1. Introdução

Na aula TP de SDist desta semana, trabalharemos no exercício apresentado em [1]. O principal objetivo desta aula é que vocês entendam os conceitos de base da comunicação sobre Java RMI.

No final de [1] encontram um tutorial útil sobre comunicação Java RMI (este [2]).

Tanto o exercício desta aula como o exemplo do tutorial Oracle [2], concentram-se no desenvolvimento de uma interface RMI entre um cliente e um servidor que poderá depois, de alguma forma, ser reutilizada no primeiro projecto avaliado.

(especificamente na comunicação entre o *TestClient* e o *Peer*, caso pretendam implementar essa comunicação usando o RMI).

2. Comunicação RMI

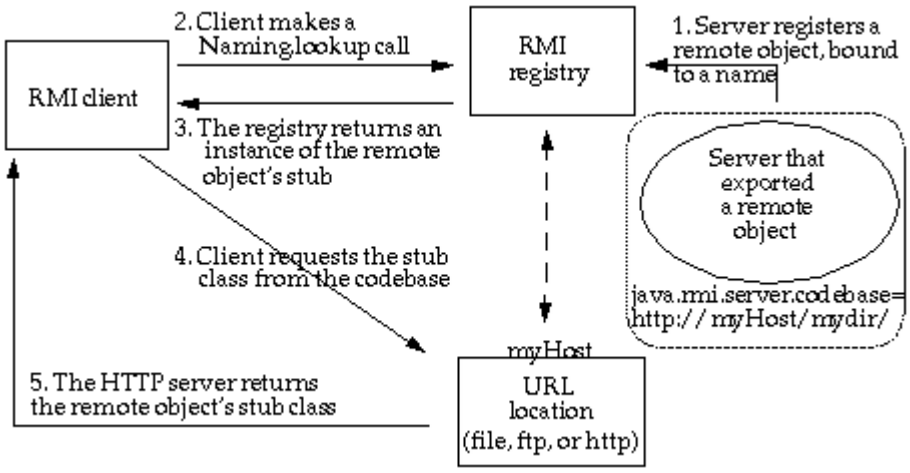

Figure 1 – Esquema Básico de Comunicação por Java RMI

Na comunicação RMI, a interacção entre um servidor e um cliente é assistida (numa fase inicial) por outra aplicação, i.e. o *rmiregistry*.

O *rmiregistry* é um serviço de registo de nomes (nomes de outros serviços) para *bootstrap* da comunicação entre clientes e servidores. O *rmiregistry* regista assim uma associação entre nomes de serviços e os *remote objects* que os implementam, mais especificamente entre nomes de serviços e as referências para esses *remote objects* (designadas como os *stubs* desses *remote objects*).
Os servidores procedem ao registo dessas associações, i.e. inscrevem no *rmiregistry* pares de *nome-de-serviço/stub*.
Os clientes, tanto no *host* local como em *hosts* remotos, contactam o *rmiregistry* para obterem os *stubs* de serviços específicos, que depois empregam para invocar tais serviços.

No entanto, os *stubs* (ou as referências para *remote objects*) registados no *rmiregistry* não são suficientes para que o cliente possa aceder ao servidor. O cliente precisa dos ficheiros *.class* com a definição da classe do *stub* (mais especificamente a classe que implementa o serviço remoto) e das restantes classes de que esta necessite.

No contexto de Java RMI, o local/serviço onde estes ficheiros *.class* estão armazenados, para serem disponibilizados ao lado cliente, chama-se *codebase* (a qual é independente do *rmiregistry*)

A *codebase* pode ser assim definida como a fonte a partir da qual os ficheiros *.class*, com as implementações dos artefactos de comunicação, podem ser carregados para uma máquina virtual (local ou remota). Para obterem uma informação mais detalhada sobre a *codebase* e sobre como o lado do servidor deve lidar com ela, podem consultar a página indicada em [3].

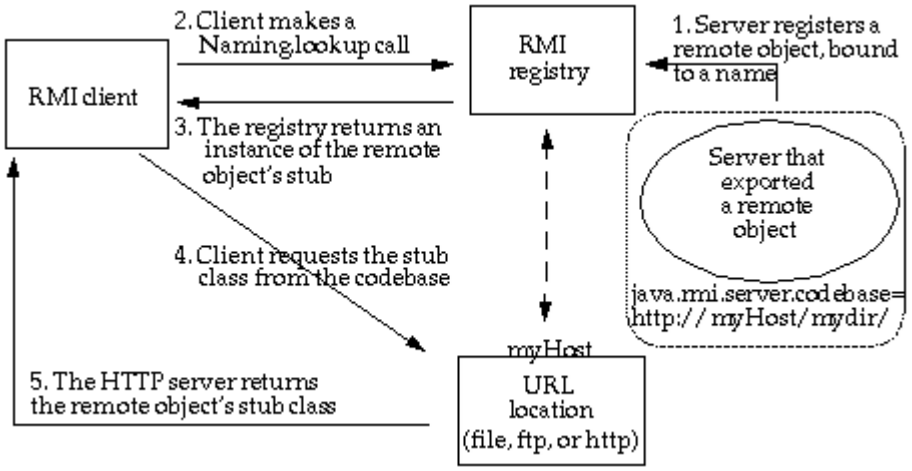

Figure 2 – Repetição da Figure 1

A operação básica da comunicação em RMI Java é a seguinte (de [3]):

1. O servidor registra um *remote object*, associando-o a um nome, no *rmiregistry*. O que é registado no *rmiregistry* não é o efectivo *remote object*, é apenas uma referência para um *remote object* (ou *stub*). Esse *stub* é gerado de forma dinâmica (programaticamente) no servidor.
   Para que o *stub* possa ser executado no lado cliente é necessário que este (o cliente) consiga obter a informação em falta no *stub* (os ficheiros *.class* com as definições das classes que implementam o serviço remoto) que está armazenada na *codebase*. O que acontece em Java RMI é que a localização dessa *codebase*, é anotada no *stub* quando este é produzido (é daí que o cliente a obtém).
   Assim, aquando de construção dinâmica do *stub* pelo lado servidor, a sua VM tem de saber onde está a *codebase*. Para isso esta propriedade, da VM em questão, deve ser previamente estabelecida (configurando a propriedade *java.rmi.server.codebase* na linha de comando ou programaticamente).
   Só depois de estar configurada a referida propriedade, e gerado o *stub* é que se pode proceder ao registo efectivo.

2. O cliente solicita, ao *rmiregistry*, uma referência para um *remote object* específico.

3. O *rmiregistry* retorna a referência (*stub*) em questão.

4. O cliente trata de obter a(s) definição(s) da(s) classe(s) necessárias à execução do *stub*. Se estas definições (*.class*) puderem ser obtidas da *classpath* do cliente (que é sempre pesquisada antes da *codebase*), o cliente carregará a classe localmente. Caso contrário o cliente tentará obter as definições de classes da *codebase*, empregando o valor da *codebase* (URL) que está anotado no próprio *stub*.

5. A definição de classe para o *stub* (e quaisquer outras classes necessárias) é retornada ao cliente.

6. Neste ponto, o cliente tem todas as informações necessárias para invocar os métodos oferecidos pelo *remote object*. A instância do *stub* funciona assim como um *proxy* para o

remote object que executa do lado servidor. Desta maneira, e diferentemente de um *applet* que recorre a uma *codebase* para obter código remoto e executá-lo na sua VM local, o cliente Java RMI recorre a uma *codebase* para obter código que desencadeia a execução de código noutra VM potencialmente remota.

3. Exercício do Lab 3

3.1 Arquitectura

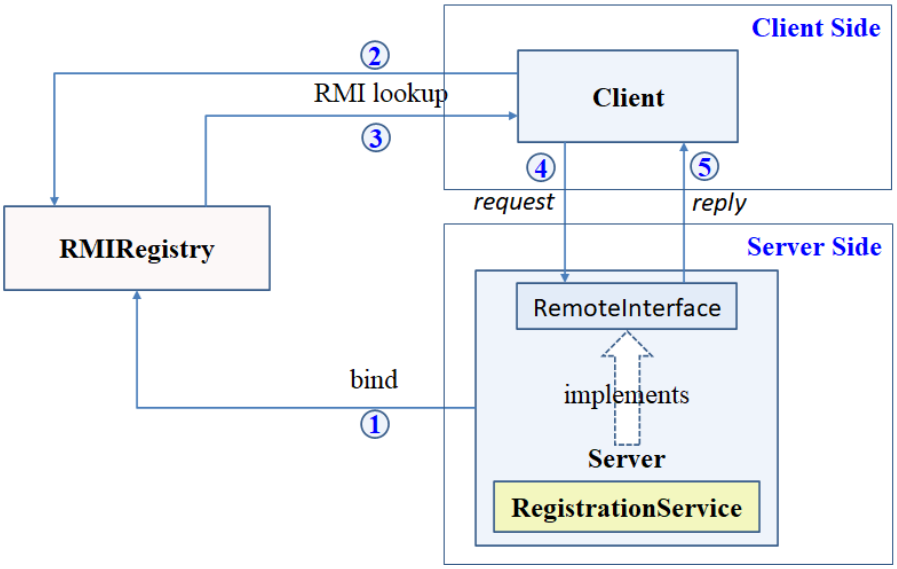

Figure 3 – Arquitectura do Lab3

A imagem acima apresenta uma descrição simples da arquitetura que deverão implementar no lab 3 (bem como no tutorial da Oracle [2]). É muito próximo daquilo que já foi apresentado na imagem anterior, e assim, também é a sua explicação.

3.2 Implementação

As etapas para desenvolver uma aplicação cliente-servidor utilizando Java RMI são as seguintes:

1. Definir a interface remota oferecida pelo servidor. Terão assim de definir um interface que estenda o interface *java.rmi.Remote*. Estender este interface implica as seguintes restrições:
   a) todos os métodos declarados (no interface do serviço) devem lançar uma excepção do tipo *java.rmi.RemoteException* (essa excepção é lançada pela JVM no caso de problemas de comunicação entre as JVMs);
   b) os argumentos e valores de retorno de todos os métodos declarados devem implementar a interface *java.io.Serializable*.

2. Desenvolver a classe do servidor, ou seja, a classe que implementa a interface definida no ponto anterior. Esta classe (ou a classe que inicia a execução do servidor) deve efectuar os seguintes passos:
   a) fazer, programaticamente, o set da propriedade *java.rmi.server.codebase* com o valor da *codebase* (para que este valor seja conhecido pela VM e possa ser anotado no *stub* aquando da sua criação);
   b) instanciação do "*remote object*". Para isso o primeiro passo é criar uma instância da classe servidor (usando o seu constructor). Depois é necessário "exportar" o *remote object*, ou seja, colocar o serviço a correr numa determinada porta e produzir o

respectivo *stub*. As duas tarefas são realizadas pelo método método *static export(….)* da classe *java.rmi.server.UnicastRemoteObject.* O valor de retorno deste método é o *stub*;

c) registo do *stub*, associado a um nome, no *rmiregistry*. Para isso existem duas opções:
   a. empregar os métodos *static bind(..)* ou *rebind(…)* da classe *java.rmi.Naming*. Estes permitem efectuar o registro de uma interface remota no *rmiregistry*;
   b. empregar o método *getRegistry()* da classe *java.rmi.registry.LocateRegistry*. Obtém-se assim uma referência a um objecto que implementa o interface *java.rmi.registry.Registry* (ou seja, retorna um objeto que representa o *rmiregistry*), a qual oferece os métodos descritos acima *bind()* e *rebind()*.

Em ambas as situações é preferível usar *rebind(…)*, pois *bind(…)* lançará uma excepção caso nome registrado anteriormente (ou seja, uma *String*) seja reutilizado. *rebind(…)* reassocia o nome ao "novo" *remote object*.

3. Desenvolver o cliente. Este deverá executar os seguintes passos:
   a) obter, do *rmiregistry*, o *stub* de acesso ao serviço remoto, ou seja, fazer o *lookup* do serviço. Para isso há duas alternativas (de forma semelhante ao que é descrito em 2c):
      a. empregar o método *static lookup(..)* da classe *java.rmi.Naming*. Este permite a obtenção de um *stub* para um *remote object* registrado no *rmiregistry*.
      Neste caso o argumento do método *lookup(…)* deve assumir o seguinte formato: //<host_name> [: < port_number >] / <obj_name> - em que < port_number > é o número da porta usada pelo *rmiregistry* (por defeito é 1099);
      b. empregar o método *getRegistry()* da classe *java.rmi.registry.LocateRegistry*. Obter assim uma referência a um objecto que implementa o interface *java.rmi.registry.Registry* e assim ter acesso ao método *lookup(…)*.
      Neste caso o argumento *String* do método *lookup(…)* pode ser apenas <obj_name>, pois os valores de <host_name> e de < port_number > podem ser especificados usando os argumentos transmitidos ao método *getRegistry()*;

   Ambos os métodos retornarão uma instância de um objecto que implementa o interface do serviço definido no ponto 1.
   b) após os passos anteriores o cliente estará então pronto para invocar os métodos remotos oferecidos pelo serviço de DNS de acordo com a assinatura dos mesmos. Essa invocação pode levar a uma excepção do tipo *java.rmi.RemoteException*, e deve, portanto, ser feita dentro de um bloco *try-catch*.

3.3 Execução

Notas relativas à execução do lab 3 (execução de rmiregistry, servidor e cliente):

1. O *rmiregistry* deve estar a correr antes de se executar o servidor. Portanto, primeiro corre-se o *rmiregistry*, depois o servidor e depois o cliente;

2. O *rmiregistry* pode ser executado a partir da linha de comando ou programaticamente. Recomenda-se a primeira alternativa;

3. Para poder executar o *rmiregistry* (a partir da linha de comando), o seu executável deve estar na *classpath* da *shell*. É necessário verificar se o diretório *bin* da instalação Java (onde se encontra o *rmiregistry.exe*) foi adicionado à variável de ambiente *Path* (esse é o nome da variável no Windows);

4. Para simplificar a execução do *rmiregistry* este deve ser executado no mesmo diretório em que o servidor é executado (assumindo que este é executado na base da árvore directorial que contém os *.class*).

Executar o *rmiregistry* no diretório em questão é vantajoso pela seguinte razão: se nada for especificado, o *rmiregistry* assume que seu diretório de execução é também a *codebase*. Dessa maneira, se o *rmiregistry* for executado no mesmo diretório do servidor, o primeiro encontrará automaticamente os ficheiros *.class* com a implementação do serviço;

5. O *rmiregistry* também pode ser executado a partir de outro diretório. Nesse caso, a propriedade *java.rmi.server.codebase* deve ser configurada. Para isso, no momento da inicialização do *rmiregistry*, a codebase deve ser especificado (por exemplo, *rmiregistry -J-Djava.rmi.server.codebase=file:///a/b/SDist_2018/L03/bin/* onde file:///a/b/SDist_2021/L03/bin/ é um valor a adaptar às características da vossa implementação;

6. No que respeita à execução do servidor, caso a propriedade *java.rmi.server.codebase* não seja configurada programaticamente, isso deverá então ser feito na linha de comandos tal como é explicado no ponto 5:

7. Em relação ao cliente, poderá ser necessário configurar a política de segurança para que a VM carregue "executáveis" remotos. Para isso recomenda-se a consulta do ponto D da seção 6.0 da referência [3].

4. Referências

[1] Lab on RMI – https://web.fe.up.pt/~pfs/aulas/sd2021/labs/l03/rmi_l03.html

[2] Getting Started Using Java RMI, Oracle Tutorial, https://docs.oracle.com/javase/6/docs/technotes/guides/rmi/hello/hello-world.html

[3] Dynamic code downloading using Java RMI, Oracle, https://docs.oracle.com/javase/6/docs/technotes/guides/rmi/codebase.html