TSR - LAB 2
TERM 2017/2018

ØMQ

The individual exam for this lab project will be done on December 11

This lab consists of three sessions. Its main goal can be described as follows:

"To make the student familiar with the usage of the 0MQ messaging system. At the end of this lab, the student should know how to develop 0MQ applications on NodeJS."

To reach our goal, we will adhere to the following plan:

The basic 0MQ communication patterns, already seen in the seminars, will be studied, carrying out modifications over provided programs implementing those patterns.

You will be asked to implement a request forwarder service as follows:

- The system will be implemented by means of a server (referred to as broker or proxy), offering a **0MQ** socket of type **router.** Incoming connections from clients will be accepted through this socket.
- Additionally, the server will maintain connections with a pool of workers, ready to handle the requests arriving to the server. The best connection strategy between the workers and the server must be selected among the possibilities provided by 0MQ.
- When a client message arrives, the proxy selects one of the workers, and forwards the message to it.
- When the worker has handled the message, it sends a response to the proxy. The proxy will act sending it to the client originating the corresponding request.

Finally, students will modify this system implementing several modifications on which the exam for this lab will be focused.

Part 1 of this document describes the activities related to our first specific goal (analysis of the basic ØMQ communication patterns). **Part 2** addresses the second specific goal (implementing the request forwarder). **Part 3** describes the modifications to carry out on the system resulting from the implementation suggested in the previous section. [NOTE: This document version does not contain yet that last part. A complete version will be posted at the PoliformaT site next week, including Part 3.]

We recommend storing the material developed within a folder, naming it **TSRlab2**.

To carry out this lab you must have access to a development environment. That environment was already used in the first lab project: each student has a virtual machine that can be accessed through the VPN of *portal-ng.dsic.cloud*.

CONTENTS

0 PART 0. INTRODUCTION TO ØMQ (REVIEW)

In this lab we use **ØMQ** (or **ZeroMQ)**, version 4.1.x, a communications *middleware* oriented to message queues. This middleware has already been installed on the lab machines. We plan to use it within the NodeJS environment. This requires installing the 0MQ NodeJS wrapper, preferably in your $HOME directory. That task has already been done in your default virtual machine environment (i.e., that for the **root** user) that you are using in the labs. To this end, this command was used:

```
bash-4.1$ npm install zmq
```

A correct execution of the above command produces a `$HOME/node_modules/zmq` directory containing everything needed to use ØMQ from within the NodeJS environment.

From this moment on, NodeJS programs needing to use ØMQ, should proceed as seen in the seminar, including a line like this.

```
var zmq = require('zmq');
```

If execution of the above line produces an error message similar to this one...

```
module.js:340
    throw err;
          ^
Error: Cannot find module 'zmq'
    at Function.Module._resolveFilename (module.js:338:15)
    at Function.Module._load (module.js:280:25)
    at Module.require (module.js:364:17)
    …
```

…it means that command `npm install zmq` was not carried out correctly.

There is plenty of information about ØMQ directly on their *website*: http://zeromq.org. Concretely, you are advised to peruse, and resolve doubts via the following docs:

- "**ØMQ – The Guide**", available at http://zguide.zeromq.org/page:all
- "**The ØMQ Reference Manual**", available at http://api.zeromq.org

That documentation uses C as the reference programming language for the various examples provided. Despite this, it is possible to also find some of the examples implemented using several of the languages on which ØMQ is available (C++, Java, Haskell, Node.js, Python, etc.).

In "ØMQ – The Guide" you can find a reference to a public Git repository containing all the examples within the guide. You are free to clone it using `git clone` and the URL specification of the repository (the guide contains the exact command). However, this directory contains many files that won't be useful to us (examples in languages other than JavaScript).

We have prepared a folder (`/asigDSIC/ETSINF/tsr/lab2`), accessible from the lab machines, in which file **zmq_nodejs_examples.tgz** contains the relevant examples for our NodeJS environment.

Besides this, in the virtual machines there is also a directory (**/root/zmq_nodejs_examples**) holding all those files.

In the next section (Part 1), we will study several of the examples in ***zmq_nodejs_examples***, introducing some interesting modifications. Those examples are correct implementations of several basic communication patterns provided by ØMQ. We remind you that to avoid getting errors when running those examples, the ports they use need to be modified to fit the available port range within the laboratory machines: **ports 8000 to 8100,** inclusive.

1 PART 1. BASIC ØMQ PROGRAMMING

Once you get a copy of ***zmq_nodejs_examples***, you will introduce changes in some of the example programs. We will proceed by first creating a new folder under **TSRlab2**, name it ***zmqbasico***. We will add to it the example programs to be changed.

1.1 Analysis of the REQ – REP pattern

To begin with, we analyse two programs (their code is included in the following subsections) that show the easiest interaction pattern using ØMQ sockets: the REQ-REP pattern.

1.1.1 hwclient.js

```
01:  // Hello World client
02:  // Connects REQ socket to tcp://localhost:5555
03:  // Sends "Hello" to server.
04:
05:  var zmq = require('zmq');
06:
07:  // socket to talk to server
08:  console.log("Connecting to hello world server...");
09:  var requester = zmq.socket('req');
10:
11:  var x = 0;
12:  requester.on("message", function(reply) {
13:    console.log("Received reply", x, ": [", reply.toString(), ']');
14:    x += 1;
15:    if (x === 10) {
16:      requester.close();
17:      process.exit(0);
18:    }
19:  });
20:
21:  requester.connect("tcp://localhost:5555");
22:
23:  for (var i = 0; i < 10; i++) {
24:    console.log("Sending request", i, '...');
25:    requester.send("Hello");
26:  }
27:
28:  process.on('SIGINT', function() {
29:    requester.close();
30:  });
```

1.1.2 hwserver.js

```
01:  // Hello World server
02:  // Binds REP socket to tcp://*:5555
03:  // Expects "Hello" from client, replies with "world"
04:
05:  var zmq = require('zmq');
06:
07:  // socket to talk to clients
08:  var responder = zmq.socket('rep');
09:
10:  responder.on('message', function(request) {
11:    console.log("Received request: [", request.toString(), "]");
12:
13:    // do some 'work'
14:    setTimeout(function() {
15:
16:      // send reply back to client.
17:      responder.send("World");
18:    }, 1000);
19:  });
20:
21:  responder.bind('tcp://*:5555', function(err) {
22:    if (err) {
23:      console.log(err);
24:    } else {
25:      console.log("Listening on 5555...");
26:    }
27:  });
28:
29:  process.on('SIGINT', function() {
30:    responder.close();
31:  });
```

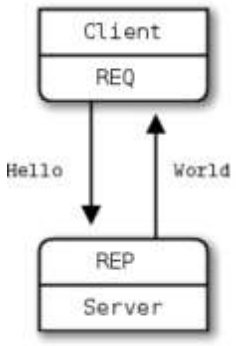

***Figure** 1: Client-server communication with the REQ-REP pattern*

Remember that you must use the VM you are provided with to run all these programs.

1.1.3 Program modifications: Invocation arguments

These two programs implement the **REQ** - **REP** pattern. After reading and understanding the code you should perform the following changes (within project ***zmqbasico***) on new versions of programs ***hwclient.js*** y ***hwserver.js***

1) The **client** program should accept the following arguments:

a. The server's *endpoint* URL (e.g., **tcp://localhost:8000**).
b. The number of requests to be sent (e.g., 10).
c. Text to send with the requests (e.g. **WORK**).

2) The **server** program should accept these arguments:
   a. Server port (e.g., 8000, in order to match what has been used by the client).
   b. Number of seconds to spend on each request before sending the answer (e.g., 5).
   c. Text to send in the answer message (e.g., **DONE**).

Functional requirements

- The answer sent from the server to the client will be constructed by concatenating the client's request and the server's own answer text (the text of the last argument of its invocation).

- If the number of arguments given to any of the programs is incorrect, the program must show a help message and exit immediately. That help message must show the right way to invoke that program.

Solution proposal

NodeJS programs may receive arguments from the command line. Thus, programs may receive their port numbers, identifiers and other data from the command line, instead of declaring them as constants. Indeed, we may use a hybrid mode, using logical expressions like A||B for testing whether A (possibly, a command-line argument) has any value, using value B otherwise. Thus, this technique might be applied in `hwclient.js`, generating this code:

```
var args = process.argv.slice(2);
var servURL = args[0] || "tcp://localhost:8055";
```

This means that the invocation arguments are copied into the `args` array. Besides, `servURL` gets its value from the first argument, but when no argument has been passed, it is initialised to "**tcp://localhost:8055**".

Now, the original line 21 should be: `requester.connect(servURL);`

We should apply the same strategy in `hwserver.js`.
Besides, we must manage the other two arguments in the same way, using a variable (with a default value) instead of the original code constants.
The programs generated by these modifications must have these names: hwclient_p and hwserver_p.

Testing

It must be possible to run client and server on the same computer (*localhost*) or on two different computers. You must use the VM you are provided with. In order to choose which program to run within the VM, and which one to run on the local environment, keep in mind that the VM has a well-known IP address that will not change…

The server must be able to attend requests from several clients: you should observe the order in which the server handles concurrent requests from different clients[1].

[1] Remember the basic REQ-REP pattern for multiple requesters, studied in Seminar 3.

1.2 Analysis of the PUB - SUB pattern

Our next step is to analyse another couple of programs, available in ***zmq_nodejs_examples***. Their code is shown below:

1.2.1 subscriber.js

```
01:  var zmq = require('zmq')
02:  var subscriber = zmq.socket('sub')
03:
04:  subscriber.on("message", function(reply) {
05:    console.log('Received message: ', reply.toString());
06:  })
07:
08:  subscriber.connect("tcp://localhost:8088")
09:  subscriber.subscribe("")
10:
11:  process.on('SIGINT', function() {
12:    subscriber.close()
13:    console.log('\nClosed')
14:  })
```

1.2.2 publisher.js

```
01:  var zmq = require('zmq')
02:  var publisher = zmq.socket('pub')
03:
04:  publisher.bind('tcp://*:8088', function(err) {
05:    if(err)
06:      console.log(err)
07:    else
08:      console.log("Listening on 8088...")
09:  })
10:
11:  for (var i=1 ; i<10 ; i++)
12:      setTimeout(function() {
13:          console.log('sent');
14:          publisher.send("Hello there!")
15:      }, 1000 * i)
16:
17:  process.on('SIGINT', function() {
18:    publisher.close()
19:    console.log('\nClosed')
20:  })
```

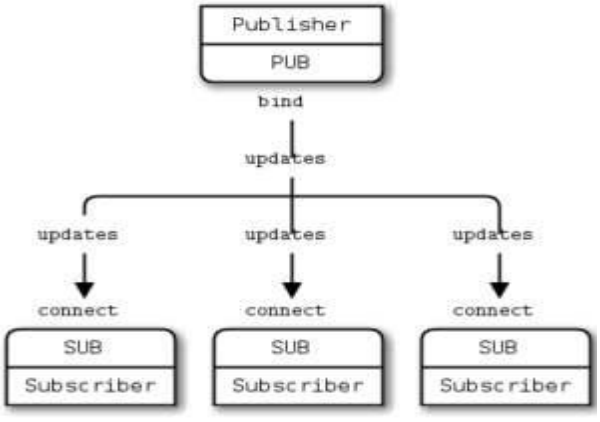

***Figure** 2: Publisher-subscriber communication using PUB-SUB sockets.*

1.2.3 Program modifications: Invocation arguments

These two programs implement the **PUB** - **SUB** communication pattern as provided by ØMQ sockets. After reading and understanding the code, you should apply the following changes to the new ***subscriber.js*** and ***publisher.js*** programs:

1) The subscriber program must accept the following arguments:
    a. URL of the *endpoint* at the publisher (e.g., **tcp://localhost:8000**).
    b. Subscription descriptor filter (e.g., **NEWS**).
2) The publisher program accepts the following arguments:
    a. Publisher's port (e.g., **8000**).
    b. Number of messages to publish (e.g., **20**).
    c. First type of messages to publish (e.g., **NEWS)**
    d. Second type of messages to publish (e.g., **SALES)**.

Functional requirements

- The publisher will publish the same number of messages of the two specified kinds. Each published message is built concatenating the descriptor for its type of message with a random number.

> **NOTE:** You can produce a random number within a given range by using function randNumber, provided by module **auxfunctions1718.js** (code available in Annex 1. Auxiliary functions). To this end, this line should be used at the beginning of your program:
>
> ```
> aux = require("./auxfunctions1718.js");
> ```
>
> and the invocation should be like this: `aux.randNumber(…)`.
> (We assume that file ***auxfunctions1718.js*** is placed in the folder that contains publisher.js and subscriber.js).

- Subscribers must connect to the publisher, using the provided URL, and using the proper subscription filter (as provided in the arguments).

Testing

You must verify proper behaviour of the PUB-SUB pattern by either running the programs at the same machine or on two different machines. Use the assigned VM…

You must verify the behaviour of at least two subscribers connected to the same publisher, using each a different subscription filter.

1.2.4 Program modifications: Verbose mode

This mode generates a trace of the most important steps in the execution of the programs. In our case, those steps are message sending and delivery.

This mode is needed when we debug our programs, especially for observing the internal evolution of each process. Since we are not always interested in those traces, their generation depends on another optional command-line argument (the word "verbose"). It should be passed in order to activate this verbose mode.

The code being needed for managing this optional argument could be like this:

```
var args = process.argv.slice(2);
var verbose = false;
if (args[args.length-1] == "verbose") {
  verbose = true;
  args.pop(); // eliminate only if it appears at the end
}             // rest of argument processing will follow
```

Besides, we should replace trace-printing statements like:

```
    console.log("Listening on 8088...")
```

with others that only show the information when ***verbose*** is active:

```
    if (verbose)
      console.log("Listening on 8088...")
```

The programs to be obtained when these modifications are applied will be named publisher_pv and subscriber_pv.

1.2.5 Program modifications: round-robin publisher

Taking as a base publisher_pv, with no modification on the subscriber, that publisher must be extended implementing this behaviour:

- It should accept, as command-line arguments, an indeterminate amount of topic names.
- This new publisher (publisher2_pv) must send the total amount of messages stated in the first argument.

For instance, a valid command line for starting a publisher could be like this:

```
node publisher2_pv 8088 100 entertainment health business sport verbose
```

This means that this publisher must send 100 messages, alternating these topics: entertainment, health, business and sport; i.e., 25 messages in each topic, to be sent in a circular way.

2 PART 2. INTERMEDIATE PROXY

2.1 Introduction

The goal in this part is the implementation of a proxy[2] that manages request messages. In the classical problems of concurrent programming there is a model known as **producer-consumer** where a bounded buffer is used for resource management.

- Producers insert (i.e., a *write* action) elements in the buffer. Those elements are extracted (i.e., a *read and delete* action) by consumers.

Solutions to this problem are focused on coordinating (using conditional synchronisation) those processes, keeping in mind that the buffer may become full or empty.

- When the buffer is empty, consumers must wait until a new element is inserted.
- When the buffer is full, producers must wait until one of the elements is extracted.

An edge case consists in using a buffer with zero capacity, implementing all operations as synchronous message sendings or receptions. That is the case being assumed in this part.

In detail, the expected behaviour for each role is:

- There are processes (clients) that request service. To this end, they produce a message to be sent to any server (worker). If there is any available worker, it will process that request; otherwise, the client must wait.
    - Once a free worker is found, the client still remains blocked, waiting for its result (another message). Once the result is obtained, the client resumes its activity.
- Worker processes behave as servers. To this end, they produce a message in order to announce their availability. If there is any waiting client, that new worker gets the first enqueued client request; otherwise, that worker blocks, waiting for a new request.
    - Once a worker gets a request, it processes that request and sends a reply message. That reply has two complementary meanings: (1) it carries the result for the client, and (2) it notifies the proxy that such worker is available now.
    - As an exception, the first message sent by a worker only announces its availability, since it has not got yet any request to be replied.
- A third agent is needed for managing the messages to be exchanged by those two roles. That agent is the proxy (also known as broker). It keeps track of the current state of each worker and correctly forwards the replies to their intended clients. Besides, it hides the current set of workers to the requesting clients, providing thus some level of replication, location and failure transparency.

2.2 Implementing the Proxy with ØMQ

[2] We also use the terms “intermediary” and “broker” in order to refer to this agent.

In this part, our goal is to implement a proxy managing service request messages. This proxy, or broker, behaves as follows:

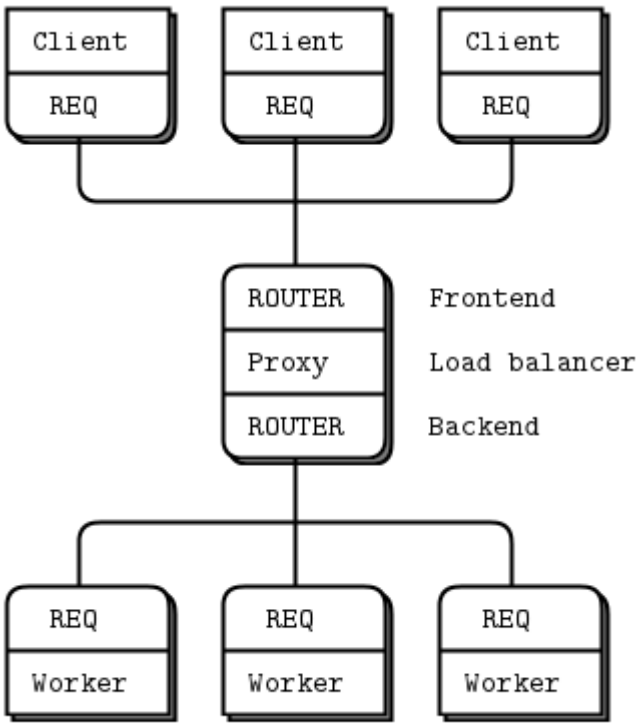

Figure 3: Scheme for a broker using ROUTER-ROUTER pattern

- It exposes a **ROUTER ØMQ socket**, accepting remote connections from the clients (which use this socket to send service request messages). We will refer to this socket as the ***frontend***.
- It exposes another **ROUTER ØMQ socket**, accepting remote connections from the workers (which are the ones actually serving the client requests). We will refer to this socket as the ***backend***.
- When a client, using its **REQ ØMQ socket**, sends a service request to the *frontend*, the proxy chooses a worker, forwarding it the request message through the *backend*.
- Once a request message is handled by one of the workers, that worker uses its **REQ ØMQ socket** to send back the result, which will reach the *backend* at the proxy. Note that the result is sent as a request message to the *backend*.
- When the *backend* receives a "result" message from a worker, the proxy forwards it to the client whose request was handled by the worker sending the response to the backend. The proxy uses the *frontend*, to send this message.
- The message management just described requires that the proxy knows the identities of clients and workers.

Internal operation

Regarding conditional synchronisation:

- Each time a new worker is registered, or any other worker replies its current request, the broker checks whether there are any waiting clients.

- If so, the first pending client request is removed from that queue and it is forwarded to that free worker.
- Otherwise, the worker is inserted in the queue of available workers.

- Each time a new client request arrives, the broker checks whether there is any available worker.
  - If so, it selects one of them and forwards the request to that worker.
  - Otherwise, such client request is inserted in the queue of waiting clients.

Regarding agent life cycles:

- A client terminates when it receives replies for all its requests.
- Workers and broker never terminate. They must indefinitely wait for incoming requests.

2.2.1 Client description

The client program (***myclient.js***) contains the code being needed in order to run a client process. It is connected to the broker frontend.

Functionality

- Clients use a **REQ** socket for sending request messages. In its default version, the client program only sends one request.
- Clients wait for a reply. Once such reply is received, clients terminate.

2.2.1.1 Initial client program (myclient.js)

```
01:  // myclient in NodeJS
02:  var zmq       = require('zmq')
03:    , requester = zmq.socket('req');
04:
05:  var brokerURL = 'tcp://localhost:8059';
06:  var myID = 'NONE';
07:  var myMsg = 'Hello';
08:
09:  if (myID != 'NONE')
10:    requester.identity = myID;
11:  requester.connect(brokerURL);
12:  console.log('Client (%s) connected to %s', myID, brokerURL)
13:
14:  requester.on('message', function(msg) {
15:    console.log('Client (%s) has received reply "%s"', myID, msg.toString());
16:    process.exit(0);
17:  });
18:  requester.send(myMsg);
```

2.2.2 Worker description

The worker program (***myworker.js***) contains the code being needed in order to run a worker process. It is connected to the broker backend.

Functionality

- Workers use a **REQ** socket to interact with the broker:
  - They send a “request” in order to register their availability in the broker, when they are started.

- Later on, for each incoming request and once such request has been processed, they send a reply to the broker. Such reply is forwarded by the broker to the appropriate client.

- Workers never terminate: they remain indefinitely waiting for incoming requests.

2.2.2.1 Initial worker program (myworker.js)

```
01:  // myworker server in NodeJS
02:  var zmq = require('zmq')
03:    , responder = zmq.socket('req');
04:
05:  var backendURL = 'tcp://localhost:8060';
06:  var myID = 'NONE';
07:  var connText = 'id';
08:  var replyText = 'World';
09:
10:  if (myID != 'NONE')
11:    responder.identity = myID;
12:  responder.connect(backendURL);
13:  responder.on('message', function(client, delimiter, msg) {
14:      setTimeout(function() {
15:      responder.send([client,'',replyText]);
16:      }, 1000);
17:  });
18:  responder.send(connText);
```

2.2.3 Broker description

The broker program (***mybroker.js***) contains the code being needed in order to run a broker (a.k.a. proxy or intermediary) process. Its basic functionality has already been described at the beginning of this lab part. However, additional details are given in this section.

Functionality

- It uses two **ROUTER** sockets for communication management.
- It stores in a given data structure (queue, list, table…) the identifiers of the currently available workers.
- When a client request is received (in its frontend socket), it checks whether any available workers exist. If so, it chooses the best one (according to a given criterion; for instance: load balancing), forwarding that request to it (using the backend socket to this end). Otherwise, the request is kept in a queue.

2.2.3.1 Initial broker program (mybroker.js)

```
01:  // ROUTER-ROUTER request-reply broker in NodeJS
02:  var zmq      = require('zmq')
03:    , frontend = zmq.socket('router')
04:    , backend  = zmq.socket('router');
05:
06:  var fePortNbr = 8059;
07:  var bePortNbr = 8060;
08:  var workers = [];
09:  var clients = [];
10:
11:  frontend.bindSync('tcp://*:'+fePortNbr);
12:  backend.bindSync('tcp://*:'+bePortNbr);
13:
14:  frontend.on('message', function() {
15:    var args = Array.apply(null, arguments);
16:    if (workers.length > 0) {
17:      var myWorker = workers.shift();
```

```
18:       var m = [myWorker,''].concat(args);
19:       backend.send(m);
20:     } else
21:       clients.push( {id: args[0],msg: args.slice(2)});
22:   });
23:
24:   function processPendingClient(workerID) {
25:     if (clients.length>0) {
26:       var nextClient = clients.shift();
27:       var m = [workerID,'',nextClient.id,''].concat(nextClient.msg);
28:       backend.send(m);
29:       return true;
30:     } else
31:       return false;
32:   }
33:
34:   backend.on('message', function() {
35:     var args = Array.apply(null, arguments);
36:     if (args.length == 3) {
37:       if (!processPendingClient(args[0]))
38:         workers.push(args[0]);
39:     } else {
40:         var workerID = args[0];
41:         args = args.slice(2);
42:         frontend.send(args);
43:         if (!processPendingClient(workerID))
44:           workers.push(workerID);
45:     }
46:   });
```

2.3 Internal operation of brokers with ROUTER-ROUTER sockets

Let us explain the **REQ-ROUTER** and **ROUTER-REQ** communication patterns. They have been already partially described in Seminar 3. They are needed form managing the interactions between the three agents presented above: client-proxy and proxy-worker.

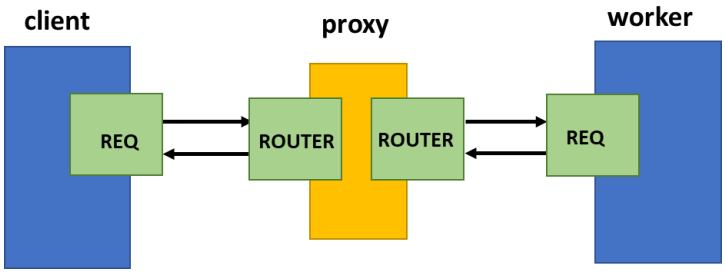

Figure 4: Global scheme of the interacting agents.

Let us describe first how a client request reaches a worker.

2.3.1 Forwarding a request to a worker

The client program (*myclient.js*) sends the request message (**"Work"**) from its REQ socket to the broker (*mybroker.js*) ROUTER socket. Let us assume that the client ID is "**CLIENT1**".

NOTE: to specify the identity, you must use the following statement in *myclient.js* before connecting to the broker ROUTER socket (line 10 in *myclient.js*):

```
…
requester.identity=myID;//Let's assume myID="CLIENT1".
…
requester.connect(brokerURL);
…
```

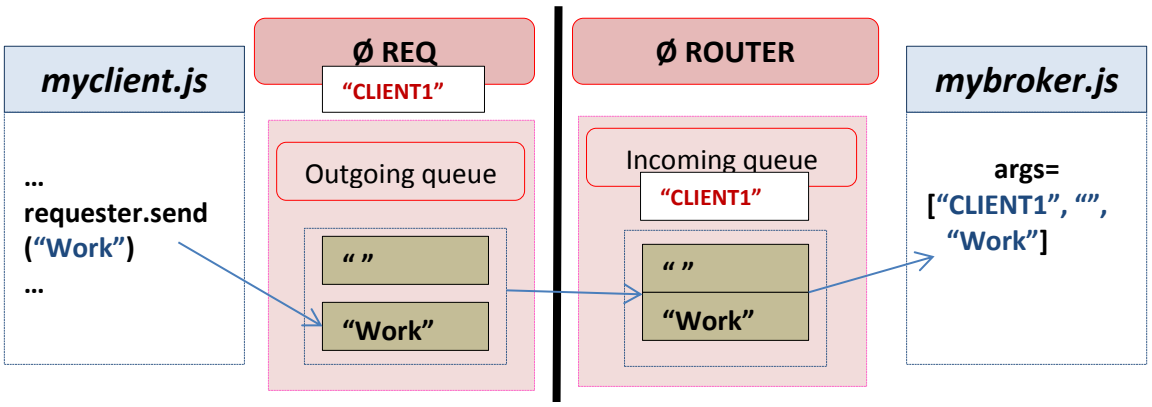

***Figure** 5: Sending messages from client to broker using REQ-ROUTER socket communication*

The REQ socket prepends an empty segment (a delimiter) to the message being sent by the client program. The ROUTER socket gets both message segments within its queue for CLIENT1. The ROUTER socket prepends a new segment to the message containing "**CLIENT1**" (the client's ID), before handing the message to the broker code (see **Figure** 5). Thus, *mybroker.js* receives a message with three segments [*"CLIENT1","","Work"*]. That message is assigned to the ***args*** variable.

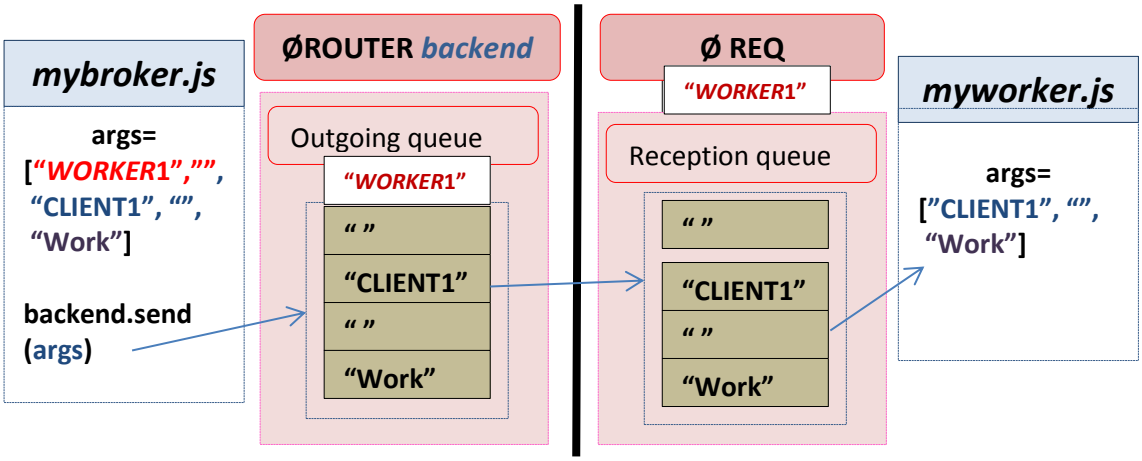

Figure 6: Sending a request from the broker to the server using ROUTER-REQ communications.

Let us now see how the broker sends the message to a server (worker).

Since the backend socket in the broker is a ROUTER, *mybroker.js* may choose to which worker the request message is sent. To this end, is should provide the identity of that worker. That identity, and a delimiter, must be inserted as the first two segments of this new message (take a look at the segments in red in variable **args** at the *mybroker.js* program, in Figure 6).

NOTE: In this case the worker identity is not automatically prepended by the ØMQ socket. Instead, it should be chosen by the program, from the list of available workers. This identity is needed for placing the outgoing message in the appropriate outgoing queue for the intended worker. Besides, that worker is "unavailable" now; i.e., it cannot receive other requests until its reply is obtained.

Assuming that the worker identity is "*WORKER1*", the message to be placed in the outgoing (i.e., sending) queue at the backend is [*"WORKER1","","CLIENT1","","Work"*], as Figure 6 shows.

As it is shown in Figure 6, the backend ROUTER socket uses the outgoing queue for "*WORKER1*". That socket removes the identity of the worker (WORKER1) to route the message. Thus, the message received by the REQ socket has a delimiter in its first segment. The REQ socket eliminates this delimiter, and handles the rest to *myworker.js*. Thus, the worker receives a message with three segments: [*"CLIENT1","","Work"*].

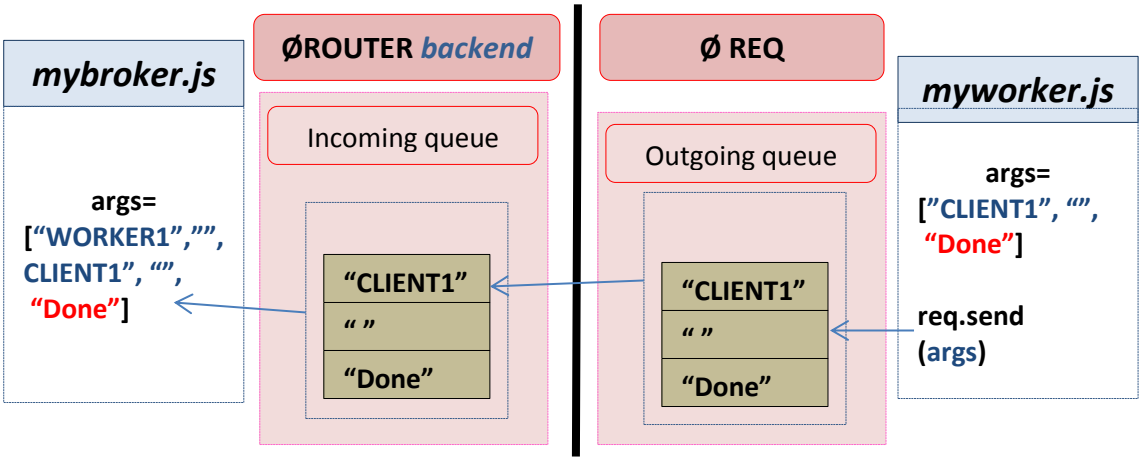

Figure 7: Sending the answer from the worker using a REQ-ROUTER connection

2.3.2 Sending the server response

Figure 7 shows that the worker answers using a similar message. It consists of three segments, but the last one contains now the reply, instead of the request. Let us assume that such reply is the string "*Done*". Thus, the worker sends these segments as its reply: [*"CLIENT1","","Done"*]. Note that now the REQ socket prepends a delimiter segment before sending the message to the ROUTER socket.

Upon reception of the response message, the ROUTER backend socket prepends the identity of the worker, and hands the resulting message to *mybroker.js*. Thus, the broker receives a message with five segments: [*"WORKER1","","CLIENT1","","Done"*]. That message is saved in the ***args*** variable.

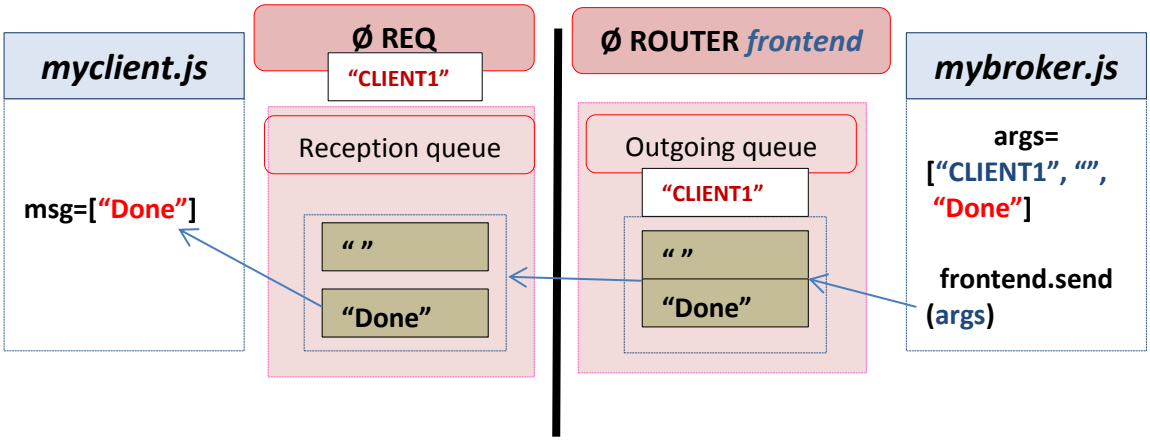

Figure 8: *Sending the answer from the broker to the client using a ROUTER-REQ communication pattern.*

To actually send back the response to the client, it should find the client identity. Fortunately, this information is in the message received, just after its first two segments. Thus, *mybroker.js* extracts those segments, containing the worker ID (which it uses to mark the worker as available). What remains is a message with these segments: [*"CLIENT1","","Done"*]. Its first

segment specifies which ROUTER connection must be used to send this message. It corresponds to the "*CLIENT1*" client. The frontend ROUTER socket extracts that segment from the message and sends it to that client.

That stage is shown in Figure 8. This means that, when the message arrives at the client REQ socket, it contains only two segments: [*"","Done"*]. REQ sockets extract the first segment in the message at this reception step. It must be a delimiter. Thus, this REQ socket delivers to *myclient.js* a message with a unique segment: ["*Done*"].

Figure9 presents a summary of the relevant interactions when sending a request and receiving a reply from a worker in the proposed architecture.

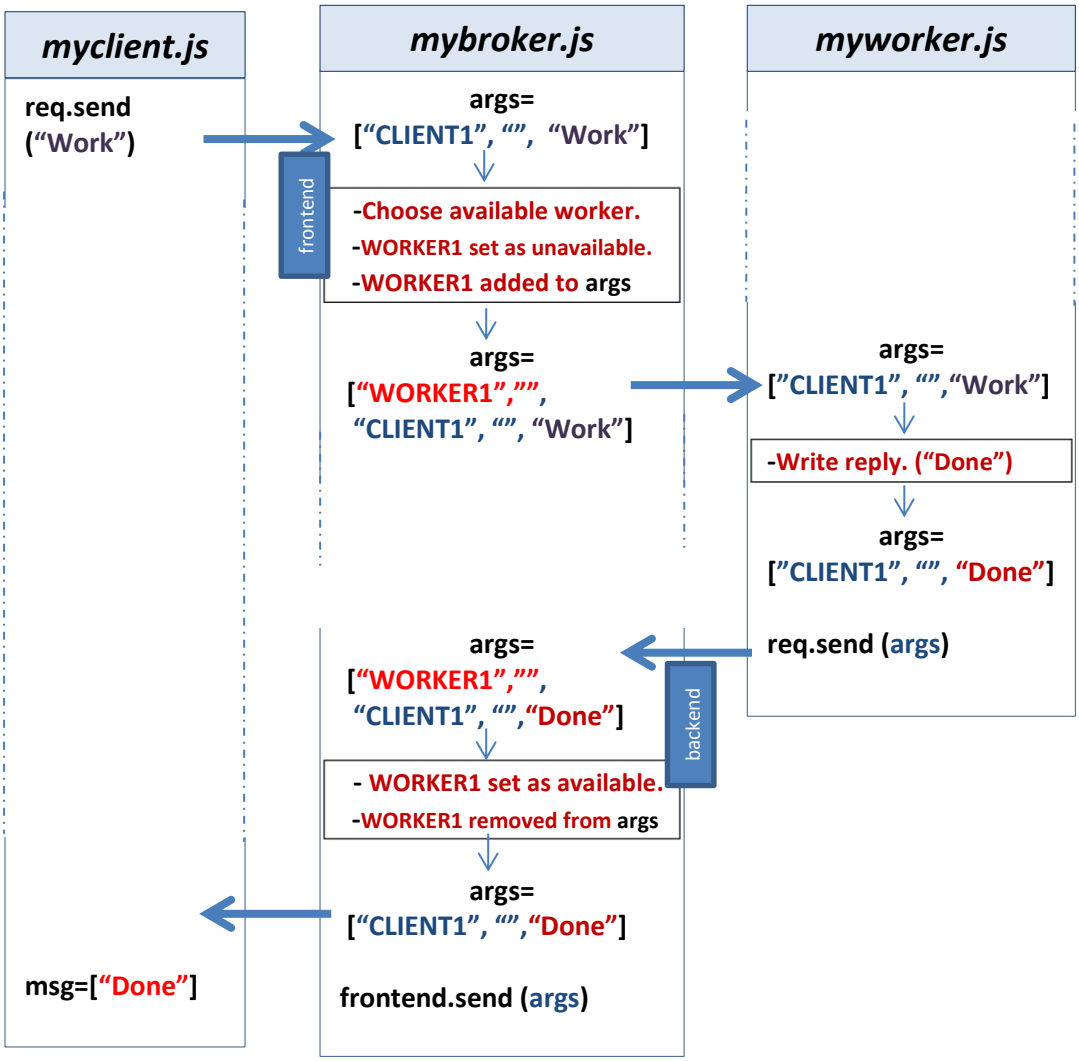

Figure 9: Scheme showing sends and receives for the proposed broker architecture

2.4 Implementation and first tests

The goal of these first tests is to check the correct transmission of the messages between client and worker, using the broker as an intermediary. Besides, we must also check which the message contents are on each transmission stage.

To test the first versions of the programs *mybroker.js*, *myworker.js* and *myclient.js*, we proceed conservatively: we run the proxy, one client, and one worker.

Unfortunately, those initial program versions do not provide any help for tracing what happens when they are running. In case of errors, we should manually revise each statement. To this end we may use statements that print the current variable values and the message contents. This is the aim of a *verbose* mode.

2.4.1 Verbose mode

The ***verbose*** mode should show something close to a trace of the program execution, as a service request traverses the system. Annex 2 shows an example of the console output produced by the programs in *verbose* mode. That is just an example of how the output looks like. Your implementation will produce different output.

The following code fragments show how this verbose mode could be implemented...

For instance, in ***myworker.js***:

```
if (verbose)
  console.log('Worker (%s) connected to %s', myID, backendURL);

responder.on('message', function(client, delimiter, msg) {
  if (verbose) {
    console.log('Worker (%s) has received request "%s" from client (%s)',
      myID, msg.toString(), client);
  }
  setTimeout(function() {
    responder.send([client,'',replyText]);
    if (verbose)
      console.log('Worker (%s) has sent its reply "%s"',
        myID, replyText);
  }, 1000);
});
responder.send(connText);
if (verbose)
  console.log('Worker (%s) has sent its first connection message: "%s"',
    myID, connText);
```

And in ***mybroker.js*** we need some statements that show the contents of the messages each time they are received or sent. To this end, we may use the **showMessage()** function from the **auxfunctions1718.js** module:

```
function showMessage(msg) {
  msg.forEach( (value,index) => {
    console.log( '     Segment %d: %s', index, value );
  })
}
```

Such function should be used as follows:

```
var aux = require("./auxfunctions1718.js");
…
function sendToWorker(msg) {
  if (verbose) {
    console.log('Sending client (%s) request to worker (%s) through backend.',
      msg[2], msg[0]);
    aux.showMessage(msg);
  }
  backend.send(msg);
}
```

2.4.2 Managing command-line arguments

Many applications may accept arguments from the command line. With them, those programs may get values for many of their relevant parameters; e.g., port numbers, IP addresses, identifiers, etc. Indeed, we may use a *hybrid* mode, using logical operators in order to assign default values to the arguments that have not been received from the command line. For instance, the value of the A||B logical expression is A if A is neither undefined nor false; otherwise, it is B. Thus, this fragment...

```
var args = process.argv.slice(2);
var fePortNbr = args[0] || 8059;
```

...means that the command line arguments are now stored in ***args***. Besides, ***fePortNbr*** takes as its value that of the first command line argument. If no argument has been given, then its value is 8059.

Let us name *myworker_vp.js, mybroker_vp.js* and *myworker_vp.js* the files generated once this and the previous extensions have been applied. The following paragraphs describe the arguments to be assumed in each agent, with the code needed for managing them:

Arguments and code in myclient_vp.js

a. URL of the frontend ROUTER socket of the broker.
b. String to be used as the client identity.

> **NOTE:** Since the client is connected with the broker through a ROUTER socket, an identity is needed for each of the connected agents. Using an argument, the user may specify an easily readable identity, instead of the ID automatically assigned by default (a string difficult to read and memorise).

c. Text of the service request. For instance: '**WORK**'.

```
04:  var args = process.argv.slice(2);
05:  var brokerURL = args[0] || 'tcp://localhost:8059';
06:  var myID = args[1] || 'NONE';
07:  var myMsg = args[2] || 'Hello';
```

Arguments and code in myworker_vp.js

a. URL of the backend ROUTER socket of the broker.
b. String to be used as the worker identity.
c. Text of the availability message[3]. For instance: '***READY***'.
d. Text of the reply message. For instance: '***DONE***'.

```
04:  var args = process.argv.slice(2);
05:  var backendURL = args[0] || 'tcp://localhost:8060';
06:  var myID = args[1] || 'NONE';
07:  var connText = args[2] || 'id';
08:  var replyText = args[3] || 'World';
```

[3] Initial request message to register that worker in the *broker*.

Arguments and code in mybroker_vp.js

a. Communication port for clients (*frontend*).
b. Communication port for workers (*backend*),

```
05:   var args = process.argv.slice(2);
06:   var fePortNbr = args[0] || 8059;
07:   var bePortNbr = args[1] || 8060;
```

Tests

You must check the correct operation of your implemented agents, while they run in a single machine (the virtual one) or in two computers (placing in that case the broker in the virtual machine and the other agents at your initial computer). Please, try at least these cases:

a) One client and two servers. Each server should use a different reply. For instance: "***DONE1***" and "***DONE2***". Check how the requests are distributed between the servers.
b) Two clients and one server. Each client should use a different request string. For instance: "***WORK1***" and "***WORK2***". Check whether the server fairly processes the requests sent by each client.
c) Two clients and three servers, with different request and reply messages for each of them. Verify that all requests have been replied. Also, check how the requests are distributed among the servers.

2.5 Further implementation and complete tests

If the previous tests were successful, we are ready to increase the number of clients and workers. At this point we should be able to verify if our broker implements a fair distribution of the work among the pool of workers.

We recommend initially going easy with the number of clients and workers. For instance, start with 3 workers and 10 clients.

From just one terminal we can easily launch[4] as many concurrent workers as we need. For instance, two workers can be launched like so:

```
bash-4.1$ node myworker_vp tcp://my-public-IP:8060 WORKER1 Ready DONE &
          node myworker_vp tcp://my-public-IP:8060 WORKER2 Ready OK &
```

Be careful with launching background executions of programs that never end. We recommend modifying the worker so that it exits after some finite time (e.g., using *setTimeout* to make sure *process.exit();* is executed at some point).

If you want to test with a large number of clients (e.g., 100) you should use a different strategy: implement a ***shell script*** [5].

[4] We assume a public IP address like 192.168.1.1 that is different to that of localhost. We use **my-public-IP** in order to refer to that public address.

Let us consider the following simple *script*:

```
01:    #!/bin/bash
02:
03:    number=1
04:    while [ $number -lt $1 ]; do
05:        echo "ZeroMQ != $number MQ"
06:        number=$((number + 1))
07:    done
08:    echo "ZeroMQ == 0MQ"
```

Starting with the general shape in this script, it should be easy to implement a couple of scripts ***myclients_script.sh*** and ***myworkers_script.sh***, which will let us launch a number arbitrarily large of clients and workers with just one shell command.

2.6 Other minor extensions

2.6.1 Statistics

The broker must be able to report how many requests have been processed by each registered worker. To this end, some information should be collected in a data structure. For instance, we may use this array in the broker:

```
// Array with the counters of how many requests have been processed
// by each worker.
var requestsPerWorker = [];
```

In **backend.on**, when a new worker is registered (`args.length==3`), its array slot is initialised: `requestsPerWorker[args[0]]=0`, otherwise, only an increment is needed: `requestsPerWorker[args[0]]++`. Recall that `args[0]` contains the worker identity.

In order to report those statistics (e.g., when [Ctrl]+[C] is pressed by the user), this code may be added in the broker:

```
// Function that shows the service statistics.

function showStatistics() {
      var totalAmount = 0;
      console.log('Current amount of requests served by each worker:');
      for (var i in requestsPerWorker) {
            console.log('   %s : %d requests', i, requestsPerWorker[i]);
            totalAmount += requestsPerWorker[i];
      }
      console.log('Requests already served (total): %d', totalAmount);
}

// Show the statistics each time [Ctrl]+[C] is pressed.

process.on('SIGINT', showStatistics);
```

[5] A good reference is: http://linuxcommand.org/writing_shell_scripts.php. Where, in particular, you can find how to use loops: http://linuxcommand.org/wss0120.php#loops

2.6.2 Broker chaining

Let us extend our initial broker, replacing it by two collaborating components (e.g., broker1 and broker2).

- Clients send their request to broker1. Broker1 forwards them to broker2. Finally, broker2 sends them to the chosen worker.
- The path to be followed by replies is the opposite one.

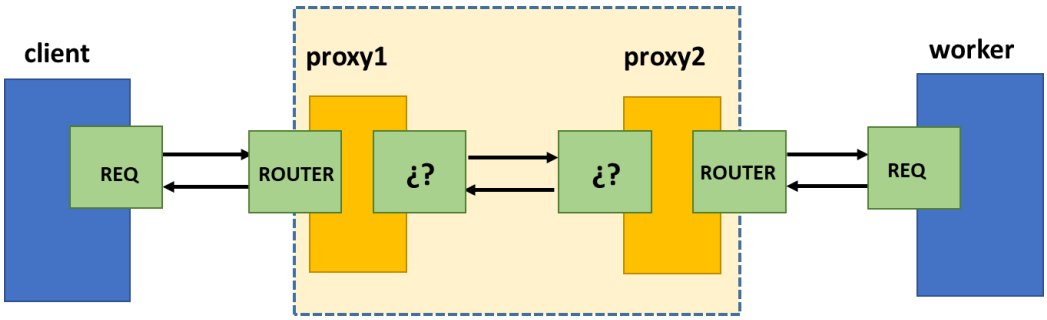

Figure 10: *Global scheme of agents in a ROUTER-ROUTER pattern, with chained brokers.*

You must implement the program for both brokers, ensuring that the resulting pair handles clients and workers in a transparent way. This means that both *myclient.js* and *myworker.js* do not need any change for interacting with proxy1 and proxy2, respectively. Please be careful in your choice of the communication pattern to be used between proxy1 and proxy2 (types of sockets and message segments).

You must also activate the verbose mode in both brokers in order to:

- Show the client identity for each request message being received by broker1: “Client XX has sent a request.”
- Show a reporting info for each reply message to be forwarded by broker1 to a given client: “Forwarding a reply message to client XX.”

3 SOME EVOLUTIONS OF THE ROUTER-ROUTER PATTERN

Let us describe some new goals taking as a base the system we have seen in the previous section. The first two extensions are described and solved in detail, while the other is only proposed. You must carefully implement that third extension. Besides, the two first extensions should be carefully read and run, in order to answer their related questions.

These extensions are:

1. **Job types**. Workers may be classified according to the type of job (i.e., type of requests) they are able to manage.
2. **Failure detection**. The broker, using some kind of worker liveness monitor (e.g., "heartbeats" or periodical echo messages), is able to detect the failure of any worker.
3. **Load balancing.** If the broker knows the current workload level at each worker, it may choose that with the lowest load level for serving each incoming request.

3.1 Job types

In some scenarios, server agents may have different resources and computing capacity. There may be nodes with powerful graphic cards whose GPU consists of thousands of cores, other nodes may have a very large storing capacity, and others may have other distinguishing characteristics.

- The broker might analyse each request, choosing the most appropriate worker for each of them. However, such analysis must be very fast, requiring a very short processing interval. Otherwise, the overall service throughput will be severely penalised. Moreover, that analysis cannot be achieved without some help from the remaining components.
- Thus, in order to make this easy, clients should specify the request type in each sent message, and workers should also state which types of job (i.e., types of request) they may accept. This implies that sent messages are extended in both roles (clients and workers).

Let us explain an example of this technique, assuming that there are three types of request (e.g., {**R, G, B**}). Those types must be stated in the client request messages and in the worker availability ones.

Both agents (clients and servers) receive a `classID` argument from the command line.

- **Clients** specify that `classID` as one of the parts in each request message.
- **Workers** specify their type of accepted jobs as their initial registering message. For instance, if they manage requests of type B, they will register with `s.send('B')`.

We must write a new version of our agents (`myclient1`, `mybroker1` and `myworker1`) according to that described scheme. No new socket is needed in them.

An example of command line for starting `myworker1.js` is:

```
node myworker1.js Worker1 localhost:8099 R
```

that, in this case, means the following:

- The worker identifier is `Worker1`

- It connects to the broker at `tcp://localhost:8099`
- It processes requests of type R

On the other hand, clients (`myclient1.js`) may be started with:

```
node myclient1.js Client1 localhost:8098 G
```

that, in this case, means:

- The identifier for this client is `Client1`
- It connects to the broker at `tcp://localhost:8098`
- It sends requests of type G

In order to check this extension, we should run concurrently an instance of mybroker1 and several[6] instances of clients and workers. An example is given here:

```
node mybroker1.js 8098 8099
node myworker1.js Worker1 localhost:8099 R
node myworker1.js Worker2 localhost:8099 G
node myclient1.js Client1 localhost:8098 G
node myclient1.js Client2 localhost:8098 G
node myclient1.js Client3 localhost:8098 R
node myclient1.js Client4 localhost:8098 G
```

3.1.1 Solution keys

The management of N types of job is similar to using N brokers. With the latter, clients and workers interested in jobs of type K use broker K for interacting.

That behaviour can be emulated using N types of job and a single broker. To this end, the broker uses internally different queues (`workers[]` and `clients[]`) for different job types.

3.1.2 Solution

Let us present in the following paragraphs a simplified solution for this extension. Instead of managing command-line arguments (as stated above), we assume a constant type (e.g., 'B') for both clients and workers.

The modifications to be applied onto the original files are minimal. In the broker, we must replace the original queues with arrays of queues, using the last message segment (`classID`) for choosing the adequate array element (i.e., the adequate queue).

[6] It is convenient to start each instance in a different window. Thus, their output can be clearly shown and read.

3.1.2.1 myclient1.js

```
01:  // myclient1 in NodeJS, classID='B'
02:  var zmq       = require('zmq')
03:    , requester = zmq.socket('req');
04:
05:  var brokerURL = 'tcp://localhost:8061';
06:  var myID = 'NONE';
07:  var myMsg = 'Hello';
08:  var classID = 'B';
09:
10:  if (myID != 'NONE')
11:    requester.identity = myID;
12:  requester.connect(brokerURL);
13:  console.log('Client (%s), class (%s) connected to %s', myID, classID,
     brokerURL)
14:
15:  requester.on('message', function(msg) {
16:    console.log('Client (%s) has received reply "%s"', myID, msg.toString());
17:    process.exit(0);
18:  });
19:  requester.send([myMsg, classID]);
```

3.1.2.2 myworker1.js

```
01:  // myworker1 server in NodeJS, classID='B'
02:  var zmq = require('zmq')
03:    , responder = zmq.socket('req');
04:
05:  var backendURL = 'tcp://localhost:8062';
06:  var myID = 'NONE';
07:  var connText = 'id';
08:  var replyText = 'World';
09:  var classID = 'B';
10:
11:  if (myID != 'NONE')
12:    responder.identity = myID;
13:  responder.connect(backendURL);
14:  responder.on('message', function(client, delimiter, msg) {
15:      setTimeout(function() {
16:      responder.send([client,'',replyText, classID]);
17:      }, 1000);
18:  });
19:  responder.send([connText, classID]);
```

3.1.2.3 mybroker1.js

```
01:  // ROUTER-ROUTER request-reply broker in NodeJS with classIDs
02:  var zmq      = require('zmq')
03:    , frontend = zmq.socket('router')
04:    , backend  = zmq.socket('router');
05:
06:  var fePortNbr = 8061;
07:  var bePortNbr = 8062;
08:  var workers = [];
09:  var clients = [];
10:  const classIDs = ['R', 'G', 'B']
11:  for (var i in classIDs){
12:     workers[classIDs[i]]=[];
13:     clients[classIDs[i]]=[];
14:  }
15:
16:  frontend.bindSync('tcp://*:'+fePortNbr);
17:  backend.bindSync('tcp://*:'+bePortNbr);
18:
19:  frontend.on('message', function() {
20:    var args = Array.apply(null, arguments);
21:    var classID = args.pop();
```

```
22:     if (workers[classID].length > 0) {
23:       var myWorker = workers[classID].shift();
24:       var m = [myWorker,''].concat(args);
25:       backend.send(m);
26:     } else
27:       clients[classID].push( {id: args[0],msg: args.slice(2)});
28:   });
29:
30:   function processPendingClient(workerID, classID) {
31:     if (clients[classID].length>0) {
32:       var nextClient = clients[classID].shift();
33:       var m = [workerID,'',nextClient.id,''].concat(nextClient.msg);
34:       backend.send(m);
35:       return true;
36:     } else
37:       return false;
38:   }
39:
40:   backend.on('message', function() {
41:     var args = Array.apply(null, arguments);
42:     var classID = args.pop();
43:     if (args.length == 3) {
44:       if (!processPendingClient(args[0], classID))
45:         workers[classID].push(args[0]);
46:     } else {
47:         var workerID = args[0];
48:         args = args.slice(2);
49:         frontend.send(args);
50:         if (!processPendingClient(workerID, classID))
51:           workers[classID].push(workerID);
52:     }
53:   });
```

This shown basic solution may be further extended; e.g., using command-line arguments for specifying the job types, adding the verbose mode... However, the most interesting activities are:

1. Let us assume that the amount and identifiers of the job types are unknown at program development time. Modify the broker program for using one of these two alternatives:
   - Job types are specified as arguments when the broker is started. Thus, a command like `node broker A B C D E F` generates a broker with 6 queues; one per job type (the first for 'A', the second for 'B', and so on).
   - Job types are dynamically created. Each time a previously unknown type is seen in a client or worker message, the needed queues for it are created. These queues are never removed (even when they become empty).
2. Explain, without implementing any program, the key implementation aspects for allowing a worker to manage several types of job.

3.2 Failure detection

The goal in this extension is that the broker detects the failure of any of its attached workers. Thus, it may hide that failure to the clients and it may reassign the request being served by a

crashed worker to other workers. In this way, we may build a simple fault tolerance mechanism, managed by the broker.

A feasible first solution may consist in using another ROUTER socket at the broker and a second REQ socket in workers. Those new sockets are only intended for implementing a heartbeat monitoring mechanism. To this end, the broker sends **periodically** heartbeat messages to each worker using this new ROUTER socket. Workers reply to the broker using their new REQ socket.

- If the broker does not receive that reply, it considers that such worker has failed.
- Workers must be extended, using the adequate code fragment for replying to heartbeat messages.
- This is an expensive solution, since many messages must be exchanged between broker and workers, but it is safe (failures are detected soon, and independently on the workload level; i.e., even when no clients are interacting with those workers).

However, there is a simpler solution: to rely in the default replies to client request messages. If a client is able to send those replies in a reasonable time, it is certainly alive.

- This might be complemented with explicit heartbeat messages when there is no client activity.
- It is similar to first solution described above, but it saves many messages.

Let us describe the easiest strategy: to monitor the time taken by each worker for replying to its current request. If that time exceeds a given limit, the broker considers that it has failed.

- It is a light alternative. It allows the recovery of that request: resending it to another worker.
- Only the broker needs to be extended.

3.2.1 Solution keys

Each time the broker forwards a client request to a worker, it saves that request and sets a timeout (answerInterval) for that worker. If no reply is returned in that interval, the broker considers that such worker has failed and it forwards again the same request to another available worker, if any.

3.2.2 Solution

The next box shows the original broker code with the extensions being needed for detecting worker failures.

A sendToWorker() function has been added. It wraps the backend.send() call, writing down which is the assigned worker, which message has been sent to it, and starting its reply timeout. Besides, another sendRequest() function exists. It calls internally to sendToWorker(), once it has chosen an available worker.

A few other lines have been used for managing the timeout and resending the message.

```
01:   // mybroker2.js: ROUTER-ROUTER request-reply broker in NodeJS
02:   // Worker availability-aware variant.
```

```
03:  // It interacts with the other original agents: myclient.js and myworker.js
04:  var zmq      = require('zmq')
05:    , frontend = zmq.socket('router')
06:    , backend  = zmq.socket('router');
07:
08:  var fePortNbr = 8059;
09:  var bePortNbr = 8060;
10:
11:  var workers = [];
12:  var clients = [];
13:
14:  // Reply awaiting period, in ms.
15:  const answerInterval = 2000;
16:  // Array of busy workers. Each slot contains an array with all
17:  // the segments being needed for resending the current message
18:  // being processed by that worker.
19:  var busyWorkers = [];
20:
21:  // Send a message to a worker.
22:  function sendToWorker(msg) {
23:    var myWorker = msg[0];
24:    // Send the message.
25:    backend.send(msg);
26:    // Initialise busyWorkers slot object.
27:    busyWorkers[myWorker] = {};
28:    // Recall that such message has been sent.
29:    busyWorkers[myWorker].msg=msg.slice(2);
30:    // Set a timeout of its response.
31:    busyWorkers[myWorker].timeout=
32:      setTimeout(generateTimeoutHandler(myWorker),answerInterval);
33:  }
34:
35:  // Function that sends a message to a worker, or
36:  // holds the message if no worker is available now.
37:  // Parameter 'args' is an array of message segments.
38:  function sendRequest(args) {
39:    if (workers.length > 0) {
40:      var myWorker = workers.shift();
41:      var m = [myWorker,''].concat(args);
42:      sendToWorker(m);
43:    } else {
44:      clients.push( {id: args[0],msg: args.slice(2)});
45:    }
46:  }
47:
48:  // Function that creates the handler for a reply
49:  // timeout.
50:  function generateTimeoutHandler(workerID) {
51:    return function() {
52:      // Get the message to be resent.
53:      var msg = busyWorkers[workerID].msg;
54:      // Remove that slot from the busyWorkers array.
55:      delete busyWorkers[workerID];
56:      // Resend that message.
57:      sendRequest(msg);
58:    }
59:  }
60:
61:  frontend.bindSync('tcp://*:'+fePortNbr);
62:  backend.bindSync('tcp://*:'+bePortNbr);
63:
64:  frontend.on('message', function() {
65:    var args = Array.apply(null, arguments);
66:    sendRequest(args);
67:  });
68:
```

```
69:   function processPendingClient(workerID) {
70:     if (clients.length>0) {
71:       var nextClient = clients.shift();
72:       var msg = [workerID,'',nextClient.id,''].concat(nextClient.msg);
73:       sendToWorker(msg);
74:       return true;
75:     } else
76:     return false;
77:   }
78:
79:   backend.on('message', function() {
80:     var args = Array.apply(null, arguments);
81:     if (args.length == 3) {
82:       if (!processPendingClient(args[0]))
83:         workers.push(args[0]);
84:     } else {
85:       var workerID = args[0];
86:       // Cancel the reply timeout.
87:       clearTimeout(busyWorkers[workerID].timeout);
88:       args = args.slice(2);
89:       frontend.send(args);
90:     if (!processPendingClient(workerID))
91:       workers.push(workerID);
92:     }
93:   });
```

3.2.3 Tests

In order to check that these extensions are correct, we need to compare the behaviour of the resulting program (*mybroker2.js*) with that of the original broker. To this end, let us start an instance of *mybroker.js* and three *myworker_vp.js* processes.

```
$ node mybroker &
[1] 12701
$ node myworker_vp 'tcp://*:8060' Worker1 id Answer1 &
[2] 12703
$ node myworker_vp 'tcp://*:8060' Worker2 id Answer2 &
[3] 12704
$ node myworker_vp 'tcp://*:8060' Worker3 id Answer3 &
[3] 12705
```

Later, we eliminate the first of those workers using the **kill** command, with its PID as an argument (in this example, that PID is shown after [2] and its value is 12703):

```
$ kill 12703
```

Now, we start three client processes as follows:

```
$ node myclient & node myclient & node myclient &
```

Let us write down the output of that execution. How many replies have been shown? Which are the workers that have generated them? (You may use the command **ps –o pid,args** in order to find out which processes are still alive in your terminal).

Kill all node processes in that execution using this command:

```
$ killall node
```

Now, let us repeat that execution but using the extended broker instead. So, the first command should be this instead of **node mybroker &**:

```
$ node mybroker2 &
```

Then, let us continue starting three workers, killing the first of them. Once we have started our three clients... What is the output shown in this second execution? How many answers have been received by these clients? How many workers have sent them? Why? Is now any pending client still running?

When we use *mybroker2.js*, if we assume that workers are replicas of the same service and every worker generates the same reply for the same request... do we achieve replication transparency? Do we provide failure transparency?

3.3 Load balancing

3.3.1 Problem keys

In this extension, the broker must use a load balancing criterion based on the current workload at each worker.

- There is a function that returns the workload level at the local machine, ***getLoad***. Its code is in the *auxfunctions1718.js* module (Appendix 1).
- Besides, a similar problem was already described in part 3 of the first lab project (final application: reverse TCP/IP proxy). Revise it, considering that the remote services shown in that example are now workers.

3.3.2 Solution keys

The client program is not changed in this extension. The broker needs now a data structure that stores the identities of all workers, and their availability.

The worker identity must be chosen by the broker from that list of available workers. Instead of a FIFO order, it must use now a load balancing criterion. This identity is needed for specifying which backend ROUTER connection is used at request forwarding time.

Ideally, the broker should be always aware on the workload level at each worker. However, this might imply a periodical workload level reporting from every worker. The management of those reporting messages could overload the broker. Therefore, instead of a continuous reporting, it will be enough if each time a request needs to be forwarded, the broker **has a general idea about the most recent workload level for each available worker**. Thus:

- Workers do not need to report anything while they are serving a request. They are not available then.
- Available workers do not modify their workloads while they are waiting for the next request. So, their last reported workload level is still valid.

Therefore, each worker must report its current workload level when it notifies that it is available. This simplifies our implementation.

Thus, the solution keys are:

- Workers report their effective workload level in every message they send to the broker. The broker uses that information for implementing its worker election criterion.

- Each time the broker should choose a worker, it elects that with the minimal workload, forwarding the request to that worker and removing it from the list of available workers.

There are two basic alternatives for implementing in an efficient way an ordered list of workers. They are:

a) Keep an unordered array, inserting new elements at an arbitrary position (either the first or the last one), spending time for finding and extracting the appropriate worker.
b) Keep the array in an ordered way. This demands time when elements are inserted, but allows a fast election and extraction.

In *auxfunctions1718.js* there are some functions for managing an ordered list as a mix of both alternatives. They are: *orderedList* (list creation), *insert* (adds an element), *lowest* (returns and extracts the minimum element). They are based on arrays and on the *sort* method for JavaScript arrays (https://www.w3schools.com/jsref/jsref_sort.asp).

- There is a flag (*ordered*) that is set to **false** each time an element is inserted, and to **true** each time the list is ordered. Its *lowest()* method checks whether *ordered* is already **true** (calling *sort()* otherwise) and returns the first element afterwards.

Please, develop new versions of the worker (*lbworker*) and the broker (*lbbroker*) programs in order to solve this load balancing problem. Verify that your implemented agents implement a correct solution! Check the output being provided by each agent and show the workload being reported by each worker!

Use the verbose mode!

4 ANNEX 1. AUXILIARY FUNCTIONS

4.1.1 auxfunctions1718.js

```
01:   //auxfunctions1718
02:
03:     // *** getLoad function
04:
05:   function getLoad() {
06:     var fs     = require('fs')
07:       , data   = fs.readFileSync("/proc/loadavg") // synchronous version
08:       , tokens = data.toString().split(' ')
09:       , min1   = parseFloat(tokens[0])+0.01
10:       , min5   = parseFloat(tokens[1])+0.01
11:       , min15  = parseFloat(tokens[2])+0.01
12:       , m      = min1*10 + min5*2 + min15;
13:     return m;
14:   }
15:
16:     // *** randNumber function
17:
18:   function randNumber(upper, extra) {
19:     var num = Math.abs(Math.round(Math.random() * upper));
20:     return num + (extra || 0);
21:   }
22:
23:     // *** randTime function
24:
25:   function randTime(n) {
26:     return Math.abs(Math.round(Math.random() * n)) + 1;
27:   }
28:
29:     // *** showMessage function
30:
31:   function showMessage(msg) {
32:     msg.forEach( (value,index) => {
33:       console.log( '     Segment %d: %s', index, value );
34:     })
35:   }
36:
37:     // *** ordered list functions for workers management
38:     // list has an "ordered" property (true|false) and a data property (array)
39:     // data elements consists of pairs {id, load}
40:
41:   function orderedlist() {
42:     return {ordered: true, data: []};
43:   }
44:
45:   function nonempty() {
46:     return (this.data.length > 0);
47:   }
48:
49:   function lowest() {  // ordered reads
50:     if (!this.ordered) {
51:       this.data.sort(function(a, b) {
52:         return parseFloat(b.load) - parseFloat(a.load);
53:       }
54:     }
55:     this.ordered = true;
56:     return this.data.shift();
57:   }
58:
59:   function insert(k) { // unordered writes
60:     this.ordered = false;
61:     this.data.push(k);
```

```
62:     }
63:
64:     module.exports.getLoad = getLoad;
65:     module.exports.randString = randString;
66:     module.exports.randNumber = randNumber;
67:     module.exports.randTime = randTime;
68:     module.exports.showMessage = showMessage;
69:     module.exports.orderedlist = orderedList;
70:     module.exports.nonempty = nonempty;
71:     module.exports.lowest = lowest;
72:     module.exports.insert = insert;
```

5 ANNEX 2. THE VERBOSE MODE

Let us show in these traces some examples where either workers or clients must wait because there are no pending requests (in case of new available workers) or no available workers (in case of a new received request).

These executions are:

- Arrival order: broker, client1, client2, client4, client3, worker2, worker1 (***bc1c2c4c3w2w1***). Clients must wait.
- Arrival order: broker, worker1, worker2, client1, client2, client3, client4 (***bw1w2c1c2c3c4***). Workers must wait.

5.1.1 Verbose execution of mybroker_vp.js

```
node mybroker_vp.js 8059 8060 verbose
```

Order *bc1c2c4c3w2w1*, broker screen

```
broker: frontend-router listening on tcp://*:8059
broker: backend-router listening on tcp://*:8060
Received request: "Hello1" from client (Client1).
     Segment 0: Client1
     Segment 1:
     Segment 2: Hello1
Pushing client (Client1) to clients' queue (size:
1).
Received request: "Hello2" from client (Client2).
     Segment 0: Client2
     Segment 1:
     Segment 2: Hello2
Pushing client (Client2) to clients' queue (size:
2).
Received request: "Hello4" from client (Client4).
     Segment 0: Client4
     Segment 1:
     Segment 2: Hello4
Pushing client (Client4) to clients' queue (size:
3).
Received request: "Hello3" from client (Client3).
     Segment 0: Client3
     Segment 1:
     Segment 2: Hello3
Pushing client (Client3) to clients' queue (size:
4).
Received backend request: "id" from worker
(Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: id
Sending client (Client1) request to worker
(Worker2) through backend.
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client1
     Segment 3:
     Segment 4: Hello1
Received backend request: "id" from worker
(Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: id
Sending client (Client2) request to worker
(Worker1) through backend.
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client2
     Segment 3:
     Segment 4: Hello2
Received reply: "World2" from worker (Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client1
     Segment 3:
     Segment 4: World2
Sending worker (Worker2) reply to client (Client1)
through frontend.
     Segment 0: Client1
     Segment 1:
     Segment 2: World2
Sending client (Client4) request to worker
(Worker2) through backend.
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client4
     Segment 3:
     Segment 4: Hello4
Received reply: "World1" from worker (Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client2
     Segment 3:
     Segment 4: World1
Sending worker (Worker1) reply to client (Client2)
through frontend.
     Segment 0: Client2
     Segment 1:
     Segment 2: World1
Sending client (Client3) request to worker
(Worker1) through backend.
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client3
     Segment 3:
     Segment 4: Hello3
Received reply: "World2" from worker (Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client4
     Segment 3:
     Segment 4: World2
Sending worker (Worker2) reply to client (Client4)
through frontend.
     Segment 0: Client4
     Segment 1:
```

```
     Segment 2: World2
Pushing worker (Worker2) to workers' queue (size:
1).
Received reply: "World1" from worker (Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client3
     Segment 3:
     Segment 4: World1
Sending worker (Worker1) reply to client (Client3)
through frontend.
     Segment 0: Client3
     Segment 1:
     Segment 2: World1
Pushing worker (Worker1) to workers' queue (size:
2).
```

Order *bw1w2c1c2c3c4*, broker screen

```
broker: frontend-router listening on tcp://*:8059
broker: backend-router listening on tcp://*:8060
Received backend request: "id" from worker
(Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: id
Pushing worker (Worker1) to workers' queue (size:
1).
Received backend request: "id" from worker
(Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: id
Pushing worker (Worker2) to workers' queue (size:
2).
Received request: "Hello1" from client (Client1).
     Segment 0: Client1
     Segment 1:
     Segment 2: Hello1
Sending client (Client1) request to worker
(Worker1) through backend.
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client1
     Segment 3:
     Segment 4: Hello1
Received reply: "World1" from worker (Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client1
     Segment 3:
     Segment 4: World1
Sending worker (Worker1) reply to client (Client1)
through frontend.
     Segment 0: Client1
     Segment 1:
     Segment 2: World1
Pushing worker (Worker1) to workers' queue (size:
2).
Received request: "Hello2" from client (Client2).
     Segment 0: Client2
     Segment 1:
     Segment 2: Hello2
Sending client (Client2) request to worker
(Worker2) through backend.
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client2
     Segment 3:
     Segment 4: Hello2
Received reply: "World2" from worker (Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client2
     Segment 3:
     Segment 4: World2
Sending worker (Worker2) reply to client (Client2)
through frontend.
     Segment 0: Client2
     Segment 1:
     Segment 2: World2
Pushing worker (Worker2) to workers' queue (size:
2).
Received request: "Hello3" from client (Client3).
     Segment 0: Client3
     Segment 1:
     Segment 2: Hello3
Sending client (Client3) request to worker
(Worker1) through backend.
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client3
     Segment 3:
     Segment 4: Hello3
Received reply: "World1" from worker (Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client3
     Segment 3:
     Segment 4: World1
Sending worker (Worker1) reply to client (Client3)
through frontend.
     Segment 0: Client3
     Segment 1:
     Segment 2: World1
Pushing worker (Worker1) to workers' queue (size:
2).
Received request: "Hello4" from client (Client4).
     Segment 0: Client4
     Segment 1:
     Segment 2: Hello4
Sending client (Client4) request to worker
(Worker2) through backend.
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client4
     Segment 3:
     Segment 4: Hello4
Received reply: "World2" from worker (Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client4
     Segment 3:
     Segment 4: World2
Sending worker (Worker2) reply to client (Client4)
through frontend.
     Segment 0: Client4
     Segment 1:
     Segment 2: World2
Pushing worker (Worker2) to workers' queue (size:
2).
```

5.1.2 Verbose execution of myworker_vp.js

```
node myworker_vp.js tcp://localhost:8060 Worker1 id World1 verbose
```

Order *bc1c2c4c3w2w1*, worker1 screen

```
Worker (worker1) connected to tcp://localhost:8060
Worker (worker1) has sent its first connection message: "id"
Worker (worker1) has received request "Hello2" from client (Client2)
Worker (worker1) has sent its reply "World1"
Worker (worker1) has received request "Hello3" from client (Client3)
Worker (worker1) has sent its reply "World1"
```

Order *bw1w2c1c2c3c4*, worker1 screen

```
Worker (worker1) connected to tcp://localhost:8060
Worker (worker1) has sent its first connection message: "id"
Worker (worker1) has received request "Hello1" from client (Client1)
Worker (worker1) has sent its reply "World1"
Worker (worker1) has received request "Hello3" from client (Client3)
Worker (worker1) has sent its reply "World1"
```

6 ANNEX 3: SOURCE CODE

6.1 Commented source code for all components

6.1.1 myclient.js

```
01:  // myclient in NodeJS
02:  // By default:
03:  // - It connects REQ socket to tcp://localhost:8059
04:  // - It sends "Hello" to server and expects "World" back
05:  // Using the command line arguments, that default behaviour
06:  // may be changed. To this end, those arguments are interpreted
07:  // as follows:
08:  // - 1st: URL of the broker frontend socket.
09:  // - 2nd: Client ID, to tag the connection to the frontend router
10:  //        socket of the broker.
11:  // - 3rd: String to be sent to the servers (i.e., worker agents).
12:
13:  var zmq       = require('zmq')
14:    , requester = zmq.socket('req');
15:
16:  // Command-line arguments.
17:  var args = process.argv.slice(2);
18:  // Get those arguments.
19:  var brokerURL = args[0] || 'tcp://localhost:8059';
20:  var myID = args[1] || 'NONE';
21:  var myMsg = args[2] || 'Hello';
22:
23:  // Set the connection ID. This must be done before the
24:  // connect() method is called.
25:  if (myID != 'NONE')
26:        requester.identity = myID;
27:  // Connect to the frontend ROUTER socket of the broker.
28:  requester.connect(brokerURL);
29:  // Print trace information.
30:  console.log('Client (%s) connected to %s', myID, brokerURL)
31:
32:  // A single reply is expected...
33:  requester.on('message', function(msg) {
34:    console.log('Client (%s) has received reply "%s"', myID, msg.toString());
35:    process.exit(0);
36:  });
37:
38:  // Send a single message.
39:  requester.send(myMsg);
40:  // Print trace information.
41:  console.log('Client (%s) has sent its message: "%s"', myID, myMsg);
```

6.1.2 myworker.js

```
01:  // myworker server in NodeJS
02:  // By default:
03:  // - It connects its REQ socket to tcp://*:8060
04:  // - It expects a request message from client, and it replies with "World"
05:  // Multiple non-mandatory command-line arguments can be provided in
06:  // the following order:
07:  // - 1st: URL of the broker backend socket.
08:  // - 2nd: Worker identifier (a string).
09:  // - 3rd: Text to be used in its initial connection message.
10:  // - 4th: Text to be used in the replies to be returned to clients.
```

```
11:
12:   var zmq = require('zmq')
13:     , responder = zmq.socket('req');
14:
15:   // Get the command-line arguments, if any.
16:   var args = process.argv.slice(2);
17:   var backendURL = args[0] || 'tcp://localhost:8060';
18:   var myID = args[1] || 'NONE';
19:   var connText = args[2] || 'id';
20:   var replyText = args[3] || 'World';
21:
22:   // Set the worker identity to the connection.
23:   // Note that a random ID is assigned by default.
24:   // Thus, when no command-line arguments are given,
25:   // no identity must be set.
26:   if (myID != 'NONE')
27:         responder.identity = myID;
28:   // Connect to the broker backend socket.
29:   responder.connect(backendURL);
30:
31:   // Process each incoming request.
32:   responder.on('message', function(client, delimiter, msg) {
33:         setTimeout(function() {
34:               responder.send([client,'',replyText]);
35:         }, 1000);
36:   });
37:
38:   // This is the first message sent by a worker.
39:   // Since this "responder" socket has been connected
40:   // to a ROUTER socket, ZeroMQ will prepend the identity
41:   // of the worker to every sent message. That identity
42:   // is used by the broker to "register" this
43:   // worker in its appropriate data structures.
44:   responder.send(connText);
```

6.1.3 mybroker.js

```
01:   // ROUTER-ROUTER request-reply broker in Node.js
02:   // It uses a FIFO policy to administer its pending
03:   // clients and available workers.
04:   // Each kind of agent is placed in its corresponding
05:   // array, that is managed as a queue by default.
06:
07:   // - 1st: Port number for its frontend socket (8059)
08:   // - 2nd: Port number for its backend socket (8060)
09:
10:   var zmq      = require('zmq')
11:     , frontend = zmq.socket('router')
12:     , backend  = zmq.socket('router');
13:
14:   // Get the command-line arguments.
15:   var args = process.argv.slice(2);
16:   // Port number for the frontend socket.
17:   var fePortNbr = args[0] || 8059;
18:   // Port number for the backend socket.
19:   var bePortNbr = args[1] || 8060;
20:
21:   // Array of available workers.
22:   var workers = [];
23:   // Array of pending clients.
24:   var clients = [];
25:
26:   frontend.bindSync('tcp://*:'+fePortNbr);
```

```
27:  backend.bindSync('tcp://*:'+bePortNbr);
28:
29:  frontend.on('message', function() {
30:    // Note that separate message parts come as function arguments.
31:    var args = Array.apply(null, arguments);
32:    // Check whether there is any available worker.
33:    if (workers.length > 0) {
34:      // Remove the oldest worker from the array.
35:      var myWorker = workers.shift();
36:      // Build a multi-segment message.
37:      // & Send it.
38:      var m = [myWorker,''].concat(args);
39:      backend.send(m);
40:    } else
41:      // When no available worker exists, save
42:      // the client ID and message into the clients array.
43:      clients.push( {id: args[0],msg: args.slice(2)});
44:  });
45:
46:  function processPendingClient(workerID) {
47:    // Check whether there is any pending client.
48:    if (clients.length>0) {
49:      // Get the data from the first client and remove it
50:      // from the queue of pending clients.
51:      var nextClient = clients.shift();
52:      // Build the message and send it.
53:      // Note that a message is an array of segments. Therefore,
54:      // we should prepend: (a) the worker ID + a delimiter,
55:      // to choose the worker connection to be used, (b) the client ID +
56:      // a delimiter, to build the needed context for adequately
57:      // forwarding its subsequent reply at the worker domain.
58:      // & Send the message.
59:      var m = . [workerID,'',nextClient.id,''].concat(nextClient.msg);
60:      backend.send(m);
61:      // Return true if any client has been found
62:      return true;
63:    } else
64:      // Return false if no client is there.
65:      return false;
66:  }
67:
68:  backend.on('message', function() {
69:    var args = Array.apply(null, arguments);
70:    // If this is an initial ID message from a new worker,
71:    // save its identity in the "workers" array...
72:    if (args.length == 3) {
73:      // It may happen that some clients are already waiting
74:      // for available workers. If so, serve one of them!
75:      if (!processPendingClient(args[0]))
76:        // Otherwise, save this worker as an available one.
77:        workers.push(args[0]);
78:    // ...otherwise, return the message to the appropriate
79:    // client.
80:    } else {
81:    // Save the worker identity.
82:    var workerID = args[0];
83:    // Remove the first two slots (id+delimiter)
84:    // from the array.
85:    args = args.slice(2);
86:    // Send the resulting message to the appropriate
87:    // client. Its first slot holds the client connection
88:    // ID.
89:    frontend.send(args);
90:    // Check whether any client is waiting for attention.
91:    // If so, the request from the first one is sent and
```

```
92:     // it is remove from the pending queue.
93:     // Otherwise, a false value is returned.
94:     if (!processPendingClient(workerID))
95:       // In that case, the worker ID is saved in the
96:       // queue of available workers.
97:       workers.push(workerID);
98:     }
99:   });
```