -> (ConnectResult -> StateT s m ()) ->
          StateT s m ()
```

This function simply takes the host name and port number to connect to, an optional tuple of the certificate to use and a function evaluating certificates for TLS (if Nothing is provided, the connection will not be TLS secured), and another optional tuple with user name, password, and an optional resource for authentication (analogously, providing Nothing here causes Pontarius XMPP not to authenticate). The final paramter is a callback function providing the result of the connect action.

For more fine-grained control of the connection, use the openStream, secureWithTLS, and authenticate functions.

3.3 Managing XMPP addresses

There are four functions dealing with XMPP addresses (or JIDs, as they are also called):

```
fromString :: String -> Maybe Address
fromStrings :: Maybe String -> String ->
          Maybe String -> Maybe Address
isBare :: Address -> Bool
isFull :: Address -> Bool
```

These functions should be pretty self-explainatory to those who know the XMPP: Core standard. The fromString functions takes one to three strings and tries to construct an XMPP address. isBare and isFull checks whether or not the bare is full (has a resource value).

3.4 Sending stanzas

Sending messages is done using this function:

```
sendMessage :: MonadIO m => Session s m -> Message ->
        Maybe (Message -> StateT s m Bool) ->
        Maybe (Timeout, StateT s m ()) ->
        Maybe (StreamError -> StateT s m ()) ->
        StateT s m ()
```

Like in section 3.2, the first parameter is the session object. The second is the message (check the Message record type in the API). The third parameter is an optional callback function to be executed if a reply to the message is received. The fourth parameter contains a Timeout (Integer) value, and a callback that Pontarius XMPP will call when a reply has not been received in the window of the timeout. The last parameter is an optional callback that is called if a stream error occurs.

Presence and IQ stanzas are sent in a very similar way.

Stanza IDs will be set for you if you leave them out. If, however, you want to know what ID you send, you can acquire a stanza ID by calling the getID function:

```
getID :: MonadIO m => Session s m -> StateT s m String
```

3.5 Concurrent usage

Sometimes clients will want to perform XMPP actions from more than one thread, or in other words, they want to perform actions from code that is not a Pontarius XMPP callback. For these use cases, use injectAction:

```
injectAction :: MonadIO m => Session s m ->
        Maybe (StateT s m Bool) -> StateT s m () ->
        StateT s m ()
```

The second parameter is an optional predicate callback to be executed right before the third parameter callback is called. If it is provided and evaluates to False, then the action will not be called. Otherwise, the action will be called.

3.6 Example echo server

We provide an example to further illustrate the Pontarius XMPP API and to make it easier for developers to get started with the library. The program illustrates how to connect, authenticate, set a presence, and echo all messages received. It only uses one client handler. The contents of this example may be used freely, as if it is in the public domain. You find it in the Examples directory of the Pontarius XMPP source code.