



UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS NATURAIS
FACULDADE DE COMPUTAÇÃO
SISTEMAS DISTRIBUÍDOS - EN05227
TURMA 7M3456
DOCENTE: HELDER MAY NUNES DA SILVA OLIVEIRA

BRUNO CONDE COSTA DA SILVA
MATRÍCULA: 201506840054
ENGENHARIA DA COMPUTAÇÃO - VESPERTINO
INSTITUTO DE TECNOLOGIA

PRIMEIRA AVALIAÇÃO

Implementar Sistema Distribuído
Cliente/Servidor que compartilham objetos em
rede usando Java RMI ou Pyro

BELÉM
2020

Introdução

O objetivo deste trabalho é implementar um objeto servidor de perfis que possa ser acessado em máquinas ou redes diferentes por um cliente utilizando uma das duas tecnologias dedicadas a esta tarefa: Java RMI ou Pyro. Logo, teremos uma base de dados de perfil, acessível pelo cliente através de um objeto servidor compartilhado em rede.

Serão duas aplicações: a aplicação cliente e a aplicação servidor. A aplicação servidor irá armazenar os perfis e a aplicação cliente poderá consultar/inserir informações aos perfis somente acessando a aplicação servidora, sem acessar os perfis diretamente. Logo, a aplicação servidora protege o acesso direto aos perfis por parte do cliente e define o que e como o cliente pode acessar em relação aos perfis.

Cliente e servidor executarão em redes diferentes, logo, o objeto servidor deve ser disponibilizado ao cliente remotamente para que assim o cliente consiga ter acesso aos perfis.

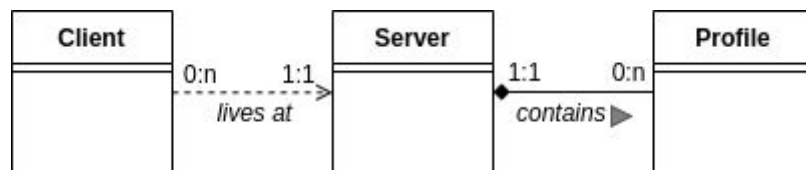


Figura 01: Esboço geral da aplicação.

Como vemos na figura 01, um cliente (aplicação cliente, e não um objeto) acessa um e somente um objeto servidor, e um objeto servidor pode estar sendo acessado por nenhum ou muitos clientes. Logo, a relação entre cliente e servidor é de dependência, pois o cliente depende do servidor.

Da figura 01, também temos que um objeto servidor contém 0 ou muitos objetos de perfis e que todo objeto de perfil deve estar contido em um e somente um objeto servidor. Essa é uma relação de associação/composição, onde a classe que possui o diamante negro (*Server*) controla a associação e, além disso, a outra classe (*Profile*) não pode estar associada a outras instâncias.

Metodologia

Para realizar este trabalho, será utilizada a linguagem python 3.8 e as seguintes bibliotecas: Pyro versão 4.80, typing versão 3.7.4.3, matplotlib versão 3.3.3 e PyQt5 versão 5.15.2.

Pyro foi utilizada para disponibilizar o acesso remoto ao objeto servidor de perfis, typing foi utilizada para criar anotações de tipos de variáveis e retorno de métodos que ajudam o programador a entender o código com muito mais clareza e facilidade, matplotlib para criar gráficos e PyQt5 para renderizar os gráficos na tela.

Considerando que não foi especificado qual banco de dados que deve ser usado, tampouco se o mesmo deve ser relacional ou não-relacional, foi criada uma classe chamada *Profile*, que será utilizada, pelo servidor, para criar objetos de perfis. O servidor armazenará esses objetos em uma lista.

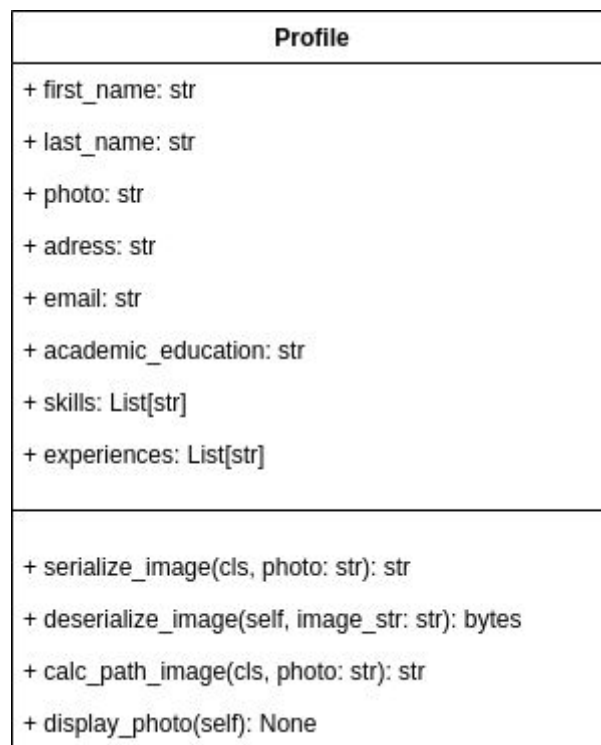


Figura 02: Diagrama UML da classe *Profile*.

Todos os atributos de *Profile* são públicos. A classe *Profile* possui um método construtor que requer cada um dos atributos listados. Importante ressaltar que como os objetos perfis são possuem uma imagem/foto, logo, foram adicionados métodos para serializar e deserializar a imagem.

No método construtor da classe *Profile*, o nome da foto é recebido em string, e passado para a função *serialize_image*, que então invoca o método *calc_path_image*, que por sua vez retorna para a função que o invocou o caminho absoluto para a imagem, onde então a imagem é aberta em sua forma binária, codifica para base 64 e convertida para string, onde então essa string referente codificação em base 64 é armazenada no atributo *photo* da instância da *Profile*. Isso permite que a imagem trafegue em forma de string na rede.

A aplicação cliente irá utilizar o método *display_photo* para exibir a imagem no lado do cliente. Este método invoca o método *deserialize* para transformar a imagem de volta para bytes e exibir na tela.

O objeto servidor de perfis é definido no diagrama abaixo.

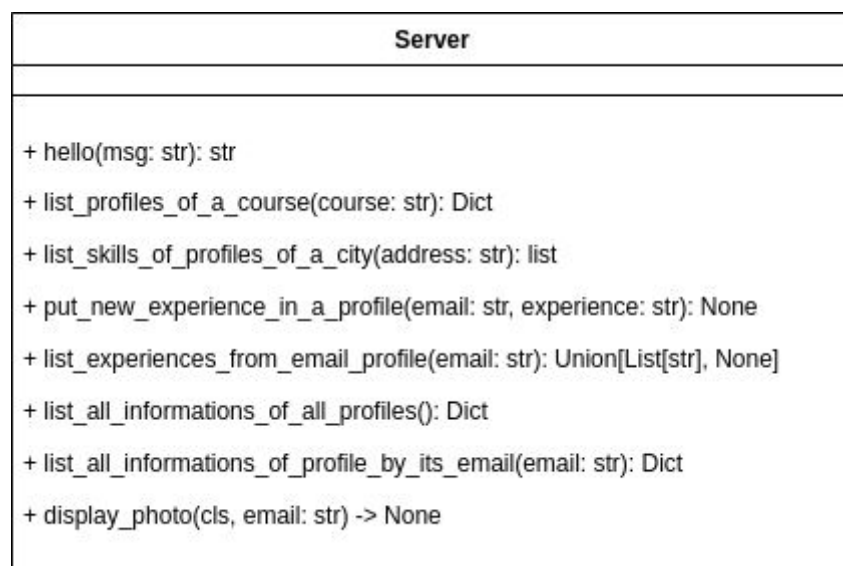


Figura 03: Diagrama UML da classe *Server*.

Todos os métodos da classe *Server* são públicos e estáticos. O método construtor da classe *Server* é vazio. Cada método é auto explicativo e especifica os tipos de argumentos e o tipo de retorno.

É importante ressaltar que a instância de *Server* não retorna para a aplicação cliente os objetos em si (pois somente a instância de *Server* está disponível remotamente), e sim as informações contidas na lista de instâncias de *Profile* em forma de dicionário ou lista de dicionários, pois dessa forma, a base de dados em memória (que é a lista de instâncias de *Profile*) se mantém seguras.

- O método *hello* é apenas para teste, para saber se a aplicação cliente conseguiu acessar a instância de *Server*, onde a aplicação cliente envia uma string que vai ser exibida no console da aplicação servidora;
- O método *display_photo* é uma *wrapper* para o método *display_photo* da instância de *Profile* que se deseja exibir a foto;
- Os demais métodos são auto explicativos e internamente.

As fotos dos perfis foram armazenadas na pasta *photos*. A aplicação servidora, por efeitos de estudo, e não de cenário real, poderá instanciar 3 perfis. Segue abaixo a foto dos 3 perfis.



Figura 04: Foto do perfil do Bruno.

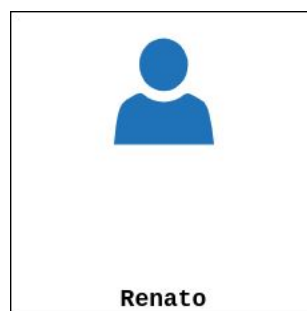


Figura 05: Foto do perfil do Renato.

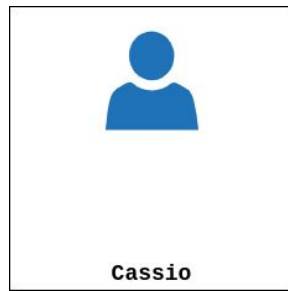


Figura 05: Foto do perfil do Cassio.

Após as classes definidas, foi dado início a construção das aplicações cliente e servidor que foram criadas, respectivamente nos arquivos `app_client.py` e `app_server.py`.

- **Aplicação Servidor**

app_server.py

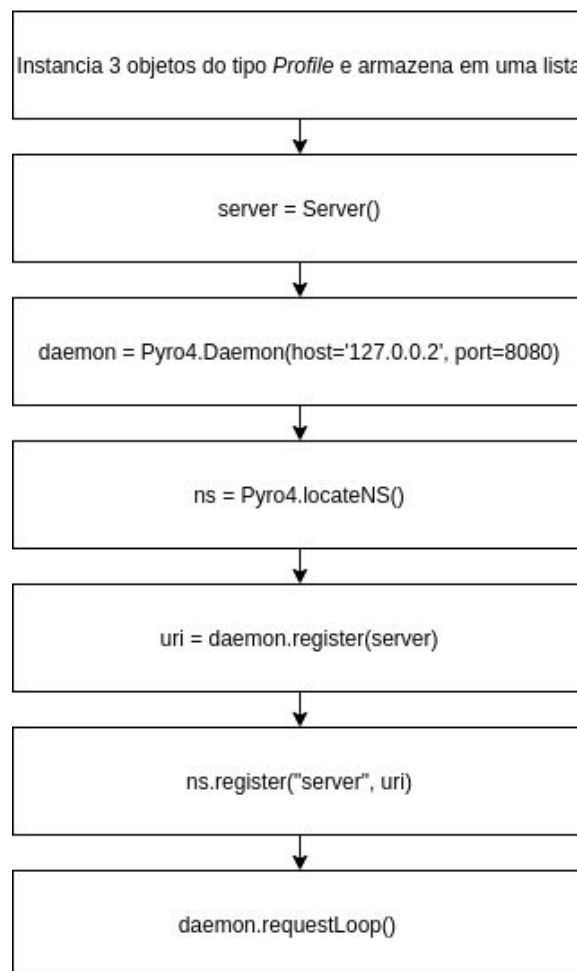
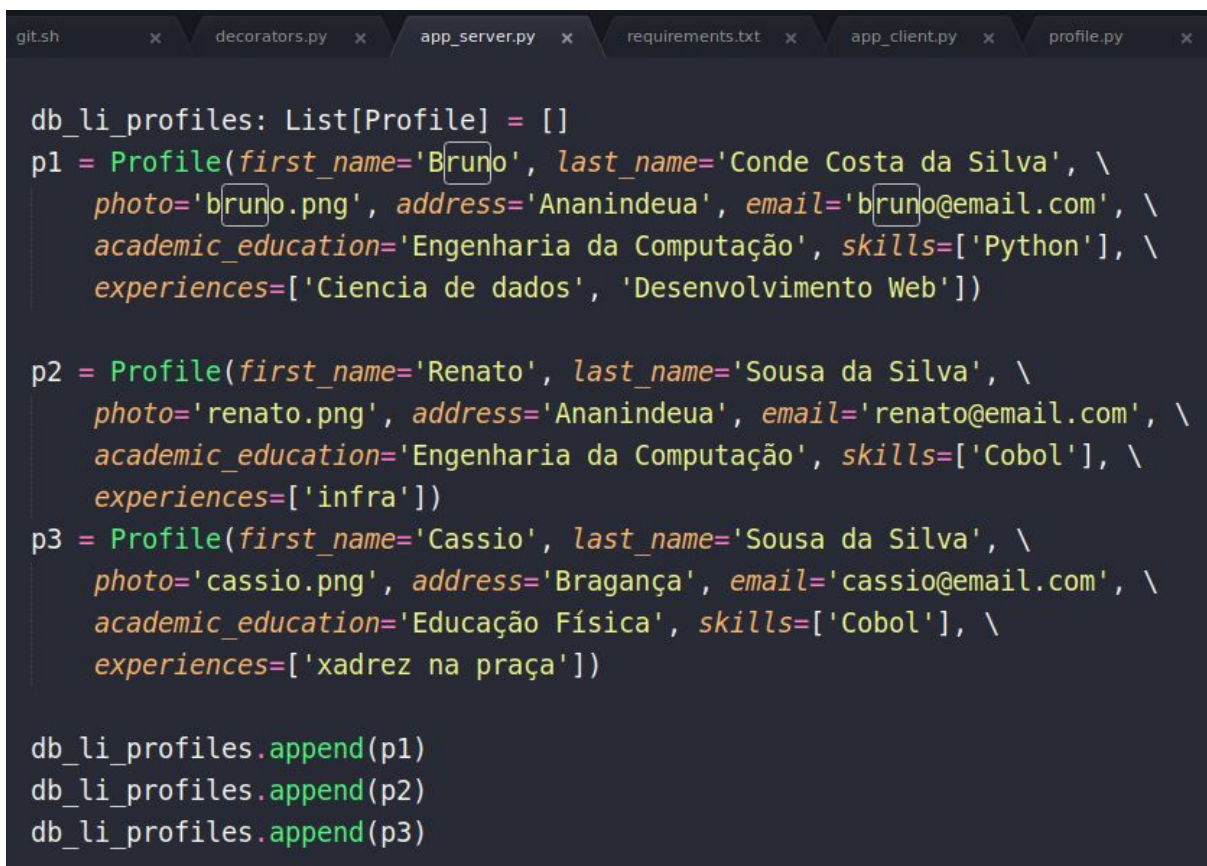


Figura 06: aplicação servidora (`app_server.py`)

Da figura 06:

- Inicialmente, foi instanciado os perfis de Bruno, Renato e Cassio e armazenado em uma lista, que atuará como a base de dados em memória de perfis;
- Foi instanciado o objeto servidor de perfis;
- `ns = Pyro4.locateNS()`: aloca-se o *nameserver* ;
- `daemon = Pyro4.Daemon(host='127.0.0.2', port=8080)`: cria um daemon Pyro para ouvir chamadas remotas, onde vale ressaltar que todo objeto Pyro é registrado em um ou mais daemons
- `uri = daemon.register(server)`: registra o objeto servidor de perfis como um objeto Pyro, onde *uri* (*unique resource locator*) é o identificador do objeto servidor de perfis na rede;
- `ns.register("server", uri)`: o objeto servidor de perfis é registrado com um nome no servidor de nomes (*nameserver*)



```
git.sh x decorators.py x app_server.py x requirements.txt x app_client.py x profile.py x

db_li_profiles: List[Profile] = []
p1 = Profile(first_name='Bruno', last_name='Conde Costa da Silva', \
    photo='bruno.png', address='Ananindeua', email='bruno@email.com', \
    academic_education='Engenharia da Computação', skills=['Python'], \
    experiences=['Ciencia de dados', 'Desenvolvimento Web'])

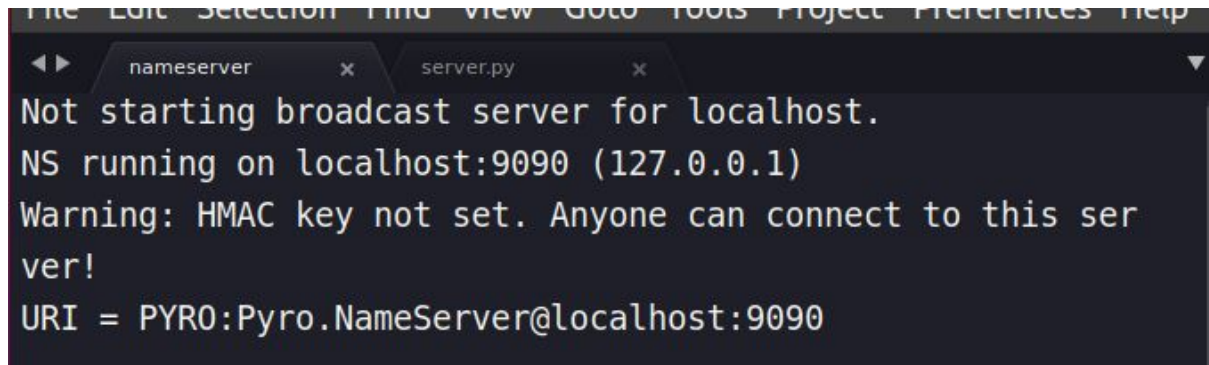
p2 = Profile(first_name='Renato', last_name='Sousa da Silva', \
    photo='renato.png', address='Ananindeua', email='renato@email.com', \
    academic_education='Engenharia da Computação', skills=['Cobol'], \
    experiences=['infra'])

p3 = Profile(first_name='Cassio', last_name='Sousa da Silva', \
    photo='cassio.png', address='Bragança', email='cassio@email.com', \
    academic_education='Educação Física', skills=['Cobol'], \
    experiences=['xadrez na praça'])

db_li_profiles.append(p1)
db_li_profiles.append(p2)
db_li_profiles.append(p3)
```

Figura 07: Instanciando os 3 perfis mencionados acima.

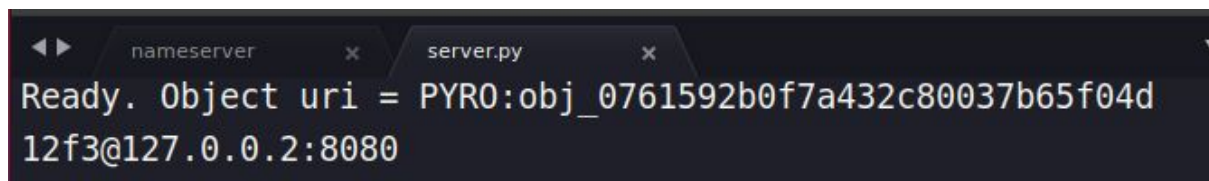
Após a criação da base de dados em memória de perfis, foi configurado um *nameserver* no endereço 127.0.0.1, em uma porta arbitrária definida pelo pacote Pyro4, que vai servir o objeto *server*, que é uma instância da classe *Server*, que irá prover informações sobre os objetos perfis (instâncias da classe *Profile*). Fazemos isso ao executar o comando **python -m Pyro4.naming**



```
File Edit Selection Find View Goto Tools Project Preferences Help
nameserver x server.py x
Not starting broadcast server for localhost.
NS running on localhost:9090 (127.0.0.1)
Warning: HMAC key not set. Anyone can connect to this server!
URI = PYRO:Pyro.NameServer@localhost:9090
```

Figura 08: nameserver.

Após inicializar o *nameserver*, é inicializado no endereço 127.0.0.2, porta 8080, o *daemon*, que fica , em rodando, em background, um loop multi thread para receber requisições do cliente, via Proxy, para o objeto remoto, no endereço 127.0.0.1, porta randômica, hospedado pelo *nameserver*, via intermédio do servidor de *daemon*.



```
nameserver x server.py x
Ready. Object uri = PYRO:obj_0761592b0f7a432c80037b65f04d12f3@127.0.0.2:8080
```

Figura 09: daemon.

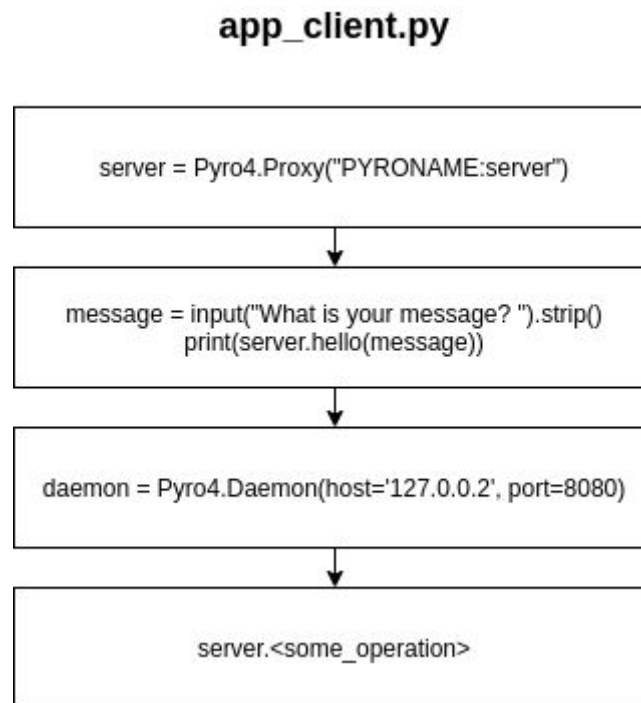


Figura 10: aplicação cliente (app_client.py)

Na figura 10, temos a aplicação cliente:

- *server = Pyro4.Proxy("PYRONAME:server")*: cliente acessa a instância de *Server* de forma remota via Proxy, passando o uri server, que foi o uri atribuído ao objeto *server* e registrado no *nameserver*. Agora o cliente pode acessar qualquer método remoto da instância de *Server* como se fosse um objeto local.
- Após isso, o cliente envia uma mensagem para o servidor, que vai ser exibida no console do servidor. Isto só serve para saber que o cliente está acessando corretamente o servidor remoto.
- Após isso, basta o cliente executar qualquer método utilizando o objeto *server* local, que na verdade é um acesso via proxy para o objeto remoto que realmente está instanciado.

Para fazer a contagem do tempo de cada operação e plotar os gráficos foram criadas, respectivamente, as funções *calc_time* e *calc_plot*. Elas são funções decoradoras, que atuam como wrapper para as operações que queremos calcular o tempo e plotar o gráfico. Funciona da seguinte forma: antes de cada operação/método da classe *Server*, é colocado um decorator *@calc_plot* e *@calc_time*. *@calc_time* atua como wrapper para operação/método em si, e toda vez que a operação precedida por *@calc_time* é invocada, *calc_time* executa a

operação/método em seu escopo interno e, antes de executar, registra o tempo antes e depois do final de execução da operação e assim calcula o tempo de execução da operação bem como obtém também o resultado da operação/método em si. calc_time passa para a função calc_plot a lista de tempo de execuções que a operação levou em cada execução e o resultado da operação/método em si, e então, calc_plot faz o gráfico e salva ou plota, dependendo se a variável DEBUG do módulo decorators.py (que contém as duas funções decoradoras) estiver setada como True ou False.

Ainda em decorators.py, a variável N_PLOTS e CONFIDENCE recebem, respectivamente, o número de vezes que é pra executar o método e o valor de confiança para o cálculo estatístico.

Resultados

Para efeito de avaliação de desempenho, foi definido N_PLOTS=20.

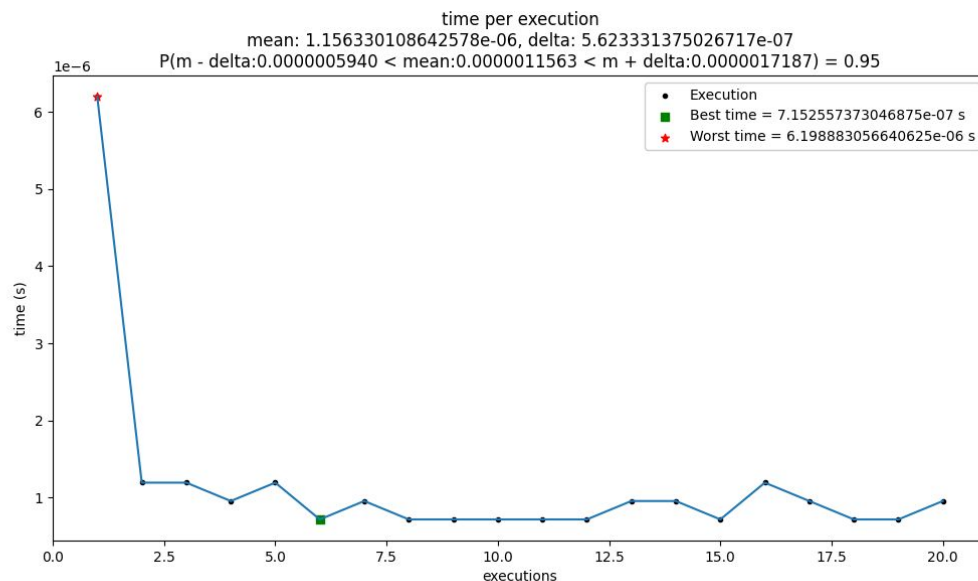


Figura 11: `server.list_profiles_of_a_course(course='Engenharia da Computação')`

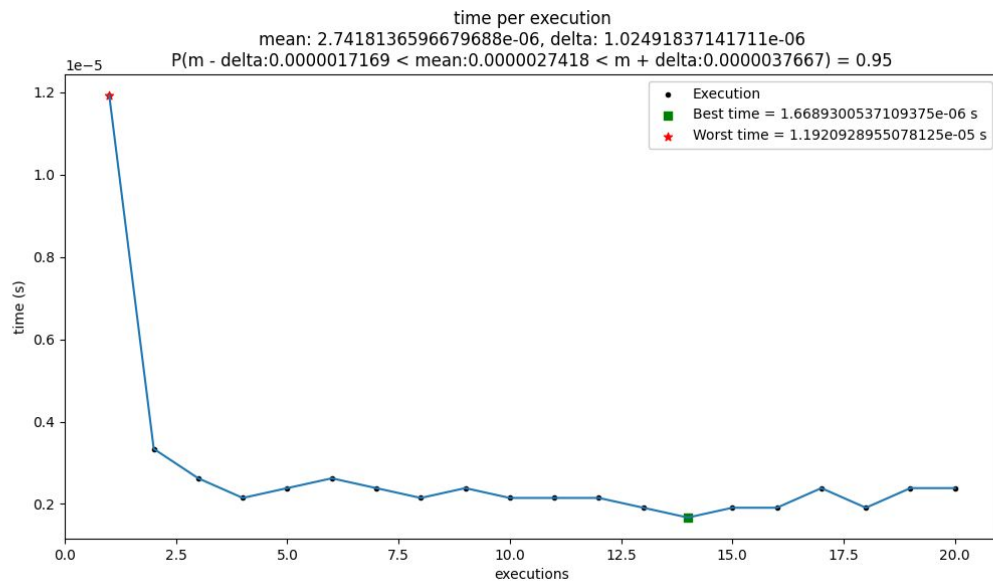


Figura 12: server.list_skills_of_profiles_of_a_city(address="Ananindeua")

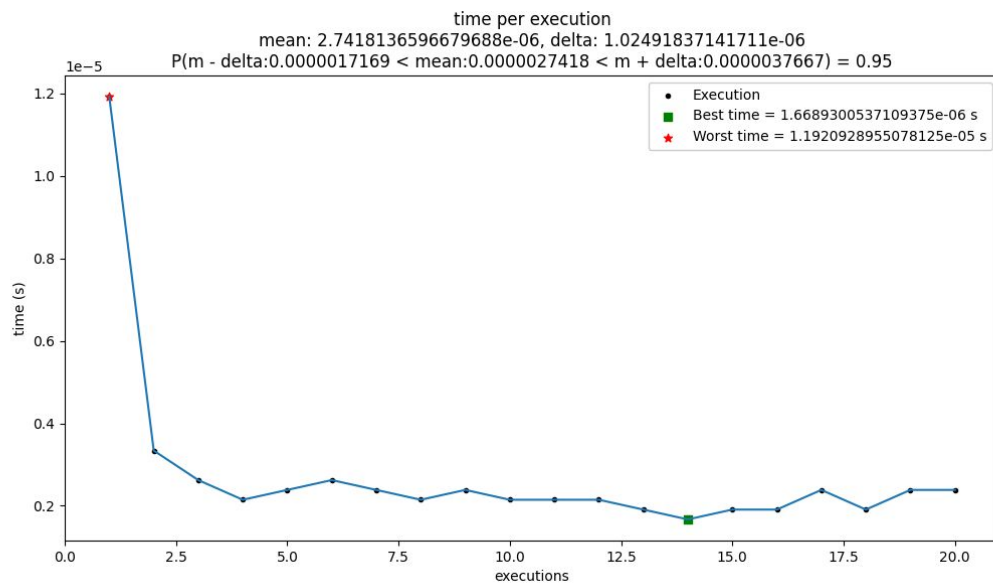


Figura 13: server.list_experiences_from_email_profile(email='bruno@email.com')

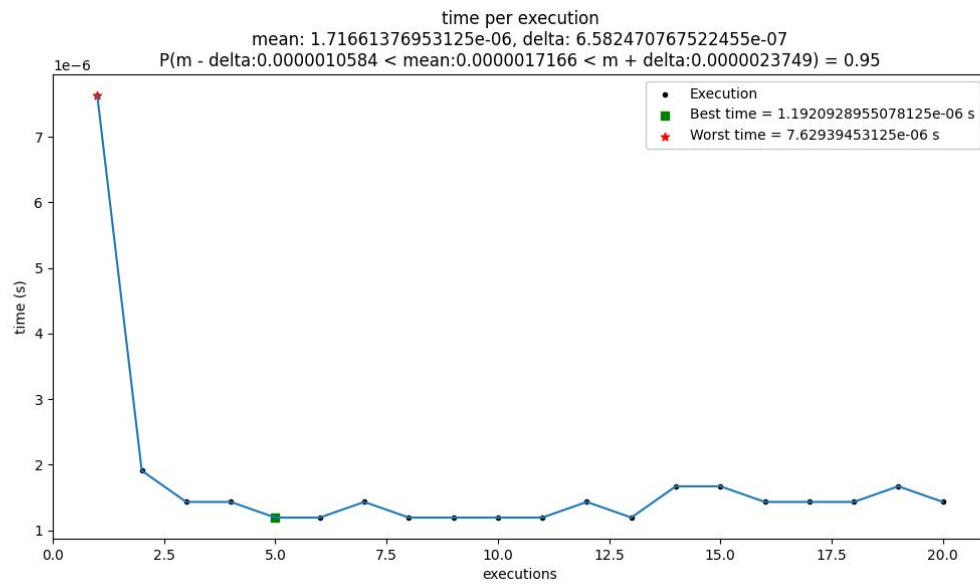


Figura 14: `server.list_experiences_from_email_profile(email='cassio@email.com')`

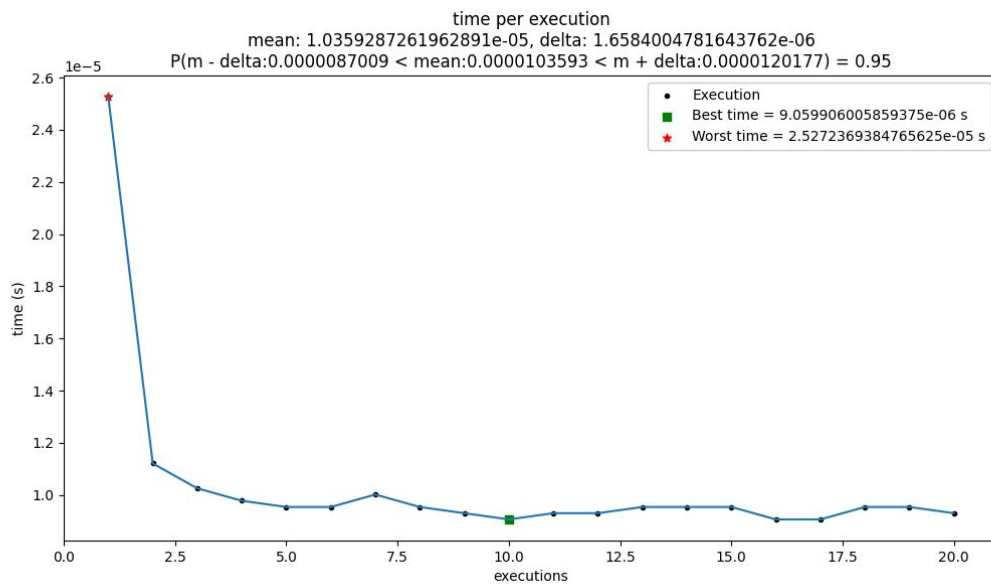


Figura 15: `server.list_all_informations_of_all_profiles()`

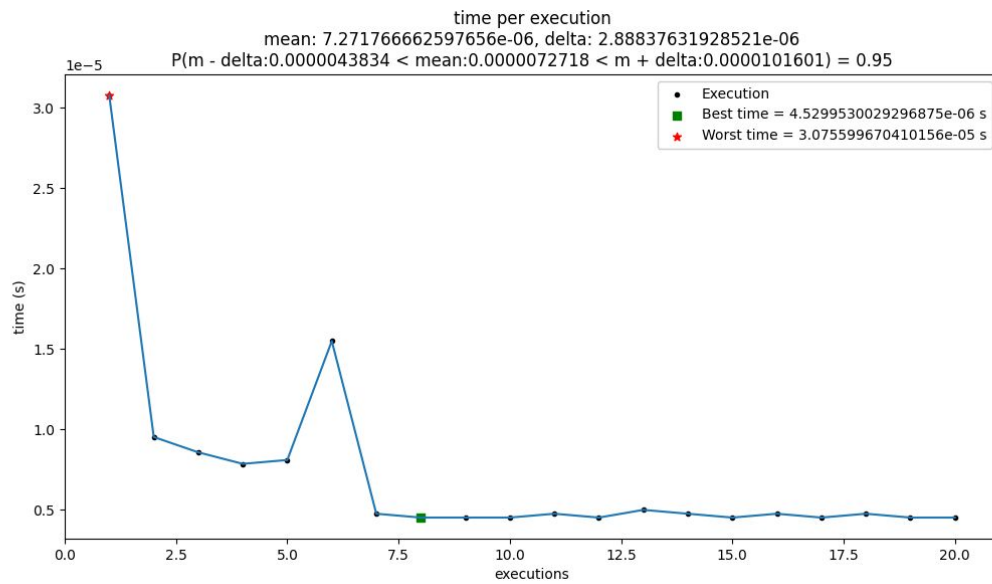


Figura 16: `server.list_all_information_of_profile_by_its_email(email="renato@email.com")`

Como podemos ver, a aplicação apresenta bom desempenho e tempo médio de execução baixo.