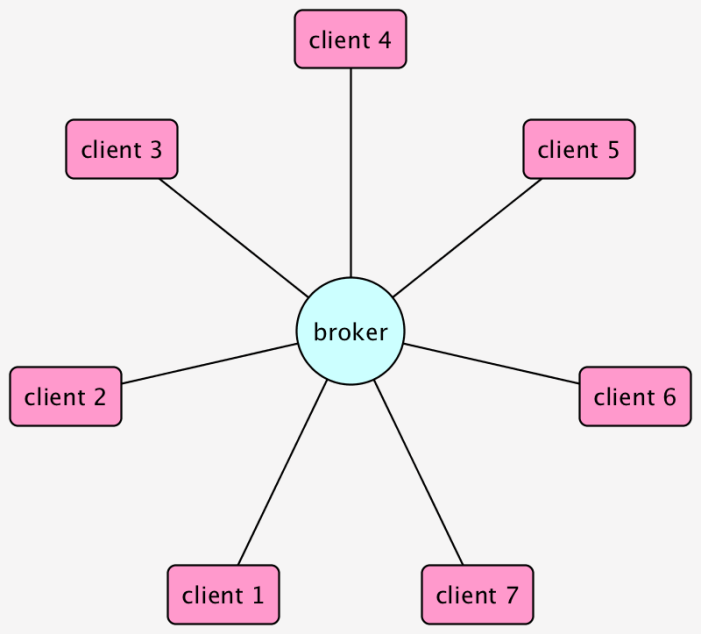
MicroMQ

MCS Lab 2016

Marco Molteni
Version 0.3, 2016-06-02

1. Introduction

Implement MicroMQ, a *minimal* pub/sub message broker in Erlang.



The clients can be simple `telnet` clients, while the server will be an Erlang program.

In any case, for testing, you will want to use an Erlang client :-)

2. Logistics

Small groups of 3 to 4 people.

3. Protocol

3.1. Connection

On connection, the broker will send to the client its ID as follows:

```
client_id: <client id>
```

(an unsigned integer)

3.2. Publish

To allow to use a `telnet` client, we will use a simple text-based protocol as follows:

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

where the message framing is expressed by two consecutive newlines (as in HTTP).

To which the broker will respond:

```
accepted: <topic name>
```

Topic creation is automatic: there is no need for the publisher to declare beforehand a topic: the first message sent on a topic (see example above) will also create the topic on the server side.

3.3. Subscribe

On the other hand, the subscriber will need to declare its interest explicitly:

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

to which the broker will respond:

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

If a client subscribes to a topic unknown to the broker, the broker will in any case accept the subscription and act upon it on the first message to the given topic.

3.4. Wildcard topic

There is a special topic, the wildcard: * . Subscribing to such topic will ask the broker to forward *all* messages received.

3.5. Forwarding

Upon reception of a message, the broker will forward it to all clients subscribed to the given topic (and to all the wildcard subscribers) as follows:

```
from: <client id>
topic: <topic name>
body: <msg body>
```

3.6. Status

When the client sends the command

```
status
```

the broker will respond with a summary of the status associated to the client:

```
status for client <client id>:
subscribed topics: <list of subscribed topics>
published topics: <list of published topics>
received messages: <number of received messages>
sent messages: <number of sent messages>
```

4. Configuration

The broker will have the following configuration parameters:

- TCP address (default: localhost only)
- TCP port (default: 5017)
- Maximum number of topics (default: 10'000)
- Maximum number of clients (default: 10'000)

5. Data structures

For the first time we are considering *big* data structures, and we also would like to survive a process crash.

Instead of keeping everything in the process state (probably a map or list in the `loop`), consider using an ETS table.

More information about ETS tables at:

- LYSE: ETS
- ETS reference documentation

6. Text manipulation

Remember that a socket should send and receive text as binary:

```
<<"topic: <topic name>\nbody: <msg body>\n\n">>
```

7. Documentation

Same short "rapport" as for PCAP and DRUM.

8. Build, test and run

The code must be buildable and eunit-testable with rebar, and must respect standard OTP directory layout: BEAM files go below `ebin/`, not below `src/`.

Compile without warnings:

```
$ rebar compile
```

Run tests without failures:

```
$ rebar eunit
```

Start Erlang (see also slide deck #2):

```
$ erl -pa ebin
```

The minimal rebar configuration *must* be (as the one in the MCS repo):

```
% Options for the Erlang compiler.
{erl_opts, [
    % DO NOT DISABLE THIS. If you get a warning, it means you are doing something wrong.
    warnings_as_errors
]}.

% Enable code coverage and generating the HTML details.
{cover_enabled, true}.

% Enable printing the coverage summary at the end of the build.
{cover_print_enabled, true}.
```

9. API

Please remember to respect the API.

Module containing the API: `micromq`.

You can then add as many implementation modules as you see fit.

API

Start the broker with default values:

```
-spec start_link() -> ok | {error, Reason}.
```

Start the broker with custom values:

```
-type option() ::
    {address, Address} |
    {port, Port} |
    {max_topics, N} |
    {max_clients, N}.

-spec start_link(Options) -> ok | {error, Reason} when
    Options :: [option()].
```

Stop the broker. Since it has arity 0, you will need a registered process:

```
-spec stop() -> ok.
```

10. Spawning and socket ownership

Depending how how you choose to spawn a process to server the socket, you might need to call `gen_tcp:controlling_process(Socket, Pid)`. Please refer to http://learnyousomeerlang.com/buckets-of-sockets and search for the text "controlling_process" for details.

11. Optional

- Implement the `system` topic (see below).
- Implement zeroconf autodiscovery of the broker.
- Implement message persistence (see below).

11.1. System topic

The broker offers a special topic, `system`. If a client sends a message to topic `system`, the broker will respond:

```
error: system is a reserved topic
```

Any client can subscribe to the system topic.

The broker will send, each 20 seconds (configurable), a message on the system topic as follows

```
from: broker
topic: system
body:
    system uptime: <system uptime in seconds>
    connected clients: <number of connected clients>
    subscribed topics: <list of subscribed topics>
    published topics: <list of published topics>
    received messages: <number of received messages>
    sent messages: <number of sent messages>
```

The period of the `system` topic is configurable by passing the following option to `start_link/2`:

```
{system_topic_period, N}
```

11.2. Message persistence

If a client, before disconnecting, sends the command

```
persist: <key>
```

to which the broker will respond

```
persisted: <key>
```

the server will associate `key` to the current `client id` and store all messages that match the client subscriptions.

When the client reconnects and sends the command

```
restore: <key>
```

the broker will replay to it all the stored messages.

Note: consider how to optimize the case when multiple clients send the `persist` command and are subscribed to the same topics: what do you do? Do you copy N times the same message in N persistence queues or can you do something better?

If message persistence is implemented, the `system` topic will have an additional information:

```
persisted clients: <N>
persisted messages: <M>
```