Faculty of Computers, Informatics and Microelectronics

Technical University of Moldova

PAD

Laboratory work # 4,5

Processing and distribution of xml and json data.
Validation of xml data

Authors:
Petru Negrei
Victor Vasilica

Supervisor:
D. Ciorba

November 2014

1 Introduction

1.1 Topic

Processing and distribution of xml and json data.

1.2 Objective

The aim of the laboratory work lies in the study of models for processing XML data (DOM / SAX) and JSON to distribution.

1.3 Generic requirements

1.3.1 Task

Develop a system of distributed heterogeneous data, centralized in one node type warehousing.

1.3.2 Report

Report will contain a short description of work done, and will present necesary information about tools, algorithms used or studied.

2 Structure

Below you can see the structure of the application, there are represented the classes used and their variables and methods. The application is composed of three main clasess *Client* and *Server* and *Main Server*.

The *Client* and *Server* classes use the *HttpHandler* helper class in order to send the necesary requests to the *Main Server* and also recieve the response from it.

The requests are handled and analyzed on *Main Server* in different threads and saved on a single file, the server doesn't duplicate the records it already has them in his json database file, for this it uses an additional *Hash* file that saves the messages received in a Hash format.

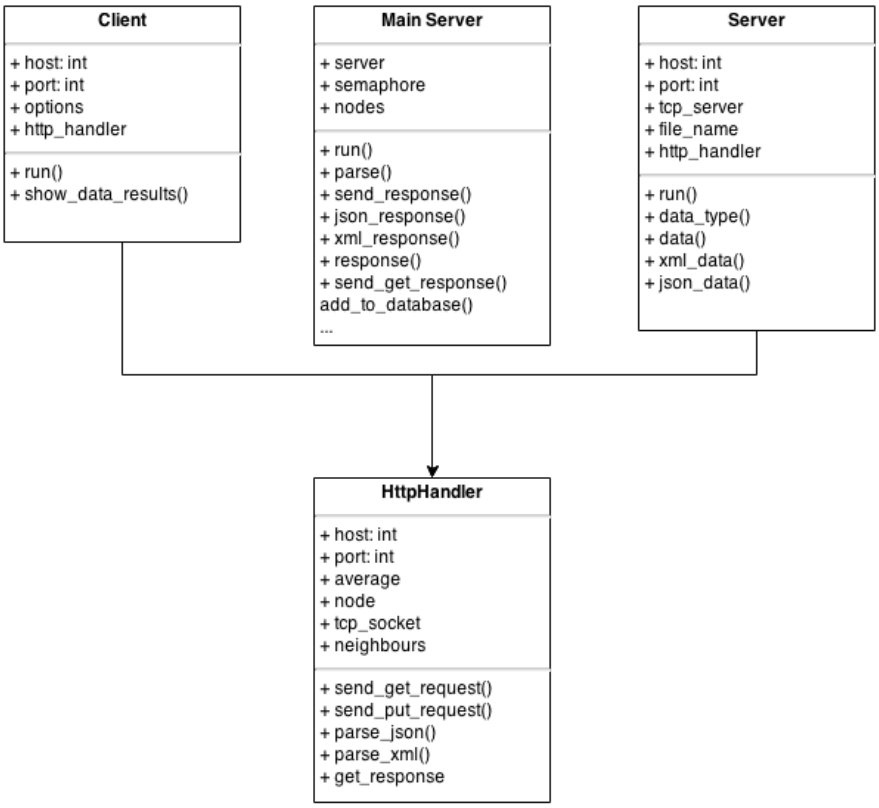

Fig. 1 Class diagram

3 Implementation

3.1 Main server

The most important function of the main server is to receive different request, and depending on type and parameters of request to perform certain actions.

The *run* function is the function that receives requests from nodes or client in a thread safe way and depending of the type of request it acess different methods.

```
# ...
 def run
   loop do
    Thread.start(server.accept) do |client|
      @semaphore.synchronize do
        request = client.gets
        request =~ /^GET.*/ ? send_response(client, request) :
            add_to_database(client.read)
        client.close
      end
    end
  end
# ...
```

Below I will describe how the *GET* sequence it is executed. The *PUT* request processing was implemented and described by my colegue.

The first function that is accessed is

- *send_response* - the method that sends data depending on the type of request received.
- *json_response* - returns the json response.
- *xml_response* - returns the xml response.
- *response* - the method that returns the data depending of the user request.
- *send_get_response* - sends the response in the right format with the right data.

```
# ...
def send_response client, request
 @header, _, type = parse_request(request << client.gets)
 type == 'json' ? send_get_response(client, json_response) :
    send_get_response(client, xml_response)
end

def json_response
  JSON.generate response
 end

 def xml_response
  response.to_xml(:root => 'employees')
 end

 # needs refactor, bad implementation
 def response
  id = @header.scan(/\d+/)
  if id.empty?
   current_employees
  else
   (current_employees.size < id.first.to_i-1) ? {"problem" => "There is no
      such id" } : current_employees[id.first.to_i-1]
  end
 end

 def send_get_response client, response
   client.print "HTTP/1.1 200 OK\r\n" +
        "Content-Type: text/plain\r\n" +
        "Content-Length: #{response.bytesize}\r\n" +
        "Connection: close\r\n"

  client.print "\r\n"
  client.print response
  client.close
 end
 # ...
```

3.2 Client Side

The client side suffered a small number of modifications compared to previous versions, because the all the necessary data is received with *GET* request, there is no need to establish a TCP connection with other nodes to get the data.

3.2.1 HTTP Handler

The handler permforms the same functions that it done previously but with some adjustments.

- *send_get_request* - method that will send get request that contains the type.
- *get_response* - method that receives the *GET* response from the main server.
- *parse_json* - parse the response with json.
- *parse_xml* - parses the xml response with json.

```
# ...
  def send_get_request request, type
   header =["GET #{request} HTTP/1.0",
       "Accept-Type: application/#{type}"].join("\r\n")

    tcp_socket.puts header + "\r\n\r\n"
    type == 'json' ? parse_json(get_response) : parse_xml(get_response)
  end

  def parse_json response
    p "Received json"
    @data = JSON.parse response
  end

  def parse_xml response
    p "Received xml"
    @data = JSON.parse(Hash.from_xml(response).to_json)
  end

  def get_response
    request = tcp_socket.read
    _, body = request.split("\r\n\r\n")
    tcp_socket.close
    body
  end
  # ...
```

3.2.2 Data manipulation

Data manipulation class as the name says it responsible to analyze, parse and show received data in a readable form.

- *show_all* - display all the employers received from *GET* request.
- *show_entry* - display the entry in a table format.
- *table_format* - add the table format to the parsed data.
- *check_xml?* - checks if received data was in xml format.

```
class DataManipulation
  # ...
  def show_all
    @data = @data["employees"] if check_xml?
    puts table_format(data)
  end

  def show_entry
    @data = @data["employees"] if check_xml_entry?
    puts entry_table_format(data)
  end

  private

  def table_format data
    # ...
  end

  def entry_table_format data
    # ...
  end

  def check_xml?
    data.is_a? Hash
  end

  def check_xml_entry?
    data.has_key? "employees"
  end

end
```

3.2.3 Main

The main class *Client* contains all methods that are necesary to start and perform necesary work. Below there is a short description of methods that it has, for the implementation you can see the full code.

- In the *contructor* we save provided options and setup http handler.
- *run* - performs get request to the main server.
- *show_data_results* - show data received from mian server, based on option from console.

```ruby
class Client

  HOST = 'localhost'
  FILE_NAME = "client_data.json"

  attr_reader :http_client, :options

  def initialize options
    @options = options
    @http_client = HttpHandler.new HOST, 8000
  end

  def run
    http_client.send_get_request options[:request], options[:type]
    show_data_results
  end

  private

  def show_data_results
    dt = DataManipulation.new(http_client.data)
    options[:request] =~ /\d+/ ? dt.show_entry : dt.show_all
  end

end
```

The script receives two kind of arguments the request and the type of it.

```ruby
options = {}
OptionParser.new do |opts|
  opts.banner = "Usage: ruby client.rb [options]\n\n"
  opts.on("-r", "--request [request]", "get request to main server") do |request|
    options[:request] = request || "/emploees"
  end

  opts.on("-t", "--type [TYPE]", "file name containing data") do |type|
    options[:type] = type || "json"
  end
end.parse!
```

The client actions are started with the following line, it receive the actions that need to be applied on the data.

```ruby
Client.new(options).run
```

3.3 Server

The server part was done by Vasilica Victor and the handling of the *PUT* requset on the main server.

3.4 Xml check

The nodes send the contents of their files with help of *PUT* request, the *Main Server* has the functionality to check if the contents of the send message are valid and then save them into the file.

First he loads the schema file of the xml that should match the xml it will receive.

```
@xml_schema = Nokogiri::XML::Schema(File.read("dataSchema.xsd"))
```

And the main check take place before saving to the database, it checks if the type is xml and then validates with help of *Nokogiri* librari with in case of errors will return an array of errors. Then we check if that array is empty then we pass the xml to json to our save method, otherwise the return a message of error to the terminal.

```
def add_to_database http_body
  _, _, type = parse_request http_header
  json = http_body

  if type == "xml"
    errors = xml_schema.validate(Nokogiri::XML(http_body))
    if errors.empty? #valid xml
      json = xml_to_json(http_body)
    else #invalid xml
      puts "ERROR: invalid xml"
      return
    end
  end
  save(json) unless check_uniq?(json)
end
```

4 Conclusion

In this laboratory work we build a distributed system that uses different type of data and different requests, each doing its own predefined task. Thus after learning more about the uses and formats of data, we apply each to solve their individual task.

We build a helper class that handles all required requests *GET* and *PUT*, and receives the responses from the *Main Server*. Depending on the request "/employess" or specific one "employees/id=1" and the type specified "json" or "xml", the *Main Server* will return the list of users or a single required user.
We also worked with *json* in order to store our collection of necesary data, on the *Main Server* and also a array of hashes to prevent duplicated entries.
Thus we learned how to build a small distributed system with a collection of objects using different tpes and requests.

Link to Repository: `https://gitlab.ati.utm.md/petru.negrei/lab4/tree/wip/petru`

5 References

- Ruby Socket http://www.ruby-doc.org/stdlib-1.9.3/libdoc/socket/rdoc/Socket.html
- Ruby TCP Socket http://www.ruby-doc.org/stdlib-1.9.3/libdoc/socket/rdoc/TCPSocket.html
- Ruby Mutex http://wwwi.ruby-doc.org/core-2.1.4/Mutex.html
- Ruby JSON http://www.ruby-doc.org/stdlib-2.0.0/libdoc/json/rdoc/JSON.html
- Ruby OptionParser http://ruby-doc.org/stdlib-2.1.0/libdoc/optparse/rdoc/OptionParser.html