TSR - Lab 2: **0MQ**

2019/20

Contents

1. Introduction

1.1. Goals

- Consolidate the theoretical concepts introduced in Unit 3.
- Test different design patterns (basic communication patterns) and socket types.
- Delve into the broker-based design pattern (reverse proxy)

1.2. Scheduling proposal

- Session 1.- Round-robin publisher, chat application
- Session 2.- Configurable broker, statistics-reporting broker
- Session 3.- Split broker (broker for clients + broker for workers)
- Session 4.- Broker extensions, Fault-tolerant broker

1.3. Recommendations

- In order to simplify your tasks, start all application components in the same computer (using 'localhost' as the server IP address). Consider, however, that all provided programs could run deploying them in multiple computers.
- The port numbers shown in these examples should be modified according to the instructions given in the bulletin of Lab 1.
- All programs should be run in the labs. The examples in the bulletin use constant ports, hosts, messages, etc. However, in real programs, those values should be passed using command-line arguments.
- File `RefZMQ.pdf` provides additional reference contents that will help you in these tasks.
- File `fuentes.zip` provides all the programs mentioned in `RefZMQ.pdf`.

2. Tasks

2.1. Round/robin publisher

Develop a program `publisher.js`, using the `pub/sub` communication pattern (described in `RefZMQ.pdf`), to be started as follows:

```
node publisher port numMessages topic1 topic2 ...
```

where:

- `port` is the port number where subscribers should connect to.
- `numMessages` is the total amount of messages to publish (one message per second). Once those messages have been sent, the publisher ends.
- `topic1 topic2 ...` is a given list of topics that are scanned following a circular turn.

For instance, with this command `node publisher 8888 5 Politics Soccer Culture`, the publisher should send these messages:

- In 1 second: `1: Politics 1`
- In 2 seconds: `2: Soccer 1`
- In 3 seconds: `3: Culture 1`
- In 4 seconds: `4: Politics 2`
- In 5 seconds: `5: Soccer 2`

In them, there is a first part with the ellapsed time (in seconds), a second with the topic and a third part with the amount of sent messages in that topic.

2.2. Chat application

`RefZMQ.pdf` describes the design and implementation of a simple chat application. Please, read carefully its code and run it.

2.3. Configurable broker

Taking as a base the broker pattern implemented with `router/router` sockets described in `RefZMQ.pdf`, modify its code in order to accept the following command-line arguments:

- `node broker.js portFrontend portBackend`
- n commands `node worker.js urlBackend nickWorker txtReply`
- m commands `node client.js urlFrontend nickClient txtRequest`

Argument values should be consistent among them. For instance:

- `node broker.js 8000 8001`
- `node worker.js tcp://localhost:8001 W1 Resp1`
- `node worker.js tcp://localhost:8001 W2 Resp2`
- `node client.js tcp://localhost:8000 C1 Hello`
- `node client.js tcp://localhost:8000 C2 Hola`
- `node client.js tcp://localhost:8000 C3 Hi`

Thus, when a client sends a request (e.g. "txtRequest" message), it should receive a reply "txtReply n", where n is a numerical value that states how many replies have been generated up to now by all workers.

In order to arrange a test with several clients and workers, we should consider that:

- Instead of using a terminal for each command, we may start multiple instances with a single command line:
  - `node client.js & node client.js & ...`
  - or with a shell-script (passing the number of processes as an argument to it)
- In that case, in order to kill each one of those processes we won't be able to use ctrl+C. So:
  - We may use `kill pid1 pid2 ...`, where `pidX` is the process ID shown when each process was started.
  - Another alternative is to limit the duration of each client or worker using in their programs this statement: `setTimeout(()=>{process.exit(0)}, ms)`, where `ms` states how many milliseconds will run that process.

2.4. Statistics-reporting broker

Considering the broker pattern with `router/router` sockets presented in `RefZMQ.pdf`, extend its broker in order to keep the total amount of served requests and the amount of requests managed by each worker. That information should be shown on the screen every 5 seconds.

2.5. Split broker

Considering the broker pattern with `router/router` sockets presented in `RefZMQ.pdf`, divide that original broker in two halves (broker1 and broker2):

- Broker1 interacts with the clients and keeps the queue of pending requests.
- Broker2 handles the workers (initial registration, workload balancing…)

Those brokers behave as follows:

- Clients send their requests to Broker1, who forwards them to Broker2, and the latter to the appropriate worker.
- Broker2 receives the worker answers, and forwards them to Broker1, who finally propagates them to the appropriate clients.

Your implementation should preserve the same external characteristics shown by workers and clients when they interacted with the original broker.

2.6. Fault-tolerant broker

`RefZMQ.pdf` describes several implementations of the broker pattern. Please compare the behaviour of the original `router/router` broker with that of the broker that tolerates worker failures.

1. Let us start a `router/router` broker and three workers. Then, kill the first of those workers with the `kill` command. Start later three clients:
   - ```
     $ node broker & node worker & node worker & node worker &
     [1] 10300 [2] 10301 [3] 10302 [4] 10303
     $ kill 10301
     $ node client & node client & nodel client &
     ```
   - Write down how many answers are delivered to clients, which workers sent them and whether there are any clients with pending answers.
   - Kill all these processes: `killall node`
     ```

2. Repeat the same scenario, but using the fault-tolerant broker instead of the standard `router/router` one.