Laboratory Work #3

on PPC

“Cluster programming”

Done by: Ursu Dumitru
Verified by: Ciorbă Dumitru

Chișinău, 2014

Task:

Using Message Passing Interface to build a cluster-grade application

Domain of the problem:

In the Ruby world, scalability was always a concern, as some huge websites and applications run on top of it (Github.com, twitter.com, basecamp, hulu, scribd, etc.). One way such big application involves a Message Passing interface, or, in the land of Ruby, a Remote Procedure Call (RPC). There are lots of related technologies, like Redis, RabbitMQ, MessagePack. For this laboratory work I decided to stick with the basics, and use the RPC library shipped with the default ruby, XMLRPC.

I used the condition from the laboratory work nr3 for our calls to make something even remotely useful: The teacher now runs on the client, using TCP to communicate with the server, where the 15 students are “born” and take guesses. One thing that I don't like is that the number 15 is hardcoded.

Source code analysis:

```ruby
# THE CLIENT
require "xmlrpc/client"

server = XMLRPC::Client.new("localhost", "/RPC2", 2000)

students_answers = []
15.times do
  ok, param = server.call2("student.call")
  if ok then
    puts param
    students_answers = param
```

Here we call the student.call from the server, and receive a param back, with their answers.

```ruby
  else
    puts "Error:"
    puts param.faultCode
    puts param.faultString
  end
end

def examen(student, true_answer)
  # the student and the "distance" of it's guess are returned
  [student[0], (student[1] - true_answer).abs]
end

def f(k)
  k * k + 20
end
```

```
questions = Array.new 15
questions.map! { rand 100 }

the_register = []
students_answers.each {|answer|
  number = rand 15;
  the_register << examen(answer, f(questions[number]))
}

# sorting the registry
the_register.sort_by!{|e| e[1]}

# and, the best students are...
the_register[0..2].each do |best|
  puts "#{best[0]} : #{best[1]} <- plin de bucurie"
end
```

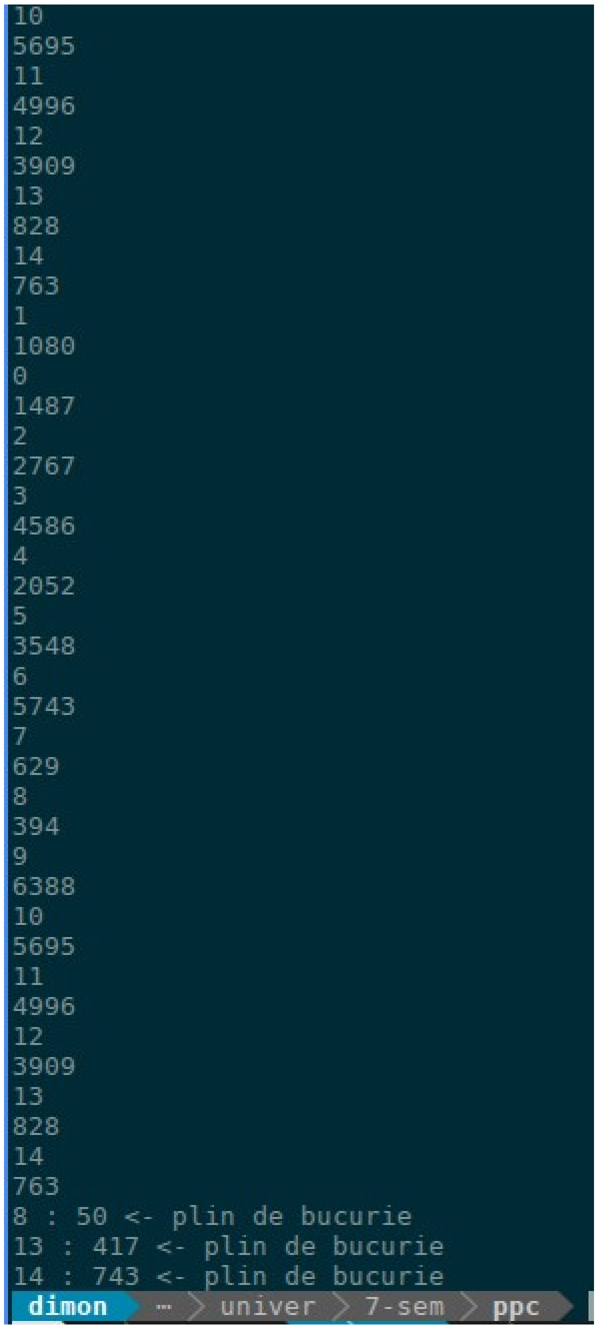
Pretty much all the code is unchagend, with the exception of the place where we get the students answers (from the param that we receive form the server).

```
#THE SERVER
require "xmlrpc/server"
require 'thread'

s = XMLRPC::Server.new(2000)

mutex = Mutex.new
teacher = ConditionVariable.new
$answers = []

students = []
15.times do |n|
  students << Thread.new(n) {
    # one student at a time, please
    mutex.synchronize {
      # wait the teacher
      teacher.wait(mutex)
      $answers << [n, rand(9000)]
    }
  }
end

students.each(&:join)
s.add_handler("student.call") do
  teacher.signal
  $answers
end
```

The last value of this block above is the one that it's returned to the client. The

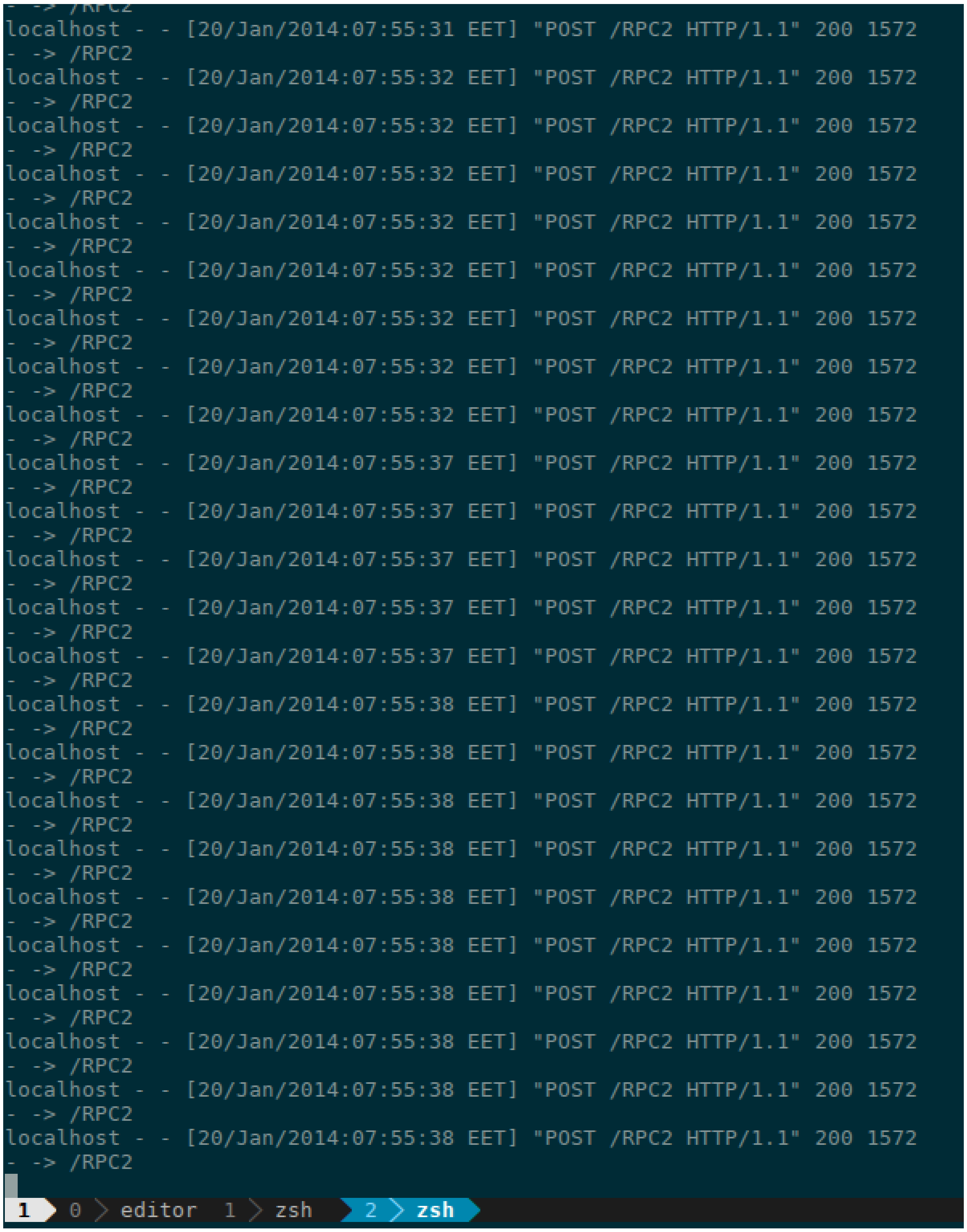
XML communication is transparent to us, see the “params” as and array. This means that we don't have to pre-process (deserialize) the data on the other side.

```
s.set_default_handler do |name, *args|
  raise XMLRPC::FaultException.new(-99, "Method #{name} missing" +
                                   " or wrong number of parameters!")
end
s.serve
```



```
10
5695
11
4996
12
3909
13
828
14
763
1
1080
0
1487
2
2767
3
4586
4
2052
5
3548
6
5743
7
629
8
394
9
6388
10
5695
11
4996
12
3909
13
828
14
763
8 : 50 <- plin de bucurie
13 : 417 <- plin de bucurie
14 : 743 <- plin de bucurie
dimon  …  univer  7-sem  ppc
```

Figure 1. The client window.



```
localhost - - [20/Jan/2014:07:55:31 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:32 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:32 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:32 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:32 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:32 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:32 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:32 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:32 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:37 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:37 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:37 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:37 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:37 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:38 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:38 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:38 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:38 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:38 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:38 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:38 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:38 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:38 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2
localhost - - [20/Jan/2014:07:55:38 EET] "POST /RPC2 HTTP/1.1" 200 1572
- -> /RPC2

 1  0  editor   1  zsh   2  zsh
```

Figure 2. The server output

Conclusion:

Remote procedure call is surpisingly easy with ruby. The standard library makes the communication between the client and the server almost transparent, with little to no modification of the previous source code.

Bibliography:

1. http://www.ruby-doc.org/stdlib-2.1.0/libdoc/xmlrpc/rdoc/XMLRPC/Client.html