RMI em Java
Com Java RMI, podemos ter uma aplicação em que se invoca um método de um objeto remoto (que pertence a um processo diferente, numa outra *Java Virtual Machine*, talvez num computador diferente).

Camadas do Modelo Java RMI

aplicação servidor	aplicação cliente
dispatcher + skeleton	stub de cliente/proxy
camada de referências remotas	camada de referências remotas
camada de transporte	camada de transporte

Componentes "novos":

proxy
fica do lado do cliente (que invoca)
torna a invocação remota transparente para o cliente
faz marshalling de argumentos... e unmarshall retorno
- GERADO AUTOMATICAMENTE (comando `rmic` para stubs estáticos / `java` se os stubs são dinâmicos)

dispatcher
recebe um pedido do módulo de comunicação
invoca o método respectivo no skeleton
- FICA JUNTO AO SKELETON
- GERADO AUTOMATICAMENTE (`rmic/java`)

skeleton
implementa os metodos da Remote Interface
faz o unmarshalling/marshalling de objetos (in/out)
passando os argumentos na invocação local sobre o Objeto Remoto. Faz marshalling do objeto de retorno ou eventual excepção para devolver como resposta.
- GERADO AUTOMATICAMENTE (`rmic/java`)

Do ponto de vista do programador:

- **Servidor**

interface remota
objeto remoto: implementação da interface remota (parte funcional)
dispatcher + skeleton (gerados automaticamente)

- **Cliente**

proxy do objeto remoto (gerado automaticamente)

O que é necessário nas aplicações:

Servidor faz uma inicialização:

- criar instância de objeto remoto
- registar o objeto remoto no BINDER

Cliente:

- inicia com um lookup ao objeto no BINDER
- fica com a referência remota, associada ao Proxy
- pode invocar

A classe que implementa a Interface Remota (cujas instâncias são objetos remotos)
- estender `UnicastRemoteObject`
- alternativa: `UnicastRemoteObject.exportObject(obj)` , no Servidor

A classe `UnicastRemoteObject` inclui funcionalidade relacionada com *marshalling* e operações necessárias para o RMI.

Métodos na Interface Remota:
Os argumentos ou retorno dos metodos a invocar devem ser serializáveis:
- tipos primitivos
- ... implements Serializable (como String, Vector ou outras...)

O **Binder** ou **Registry** por defeito fica a escutar no porto 1099
podemos mudar isso com um argumento no `rmiregistry`
e atualizando o local do (re)bind e lookup
com um URI do tipo: "//host:port/name" OU "rmi://host..."
em `java.rmi.Naming`
(ou `java.rmi.registry.Registry`)

Nota: Nas versões mais recentes de Java é possível ter o serviço de nomes do RMI **a executar na mesma aplicação** que o servidor (aplicação com o objeto remoto).
Para simplificar e aclarar as responsabilidades de ambos, nesta UC, **separamos** os papeis de servidor e binder.

SecurityManager ou (RMISecurityManager)
gestor de segurança, que garante que as classes lidas remotamente (ex: por *reflection*) não violam as regras de proteção definidas na "java.policy" - configurável
(escrita de ficheiros, acesso à rede, etc...)

Se não for especificado, as aplicações não poderão usar classes para além das disponíveis na sua Classpath.

Um SecurityManager baseia-se em REGRAS, num ficheiro java.policy:
$JAVA_HOME/jre/lib/security/java.policy
~/.java.policy

com o formato:

```
grant {
  permission java.net.SocketPermission "localhost:1024-", "connect";
  permission java.net.SocketPermission "localhost:1024-", "accept";
};
```

NOTA: Em RMI, o servidor fica **CONCORRENTE**!

NOVOS COMANDOS:
`rmiregistry`

---

exercício 1
(Garanta a existência de JDK adequado.)

Imagine um serviço com duas operações:
- obter a primeira palavra de uma frase
- obter um vector com cada palavra presente numa frase

1.a- Obtenha o código inicial:
- formatos (projeto NetBeans)

1.b- Observe no código-fonte a seguinte sequência:
- declaração dos métodos remotos (na Interface Remota)
- implementação da funcionalidade (classe do Objeto Remoto)
- operações da aplicação servidor
- operações da aplicação cliente, incluindo a invocação remota

No servidor, é mostrado o código para incorporar o serviço de nomes na mesma aplicação (em comentário) do servidor.
No nosso caso, usamos o `rmiregistry`, uma aplicação do Java que tem o papel de binder ou serviço de nomes para o RMI.

No cliente, o endereço e o porto usados para a primeira ligação à rede **não** são relativos ao servidor, mas sim ao serviço de nomes onde o Objeto Remoto é registado para o serviço!

1.c- Atualize o código do servidor:
Crie uma nova instância do objeto remoto, para ser registado como prestador (*servant*) do serviço...
(*o local exato está assinalado no código fonte da classe do servidor*)

1.d- Atualize a seguinte no cliente:
Insira o código para invocar remotamente os métodos disponíveis neste serviço. Consulte a interface remota para saber que métodos poderá invocar sobre o objeto remoto.

2.a- Compile.
2.b- Veja as classes e interface remota na pasta de destino (no caso: 4 ficheiros).

2.c- (Este exercício serve para ver os subs estátivos... que Não usaremos!)
Este comando **seria** usado para gerar os stubs **estáticos** de cliente e servidor (Proxy, Skeleton+Dispatcher).

```
rmic -vcompat -classpath build/classes:. -d build/classes sd.rmi.PalavrasImpl
```

no windows, troque : por ;
Surgiriam mais 2 ficheiros ".class". São gerados pelo `rmic`, um compilador do JDK que gera os artefactos RMI.

```
build/classes/
└── sd
    └── rmi
        ├── Palavras.class
        ├── PalavrasClient.class
        ├── PalavrasImpl.class
        ├── PalavrasImpl_Skel.class   ###
        ├── PalavrasImpl_Stub.class   ###
        └── PalavrasServer.class
```

O procedimento atual de Java RMI desaconselha o uso de stubs estáticos... e este passo é desnecessário/desaconselhado.

3.a- Execute a aplicação **servidor**:
No netbeans, execute, num terminal, na pasta base do projeto:

```
java -classpath  build/classes sd.rmi.PalavrasServer 9000
```

Se viu

```
java.rmi.ConnectException: Connection refused
```

é porque o servidor tentou ligar-se a outra aplicação... que não estava à escuta no porto indicado (o serviço de nomes).

3.b- Execute as 3 aplicações, pela ordem:

- registry
  - `rmiregistry -J-classpath -Jbuild/classes 9000`
- servidor
- cliente

comprove a simplicidade do código para uma operação (**sofisticada**) como a invocação remota de métodos...

Notas

- se já tem o serviço de nomes ativo, a execução de servidor e cliente pode fazer-se dentro do IDE, ou fora, se preferir a linha de comandos
- caso tenha integrado o binder (serviço de nomes) na própria aplicação servidor, pode executar as aplicações dentro ou fora do IDE.

---

exercício 2
Utilizando a tecnologia RMI, implemente um serviço "reservasParaJantar", que ofereça as operações:

- adicionar uma reserva em determinado nome
- obter a lista de nomes com reserva
- obter o nº de reservas existentes

fase 1
Crie a interface remota para o serviço (3 operações).

fase 2
Implemente a classe do Objeto Remoto (onde está a funcionalidade do serviço... os métodos remotos). Assuma que os dados não precisam de repositório persistente... use apenas uma lista.

fase 3
Implemente aplicações cliente e servidor. O cliente não precisa de menu interativo.

fase 4
Execute um `Registry` para serviço de nomes, seguido do servidor.
Nota: Cuidado com o porto usado no Registry, para não haver conflito entre os programas de diferentes alunos, especialmente se executarem as aplicações num mesmo servidor (como `alunos.di.uevora.pt`).

fase 5
Execute o cliente e teste todas as opções.

Contactar suporte do site

Nome de utilizador: Rodrigo Alves (Sair)
Resumo da retenção de dados
Obter a Aplicação móvel

Fornecido por Moodle