Faculty of Computers, Informatics and Microelectronics

Technical University of Moldova

PAD

Laboratory work # 2

Distributed systems

Authors:
Petru Negrei
Victor Vasilica

Supervisor:
D. Ciorba

September 2014

1 Introduction

1.1 Topic

Building a distributed system.

1.2 Objective

Study of different transport protocols TCP/IP in order to build a application that contains collection of distributed data.

1.3 Generic requirements

1.3.1 Task

- Use UDP protocol for unicast and multicast transmission.
- Use TCP protocol for data transmission.
- Analyze and process collection of objects.

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 main are client and server which use the two helper classes (can be named modules) that implement a certain functionality.

The client will be responsible for sending at first UDP request to all nodes, then find the node with max connections and establish a TCP connection with it. After it receive all necesary data, this data will be analyzed and shown to the user.

In order to allow multiple responses of server nodes, each connection will be handled in different thread.

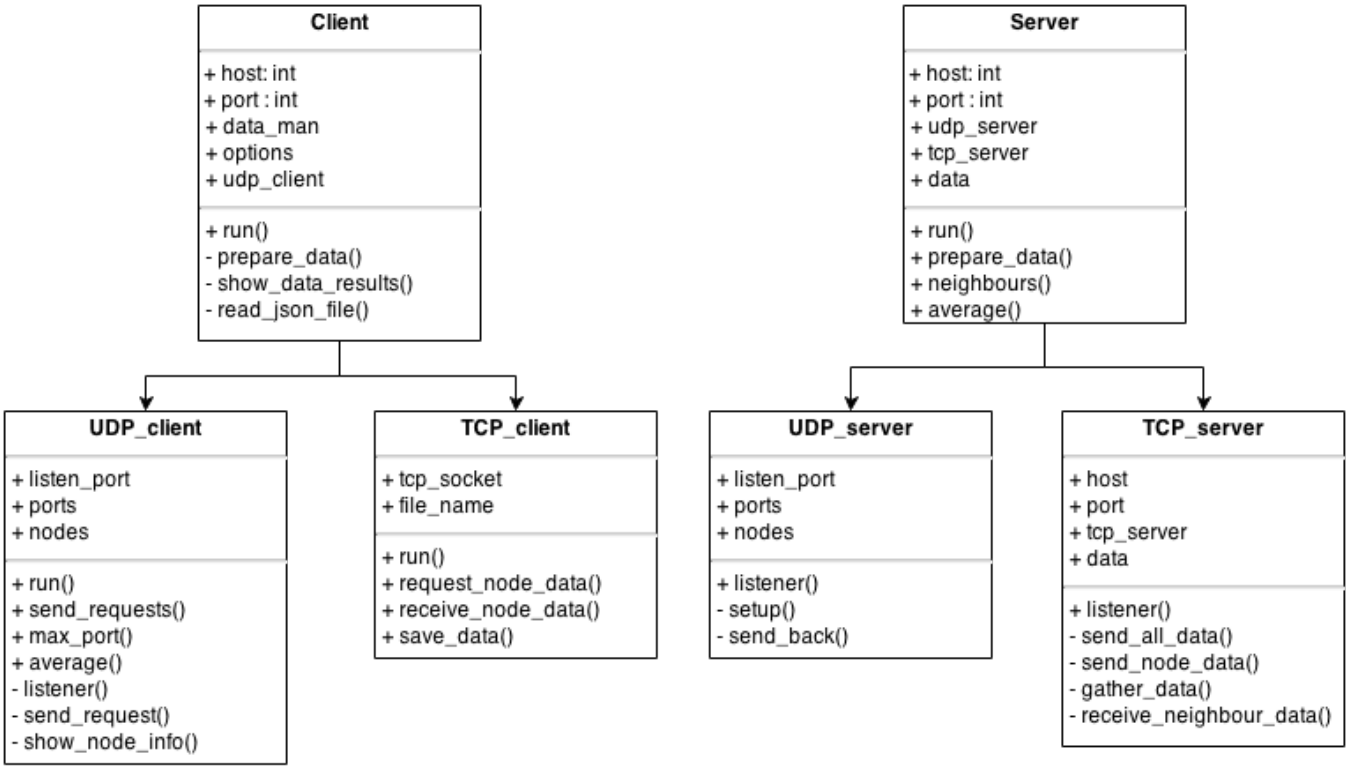

Fig. 1 Class diagram

3 Implementation

3.1 Stored data

All information of servers is stored in one single *json* file (*data.json*) in order to reduce redundancy in real project each server will have separate file that will contain its specific data.

JSON or JavaScript Object Notation, is an open standard format that uses human-readable text to transmit data objects consisting of attributevalue pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML.

With this implementation all servers can easily access their data, and thus reduce redundancy in code, and optimize the creation and initialization of the servers. The data contains a list of available nodes, more specifically their port number, which subsequently contain information about their neigthbours and the list of all employes.

In the code below you can see the the configuration of servers with the information about their neighbours, and list of employers.

```
{
  "10000": {
    "neigthbours": [10001],
    "employers":[ ... ]
  },
  "10001": {
    "neigthbours": [10000, 10002],
    "employers":[ ... ]
  },
  "10002": {
    "neigthbours": [10001, 10003],
    "employers":[ ... ]
 },
  "10003": {
    "neigthbours": [10002, 10004],
    "employers":[ ... ]
 },
 "10004": {
    "neigthbours": [10003],
    "employers":[ ... ]
 }
}
```

```
# ...
   "employers":[
      {"firstName":"John", "lastName":"Doe", "salary": 150, "department":
         "Marketing"},
      ...
      {"firstName":"Alex", "lastName":"Carturari", "salary": 478, "department":
         "Testing"}
    ]
# ...
```

3.2 Dependencies

In the application we used some external packages, (*gems*), that provide additional functionality. Below I will provide a small description of each one.

- *socket* - provides access to the underlying operating system socket implementations. It can be used to provide more operating system specific functionality than the protocol-specific socket classes.
- *json* - allow to generate or parse json easily.
- *optparse* - is a class for command-line option analysis. It is much more advanced, yet also easier to use.
- *rainbow* - is a ruby gem for colorizing printed text on ANSI terminals.
- *terminal-table* - is a fast and simple, yet feature rich ASCII table generator written in Ruby.

3.3 Client Side

The client part of the application is composed of four main file that together provide functionality necesary to assure the request to server, receive, save, parse and show the response in form of the list of emplyoers from server with the maximum nodes and its neighbours.

3.3.1 UDP protocol

The class below provides the functionality that assures the transmission between server and client using UDP protocol.

- *initialize* - creates a udp socket.
- *send_requests* - method that will send requests to the ports available in its list.
- *listener* method serves for receiving udp messages, and store the responses in *@nodes* array, that will be used to analyze this information.
- *send_requests* - method that sends request to the ports available in its list for information.
- *max_port* - returns the node with max connections.
- *average* - computes the total average of all nodes.

```
class UdpClient

  LISTEN_PORT = 20001
  PORTS = [10000, 10001, 10002]

  def initialize host
    BasicSocket.do_not_reverse_lookup = true
    @udp_server = UDPSocket.new
    @udp_server.setsockopt(Socket::SOL_SOCKET,Socket::SO_BROADCAST, true)
    @udp_server.bind(host, LISTEN_PORT)
  end

  def send_requests
    PORTS.each {|port| send_request port}
  end

  def run
    loop { listener }
  end

  def max_port
    nodes.max_by { |info| info[:neighbours] }[:port]
  end

  def average
    sum = nodes.reduce(0) { |sum, node| sum += node[:average].to_i }
    sum / nodes.size
  end
```

```
  private
  def listener
    response, client = @udp_server.recvfrom(1024)
    Thread.new(client) do |clientAddress|
      neighbours, average = response.chomp.split(":")
      nodes << {port: clientAddress[1], neighbours: neighbours, average: average}
      show_node_info(nodes.last)
    end
  end
  # ...
end
```

3.3.2 TCP protocol

The class below provides the functionality that assures the transmission between server and client using TCP protocol.

- *initialize* - First we create a TCP socket, and
- *run* - here we call the following methods.
- *request_node_data* - send a tcp request to receive emplyoers information.
- *receive_node_data* - receives all information from server.
- *save_data* - saves received data into a file.

```
class TcpClient

  def initialize host, file_name, port
    @tcp_socket = TCPSocket.open(host, port)
    @file_name = file_name
  end

  def run
    request_node_data
    save_data(receive_node_data)
  end

  def request_node_data
    tcp_socket.puts "*all*node*data*"
  end

  def receive_node_data
    # ...
  end

  def save_data data
    # ...
  end
end
```

3.3.3 Data manipulation

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

- *show_** - display the information at the console.
- *filtered_data* - analyze and parse the data.
- *table_format* - add the table format to the parsed data.

```ruby
class DataManipulation

  def initialize data, average
    @data = data
    @average = average
  end

  def show_all
    puts table_format(data)
  end

  def show_filtered
    # ...
  end

  private

  def filter_data
    data.select{ |employer| employer["salary"] > average }
          .sort_by{ |employer| employer["lastName"] }
          .group_by{ |employer| employer["department"] }
  end

  def table_format data
    # ...
  end

end
```

3.3.4 Main

The main class *Client* contains all methods that are necesary to start and perform necesary work, it also has UDP and TCP functionality by using the *classes* with the same name. 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 udp server.
- *receive_data* - we setup a tcp server, send requests to servers and save response to a file.
- *prepare_data* - read the json file and initiate the data manipulation class.

- *show_data_results* - show data received from server, based on option from console.

```ruby
class Client

  HOST = 'localhost'
  WAIT_TIME = 3
  FILE_NAME = "client_data.json"

  def initialize options
    @options = options
    @udp_client = UdpClient.new(HOST)
  end

  def run
    # ...
  end

  private

  def receive_data
    tcp_client = TcpClient.new(HOST, FILE_NAME, udp_client.max_port)
    tcp_client.run
  end

  def prepare_data
    data = read_json_file
    @data_man = DataManipulation.new(data, udp_client.average)
  end

  def show_data_results
    case options[:show]
    when "default" then data_man.show_all
    when "filtered" then data_man.show_filtered
    else
      puts "Invalid option"
    end
  end

  def read_json_file
    JSON.parse File.read(FILE_NAME)
  end

end

# input arguments from terminal
options = {}
options[:show] = "default"
OptionParser.new do |opts|
  opts.banner = "Usage: ruby client.rb [options]\n\n"
  opts.on("-s", "--show [SHOW]", "show data by method") do |s|
    options[:show] = s || "default"
  end
end.parse!
```

One of its most important methods is ‘run‘ method, here we first send upd request to all servers that this client knows, then after a interval of 3 seconds, we close the receiving udp port, and then we establish a tcp connection with the node with maximum neighboars, receiving necesary information from it, parse it and show it in the console.

```
# ...
  def run
    udp_client.send_requests
    begin
      Timeout::timeout(WAIT_TIME) { udp_client.run }
    rescue Timeout::Error => e
      # abort with coupling
      abort("There are no available nodes") if udp_client.nodes.empty?
      p "Port with max connections is #{udp_client.max_port}"

      # save - read - show
      receive_data
      prepare_data
      show_data_results
    end
  end
  # ...
```

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

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

3.4 Servers

The server part was done by Vasilica Victor but a part of functionality of TCP class was done working together, (pair programming) in order to establish the correct connection between servers and client.

4 Conclusion

In this laboratory work we build a distributed system that has at its basis the different transport protocols TCP and UDP, that were used for individual purposes, each doing its own predefined task. Thus after learning more about the advantages and disadvantages of each protocol, we apply each to solve their individual task.

At first we used UDP protocol for unicast and multicast transmission in order to receive information related to the number of neighbours node of each server (node), and at the same time the average *salary* information. This decision was implemented because the *average salary* is represented by a number, and it is easy to send using the udp protocol, the downside of this approach is that there is a high chance to lose this data, and this will result in wrong calculation of total average of nodes, and represeting the information to the client.

We also worked with *json* in order to store our collection of necesary data, that contains the port number, list of neighboars and the list of employers of each node, by reading, parsing and then saving in the same format type.

Thus we learned how to build a small distributed system with a collection of objects using different types of transport protocols.

Link to Repository: `https://gitlab.ati.utm.md/petru.negrei/lab2/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 UDP Socket `http://ruby-doc.org/stdlib-1.9.3/libdoc/socket/rdoc/UDPSocket.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`