TSR: 3rd Exercise

These programs are a client and a server. The server receives an argument from the command line and returns a reply that consists of two segments. The first one carries the request contents and the second segment provides the actual answer. That answer is the request value multiplied by the value passed from the command line. Once the client receives each reply, it shows its message contents.

Act3-client.js

```
var zmq = require('zmq');
var req = zmq.socket('req');
var counter = 1;
const interval = 1000;

req.bindSync('tcp://*:8000');

function sendMsg() {
  req.send(counter++);
}

function handler(segm1,segm2) {
  console.log('Request: %d, reply: %s',
             segm1,segm2);
  if (segm1>99) process.exit(0);
}

req.on('message', handler);

setInterval(sendMsg, interval);
```

Act3-server.js

```
var zmq = require('zmq');
var rep = zmq.socket('rep');
var args = process.argv.slice(2);
if (args.length < 1) {
  console.error('Please provide an argument!');
  process.exit(1);
}
rep.connect('tcp://127.0.0.1:8000');

rep.on('message', function(msg) {
  rep.send([msg,msg*args[0]]);
});
```

Please, answer the following questions about these programs in a separate sheet, justifying your answers.

1. Justify whether these programs implement in a correct way what has been described in the wording. If any of them is wrong, please explain which of its sentences are incorrect and provide a correct version of them. (1 point)
2. Explain how many messages are sent by a client and at which sending rate (i.e., amount of messages per time unit). (1 point)
3. Explain whether there is any limit on the number of clients that can be started. (1 point)
4. Explain whether there is any limit on the amount of servers that can be started. (1 point)
5. Explain whether, with the code presented above, clients and servers may be placed in different computers. (1 point)
6. Explain what will be printed by the client in its first 10 seconds of execution if this sequence of actions occurs in a given computer: (1) A client is started at time 0 sec.; (2) at time 2.9 sec., a server is started using "**node Act3-server 2**"; (3) at time 5.2 sec., a second server is started using "**node Act3-server 5**". (3 points)
7. Explain how multiple servers may be started in order to generate the following client output (2 points):

```
Request: 1, reply: 3
Request: 2, reply: 6
Request: 3, reply: 9
Request: 4, reply: 8
Request: 5, reply: 25
Request: 6, reply: 18
Request: 7, reply: 14
Request: 8, reply: 40
Request: 9, reply: 27
Request: 10, reply: 20
...
```

3rd Exercise. Solution

Solutions

1. Both programs correctly develop what has been described in the wording. Act3-server.js copies received arguments in the **args** array. The block in lines 4 to 7 checks whether at least an argument has been received and shows an error message otherwise. In that block, using **process.exit(1)**, the process is terminated. The answer to be returned by the server is built in the last program statement. There, an array with two slots is passed as an argument to the **send()** call. Each slot is a message segment. The first segment is the value transmitted in the request and the second segment is obtained multiplying such first segment by the value received as a command line argument. Client behaviour is described in the next answer. It also complies with what has been requested in the wording.
2. The last sentence in the client program states that function **sendMsg** will be invoked once per second (i.e., the message sending rate is 1 msg/sec). In **sendMsg** a counter is increased in each call. That same value, before being increased, is sent as the request contents to the server. Later, in function **handler**, a message is shown, reporting which has been the sent value and that returned as its reply. If the value of the first received segment exceeds 99, the client is terminated. Since both programs use the REQ-REP communication pattern, this means that the client generates 100 messages and, once this is done, it ends.
3. This client uses **bindSync** passing a constant port number (8000). Since two consecutive **bind** or **bindSync** calls using the same port number aren't allowed, this means that a single client may be run.
4. Servers connect to the client URL. Multiple connections to the same port are allowed. This means that there is no upper bound on the number of servers.
5. Client and server cannot be placed in different computers since the server uses a local URL to connect to. Thus, they must be placed in the same computer in order to interact without problems.
6. The client sends 1 msg/sec. This means that at time 1" the first request is sent. At that time, there is no server yet. So, that message is held in the output queue of the REQ socket. The same happens at time 2". At time 2.9", the first server is started. Thus, the first message is taken from the output queue and sent to the server. Since both processes are in the same computer, in a few milliseconds (depending on the load in that computer and the scheduling policy being used by the operating system) its reply will be delivered. At that time, the client shows "**Request: 1, reply: 2**". Once such reply is delivered, the REQ socket is able to send the second pending request. Thus, in a few milliseconds "**Request: 2, reply: 4**" is also shown. Later on, at times 3", 4" and 5" the following messages will be shown: "**Request: 3, reply: 6**", "**Request: 4, reply: 8**", and "**Request: 5, reply: 10**". At time 6" there are two servers connected with this client. Due to this, the socket REQ in the client uses a Round-Robin policy in order to propagate its messages to those servers. Because of this, the messages being printed in seconds 6 to 10 will be: "**Request: 6, reply: 12**" (x2), "**Request: 7, reply: 35**" (x5), "**Request: 8, reply: 16**" (x2), "**Request: 9, reply: 45**" (x5), and "**Request: 10, reply: 20**" (x2).
7. To begin with, we should consider which operand has been used in order to apply the multiplication in each reply. So, we see that messages 1, 2, 3, 6 and 9 have multiplied the request value by 3, while messages 4, 7 and 10 have used 2 as their operand and messages 5 and 8 have used value 5. Therefore, the execution was using three servers in its final steps, but its first three messages were processed by a single server (one that multiplies by 3).
   A way for generating that trace could be:
   - In the interval from 0" to 3.5" a first server was started with "**node Act3-server 3**". It multiplies its requests by 3.

3rd Exercise. Solution

- Before time 4”, but after time 2.1” and once the first server had been started and it processed the first 3 messages, a second server is started using the command “**node Act3-server 2**”. Thus, this second server processes the fourth message sent by the client.
- Before time 5” and once the second server has been started, a third server is generated using “**node Act3-server 5**”.