Удаленное развертывание с помощью RMI

Распределенные вычисления

Все говорят, что поддерживать отношения на расстоянии — это сложно. Но только не с RMI. Неважно, как далеко мы друг от друга *на самом деле*, ведь с RMI мы *как будто* совсем рядом.

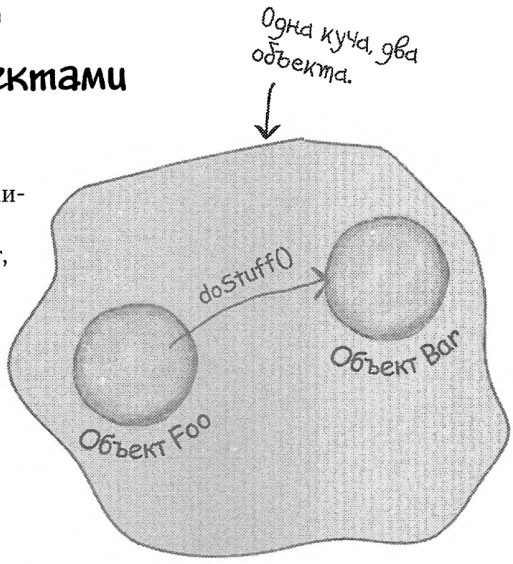
Находиться далеко — это не всегда плохо. Конечно, задача упрощается, когда компоненты приложения собраны в одном месте и всем управляет одна JVM. Но этого не всегда можно добиться. И это не всегда целесообразно. Как быть, если приложение выполняет сложные вычисления, но должно работать на маленьком симпатичном пользовательском устройстве? Что делать, если приложению нужна информация из базы данных, но в целях безопасности к ней может получить доступ только код на вашем сервере? Представьте большой торговый сервер, на котором работает система управления транзакциями. Иногда одна часть вашего приложения *должна* выполняться на сервере, а другая — на удаленном (как правило, клиентском) компьютере

Вызовы методов между двумя объектами всегда происходят в одной куче

До этого момента и вызывающий код, и все методы в книге принадлежали объектам, выполняющимся в одной виртуальной машине. Иными словами, оба объекта (тот, который вызывает метод, и тот, из которого вызывают) находились в одной куче.

```
class Foo {
    void go() {
        Bar b = new Bar();
        b.doStuff();
    }
    public static void main (String[] args) {
       Foo f = new Foo();
       f.go();
    }
}
```

Мы знаем, что в этом коде экземпляр Foo, на который ссылается f, и объект Bar, на который ссылается b, находятся в одной куче и управляются одной JVM. Помните, JVM отвечает за то, какие биты хранятся внутри ссылочной переменной, то есть как представлен *путь к объекту в куче*. JVM всегда знает местонахождение всех объектов и способ до них добраться. Но только в том случае, если эти объекты находятся в ее *собственной* куче! Например, вы не можете сделать так, чтобы JVM, выполняющаяся на одном компьютере, знала размер кучи JVM, запущенной на *другой* машине. Фактически это утверждение справедливо даже для тех случаев, когда виртуальных машин две, а компьютер *один*. Количество физических устройств не играет роли; важно только то, что мы имеем дело с двумя разными экземплярами JVM.



Когда один объект вызывает метод из другого, в большинстве приложений они принадлежат одной куче. Иначе говоря, оба управляются одной JVM.

Что делать, если нужно вызвать метод из объекта, который выполняется на другом компьютере?

Мы знаем, как передавать информацию между компьютерами, — нужно использовать сокеты и ввод/вывод. Открываем серез сокет соединение с другой машиной, получаем поток OutputStream и записываем туда данные.

Но что делать, если нужно *вызвать метод* из объекта, выполняющегося на другом компьютере... на другой JVM? Конечно, можно создать собственный протокол, чтобы отправлять данные на ServerSocket, разбирать их, выяснять, что именно от нас хотят, выполнять работу и возвращать результат. Но это очень хлопотно. Подумайте, как бы все упростилось, если бы можно было получать ссылку на объект, размещенный на другом компьютере, и просто вызывать нужный метод.

Представьте, что у нас два компьютера...

У Здоровяка есть то, чем Кроха хотел бы обладать. Вычислительная мощь.

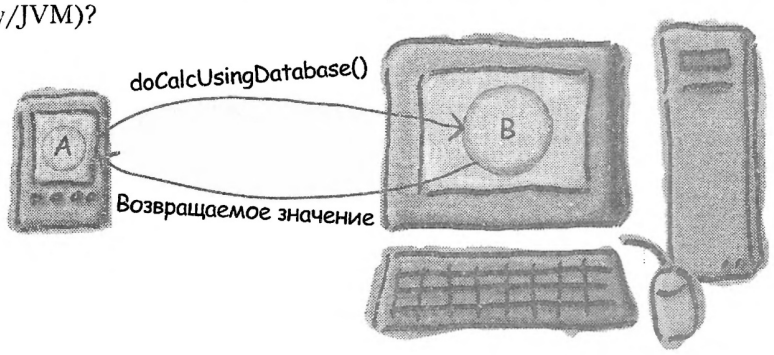
Кроха хочет передать Здоровяку данные, чтобы тот обработал их. Крохе хочется просто вызвать метод...

```
double doCalcUsingDatabase(CalcNumbers numbers)
```

и получить результат.
Но как ему получить ссылку на объект, принадлежащий Здоровяку?

Объект A, работающий на Крохе, хочет вызвать метод из объекта B, работающего на Здоровяке

Как добиться того, чтобы объект на одном компьютере смог вызвать метод из другого компьютера (а это подразумевает другую кучу/JVM)?



Но вы не можете этого сделать

Во всяком случае, не можете напрямую. Нельзя получить ссылку на элемент в другой куче. Допустим, вы напишете:

Dog d = ???

Элемент, на который ссылается d, должен находиться в той же куче, что и код с этим выражением.

Но представьте, что вам захотелось разработать систему, которая будет использовать сокеты и ввод/вывод, чтобы связать воедино рассматриваемый процесс (вызов метода из объекта, выполняющегося на другом компьютере) так, *будто* вы вызываете локальный метод.

Проще говоря, вы хотите вызывать метод из *удаленного* объекта (то есть объекта из другой кучи), но чтобы код при этом *выглядел* так, словно вызов локальный. Прозрачность старых добрых вызовов и возможность работы с удаленными методами — это наша цель.

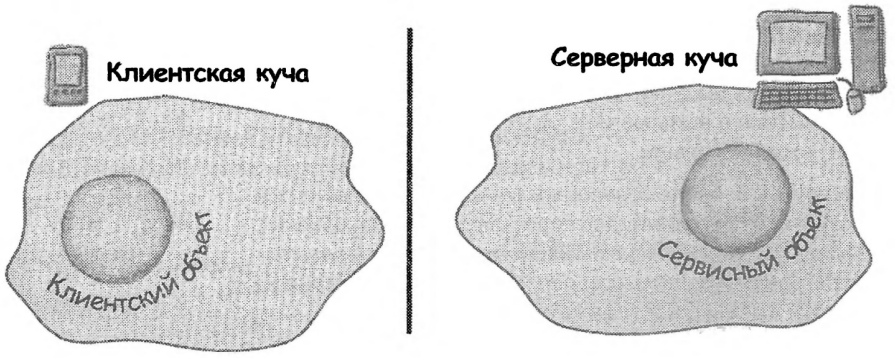
Именно это дает вам RMI (технология вызова удаленных методов)!

Но вернемся немного назад и рассмотрим, как бы мы спроектировали RMI. Понимание того, что нужно для создания подобной технологии, позволит лучше понять принцип ее работы.

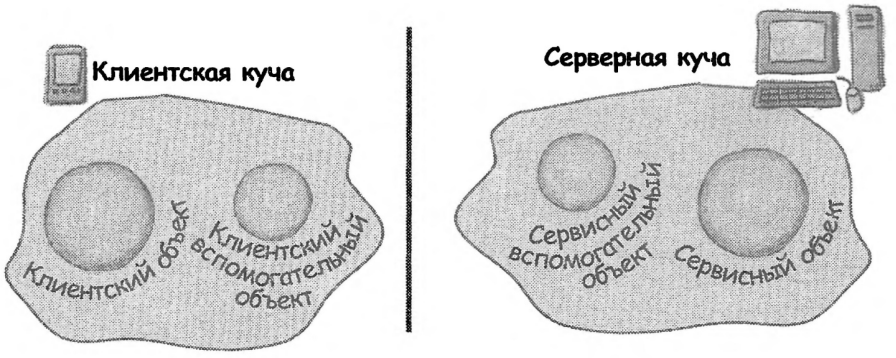
Проектирование системы вызова удаленных методов

Создадим четыре сущности: сервер, клиент, вспомогательный серверный и вспомогательный клиентский объекты.

1. Создаем клиентское и серверное приложения. Второе будет выступать в роли **удаленного сервиса**, хранящего объект, метод из которого клиент хочет вызвать.



2. Создаем вспомогательные объекты для сервера и клиента. Они возьмут на себя все низкоуровневые операции с сетью и вводом/выводом, чтобы ваши клиент и сервис могли вести себя так, будто находятся в одной куче.



Роль вспомогательных объектов

Вспомогательные объекты отвечают непосредственно за соединение. Они позволяют клиенту *вести* себя так, словно он вызывает метод из локального объекта. Фактически так оно и есть. Клиент вызывает метод из вспомогательного объекта *как из сервиса. Вспомогательный клиентский объект выступает в роли прокси для настоящего сервиса.*

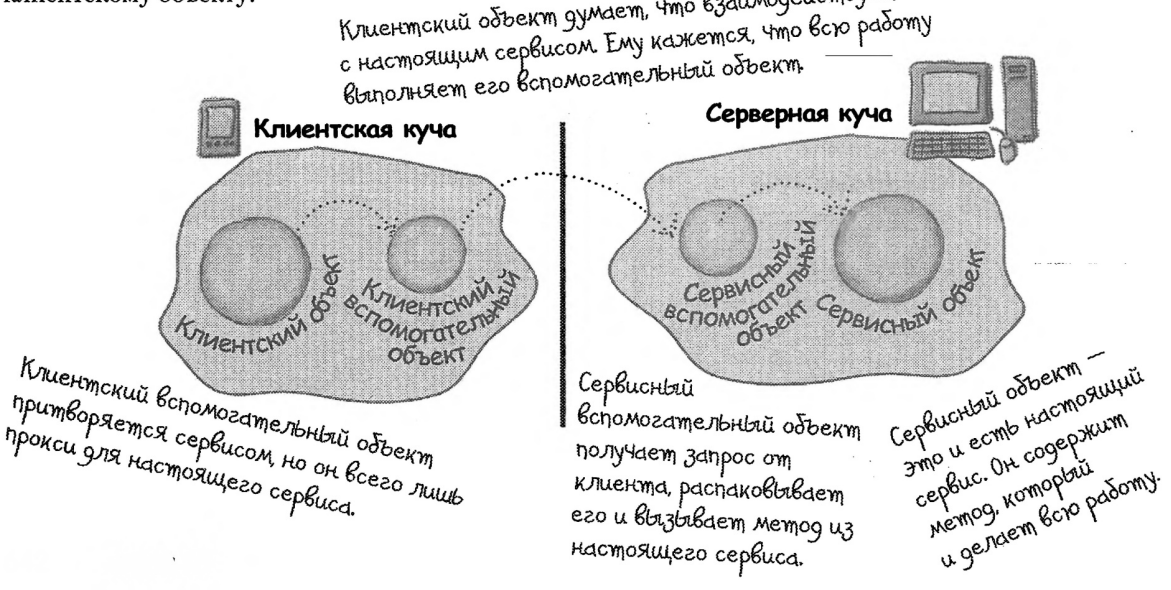
Иными словами, клиентский объект *думает*, что вызывает метод из удаленного сервиса, так как вспомогательный объект «притворяется» им. ***Он притворяется сущностью с методом, который клиент хочет вызвать!***

Но клиентский вспомогательный объект — это на самом деле не сервис. Хоть он и *ведет* себя именно так (потому что содержит метод, наличие которого декларирует сервис), у него нет логики (соответствующего кода), на которую рассчитывает клиент. Вместо этого клиентский вспомогательный объект связывается с сервером, передает ему информацию о вызываемом методе (то есть имя метода, аргументы и т. д.) и ожидает ответа.

В это же время на другой стороне серверный вспомогательный объект получает запрос от клиента (соединяясь через сокет), распаковывает информацию о вызове и запускает *настоящий* метод из *настоящего* сервисного объекта. Таким образом, для сервисного объекта это выглядит как локальный вызов. Он инициируется на том же компьютере вспомогательным объектом, а не удаленным клиентом.

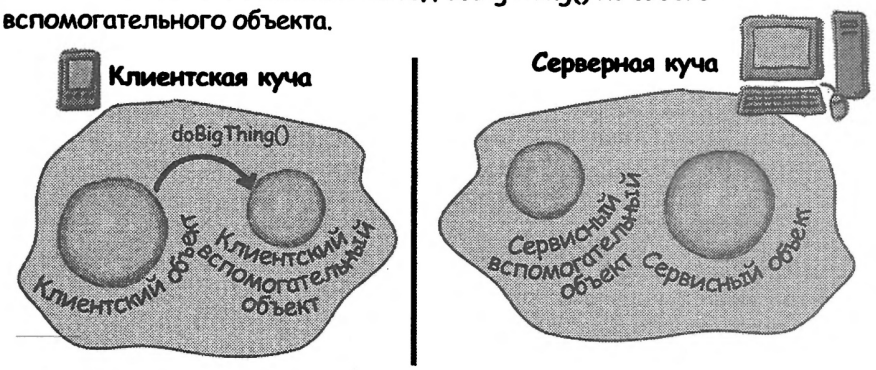
Сервисный вспомогательный объект получает результат от сервиса, упаковывает его и отправляет клиенту (по исходящему потоку через сокет). Там результат принимается клиентским вспомогательным объектом, информация распаковывается и значение возвращается клиентскому объекту.

Ваш клиентский объект ведет себя так, будто вызывает удаленный метод. На самом деле он лишь запрашивает методы из локального прокси-объекта, находящегося в одной куче с ним и скрывающего все низкоуровневые подробности работы сокетов и потоков.

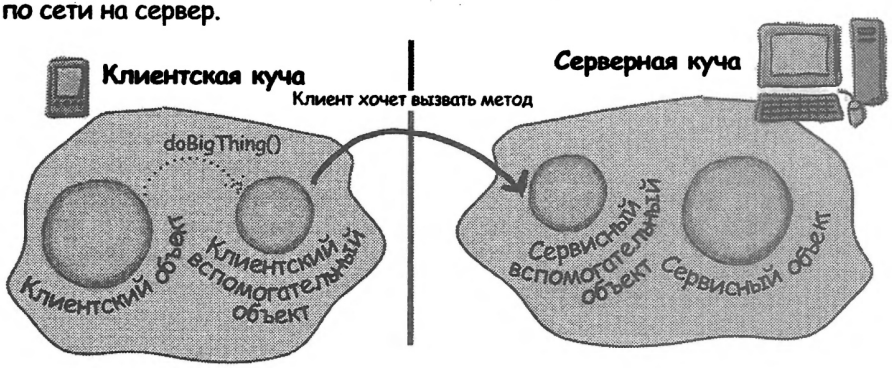


Как осуществляется вызов метода

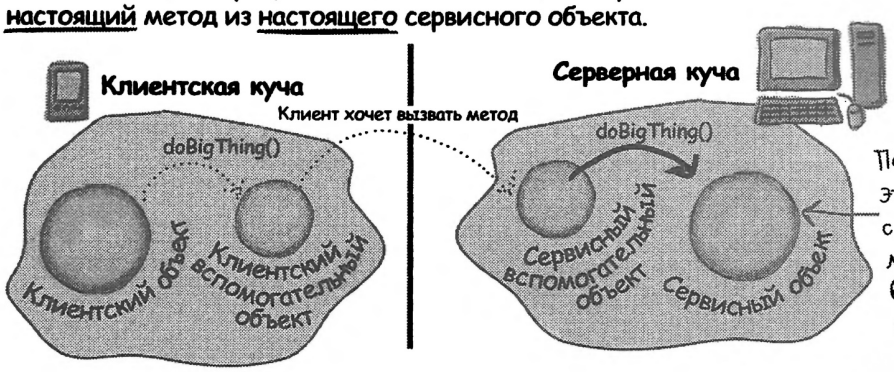
1. Клиентский объект вызывает метод doBigThing() из своего вспомогательного объекта.



2. Вспомогательный объект упаковывает информацию о вызове (аргументы, имя метода и т. д.) и пересылает ее по сети на сервер.



3. Сервисный вспомогательный объект распаковывает информацию, полученную от клиента, определяет, какой метод (и откуда) необходимо вызвать, и запускает настоящий метод из настоящего сервисного объекта.



Помните, что это объект с настоящим методом, в котором выполняется настоящая работа.

Технология RMI предоставляет клиентские и сервисные вспомогательные объекты!

В Java RMI создает для вас клиентские и сервисные вспомогательные объекты. Кроме того, эта технология сама знает, как сделать клиентский вспомогательный объект похожим на настоящий сервис. Проще говоря, она умеет создавать объекты, которые содержат те же методы, что и удаленный сервис.

К тому же RMI обеспечивает всю необходимую инфраструктуру для работы, включая поисковый сервис, чтобы клиент мог найти и получить клиентский вспомогательный объект (прокси для настоящего сервиса).

Используя RMI, вы избавляетесь от необходимости работать с сетевым кодом или вводом/выводом. Клиент сможет вызывать удаленные методы (содержащиеся в настоящем сервисе), как будто они принадлежат обычному объекту, работающему под управлением той же локальной JVM.

Почти.

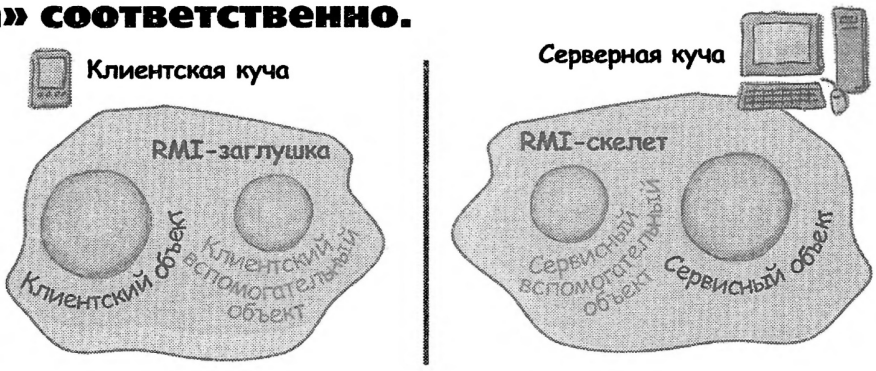
Есть одно различие между вызовами локальных (обычных) методов и RMI-вызовами. Хотя с точки зрения клиента вызов метода локален, вспомогательный объект пересылает его по сети. Идет работа с сетью и вводом/выводом. А что вы знаете о ней?

Она рискованная!

Во время такой работы в любом месте могут быть выброшены исключения, поэтому клиент должен учитывать риск. Он должен понимать, что при вызове удаленного метода, даже если при этом используется локальный вспомогательный/прокси-объект, *в конечном счете* задействуются сокеты и потоки. Изначально вызов клиента локален, но прокси делает его *удаленным*. Иными словами, вызывается метод из объекта, управляемого другой JVM. *Как* информация будет передаваться от одной виртуальной машины к другой, зависит от протокола, используемого вспомогательными объектами.

RMI позволяет выбрать один из двух протоколов — JRMP или IIOP. Первый считается стандартным протоколом для RMI, специально созданным для вызовов между Java-приложениями. IIOP предназначен для технологии CORBA (Common Object Request Broker Architecture — общая архитектура брокера объектных запросов) и позволяет делать удаленные запросы к сущностям, которые не обязательно должны быть Java-объектами. Как правило, технология CORBA *намного* сложнее в применении, чем RMI, так как на обоих концах может быть что угодно, и необходимо выполнять чрезвычайно много работы по транслированию и обработке информации.

В RMI клиентский и серверный вспомогательные клиенты выступают в роли «заглушки» и «скелета» соответственно.



Создание удаленного сервиса

Здесь мы кратко перечислим пять шагов, которые требуются для создания удаленного сервиса (выполняющегося на сервере). Не волнуйтесь, каждый шаг подробно объясняется на следующих нескольких страницах.

Шаг первый

Создаем удаленный интерфейс.

Этот интерфейс описывает методы, которые клиент может вызывать удаленно. Он будет использоваться клиентом в качестве полиморфического типа класса для вашего сервиса. Его должны реализовывать как «заглушка», так и сам сервис!

В этом интерфейсе объявляются удаленные методы, которые будет вызывать клиент.

Шаг второй

Создаем реализацию удаленного интерфейса.

Это класс, который выполняет настоящую работу. Он содержит реализацию методов, объявленных в удаленном интерфейсе. Клиент будет вызывать методы из объекта этого класса.

Настоящий сервис. Класс с методами, которые выполняют всю работу. Он реализует удаленный интерфейс.

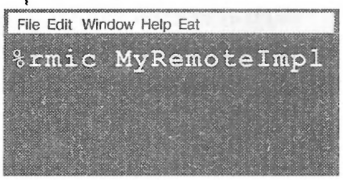
На выходе получаем два новых класса для вспомогательных объектов.

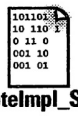
Шаг третий

Генерируем «заглушки» и «скелеты» с помощью утилиты rmic.

Это клиентские и серверные вспомогательные объекты. Их не нужно писать вручную или даже заглядывать в исходный код. Все это делается автоматически с помощью утилиты rmic, которая поставляется вместе с пакетом разработки Java (JDK).

Выполнение rmic для класса, реализующего сервис...

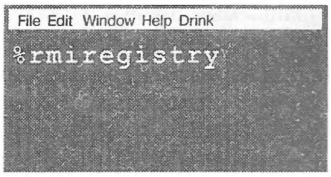




Шаг четвертый

Запускаем реестр RMI (rmiregistry).

rmiregistry — нечто вроде белых страниц телефонного справочника. Это место, куда пользователь обращается, чтобы получить прокси (клиентскую «заглушку»/вспомогательный объект).

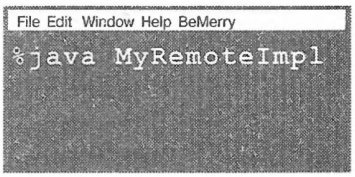


Запустите это в отдельном терминале.

Шаг пятый

Запускаем удаленный сервис.

Вы должны сделать так, чтобы сервисный объект начал работать. Для этого нужно создать экземпляр класса, реализующего сервис, и поместить его в реестр RMI. Пройдя регистрацию, он станет доступным для клиентов.



Шаг первый: создаем удаленный интерфейс

MyRemote.java

1 Расширяем java.rmi.Remote.

Remote — это интерфейс-маркер (у него нет методов). Он особо значим для RMI, поэтому его нельзя игнорировать. Обратите внимание, что мы используем здесь слово «расширять». Один интерфейс может *расширять* другой.

```
public interface MyRemote extends Remote {
```

Ваш интерфейс должен объявить, что он предназначен для вызовов удаленных методов. Интерфейс не может ничего реализовывать, но может расширять другой интерфейс.

2 Объявляем, что все наши методы выбрасывают исключение RemoteException.

Удаленный интерфейс применяется клиентом в качестве полиморфического типа для сервиса. Иными словами, клиент вызывает методы из объекта, реализующего удаленный интерфейс. В роли этого объекта, конечно же, выступает «заглушка», и, поскольку она работает с сетью и вводом/выводом, могут произойти разные неприятные вещи. Клиент должен учитывать риски, обрабатывая или объявляя удаленные исключения. Во втором случае код, который вызывает методы из ссылки этого типа (типа интерфейса), также должен обработать или объявить подобные исключения.

```
import java.rmi.*;
```

Интерфейс Remote находится в пакете java.rmi.

```
public interface MyRemote extends Remote {
    public String sayHello() throws RemoteException;
```

Каждый вызов удаленного метода рассматривается как опасный. Объявляя для любого из них исключение RemoteException, вы заставляете клиента обратить внимание на то, что код может не работать.

3 Убедитесь, что аргументы и возвращаемые значения — это примитивы или реализации интерфейса Serializable.

Аргументы и возвращаемые значения удаленного метода должны быть либо примитивами, либо реализациями интерфейса Serializable. Подумайте об этом. Любой аргумент для удаленного метода упаковывается и пересылается по сети — и все благодаря сериализации. То же самое касается возвращаемых значений. При использовании примитивов, строк и большинства классов из API (включая массивы и коллекции) вам нечего опасаться. Если же вы передаете объекты собственных типов, убедитесь, что они реализуют Serializable.

```
public String sayHello() throws RemoteException;
```

Это возвращаемое значение будет переправлено по проводам от сервера к клиенту, поэтому оно должно быть сериализуемым. Именно так аргументы и возвращаемые значения упаковываются и отправляются.

Шаг второй: создаем реализацию удаленного интерфейса

MyRemoteImpl.java

1 Реализуем удаленный интерфейс.

Сервис должен реализовывать удаленный интерфейс, методы которого будет вызывать ваш клиент.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {
        return "Сервер говорит: 'Привет'";
    }
    // Остальной код класса
}
```

Компилятор проверит, реализованы ли все методы этого интерфейса. В данном случае метод всего один.

2 Расширяем UnicastRemoteObject.

Чтобы выполнять функции удаленного сервиса, ваш объект должен обладать определенными качествами, которые делают его удаленным. Проще всего этого достичь, расширив класс UnicastRemoteObject (из пакета java.rmi.server), который сделает за вас всю работу.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
```

3 Создаем конструктор без аргументов, объявляя исключение RemoteException.

У вашего нового родительского класса UnicastRemoteObject есть одна небольшая проблема — его конструктор выбрасывает исключение RemoteException. Единственный способ решить эту проблему — создать конструктор для вашей реализации удаленного интерфейса. Только так можно объявить RemoteException. Помните, что при создании экземпляра вызывается и родительский конструктор. Если он выбрасывает исключение, вам не остается ничего другого, кроме как объявить свой конструктор, выбрасывающий это же исключение.

```
public MyRemoteImpl() throws RemoteException { }
```

Не нужно помещать весь код в конструктор. Следует как-то объявить, что конструктор вашего родительского класса выбрасывает исключение.

4 Заносим сервис в реестр RMI.

Теперь, когда вы получили удаленный сервис, требуется сделать его доступным для удаленных клиентов. Создайте экземпляр сервиса и поместите его в реестр RMI (который должен в это время работать, иначе выделенная строка приведет к ошибке). При регистрации сервисного объекта RMI на самом деле помещает в реестр *«заглушку»*, так как именно она нужна клиенту. Весь процесс осуществляется с помощью статического метода rebind() из класса java.rmi.Naming.

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("Remote Hello", service);
} catch(Exception ex) {...}
```

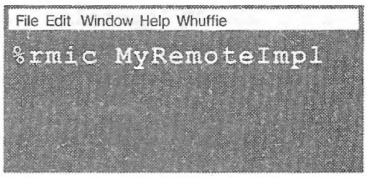
Назовите свой сервис (чтобы клиент мог находить его в реестре по имени) и поместите в реестр RMI. При этом регистрируется не сам сервис, а его «заглушка».

Шаг третий: генерируем «заглушки» и «скелеты»

1. **Запускаем rmic в сочетании с реализацией удаленного интерфейса (но не с самим интерфейсом).**
Утилита rmic, поставляемая вместе с пакетом для разработки на Java, берет реализацию сервиса и создает два новых класса — «заглушку» и «скелет». При этом учитывается соглашение об именовании, согласно которому к названию класса-реализации добавляются суффиксы _Stub и _Skeleton. Утилита поддерживает различные параметры, с помощью которых можно отказаться от генерации «скелета», просмотреть исходный код новых классов и даже определить IIOP в качестве протокола. Мы же используем *rmic* так, как в большинстве случаев это сделаете вы. Классы будут отправлены в текущую директорию (то есть туда, куда вы в последний раз перемещались). Помните, что утилита rmic должна «видеть» ваш класс-реализацию, поэтому лучше запускать ее там, где эта реализация находится.

Чтобы упростить пример, мы намеренно отказались от пакетов. На практике вам придется соблюдать структуру каталогов пакета и использовать полные имена классов.

Заметьте, что в конце не нужно добавлять .class. Указываем только имя класса.



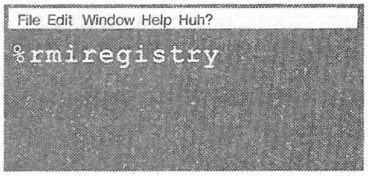
MyRemoteImpl_Stub.class

Генерирует два новых класса для вспомогательных объектов.

MyRemoteImpl_Skel.class

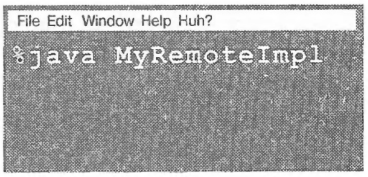
Шаг четвертый: запускаем rmiregistry

1. **Открываем командную строку и запускаем rmiregistry.**
Убедитесь, что вы запустили утилиту из каталога, который имеет доступ к вашим классам. Проще всего это делать из директории classes.

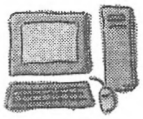


Шаг пятый: запускаем сервис

1. **Открываем еще один терминал и запускаем свой сервис.**
Код, в котором экземпляр сервиса создается, заносится в реестр и запускается, можно разместить как в методе main() внутри реализации удаленного интерфейса, так и в отдельном классе, предназначенном специально для этого. В нашем простом примере выбран первый вариант.



Полная версия серверного кода



Удаленный интерфейс:

```
import java.rmi.*;

public interface MyRemote extends Remote {

    public String sayHello() throws RemoteException;
}
```

Исключение RemoteException и интерфейс находятся в пакете java.rmi.

Интерфейс должен быть унаследован от java.rmi.Remote

Все удаленные методы должны содержать объявление RemoteException.

Удаленный сервис (реализация):

```
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {

   public String sayHello() {
       return "Сервер говорит: 'Привет'";
   }

   public MyRemoteImpl() throws RemoteException { }

   public static void main (String[] args) {

       try {
           MyRemote service = new MyRemoteImpl();
           Naming.rebind("Удаленный привет", service);
       } catch(Exception ex) {
           ex.printStackTrace();
       }
   }
}
```

UnicastRemoteObject находится в пакете java.rmi.server.

Наследование UnicastRemoteObject — самый простой способ создать удаленный объект.

Конечно, необходимо реализовать все методы интерфейса. Но заметьте, что необязательно объявлять RemoteException.

Нужно реализовать свой удаленный интерфейс!

В объявлении конструктора родительского класса UnicastRemoteObject содержится исключение, поэтому нужно создать свой конструктор, ведь его наличие говорит о вызове опасного кода (конструктора его родительского класса).

Создаем удаленный объект, а затем помещаем его в реестр, используя статический метод Naming.rebind(). Указанное имя будет использоваться клиентами для поиска объекта в реестре RMI.

Как клиент получает объект «заглушки»

Клиент должен получить объект-«заглушку», так как именно из него будут вызываться методы. Здесь и пригодится реестр RMI. Клиент осуществляет «поиск», будто берет в руки телефонный справочник и говорит: «Я хотел бы найти „заглушку“ с таким-то именем».

lookup() — статический метод из класса Naming.

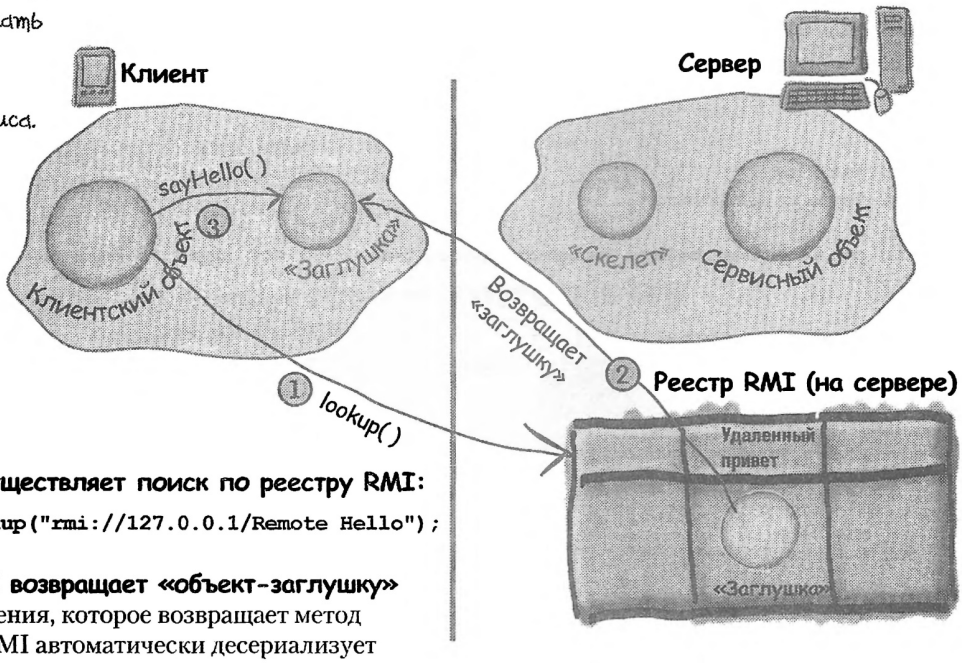
Здесь нужно указать имя, под которым сервис зарегистрирован.

```
MyRemote service = (MyRemote)   Naming.lookup("rmi://127.0.0.1/Remote Hello");
```

Клиент всегда использует удаленную реализацию в качестве типа сервиса. Фактически ему не нужно знать настоящее имя класса вашего удаленного сервиса.

Необходимо привести это значение к типу интерфейса, так как метод lookup возвращает Object.

Здесь находится имя домена или IP-адрес.



1. **Клиент осуществляет поиск по реестру RMI:**

   ```
   Naming.lookup("rmi://127.0.0.1/Remote Hello");
   ```

2. **Реестр RMI возвращает «объект-заглушку»** (в виде значения, которое возвращает метод lookup), и RMI автоматически десериализует «заглушку». Ваш клиент должен иметь класс «заглушки» (сгенерированный с помощью rmic), иначе десериализации не будет.

3. **Клиент вызывает метод из «заглушки», как будто она является настоящим сервисом.**

Как клиент получает класс «заглушки»

Мы подошли к интересному вопросу. Так или иначе, на момент поиска клиент должен иметь класс «заглушки» (который вы сгенерировали с помощью rmic), иначе он не сможет ее десериализовать. Если вы работаете с простой системой, то можете вручную передать этот класс клиенту.

Существует намного более изящное решение, которое выходит за рамки этой книги. Речь идет о динамической загрузке классов, когда объект «заглушки» ассоциируется с URL-адресом. По этому адресу клиентская часть RMI может найти класс объекта. Затем, в процессе десериализации, если система RMI не находит класс локально, она использует вышеупомянутый адрес, чтобы сделать запрос GET по HTTP и получить нужный файл. Итак, чтобы хранить файл с классом, вам понадобится обычный веб-сервер. Кроме того, придется изменить некоторые настройки безопасности на клиенте. Есть еще несколько неочевидных проблем, связанных с динамической загрузкой классов, но мы не станем их разбирать, так как сделали лишь краткий обзор.

Полная версия кода клиента

```
import java.rmi.*;

public class MyRemoteClient {
   public static void main (String[] args) {
        new MyRemoteClient().go();
   }

  public void go() {

     try {
        MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/Remote Hello");

        String s = service.sayHello();

        System.out.println(s);
      } catch(Exception ex) {
        ex.printStackTrace();
      }
   }
}
```

Класс Naming (для поиска по реестру) находится в пакете java.rmi.

Реестр возвращает значение класса Object, поэтому не забудьте привести его к соответствующему типу.

Вам нужен IP-адрес или имя домена.

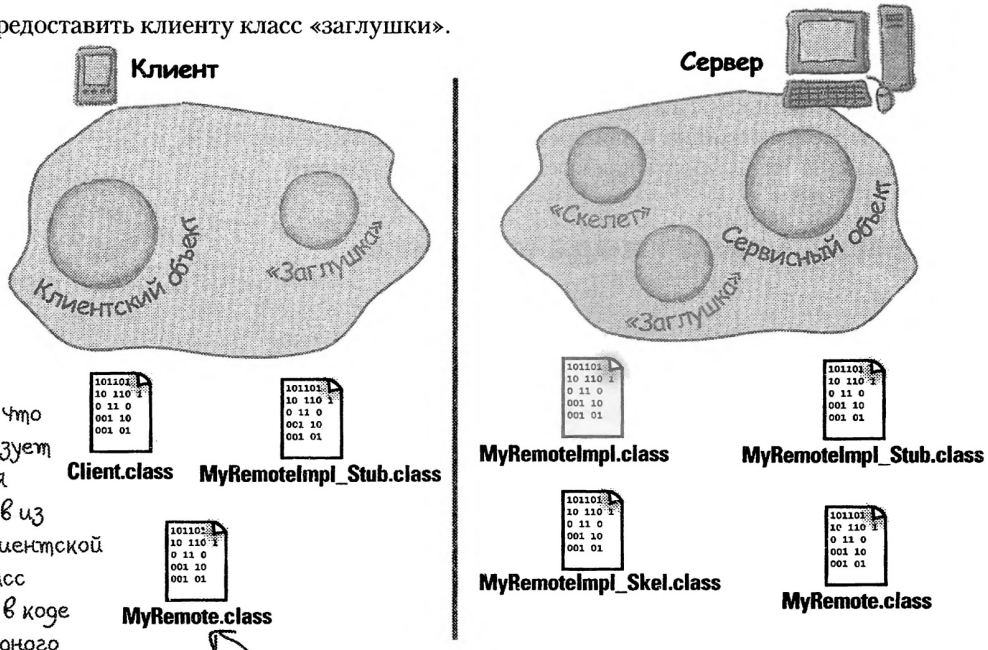
И имя, которое использовалось при регистрации сервиса.

Выглядит как вызов обычного метода (за исключением того, что необходимо учитывать исключение RemoteException)!

Убедитесь, что на всех компьютерах есть нужные class-файлы

Программисты чаще всего допускают следующие три ошибки при работе с RMI.

- Забывают запустить rmiregistry перед запуском удаленного сервиса (когда вы регистрируете сервис с помощью Naming.rebind(), rmiregistry должен работать).
- Забывают сделать аргументы и возвращаемые значения сериализуемыми (ошибка не выявляется на этапе компиляции, то есть вы не узнаете о ней, пока не запустите программу).
- Забывают предоставить клиенту класс «заглушки».



Не забывайте, что клиент использует интерфейс для вызова методов из «заглушки». Клиентской JVM нужен класс «заглушки», но в коде о нем нет ни одного упоминания. Клиент всегда использует удаленный интерфейс, так как именно через него предоставляется доступ к удаленному объекту.

Кроме сервиса и удаленного интерфейса серверу нужны классы для «заглушки» и «скелета». «Заглушка» — это замена настоящего сервиса, которая помещается в реестр RMI.

Наточите свой карандаш

Взгляните на набор событий, приведенных ниже. Разместите их в порядке, в котором они возникают внутри Java-приложения, использующего RMI.

Что было раньше?

1
2
3
4
5
6
7

КЛЮЧЕВЫЕ МОМЕНТЫ

- Объект, находящийся в куче, не может получить обычную для Java ссылку на объект из другой кучи (работающий под управлением другой JVM).
- Технология вызова удаленных методов в Java (RMI) делает возможной работу с удаленным объектом (выполняющимся на другой JVM) так, будто он локальный.
- Вызывая метод из удаленного объекта, клиент на самом деле вызывает его из прокси для этого объекта. Прокси называется «заглушкой».
- «Заглушка» выступает в роли вспомогательного клиентского объекта, который берет на себя всю низкоуровневую сетевую работу (сокеты, потоки, сериализацию и т. д.), упаковывая и отправляя вызовы методов на сервер.
- Чтобы создать удаленный сервис (проще говоря, объект, из которого удаленный клиент может вызывать методы), нужно начать с удаленного интерфейса.
- Необходимо, чтобы удаленный интерфейс был унаследован от пакета java.rmi.Remote, а все его методы объявляли исключение RemoteException.
- Ваш удаленный сервис должен реализовывать ваш удаленный интерфейс.
- Удаленный сервис должен расширять класс UnicastRemoteObject (формально существуют и другие способы создания удаленного объекта, но это самый простой).
- Класс вашего удаленного сервиса должен содержать конструктор, в объявлении которого есть RemoteException (по аналогии с конструктором родительского класса).
- Необходимо создать экземпляр своего удаленного сервиса и поместить его в реестр RMI.
- Для регистрации удаленного сервиса используйте статический метод Naming.rebind("Имя сервиса", serviceInstance);.
- Реестр RMI должен работать на том же компьютере, что и удаленный сервис, и должен быть запущен до того, как вы попытаетесь зарегистрировать в нем удаленный объект.
- Клиент ищет ваш удаленный сервис с помощью статического метода Naming.lookup("rmi://MyHostName/ServiceName").
- Почти все методы, связанные с RMI, могут выбросить исключение RemoteException (оно проверяется компилятором). Это касается регистрации и поиска удаленного сервиса в реестре, а также вызова клиентом всех удаленных методов из «заглушки».

Кто на самом деле использует технологию RMI

СЕРВЛЕТЫ

Удаленное развертывание с помощью RMI

Что насчет сервлетов?

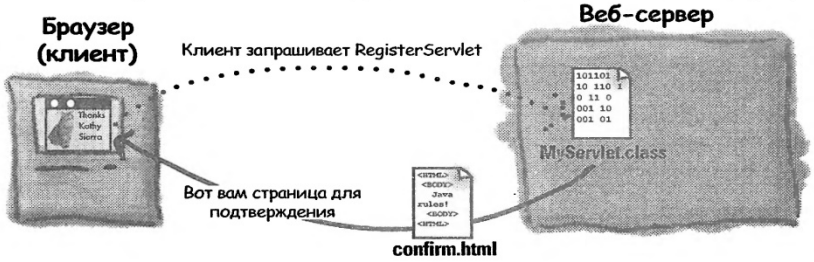
Сервлеты — это приложения на языке Java, которые выполняются на веб-сервере через протокол HTTP. Когда клиент использует браузер для работы с веб-страницей, запросы поступают на сервер. Если запросу нужна помощь сервлета, то сервер запускает (или вызывает, если сервлет уже работает) его код. Этот код выполняет работу в ответ на клиентские запросы (например, сохраняет информацию в текстовый файл или базу данных на сервере). Если вы знакомы с CGI-сценариями, написанными на языке Perl, то должны хорошо понимать, о чем идет речь. Веб-разработчики используют CGI-сценарии или сервлеты для выполнения любых действий, начиная с передачи пользовательской информации в базу данных и заканчивая управлением форумами.

Кроме того, сервлеты могут использовать RMI!

Сегодня технология J2EE чаще всего применяется для совмещения сервлетов (выступающих в роли клиентов) и компонентов EJB (работающих на стороне сервера). В таких случаях *сервлеты применяют RMI для взаимодействия с EJB*, хотя это *немного* отличается от того, что вы могли наблюдать несколькими страницами ранее.

1. Клиент заполняет форму для регистрации и нажимает кнопку Submit (Отправить). HTTP-сервер (то есть веб-сервер) получает запрос и пересылает его сервлету, для которого он и предназначен.

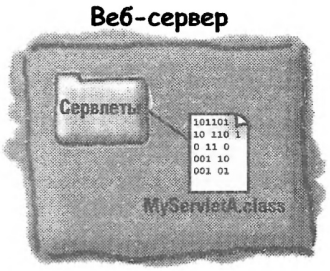
2. Сервлет (код на языке Java) запускается, заносит информацию в базу данных, затем на основе полученных сведений формирует веб-страницу и вновь отправляет ее клиенту, где она отображается в окне браузера.



Этапы создания и запуска сервлета

1 Выясняем, куда должны быть помещены сервлеты.

В примерах подразумевается, что у вас уже есть рабочий веб-сервер, который сконфигурирован для поддержки сервлетов. Самое главное — выяснить, где именно должны находиться class-файлы вашего сервлета, чтобы сервер их «увидел». Если вы размещаете сайт на серверах своего провайдера, то узнайте у него, где должны храниться сервлеты, как узнавали о том, куда нужно поместить CGI-сценарии.



2 Получаем файл servlet.jar и помещаем его по адресу, предусмотренному переменной CLASSPATH.

Сервлеты не относятся к стандартной библиотеке Java. Нужно, чтобы их скомпилированные файлы были упакованы в архив servlets.jar. Вы можете загрузить его с сайта java.sun.com или получить в комплекте с веб-сервером, поддерживающим Java (например, Apache Tomcat, который можно найти по адресу apache.org). Без этих классов вы не сможете скомпилировать свои сервлеты.


servlets.jar

3 Пишем сервлет class, расширяющий HTTPServlet.

Сервлет — это просто класс Java, который расширяет HTTPServlet (из пакета javax.servlet.http). Можно создать другие типы сервлетов, но большую часть времени мы посвятим HTTPServlet.

MyServletA.class

```
public class MyServletA extends HttpServlet { ... }
```

4 Создаем HTML-страницу, которая будет вызывать сервлет.

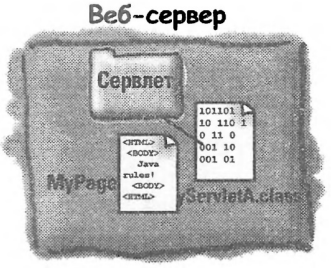
Когда пользователь щелкает на ссылке, которая указывает на ваш сервлет, веб-сервер находит этот сервлет и вызывает соответствующий метод в зависимости от HTTP-команды (GET, POST и т. д.).

MyPage.html

```
<a href="servlets/MyServletA">This is the most amazing servlet.</a>
```

5 Делаем сервлет и HTML-страницу доступными для своего сервера.

Это полностью зависит от того, какой у вас веб-сервер и с какой *версией* Java-сервлетов вы работаете. Провайдер может порекомендовать вам только скопировать все файлы в директорию Servlets на вашем сайте. Но если вы используете, скажем, последнюю версию Tomcat, то придется выполнить гораздо больше действий, чтобы ваш сервлет (наряду с веб-страницей) оказался в нужном месте (так уж получилось, что у нас есть книга и по этой теме).



Очень простой сервлет

Помимо io, нам нужно импортировать два пакета для сервлетов. Помните, что они не входят в состав стандартной библиотеки Java — придется загрузить их отдельно.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

Большинство обычных сервлетов расширяют класс HttpServlet и переопределяют один или несколько его методов.

```
public class MyServletA extends HttpServlet {
```

Переопределяем doGet для обработки простых GET-сообщений по HTTP.

Веб-сервер вызывает этот метод, передавая ему клиентский запрос (вы можете извлечь из него данные) и объект response, который вы будете использовать для возвращения ответа (страницы).

```
    public void doGet (HttpServletRequest request, HttpServletResponse response)
                                        throws  ServletException, IOException  {
```

Этим мы говорим серверу и клиенту, какой тип ответа будет возвращен сервером в качестве результата выполнения сервлета.

```
        response.setContentType("text/html");
```

В переменной response хранится исходящий поток, с помощью которого можно записывать информацию обратно на сервер.

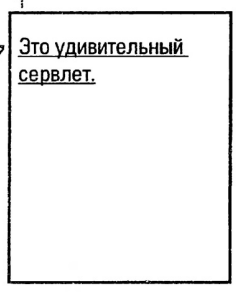
```
        PrintWriter out = response.getWriter();

        String message = "Если вы это читаете, сервлет работает!";

        out.println("<HTML><BODY>");
        out.println("<H1>" + message + "</H1>");
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

Это код, который мы записали для HTML-страницы! Она будет доставлена в браузер через сервер, как и любая другая веб-страница, даже если она не существовала до этого момента. Иными словами, на сервере не существует HTML-файла с такими данными.

HTML-страница со ссылкой на сервлет

Чтобы запустить сервлет, щелкните на ссылке.

Как будет выглядеть страница:




```
<HTML>
  <BODY>
     <a href="servlets/MyServletA">Это удивительный сервлет.</a>
  </BODY>
</HTML>
```

Очень простой сервлет

КЛЮЧЕВЫЕ МОМЕНТЫ

- Сервлеты — это Java-классы, которые полностью выполняются при участии веб-сервера и/или внутри него.
- Сервлеты используются для выполнения серверного кода, который представляет собой результат действий клиента на веб-странице. Например, если клиент отправил информацию через форму, то сервлет может ее обработать и добавить в базу данных, после чего вернуть персонализированную страницу-подтверждение.
- Чтобы скомпилировать сервлет, понадобятся пакеты, хранящиеся в файле servlet.jar. Классы для сервлетов не входят в состав стандартной библиотеки Java, поэтому вы должны загрузить их с сайта java.sun.com или получить вместе с сервером, поддерживающим эту технологию. Примечание: библиотека для сервлетов включена в Java 2 Enterprise Edition (J2EE).
- Для запуска сервлетов нужен совместимый с ними веб-сервер, например Tomcat, который можно найти на сайте apache.org.
- Место, куда вы должны поместить свой сервлет, зависит от конкретного сервера, поэтому лучше узнать о нем заранее. Если обслуживанием вашего сайта занимается ваш провайдер, можете узнать у него, где нужно хранить сервлеты.
- Типичный сервлет расширяет класс HttpServlet, переопределяя один или несколько таких его методов, как doGet() или doPost().
- Веб-сервер запускает сервлет и вызывает из него соответствующий метод (doGet() и т. д.) в зависимости от типа клиентского запроса.
- Сервлет может возвращать ответ с помощью исходящего потока PrintWriter, который хранится в параметре response метода doGet().
- Сервлет «формирует» и возвращает HTML-страницу целиком, с тегами.

Это не глупые вопросы

В: Что такое JSP и как оно связано с сервлетами?

О: JSP расшифровывается как Java Server Pages. Благодаря веб-серверу результаты работы сервлетов и JSP могут ничем не отличаться. Разница между технологиями заключается в том, *что* вы как разработчик при этом создаете. В случае с сервлетом вы пишете *класс*, который формирует *HTML-страницу* с помощью исходящего потока (если вы действительно шлете клиенту страницу). С JSP все наоборот — вы создаете *HTML-страницу*, которая содержит код на языке Java!

JSP позволяет создавать динамические веб-страницы по тому же принципу, что и обычные HTML-файлы, и встраивать в них код на языке Java (а также другие теги, вызывающие Java-код), который исполняется на ходу.

Главное преимущество JSP перед сервлетами заключается в том, что HTML-код легче писать в виде JSP-страницы, а не формировать его внутри сервлета, используя методы println. Представьте себе в меру сложный HTML-документ и подумайте, как бы вы его отформатировали этими методами.

Однако многие приложения не нуждаются в JSP, ведь сервлетам необязательно возвращать динамический ответ, или же HTML-код достаточно прост, чтобы его формирование не доставляло неудобств. К тому же существует множество веб-серверов, поддерживающих сервлеты, но не умеющих работать с JSP.

У JSP есть еще одно преимущество: вы можете разделить работу между Java-программистами, пишущими сервлеты, и верстальщиками, создающими JSP-страницы. По крайней мере в теории. На практике же для написания JSP-страниц (как и HTML-страниц) нужна специальная квалификация, поэтому не стоит надеяться, что верстальщик сможет без проблем выполнять эту работу. Для этого ему понадобится определенный инструментарий. Но есть и хорошие новости — на рынке начинают появляться средства разработки, способные помочь дизайнерам создавать JSP без необходимости писать весь код с нуля.

В: Вам больше нечего сказать о сервлетах? И это после такого подробного обзора RMI?

О: Да. RMI — это часть языка Java, и все его классы находятся в стандартной библиотеке. Сервлеты и JSP рассматриваются как *стандартные расширения*. Вы можете использовать RMI на любой современной JVM, но для работы с JSP и сервлетами понадобится тщательно настроенный веб-сервер с поддержкой «контейнеров». Мы так деликатно намекаем на то, что эти технологии выходят за рамки нашей книги. Но вы можете узнать о них больше, прочитав замечательную книгу «*Head First Servlets and JSP*».

Ради забавы сделаем так, чтобы наш генератор фраз работал в виде сервлета

Несмотря на обещание больше ни слова не говорить о сервлетах, мы не смогли устоять перед искушением сервлетизации (да, у нас богатый словарный запас) генератора фраз из главы 1. В конце концов сервлет — это лишь Java-код, и он может вызывать другой код из других классов, в том числе и из нашего генератора фраз. Вам нужно только скопировать класс PhraseOMatic в одну директорию со сервлетом. Код генератора фраз находится на следующей странице.


```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class KathyServlet extends HttpServlet {
    public void doGet (HttpServletRequest request, HttpServletResponse response)
                                          throws ServletException, IOException  {

        String title = "PhraseOMatic has generated the following phrase.";

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println("PhraseOmatic");
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>" + PhraseOMatic.makePhrase());
        out.println("<P><a href=\"KathyServlet\">Создать другую фразу</a></p>");
        out.println("</BODY></HTML>");

        out.close();
    }
}
```

Видите? Ваш сервлет может вызывать методы из другого класса. Здесь мы вызываем статический метод makePhrase() из класса PhraseOMatic (который находится на следующей странице).

Код генератора фраз, адаптированный для сервлета

Это немного измененная версия кода из первой главы. В оригинале все действия выполнялись в методе main(). Для того чтобы сгенерировать новую фразу, нам каждый раз приходилось запускать программу из командной строки. В этой версии вызов статического метода makePhrase() просто возвращает строку с фразой. Таким образом, можно вызывать этот код из любого участка программы и получать в ответ сгенерированную случайным образом фразу.

Обратите внимание на то, что длинные объявления строковых массивов пострадали при форматировании текста — не перепечатывайте дефисы в конце строк! Просто продолжайте набирать код дальше и позвольте своему редактору автоматически переносить текст. И ни в коем случае не нажимайте Enter посередине строки (внутри двойных кавычек).

```
public class PhraseOMatic {
   public static String makePhrase() {

      // Создайте три набора слов для выбора. Добавляйте свои собственные!
      String[] wordListOne = {"круглосуточный", "трех-звенный", "30000-футовый",
"взаимный", "обоюдный выигрыш", "фронтэнд", "на основе веб-технологий", "проникающий",
"умный", "шесть сигм", "метод критического пути", "динамичный"};

      String[] wordListTwo = {"уполномоченный", "трудный", "чистый продукт",
"ориентированный", "центральный", "распределенный", "кластеризованный", "фирменный",
"нестандартный ум", "позиционированный", "сетевой", "сфокусированный", "использованный
с выгодой", "выровненный", "нацеленный на", "общий", "совместный", "ускоренный"};

      String[] wordListThree = {"процесс", "пункт разгрузки", "выход из положения", "тип
структуры", "талант", "подход", "уровень завоеванного внимания", "портал", "период
времени", "обзор", "образец", "пункт следования"};

    // Вычисляем, сколько слов в каждом списке
    int oneLength = wordListOne.length;
    int twoLength = wordListTwo.length;
    int threeLength = wordListThree.length;

    // Генерируем три случайных числа, чтобы выбрать случайные слова из каждого списка
    int rand1 = (int) (Math.random() * oneLength);
    int rand2 = (int) (Math.random() * twoLength);
    int rand3 = (int) (Math.random() * threeLength);

    // Теперь строим фразу
    String phrase = wordListOne[rand1] + " " + wordListTwo[rand2] + " " +
wordListThree[rand3];

    // Возвращаем ее
    return ("Все, что нам нужно, — это" + phrase);
  }
}
```

Enterprise JavaBeans: RMI на стероидах

RMI отлично подходит для создания и запуска удаленных сервисов. Но для работы таких сайтов, как Amazon или eBay, одного RMI недостаточно. Для крупных и очень серьезных промышленных приложений требуется что-то большее. Вам нужна технология, которая справлялась бы с транзакциями, сложным распараллеливанием (представьте, что несметное количество людей одновременно ринулось на ваш сервер, чтобы купить органический корм для своих домашних любимцев), безопасностью (не всем ведь можно видеть содержимое базы данных с платежными ведомостями вашей компании) и управлением данными. Для этого требуется *сервер приложений промышленного уровня*.

В Java эту роль выполняет сервер, поставляемый вместе с Java 2 Enterprise Edition (J2EE). Он включает в себя как простой веб-сервер, так и сервер для JavaBeans (EJB), чтобы вы могли развертывать приложения, основанные и на сервлетах, и на EJB. Технология EJB, как и сервлеты, абсолютно не вписывается в рамки нашей книги. Невозможно просто показать «небольшой» пример кода для EJB, но мы все же попытаемся сделать ее обзор и объяснить, как она работает.

Для более тесного знакомства с EJB мы рекомендуем учебное пособие для сертификации по EJB от Head First.

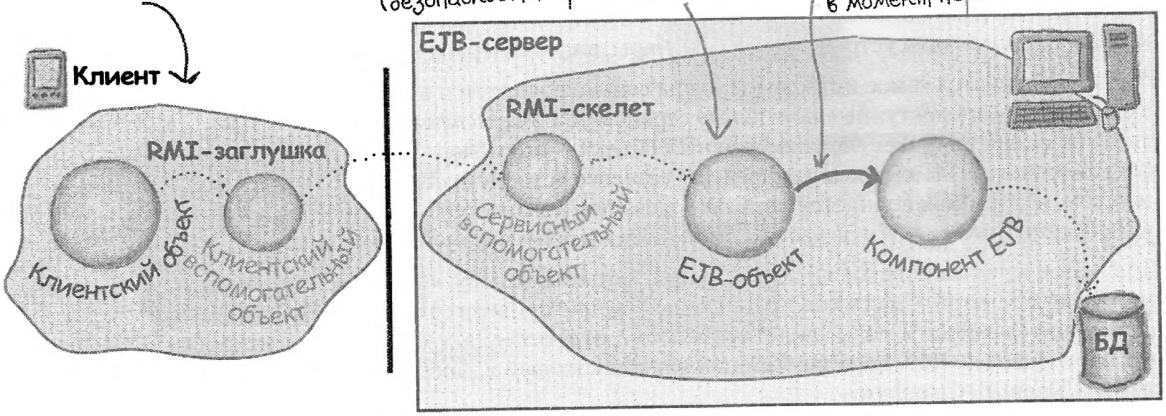
Сервер для EJB поддерживает множество возможностей, которыми не обладает обычный RMI. Это транзакции, безопасность, распараллеливание, управление базами данных и работа с сетью.

Сервер для EJB вмешивается в работу RMI-вызова, подключая все свои сервисы.

Компонент (bean) защищен от прямого доступа со стороны клиента! С ним разрешено взаимодействовать только серверу. Сервер может сказать нечто вроде: «Стоп! Этот клиент не имеет права вызывать данный метод...» Почти все, за что стоит использовать EJB-сервер, происходит прямо здесь, в момент перехвата!

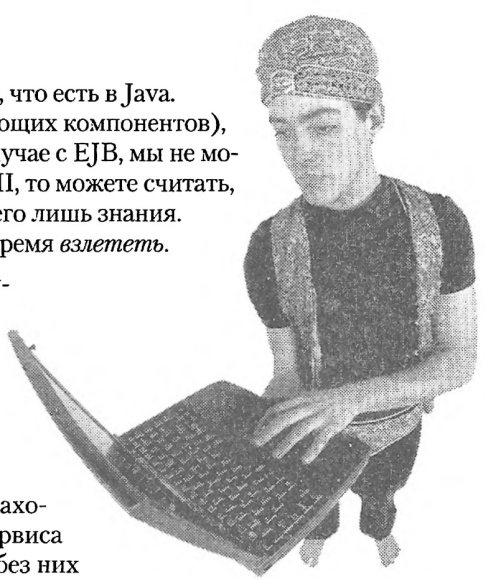
Этот клиент может представлять собой что угодно, но для EJB, как правило, в роли клиента выступает сервлет, работающий на том же J2EE-сервере.

Вот где EJB вступает в игру! Экземпляр EJB перехватывает вызовы к компоненту (он называется bean — «зерно» и описывает бизнес-логику), подключая весь набор сервисов, предоставляемых EJB-сервером (безопасность, транзакции и т. д.)



Это лишь краткое описание работы EJB!

И напоследок немного о Jini



Мы любим Jini. Мы считаем, что Jini — практически лучшее, что есть в Java. Если EJB — это RMI на стероидах (со множеством управляющих компонентов), то Jini — это RMI с *крыльями*. Сущее *блаженство*. Как и в случае с EJB, мы не можем посвятить Jini много страниц, но если вы знакомы с RMI, то можете считать, что 3/4 знаний об этой технологии у вас уже есть. Но это всего лишь знания. Пришло время изменить свой *взгляд на вещи*. Нет, пришло время *взлететь*.

Jini использует RMI (хотя могут быть задействованы и другие протоколы), но обладает несколькими ключевыми возможностями, в числе которых:

адаптивная маршрутизация (Adaptive discovery);

самовосстанавливающиеся сети (Self-healing networks).

В RMI, как вы помните, клиент должен знать имя и местонахождение удаленного сервиса. Клиентский код для поиска сервиса обязан содержать IP-адрес или доменное имя (потому что без них нельзя узнать, где запущен реестр RMI) *и* логическое имя, под которым сервис зарегистрирован.

Но Jini требует, чтобы клиент знал только **интерфейс**, *реализованный сервисом*! И больше ничего.

Как же выполняется поиск? Вся хитрость заключается в наличии у Jini поискового сервиса, который по своей гибкости и мощи намного превосходит реестр RMI. С одной стороны, поисковый сервис Jini *автоматически* объявляет о своем присутствии в сети. При запуске он шлет пользователям сети сообщение (через групповую передачу Multicast), в котором говорится: «Если кому-нибудь интересно, я здесь».

Но это еще не все. Если вы (клиент) появились в сети *после* того, как поисковый сервис объявил о своем присутствии, то можете отправить всем сообщение следующего содержания: «Есть ли где-то поисковые сервисы?»

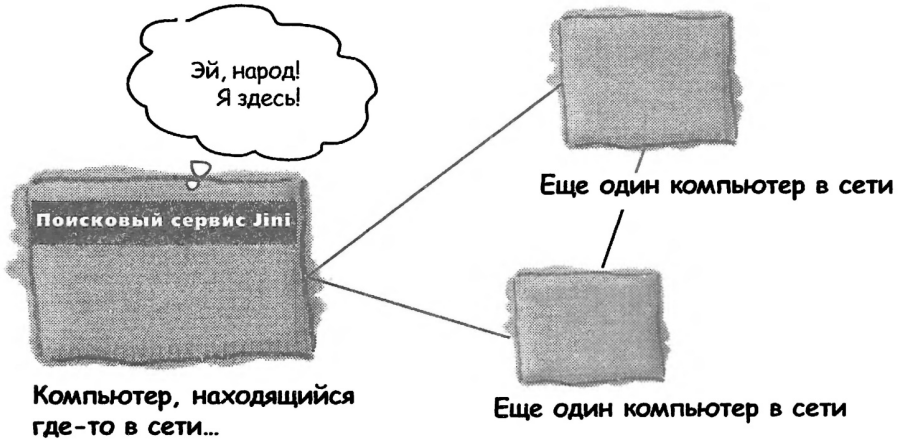
При этом вы заинтересованы не в *самом* поисковом сервисе, а в зарегистрированных в нем сервисах. Это касается удаленных RMI-сервисов, других сериализуемых Java-объектов и даже таких устройств, как принтеры, видеокамеры и кофеварки.

И здесь начинается самое интересное: когда сервис появляется в сети, он автоматически находит все доступные поисковые сервисы Jini и *регистрируется* в них. При регистрации он получает сериализованный объект — это может быть «заглушка» для удаленного RMI-сервиса, драйвер для сетевого устройства или даже весь сервис целиком, который выполняется локально на вашем компьютере. В данном случае для регистрации применяется не *имя*, а *интерфейс*, который реализуется сервисом.

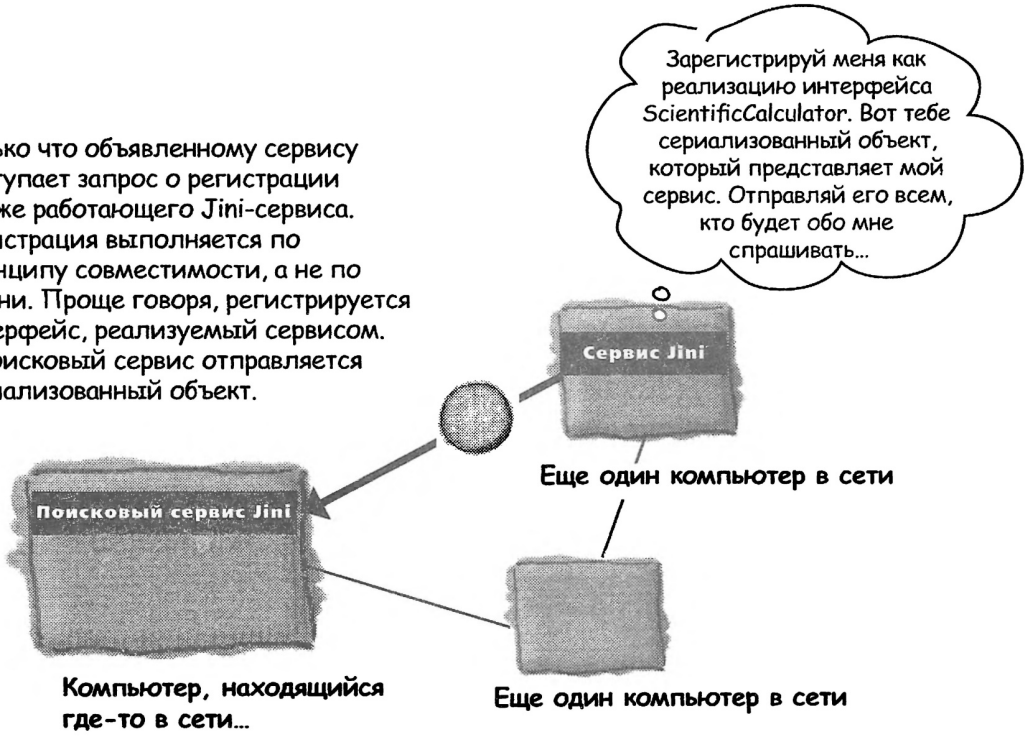
Получив ссылку на поисковый сервис, вы можете сказать ему: «Привет. У тебя есть реализация ScientificCalculator?» Поисковый сервис проверит список зарегистрированных интерфейсов и, если найдет совпадение, ответит: «Да, у меня есть реализация этого интерфейса. Вот сериализованный объект, который я зарегистрировал для сервиса ScientificCalculator».

Адаптивная маршрутизация в действии

1. Поисковый сервис Jini запускается и выходит в сеть, объявляя об этом в широковещательных сообщениях.

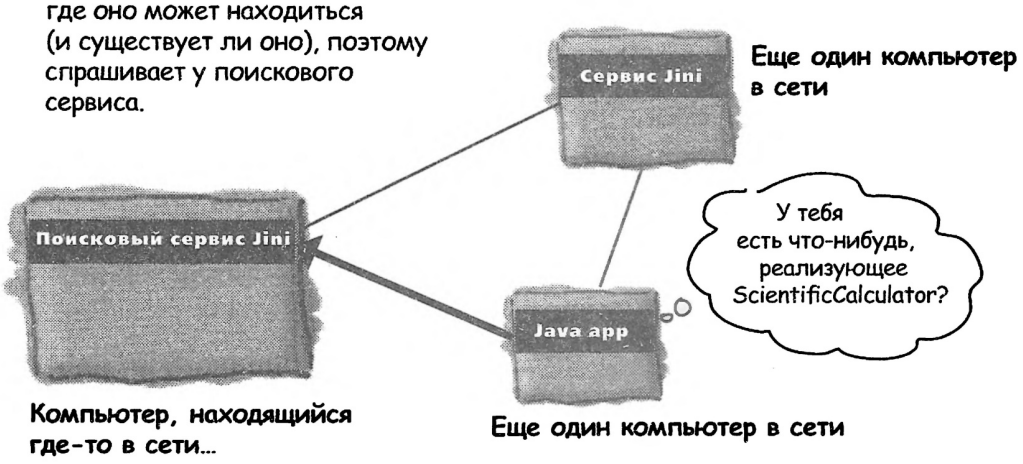


2. Только что объявленному сервису поступает запрос о регистрации от уже работающего Jini-сервиса. Регистрация выполняется по принципу совместимости, а не по имени. Проще говоря, регистрируется интерфейс, реализуемый сервисом. В поисковый сервис отправляется сериализованный объект.

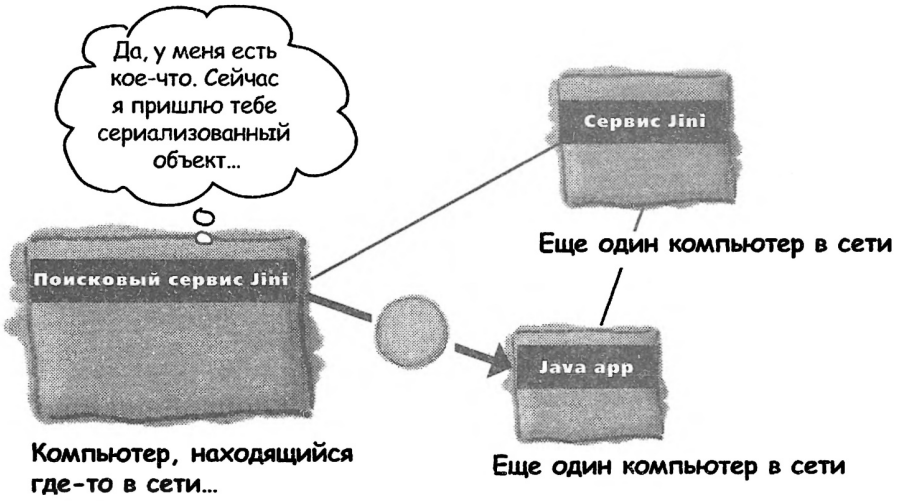


Продолжаем рассматривать адаптивную маршрутизацию

3. Клиенту в сети нужно что-то, реализующее интерфейс ScientificCalculator. Он не знает, где оно может находиться (и существует ли оно), поэтому спрашивает у поискового сервиса.

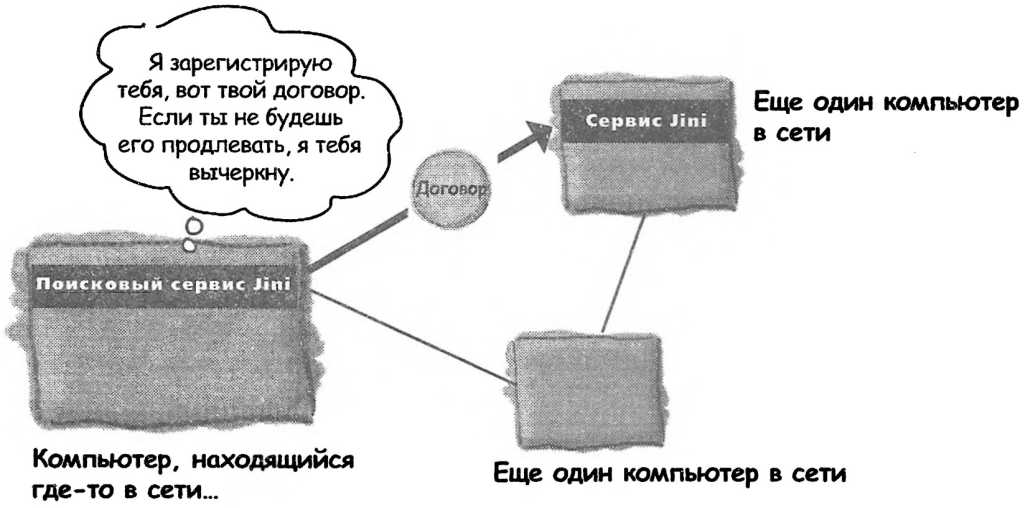


4. Поисковый сервис отвечает утвердительно, если у него зарегистрирована реализация интерфейса ScientificCalculator.

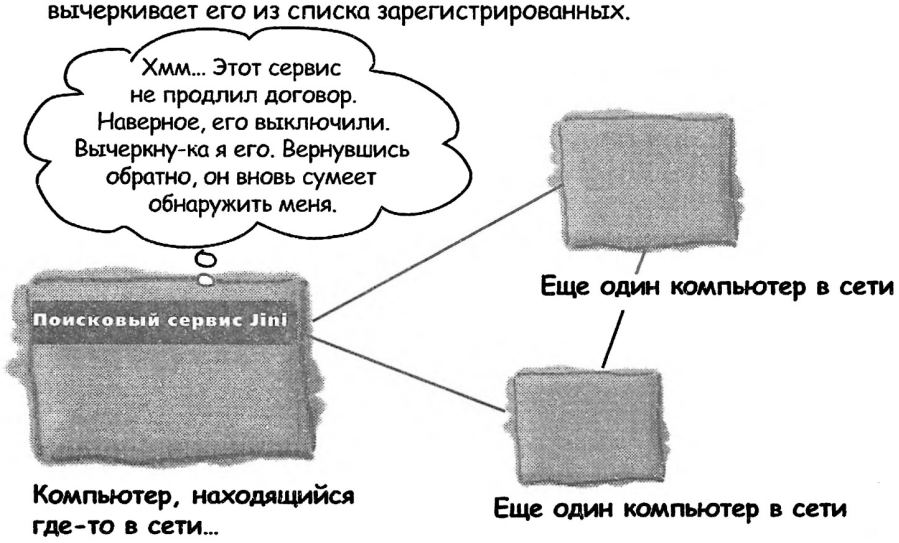


Самовосстанавливающаяся сеть в действии

1. Сервис Jini пытается зарегистрироваться в поисковом сервисе, на что тот отвечает: «Договорились». Только что зарегистрированный сервис должен время от времени продлевать этот «договор», иначе поисковый сервис решит, что тот покинул сеть. Поисковый сервис всегда хочет предоставлять участникам сети актуальную информацию о доступных сервисах.

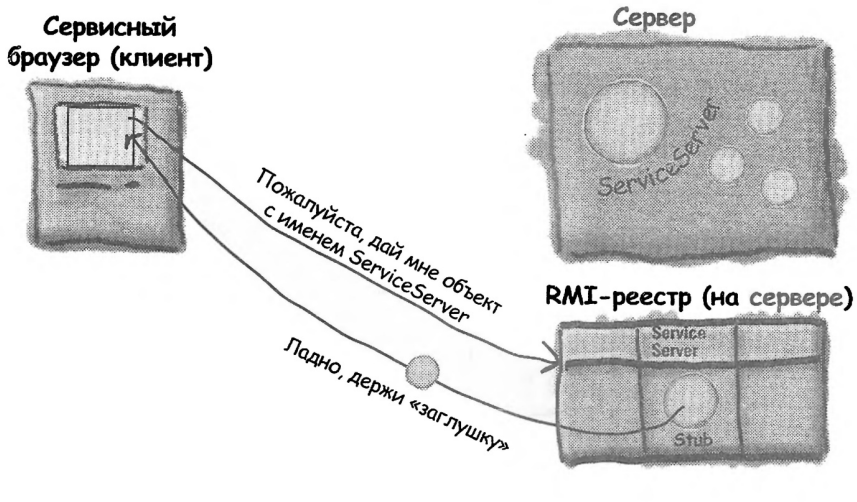


2. Сервис покидает сеть (кто-то его выключил), поэтому не может продлить договор с поисковым сервисом. Поисковый сервис вычеркивает его из списка зарегистрированных.

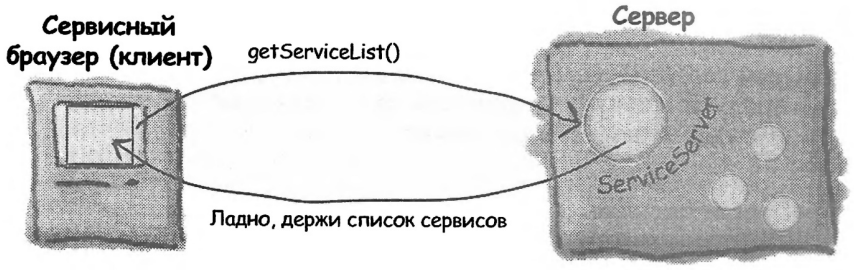


Как это работает:

1. Клиент запускается и ищет сервис ServiceServer в реестре RMI, получая в ответ «заглушку».



2. Клиент вызывает из «заглушки» метод getServiceList(). ServiceServer возвращает список сервисов.



3. Клиент отображает список сирвисов в GUI.