Micro-MQ

A Micro MQTT broker implemented in Erlang, following the given specification.

Lab of MCS Modern Concurrent Systems class at HEIG-VD, Switzerland, 2016, with prof. Marco Molteni.

Authors: Mélanie Huck, Jim Nolan, Valentin Minder, on our git repository.

Usage

Setup

This programs has two dependencies:

- `reloader` watches compiled sources and automatically reloads them in the erlang shell whenever they are changed
- `meck` is a mocking library for Erlang. (Note: do we really need it?)

To setup those dependencies, type from the server's directory:

- `rebar get-deps` (retrieve dependencies)
- `rebar update-deps` (update dependencies)

Simple compilation

With this method, you need to repeat steps 2 and 3 every time you edit the sources.

1. Launch your terminal in the server's directory
2. Compile sources: `rebar compile`
3. Run program: `erl -pa ebin`

Automatic reloading

With this method, whenever the sources are recompiled, the code is reloaded in the erlang shell using the `reloader` dependency.

1. Launch your terminal in the server's directory
2. Compile sources: `rebar compile`
3. Run program with the reloader (2 seconds refresh):
   `erl -pa ebin -pa deps/*/ebin -run reloader start 2`

API

`start_link`

Spawns the server. The server in its turn spawn an acceptor process that will listen for new tcp connexions. Whenever a new client connects, the acceptor process spawns a worker process.

It supports the following options:

```
micromq:start_link([
{max_clients, 10000},
{max_topics, 10000},
{address, localhost},
{port, 5017}
{verbosity, verbose}
])
```

Changing the port, the address and the verbosity works. However, max*clients and max*topics are not used.

`stop`

Retrieves the PID of the listener process and sends him a `stop` message.

Protocol

It is text-based protocol, with double line-return being the *commit* indicator. To connect type `telnet localhost 5017` or `telnet <host> <port>`. Upon connection client recieve their ID `client_id: <ID>`.

Welcome! I am a simple MQTT server. Allowed commands are: `topic:`, `subscribe:`, `status` and `help`. Simple line-return does nothing and allows multi-line commands and multi-line text body. Double line-return commits your command to the server. Warning: all commands headers are case-sensitive!

Status & Help

The `status` command replies with the current state of the client.

The `help` command display the documentation to the client.

Subscribe:

The `subscribe` command subscribes the clients to all the topics given, and confirms. From now on, the client will recieve all messages sent to this topic.

Format:

```
subscribe: <topic1> [ , <topic2>, ... ]
```

- Topics MUST be a on single line.
- `<topicX>` MUST not contain commas or line-return, but MAY contains spaces.\n">>
- There is a special topic `*`. Subscribing to this will cause all futher messages to be recieved.
- The client record is updated: the list of new topics subscribed is appended to the TopicSubscribed list.
- It is not precised in the specification what should happen if a client subscribes twice to the same topic. For simplicity we chooses to subscribe him twice.
- There is no way to unsubscribe, it is not asked in the specification. However, there is no persistence, and the client may disconnect to remove all the data.

Publish (topic)

The *publish* command allows to write a message (the `body`) on a `topic`, and confirms. The message will be forwarded to all clients subscribed to this `topic`, and to all clients subscribed to the special topic `*`.

Format:

```
topic: <topic name>
body: <msg body>
```

- This command MUST contain at least 2 lines, the first for `topic:`, the second for `body:`
- `<topic name>` MUST not contain commas or line-return, but MAY contains spaces.
- `<msg body>` MAY be any text and contain single line-return and commas.
- It will update the `NbMsgSent` and `TopicPublished` of the client record of the sender.
- It will update the `NbMsgRecieved` of the client record of all the recievers.

- It is not precised in the specification what should happen if a client sends a message on a topic he is subscribed to. For simplicity we choosed that he will recieve its own message in return as well.

- It is not precised in the specification what should happen if a client is subscribed to a specific topic `X` and the topic `*`, or if `X=*`. For simplicity, we choose he will recieve the message twice.

Tests

We built our program using a test driven approach. Our program pass all 80 tests. Our tests cover the following parts of our code:

```
 All 80 tests passed.
Cover analysis: /Users/val/erlang/erlang-fun/starfish/broker/.eunit/index.html

Code Coverage:
micromq                        :  95%
micromq_client_connect_tests   : 100%
micromq_client_protocol_tests  :  98%
micromq_server_start_tests     : 100%

Total                          : 97%
```

To allow a test to fail and close the server anyway, we used the following scheme, with start and stop taking care of the start-up and tear down.

```
dumb_test_() ->
    {setup,
    fun start/0,
    fun stop/1,
    fun dumb_tests/1}.
```

With test function returning an array of tests.

```
dumb_tests(_) ->
    [?_assertEqual(ok, ok)].
```

Our tests test the following aspects:

- start & stop the server in `micromq_server_start_tests.erl`
- connect & disconnet the client in `micromq_client_connect_tests.erl`
- test the protocol in `micromq_client_protocol_tests.erl`, including bad protocol usage and message framing.

Technical considerations

Processes and function

- **API**: API calls returns immediately. The `start` functions call the start-link internal. The `stop` sends the stop message to the server.
- **Start-link internal**: It is the internal start-up function and initializer of all shared internal structures. It **spawns the server** with the Network parameters and returns.
- **Server**: The server is the top-supervisor process, it **spawns the Acceptor** with the Network parameters and then wait for an Erlang message to handles the tear-down (and kill the Acceptor).
- **Acceptor (Listener)**: The Acceptor is an **infinite loop** that accepts new clients on a listening socket, until it is killed by the server. **Each new client spawns its own worker**, to be available for futher connecting clients
- **Worker**: The worker instantiate data specific to the client and calls the looper.
- **Looper**: The Looper is an **infinite loop** reading bytes on the client socket until it gets a complete message, calls the parser which will return the appropriate reply or a bad request message. The looper stops when the socket is closed, either by the client or the server.

Data persistance

To persist data further than a process life and share data among processes, we used ETS tables (see Erlang doc or see LYSE). We used them in 3 cases:

1. For **Client Records** in `clients_by_records`:
    - It stores the *current state* of the client (aka some statistics).
    - It is public (all processes may retrieve or write/update data)
    - It's a `set` (a client has a single record, any further write will overwrite previous records)
    - The format expected is:
      `{ClientID, Socket, [TopicSubscribed], [TopicPublished], NbMsgReceived, NbMsgSent}`
      of types
      `{integer(), inet:socket(), [binary()], [binary()], integer(), integer()}`
      where names obviously denotes their function.
    - A new record with default values is created when the connection is set up (when the listener get a new socket) and removed when the socket is closed by the client (when it's closed by the server every thing will be deleted anyway).
2. For **Client Subscriptions** in `clients_by_topic`:
    - It stores the subscription(s) of the client.
    - It is public (all processes may retrieve or write/update data).
    - It's a `bag` (a topic may obviously have many subscribers).
    - The format expected is: `{Topic, ClientID}` of types `{binary(), integer()}` where names obviously denotes their function.
3. For **Custum Options** in `options`:
    - It stores some of the server configuration (Network and Limits).
    - It is protected (only the launching process may edit it but all processes may access it).
    - It is a `set` (a property may exist only once!)
    - The format expected is: `{Property, Value}` of types `{atom(), term()}` where names obviously denotes their function.
    - Upon start-up, it is constructed and filled with default values from the macros (`?MACRO`), in `create_options()`.
    - Then, the custum start-up options overwrite the default entries in the `set`, in `handle_options()`.

Message framing

According to the protocol, messages from clients are framed by double line-breaks (our implementation supports both `\n` and `\r\n`

The challenge is that `tcp` sends chunks of text that may contain:

- A partial frame
- One or more complete frame(s), often followed by a partial frame

This is how we parse messages frames with tcp chunks:

1. Read tcp chunk
2. Find first occurrence of a double line break
3. If an occurrence was found, extract frame until the line-break and remove it from the chunk
4. If not, read another tcp chunk and concatenate it to the previous tcp chunk
5. repeat from `2.` with the updated tcp chunk

Bad protocol

The server should be designed not to crash when the client sends a message that doesn't follow the protocol's format. The specification does not tell how to respond to those messages. We choose to discard message that don't match the protocol in any case. To allow flexibility for this problem, we choose to have a macro parameters `?VERBOSITY` that may take values:

- `none` : quietly discard message that don't match the protocol.
- `minimal` : sends back a generic `400 Bad Request` .
- `verbose` : sends back an error-specific message to the client, to let him know what happened.