MC833 - Projeto 3
Cliente e Servidor RMI

Luciano Zago - 182835
Vinicius Couto - 188115

Introdução

O projeto tem como objetivo criar uma comunicação RMI entre um cliente e um servidor. A aplicação consiste em um sistema para armazenar perfis de diversos usuários, com as seguintes informações: email (chave), nome, sobrenome, residência, formação acadêmica, habilidades e experiência profissional. O servidor deve ser capaz de receber as requisições do cliente, localizado em uma máquina diferente, e transmitir todas as informações disponíveis. O cliente pode visualizar todas as informações disponíveis e adicionar novas experiências em um perfil.

Sistema

1. Descrição Geral

O código foi dividido em três partes: *client*, *compute* e *engine*. A *client* contém o cliente RMI; a *compute* apresenta a interface remota (*compute.jar*) que define os métodos acessíveis ao cliente; e, por fim, a *engine* define a classe *ComputeEngine*, que consiste no executável do servidor, e a classe *Profile* utilizada para a estruturação e acesso dos dados dos perfis. Nota-se que a *engine* também contém uma classe *DataGen* que tem como objetivo gerar e serializar perfis de teste, resultando no arquivo *data.ser* que contém os dados a serem acessados pelo servidor.

A classe cliente, implementada em *client/Client.java*, tem como função buscar o registro criado no servidor e referenciar o objeto remoto *ComputeEngine* para ter acesso aos métodos listados pela interface *Compute*. Além disso, a classe *Client* realiza a leitura do comando realizado pelo usuário e repassa este para o método remoto *executeRequest*. Ao receber a *String* de resposta do servidor, imprime-a para o usuário.

As classes *ComputeEngine* contém o *setup* do RMI e o tratamento das *requests* do cliente, enquanto a *Profile* contém os dados e os métodos de acesso dos perfis; fazendo destas as principais componentes do servidor. O método *executeRequest*, pertencente a classe *ComputeEngine*, é acessado remotamente pelo cliente recebendo uma *String* como parâmetro da *request*. A partir da *String* em questão, são efetuadas chamadas dos métodos da classe *Profile* para acessar/modificar as informações desejadas de acordo com a *request* do usuário. O retorno da função *ExecuteRequest* trata-se de uma *String* contendo a resposta para a *request* do cliente.

A compilação e execução dos programas foram feitas como descrito no arquivo Makefile.

2. **Casos de Uso**

help	- Mostra os comandos disponiveis
1 <curso>	- Lista todas as pessoas formadas em um determinado curso
2 <cidade>	- Lista as habilidades dos residentes da cidade
3 <email> <experiência>	- Adiciona determinada experiência em um perfil
4 <email>	- Lista todas as experiências de um perfil
5	- Lista todas as informações de todos os perfis
6 <email>	- Lista todas as informações de um perfil

Estrutura de dados e armazenamento

Os perfis de teste foram armazenados através do uso de serialização (funcionalidade em Java que permite a escrita e leitura de instâncias a partir de arquivos). A partir da classe *DataGen.java* criou-se um *HashMap* de instâncias de *Profile*, assim, a cada perfil instanciado, inseriu-se o perfil ao *HashMap* usando o e-mail como chave. O *HashMap* foi serializado (escrito) no arquivo *data.ser* junto com os perfis que contém, permitindo o armazenamento persistente e estruturado dos perfis. Reitera-se que a cada alteração dos dados o *HashMap* modificado é serializado novamente, garantindo a persistência de mudanças nos dados. Para ler as informações, o *HashMap* do *data.ser* é desserializado e os perfis contidos neste consultados a partir do email do perfil desejado e dos métodos fornecidos pela classe Profile.
A classe *Profile* que abstrai a estrutura de dados possui a seguinte declaração de atributos:

```
public class Profile implements Serializable {
   private String email;
   private String nome;
   private String sobrenome;
   private String residencia;
   private String formacao;
   private String habilidades;
   private List<String> experiencias;
   [...]
}
```

Para cada atributo definiu-se um *getter* para recuperar a informação, e, no caso das experiências, também definiu-se um *setter* para adicionar experiências.

Implementação do RMI

O RMI [1] foi implementado visando o armazenamento de dados de forma persistente e a consulta destes a partir de um objeto remoto, além de concorrência entre clientes e confiabilidade na troca de mensagens. Devido ao fato de que o RMI (*Remote Procedure Call*) é uma API nativa do Java, sua implementação desfruta de um alto nível de abstração quando comparado com *socket programming* em *C*. Dado o alto nível de abstração do RMI, a maior parte do código consiste no gerenciamento de dados.

O servidor é inicializado com o RMI runtime, que se refere a um thread autônomo que responde requests de outras JVMs através de comunicação por *socket*. Ao executar o comando "*rmiregistry &"* o thread é inicializado na porta *default* (1099) e aguarda chamadas remotas, além de referenciar todos os métodos disponibilizados à acessos remotos no servidor. Também é necessária uma interface remota (a interface *Compute* no caso) que estenda da interface *java.rmi.Remote* e contenha os métodos remotos a serem disponibilizados para o cliente. Com isso, garante-se que as instâncias de classes que implementam *Compute* sejam tratadas como objetos remotos, possibilitando o envio de um *stub* (ao invés de uma cópia do objeto) para o cliente, e que o método *ExecuteRequest* (contido na interface remota *Compute*) destas instâncias sejam acessíveis à outras JVMs. O *stub* atua como uma referência local do objeto remoto no cliente. Nota-se que, como chamadas remotas possuem a *checked exception RemoteException* (um erro que é recuperável e, logo, deve ser tratado obrigatoriamente), a classe *java.rmi.RemoteException* também é importada na *Compute*.

Por fim, para garantir que o objeto remoto seja visível para outras JVMs, a função *main* de *ComputeEngine*, ao ser executada, cria um objeto a partir da classe *ComputeEngine* e utiliza o método estático *UnicastRemoteObject.exportObject* para realizar o "*binding*" do objeto criado com o registro RMI, retornando um *stub* para recebimento de chamadas remotas e tornando o objeto acessível à outras JVMs. Observa-se que a classe *UnicastRemoteObject* utiliza *sockets* TCP nos stubs gerados para o servidor/cliente, além de implementar atendimento *multithreaded* à chamadas concorrentes, garantindo a confiabilidade e concorrência do servidor.

O único método remoto disponibilizado, *ExecuteRequest*, se responsabiliza por *desserializar* um *HashMap* de perfis, identificar o método a ser chamado pela *request* e re-*serializar* o *HashMap* arquivo de dados *data.ser* para caso haja alguma *request* de escrita. A partir do identificador da *request*, a *ExecuteRequest* invoca um dos seis métodos disponíveis na *ComputeEngine*. Estes métodos usam os *getters* e *setters* das instâncias de *Profile* contida no *HashMap* desserializado para ler/escrever as informações desejadas. Para *requests* de leitura, a *ExecuteRequest* devolve uma *String* para o RMI repassar ao cliente com a resposta da operação.

Visando praticidade, supôs-se que todos os comandos fornecidos pelo cliente eram válidos dentro do escopo de operações do servidor; também assumiu-se que as conexões são sempre estáveis. A finalidade das suposições é de evitar tratamento de erros, o que aumentaria a complexidade do servidor e divergiria do objetivo do projeto. Além disso, a politica de segurança do Java foi desativada, através da remoção da instanciação da classe SecurityManager no cliente e na engine, para simplicar a implementação do projeto.

Resultados

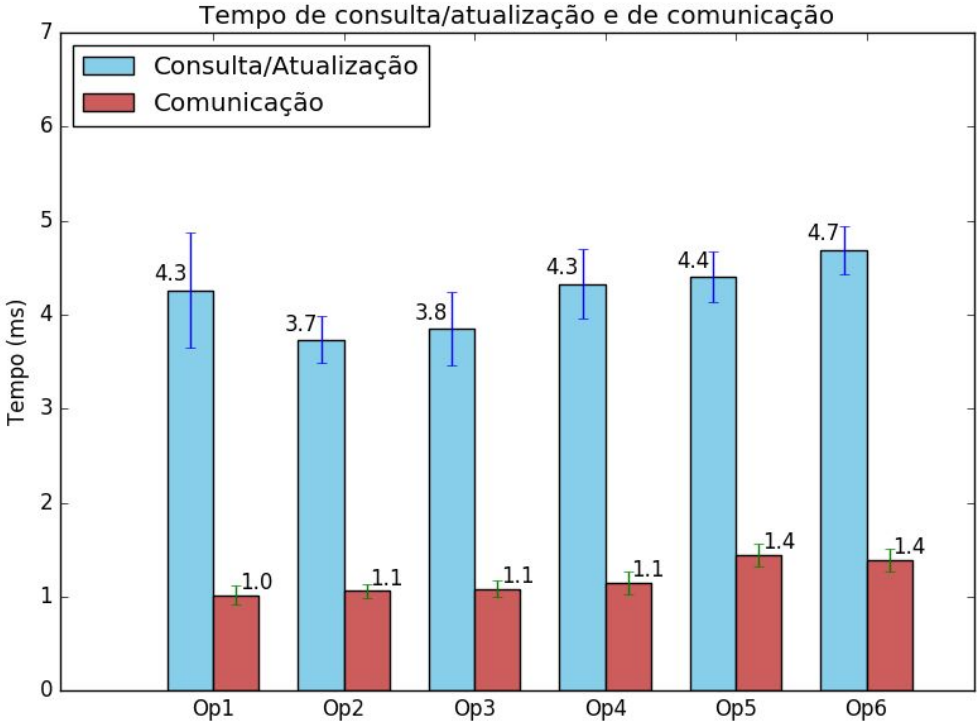
Os resultados correspondem à consultas realizadas em máquinas diferentes em uma rede local.

O cálculo do erro com intervalo de confiança de 95% foi realizado como $\sigma_M = 1.96 \times \frac{\sigma}{\sqrt{N}}$ sendo σ o desvio padrão e N o número total de iterações. O valor 1.96 é a constante que representa o intervalo de confiança de 95% [2].

Tempos de consulta/atualização do client						
Iterações	**Opções**					
-	**1**	**2**	**3**	**4**	**5**	**6**
1	7.124331	3.707414	5.363570	5.465883	6.816592	6.672291
2	6.881296	4.258532	6.419287	5.684103	6.846330	6.840368
3	6.012811	4.773182	6.946546	5.936630	5.941169	5.615614
4	4.507938	4.229424	5.655360	6.839857	4.956674	5.736047
5	4.278487	4.612018	5.782214	6.587547	4.283730	6.574533
6	4.174975	4.027690	5.907383	6.125118	4.427358	8.166734
7	4.467715	4.452831	5.902399	5.098644	5.249119	6.466915
8	4.279333	4.420325	5.528016	4.423364	5.511865	6.548907
9	4.618066	4.470990	6.148356	4.479924	5.305041	6.299548
10	4.479820	5.145762	4.241624	4.787000	6.089570	6.448726
11	4.933687	5.646268	4.281548	4.645940	6.036904	6.296750
12	5.756205	5.097786	4.252832	4.676499	6.309250	5.729689
13	6.419396	5.834436	4.096712	5.431302	5.973290	6.221278
14	10.303421	6.270196	4.267003	5.416550	6.495597	4.722303
15	5.211358	5.992162	3.902852	7.054809	6.325209	5.147656
16	4.668909	4.500954	4.179483	5.783030	6.474589	5.780350
17	4.319061	4.862443	4.167897	6.863155	5.785151	5.877540
18	4.224458	4.614767	4.201240	6.507902	6.284887	5.714651
19	4.573219	4.581701	3.893690	3.825976	5.565383	5.791653
20	4.302657	4.425273	3.477701	3.746417	6.348426	4.897879
Média (ms)	**5.276857**	**4.796208**	**4.930786**	**5.468983**	**5.851307**	**6.077472**
Erro (95%)	**0.657884**	**0.296747**	**0.448167**	**0.439566**	**0.315668**	**0.334421**

Tempos de operação do servidor						
Iterações	**Opções**					
-	**1**	**2**	**3**	**4**	**5**	**6**
1	5.648013	2.764415	4.105527	4.150822	4.895524	5.048927
2	5.902299	3.330545	5.066399	4.630795	5.176141	5.283527
3	5.030080	3.729928	5.613419	4.860065	4.218068	4.346473
4	3.661894	3.310878	4.581935	5.597089	3.7787	4.601373
5	3.439440	3.528299	4.647129	5.486911	3.274385	4.592216
6	3.494855	3.365703	4.751408	4.991122	3.272139	6.300092
7	3.468894	3.473346	4.712608	3.973996	3.67855	4.737029
8	3.334640	3.407175	4.275276	3.208343	3.952203	4.891216
9	3.522110	3.647125	5.009698	3.444892	3.912008	4.887809
10	3.714801	4.252366	3.090573	3.860854	4.833857	5.359705
11	4.068904	4.456181	3.050962	3.668363	4.7692	4.998066
12	4.391449	3.883648	3.120237	3.897011	4.990559	4.414866
13	5.093075	4.407885	3.08966	4.56923	4.923658	4.873371
14	9.199236	4.943753	3.378369	4.313989	4.857974	3.773536
15	3.805146	4.785464	3.037175	5.778309	4.535722	4.012972
16	3.638972	3.430865	3.16507	4.010278	4.70424	4.457281
17	3.352833	3.746375	3.082987	5.166141	4.170953	4.689994
18	3.555497	3.510430	3.074493	4.968053	5.248433	4.299803
19	3.550372	3.359753	3.200835	2.910949	4.088195	4.352884
20	3.347430	3.379204	2.89632	3.045484	4.796331	3.823197
Média (ms)	**4.260997**	**3.735667**	**3.847504**	**4.326635**	**4.403842**	**4.687217**
Erro (95%)	**0.616119**	**0.243934**	**0.390795**	**0.373852**	**0.267154**	**0.252629**

Tempos de comunicação						
Iterações	**Opções**					
-	**1**	**2**	**3**	**4**	**5**	**6**
1	1.476318	0.942999	1.258043	1.315061	1.921068	1.623364
2	0.978997	0.927987	1.352888	1.053308	1.670189	1.556841
3	0.982731	1.043254	1.333127	1.076565	1.723101	1.269141
4	0.846044	0.918546	1.073425	1.242768	1.177974	1.134674
5	0.839047	1.083719	1.135085	1.100636	1.009345	1.982317
6	0.680120	0.661987	1.155975	1.133996	1.155219	1.866642
7	0.998821	0.979485	1.189791	1.124648	1.570569	1.729886
8	0.944693	1.013150	1.252740	1.215021	1.559662	1.657691
9	1.095956	0.823865	1.138658	1.035032	1.393033	1.411739
10	0.765019	0.893396	1.151051	0.926146	1.255713	1.089021
11	0.864783	1.190087	1.230586	0.977577	1.267704	1.298684
12	1.364756	1.214138	1.132595	0.779488	1.318691	1.314823
13	1.326321	1.426551	1.007052	0.862072	1.049632	1.347907
14	1.104185	1.326443	0.888634	1.102561	1.637623	0.948767
15	1.406212	1.206698	0.865677	1.276500	1.789487	1.134684
16	1.029937	1.070089	1.014413	1.772752	1.770349	1.323069
17	0.966228	1.116068	1.084910	1.697014	1.614198	1.187546
18	0.668961	1.104337	1.126747	1.539849	1.036454	1.414848
19	1.022847	1.221948	0.692855	0.915027	1.477188	1.438769
20	0.955227	1.046069	0.581381	0.700933	1.552095	1.074682
Média (ms)	**1.015860**	**1.060541**	**1.083282**	**1.142348**	**1.447465**	**1.390255**
Erro (95%)	**0.100033**	**0.077996**	**0.086935**	**0.122500**	**0.120674**	**0.120734**



Os tempos foram obtidos rodando o servidor com o seguinte comando:

java engine/ComputeEngine -Djava.rmierver.hostname=localhost > output_server.txt

E rodando o cliente com o seguinte comando:

java client/Client {ip} output_client.txt < input.txt

Os tempos de Consulta/Atualização do servidor são elevados e próximos para todas as operações executadas pelo servidor, considerando que a margem de erro de todos os pares de operações se interseccionam (com exceção do par <**Op2**,**Op6**>). Essa observação deriva de dois fatores. Primeiramente, o uso da serialização (e desserialização) do *HashMap* com todos os perfis para qualquer consulta realizada, uma vez que a função *ExecuteRequest* sempre realiza a desserialização e serialização de todos os perfis existentes no banco de dados. Além disso, temos que Java trata-se de uma linguagem alto nível, logo sua performance não é tão rápida quando outras linguagens baixo nível como *C*, por exemplo. Assim, temos que o tempo de consulta é regido pela serialização e alto nível de abstração de Java, tornando as diferenças de complexidade de diferentes *requests* menos notáveis nos tempos de consulta.

Analisando os tempos de comunicação, também nota-se que estes são elevados e similares. Analogamente aos tempos de consulta, justifica-se essa observação pelo fato de que o meio de comunicação utilizado (RMI) consiste em uma API que abstrai diversas inconveniências de comunicação por *sockets*, facilitando a implementação e uso deste tipo de comunicação. Mas as facilidades disponibilizadas vem com um custo de performance no *delay* para a troca de mensagens, uma vez que o RMI acrescenta um *overhead* de

pré-processamento para realizar a comunicação. Devido ao alto nível de abstração fornecido pelo RMI, temos que o *overhead* que esta abstração acarreta rege o tempo de comunicação, todavia, nota-se que operação como a **Op5** e **Op6** apresentam tempos significativamente maiores, o que é coerente uma vez que estas exigem a recuperação de todos os perfis e a busca de um perfil específico no *HashMap*, respectivamente.

Conclusão

O projeto criou uma comunicação RMI entre um cliente e servidor, localizados em máquinas diferentes em uma rede local, para atender os requisitos das seis operações citadas e as condições de paralelismo e persistência.

Analisando-se os resultados, nota-se que os tempos de comunicação foram sempre menores do que os tempos de consulta/atualização, o que é consistente com o esperado, pois o tempo de comunicação está contido no tempo total de consulta. O tempo de processamento (diferença entre consulta e comunicação) mostrou-se elevado, o que e coerente considerando-se que Java é uma linguagem de alto nível e que foram utilizadas ferramentas como serialização que acrescentam *overhead*.

Dessa forma, o projeto atendeu os requisitos para um servidor TCP concorrente e apresentou uma performance coerente com o esperado de um meio de comunicação alto nível. A persistência e confiança na transmissão de arquivos foi garantida, em ambos servidor e cliente, pelo RMI (que faz uso de protocolo TCP), assim como a concorrência através do uso de *threads*.

Referências

1. The Java™ Tutorials: RMI (https://docs.oracle.com/javase/tutorial/rmi/index.html)
2. Confidence Interval on the Mean (http://onlinestatbook.com/2/estimation/mean.html)