MC833 - Projeto 3

Carlos Avelar (168605) e Tiago Loureiro Chaves (187690)

27 de Junho, 2019

1 Introdução

Este projeto teve como objetivo a implementação e comparação de um sistema cliente-servidor concorrente por threads, utilizando a interface Java RMI [1].

O sistema baseia-se em uma rede de perfis profissionais com opções de consulta e alteração de dados oferecidas pelo servidor ao cliente. O cliente pode requisitar opções (p.e. listagem de um perfil, listagem de todos os perfis, etc.) até que decida finalizar a comunicação.

Desse modo, cada opção solicitada pelo cliente leva à chamada e à execução de uma função remota (no servidor), de forma análoga a uma RPC [2]. Porém, isso é realizado por meio do RMI (*remote method invocation*), que abstrai o *marshalling / unmarshalling* e a transferência dos dados de uma máquina virtual Java para outra.

Por fim, realizou-se uma análise do tempo médio de comunicação das opções, considerando tanto seu tempo de execução no servidor, como o tempo total observado no cliente, e comparou-se os resultados obtidos com sistemas cliente-servidor implementados na linguagem C com os protocolos UDP e TCP.

2 Sistema

2.1 Inicialização

Para iniciar-se o servidor deve-se primeiramente executar `rmiregistry [PORT]`, que inicializa o *naming service* utilizado pelo RMI para realizar *bind* de objetos remotos. O argumento `PORT` é opcional, sendo a porta 1099 a padrão.

Em seguida, compila-se o projeto com `./compile.sh` e então executa-se o servidor:

`java Server [-p PORT] [-a ADDRESS]`,

onde `-p` define a porta `PORT` utilizada e `-a` indica o nome `ADDRESS` associado ao endereço em que o servidor executa (`localhost` por padrão).

De forma análoga, o cliente é executado com: `java Client [-p PORT] [-a ADDRESS]`. Nota-se que as opções `-p` e `-a` são opcionais, porém a mesma porta `PORT` utilizada no `rmiregistry` deve ser passada para ambos cliente e servidor.

Assim, após iniciados (mesmo que em máquinas distintas) é possível a execução de métodos remotos.

Instruções de como iniciar o sistema também podem ser vistas em: https://github.com/laurelkeys/computer-networks/tree/master/project3

2.2 Descrição geral

Ao executar-se o cliente, é *printado* no terminal do usuário a lista de opções disponíveis:

- (1) listar todas as pessoas formadas em um determinado curso;
- (2) listar as habilidades dos perfis que moram em uma determinada cidade;
- (3) acrescentar uma nova experiência em um perfil;
- (4) dado o email do perfil, retornar sua experiência;
- (5) listar todas as informações de todos os perfis;
- (6) dado o email de um perfil, retornar suas informações;
- (7) sair.

Considera-se que o usuário digita corretamente o número da opção que deseja, e possíveis argumentos que elas necessitam (p.e. o curso desejado, na opção 1).

O cliente então, baseado na opção escolhida, invoca o método apropriado de sua variável do tipo `DataKeeper` (veja `utils/DataKeeper.java`), interface implementada pelo programa servidor que estende `Remote` [3] para permitir a execução remota de métodos com RMI.

Logo, de forma transparente, o método é executado na máquina virtual Java (JVM [4]) onde roda o servidor e seu resultado é obtido na máquina cliente, que então o imprime no terminal.

Por fim, as opções disponíveis são novamente mostradas ao cliente, e repete-se o fluxo descrito acima, até que o usuário escolha sair (opção 7).

2.3 Especificidades RMI

Uma visão geral do funcionamento RMI é ilustrada na Figura 1:

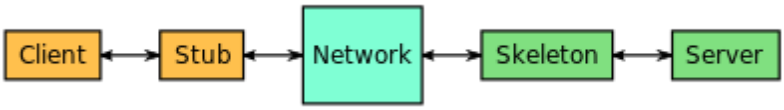

Figura 1: Arquitetura de uma aplicação Java RMI

Para possibilitar a comunicação entre as JVMs cliente e servidor, é criado um *Registry* de objetos remotos [5], que funciona como um servidor de resolução de nomes (similar ao DNS), e então métodos de interfaces que estendam `Remote` podem ser chamados.

Ao chamar um método remoto, o *Stub* cliente faz o *marshalling* dos dados e os passa para a camada de referência remota (RRL) que transmite os dados para a máquina virtual do servidor utilizando o protocolo TCP da camada de transporte. Ao receber os dados, a RRL do servidor passa-os para o *Skeleton* que faz o *unmarshalling* dos parâmetros recebidos e invoca o método necessário.

Finalmente, após calculado o resultado, ele é enviado de volta ao cliente pelo caminho reverso, de forma análoga.

3 Armazenamento e estrutura de dados do servidor

Diferente dos projetos anteriores ([6, 7]), os dados são armazenados no servidor em um objeto do tipo `HashMap<String, Person>`, ao invés de utilizar-se um banco de dados (p.e. SQLite). Assim, mapeamos os perfis, objetos da classe `Person` criada, por seus emails (`String`s).

Os dados inseridos são os seguintes (email, nome, sobrenome, cidade, formação, habilidades e experiências):

uno@mail.com	tres@mail.com	cinco@mail.com
Uno	Tres	Cinco
Dos	Cuatro	Seis
Campinas	Campinas	Seattle
Linguistics	CS	CS
Acoustic Engineering, English	Code, Read, Write	English, Reap, Sow, Spanish
Research, Work	Study, Work	Study, Study more, Work

4 Detalhes da implementação do servidor

Como mencionado na Seção 2.3, a classe `Server` implementa a interface `DataKeeper` que, por sua vez, implementa a interface RMI `Remote`.

Seis métodos são definidos em `DataKeeper`, referentes às opções existentes, e todos tem `DataResult` como tipo de retorno, uma interface feita para garantir que: 1) todos os valores retornados sejam `Serializable` (necessário para que o *Stub/Skeleton* possa transmití-lo); 2) defina-se uma função `printable()`, chamada no cliente, que especifica a forma como os dados serão impressos (veja as classes que implementam `DataResult` em `utils/results/options/`).

Quando o servidor é criado, chama-se a função estática `rebind` da classe `Naming` [8] para ligar um nome a uma nova instância de `Server`. Já no cliente, utiliza-se a função `lookup` para obter-se o *Stub* associado ao objeto do servidor, passando o mesmo nome usado no servidor como parâmetro, para que possa-se então realizar a chamada remota de métodos.

A Figura 2 elucida os conceitos mencionados acima, mostrando os principais componentes da implementação RMI.

```java
class Client {
    private static DataKeeper server;
    public static void main() throws Exception {
        String name = // ...
        server = (DataKeeper) Naming.lookup(name);
        // read option, call server.optionMethod(...), repeat
    }
}
class Server extends UnicastRemoteObject implements DataKeeper {
    public static void main() {
        String name = // ...
        Naming.rebind(name, new Server());
    }
}
interface DataResult<T> extends Serializable {
    T getData();
    String printable();
}
interface DataKeeper extends Remote {
    DataResult optionMethod(...) throws RemoteException;
    // ...
    DataResult optionMethod(...) throws RemoteException;
}
```

Figura 2: Principais componentes da implementação RMI

5 Resultados obtidos

Executando-se os programas cliente e servidor em máquinas distintas, porém na mesma rede (no laboratório CC00 do IC3), marcou-se 100 vezes, para cada opção, os tempos total (visto pelo cliente) e de processamento no servidor, a partir dos quais podemos estimar o tempo de comunicação conforme mostra a Figura 5.

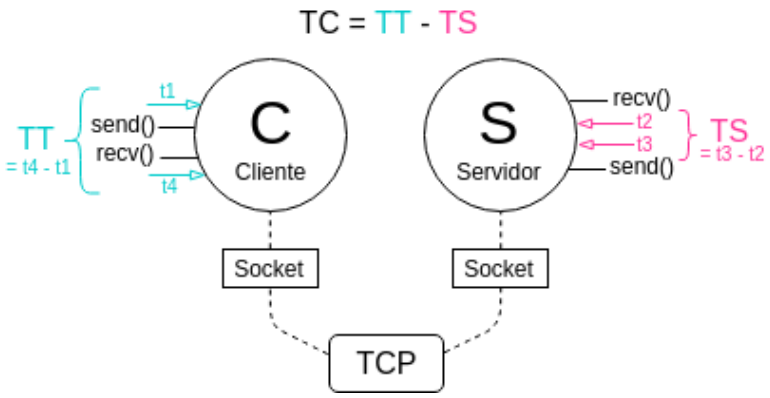

Figura 3: Tempos de comunicação (TC), total (TT) e de processamento no servidor (TS)

A tabela abaixo mostra os resultados do experimento, sendo *Avg Com. Time* o tempo de comunicação (TC) e *Avg Op. Time* o tempo de processamento por opção no servidor (TS). O desvio padrão e intervalo de 95% de confiança são referentes ao TC:

Op.	Avg Com. Time (ms)	Std Deviation (ms)	95% Conf Interval (ms)	Avg Op. Time (ms)
1	3,5467	1,5698	3,5386 to 3,5547	1,2000
2	3,2400	2,0356	3,2272 to 3,2528	1,0600
3	3,0267	0,7347	3,0213 to 3,0320	1,7467
4	3,0700	0,8558	3,0646 to 3,0754	1,0600
5	3,3520	1,1017	3,3458 to 3,3582	1,2800
6	3,7680	1,5141	3,7595 to 3,7765	1,3840

Apresenta-se, no Gráfico 4, para cada operação o tempos de comunicação calculado, a média das execuções (em vermelho), e barras delimitando o intervalo com nível de 95% de confiança.

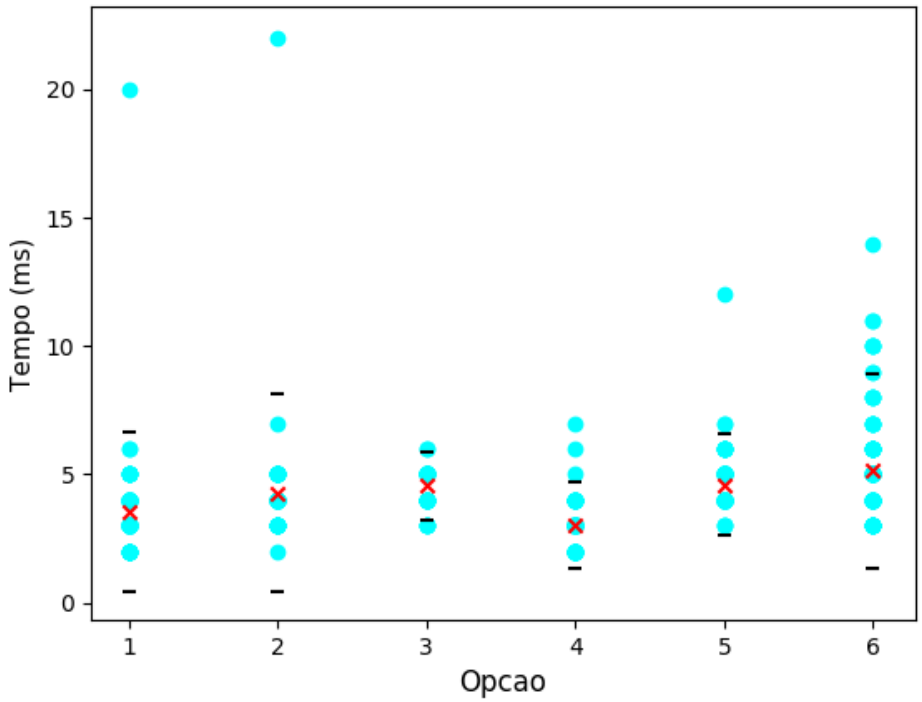

Figura 4: Gráfico com os tempos de comunicação

6 Comparação das tecnologias

A implementação do servidor de RMI em Java é rápida e direta, o código tem cerca de 600 linhas, sendo que a parte onde é efetivamente feita a comunicação o processamento corresponde a apenas metade dessas linhas (considerando o cliente e servidor juntos). Para se ter uma ideia melhor da comparação, apenas o servidor UDP/TCP em C somava 460 linhas [6, 7].

Em Java praticamente toda a parte da comunicação fica implícita, além disso, o tratamento de *Strings* é muito mais simples, fazendo com que o tamanho do código seja bem menor, e que sua implementação seja mais direta e feita em mais "alto nível".

Notamos que os tempos de comunicação do RMI são mais próximos dos tempos resultantes ao utilizarmos um sistema UDP iterativo do que um sistema cliente-servidor concorrente por threads

com o protocolo TCP (o qual apresenta tempos maiores), tendo como referência o Projeto 2 [7]. Entretanto, comparando-o com a implementação TCP do Projeto 1 [6], observamos tempos similares para a maioria das operações, já que o RMI usa o protocolo TCP na camada de transporte.

7 Conclusão

Após implementarmos o sistema descrito com três métodos diferentes (RMI em Java, UDP e TCP em C) nos três projetos da disciplina, concluímos que RMI é a opção mais adequada quando não há restrições para o tipo de implementação (p.e. podemos utilizar JVMs), pois é muito mais fácil e rápido de implementar e manter, e, além disso, a performance atingida (medida pelos tempos de comunicação) é semelhante à dos métodos implementados em C.

Referências

[1] "Java remote method invocation — Wikipedia, the free encyclopedia," accessado em junho de 2019. [Online]. Available: https://en.wikipedia.org/wiki/Java_remote_method_invocation

[2] "Remote procedure call — Wikipedia, the free encyclopedia," accessado em junho de 2019. [Online]. Available: https://en.wikipedia.org/wiki/Remote_procedure_call

[3] "2.4.1 The java.rmi.Remote Interface," accessado em junho de 2019. [Online]. Available: https://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmi-objmodel5.html

[4] "Java virtual machine — Wikipedia, the free encyclopedia," accessado em junho de 2019. [Online]. Available: https://en.wikipedia.org/wiki/Java_virtual_machine

[5] "rmiregistry - The Java Remote Object Registry." [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/rmiregistry.html

[6] C. Avelar and T. Chaves, "MC833 - Projeto 1." [Online]. Available: https://github.com/laurelkeys/computer-networks/blob/master/reports/relatorio_projeto1.pdf

[7] ——, "MC833 - Projeto 2." [Online]. Available: https://github.com/laurelkeys/computer-networks/blob/master/reports/relatorio_projeto2.pdf

[8] "Naming (Java Platform SE 7)," accessado em junho de 2019. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/rmi/Naming.html