Java RMI

Santi Caballé Llobet

PID_00185400

Universitat Oberta de Catalunya

Índex

Introducció

En aquest mòdul estudiarem el mecanisme d'invocació remota RMI de Java. Tal com hem vist en el mòdul "Introducció a les plataformes distribuïdes", Java posa RMI al centre del seu model d'objectes distribuït i representa la capa subjacent de comunicació distribuïda de la tecnologia de components.

Vegeu també

Veurem la tecnologia de components Java en el mòdul "Java EE".

Veurem com, tot i la simplicitat i limitacions d'aquest mecanisme, entendre RMI és fonamental per a disposar d'un bon coneixement de la comunicació distribuïda interna de Java. Aquest coneixement ens permetrà entendre el funcionament de tecnologies distribuïdes més complexes.

Al llarg del mòdul presentem, primer, les característiques principals d'RMI per al desenvolupament d'aplicacions distribuïdes en Java. Entre aquestes característiques explicarem en detall el protocol de comunicació distribuïda de Java basat en el mecanisme de la serialització de dades.

Seguidament, presentarem l'arquitectura RMI i es descriurà cada un dels seus elements, el que ens donarà una visió del seu funcionament intern. Veurem com a partir de la simplicitat i la transparència d'RMI, i també a la fidelitat d'aquest mecanisme d'invocació remota a la programació orientada a objectes, no es perceben grans diferències en passar de programar en Java aplicacions d'escriptori o locals a aplicacions distribuïdes.

Finalment, proposarem un cas d'estudi d'aplicació distribuïda desenvolupat amb RMI. Tot i la simplicitat del cas, ens permetrà afermar en la pràctica els conceptes vistos fins al moment d'aquesta tecnologia.

Objectius

Aquest mòdul us permetrà consolidar els objectius següents:

1. Entendre què és el mètode d'invocació remota (RMI) de Java.
2. Entendre els avantatges i les limitacions d'RMI comparat amb altres paradigmes de programació distribuïda.
3. Identificar les característiques internes d'RMI.
4. Conèixer l'arquitectura i el funcionament intern d'RMI.
5. Conèixer alguna aplicació pràctica d'RMI.

1. Introducció a RMI

En aquest apartat introduirem el mecanisme d'invocació remota de Java, anomenat ***RMI***[1]. Veurem les seves característiques intrínseques i objectius essencials que converteixen aquesta tecnologia en una de les més simples d'usar per a construir aplicacions distribuïdes.

(1)De l'anglès *remote method invocation*.

Una mica d'història

RMI forma part de Java des de la versió 1.1 del *Java development kit* (JDK), alliberat el 1997, i fou desenvolupat en l'empresa Sun Microsystems. RMI va ser revisat i s'hi van introduir importants millores a partir del JDK 1.5, alliberat el 2004, que perduren fins avui. Al gener de 2010, l'empresa Oracle va adquirir Sun Microsystems, i a partir d'aquest moment va adquirir tots els drets del llenguatge Java, inclòs RMI.

Entrarem a fons en les característiques més importants d'RMI, els quals aquesta tecnologia aconsegueix fer transparent al desenvolupador d'aplicacions distribuïdes de la complexitat interna de la xarxa, tant en la codificació de les dades per transmetre com en la gestió de les connexions remotes. Aquesta transparència representa la pedra angular del suport RMI a les aplicacions distribuïdes.

A partir del coneixement adquirit en aquest apartat, estarem en disposició de prendre decisions durant el desenvolupament de les nostres aplicacions distribuïdes i, d'aquesta manera, potenciar-ne la capacitat distribuïda.

1.1. Què és RMI

Nota

Una JVM (*Java Virtual Machine*, en anglès) és una màquina virtual executable en diferents plataformes, capaç d'interpretar i executar instruccions de codi binari especial (el *bytecode* Java), el qual és generat pel compilador del llenguatge Java.

El mecanisme d'invocació remota de Java, RMI, permet invocar mètodes d'objectes que es troben en JVM diferents de l'objecte que invoca (**invocació remota**) seguint el mateix procediment que una invocació de mètodes d'objectes que es troben en la mateixa màquina i JVM (**invocació local**). Una invocació remota implica comunicar processos separats situats en una mateixa màquina o en màquines separades situades en diferents punts geogràficament distants.

RMI representa el model d'objectes distribuïts que proposa Java com a solució per a desenvolupar aplicacions distribuïdes.

CORBA

Java també dóna suport al model d'objectes distribuïts de CORBA, mitjançant RMI-IIOP. Repasseu aquests conceptes en el mòdul "Introducció a les plataformes distribuïdes" d'aquest material didàctic. Tanmateix, en aquest mòdul no parem atenció al model CORBA des de Java, ja que té un ús marginal.

En el context d'una invocació remota, l'objecte que invoca es denomina ***client***, mentre que l'objecte remot, els mètodes del qual s'estan invocant, es denomina ***servidor***. En el cas general, l'objecte local que efectua la invocació interpreta el rol de client invocant els mètodes d'un objecte remot que interpreta el rol de servidor. Tanmateix, de vegades l'objecte local i remot poden intercanviar els seus rols dinàmicament (per exemple, durant les notificacions i avisos al client, és el servidor qui inicia la interacció per informar el client).

Nota

Observem que la resposta del servidor a la invocació d'un client no comporta cap canvi de rol, en estar la resposta del servidor inclosa en el procés mateix d'invocació del client.

Durant una invocació local (dins d'una mateixa JVM), un objecte local fa referència directament a un altre objecte local per invocar els seus mètodes. En contraposició, en una invocació remota, l'objecte client mai no fa referència directament a l'objecte servidor, sinó que fa referència a una interfície remota,

la qual estableix una clara separació entre la definició i la implementació de l'objecte servidor. D'aquesta manera, s'estableix una separació entre les signatures dels mètodes i el codi d'aquests mètodes.

Vegeu també

Explicarem el funcionament intern d'RMI detalladament en l'apartat "Arquitectura RMI" d'aquest mòdul.

Això darrer permet que la definició i la implementació estiguin en JVM separades. Aquesta separació encaixa amb les necessitats d'un sistema distribuït típic, en què el client només està interessat en la definició del servei (**interfície**) que sol·licita al servidor, el qual és responsable de tenir implementat aquest servei per a proporcionar-lo al client.

Per la part negativa, RMI ha estat criticat per utilitzar un protocol de comunicació restrictiu conegut com a *Java remote method protocol* (JRMP). Aquest protocol, en estar implementat en Java, obliga que client i servidor hagin d'estar implementats en Java. Aquesta restricció resta flexibilitat al model RMI envers altres tecnologies que admeten diferents tecnologies d'implementació.

Vegeu també

Repasseu el protocol JRMP en el mòdul "Introducció a les plataformes distribuïdes". En aquest mòdul donem per assumit que quan parlem de la comunicació RMI es reconeix el protocol de comunicació JRMP.

La figura 1 exemplifica gràficament el comportament general d'un servei distribuït en RMI. En una calculadora distribuïda, el client sol·licita fer una operació de suma de dos operands sencers, mentre que el servidor fa aquesta operació i retorna el resultat al client. El client únicament veu la definició del servei i la utilitza per a sol·licitar l'operació de suma. Aquesta definició del servei es correspon amb la signatura del mètode `public int sumar(int a, int b);`. El servidor es fa càrrec de la implementació d'aquest servei proporcionant el codi del mètode de suma.

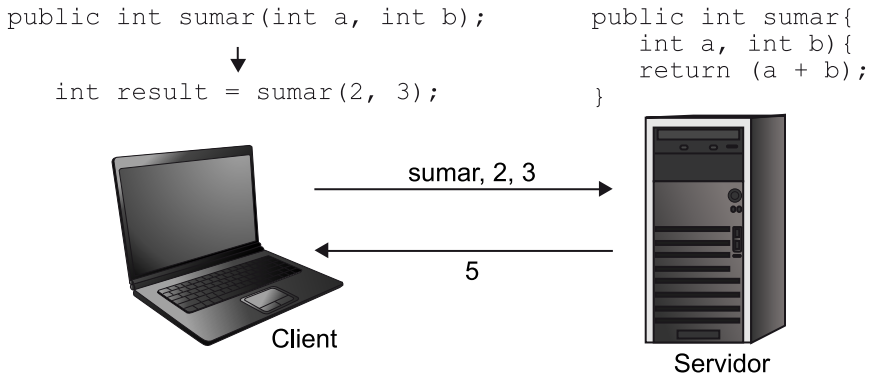

Figura 1

1.2. Objectius d'RMI

Quan es va desenvolupar RMI, es va plantejar com una solució per a entorns exclusius Java amb la finalitat última de simplificar el desenvolupament d'aplicacions distribuïdes orientades a objectes i escrites en Java. Es volia que RMI proporcionés els mecanismes necessaris per a fer transparent la complexitat de la comunicació en xarxa, fent que el programador es pogués concentrar només en la lògica particular de l'aplicació que desenvolupa.

Per a assolir aquests reptes, els creadors d'RMI es van fixar els objectius següents:

- **Transparència**. Permetre la invocació de mètodes d'objectes remots que s'executen en altres JVM com a processos separats, amb independència que aquests objectes estiguin en la mateixa màquina o en màquines diferents.

- **Orientat a objectes**. Integrar el model d'objectes distribuïts al llenguatge Java de manera natural i preservant, sempre que sigui possible, la semàntica d'objectes de Java.

- **Simple**. Aplicar a la programació distribuïda els mateixos criteris de simplicitat de la programació Java estàndard.

- **Seguretat**. Ús d'un model de seguretat basat en els gestors de seguretat i carregadors de classes locals.

- **Portabilitat**. Fer que qualsevol sistema distribuït RMI sigui portable a qualsevol plataforma on hi hagi una JVM.

- **Escalabilitat**. Mantenir la mateixa arquitectura amb independència de la grandària de l'aplicació distribuïda.

- **Robustesa**. Proporcionar una jerarquia de gestió d'excepcions remotes per a detectar i gestionar els errors generats en el servidor.

- **Eficiència**. Permetre l'activació, desactivació i reactivació, i també l'alliberament automàtic d'objectes remots no referenciats, millorant la gestió de recursos del servidor.

En els apartats següents veurem en detall les característiques que ofereix RMI a la programació distribuïda, amb els quals aconsegueix assolir aquests objectius.

1.3. Característiques RMI

RMI és una solució client-servidor basada en el conegut protocol RPC, al qual afegeix el paradigma de la programació orientada a objectes i permet la comunicació entre objectes remots que es troben en JVM separades. RMI està construït sobre la base que forma el següent:

Vegeu també

Reviseu el protocol *remote procedure call* (RPC) a l'assignatura *Xarxes i aplicacions Internet*.

- El model orientat a objectes de Java.
- El suport Java a la programació per *sockets* en xarxes TCP/IP.
- El mecanisme de serialització d'objectes de Java.

El primer i segon punts, el model orientat a objectes de Java i la programació per *sockets*, han quedat suficientment coberts en assignatures anteriors. El tercer punt clau, el mecanisme de la serialització, es tractarà en detall a continuació.

Vegeu també

Reviseu l'orientació a objectes de Java en l'assignatura *Disseny i programació orientada a objectes* i la programació per *sockets* en l'assignatura *Xarxes i aplicacions Internet*.

També analitzarem altres característiques importants d'RMI com són el pas de paràmetres, el procés d'activació d'objectes, el recol·lector d'escombraries distribuït, el procés de localització d'objectes remots, l'ús d'excepcions en RMI i el suport a la seguretat.

Les característiques RMI les ofereix Java de manera transparent com a part de la implementació del seu model d'objectes distribuïts.

Gràcies a aquesta tranparència, el programador no s'haurà d'implicar en aquestes tasques, tret que tingui requisits d'eficiència crítics, entre altres raons. En aquests casos, haurà d'escollir l'opció "manual" que Java sempre posa a la disposició del programador i implementar aquestes característiques amb el seu propi codi. Tanmateix, per a la majoria de casos, i els que veurem en aquesta assignatura, s'utilitzarà la implementació RMI per defecte que proporciona Java.

Escalabilitat de l'aplicació

L'escalabilitat de l'aplicació pot ser una altra raó per la qual no es puguin utilitzar les opcions RMI per defecte.

1.3.1. Serialització

La **serialització** d'objectes és l'acció de codificar un objecte en una seqüència de dades.

Més concretament, el mecanisme de serialització que proporciona RMI consisteix a convertir classes d'objectes en una seqüència de bytes. Si invertim aquest procés, obtindrem una còpia de l'objecte original a partir d'aquests mateixos bytes de dades, i farem el que es coneix com a ***desserialització***.

Serialització i desserialització

El procés de serialització també es coneix com a *marshalling*, mentre que la desserialització es coneix com a *unmarshalling*.

Aquests processos es fan sense pràcticament intervenció del programador, cosa que proporciona un elevat grau de transparència, gràcies a l'abstracció dels *streams* que proporciona Java, que representen un flux o seqüència de bytes. Els *streams* permeten a dos programes intercanviar-se dades en forma de seqüència de bytes. Aquests programes es poden estar executant en dues màquines o sistemes de comunicació diferents, i estar connectats en xarxa mitjançant una altra abstracció molt important, els *sockets*.

Nota

Els *streams* de Java s'expliquen en detall a continuació.

En RMI, les dades (objectes) que passem per paràmetre durant la invocació d'un mètode remot són serialitzades i convertides en un *stream* de bytes pel procés que invoca. Aquest procés escriu en un *socket* que retransmet l'*stream* per la xarxa fins a arribar al *socket* de destinació.

El procés invers és simètric. El procés remot llegeix del *socket* i desserialitza l'*stream*. Les dades desserialitzades seran una còpia de la dada original, que serà utilitzada com a paràmetre en el mètode remot.

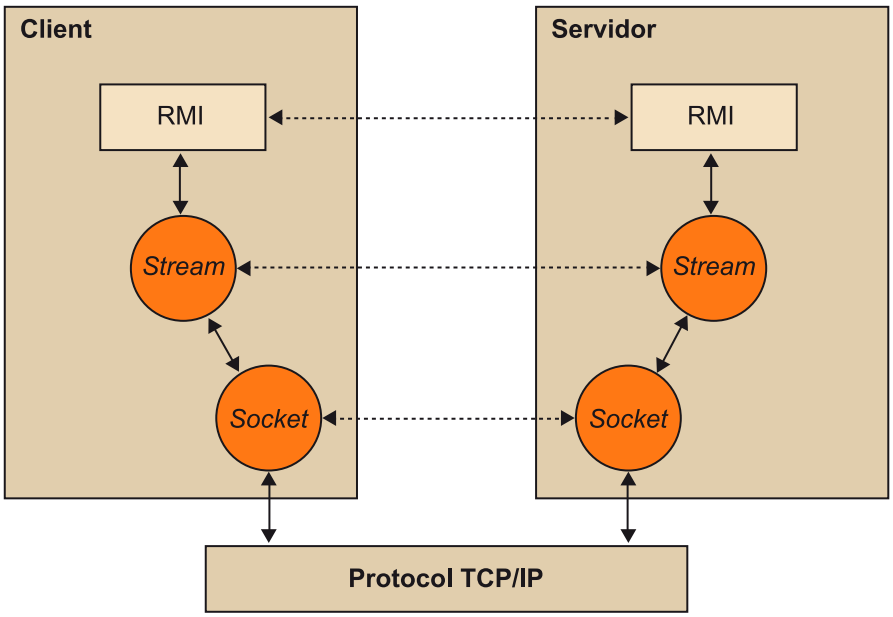

Figura 2

En la figura 2 veiem la relació entre RMI, *sockets* i *streams* durant una connexió típica de xarxa client/servidor. RMI abstreu la funcionalitat dels *sockets*, que a la seva vegada fan servir *streams* per a enviar seqüències de dades entre programes durant una comunicació en xarxa a baix nivell. Les fletxes puntejades bidireccionals representen la comunicació lògica en tots dos sentits. La capa de transport i xarxa TCP/IP representa la comunicació activa entre client i servidor.

Encara que ni els *sockets* ni els *streams* formen part directament d'RMI, per a comprendre el funcionament de la comunicació en xarxa de Java i el mecanisme de serialització d'RMI primer cal comprendre els *sockets* i els *streams*. A continuació descrivim els *streams*.

Vegeu també

La comprensió dels *sockets* i de la comunicació en xarxa de Java queda fora de l'abast d'aquesta assignatura. Reviseu aquests conceptes a l'assignatura *Xarxes i aplicacions Internet*.

Streams

Els ***streams*** són una estructura de dades que permet emmagatzemar i recuperar informació de manera inevitablement seqüencial.

Nota

Per comoditat, en aquest mòdul farem servir la nomenclatura anglesa de *streams* en comptes de flux o seqüència de bytes.

Les dades (típicament bytes) solament es poden afegir a l'*stream* d'un en un, i recuperar-los de la mateixa manera. Cal, doncs, començar a recórrer la seqüència sempre des del inici fins al final. No és possible recórrer la seqüència d'una altra manera (anar enrere, saltar llocs o començar al mig). Una vegada recuperada una dada de l'*stream* cal moure's a la posició següent de la seqüència i una vegada escrita una dada no es pot esborrar.

Encara que pot semblar que aquesta manera de tractar la informació és pesada i poc útil, els *streams* tenen una característica molt important: la **simplicitat**. L'accés seqüencial a la informació fa que els *streams* siguin compatibles amb qualsevol dispositiu, sigui físic (per exemple, una impressora) o abstracte (per exemple, un fitxer). Aquesta simplicitat permet uniformar l'accés al dispositiu per mitjà d'un procés trivial: primer, s'associa un *stream* a un dispositiu, i després, la informació és llegida o escrita seqüencialment en aquest dispositiu per mitjà de l'*stream*.

Les seqüències són una primitiva per a representar informació, i tota forma més complexa de representació es pot veure com una aplicació recursiva d'aquestes. Per exemple, per a manejar blocs de 10 dades, primer es recuperen 10 dades consecutives d'una seqüència i després es crea un bloc amb aquestes dades. D'aquesta manera s'aconsegueix treballar amb estructures més complexes a partir d'un *stream* primitiu.

Vegeu també

Repasseu les seqüències com a tipus de dades en l'assignatura *Disseny d'estructures de dades*.

Java ofereix dues classes principals per a treballar amb *streams*: `OutputStream` i `InputStream`.

`OutputStream` és una classe abstracta que representa un flux de bytes de sortida. Un `output stream` accepta bytes de sortida, que escriu en un dispositiu genèric de sortida. El mètode més important d'aquesta classe és *`write(byte)`*, que va escrivint en el dispositiu un byte cada vegada que és cridat. Les subclasses d'`OutputStream` implementen la sortida de bytes a dispositius específics (per exemple, `FileOutputStream` i `PipeOutputStream` involucren fitxers i *pipes* respectivament).

Dispositiu de sortida

Un dispositiu de sortida es considera apropiat mentre sigui capaç de rebre o emmagatzemar les dades que li arriben d'un `OutputStream` (per exemple, fitxers, *pipes*, servidors, etc.).

De manera simètrica, `InputStream` és una classe abstracta que representa un flux de dades d'entrada. Un *input stream* accepta bytes d'entrada, que llegeix d'un dispositiu genèric d'entrada. El mètode més important d'aquesta classe és `byte`, que recupera un byte del dispositiu cada vegada que és cridat. Les subclasses d'`InputStream` implementen l'entrada de bytes des de dispositius específics.

Dispositiu d'entrada

En aquest cas, un dispositiu d'entrada es considera apropiat si pot treballar com a font de dades i coincideix amb els dispositius de sortida que hem vist.

`OutputStream` i `InputStream` ofereixen altres mètodes de lectura i escriptura que permeten treballar amb blocs de bytes en comptes de tractar els bytes individualment. *`byte[] read()`* i *`write(byte[])`* són dos mètodes que ofereixen aquesta característica. No obstant això, internament els bytes són sempre tractats un per un de manera seqüencial.

Combinar *streams*

Consulteu en el paquet `java.io` de l'API de Java altres maneres de combinar els *streams* amb blocs de dades, juntament amb la resta de mètodes disponibles en `InputStream` i `OutputStream`.

Un *stream* pot envoltar altres *streams* recursivament per a aconseguir una funcionalitat incremental.

A partir d'aquesta idea, hi ha dos tipus de *streams*:

1) *Streams* primitius. Són els que parlen amb els dispositius directament llegint o escrivint els bytes tal com arriben de manera seqüencial. Aquests *streams* són els que hem vist fins ara (per exemple, `FileOutputStream` i `FileInputStream`).

2) *Streams* intermedis. Aquests mai no parlen directament amb els dispositius. La seva funció és la d'envoltar un *stream* existent (pot ser un *stream* primitiu o un d'intermedi) per guanyar en funcionalitat. Parlem llavors de ***streams* contenidors** i ***streams* continguts**. El propòsit més habitual dels *streams* intermedis és per a tasques com l'emmagatzematge a la memòria intermèdia, *pipeling*, compressió i serialització de dades.

Exemple de *streams* intermedis

Un exemple de fluxos intermedis són les classes `BufferedOutputStream` i `BufferedInputStream`, molt usades per a proporcionar capacitat d'emmagatzematge temporal (*buffering*).

Normalment, l'*stream* contenidor té l'*stream* contingut com a paràmetre (l'envolta) i així li afegeix la funcionalitat del contenidor. Els *streams* intermedis poden manejar dades més complexes que simples bytes, però han ser convertits a bytes quan escriguin en el dispositiu. La lògica (operacions) de l'*stream* més extern s'anirà propagant per tots els *streams* fins a arribar al més intern (*stream* primitiu) que llegirà o escriurà en bytes en el dispositiu.

Durant la serialització, els *streams* intermedis més utilitzats són les classes `ObjectOutputStream` i `ObjectInputStream`, que converteixen objectes en bytes i al revés.

Els mètodes principals d'aquestes dues classes són `writeObject(Object)`, que representa el mecanisme de serialització en essència (tradueix objectes en seqüències de dades de tipus byte), i el mètode `Object readObject()`, que representa el mecanisme invers (desserialització).

Mecànica de la serialització

El procés de serialitzar un objecte s'aconsegueix en dos passos:

1) Crear un objecte `ObjectOutputStream`.

2) Cridar el mètode `writeObject(Object)` passant per paràmetre l'objecte per serialitzar; aquest convertirà l'objecte en bytes i l'escriurà en l'*stream* que s'hagi passat per paràmetre al constructor d'`ObjectOutputStream`.

Per a desserialitzar se segueix un procés simètric:

1) Crear un objecte `ObjectInputStream`.

2) Cridar el mètode `Object readObject()`, que construirà un objecte a partir dels bytes de l'*stream* passat al constructor d'`ObjectInputStream`.

Atès que `ObjectOutputStream` i `ObjectInputStream` proporcionen el comportament per defecte de tota serialització, RMI no les utilitza per raons d'eficiència. En el seu lloc crea subclasses d'aquestes en què redefineix alguns mètodes protegits de les superclasses.

Nota

Per a la comprensió d'aquest apartat n'hi ha prou de conèixer aquelles classes que fan possible el mecanisme de serialització per defecte.

Un cas especial són les aplicacions amb grans jerarquies d'herència, en què la serialització no s'hauria d'aplicar en les superclasses (especialment classes abstractes i interfícies), sinó solament en les subclasses amb les quals treballem realment. Altrament, haurem de controlar minuciosament i evitar la propagació de la serialització de la superclasse a aquelles subclasses o certs atributs d'aquestes que no vulguem serialitzar.

Per als casos especials s'acostuma a manejar els atributs de la superclasse en les subclasses mitjançant la redefinició dels mètodes `writeObject()` i `readObject()` que hem vist. No obstant això, per simplificar i evitar la serialització manual, en els casos més habituals és millor serialitzar les superclasses.

Usos de la serialització

En Java, l'algorisme de serialització que converteix objectes en *streams* de bytes és únic. El procés de conversió sempre és el mateix, però, en canvi, els usos potencials que es poden fer d'aquest procés són múltiples.

Per exemple, si assignem un `FileOutputStream` a un fitxer, les dades que escriguem en l'*stream* quedaran emmagatzemades de manera persistent en el fitxer. D'aquesta manera es pot crear un autèntic mecanisme de persistència.

També podem usar la serialització com un mecanisme de còpia d'objectes en memòria mitjançant l'ús de `ByteArrayOutputStream`, gràcies al qual les dades escrites en l'*stream* es desen temporalment en una matriu de tipus byte. D'aquesta manera, mitjançant matrius de bytes, podem fer còpies d'un objecte original en la memòria temporal de l'ordinador.

Per als propòsits d'RMI, el principal ús de la serialització és com a mecanisme de comunicació. Si escrivim dades en un *stream* assignat a un *socket*, aquestes dades seran transmeses automàticament per la xarxa fins al *socket* de destinació, en què un altre procés recuperarà aquestes dades mitjançant el corresponent *stream* de lectura i se'n farà càrrec.

Identificador universal de serialització

Nota

Imaginem els problemes que tindríem si client i servidor treballessin amb versions diferents de les mateixes classes i en una s'haguessin fet modificacions.

Les classes que serialitzem per a ser enviades per la xarxa, que són posteriorment carregades tant en el client com en el servidor, han de ser les mateixes versions.

> Java garanteix la unicitat de les classes mitjançant la generació en temps de compilació d'un identificador de versió anomenat ***identificador universal de versió*** o *serialVersionUID*, que genera un identificador diferent cada vegada que es modifica una classe.

Aquest identificador s'afegeix a la classe que volem serialitzar de manera que client i servidor puguin conèixer si treballen amb les mateixes versions de la classe. En cas que no sigui així, es llança una excepció del tipus `InvalidClassException` per a indicar que les classes serialitzables que utilitzen client i servidor no són les mateixes internament.

No obstant això, aquest identificador que genera Java per defecte és altament sensible al tipus de compilador que estem utilitzant i pot generar diferents identificadors encara que no s'hagin fet modificacions en la classe. Per a evitar aquest problema es recomana generar un identificador explícitament dins del codi de la classe. Això evitarà que Java generi el seu identificador propi per defecte i assegurem que és el mateix identificador amb independència del compilador que estiguem usant. El *serialVersionUID* es desa com un atribut més de la classe que volem serialitzar, amb aquests modificadors d'accés i nom:

```
private static final long serialVersionUID =
7526472295622776147L
```

Encara que l'identificador pot ser qualsevol nombre, per a evitar duplicats i problemes de format, el millor és generar-lo automàticament utilitzant l'eina anomenada *serialver*, que ve amb el JDK. Aquesta eina generarà un número únic de versió per a la classe que es basa en els propis elements que conté la classe.

serialver

El JDK incorpora una versió gràfica del *serialver*. Escrivim en consola `serialver -show`, i ens demanarà la localització de la classe que volem versionar. A continuació ens donarà un identificador únic per la classe.

Finalment, és important que mantinguem el modificador *private* per a aquest atribut. La raó és que l'identificador de versió sigui exclusiu de la classe que estem serialitzant i no es propagui a les seves subclasses.

Avantatges i problemes de la serialització

En una comunicació RMI les invocacions a mètodes remots amb arguments d'instàncies de classe, els paràmetres passats durant la invocació són còpies de les instàncies originals (pas de paràmetres per valor), en comptes de referències directes a aquestes (pas de paràmetres per referència).

En un context d'una invocació remota (en la mateixa màquina o en màquines diferents), invocador i invocat es troben sempre en diferents processos. Si el procés invocador enviés referències seves al procés invocat s'hauria de produir cada vegada un canvi de procés (travessant la xarxa si els processos estan separats en màquines diferents) per a efectuar una nova invocació. Tot això reduiria el rendiment de l'aplicació de manera inacceptable.

Vegeu també

En el subapartat "Pas de paràmetres i valors de retorn" tractarem detalladament el pas per paràmetres en el context d'una invocació remota.

Vegem a la figura 3 un exemple fictici de la situació que ens hem creat. En la nostra calculadora remota, si els operands sencers els passem com a referències a objectes de tipus `Integer` en el servidor, aquest en voler recuperar l'*int* que conté haurà d'invocar el mètode `intValue()` de l'objecte `Integer` que es troba en el client. Per tant, serà necessari travessar la xarxa de nou i crear una nova invocació, aquesta vegada en el client. Veiem que en total es faran 6 invocacions per a efectuar l'operació de suma.

Exemple de la calculadora

Noteu que en aquest exemple el més lògic seria treballar amb operands primitius de tipus *int*, que són serialitzables per defecte. Hem forçat l'exemple a operands de tipus *Integer* per mostrar la problemàtica.

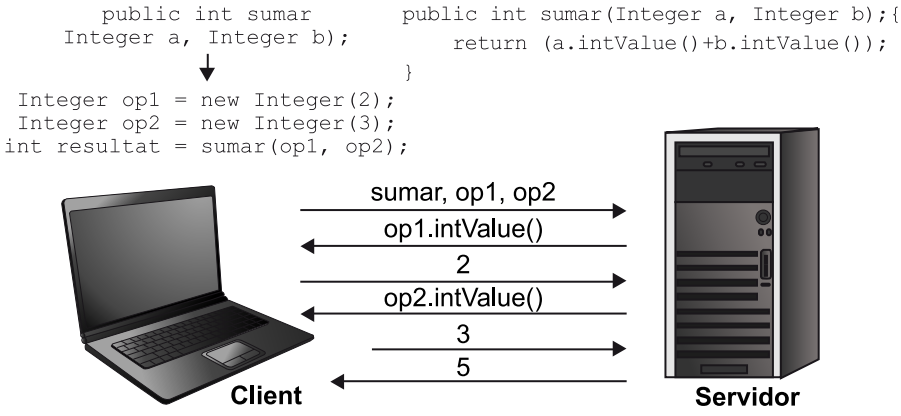

Figura 3

Les contínues invocacions remotes comportarien allargar el trànsit per la xarxa, la qual cosa significa augmentar el temps de resposta i incrementa la possibilitat de fallades a la xarxa. Tampoc no seria possible accedir als atributs públics de l'objecte (si n'hi hagués), ja que en les invocacions via RMI només s'accedeix a mètodes d'objecte, mai a atributs.

La figura 4 descriu la mateixa invocació de mètodes remots però amb arguments que són còpies d'objectes en comptes de referències a l'objecte, tal com realment succeeix en el context d'RMI. Els operands sencers els passem com a objectes de tipus `Integer` al servidor, aquest en voler recuperar l'*int* que conté, haurà d'invocar el mètode `intValue()` de l'objecte `Integer`, igual que abans, però ara es tracta d'una invocació local, dins del mateix procés del servidor. Els objectes dels arguments es troben en el servidor mateix i, per tant, no hi ha necessitat de sortir a la xarxa, estalviant el trànsit de xarxa.

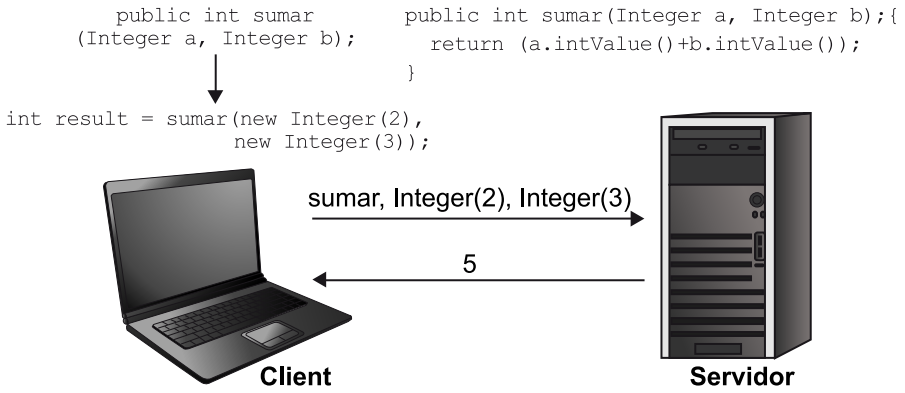

Figura 4

Vegem ara alguns problemes que també comporta la serialització.

Passar còpies en comptes de referències implica problemes potencials d'inconsistència amb l'objecte original. És a dir, els canvis que faci el procés remot sobre la còpia no es propagaran a l'objecte original, ja que en cas contrari representaria transmetre contínuament aquests canvis per mitjà de la xarxa.

D'altra banda, és necessari que les còpies siguin completes, amb totes les còpies dels objectes i tipus primitius afectats. Això significa que en passar un objecte, internament els atributs d'aquest objecte poden referenciar altres objectes o tipus primitius, cosa que obliga recursivament a incloure una còpia per a cadascun d'aquests objectes i tipus primitius interns.

Podem veure l'estat d'un objecte com un graf en forma d'arbre en què els nodes són els atributs de tipus objecte i les fulles són els atributs de tipus primitiu. Per tant, una còpia completa d'un objecte significa copiar tot el graf de l'objecte amb tots els seus objectes interns, que estaran representats en forma de sub-

Tipus objecte

Per **tipus objecte** ens referim a instàncies de la classe `Object` de Java o qualsevol de les seves subclasses, mentre que **tipus primitius** són els tipus primitius de Java (*int*, *char*, *long*, etc.).

graf. D'aquesta manera, en el servidor hi haurà fàcilment molts objectes duplicats, ja que cada crida a un mateix mètode remot que tingui un argument de tipus objecte significarà enviar cada vegada una nova còpia del mateix objecte.

No obstant això, aquests inconvenients són de poca magnitud en comparació dels perjudicis que implicaria el pas de paràmetres per referència, com hem vist.

Serialització d'objectes en RMI

Amb tot el que hem vist fins ara, podem serialitzar objectes per a la comunicació en xarxa amb RMI.

En RMI, tots els paràmetres d'una invocació remota i els valors retornats pels objectes remots han de ser serialitzables. Per tant, tota dada que hagi de viatjar per la xarxa haurà de ser convertida abans en un *stream* de bytes.

Hi ha poques excepcions a aquesta regla general, encara que en certs casos hi haurà dades que no tingui sentit serialitzar. Per a aquests casos especials, disposem del modificador *transient* per a evitar la serialització d'aquells atributs en què no és permès o que no volem serialitzar.

Casos especials

Un cas especial és `FileInputStream`; si la serialitzéssim, en desserialitzar-la més endavant en el servidor, podria referenciar algun fitxer no vàlid o sense sentit en el nou context. Un altres cas és un objecte de tipus `Thread`, que no té sentit serialitzar-lo, en depèndre el seu estat de la plataforma subjacent.

Tots els tipus primitius de Java són serialitzables per defecte, ja que es poden convertir a bytes directament (el tipus *int* és una seqüència de 4 bytes). És per això que els tipus primitius no necessiten cap transformació per a ser transmesos per la xarxa. Quant als tipus objecte, cal fer-los serialitzables explícitament seguint les indicacions següents:

`Serializable`

Trobareu més informació sobre `Serializable` en el paquet *java.io* de l'API de Java.

- La classe que volem serialitzar ha d'implementar (`implements`) la interfície `Serializable`. Aquesta interfície és, de fet, una interfície buida, això és, no conté cap mètode. La seva funció és, simplement, **marcar** les classes que volem fer serialitzables per diferenciar-les de la resta. Per tant, RMI solament serialitzarà els atributs d'aquelles classes que implementin aquesta interfície.

- Generar un *serialVersionUID* per a la classe serialitzable i afegir-lo al codi en forma d'atribut. Aquest atribut tindrà modificador d'accés *private* per a evitar la propagació a les seves subclasses.

- Els atributs de l'objecte per serialitzar han de ser, o bé de tipus primitiu, o de tipus objecte serialitzable (és a dir, que aquest objecte, al seu torn, sigui serialitzable). Si no és així aconseguirem una excepció de tipus `NotSerializableException`. Aquells atributs que no volem incloure en la serialització els modificarem amb *transient*. Cal notar que els atributs *static*

tampoc no se serialitzaran, ja que el seu estat té a veure amb la classe, i no amb la instància que volem serialitzar.

- Assegurar-se que la superclasse de l'objecte que volem serialitzar és una classe serialitzable. Si no és així, encara és possible la serialització fent persistent l'estat de la superclasse. Per a fer-ho cal redefinir els mètodes `writeObject()` i `readObject()` per a manejar l'estat de la superclasse explícitament des de la subclasse.

- Per a evitar el problema que hem identificat de la duplicació de còpies d'objectes passats per paràmetre en successives invocacions del mateix objecte, ens hem d'assegurar que treballem sempre amb la còpia correcta. Pot ser necessari comprovar si dos objectes són el mateix mitjançant la redefinició dels mètodes `equals()` i `hashcode()` de l'objecte abans de serialitzar-lo, perquè el servidor pugui efectuar aquestes comprovacions.

`equals()` i `hashcode()`

Els mètodes `equals()` i `hashcode()` formen part de la classe arrel `Object`.

Finalment, cal fer esment dels casos especials en què el rendiment de l'aplicació distribuïda és crític i l'ús de la serialització que ofereix RMI per defecte pot resultar lent. Això és degut principalment a l'ús que es fa de l'API *Reflection* de Java per a descobrir informació sobre l'objecte que s'està serialitzant.

API *Reflection*

Consulteu l'API *Reflection* en el paquet *java.lang.reflect* de l'API general de Java.

Per a aquests casos, Java proporciona la interfície `Externalizable`, la qual cal que implementin les classes en què volem un control total en les tasques de serialització i desserialització. Aquestes tasques es programaran manualment *ad hoc* per als objectes que interessa serialitzar. `Externalizable`, de manera semblant a `Serializable`, marca les classes que no volem fer serialitzables automàticament i delega a la classe aquesta responsabilitat. Amb la programació manual, el mecanisme de serialització coneixerà en temps d'execució tota la informació d'aquests objectes i evitarà l'ús de l'API *Reflection*.

`Externalizable`

`Externalizable` és una altra interfície del paquet *java.io* que hereta de `Serializable`.

No obstant aquest punt darrer, la serialització manual comporta un esforç important i cal disposar d'un alt nivell tècnic en programació. En la gran majoria de casos, utilitzarem la serialització genèrica d'RMI, ja que cobreix suficientment els casos habituals.

1.3.2. Pas de paràmetres i valors de retorn

Acabem de veure que la interfície `Serializable` és una manera de **marcar** una classe per a informar que es tracta d'un objecte local que serialitzem. L'objectiu és informar el compilador i l'entorn d'execució de Java que haurà de passar (pas de paràmetres) per valor còpies dels objectes d'aquesta classe des de la JVM local a la JVM remota en un entorn RMI.

Vegeu també

En l'assignatura *Fonaments de programació* heu vist el pas de paràmetre per valor i per referència.

En una invocació remota hi ha també la possibilitat de passar per paràmetre tipus d'objectes remots. Els tipus d'objecte remot s'identifiquen amb la interfície `Remote`.

`Remote`

Trobareu més informació sobre `Remote` en el paquet *java.rmi* de l'API de Java.

`Remote` és una interfície sense mètodes amb el propòsit de **marcar** una classe per a informar que es tracta d'un objecte remot que en una invocació remota de mètodes passarem per referència.

Per a entendre millor els tipus objectes remots[2], podem imaginar un sistema distribuït complex en què hi ha diversos entorns RMI funcionant al mateix temps. Durant les invocacions és possible passar com a paràmetre tipus d'objectes remots pertanyents a altres entorns RMI diferents d'aquell en què ens trobem. No és possible serialitzar un objecte d'aquest tipus i passar una còpia seva al nostre servidor remot, ja que estaríem enviant aquesta còpia a una ubicació diferent de l'original, cosa que no té sentit. L'única manera possible és passar una referència d'aquest objecte, que sempre es trobarà en la seva ubicació original, i en ser remot, significa que té capacitat de ser accedit remotament. En conseqüència, els paràmetres de tipus objecte remot els passarem per referència.

[2] Tingueu en compte que els tipus objectes remots són poc habituals en una invocació remota.

Durant una invocació a un mètode d'un objecte remot, els paràmetres que conté la invocació poden ser de tres tipus possibles:

1) Primitius (*int*, *float*, *char*, etc.), els quals passarem sempre per valor, ja que, com hem dit, són serialitzables per defecte (una dada de tipus *int* és ja una seqüència de 4 bytes).

2) Objectes serialitzables (o no remots) són objectes la ubicació dels quals no és important per a mantenir-ne l'estat intern. Es poden trobar tant en la JVM local com en la JVM remota sense que això n'afecti l'estat. Durant la invocació, passarem per valor aquests objectes, és a dir, en passarem una còpia. Juntament amb les dades de tipus primitiu, són el cas més habitual i freqüent de tipus de dades en el pas de paràmetres d'una invocació remota.

3) Objectes remots són objectes la ubicació dels quals sí que és important per al seu estat intern i, en conseqüència, no en podem canviar la localització enviant-ne una còpia a una altra JVM. Aquests objectes representen els objectes remots mateixos, als mètodes dels quals fem referència en una invocació remota. A diferència dels tipus primitius i objectes no remots, els objectes remots són un cas poc habitual de tipus de dades en el pas de paràmetres.

A la taula següent, a manera de resum, identifiquem els diferents tipus de pas per paràmetre i situacions que hi ha en Java, tant per a invocacions locals com remotes. Observem com els tipus primitius sempre són passats per valor, mentre que els tipus d'objecte remot són passats per referència.

Invocació	Tipus primitiu	Tipus d'objecte no remot	Tipus d'objecte remot
Local	PV	PR	PR
Remota	PV	PV	PR

PV = per valor, PR = per referència.

Han sortit crítiques al diferent tractament que es dóna al pas de paràmetres en posar en dubte el caràcter de transparència d'accés en RMI. El programador ha de ser conscient en tot moment de si els paràmetres passats en una invocació remota són de tipus objecte remot o no remot. En el primer cas, haurà de ser serialitzat per a poder enviar una còpia de l'objecte no remot per la xarxa. En el segon cas s'enviarà solament la referència de l'objecte remot.

Transparència d'accés

La **transparència d'accés** es pot definir com la capacitat d'un sistema d'oferir sempre les mateixes interfícies en fer les crides als seus serveis, ja que en cas de no oferir aquesta transparència seria necessari adaptar la crida cada vegada que canvia la interfície.

Finalment, quan l'objecte servidor està retornant un valor com a resultat d'una invocació remota de mètodes, aquesta dada sempre es copiarà de la JVM remota a la JVM local, ja que es tractarà, o bé d'un objecte serialitzable, o bé d'un tipus primitiu. No és possible retornar tipus d'objectes remots.

1.3.3. Localització

En un sistema distribuït amb gran quantitat d'objectes emmagatzemats en un gran nombre de màquines resulta imprescindible disposar d'un sistema de localització. Aquest sistema cal que sigui simple i que permeti fàcilment als clients trobar i connectar amb la màquina servidora en què es troba l'objecte els mètodes del qual volen invocar.

Hi ha diferents solucions a aquest problema, però poques de simples i efectives. Tenim l'opció que l'aplicació client usi directament l'adreça física del servidor. No obstant, això trencaria la qualitat de transparència de localització, ja que implicaria actualitzar i recompilar constantment l'aplicació a cada canvi de localització del servidor. Una altra possibilitat és que l'usuari de l'aplicació client conegui i proporcioni l'adreça del servidor com a entrada de dades, però això comporta un treball incòmode per part de l'usuari.

Transparència de localització

La **transparència de localització** (també coneguda com a transparència d'ubicació) es pot definir com la capacitat d'un sistema d'oferir un servei sense preocupar-se d'on és ubicat físicament aquest servei.

Una solució millor per a localitzar un objecte en un entorn distribuït és crear un servei de noms, com un directori telefònic, en què un nom (adreça lògica) es correspon a un servidor (adreça física). Aquest servei es trobaria en un servidor conegut que, en cas de canvi de localització dels servidors, actualitzaria constantment la part d'informació referent a l'adreça física on es troben els objectes remots. D'aquesta manera, l'adreça lògica es correspondrà amb

DNS

El conegut sistema de noms de domini (*Domain Name System*, en anglès) és un exemple de servei de noms.

l'autèntic servidor, i serà transparent al client quant als canvis de localització. Serà també suficientment flexible per a conduir la crida al servidor requerit durant la invocació (en temps d'execució), sense cap intervenció de l'usuari.

RMI proporciona un servei de noms molt simple anomenat *RMIRegistry* que permet al servidor enregistrar els objectes remots perquè els clients els puguin trobar.

Aquest servei és vàlid per a aplicacions poc complexes amb pocs objectes remots, però no és adient per a aplicacions distribuïdes grans i dinàmiques, per la manca d'escalabilitat, i també d'independència de la màquina on es troba.

Vegeu també

Veurem l'*RMIRegistry* en més profunditat en el subapartat "*RMIRegistry*" d'aquest mòdul.

1.3.4. Activació d'objectes remots

En un model d'objectes local, els objectes instanciats es troben en la memòria interna de la màquina llestos per a ser usats. Quan l'aplicació finalitza, o bé es produeix una apagada en la màquina, aquests objectes es destrueixen. En aquest context, es tracta d'un comportament adequat, ja que en finalitzar l'aplicació (pel motiu que sigui), l'existència d'aquests objectes no té sentit.

No obstant això, aquest comportament no és apropiat quan passem al model d'objectes distribuïts, per diferents raons:

- El servidor que conté l'objecte remot instanciat pot ser apagat, avariar-se, es poden esgotar els seus recursos, etc., amb la qual cosa els objectes que es trobaven en memòria es destrueixen i desapareixen. En una aplicació distribuïda, aquestes situacions no haurien de significar la fi de l'aplicació, la qual s'ha de continuar executant per a donar servei als seus usuaris.

- L'objecte remot és solament necessari mentre s'està executant l'aplicació. No obstant això, una aplicació distribuïda pot passar llargs períodes de temps sense activitat (per exemple, durant la nit o els festius), però els objectes remots romandran en la memòria del servidor tot el temps. Això comporta un malbaratament important de recursos en el servidor.

Calen, doncs, mecanismes addicionals que evitin les situacions descrites. D'una banda, és necessari que els objectes remots consumeixin recursos del servidor just el temps necessari que puguin ser invocats. D'altra banda, en el cas que l'objecte es destrueixi involuntàriament, aquest hauria de ser recuperat completament, i mantenir-ne l'estat previ.

En el model d'objectes distribuïts, hi ha dues operacions, anomenades ***activació*** i ***desactivació***. L'*activació* instancia objectes remots que han estat prèviament desactivats o destruïts, mentre que la *desactivació* fa el procés invers.

Un objecte remot inactiu no es troba en la memòria del servidor i, per tant, no consumeix recursos. L'estat que tenia l'objecte quan estava actiu es desa en suport persistent, juntament amb dades de localització del codi de la classe dins del servidor, entre altres dades. Totes aquestes dades s'actualitzen constantment mentre l'objecte es troba actiu, i es mantenen persistents en el servidor. Podem veure que els objectes remots es troben en la memòria temporal, però al mateix temps també es manté una còpia persistent de l'objecte.

Objectes locals

Aquesta diferència és clara pel que fa als objectes locals, els quals es troben en memòria temporal solament.

El client, per la seva banda, manté una referència a la informació persistent sobre els objectes del servidor. Això li permet activar l'objecte remot en cas que aquest no estigui instanciat (inactiu). Aquesta informació permet al client buscar, activar (és a dir, instanciar de nou) i inicialitzar un objecte remot amb les dades del seu últim estat conegut quan aquest es troba inactiu. D'aquesta manera, es recupera l'objecte remot, amb el mateix estat que tenia i torna a estar actiu i llest per a donar servei als clients.

Objecte inactiu

Es diu que si l'objecte passa a inactiu o s'ha destruït involuntàriament, no mor, sinó que passa a *stand by*.

Tots aquests mecanismes són, o haurien de ser, totalment transparents a usuaris i programadors. No obstant això, hi ha desavantatges que habitualment trenquen aquesta transparència. Vegem-ne algunes raons.

Quan un objecte passa cert temps sense ser invocat per un client, aquest és desactivat. Per a això, cal preveure estratègies i polítiques d'activació (llindar d'inactivitat, quins objectes són més crítics i han d'estar actius més temps que altres, etc.). Aquestes estratègies han de ser decidides i dissenyades *a priori* i, en alguns casos, els usuaris n'han de ser conscients. D'altra banda, un objecte remot inactiu s'ha d'activar abans de ser invocat pels clients. Això comporta més tasques i un sobrecost en comparació de la invocació d'objectes local, que es tradueix en una latència més gran del procés d'invocació.

Java, en el seu model d'objectes distribuïts RMI, disposa d'un mecanisme d'activació d'objectes transparent als usuaris i programadors.

Aquest mecanisme es basa en dos elements:

1) un **activador d'objectes** que es troba en cada servidor. Està format tant per un *gestor* de JVM que es troba en cada JVM del servidor com una taula de *descriptors d'activació* amb informació persistent dels objectes que s'executen (objectes actius) en el servidor. Aquesta taula es fa persistent en el servidor junt amb informació sobre la JVM del servidor on s'executa l'objecte, el nom

de la classe de l'objecte, l'URL on es troba el codi *bytecode* de l'objecte i dades d'inicialització de l'objecte (per exemple, el nom d'un fitxer amb l'estat de l'objecte).

2) un **apuntador intel·ligent** que es troba en el client apuntant a la *taula de descriptors* del servidor. Aquest apuntador conté tant un *identificador d'activació* de l'objecte remot (que apunta a l'activat del servidor on es troba l'objecte) com una *referència viva* a l'objecte remot, a partir de la qual és possible conèixer si l'objecte està actiu o inactiu.

Quan s'invoca un objecte remot, la *referència viva* del client indica en quin estat es troba l'objecte. Si aquest està *inactiu*, s'usa l'*identificador d'activació* per a trobar l'*activador* de l'objecte. Aquest activador ens situa en el servidor on es troba l'objecte. La *taula de descriptors* de l'activador proporciona tota la informació necessària per a activar l'objecte. En cas que la JVM corresponent a aquest objecte no estigui en execució, el *gestor* d'aquesta JVM l'arrencarà. En tots els casos, l'*activador* carrega el *bytecode* de l'URL, instància l'objecte i l'inicialitza amb les dades d'inicialització d'aquest objecte que estan desades. Finalment, una vegada activat l'objecte remot, es retorna la referència remota a l'*activador*, el qual la retorna al client per a actualitzar la *referència viva* (ara, l'objecte es troba *actiu*).

1.3.5. Recollidor d'escombraries distribuït

Tots els llenguatges orientats a objectes disposen de mecanismes manuals o automàtics per a eliminar de la memòria aquells objectes que deixen de ser necessaris. Aquest procés d'eliminació és fonamental per a una gestió correcta i eficient dels recursos de la màquina, en alliberar recursos que es poden destinar a altres objectes.

El criteri per a decidir quins objectes no són necessaris es basa en comprovar si es troben referenciats per algun client. Quan no hi ha cap referència a l'objecte, es diu que aquest ja no és necessari i s'elimina.

Java, com altres llenguatges orientats a objectes, disposa d'un mecanisme automàtic per a recollir aquests objectes eliminats anomenat ***recol·lector d'escombraries***[3], que és transparent al programador. Altres llenguatges no disposen d'aquest sistema automàtic i deleguen aquest procés al programador, que ha de determinar quan i com es du a terme. Això comporta molts problemes per la dificultat de gestionar manualment els objectes i l'impacte que té la no-eliminació d'objectes no necessaris en l'eficiència global.

[3]El recol·lector d'escombraries es coneix generalment com a *garbage collector*.

Llenguatge C++

En llenguatges com C++, en programar un client s'ha de tenir en compte si hi ha altres clients en el sistema amb referències al mateix objecte remot.

En els sistemes d'eliminació automàtics, tots els objectes són visitats constantment, i aquells que no són referenciats per cap altre objecte s'eliminen. Aquest mecanisme s'aplica tant en entorns locals com distribuïts. Vegem el funcionament en cada cas:

- En **entorns locals**, hi ha en tot moment un coneixement complet de les referències existents als objectes. Les referències a objectes locals són implementades com a adreces de memòria que apunten a la zona on s'emmagatzemen els objectes. Quan un objecte en memòria no té cap apuntador a ell, la part de la memòria que ocupa aquest objecte s'allibera.

- En **entorns distribuïts**, les referències als objectes són més complexes. Una referència ha d'aportar informació com ara la localització on es troba l'objecte en el sistema, dades sobre el tipus d'objecte i informació de seguretat. Tota aquesta informació s'emmagatzema en forma de referència de llargària considerable, tant en el programari intermediari del sistema distribuït com en la part servidora. En aquest cas, és encara més important eliminar aquells objectes del sistema distribuït no referenciats, ja que també es fa necessari eliminar-ne la referència, que ocupa molt espai de memòria.

Programari intermediari

El programari o capa intermediària (*middleware*) es troba generalment entre el sistema operatiu i les aplicacions del sistema i permet resoldre de manera transparent les tasques de baix nivell de les aplicacions. També s'aplica als sistemes distribuïts com una abstracció de tots els sistemes individuals que en formen part.

El **recol·lector d'escombraries** que utilitza Java per al seu model d'objectes distribuïts RMI es basa en un comptador de referències remotes.

Vegem a continuació el funcionament d'aquest mecanisme en RMI.

En una invocació remota, en què s'invoquen els mètodes d'un objecte remot, totes les invocacions de diferents clients a aquests mètodes es refereixen al mateix objecte remot, en què prèviament els clients s'han registrat. Per a controlar les referències dels clients a aquest objecte, RMI segueix la pista a totes les referències dins de cada JVM. Cada vegada que es produeix una nova referència a un objecte remot des d'alguna JVM, el comptador de referències per a aquell objecte s'incrementa. Així, el comptador reflecteix la suma de les referències de totes les JVM clients que accedeixen al mateix objecte remot.

Quan succeeix la primera referència a l'objecte remot, s'avisa al servidor de l'objecte perquè el marqui amb l'etiqueta *referenciat*. D'aquesta manera el recol·lector d'escombraries n'és conscient i no descarta aquest objecte. El recol·lector d'escombraries va inspeccionant contínuament cada JVM client, i cada vegada que troba una referència descartada (ja no es referencia l'objecte remot des d'aquella JVM), es decrementa el comptador. Quan el comptador arriba a zero, s'avisa el servidor perquè marqui l'objecte com a *no referenciat*, la qual cosa condueix el recol·lector d'escombraries a descartar l'objecte remot.

No obstant això, hi ha problemes[4] amb aquest mecanisme. En cas que la xarxa entre clients i servidor estigui separada, la capa de transport pot *no veure* els clients i pot creure que han caigut. Això pot provocar que el recol·lector d'escombraries descarti l'objecte remot de manera prematura, en creure que els clients no hi tenen referències. Com a conseqüència, aquest mecanisme no pot garantir la integritat referencial als objectes remots, és a dir, una referència client pot no referenciar cap objecte remot, i també al revés, un objecte remot pot ser esborrat havent-hi encara referències a aquest.

(4)Hi ha solucions a aquests problemes tot i que són complexos i queden fora de l'abast d'aquest mòdul.

1.3.6. Excepcions

Durant una comunicació en xarxa poden ocórrer gran quantitat d'incidències (caiguda d'un servidor, problemes de xarxa, fallada de connexió del client, etc.). Per exemple, si el servidor cau durant la invocació d'un mètode per part d'un client, RMI llançarà automàticament una `RemoteException` en el costat del client.

Davant de qualsevol incidència sorgida, RMI llança una excepció de tipus `RemoteException` per a avisar el client que alguna cosa va malament en l'entorn RMI.

Per a assegurar la màxima robustesa en l'entorn RMI, es va introduir en el compilador l'obligació d'afegir la `RemoteException` en la clàusula `throws` com a part de la signatura de cadascun dels mètodes del servidor que puguin ser invocats pel client. Aquesta decisió comporta l'obligació per part dels clients d'invocar qualsevol mètode remot dins d'un bloc `try-catch` (*checked exception*). Una conseqüència negativa d'aquesta decisió és que obliga el programador a generar codi en el costat del client per a tractar excepcions imprevisibles, la qual cosa resulta codi sense gaire utilitat.

RemoteException

Trobareu més informació sobre `RemoteException` en el paquet *java.rmi* de l'API de Java.

Exception

Totes les excepcions que hereten d'`Exception` s'anomenen *checked exception* i s'han de tractar obligatòriament. Java també permet l'ús d'*unchecked exceptions* (hereten de `RuntimeException`) per a evitar aquesta obligació.

A més, l'obligació és doble, ja que solament es permet llançar objectes exclusivament de tipus `RemoteException` (no són permeses les subclasses). Això significa que el tractament d'altres excepcions que no siguin de comunicació (és a dir, les pròpies de l'aplicació) cal portar-lo a terme dins de la `RemoteException` mateixa, com una excepció niada, aprofitant que el constructor `RemoteException(String s, Throwable ex)` permet niar altres excepcions.

Vegeu també

Trobareu un exemple pràctic d'excepció remota niada en el cas d'estudi de l'apartat "Cas d'estudi RMI" d'aquest mòdul.

El procés de llançament d'una excepció en RMI comença quan el servidor detecta alguna incidència durant la invocació. La implementació d'algun dels mètodes llança una excepció del tipus `RemoteException`. L'*skeleton* del servidor serialitza l'objecte `RemoteException` llançat i el retransmet per la xarxa fins a l'*stub* del client, que desserialitza aquest objecte per a ser atrapat per un

Vegeu també

Explicarem el funcionament dels *stubs* i *skeletons* detalladament en l'apartat "Arquitectura RMI" d'aquest mòdul.

bloc `catch`. Per tant, RMI aprofita que mitjançant els *stubs* i *skeletons* controla la comunicació entre el client i el servidor per a propagar automàticament una excepció quan es detecta alguna incidència.

Finalment, és important considerar els mètodes locals que es troben en l'objecte servidor i que no formen part de la interfície remota que dóna servei als clients. Aquests mètodes solament podran ser invocats dins de la mateixa JVM del servidor per mitjà d'una típica invocació local. Per tant, encara que formin part de l'objecte remot, els mètodes locals no necessiten llançar una `RemoteException`.

1.3.7. Seguretat

El model de seguretat de Java per a *applets* (miniaplicacions, en català) i aplicacions es basa en l'ús de carregadors de classes, verificadors de codi *bytecode* i administradors de seguretat. No obstant això, aquest nivell de seguretat és insuficient per a aplicacions distribuïdes amb RMI.

Bibliografia complementària

No és el propòsit de l'assignatura explicar el model de seguretat de Java. Aquells que hi estigueu interessats podeu consultar l'obra: **J. Jaworski; P. J. Perrone** (2001). *Seguridad en Java*. Madrid: Pearson Alhambra.

Els **carregadors de classe** solament permeten carregar classes que es trobin en la màquina local. Per a carregar classes des d'ubicacions remotes és imprescindible un administrador de seguretat específic que permeti carregar les classes remotament.

L'administrador de seguretat que s'usa en Java per a les aplicacions distribuïdes és la classe `RMISecurityManager`.

Per a disposar d'un **administrador de seguretat** per a RMI es crea una instància de `RMISecurityManager` mitjançant `System.setSecurityManager(new RMISecurityManager())` al principi de l'execució de l'objecte client o servidor. Podem crear subclasses de `RMISecurityManager` i així obtenir administradors de seguretat a mida (i menys restrictius) segons les nostres necessitats.

`RMISecurityManager`

Trobareu més informació sobre `RMISecurityManager` en el paquet *java.rmi* de l'API de Java.

D'altra banda, hem vist que RMI es basa en la comunicació per *sockets*, per mitjà dels quals s'envien objectes serialitzats per la xarxa, en principi sense cap tipus de xifratge. Tanmateix, les dades crítiques dels objectes serialitzats i, en general, tota comunicació que transiti per la xarxa, hauria de viatjar sempre xifrada. Per a això, Java ofereix `SSLSocket`, una subclasse de `Socket` que permet usar el protocol SSL per a xifrar la informació que s'intercanvien entre *sockets*. Java també ofereix la possibilitat de crear *sockets* específics per a RMI amb `RMISocketFactory` (que són diferents de la classe `Socket`). La creació dels nostres propis *sockets* és imprescindible quan necessitem xifrar o comprimir les dades que s'envien al servidor, o bé la nostra aplicació RMI requereix diferents tipus de *sockets* per a diferents objectes remots.

Vegeu també

Reviseu el protocol *secure socket layer* (SSL) i la seguretat per *sockets* en l'assignatura *Administració de xarxes i sistemes operatius*.

`SSLSocket` i `RMISocketFactory`

Consulteu `SSLSocket` en el paquet *javax.net.ssl* i `RMISocketFactory` en el paquet *java.rmi* de l'API de Java.

A partir de la versió 1.2 de Java es va introduir un entorn de seguretat més restrictiu per a impedir dur a terme accions malicioses. Es va fer necessari crear entorns de seguretat a mida mitjançant permisos personalitzats que són atorgats a aquelles parts del codi que han de dur a terme accions crítiques. A causa d'aquestes restriccions de seguretat, és necessari donar permisos explícits al servidor durant una comunicació RMI.

Per a aquest propòsit, Java proporciona una sèrie de tipus de permisos. En una aplicació típica RMI, el permís més important és el de la classe `SocketPermission` que gestiona permisos d'un *socket* en la xarxa. Un `SocketPermission` es construeix amb un nom de servidor i una sèrie d'accions permeses per a aquest servidor. Això s'aconsegueix creant un fitxer de pòlisses que especifica en forma de permisos quines són les polítiques de seguretat que s'aplicaran al *socket*.

Vegem-ne un exemple en què es crea un fitxer de pòlisses anomenat *java.policy* en el qual especificarem els permisos donats als *sockets* que participen:

```
grant{ permission java.net.SocketPermission "*:1024-65535",
 "connect,accept";

permission java.net.SocketPermission "192.192.192.192:80",
"connect"; };
```

En aquest exemple donem permisos per a acceptar connexions. Primer es dóna permís d'acceptar connexions i connectar-se a qualsevol servidor (símbol "*") entre els ports 1024 i 65535 (és a dir, tots els ports que no són del sistema). En el segon cas es dóna permís de connexió al servidor amb adreça 192.192.192.192 per mitjà del port 80 (servei HTTP). Crearem un fitxer amb el nom *java.policy* amb aquest contingut. Per tal que l'entorn d'execució el trobi, el desarem en el servidor com a valor en la propietat de seguretat del sistema mitjançant la instrucció de l'intèrpret java `-Djava.security.policy=java.policy NomObjecteServidor`.

java.policy

Utilitzem *java.policy* com a nom del fitxer de pòlisses per conveni, tot i que és lliure i es pot canviar per un nom més adequat, només modificant les propietats del fitxer *java.security* en el directori *<JAVA HOME>/lib/security*.

2. Arquitectura RMI

Objecte remot i objecte servidor

Utilitzem els termes *objecte remot* i *objecte servidor* com a sinònims, tot i que generalment s'entén que els objectes servidors administren els objectes remots.

Com hen vist, el model d'objectes distribuït que proposa Java permet que els objectes que s'executen en una JVM invoquin els mètodes d'objectes que s'executen en altres JVM. L'objecte que fa la invocació es denomina ***objecte client***, mentre que l'objecte remot que rep la invocació dels seus mètodes es denomina ***objecte servidor***. L'objecte remot ha d'estar registrat en un servidor de noms.

Vegeu també

Parlarem àmpliament dels *stubs* i *skeletons* en el subapartat "*Stubs* i *skeletons*" d'aquest mòdul.

En aquest model, l'objecte client mai no fa referència directament a un objecte servidor sinó que fa referència a una interfície remota que és implementada per l'objecte servidor. Aquesta interfície fa de servidor intermediari, també anomenat fragment adaptador ***stub***, el qual implementa els mètodes remots. Aquest es comunica per la xarxa mitjançant ***sockets*** amb el seu homòleg del servidor, anomenat ***skeleton***, responsable de la comunicació amb els objectes remots del servidor. Tant els *stubs* com els *skeletons* són generats automàticament com a classes durant la compilació i són els autèntics responsables de la comunicació per la xarxa.

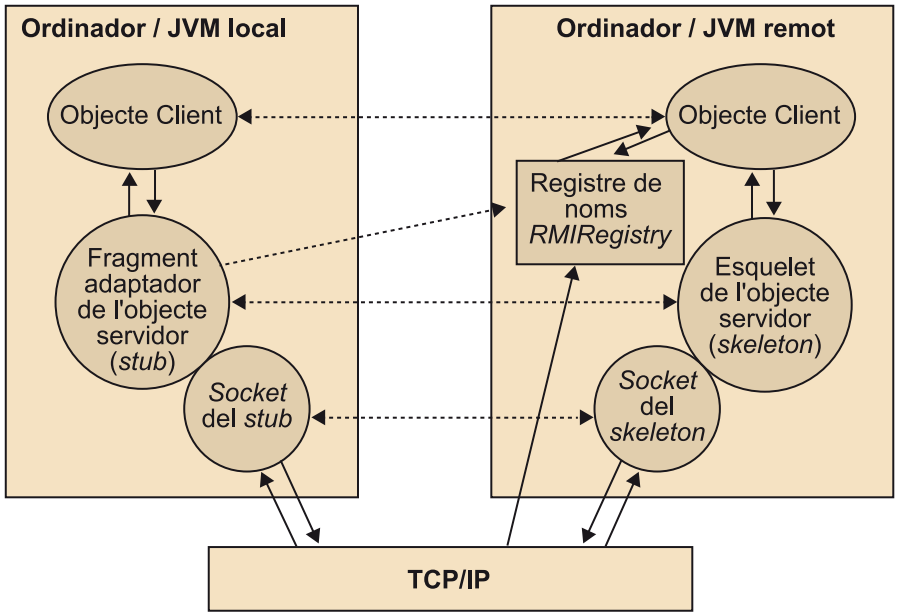

Figura 5

En la figura 5 es pot observar una vista general de l'arquitectura bàsica d'una comunicació client i servidor en l'entorn RMI per a dur a terme una invocació d'un mètode remot amb crida i retorn (fletxes bidireccionals). Les línies puntejades representen la comunicació virtual o lògica, mentre que les línies contínues representen la comunicació real (es mostra la comunicació del nivell de transport). Veiem com tant els *stubs* com els *skeletons* es comuniquen per mitjà de *sockets*, que genera l'*stub* durant la crida al mètode remot. *RMIRe-*

gistry i l'objecte servidor es comuniquen durant el registre de l'objecte, i també durant la consulta de l'*stub* per a trobar l'objecte. El protocol de xarxa TCP/IP representa el nivell de transport de la xarxa entre la JVM local i la remota.

En els apartats següents, descriurem cada un d'aquests components que formen l'arquitectura RMI en detall.

Vegeu també

L'arquitectura mostrada en la figura 5 ens servirà de guia per a la resta del mòdul.

2.1. El servidor

La **part servidora** d'una aplicació RMI conté fonamentalment els objectes remots que els clients busquen per invocar els seus mètodes i tota la infraestructura necessària per a rebre les invocacions i retornar els valors de tornada dels mètodes executats.

Com s'ha dit anteriorment, els clients no referencien directament els objectes remots sinó una interfície remota que és implementada per l'objecte servidor. En el cas més simple, en el servidor hi ha d'haver una classe *interface* que estengui la interfície `Remote` amb la signatura de tots els mètodes remots (és a dir, aquells que llancin `RemoteException` en la seva clàusula `throws`). Aquests mètodes són els que poden ser invocats pel client.

Més informació

Trobareu més informació de les classes que s'esmenten en aquest apartat en el paquet *java.rmi* de l'API de Java.

La interfície `Remote` s'utilitza de manera semblant a `Serializable`, de manera que els objectes que implementen aquesta interfície queden identificats com a objectes remots i l'entorn els proporciona comportament remot. Aquesta interfície ha de ser implementada per una classe servidora, i juntes formen l'objecte remot del servidor en temps d'execució.

Vegeu també

En el subapartat "Serialització" (punt "Serialització d'objectes RMI") d'aquest mòdul hem explicat la interfície `Serializable`.

Aquesta classe servidora es basa principalment en el paquet `java.rmi.server` i té, en el cas més bàsic, les tasques principals següents:

- **Implementar la interfície remota**, que implica implementar tots i cadascun dels seus mètodes remots. En cas que necessitem incloure mètodes locals per a ser utilitzats pel servidor mateix, s'afegiran aquests mètodes sense la clàusula `RemoteException`.

- **Estendre** `UnicastRemoteObject` per tal que el servidor contingui objectes d'aquesta classe accessibles per mitjà de connexions TCP. `UnicastRemoteObject` hereta de `RemoteServer` que, al seu torn, hereta de `RemoteObject`. Aquesta última és la que proporciona una implementació remota a la classe `Object` per mitjà de redefinir apropiadament els mètodes `equals()`, `toString()` i `hashCode()`. Amb això s'aconsegueix proporcionar un comportament remot predeterminat als objectes remots del servidor, que estenen `UnicastRemoteObject`.

`UnicastRemoteObject`

`UnicastRemoteObject` és una classe que s'encarrega de la major part de les tasques associades amb la creació d'un servidor RMI (com crear un *socket* en el servidor i escoltar invocacions de mètode del client).

- **Crear i registrar l'objecte remot**. El registre de l'objecte remot es farà en el servei de noms del sistema amb un nom identificador per a aquest objecte que el lligui (*bind*) a l'objecte remot en qüestió. S'usarà el mètode `Naming.rebind("rmi://dir_servidor:port/NomIdObjecte" obj)`, en què `dir_servidor` és la IP o el nom DNS del servidor, `port` serà un número de port que no usi el sistema, `NomIdObjecte` és l'identificador triat per a trobar l'objecte remot, i `obj` és l'objecte remot registrat. Per a la majoria d'aplicacions s'utilitzarà *RMIRegistry*, que és el servei de noms que ofereix el JDK per mitjà de l'eina *rmiregistry*.

  Vegeu també

  Detallarem el funcionament d'*RMIRegistry* en el subapartat "*RMIRegistry*" d'aquest mòdul.

  Crear i registrar l'objecte remot

  En el cas més simple s'enregistra l'objecte servidor mateix, que ja conté l'objecte remot.

  Si client i servidor es troben a la mateixa màquina, la IP és 127.0.0.1. i el nom DNS és *localhost*. Els ports que no usa el sistema (rang usuari) són: 1024-65535. *rmiregistry* escolta per defecte pel port 1099.

- **Arrencar el servidor** mitjançant la instanciació de l'objecte servidor mateix dins del seu mètode `main()`. En arrencar el servidor, es porten a terme totes les accions que acabem de comentar.

En la part servidora, és imprescindible disposar d'un esquelet de l'objecte servidor (anomenat *skeleton*) per a poder rebre i tractar adequadament les invocacions dels clients. El paquet `java.rmi.server` també inclou la interfície `Skeleton`, que implementen els *skeletons* durant la seva generació. Es tracta d'un element de més baix nivell que l'objecte remot i que, juntament amb el seu homòleg *stub* en el client, formen la connexió entre client i servidor per on circula tota la informació intercanviada.

Vegeu també

Ampliarem la informació sobre *stubs* i *skeletons* en el subapartat "*Stubs* i *skeletons*" d'aquest mòdul.

2.2. El client

El **client** és el que invoca els mètodes de l'objecte servidor. En aquest cas, el client ha de dur a terme les tasques següents, per a invocar un mètode remot:

- **Localitzar l'objecte remot**. De manera simètrica al procés seguit per a enregistrar l'objecte en el servidor, per a localitzar l'objecte enregistrat en el servei de noms s'utilitza el mètode `Naming.lookup("rmi://dir_servidor:port/NomIdObjecte")` en què, `dir_servidor` i `port` tenen el mateix significat que hem vist abans en el servidor. `NomIdObjecte` ha de ser el mateix nom usat pel servidor durant el registre per a lligar-lo (*bind*) amb l'objecte remot. El mètode retorna un objecte de tipus `Object`, al qual és necessari fer un *cast* per a convertir-lo al tipus de l'objecte remot.

  Més informació

  Trobareu més informació de les classes que s'esmenten en aquest apartat en el paquet *java.rmi* de l'API de Java.

  Vegeu també

  En l'assignatura *Disseny i programació orientada a l'objecte* heu vist el mecanisme de *cast* de Java.

- Les **invocacions als mètodes remots** s'han de dur a terme des de dins d'un bloc `try/catch` per capturar com a mínim la `RemoteException` llançada des del servidor. En cas de problemes en el procés de localització de l'objecte remot, diverses excepcions del tipus `java.net.MalformedURLException` i `java.rmi.NotBoundException` poden ser llançades de manera automàtica per la JVM, que seran capturades en aquest mateix bloc.

Vegeu també

En el subapartat "Excepcions" d'aquest mòdul hem parlat de les excepcions de Java en les invocacions remotes.

De manera simètrica a l'esquelet del servidor, al costat del client és imprescindible disposar d'un fragment adaptador de l'objecte servidor (*stub*) per a poder tractar i enviar adequadament les invocacions dels clients i rebre els valors de tornada del servidor. És una representació local d'un objecte remot que implementa tots els mètodes invocables remotament d'aquest objecte remot.

El paquet `java.rmi.server` inclou la classe `RemoteStub`, que estén `RemoteObject` i proporciona una implementació abstracta dels fragments adaptadors dels objectes servidors del costat del client. A continuació s'explicquen en detall tant els *stubs* com els *skeletons*.

2.3. *Stubs* i *skeletons*

El mecanisme d'invocació remota RMI es basa en dos tipus d'objectes generats pel compilador d'RMI: els *stubs* i els *skeletons*.

> Un ***stub*** o **fragment adaptador** es troba en el costat del client i és una representació local d'un objecte remot del servidor que referencia tots els mètodes remots d'aquest objecte. De manera equivalent, un ***skeleton*** representa un esquelet de l'objecte remot i es troba al costat del servidor.

És a dir, un *stub* té implementats els mateixos mètodes que l'objecte remot, però el codi d'aquesta implementació consisteix només en una referència completa amb tot el necessari per a comunicar-se amb els mètodes de l'objecte remot (el codi real d'implementació dels mètodes es troba en l'objecte remot).

La funció principal d'un *stub* és crear i mantenir una connexió de *sockets* oberta amb l'*skeleton* del servidor i responsabilitzar-se de la tasca de serialitzar i desserialitzar les dades en el costat del client. De manera semblant, l'*skeleton*, es responsabilitza del manteniment de la connexió de *sockets* amb l'*stub*, i també de serialitzar i desserialitzar les dades al costat del servidor.

stub

Es diu també que l'*stub* representa un objecte del servidor dins de la JVM del client que implementa els mètodes de l'objecte servidor.

Tant l'*stub* com l'*skeleton* representen classes que es generen automàticament durant la compilació (habitualment, amb el compilador d'RMI *rmic*) a partir de les classes que implementen la interfície remota. Durant la invocació, es treballa amb instàncies d'*stubs* i *skeletons* mantenint així el model d'objectes que proposa Java. Tanmateix, en les darreres versions de Java els *stubs* i *skeletons* ja no són necessaris, com s'explica en el subapartat següent.

rmic

rmic és una aplicació que ve de sèrie amb el JDK, i que permet generar *stubs* i *skeletons*. Una vegada generats és necessari col·locar l'*stub* al costat del client i l'*skeleton* al costat del servidor (o en el mateix directori si fa funcions de client i servidor en aplicacions simples).

2.3.1. Evolució dels *stubs* i *skeletons* en Java

Tot i el que hem explicat fins ara, la generació dels *skeletons* només va ser necessària abans d'arribar a la versió 1.2 de Java. A partir d'aquesta versió es va suprimir la necessitat de generar els *skeletons* substituint-los per la capacitat de *reflexió* que ofereix l'API *Reflection* de Java. Aquest mecanisme permet conèixer i invocar els mètodes de l'objecte remot en temps d'execució. Així i tot, els *skeletons* són necessaris per a la compatibilitat cap enrere amb aplicacions escrites en versions antigues de Java.

API *Reflection*

Consulteu l'API *Reflection* en el paquet *java.lang.reflect* de l'API general de Java.

A partir de la versió 1.5 de Java, també es va suprimir la necessitat de generar els *stubs*. No obstant això, en aquest cas, no significa que no existeixin ni que s'hagin substituït per un altre mecanisme, sinó que l'entorn d'execució és capaç de generar-los automàticament mitjançant tècniques de compilació dinàmica. En definitiva, en les darreres versions de Java ja no és necessari utilitzar *rmic* per a obtenir els *stubs* ni els *skeletons*.

Tanmateix, tant els *stubs* com els *skeletons* sí que són necessaris per a mantenir la compatibilitat cap enrere, a més que usar-los és conceptualment més didàctic. Per aquestes raons, nosaltres els utilitzarem en aquest mòdul, i també *rmic* per a generar-los.

2.3.2. Procés d'invocació client/servidor

Els *stubs* i *skeletons* són la clau del model que ofereix RMI, ja que estalvien al programador tot el treball que fan ells de manera automàtica (crear i mantenir els *sockets*, serialitzar i desserialitzar la informació transmesa, controlar la connexió, etc.). En fer totes aquestes funcions transparents al programador, s'aconsegueixen molts dels objectius que hem esmentat d'RMI.

Coneguem[5] aquest procés automàtic que segueix una invocació típica client-servidor, a baix nivell. Una vegada el servei *RMIRegistry* està actiu, els objectes remots estan correctament registrats, i el client ha trobat l'objecte remot, el procés bàsic per a invocar un mètode d'aquest objecte és el següent:

[5]Per tal de seguir millor aquest procés, observeu la figura 5.

1) El client obté una instància de l'*stub*, el qual, com hem dit, representa tots els mètodes remots.

2) El client crida un mètode de l'*stub* en una invocació del mateix tipus que es faria en una invocació local. És a dir, l'*stub* és una representació de l'objecte remot per al client, que es troba en la seva mateixa JVM, la qual cosa permet accedir a aquests mètodes de manera local.

3) L'*stub* al costat del client crea internament un *socket* amb l'*skeleton* al costat del servidor, la qual cosa permet tenir-los tots dos connectats. Com sabem, el *socket* enviarà les dades per mitjà de la xarxa en forma de *stream*.

Vegeu també

Reviseu el que hem parlat sobre *streams* i la serialització en el subapartat "Serialització" d'aquest mòdul.

4) L'*stub* serialitza tota la informació associada amb la invocació (nom del mètode invocat, tipus de paràmetres, etc.) i envia aquesta informació per mitjà del *socket* a l'*skeleton*.

5) L'*skeleton* desserialitza les dades que li arriben i fa la crida al mètode real del servidor. De manera anàloga a la relació entre l'*stub* i el client, la invocació entre l'*skeleton* i el servidor és també local, ja que es troben en la mateixa JVM.

6) En cas que el mètode invocat retorni un valor, aquest seguirà el camí contrari de la crida al mètode: l'*skeleton* rebrà aquest valor, el serialitzarà i l'enviarà per mitjà del *socket* a l'*stub*, que al seu torn el desserialitzarà i el retornarà a l'objecte client que ha fet la invocació.

2.4. *RMIRegistry*

Com hem introduït en apartats anteriors, el registre dels objectes es fa en RMI per mitjà del servei ***RMIRegistry***. Hi pot haver altres serveis de registre que proporcioni el nostre sistema. No obstant això, en venir de sèrie amb el JDK, és el servei més àmpliament utilitzat.

`rmiregistry`

La instrucció `rmiregistry` executa l'eina *rmiregistry* del JDK, que proporciona el servei de noms *RMIRegistry*.

El paquet `java.rmi.registry` conté totes les interfícies i classes que s'usen per a registrar i accedir a objectes remots mitjançant un nom. Quan s'executa l'eina *rmiregistry* s'obre la possibilitat de registrar els objectes remots identificant-los amb un nom. Aquest paquet conté la interfície `Registry`, que defineix mètodes `bind()`, `rebind()`, `list()` i `lookup()`, entre altres que utilitza la classe `Naming` per a associar aquests noms d'objecte i l'adreça URL on es troben i també per a localitzar-los.

`Registry` i `Naming`

La interfície `Registry` i la classe `Naming` les podeu trobar en el paquet *java.rmi* de l'API de Java.

RMIRegistry funciona com un directori d'objectes: un nom lògic o identificador de l'objecte es correspon amb l'*stub* de l'objecte. Durant el registre, el servidor ha d'enviar l'*stub* de l'objecte a *RMIRegistry* per a registrar-lo creant un lligam (`bind()`) entre un nom identificador de l'objecte i l'*stub* de l'objecte. D'aquesta manera, les aplicacions clients únicament han de conèixer la localització d'*RMIRegistry* i el nom identificador de l'objecte per a trobar-lo (`lookup()`) en temps d'execució. *RMIRegistry* proporciona una llista (`list()`) amb tots els noms identificadors que hi ha en el registre.

Per motius de seguretat, *RMIRegistry* se situa a la mateixa màquina del servidor juntament amb els objectes remots. En cas de trobar-se en màquines diferents, seria molt fàcil reemplaçar els servidors reals per altres maliciosos de manera que *RMIRegistry* apuntés a aquests últims. Això representa que *RMIRegistry* ha de funcionar com un servidor virtual dins del servidor mateix.

Com que tant *RMIRegistry* com els objectes remots han de ser a la mateixa màquina, l'*stub* coneix la màquina on es troba l'objecte (que és la mateixa màquina on es troba el servidor). D'aquesta manera, si traslladem la part servidora a una altra màquina, l'*stub* només haurà de cridar a l'URL d'*RMIRegistry* d'aquella mateixa màquina servidora, de la qual ja coneix l'adreça. A més, com que l'objecte remot ha quedat lligat a l'*stub* durant el registre, encara que l'objecte canviï de localització dins de la màquina servidora, serà transparent per a l'*stub*, que el trobarà igualment.

URL

L'URL que usaran els clients per a cridar *RMIRegistry* tindrà normalment aquest format: `rmi://adreca_servidor:port/identificador`

Per altra banda, si el servei *RMIRegistry* canvia de màquina (a una altra de diferent del servidor) o tan sols escolta en un altre port de la mateixa màquina, cal actualitzar l'URL, la qual cosa significa recompilar els clients. Per tant, RMI pot arribar a ser transparent pel que fa a la localització d'objectes dins de la mateixa màquina, però no a la ubicació de la màquina, la qual cosa és molt més important i és especialment greu en aplicacions distribuïdes altament dinàmiques.

Tot i els problemes que comporta, *RMIRegistry* és suficient per a aplicacions sense grans pretensions, com les que treballarem en aquesta assignatura, i resulta molt pràctic utilitzar-lo, en formar part dels recursos que proporciona Java de sèrie i així donar suport complet al seu model d'objectes distribuït.

És possible reemplaçar *RMIRegistry* per un altre servei de noms. En el mercat es poden trobar altres serveis de noms més flexibles i potents que *RMIRegistry*. També podem desenvolupar el nostre propi servei de noms ajustat a les nostres necessitats, encara que, com sempre, això significarà més esforç durant la programació.

3. Cas d'estudi RMI

En aquest apartat anem a posar en pràctica els conceptes apresos fins ara. Anem a construir una aplicació distribuïda, encara que molt bàsica, amb RMI. Primer veurem tota la mecànica per a seguir per a un desenvolupament i execució pas per pas manual. Seguidament, aprendrem com podem automatitzar els passos per a poder executar l'aplicació repetidament en un entorn de proves. Finalment, donarem les pautes per a traslladar l'execució a un entorn distribuït client-servidor entre ordinadors diferents.

Amb aquest cas d'estudi comprovarem els efectes de la transparència que ofereix RMI, que aconsegueix que no hi hagi diferències significatives entre invocar un mètode d'un objecte remot que es troba en una altra JVM en el mateix ordinador, i una invocació d'un mètode d'un objecte remot en un ordinador separat.

D'altra banda, també podrem comprovar l'assoliment d'altres objectius d'RMI, com són la seva simplicitat i escalabilitat, que permeten mantenir la mateixa arquitectura amb independència de la grandària de l'aplicació distribuïda. Veurem com les tasques per portar a terme amb RMI són les mateixes en una aplicació complexa que en el nostre cas d'estudi bàsic.

3.1. Un exemple d'RMI bàsic: una calculadora remota

Per a aquest cas d'estudi, implementarem una calculadora primitiva amb la funcionalitat elemental que ens permetrà sumar, restar, multiplicar i dividir dos operands sencers. La seva única particularitat serà que la podrem usar de manera remota en un entorn client-servidor. L'ordinador que contindrà l'objecte que farà els càlculs (la lògica de les operacions) farà funcions de servidor, mentre que l'ordinador amb l'objecte que representa el teclat i el visor de la calculadora (escriu els operands i operador i mostra el resultat) farà funcions de client.

Per a dur a terme un càlcul es crearan dos processos: procés servidor i client. El procés client enviarà al procés servidor els operands i l'operador. El procés servidor farà el càlcul i retornarà el resultat al procés client, que el mostrarà a l'usuari. Així doncs, el client no disposarà de la lògica de càlcul de la calculadora, però podrà igualment efectuar-hi càlculs.

Els dos processos seran independents l'un de l'altre i estaran en màquines diferents en què hi hagi instal·lada una JVM i estiguin connectades mitjançant una xarxa TCP/IP (habitualment, Internet). Solament és necessari que el client conegui l'adreça IP (o nom DNS) i el port de l'ordinador servidor.

Processos client i servidor

Els dos processos també poden córrer com dos processos independents dins de la mateixa màquina. En aquest cas, l'adreça IP (local) i el port tant del servidor com del client són els mateixos.

En els subapartats següents explicarem el desenvolupament complet del servidor i el client en una aplicació típica RMI. Procedirem primer a desenvolupar el servidor i després el client, per a disposar de l'objecte remot abans que pugui ser invocat.

Nota

Durant el desenvolupament de l'exemple no repetirem les especificacions de l'arquitectura bàsica RMI vista en l'apartat anterior, encara que insistirem en aquells punts clau que ajudin a la comprensió del codi.

Mostrarem tot el codi font de la nostra calculadora, que farà les funcions de guia durant tot l'apartat.

3.1.1. Desenvolupar l'objecte remot

Definim la part servidora, que consistirà de tres parts:

1) La **interfície remota**, que definirà els mètodes que poden ser invocats pel client.

2) La **implementació d'aquesta interfície**, que implementarà cadascun dels mètodes de la interfície. A més, pot implementar altres mètodes no remots en cas de ser necessaris per a treballar de manera local en el costat del servidor.

3) El **servidor** pròpiament dit, que durà a terme les tasques d'administració i execució del servidor: instal·lar un controlador de seguretat, instanciar l'objecte remot a partir de la interfície i la seva implementació, situar l'objecte remot en el registre per a fer-lo visible als clients i arrencar la part servidora fent possible que els clients invoquin mètodes remots.

Anem a descriure cadascuna de les tres parts juntament amb el codi generat de la nostra calculadora.

La interfície remota: definició de l'objecte remot

Com hem reiterat diverses vegades, no és possible accedir als objectes remots directament, solament són accessibles per mitjà de la seva interfície. Aquesta interfície haurà d'estendre la classe `Remote` per a identificar l'objecte com a remot i contenir la signatura de tots els mètodes remots, els quals per requisit han de llançar l'excepció `RemoteException` en la seva clàusula `throws`.

Tindrem tantes interfícies com objectes remots hi hagi en la nostra aplicació. Cada interfície es tractarà en un fitxer de codi a part. En el nostre exemple solament hi ha un objecte remot i, per tant, una sola interfície. Aquesta interfície està continguda en el fitxer *Calculadora.java*, amb el codi següent:

Calculadora

Veiem com es declaren la signatura dels quatre mètodes remots que es corresponen amb les quatre operacions que efectua la calculadora remota.

```
import java.rmi.*;
public interface Calculadora extends Remote {
  public int sumar(int a, int b) throws RemoteException;
  public int restar(int a, int b) throws RemoteException;
  public int multiplicar(int a, int b) throws RemoteException;
  public float dividir(int a, int b) throws RemoteException;
}
```

La implementació de la interfície: implementar l'objecte remot

Per a la implementació de la interfície, crearem la classe `CalculadoraImpl`. Com hem vist anteriorment, aquesta classe ha d'estendre la classe `UnicastRemoteObject` i atorgar-li capacitats per a manejar la major part de les tasques associades amb la creació d'un servidor RMI. També, com és lògic, ha d'implementar la interfície remota, en aquest cas la interfície `Calculadora`.

Impl

El sufix `Impl` s'usa com a conveni per a indicar que es tracta de la classe que implementa un objecte remot, encara que el nom d'aquesta classe és totalment lliure.

En el codi següent (fitxer *CalculadoraImpl.java*) podem veure la implementació dels mètodes que defineixen l'aritmètica de les operacions de la calculadora:

```
import java.rmi.*;
public class CalculadoraImpl extends
java.rmi.server.UnicastRemoteObject implements Calculadora {
   //Contructor
 public CalculadoraImpl() throws RemoteException {
   super();
 }
 public int sumar(int a, int b) throws RemoteException {
   return a + b;
 }
 public int restar(int a, int b) throws RemoteException {
   return a - b;
 }
 public int multiplicar(int a, int b) throws RemoteException {
   return a * b;
 }
 public float dividir(int a, int b) throws RemoteException {

    if(b==0) { //divisió per zero
   //llança excepció niada dins de RemoteException
    throw new RemoteException("",new ExcepcioDivisioPerZero(a));
   }
   return (float)a/(float)b;
 }
}
```

ArithmeticException

Java disposa de la classe `ArithmeticException` per a tractar condicions aritmètiques excepcionals, com la divisió per zero. Tanmateix, proposem una excepció pròpia (`ExcepcioDivisioPerZero`) per a tractar aquesta condició aritmètica particular com a exemple d'excepció remota niada.

La classe servidora

Per acabar amb la part servidora, crearem el servidor pròpiament dit, que durà a terme les tasques següents:

- **Establir un controlador de seguretat**. En aquest cas, és un objecte de la classe `RMISecurityManager`. Aquest controlador satisfà el requisit que imposa RMI per a descarregar els objectes al client, que són passats com a paràmetres durant la invocació de mètodes al servidor. En l'apartat següent es dóna més informació sobre els controladors de seguretat.

- **Crear l'objecte remot** `Calculadora` a partir de la classe que la implementa: `Calculadora c = new CalculadoraImpl();`.

- **S'assigna un nom identificador a l'objecte remot**, que servirà perquè els clients trobin aquest objecte remot en el moment de la invocació. El servidor assigna l'identificador `ServeiCalculadora` a l'objecte remot `Calculadora` i el registra en el servei de noms del sistema, que en el nostre cas és *RMIRegistry*. D'aquesta manera, els clients podran localitzar en el registre l'objecte remot `Calculadora` a partir d'aquest identificador. Per a efectuar el registre és necessari especificar l'URL d'*RMIRegistry*.

Calculadora

En el nostre exemple, treballarem amb la mateixa màquina per al client i el servidor. L'URL serà `rmi://localhost:1099` (*localhost* indica l'adreça local del sistema, que es correspon amb la IP `127.0.0.1`) i el port `1099` és l'assignat per conveni a *RMIRegistry*.

El codi del fitxer *CalculadoraServidor.java* queda de la manera següent:

```
public class CalculadoraServidor {
   public CalculadoraServidor() {
   try {
     if(System.getSecurityManager() == null)
         System.setSecurityManager(new java.rmi.RMISecurityManager());
     Calculadora c = new CalculadoraImpl();
     java.rmi.Naming.rebind("rmi://localhost:1099/ServeiCalculadora", c);
     }
     catch (Exception i) {
        System.out.println("Problema trobat: " + i); }
     }
     public static void main(String args[]) {
      new CalculadoraServidor();
    }
}
```

3.1.2. Creació del fitxer de polítiques

Com acabem d'esmentar, durant el desenvolupament del servidor, és necessari imposar un control de seguretat abans que el servidor pugui carregar alguna classe externa. Des de Java 1.2 és necessari configurar la política de seguretat explícitament com una propietat del sistema, en cas contrari, rebrem excepcions del tipus `SecurityException`. A aquesta propietat se li assigna com

a valor un fitxer de pòlisses que conté informació sobre tots els permisos explícits atorgats a la classe que es vol executar (en el nostre cas, `CalculadoraServidor`).

Hem vist en l'apartat "Arquitectura RMI" que aquest fitxer de polítiques, per conveni, l'anomenarem *java.policy*, i representa la política de seguretat que apliquem a un sistema que es vol protegir. En el nostre cas, aquest fitxer ha de contenir la informació a escala de permisos per a autoritzar al servidor la càrrega de classes externes. Per a les nostres finalitats, ha de proporcionar, com a mínim, els permisos de *connexió*, *acceptació*, *escolta* i *resolució* de tots els ports del sistema sense privilegis (per sobre del 1024) que pugui utilitzar l'aplicació, perquè client i servidor es comuniquin. El contingut mínim del fitxer *java.policy* hauria de ser:

```
grant {
permission java.net.SocketPermission
"*:1024-65535","connect,accept,listen,resolve";
};
```

Una vegada creat aquest fitxer, amb la informació pertinent, s'ha de trobar en el mateix directori de la classe servidora, perquè l'entorn RMI el trobi en el moment d'executar la classe `CalculadoraServidor`. Això s'aconsegueix configurant una propietat del sistema a partir del parell `nom_propietat=valor` en què `nom_propietat` = *java.security.policy*, i `valor` = *java.policy*.

3.1.3. Desenvolupar el client

Una vegada acabada la part del servidor, desenvoluparem el client, que únicament durà a terme tres tasques essencials:

1) Localització de l'objecte remot. El client buscarà l'objecte remot a partir del mètode `Naming.lookup()` i l'identificador de l'objecte. La situació és simètrica a la que acabem de veure en desenvolupar la classe servidora. Primer s'ha de contactar amb el servei de noms (en el nostre cas és *RMIRegistry*) a la mateixa adreça on hem registrat l'objecte durant el desenvolupament del servidor. Passant l'identificador per paràmetre de `lookup()`, el servei retornarà l'objecte remot de tipus `Object` que haurem de modelar (*cast*) al tipus que ens interessa (`Calculadora`). Atès que l'execució en principi és en un context local, l'adreça d'*RMIRegistry* serà `rmi://localhost:1099`, igual que en el servidor.

Port predeterminat

Encara que el port predeterminat pel qual escolta *RMIRegistry* és el 1099, es pot canviar per un altre sempre que ho canviem tant en el client com en el servidor.

2) Invocar un mètode de l'objecte remot. Una vegada localitzat i recuperat l'objecte tindrem capacitat per a invocar els seus mètodes. Això ho farem dins de la clàusula `try-catch`, per a atrapar objectes `RemoteException` que ens informin dels problemes sorgits en el servidor o la xarxa durant la invocació.

3) Rebre el resultat de l'execució, ja sigui amb el resultat esperat, o bé en forma de notificació sobre una excepció ocorreguda durant l'execució.

Finalment, el client està protegit per a diferents tipus de possibles excepcions que poden ocórrer:

- `MalformedURLException`: URL d'*RMIRegistry* no correcta.
- `RemoteException`: excepció del costat del servidor.
- `NotBoundException`: objecte remot no existent en el registre.
- `ArithmeticException`: condició aritmètica excepcional.
- `Exception`: qualsevol altra causa.

El codi client que es mostra a continuació espera 3 dades per línia d'ordre de la consola, per aquest ordre:

1) operand 1: un valor de tipus *int*
2) operador: un caràcter amb un dels 4 tipus d'operador possibles, que són +, -, x, /
3) operand 2: un valor de tipus *int*

A partir d'aquestes dades, el client seleccionarà el mètode remot per invocar segons el tipus d'operador indicat, passarà els dos operands com a paràmetres i rebrà el resultat de l'operació (sempre un enter), que mostrarà per la sortida estàndard del client mateix.

A continuació es mostra el codi del client, format per dues classes. Primer, mostrem el codi del fitxer *ExcepcioDivisioPerZero.java*, que és la classe que permet recollir i tractar en el costat del client l'excepció niada que ve dins de l'excepció llançada remotament. A continuació, mostrem el codi del client pròpiament dit (fitxer *CalculadoraClient.java*), que permet treballar amb la calculadora:

Calculadora

Noteu al codi que el missatge niat es troba al final del missatge remot llançat per `RemoteException`. La part precedent del missatge no ens interessa. Tanmateix, el podeu imprimir sencer substituint `substring(indexNiat+1)` per `substring(0)` i veure el missatge remot complet.

```
//Creem la classe d'excepció niada per a recollir-la al client
public class ExcepcioDivisioPerZero extends Exception {
   int dividend = 0;
   public ExcepcioDivisioPerZero (int pDividend) {
      dividend = pDividend;
   }
   public String getMessage() {
      return "%" + dividend + "/0";//% és un marcador
   }
}
//Implementació de la classe client
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
public class CalculadoraClient {
   public static void main(String[] args) {
      int op1, op2;
      String operador;
   if (args.length != 3) {
      System.out.println(
   "Error. Us: CalculadoraClient operand1 operador operand2");
      return;
      }
```

```
      try {
          op1 = Integer.parseInt(args[0]);
          op2 = Integer.parseInt(args[2]);
          operador = args[1];
             Calculadora c = (Calculadora)Naming.lookup(
             "rmi://localhost:1099/ServeiCalculadora");
      if (operador.equals("+"))
          System.out.println(c.sumar(op1, op2));
      else if (operador.equals("-"))
          System.out.println(c.restar(op1, op2));
      else if (operador.equals("x"))
          System.out.println(c.multiplicar(op1, op2));
      else if (operador.equals("/"))
          System.out.println(c.dividir(op1, op2));
      else
      System.out.println("Error. Us: CalculadoraClient operand1
      operador operand2");
        }
      catch (MalformedURLException murle) {
         System.out.println("MalformedURLException" + murle);
         }
      catch (RemoteException re) {
      //comprovem si porta la marca d'excepció niada
      int indexNiat = re.getMessage().indexOf('%');
      if(indexNiat!=-1)//porta niada: ExcepcioDivisioPerZero
         System.out.println("ExcepcioDivisioPerZero: "
         + re.getMessage().substring(indexNiat+1));
         //imprimim només el missatge niat
      else //no porta niada: RemoteException
            System.out.println("RemoteException" + re);
         }
      catch (NotBoundException nbe) {
         System.out.println("NotBoundException" + nbe);
         }
      catch (ArithmeticException ae) {
         System.out.println("java.lang.ArithmeticException"+ ae);
         }
      catch (Exception e) {
         System.out.println("java.lang.Exception" + e);
         }
    }
  }
```

En cas que el servidor es trobi en una altra màquina diferent del client, únicament és necessari canviar l'URL de l'*RMIRegistry*. Per exemple, si l'adreça IP de l'altra màquina fos 201.10.10.1, l'URL seria `rmi://201.10.10.1:1099/ServeiCalculadora` (o bé canviem la IP pel seu nom DNS equivalent).

3.2. Compilar i executar

Una vegada completat el codi font de l'aplicació, anem a compilar-la i executar-la seguint una sèrie de passos molt simples però ordenats. Abans de res, hem d'estar segurs que tots els fitxers de codi font es troben en el mateix directori dins del nostre sistema. Atesa la simplicitat de l'aplicació no és necessari disposar-la en diferents directoris. Aquests fitxers són: *Calculadora.java*, *CalculadoraImpl.java*, *CalculadoraServidor.java*, *CalculadoraClient.java*, i el fitxer de pòlisses *java.policy*.

Vegeu també

En el subapartat "Execució en un entorn distribuït" veurem com podeu organitzar els fitxers en un entorn d'execució distribuït i més complex.

Per al seguiment de la demostració que ve a continuació, treballarem en mode consola i executarem per línia d'ordres les eines bàsiques de desenvolupament de Java que vénen de sèrie amb el JDK. Primer hem de crear un directori de treball en el nostre sistema, que podem anomenar `CasEstudiRMI`, on desarem tots els fitxers de codi font, a més del fitxer de pòlisses *java.policy*.

Nota

Treballarem de manera independent del sistema operatiu. Per exemple, en Windows, la consola s'activa executant l'intèrpret d'ordres `cmd.exe`. En Mac, executem `Terminal` i en Linux, `bash`.

3.2.1. Compilar la interfície remota, servidor i client

En la consola, ens situem en el directori de treball que hem creat i executem per línia d'ordres el compilador de Java `javac` per a compilar tots els fitxers de codi font que hem creat fins ara, corresponents a la interfície remota, el servidor i el client.

Nota

Per al treball en consola, farem servir una notació genèrica. L'estudiant la podrà adaptar fàcilment al seu sistema operatiu.

```
CasEstudiRMI > javac *.java
```

3.2.2. Generar *stubs* i *skeletons* amb *rmic*

Una vegada tenim tot el codi compilat, disposem dels fitxers *.class* de codi binari en el mateix directori de treball. Entre aquests fitxers es troba *CalculadoraImpl.class*, amb la implementació dels objectes remots, a partir de la qual es generen els *stubs* i *skeletons* mitjançant el compilador d'*stubs* que proporciona l'entorn Java `rmic`.

Com hem vist anteriorment, a partir de la versió Java 1.5, ja no és necessari utilitzar `rmic` explícitament per a generar els *stubs*. Quant als *skeletons*, tampoc no són necessaris, en ser substituïts per la capacitat *reflexion* de Java. No obstant això, en les darreres versions de Java s'ofereix l'opció de generar tant els *stubs* com els *skeletons* per a mantenir la compatibilitat amb versions anteriors. En aquesta secció anem a generar tant els *stubs* com els *skeletons* només per entendre'n millor el funcionament.

Stubs

De fet, els *stubs* es generen sempre, encara que sigui de manera implícita, durant l'execució, i encara que no generin cap fitxer en el nostre directori de treball.

Executem l'ordre `rmic -keep -v1.1 CalculadoraImpl` per forçar la generació tant dels *stubs* com dels *skeletons*. L'opció `-v1.1` serveix per a forçar la generació de l'*skeleton* i fer compatible l'aplicació amb totes les versions des de Java 1.1. L'opció `-keep` serveix per a conservar els fitxers temporals (*stubs* i *skeletons*) en codi font. Podem comprovar en el nostre directori de treball com s'han generat els fitxers *CalculadoraImplStub.java* i *CalculadoraImplSkel.java*. Com sabem, aquestes classes són necessàries per a efectuar la comunicació entre client i servidor.

Vegeu també

Pel seu valor didàctic, es recomana l'estudi del codi dels *stubs* i *skeletons* juntament amb el seu funcionament explicat en l'apartat "Arquitectura RMI" d'aquest mòdul.

```
CasEstudiRMI > javac *.java
CasEstudiRMI > rmic -keep -v1.1 CalculadoraImpl
```

3.2.3. Arrencar el registre (*RMIRegistry*)

En aquest moment ja podem iniciar l'execució del servidor. Primer és necessari arrencar el servei de registre remot, que equival a arrencar *RMIRegistry*, perquè el servidor pugui registrar l'objecte remot i el client pugui accedir a aquest objecte remot.

Precaució

Per precaució, abans d'arrancar el servei, inicialitzarem la variable d'entorn `CLASSPATH` per a evitar problemes durant el registre d'objectes remots en cas que apunti al directori de treball.

Arrenquem *RMIRegistry* executant aquest servei per línia d'ordres de la consola mitjançant la instrucció `rmiregistry`. Aquest servei quedarà actiu (finestra de consola oberta) i escoltant per mitjà del port 1099.

Servidor virtual

En actuar com un servidor virtual, podríem arrancar *RMIRegistry* des de qualsevol directori del sistema.

```
CasEstudiRMI > javac *.java
CasEstudiRMI > rmic -keep -v1.1 CalculadoraImpl
CasEstudiRMI > SET CLASSPATH=
CasEstudiRMI > rmiregistry
```

3.2.4. Executar el servidor

Seguint el procés d'execució de la part servidora, una vegada tenim *RMIRegistry* en execució, és el moment d'arrencar el servidor. L'execució del servidor comportarà la creació de l'objecte remot `Calculadora` i el seu registre en *RMIRegistry*.

Per a dur a terme l'execució, obrim una segona consola i ens situem novament en el nostre directori de treball. Des d'allà, executem el servidor mitjançant l'ordre:

```
CasEstudiRMI > java -Djava.security.policy=java.policy CalculadoraServidor
```

L'opció `-D` permet configurar una propietat del sistema a la qual, com hem vist en apartats anteriors, assignem el fitxer `java.policy`, que s'hauria de trobar en el nostre directori de treball. De manera similar al registre de noms, una vegada el servidor és arrencat queda actiu (finestra de consola oberta) i pendent per a rebre invocacions de mètodes remots per part del client.

3.2.5. Executar el client

Finalment, executem el client i es duran a terme les invocacions als mètodes dels objectes remots. En la nostra calculadora remota el client sol·licita càlculs aritmètics mitjançant la invocació dels mètodes remots respectius.

Durant la invocació, s'envien els operands com a paràmetres juntament amb el tipus d'operador (+, -, x, /) proporcionats per l'usuari en el moment d'executar el client en la línia d'ordres. S'han de passar tres dades per línia d'ordres: *operand1*, *operador*, *operand2*.

Per a dur a terme l'execució, obrim una tercera finestra de consola. Ens situem novament en el nostre directori de treball i, des d'allà, executem el client amb aquesta ordre: `java CalculadoraClient 1 + 2`.

Precaució

No hem de tancar cap de les dues finestres de consola que tenim obertes, ja que això ocasionaria cancel·lar el servei de registre o apagar el servidor.

El resultat de l'execució es mostrarà en la mateixa consola que hem executat, bé amb el resultat de l'operació aritmètica, o bé amb la informació sobre un problema detectat durant la invocació.

```
CasEstudiRMI > java CalculadoraClient 1 + 2
3
```

Vegem ara un exemple d'execució problemàtica del client amb l'ordre `java CalculadoraClient 2 / 0`. En aquesta ocasió, el resultat de l'execució no és el resultat de la divisió, sinó la informació de l'excepció `ExcepcioDivisioPerZero` que ha llançat el servidor per haver sol·licitat una divisió per zero. Com s'ha comentat abans, aquesta excepció generada al costat del servidor arriba al client com a excepció niada de `RemoteException` i pel mateix conducte que el resultat d'una operació sobre la calculadora remota.

```
CasEstudiRMI > java CalculadoraClient 1 + 2
3
CasEstudiRMI > java CalculadoraClient 2 / 0
ExcepcioDivisioPerZero: 2/0
```

3.3. Automatització de tasques

Arribats a aquest punt, hem aconseguit dur a terme totes les tasques necessàries per a compilar i executar una aplicació distribuïda en un entorn RMI local. Aquesta aplicació ens ha estat útil per a mostrar pas per pas tot el procés bàsic que segueix qualsevol aplicació RMI desenvolupada i executada a mà, sense l'ajuda d'eines CASE[6] de suport al desenvolupador.

(6)De l'anglès *computer aided software engineering*.

Eines CASE

Les **eines CASE** són eines que ajuden a dur a terme les diferents activitats de l'enginyeria del programari. N'hi ha de molt populars, com l'Eclipse i el NetBeans, que donen suport a les activitats d'implementació, desplegament, execució i prova.

Encara essent una aplicació trivial, hem vist que el procés manual és relativament laboriós i implica un mínim de cinc passos, en els quals és necessari obrir diverses consoles per a arrencar i executar la part servidora i client. A més, en cas que volguéssim llançar diversos clients al mateix temps seria necessari obrir una consola nova per a cada client nou. Tot això converteix aquest procés manual en feixuc, especialment quan l'hem de repetir diverses vegades durant un procés de prova.

Una manera habitual d'automatitzar les tasques és crear un fitxer per lots que executi l'intèrpret d'ordres del sistema operatiu, amb totes les tasques que hem vist, i a més executi tants clients com sigui necessari. D'aquesta manera, executar aquest fitxer una única vegada serà equivalent a fer tots els passos que hem vist.

Fitxers per lots

En Windows, els fitxers per lots s'anomenen *batch* (fitxers *bat*). En Unix s'anomenen *shell scripts*.

A continuació, crearem un fitxer per lots que automatitzi el procés sencer de compilació i execució de la calculadora remota, creant dos clients consecutius. En funció del sistema operatiu, posarem extensió al fitxer i el desarem en el nostre directori de treball, juntament amb els fitxers de codi font de l'aplicació, des d'on executarem el fitxer.

Nota

Ens basem en Windows per a crear el fitxer per lots. Un cop creat l'executem escrivint el nom per línia d'ordres. Per a Unix, el contingut de l'*script* i l'execució és molt semblant.

```
@echo off
rem Fitxer calculadora.bat
rem Cas Estudi RMI: Calculadora remota
echo S'està compilant tot el codi...
javac *.java
echo Es creen els stubs i skeletons...
rmic -keep CalculadoraImpl
echo S'està inicialitzant CLASSPATH...
SET CLASSPATH=
echo S'està arrancant rmiregistry...
start rmiregistry
echo S'està arrancant el servidor CalculadoraServidor...
start java -Djava.security.policy=java.policy CalculadorServidor
echo Pausa per a donar temps perquè el servidor arrenqui
pause
echo Executa un client que demana sumar 1 + 2
java CalculadoraClient 1 + 2
echo Executa un altre client que demana dividir 2 / 0
java CalculadoraClient 2 / 0
echo Fi programa
pause
```

3.4. Execució en un entorn distribuït

Fins ara, el cas d'estudi de la calculadora remota l'hem executat entre dues JVM en un mateix ordinador (execució local). En aquest subapartat, veurem com podem executar la mateixa aplicació en un entorn distribuït, com a mínim entre dos ordinadors diferents, un com a client i un altre com a servidor. Si tenim accés a més de dos ordinadors ens podem apropar més a la realitat de les aplicacions distribuïdes connectant diversos clients amb un servidor.

Entorn distribuït

Per a fer la simulació d'un entorn distribuït podem considerar, per exemple, treballar amb l'ordinador de casa i el de l'oficina, l'ordinador d'un amic, etc.

Les característiques de simplicitat i transparència que presenta RMI fan que aquesta experiència sigui trivial, ja que únicament necessitem l'adreça IP del servidor i traslladar els fitxers de codi als ordinadors corresponents.

En el cas més simple, per a executar l'aplicació en dos ordinadors diferents, primer hem de decidir quin ordinador representarà el rol de client i quin el de servidor. A continuació, necessitem l'adreça IP pública de l'ordinador servidor per a poder accedir-hi.

Adreça IP pública

Podeu aconseguir la IP pública accedint des de l'ordinador servidor a un lloc web extern com http://www.whatismyip.com.

Una vegada obtinguda la IP pública (suposem que és: `83.36.234.52`), modificarem alguns fitxers del codi font del cas d'estudi per aconseguir que client i servidor es comuniquin fent que apuntin a la mateixa adreça d'*RMIRegistry* (que es troba en el servidor). Per tant, canviarem l'URL d'*RMIRegistry*, tant en el client com en el servidor amb la IP pública del servidor.

Precaució

Si els ordinadors implicats per a fer la prova utilitzen un encaminador (*router*), caldrà configurar-lo perquè deixi passar les connexions pel port d'*RMIRegistry* (1099). També pot caldre desconnectar els tallafocs per a deixar pas al trànsit que s'originarà durant l'execució.

En el cas del servidor, editem el fitxer *CalculadoraServidor.java* i canviem la línia:

```
java.rmi.Naming.rebind("rmi://localhost:1099/ServeiCalculadora", c);
```

per:

```
java.rmi.Naming.rebind("rmi://83.36.234.52:1099/ServeiCalculadora ", c);
```

En el cas del client, editem *CalculadoraClient.java* i actuem de manera semblant, canviant la línia:

```
Calculadora c = (Calculadora) Naming.lookup("rmi://localhost:1099/ServeiCalculadora");
```

per:

```
Calculadora c = (Calculadora) Naming.lookup ("rmi://83.36.234.52:1099/ServeiCalculadora");
```

Una vegada fetes aquestes modificacions, recopilem els fitxers modificats i traslladem tots els fitxers de l'aplicació als ordinadors corresponents, segons correspongui al client o al servidor. Aquests fitxers i les seves destinacions són:

Servidor	Client
Calculadora.class *CalculadoraImpl.class* *CalculadoraServidor.class* *CalculadoraImpl_stub.class* *CalculadoraImpl_skel.class*	*CalculadoraImpl_stub.class* *ExcepcioDivisioPerZero.class* *CalculadoraClient.class*

.class

Únicament és necessari traslladar el codi binari *.class*, no és necessari moure el codi font *.java*, si no és que volem fer modificacions posteriors.

La decisió del directori del sistema en què han d'anar instal·lats tant en la màquina servidora com en la màquina client és totalment irrellevant. En executar *RMIRegistry*, es procedirà a registrar els objectes, i hi crearà un lligam, de manera que el client trobarà sempre els objectes remots en el servidor, amb independència del directori on es trobin.

En aquest moment, ja tenim el necessari per a executar la nostra aplicació distribuïda. Únicament cal recordar que el servidor ha d'estar connectat a Internet, tenir l'*RMIRegistry* en execució i el servidor (`CalculadoraServidor`) executant-se. Podem utilitzar el mateix fitxer *Calculadora.bat* per a dur a terme totes aquestes tasques del costat del servidor. Des de l'ordinador que fa el rol de client (en el directori on tenim les classes del client), executem un client en una finestra de consola:

```
CasEstudiRMI > java CalculadoraClient 1 + 2
```

i el resultat de l'operació hauria de sortir a la consola.

Una vegada aconseguit el propòsit val la pena aturar-se un moment per analitzar el que hem aconseguit. Des del client hem fet una operació en la calculadora que no es troba en l'ordinador del client, sinó en un altre ordinador distant, amb la mateixa facilitat i percepció que si ho haguéssim invocat localment. Com a molt, percebrem una velocitat d'execució més lenta en comparació d'una execució local.

Resum

Hem vist que la programació distribuïda amb RMI fa possible el desenvolupament d'aplicacions distribuïdes en Java de manera simple i amb un alt grau de transparència.

Una de les característiques més importants que s'han vist i a la qual s'ha dedicat més espai en aquest mòdul és el funcionament del protocol de comunicació RMI, basat en la serialització de les dades que es transmeten per la xarxa. Aquest protocol és fonamental per a entendre el funcionament tant del mecanisme d'invocació remota RMI com de la comunicació interna de la tecnologia de components distribuïts de Java EE, que veurem extensament en el mòdul "Java EE".

Altres característiques importants que hem estudiat són la localització i activació d'objectes remots, el pas de paràmetres, la gestió dels objectes remots no referenciats (mecanisme de recollida d'objectes distribuïts), excepcions remotes i l'administrador de seguretat per a aplicacions basades en RMI.

Per altra part, hem estudiat en detall l'arquitectura bàsica d'RMI, fet que ens ha permès comprovar una vegada més la simplicitat i transparència en el funcionament d'invocació remota de Java, molt proper a la invocació local.

El cas d'estudi ens ha permès conèixer en la pràctica els conceptes apresos i ens ha de servir com a referència per al desenvolupament d'aplicacions distribuïdes amb RMI.

Activitats

1. Considereu dos exemples de beneficis i dos exemples de limitacions en treballar amb Java RMI, a partir de les característiques mencionades en l'apartat "Característiques RMI" d'aquest mòdul.

2. Donats els sistemes de programari següents per desenvolupar, estudieu en cada cas si és adient l'ús de Java RMI; i, en cas afirmatiu, com en podem aprofitar el potencial.

a) Un gestor de finestres en un entorn d'escriptori.

b) Un sistema de vot electrònic.

c) Un sistema de gestió del correu electrònic via Web, com per exemple, Hotmail.

d) Un controlador remot, per exemple, per a controlar a distància la pressió d'un dipòsit de gas.

e) Una aplicació de campus virtual com, per exemple, el campus UOC.

f) Un programari que gestiona el GPS d'un cotxe.

3. Reimplementeu la calculadora remota amb els requisits següents:

a) Afegir l'operació d'arrel quadrada (cal controlar l'excepció en cas de proporcionar nombres negatius per al càlcul de l'arrel quadrada).

b) Afegir les operacions inversa (1/x) i %.

c) Fer operacions amb nombres racionals.

d) Desplegueu la nova calculadora remota en un entorn real. Escriviu un fitxer *.bat* per a l'automatització de les tasques de desplegament de la calculadora remota.

4. Una companyia de subministrament de gas vol una aplicació distribuïda per a controlar alguns paràmetres dels seus tancs de gas, com són la temperatura i la pressió. Actualment, la companyia controla aquests paràmetres *in situ* i modifica (incrementa/decrementa) els valors dels paràmetres segons sigui necessari.

Amb l'aplicació distribuïda la companyia vol fer un control remot des de les seves oficines, per evitar desplaçaments i, també, pel perill que representa haver de controlar els paràmetres als tancs. Per a això, anem a suposar que al tanc de gas hi ha un sensor que permet llegir i modificar la pressió i un sensor que permet llegir i modificar la temperatura. Els valors dels paràmetres han d'estar sempre en els intervals $[P_{min}, P_{max}]$ i $[T_{min}, T_{max}]$, respectivament, en què P_{min}, P_{max} i T_{min}, T_{max} són valors crítics. En cas que el valor d'un paràmetre sortís de l'interval, l'alarma es dispararia, cosa que avisaria un empleat a intervenir per mitjà de l'aplicació per modificar el valor a un valor permès. Feu ús de les excepcions remotes niades per a controlar els valors no permesos dels paràmetres de pressió i temperatura.

5. Implementeu un convertidor de temperatura remot que permeti convertir un valor de la temperatura de Celsius a Fahrenheit i al revés: quan s'introdueixi en °C ho ha de convertir a °F i quan s'introdueixi en °F ho ha de convertir a °C.

Desplegueu el convertidor de temperatura remot en un entorn real. Escriviu un fitxer *.bat* per a l'automatització de les tasques del desplegament del convertidor de temperatura remot.

6. Implementeu un diccionari remot per a consultar els termes informàtics més comuns. Per exemple, si un client introdueix "CPU" se li retornaria la consulta com a "*central processing unit*". Useu les excepcions remotes per a controlar entrades inexistents en el diccionari.

7. Implementeu un sistema de votació remot que permeti als clients d'una companyia respondre una pregunta, amb la resposta com a "Sí"/"No". Al costat del servidor s'haurà de possibilitar conèixer el nombre total de votants, el nombre de vots "Sí" i el nombre de vots "No".

Exercicis d'autoavaluació

1. Definiu *RMI* i dieu quins són els seus objectius principals.

2. Expliqueu breument la relació entre RMI, *sockets* i *streams*.

3. Què és la serialització, quina és la seva funció principal i com s'aconsegueix en Java?

4. Quins tipus de paràmetres són possibles en RMI i com es fa el pas de paràmetres per a cada un durant una invocació remota?

5. Indiqueu els elements bàsics que formen l'arquitectura RMI.

6. Comenteu les diferències principals entre *stubs* i *skeletons*.

7. Què és una interfície remota i com s'aconsegueix en RMI? Quina classe de Java estén una classe servidora en RMI?

8. Què és *rmic*, per a què serveix i en quina etapa del desenvolupament de l'aplicació distribuïda RMI s'aplica?

9. Com es registra un objecte remot en RMI? Com es localitza un objecte remot en RMI?

10. Quines solucions proporciona Java dins del seu model de seguretat per a aplicacions distribuïdes amb RMI?

Solucionari

Exercicis d'autoavaluació

1. RMI representa el model d'objectes distribuïts que proposa Java com a solució per a desenvolupar aplicacions distribuïdes de manera simple, intuïtiva, versàtil i transparent.

Els objectius d'RMI són la transparència, la simplicitat, la seguretat, la portabilitat, l'escalabilitat, la robustesa, l'eficiència i la fidelitat al paradigma d'orientació a objectes de Java.

2. En RMI, les dades que passem per paràmetre durant la invocació d'un mètode remot són serialitzades i convertides en un *stream* de bytes pel procés que invoca. Aquest procés escriu en un *socket* que retransmet l'*stream* per la xarxa, fins a arribar al *socket* de destinació.

3. La serialització és l'acció de codificar un objecte en una seqüència de dades. En el context d'RMI, l'objectiu de la serialització és la codificació de les dades per transmetre per la xarxa.

El mecanisme de serialització en Java consisteix a convertir classes d'objectes en una seqüència de bytes. Si invertim aquest procés, obtindrem una còpia de l'objecte original a partir d'aquests mateixos bytes de dades, i farem el que es coneix com a *desserialització*.

4. Durant una invocació a un mètode d'un objecte remot, els paràmetres que conté la invocació poden ser de tres tipus possibles: tipus primitius (*int*, *char*, *float*, etc.), que es passen per valor, objectes serialitzables (o no remots), que es passen per valor, i objectes remots, que es passen per referència.

5. L'arquitectura bàsica en l'entorn RMI, per a dur a terme una invocació d'un mètode remot, està formada per un objecte client, els *stubs* al costat del client, l'objecte servidor i els *skeletons* al costat del servidor, els *sockets* per a la comunicació entre *stubs* i *skeletons*, *RMIRegistry* per a la localització de l'objecte remot i el protocol de xarxa TCP/IP al nivell de transport de la xarxa entre la JVM local i la remota.

6. Un *stub* es troba al costat del client i és una representació local d'un objecte remot del servidor, que referencia tots els mètodes remots d'aquest objecte. De manera equivalent, un *skeleton* representa un *skeleton* de l'objecte remot i es troba al costat del servidor.

La funció principal d'un *stub* és crear i mantenir una connexió de *sockets* oberta amb l'*skeleton* del servidor i responsabilitzar-se de la tasca de serialitzar i desserialitzar les dades en el costat del client. De manera semblant, l'*skeleton* es responsabilitza del manteniment de la connexió de *sockets* amb l'*stub*, i també de serialitzar i desserialitzar les dades al costat del servidor.

7. Els clients d'una aplicació distribuïda RMI no referencien directament els objectes remots, sinó una interfície remota que és implementada per l'objecte servidor. En el cas més simple, en el servidor hi ha una classe *interfície remota* que estén la interfície `Remote` amb la signatura de tots els mètodes remots que poden ser invocats pel client (és a dir, aquells que llancin `RemoteException` en la seva clàusula *throws*).

8. *rmic* és un compilador que permet generar els *stubs* i *skeletons*. Una vegada tenim tot el codi font (interfície remota objecte client i servidor) compilat i disposem dels fitxers *.class* de codi binari, a partir d'aquests fitxers es generen els *stubs* i *skeletons* mitjançant el compilador d'*stubs* que proporciona l'entorn Java *rmic*.

Tot i això, a partir de la versió Java 1.5, ja no és necessari utilitzar *rmic* explícitament per a generar els *stubs*. Quant als *skeletons*, tampoc no són necessaris, en ser substituïts per la capacitat *reflexion* de Java. No obstant això, és recomanable generar tant els *stubs* com els *skeletons* per a mantenir la compatibilitat amb versions anteriors.

9. La localització d'objectes funciona com un directori o servei de noms: un nom lògic o identificador de l'objecte es correspon amb l'*stub* de l'objecte. Durant el registre, el servidor ha d'enviar l'*stub* de l'objecte al directori per a registrar-lo creant un lligam entre un nom identificador de l'objecte i l'*stub* de l'objecte. D'aquesta manera, les aplicacions client únicament han de conèixer la localització del directori i el nom identificador de l'objecte per a trobar-lo en temps d'execució.

RMI proporciona de sèrie un servei de noms molt simple anomenat *RMIRegistry*, que permet al servidor enregistrar els objectes remots perquè els clients els puguin trobar.

10. El model de seguretat de Java per a aplicacions distribuïdes amb RMI és de tres tipus: administradors de seguretat RMI, xifratge a escala de *sockets* i permisos personalitzats.

Glossari

activació d'objectes remots *f* Procés que instancia objectes remots que han estat prèviament desactivats o destruïts. La desactivació d'objectes remots fa el procés invers.

interfície remota *f* Interfície dels mètodes de l'objecte remot que estableix una separació entre les signatures dels mètodes i el codi d'aquests mètodes.

invocació local *f* Crida a un mètode d'un objecte a un altre objecte en el context d'una mateixa JVM.

invocació remota *f* Crida a un mètode d'un objecte que es troba en una JVM a un altre objecte que es troba en una JVM separada. Les dues JVM (o instàncies de la mateixa JVM) no han d'estar necessàriament separades físicament i poden córrer a la mateixa màquina.

Java virtual machine *f* Entorn on s'executa el codi compilat (*bytecode*) de Java. Hi ha versions de JVM per a la majoria dels sistemes operatius, fet que dóna la qualitat d'independència a la plataforma de Java. A partir de crear diverses instàncies d'una mateixa JVM es pot simular una invocació remota en una mateixa màquina.
Sigla **JVM**

model d'objectes distribuïts de Java *m* Sistema distribuït basat en el model client-servidor en què els objectes són administrats per servidors i els clients invoquen els mètodes d'aquests objectes amb una invocació remota.

objecte client *m* Objecte que invoca objectes remots i pot rebre els resultats de les invocacions.

objecte remot *m* Objecte servidor administrat per un servidor que pot ser invocat per un objecte client.

objecte servidor *m* Objecte que gestiona els objectes remots que els objectes clients busquen per invocar els seus mètodes i tota la infraestructura necessària per a rebre les invocacions i retornar els resultats als objectes clients.

paràmetre per referència *m* Referències directes de les instàncies dels objectes passats per paràmetre durant una invocació.

paràmetre per valor *m* Còpies de les instàncies originals dels objectes passats per paràmetre durant una invocació. Els tipus primitius de dades també es passen per valor durant una invocació.

recollidor d'escombraries distribuït *m* Mecanisme automàtic d'eliminació d'objectes en sistemes distribuïts en què tots els objectes del sistema són visitats constantment i, aquells que no són referenciats per cap altre objecte, s'eliminen. Aquest mecanisme és equivalent al que s'usa en entorns locals.

serialització *f* Mecanisme que consisteix a convertir classes d'objectes en una seqüència de bytes. Si invertim aquest procés, obtindrem una còpia de l'objecte original a partir d'aquests mateixos bytes de dades (desserialització).

servei de noms o localització *m* Servei que permet trobar objectes en un sistema distribuït i connectar amb la màquina servidora on es troba l'objecte els mètodes del qual es volen invocar.

skeleton *m* Fragment d'un objecte que representa un esquelet de l'objecte remot i es troba al costat del servidor.

stream *m* Estructura de dades que permet emmagatzemar i recuperar informació de manera seqüencial.

***stream* intermedi** *m* *Stream* que envolta altres *streams* existents (*streams* primitius o intermedis) per a guanyar en funcionalitat.

***stream* primitiu** *m* *Stream* que parla amb els dispositius directament llegint o escrivint les dades tal com arriben de manera seqüencial.

stub *m* Fragment d'un objecte que es troba en el costat del client i és una representació local d'un objecte remot del servidor que referencia tots els mètodes remots d'aquest objecte.

transparència d'accés *f* Capacitat d'un sistema d'oferir sempre les mateixes interfícies en fer les crides als seus serveis.

transparència de localització o ubicació *f* Capacitat d'un sistema d'oferir un servei sense preocupar-se d'on està ubicat físicament aquest servei.

Bibliografia

Bibliografia bàsica

Com a bibliografia bàsica per a complementar aquest mòdul, recomanem les obres següents:

Caballé, S.; Xhafa, F. (2008). *Programación distribuida en Java con RMI*. Madrid: Delta Publicaciones Universitarias.

Grosso, W. (2001). *Java RMI*. O'Reilly.

Pitt, E.; McNiff, K. (2001). *Java.rmi. The Remote Method Invocation Guide*. Addison Wesley.

Bibliografia complementària

Per a complementar alguns aspectes més concrets, recomanem:

Emmerich, W. (2000). *Engineering Distributed Objects*. Wiley.

Jaworski, J.; Perrone, P. J. (2001). *Seguridad en Java*. Madrid: Pearson Alhambra.

Referències bibliogràfiques

Oracle (2011). "RMI". *The Java Tutorials*. [Data de consulta: agost de 2011]: http://download.oracle.com/javase/tutorial/rmi/.

Wollrath, A.; Riggs, R.; Waldo, J. (1996). *A Distributed Object Model for the Java System*. Sun Microsystems. [Data de consulta: gener de 2012]: http://pdos.csail.mit.edu/6.824/papers/waldo-rmi.pdf.