UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS DE FLORESTAL
CIÊNCIA DA COMPUTAÇÃO

ATHENA SARANTÔPOULOS (2652)
JOSE GRIGORIO NETO (3046)

SISTEMAS DISTRIBUÍDOS

TRABALHO PRÁTICO 2 - PARTE 2

FLORESTAL, MG

18 de novembro de 2020

ATHENA SARANTÔPOULOS (2652)
JOSE GRIGORIO NETO (3046)

SISTEMAS DISTRIBUÍDOS
TRABALHO PRÁTICO 2 - PARTE 2

Documentação do 2º trabalho prático de Sistemas Distribuídos que tem como objetivo realizar uma implementação de um Middleware RMI(Remote Method Invocation)

Orientador: Profa. Dra. Thais Regina de Moura Braga Silva

FLORESTAL, MG

18 de novembro de 2020

Resumo

Este trabalho tem como objetivo realizar uma implementação de um Middleware RMI, configurando como o uso de visão de processos para os elementos arquitetônicos.O sistema distribuído que será implementado possui o modelo de arquitetura cliente-servidor. Onde o cliente entrará em contato com 3 servidores: servidor aeroporto, servidor hotel e servidor passeio, gerando assim um sistema distribuído Agência de Turism, através do método de invocação remota.

Palavras-chaves: sistema distribuído, middleware, RMI, cliente-servidor.

Abstract

This work aims to carry out an implementation of a Middleware RMI, configuring as the use of process view for the architectural elements. The distributed system that will be implemented has the client-server architecture model. Where the client will get in touch with 3 servers: airport server, hotel server and tour server, thus generating a distributed Tourism Agency system, through the remote invocation method.

Key-words: distributed system, middleware, RMI, client-server.

Sumário

1 Introdução

Este trabalho é dividido em três partes. Essa documentação foi realizada para a implementação da segunda parte, que consiste em utilizar um middleware que utiliza método de invocaçao remota, configurando assim o uso da visão de processos para os elementos arquitetônicos. Sendo assim, foi utilizado o Pyro que permite que o Sistema Distribuído seja visto como uma coletânea de Objetos e utiliza o método de invocação remota. Foi utilizado a interface gráfica desenvolvida na parte um do trabalho.

O sistema distribuído possui o modelo servidor-cliente, onde possui um cliente e três servidores. O cliente então, precisa entrar em contato com o servidor passagem aérea, hospedagem e passeios, gerando assim um pacote turístico. Ao longo dessa documentação iremos dissertar sobre a escolha de cada parte desse processo de implementação.

UFV

2 Como executar

Para executar o sistema criamos dois shell simples para automatizar algumas tarefas de verificações de arquivos e a inicialização dos servidores e do cliente. Um shell executa os servidores e o cliente e o outro é responsável pela execução do Pyro. é necessário rodar primeiro o shell **run1.sh** e logo após o **run2.sh**. Abaixo será especificado algumas operações necessárias antes de executar o bash, como também sua execução.

Primeiramente, se for desejado rodar os sistemas em outro dispositivo será preciso liberar as portas que será utilizada para se comunicar com os sistemas, como usaremos um servidor de client side, não precisaremos nos comunicar com os servidores, assim será preciso liberar apenas as portas que usaremos que são as portas 5050 e 5000.

Antes de rodar o shell é preciso ter instalado especificamente o python 3.7, assim caso esteja no linux basta abrir o terminal e executar o comando:

```
$sudo apt-get python3.7
```

Após a instalação do python que será a nossa linguagem usada para programar o sistema, é necessário baixar algumas dependências, como o caso do gerenciador de pacotes python3-venv, assim para baixar este gerenciador basta executar o comando:

```
sudo apt-get install python3-venv
```

Agora o sistema estará pronto para rodar os dois Shell Script. A explicação de cada parte será explicado a seguir.

2.0.1 Shell Script 1

```
#!/bin/bash
trap ctrl_c INT

function ctrl_c() {
  fuser -k 9090/tcp
  fuser -k 5050/tcp
  echo "$(tput setaf 5)$(tput bold)I catch U!!!$(tput sgr0)"
}


source venv/bin/activate
python3 -m Pyro5.nameserver -p 5050
```

O primeiro bloco tem por função criar um listen para quando o usuário desejar terminar os processos dos sistemas(Servidores e Clientes), assim a função crtl_c finaliza os processos relacionados as portas abertas no uso dos sistemas.

O segundo bloco ativa o ambiente virtual e logo em seguida inicializa o Pyro na porta 5050 com o nome do server que foi criado no client.py.

2.0.2 Shell Script 2

```
#!/bin/bash
trap ctrl_c INT

function ctrl_c() {
  fuser -k 9090/tcp
  fuser -k 5050/tcp
  echo "$(tput setaf 5)$(tput bold)I catch U!!!$(tput sgr0)"
}

if [ ! -f tickets/passeios.json ] || [ ! -f tickets/aeroportos.json
   ] || [ ! -f tickets/hospedagens.json ]; then
  echo "$(tput setaf 3)$(tput bold)Gerando mocks$(tput sgr0)"
  cd tickets &&
  python3 ticketsGenerator.py
  cd ..
fi

if [ ! -d venv ];then
  echo "$(tput setaf 3)$(tput bold)Criando venv$(tput sgr0)"
  python3 -m venv venv
  source venv/bin/activate
  pip install pyro5
  pip install flask
  pip install Werkzeug
fi

source venv/bin/activate


python hospServer.py &
python passeioServer.py &
python ticketServer.py &
python client.py -p 5000
```

O primeiro bloco tem por função criar um listen para quando o usuário desejar terminar os processos dos sistemas(Servidores e Clientes), assim a função crtl_c finaliza os processos relacionados as portas abertas no uso dos sistemas.

O segundo bloco realiza a verificação de três arquivos que são os mocks gerados para as passagens de avião, hospedagem e passeios, assim esse bloco verificar se esse mocks já foram gerados, caso contrário ele executa um programa também em python ticketsGenerator, para gerar esse mocks.

O terceiro bloco verifica se o ambiente virtual que será usado para rodar os sistemas já está criado e com as devidas dependências, caso contrário ele realiza a criação de um ambiente virtual e instala as dependências necessárias, como o caso do Flask e Pyro.

Por fim, o quarto bloco inicializa os três servidores e o client side na porta 5000.

3 Implementação

O sistema continua possuindo uma interface gráfica através do Flask e a geração de script em Python que gera objetos de passagens, hospedagens e passeios com dados aleatórios. A única mudança feita foi o tipo de modelo implementado e a utilização de middleware. Nessa segunda parte foi utilizado o modelo de objeto distribuído, onde cada módulo seria um objeto. Esses objetos passam a ter interfaces remotas que contém métodos que ele exporta e que poderá ser acessado por outros objetos. Esses objetos estão em processos separados do processo que fez a requisição. Essa interface se chama interface remota, onde possui a especificação dos métodos disponíveis para a invocação pelos objetos que estão em outros processos, sendo assim possível a passagem de objeto como argumento de entrada ou como resultados.

3.1 Hierarquia de pastas

O projeto é composto por 4 pastas e 6 arquivos na raiz. As 4 pastas que são o static que possui arquivos de imagens e css usados no HTML(template do Flask), o template que possui os Templates das páginas web, o tickets que possui os arquivos jsons gerados pelo gerador que também está nessa pasta e a última pasta a venv que é o ambiente virtual do python.

Os 7 arquivos principais são os:

- hospServer.py : Server que gerencia as hospedagens;
- client.py: Inicializa uma nova conexão com os server e uma nova instância do client server(flask);
- passeioServer.py : Server que gerencia os passeio;
- ticketServer.py : Server que gerencia os voos;
- run1.sh : shell script que realiza algumas verificações e inicializa os servidores e o client server;
- run1.sh : shell script que ativa o ambiente virtual e executa o Pyro5

3.2 Python Pyro

Python Pyro é um middleware RMI que permite a construção de aplicações onde os objetos podem conversar entre si pela rede com o mínimo possível por parte do pro-

gramador. Apenas com uma simples chamada de método é possível promover a interação entre objetos que estão em máquinas diferentes sem a necessidade de programar a mais para isso, abaixo podemos ver um exemplo de utilização do Pyro retirado do próprio site do projeto:

Arquivo do servidor:

```
1     import Pyro4
2
3     @Pyro4.expose
4     class GreetingMaker(object):
5         def get_fortune(self, name):
6             return "Hello, {0}. Here is your fortune message:\n" \
7                 "Behold the warranty -- the bold print giveth and
                      the fine print taketh away.".format(name)
8
9     daemon = Pyro4.Daemon()             # make a Pyro daemon
10    uri = daemon.register(GreetingMaker)# register the greeting
         maker as a Pyro object
11
12    print("Ready. Object uri =", uri)   # print the uri so we can
         use it in the client later
13    daemon.requestLoop()                # start the event loop of
         the server to wait for calls
```

Arquivo do cliente:

```
1  import Pyro4
2
3  uri = input("What is the Pyro uri of the greeting object? ").strip()
4  name = input("What is your name? ").strip()
5
6  greeting_maker = Pyro4.Proxy(uri)         # get a Pyro proxy to the
      greeting object
7  print(greeting_maker.get_fortune(name))   # call method normally
```

Resultado:

```
1  \$ python greeting-client.py
2  What is the Pyro uri of the greeting object?  <<paste the uri that
      the server printed earlier>>
3  What is your name?  <<type your name; in my case: Irmen>>
4  Hello, Irmen. Here is your fortune message:
5  Behold the warranty -- the bold print giveth and the fine print
      taketh away.
```

Como pode ser observado, no servidor tudo que foi necessário ser feito foi usar o @Pyro.expose na classe que gostaríamos de tornar remota, criar um Daemon do Pyro e então é preciso registrar a classe nesse Daemon criado. Na parte do Cliente só foi necessário acessar a classe a partir do uri gerado pelo Daemon e pronto, o método get_fortune pode ser acessado normalmente como se fosse uma classe local.

3.3 Resultados obtidos nos Testes

Foi calculado uma média de tempo de espera para cada requisição em um total de 5 teste e a média dos resultados é mostrado a seguir:

- Tempo de espera para o servidor de Voos: 0,063259029s
- Tempo de espera para o servidor de Hospedagem: 0,031613636s
- Tempo de espera para o servidor de Passeio: 0,032914495s

Esse tempo será comparado com a média de tempo de espera da parte um do trabalho (sockets). Segue abaixo os resultados da parte um:

- Tempo de espera para o servidor de Voos: 0.06756556034088135s
- Tempo de espera para o servidor de Hospedagem: 0.009750843048095703s
- Tempo de espera para o servidor de Passeio: 0.018122196197509766s

4 Conclusão

Ao longo deste trabalho, pudemos ver em uma aplicação real alguns dos conceitos sobre middlewares RMI apresentados em aula, e também aprender mais sobre o uso da linguagem Python em servidores e sobre o uso do Pyro como o middleware RMI escolhido.

Comparando as duas partes do trabalho, temos que os tempos de espera nos servidores da parte 1 são visivelmente menores que os tempos de espera dos servidores com Pyro, isso se dá pelo fato de o python sockets é uma biblioteca de baixo nível com a qual implementamos apenas o necessário para que os serviços funcionassem, já o Pyro é um biblioteca de alta abstração onde várias features estão implementadas, podendo assim aumentar o tempo de resposta de algumas tarefas. Porém, em relação a implementação, o Pyro é muito superior, visto que a implementação do Pyro em um projeto pode ser feita usando poucas linhas de código e não é necessário modificar nada do código antigo para isso.

Outro ponto importante a se observar é que o middleware possui menos chance de erro de estrutura, já que o middleware oferece independência de Hardware do computador e a ocultação dos sistemas operacionais, o que oferta em conjunto uma independência de plataforma, além de possibilitar a utilização de várias linguagens de programação para realizar a implementação das partes que compõem o négocio do sistema distribuído.

5 Referências

Flask. Disponível em: <https://flask.palletsprojects.com/en/1.1.x/>. Acesso em: 25 de out. de 2020.

Bootstrap. Disponível em: <https://getbootstrap.com/>. Acesso em: 25 de out. de 2020.

ANDRADE,Ana Paula de. O que é Flask?. Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-flask/>. Acesso em: 25 de out. de 2020.

Flask. Disponível em: <https://flask.palletsprojects.com/en/1.1.x/installation/#install-instal Acesso em: 25 de out. de 2020.

Pyro 5. Disponível em: <https://pyro5.readthedocs.io/en/latest/>. Acesso em 13 de nov. de 2020.