TSR – 10th November 2017. EXERCISE 3

Please implement in NodeJS and ØMQ two programs (client.js and server.js) with all these requirements:

1) The client program must receive two command line arguments. The first argument specifies the interval (in seconds) for its periodical messages, and the second one is the string to be sent in each message. Those messages must append to that string the current message number.
2) The server should return a reply (a message with the ‘Ok’ string) to every received message. It should also print the request message to the screen.
3) Every 5 seconds, the server broadcasts a message to all its clients, reporting the identities of every client and the amount of requests that the server has received from each of them.
4) Client programs show all its received messages (both replies and reporting messages).
5) Clients and server run in the same computer.
6) You must decide how many ØMQ sockets are needed in each program and their type.
7) Client identities may be either (choose what you prefer):
   a) passed from the command line, or...
   b) generated by the client program using a random number generator (e.g., the Math.random() function that returns a float number in the 0..1 range. With a statement like this: Math.round(Math.random()*1000000) you may obtain a large integer number), with some string prefix, or...
   c) taken from the connection identifier if the server uses ROUTER sockets.

Execution example:

```
$ node server &
$ node client 3 “My message “ &
$ node client 2 “Another client “ &
$ Another client 1
Ok
My message 1
Ok
Another client 2
Ok
Messages:
+ Client RVXS1df5: 2 messages
+ Client JKS13LjV: 1 messages
Messages:
+ Client RVXS1df5: 2 messages
+ Client JKS13LjV: 1 messages
Another client 3
Ok
My message 2
Ok
...
```

SOLUTION

Perhaps, the simplest solution could be based on these features:

- The server uses a REP socket and clients use REQ sockets. Client identities are passed as a message segment.
- Clients create their identities using a random number generator.
- Periodical reports are sent by the server using a PUB socket. Clients need a SUB socket (subscribed to everything) in order to get those reports.
- The server collects its statistics in an object that has as its "properties" the identifiers of each client. The values of those properties are the amount of requests received from each client. This object is "JSON-stringified" when it is broadcast and "JSON-parsed" by the receiving clients when they show the statistics report forwarded by the server.

In that case, the code to be used is:

```
// Client variant that uses a REQ socket for transmitting
// its requests and receiving its replies, plus a SUB socket
// for receiving periodical report messages from the server.
// The identity is randomly generated.
const zmq=require('zmq');
var req=zmq.socket('req');
var sub=zmq.socket('sub');
function errorHandler(err) {
        console.log(err);
        process.exit(1);
}
// Connect both sockets to their intended counterparts.
req.connect('tcp://localhost:8000', errorHandler);
sub.connect('tcp://localhost:8001', errorHandler);
var id='client'+Math.round(Math.random()*10000);
var period=parseInt(process.argv[2]) || 2;
var message=process.argv[3] || 'Message '+id;
var msgCounter=1;

// Function that sends the client request.
function sendMessage() {
        // The message consists of two segments:
        // - The client ID.
        // - The message. Its suffix is autoincreased.
        req.send([id,message+' '+msgCounter++]);
}
// Send the message periodically,
setInterval(sendMessage,period*1000);
// Show replies on screen.
req.on('message', function(reply) {
        console.log(reply+'');
});
// Subscribe to everything.
sub.subscribe('');
// Receive and show the reporting messages.
sub.on('message', function(data) {
        var report = JSON.parse(data+'');
        // Show a header.
        console.log("Messages:");
        // Show the message contents.
        for(var i in report)
                console.log("+ Client %s: %d messages", i,
                        report[i]);
});
```

```
// Server.js
const zmq=require('zmq');
var rep=zmq.socket('rep');
var pub=zmq.socket('pub');
// "Array" of message counters, indexed with the client IDs.
var messageCounters = {};

// Bind those sockets.
rep.bindSync('tcp://127.0.0.1:8000');
pub.bindSync('tcp://127.0.0.1:8001');
// Process incoming requests.
rep.on('message', function(clientID,msg) {
        // The client ID is in the first segment.
        cID = clientID+'';
        // The received message is shown.
        console.log(msg+'');
        // A reply is sent.
        rep.send('Ok');
        // Test whether any previous message from
        // that client had been already received.
        if (!messageCounters[cID])
                // If not, set its counter to zero.
                messageCounters[cID]=0;
        // Increase the counter of messages for that
        // client.
        messageCounters[cID]++;
});
// Function that sends the report message.
function sendReport() {
        pub.send(JSON.stringify(messageCounters));
}
// Send the report message every 5 seconds.
setInterval(sendReport,5000);
```

But there are many other possible solutions. For instance, the server may use a ROUTER socket in order to automatically handle client identities. In that case, a valid solution is similar to this one:

```
// Client variant that uses a REQ socket for transmitting
// its requests and receiving its replies, plus a SUB socket
// for receiving periodical report messages from the server.
// The server uses a ROUTER socket for interacting with the
// client's REQ.
// Thus, no identity needs to be generated by the client.
const zmq=require('zmq');
var req=zmq.socket('req');
var sub=zmq.socket('sub');
function errorHandler(err) {
        console.log(err);
        process.exit(1);
}
// Connect both sockets to their intended counterparts.
req.connect('tcp://localhost:8000', errorHandler);
sub.connect('tcp://localhost:8001', errorHandler);
var period=parseInt(process.argv[2]) || 2;
var message=process.argv[3] || 'Message '+id;
var msgCounter=1;

// Function that sends the client request.
```

```
// Server.js
// ROUTER-based variant.
const zmq=require('zmq');
var rep=zmq.socket('router');
var pub=zmq.socket('pub');
// "Array" of message counters, indexed with the client IDs.
var messageCounters = {};

// Bind those sockets.
rep.bindSync('tcp://127.0.0.1:8000');
pub.bindSync('tcp://127.0.0.1:8001');
// Process incoming requests.
rep.on('message', function(clientID,del,msg) {
        // The client ID is in the first segment.
        cID = clientID+'';
        // The received message is shown.
        console.log(msg+'');
        // A reply is sent.
        rep.send([clientID,'','Ok']);
        // Test whether any previous message from
        // that client had been already received.
```

```
function sendMessage() {
        req.send(message+' '+msgCounter++);
}
// Send the message periodically,
setInterval(sendMessage,period*1000);
// Show replies on screen.
req.on('message', function(reply) {
        console.log(reply+'');
});
// Subscribe to everything.
sub.subscribe('');
// Receive and show the reporting messages.
sub.on('message', function(data) {
        var report = JSON.parse(data+'');
        // Show a header.
        console.log("Messages:");
        // Show the message contents.
        for(var i in report)
                console.log("+ Client %s: %d messages", i,
                        report[i]);
});
```

```
        if (!messageCounters[cID])
                // If not, set its counter to zero.
                messageCounters[cID]=0;
        // Increase the counter of messages for that
        // client.
        messageCounters[cID]++;
});
// Function that sends the report message.
function sendReport() {
        pub.send(JSON.stringify(messageCounters));
}
// Send the report message every 5 seconds.
setInterval(sendReport,5000);
```

There are several other valid solutions. These ones have only been shown as possible examples.