INDEX

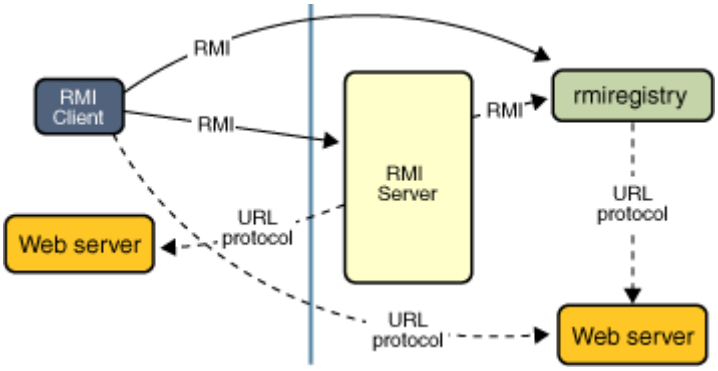
1 An Overview of RMI Applications

RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*.

Distributed object applications need to do the following:

- **Locate remote objects.** Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.
- **Communicate with remote objects.** Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- **Load class definitions for objects that are passed around.** Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.



Advantages of Dynamic Code Loading

One of the central and unique features of RMI is its ability to download the definition of an object's class if the class is not defined in the receiver's Java virtual machine. All of the types and behavior of an object, previously available only in a single Java virtual machine, can be

transmitted to another, possibly remote, Java virtual machine. RMI passes objects by their actual classes, so the behavior of the objects is not changed when they are sent to another Java virtual machine. This capability enables new types and behaviors to be introduced into a remote Java virtual machine, thus dynamically extending the behavior of an application. The compute engine example in this trail uses this capability to introduce new behavior to a distributed program.

Remote Interfaces, Objects, and Methods

Like any other Java application, a distributed application built by using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others. Objects with methods that can be invoked across Java virtual machines are called *remote objects*.

An object becomes remote by implementing a *remote interface*, which has the following characteristics:

- A remote interface extends the interface `java.rmi.Remote`.
- Each method of the interface declares `java.rmi.RemoteException` in its `throws` clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a non-remote object when the object is passed from one Java virtual machine to another Java virtual machine. Rather than making a copy of the implementation object in the receiving Java virtual machine, RMI passes a remote *stub* for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference. The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This property enables a stub to be cast to any of the interfaces that the remote object implements. However, *only* those methods defined in a remote interface are available to be called from the receiving Java virtual machine.

Creating Distributed Applications by Using RMI

Using RMI to develop a distributed application involves these general steps:

1. Designing and implementing the components of your distributed application.
2. Compiling sources.
3. Making classes network accessible.
4. Starting the application.

Designing and Implementing the Application Components

First, determine your application architecture, including which components are local objects and which components are remotely accessible. This step includes:

- **Defining the remote interfaces.** A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. If any of these interfaces or classes do not yet exist, you need to define them as well.
- **Implementing the remote objects.** Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.
- **Implementing the clients.** Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

Compiling Sources

As with any Java program, you use the `javac` compiler to compile the source files. The source files contain the declarations of the remote interfaces, their implementations, any other server classes, and the client classes.

Note: With versions prior to Java Platform, Standard Edition 5.0, an additional step was required to build stub classes, by using the `rmic` compiler. However, this step is no longer necessary.

Making Classes Network Accessible

In this step, you make certain class definitions network accessible, such as the definitions for the remote interfaces and their associated types, and the definitions for classes that need to be downloaded to the clients or servers. Classes definitions are typically made network accessible through a web server.

Starting the Application

Starting the application includes running the RMI remote object registry, the server, and the client.

The rest of this section walks through the steps used to create a compute engine.

Building a Generic Compute Engine

This trail focuses on a simple, yet powerful, distributed application called a *compute engine*. The compute engine is a remote object on the server that takes tasks from clients, runs the tasks, and returns any results. The tasks are run on the machine where the server is running. This type of distributed application can enable a number of client machines to make use of a particularly powerful machine or a machine that has specialized hardware.

The novel aspect of the compute engine is that the tasks it runs do not need to be defined when the compute engine is written or started. New kinds of tasks can be created at any time and then given to the compute engine to be run. The only requirement of a task is that its class implement a particular interface. The code needed to accomplish the task can be downloaded by the RMI system to the compute engine. Then, the compute engine runs the task, using the resources on the machine on which the compute engine is running.

The ability to perform arbitrary tasks is enabled by the dynamic nature of the Java platform, which is extended to the network by RMI. RMI dynamically loads the task code into the compute engine's Java virtual machine and runs the task without prior knowledge of the class that implements the task. Such an application, which has the ability to download code dynamically, is often called a *behavior-based application*. Such applications usually require full agent-enabled infrastructures. With RMI, such applications are part of the basic mechanisms for distributed computing on the Java platform.

2 Writing an RMI Server

The compute engine server accepts tasks from clients, runs the tasks, and returns any results. The server code consists of an interface and a class. The interface defines the methods that can be invoked from the client. Essentially, the interface defines the client's view of the remote object. The class provides the implementation.

Designing a Remote Interface

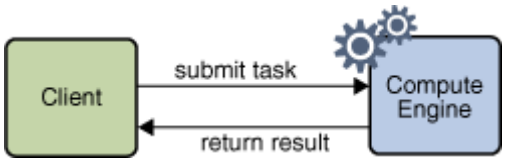
This section explains the `Compute` interface, which provides the connection between the client and the server. You will also learn about the RMI API, which supports this communication.

Implementing a Remote Interface

This section explores the class that implements the `Compute` interface, thereby implementing a remote object. This class also provides the rest of the code that makes up the server program, including a `main` method that creates an instance of the remote object, registers it with the RMI registry, and sets up a security manager.

Designing a Remote Interface

At the core of the compute engine is a protocol that enables tasks to be submitted to the compute engine, the compute engine to run those tasks, and the results of those tasks to be returned to the client. This protocol is expressed in the interfaces that are supported by the compute engine. The remote communication for this protocol is illustrated in the following figure.



Each interface contains a single method. The compute engine's remote interface, `Compute`, enables tasks to be submitted to the engine. The client interface, `Task`, defines how the compute engine executes a submitted task.

The `compute.Compute` interface defines the remotely accessible part, the compute engine itself. Here is the source code for the `Compute` interface:

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

By extending the interface `java.rmi.Remote`, the `Compute` interface identifies itself as an interface whose methods can be invoked from another Java virtual machine. Any object that implements this interface can be a remote object.

As a member of a remote interface, the `executeTask` method is a remote method. Therefore, this method must be defined as being capable of throwing a `java.rmi.RemoteException`. This exception is thrown by the RMI system from a remote method invocation to indicate that either a communication failure or a protocol error has occurred. A `RemoteException` is a checked exception, so any code invoking a remote method needs to handle this exception by either catching it or declaring it in its `throws` clause.

The second interface needed for the compute engine is the `Task` interface, which is the type of the parameter to the `executeTask` method in the `Compute` interface. The `compute.Task` interface defines the interface between the compute engine and the work that it needs to do, providing the way to start the work. Here is the source code for the `Task` interface:

```
package compute;

public interface Task<T> {
    T execute();
}
```

The `Task` interface defines a single method, `execute`, which has no parameters and throws no exceptions. Because the interface does not extend `Remote`, the method in this interface doesn't need to list `java.rmi.RemoteException` in its `throws` clause.

The `Task` interface has a type parameter, `T`, which represents the result type of the task's computation. This interface's `execute` method returns the result of the computation and thus its return type is `T`.

The `Compute` interface's `executeTask` method, in turn, returns the result of the execution of the `Task` instance passed to it. Thus, the `executeTask` method has its own type parameter, `T`, that associates its own return type with the result type of the passed `Task` instance.

RMI uses the Java object serialization mechanism to transport objects by value between Java virtual machines. For an object to be considered serializable, its class must implement the `java.io.Serializable` marker interface. Therefore, classes that implement the `Task` interface must also implement `Serializable`, as must the classes of objects used for task results.

Different kinds of tasks can be run by a `Compute` object as long as they are implementations of the `Task` type. The classes that implement this interface can contain any data needed for the computation of the task and any other methods needed for the computation.

Here is how RMI makes this simple compute engine possible. Because RMI can assume that the `Task` objects are written in the Java programming language, implementations of the `Task` object that were previously unknown to the compute engine are downloaded by RMI into the compute engine's Java virtual machine as needed. This capability enables clients of the

compute engine to define new kinds of tasks to be run on the server machine without needing the code to be explicitly installed on that machine.

The compute engine, implemented by the `ComputeEngine` class, implements the `Compute` interface, enabling different tasks to be submitted to it by calls to its `executeTask` method. These tasks are run using the task's implementation of the `execute` method and the results, are returned to the remote client.

Implementing a Remote Interface

This section discusses the task of implementing a class for the compute engine. In general, a class that implements a remote interface should at least do the following:

- Declare the remote interfaces being implemented
- Define the constructor for each remote object
- Provide an implementation for each remote method in the remote interfaces

An RMI server program needs to create the initial remote objects and *export* them to the RMI runtime, which makes them available to receive incoming remote invocations. This setup procedure can be either encapsulated in a method of the remote object implementation class itself or included in another class entirely. The setup procedure should do the following:

- Create and install a security manager
- Create and export one or more remote objects
- Register at least one remote object with the RMI registry (or with another naming service, such as a service accessible through the Java Naming and Directory Interface) for bootstrapping purposes

The complete implementation of the compute engine follows. The `engine.ComputeEngine` class implements the remote interface `Compute` and also includes the `main` method for setting up the compute engine. Here is the source code for the `ComputeEngine` class:

```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

    public static void main(String[] args) {
```

```
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
                (Compute) UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception:");
            e.printStackTrace();
        }
    }
}
```

The following sections discuss each component of the compute engine implementation.

Declaring the Remote Interfaces Being Implemented

The implementation class for the compute engine is declared as follows:

```
public class ComputeEngine implements Compute
```

This declaration states that the class implements the `Compute` remote interface and therefore can be used for a remote object.

The `ComputeEngine` class defines a remote object implementation class that implements a single remote interface and no other interfaces. The `ComputeEngine` class also contains two executable program elements that can only be invoked locally. The first of these elements is a constructor for `ComputeEngine` instances. The second of these elements is a `main` method that is used to create a `ComputeEngine` instance and make it available to clients.

Defining the Constructor for the Remote Object

The `ComputeEngine` class has a single constructor that takes no arguments. The code for the constructor is as follows:

```
public ComputeEngine() {
    super();
}
```

This constructor just invokes the superclass constructor, which is the no-argument constructor of the `Object` class. Although the superclass constructor gets invoked even if omitted from the `ComputeEngine` constructor, it is included for clarity.

Providing Implementations for Each Remote Method

The class for a remote object provides implementations for each remote method specified in the remote interfaces. The `Compute` interface contains a single remote method, `executeTask`, which is implemented as follows:

```
public <T> T executeTask(Task<T> t) {
    return t.execute();
}
```

This method implements the protocol between the `ComputeEngine` remote object and its clients. Each client provides the `ComputeEngine` with a `Task` object that has a particular implementation of the `Task` interface's `execute` method. The `ComputeEngine` executes each client's task and returns the result of the task's `execute` method directly to the client.

Passing Objects in RMI

Arguments to or return values from remote methods can be of almost any type, including local objects, remote objects, and primitive data types. More precisely, any entity of any type can be passed to or from a remote method as long as the entity is an instance of a type that is a primitive data type, a remote object, or a *serializable* object, which means that it implements the interface `java.io.Serializable`.

Some object types do not meet any of these criteria and thus cannot be passed to or returned from a remote method. Most of these objects, such as threads or file descriptors, encapsulate information that makes sense only within a single address space. Many of the core classes, including the classes in the packages `java.lang` and `java.util`, implement the `Serializable` interface.

The rules governing how arguments and return values are passed are as follows:

- Remote objects are essentially passed by reference. A *remote object reference* is a stub, which is a client-side proxy that implements the complete set of remote interfaces that the remote object implements.
- Local objects are passed by copy, using object serialization. By default, all fields are copied except fields that are marked `static` or `transient`. Default serialization behavior can be overridden on a class-by-class basis.

Passing a remote object by reference means that any changes made to the state of the object by remote method invocations are reflected in the original remote object. When a remote object is passed, only those interfaces that are remote interfaces are available to the receiver. Any methods defined in the implementation class or defined in non-remote interfaces implemented by the class are not available to that receiver.

For example, if you were to pass a reference to an instance of the `ComputeEngine` class, the receiver would have access only to the compute engine's `executeTask` method. That receiver would not see the `ComputeEngine` constructor, its `main` method, or its implementation of any methods of `java.lang.Object`.

In the parameters and return values of remote method invocations, objects that are not remote objects are passed by value. Thus, a copy of the object is created in the receiving Java virtual machine. Any changes to the object's state by the receiver are reflected only in the receiver's copy, not in the sender's original instance. Any changes to the object's state by the sender are reflected only in the sender's original instance, not in the receiver's copy.

Implementing the Server's `main` Method

The most complex method of the `ComputeEngine` implementation is the `main` method. The `main` method is used to start the `ComputeEngine` and therefore needs to do the necessary initialization and housekeeping to prepare the server to accept calls from clients. This method is not a remote method, which means that it cannot be invoked from a different Java virtual machine. Because the `main` method is declared `static`, the method is not associated with an object at all but rather with the class `ComputeEngine`.

Creating and Installing a Security Manager

The `main` method's first task is to create and install a security manager, which protects access to system resources from untrusted downloaded code running within the Java virtual machine. A security manager determines whether downloaded code has access to the local file system or can perform any other privileged operations.

If an RMI program does not install a security manager, RMI will not download classes (other than from the local class path) for objects received as arguments or return values of remote method invocations. This restriction ensures that the operations performed by downloaded code are subject to a security policy.

Here's the code that creates and installs a security manager:

```
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
}
```

Making the Remote Object Available to Clients

Next, the `main` method creates an instance of `ComputeEngine` and exports it to the RMI runtime with the following statements:

```
Compute engine = new ComputeEngine();
Compute stub =
    (Compute) UnicastRemoteObject.exportObject(engine, 0);
```

The static `UnicastRemoteObject.exportObject` method exports the supplied remote object so that it can receive invocations of its remote methods from remote clients. The second argument, an `int`, specifies which TCP port to use to listen for incoming remote invocation requests for the object. It is common to use the value zero, which specifies the use of an anonymous port. The actual port will then be chosen at runtime by RMI or the underlying operating system. However, a non-zero value can also be used to specify a specific port to use for listening. Once the `exportObject` invocation has returned successfully, the `ComputeEngine` remote object is ready to process incoming remote invocations.

The `exportObject` method returns a stub for the exported remote object. Note that the type of the variable `stub` must be `Compute`, not `ComputeEngine`, because the stub for a remote object only implements the remote interfaces that the exported remote object implements.

The `exportObject` method declares that it can throw a `RemoteException`, which is a checked exception type. The `main` method handles this exception with its `try`/`catch` block. If the exception were not handled in this way, `RemoteException` would have to be declared in the `throws` clause of the `main` method. An attempt to export a remote object can throw a `RemoteException` if the necessary communication resources are not available, such as if the requested port is bound for some other purpose.

Before a client can invoke a method on a remote object, it must first obtain a reference to the remote object. Obtaining a reference can be done in the same way that any other object reference is obtained in a program, such as by getting the reference as part of the return value of a method or as part of a data structure that contains such a reference.

The system provides a particular type of remote object, the RMI registry, for finding references to other remote objects. The RMI registry is a simple remote object naming service that enables clients to obtain a reference to a remote object by name. The registry is typically only used to locate the first remote object that an RMI client needs to use. That first remote object might then provide support for finding other objects.

The `java.rmi.registry.Registry` remote interface is the API for binding (or registering) and looking up remote objects in the registry. The `java.rmi.registry.LocateRegistry` class provides static methods for synthesizing a remote reference to a registry at a particular network address (host and port). These methods create the remote reference object containing the specified network address without performing any remote communication. `LocateRegistry` also provides static methods for creating a new registry in the current Java virtual machine, although this example does not use those methods. Once a remote object is registered with an RMI registry on the local host, clients on any host can look up the remote object by name, obtain its reference, and then invoke remote methods on the object. The registry can be shared by all servers running on a host, or an individual server process can create and use its own registry.

The `ComputeEngine` class creates a name for the object with the following statement:

```
String name = "Compute";
```

The code then adds the name to the RMI registry running on the server. This step is done later with the following statements:

```
Registry registry = LocateRegistry.getRegistry();
registry.rebind(name, stub);
```

This `rebind` invocation makes a remote call to the RMI registry on the local host. Like any remote call, this call can result in a `RemoteException` being thrown, which is handled by the `catch` block at the end of the `main` method.

Note the following about the `Registry.rebind` invocation:

- The no-argument overload of `LocateRegistry.getRegistry` synthesizes a reference to a registry on the local host and on the default registry port, 1099.

You must use an overload that has an `int` parameter if the registry is created on a port other than 1099.

- When a remote invocation on the registry is made, a stub for the remote object is passed instead of a copy of the remote object itself. Remote implementation objects, such as instances of `ComputeEngine`, never leave the Java virtual machine in which they were created. Thus, when a client performs a lookup in a server's remote object registry, a copy of the stub is returned. Remote objects in such cases are thus effectively passed by (remote) reference rather than by value.
- For security reasons, an application can only `bind`, `unbind`, or `rebind` remote object references with a registry running on the same host. This restriction prevents a remote client from removing or overwriting any of the entries in a server's registry. A `lookup`, however, can be requested from any host, local or remote.

Once the server has registered with the local RMI registry, it prints a message indicating that it is ready to start handling calls. Then, the `main` method completes. It is not necessary to have a thread wait to keep the server alive. As long as there is a reference to the `ComputeEngine` object in another Java virtual machine, local or remote, the `ComputeEngine` object will not be shut down or garbage collected. Because the program binds a reference to the `ComputeEngine` in the registry, it is reachable from a remote client, the registry itself. The RMI system keeps the `ComputeEngine`'s process running. The `ComputeEngine` is available to accept calls and won't be reclaimed until its binding is removed from the registry *and* no remote clients hold a remote reference to the `ComputeEngine` object.

The final piece of code in the `ComputeEngine.main` method handles any exception that might arise. The only checked exception type that could be thrown in the code is `RemoteException`, either by the `UnicastRemoteObject.exportObject` invocation or by the registry `rebind` invocation. In either case, the program cannot do much more than exit after printing an error message. In some distributed applications, recovering from the failure to make a remote invocation is possible. For example, the application could attempt to retry the operation or choose another server to continue the operation.

3 Creating a Client Program

The compute engine is a relatively simple program: it runs tasks that are handed to it. The clients for the compute engine are more complex. A client needs to call the compute engine, but it also has to define the task to be performed by the compute engine.

Two separate classes make up the client in our example. The first class, `ComputePi`, looks up and invokes a `Compute` object. The second class, `Pi`, implements the `Task` interface and defines the work to be done by the compute engine. The job of the `Pi` class is to compute the value of π to some number of decimal places.

The non-remote `Task` interface is defined as follows:

```
package compute;

public interface Task<T> {
    T execute();
}
```

The code that invokes a `Compute` object's methods must obtain a reference to that object, create a `Task` object, and then request that the task be executed. The definition of the task class `Pi` is shown later. A `Pi` object is constructed with a single argument, the desired precision of the result. The result of the task execution is a `java.math.BigDecimal` representing π calculated to the specified precision.

Here is the source code for `client.ComputePi`, the main client class:

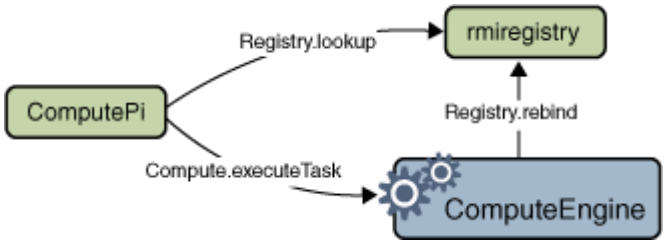
```
package client;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

Like the `ComputeEngine` server, the client begins by installing a security manager. This step is necessary because the process of receiving the server remote object's stub could require downloading class definitions from the server. For RMI to download classes, a security manager must be in force.

After installing a security manager, the client constructs a name to use to look up a `Compute` remote object, using the same name used by `ComputeEngine` to bind its remote object. Also, the client uses the `LocateRegistry.getRegistry` API to synthesize a remote reference to the registry on the server's host. The value of the first command-line argument, `args[0]`, is the name of the remote host on which the `Compute` object runs. The client then invokes the `lookup` method on the registry to look up the remote object by name in the server host's registry. The particular overload of `LocateRegistry.getRegistry` used, which has a single `String` parameter, returns a reference to a registry at the named host and the default registry port, 1099. You must use an overload that has an `int` parameter if the registry is created on a port other than 1099.

Next, the client creates a new `Pi` object, passing to the `Pi` constructor the value of the second command-line argument, `args[1]`, parsed as an integer. This argument indicates the number of decimal places to use in the calculation. Finally, the client invokes the `executeTask` method of the `Compute` remote object. The object passed into the `executeTask` invocation returns an object of type `BigDecimal`, which the program stores in the variable `result`. Finally, the program prints the result. The following figure depicts the flow of messages among the `ComputePi` client, the `rmiregistry`, and the `ComputeEngine`.



The `Pi` class implements the `Task` interface and computes the value of π to a specified number of decimal places. For this example, the actual algorithm is unimportant. What is important is that the algorithm is computationally expensive, meaning that you would want to have it executed on a capable server.

Here is the source code for `client.Pi`, the class that implements the `Task` interface:

```
package client;

import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;

public class Pi implements Task<BigDecimal>, Serializable {

    private static final long serialVersionUID = 227L;

    /** constants used in pi computation */
    private static final BigDecimal FOUR =
```

```
    BigDecimal.valueOf(4);

/** rounding mode to use during pi computation */
private static final int roundingMode =
    BigDecimal.ROUND_HALF_EVEN;

/** digits of precision after the decimal point */
private final int digits;

/**
 * Construct a task to calculate pi to the specified
 * precision.
 */
public Pi(int digits) {
    this.digits = digits;
}

/**
 * Calculate pi.
 */
public BigDecimal execute() {
    return computePi(digits);
}

/**
 * Compute the value of pi to the specified number of
 * digits after the decimal point.  The value is
 * computed using Machin's formula:
 *
 *          pi/4 = 4*arctan(1/5) - arctan(1/239)
 *
 * and a power series expansion of arctan(x) to
 * sufficient precision.
 */
public static BigDecimal computePi(int digits) {
    int scale = digits + 5;
    BigDecimal arctan1_5 = arctan(5, scale);
    BigDecimal arctan1_239 = arctan(239, scale);
    BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
                               arctan1_239).multiply(FOUR);
    return pi.setScale(digits,
                       BigDecimal.ROUND_HALF_UP);
}
/**
 * Compute the value, in radians, of the arctangent of
 * the inverse of the supplied integer to the specified
 * number of digits after the decimal point.  The value
 * is computed using the power series expansion for the
 * arc tangent:
 *
 * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 +
 *     (x^9)/9 ...
 */
public static BigDecimal arctan(int inverseX,
                                int scale)
{
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf(inverseX);
    BigDecimal invX2 =
        BigDecimal.valueOf(inverseX * inverseX);
```

```
        numer = BigDecimal.ONE.divide(invX,
                                      scale, roundingMode);

        result = numer;
        int i = 1;
        do {
            numer =
                numer.divide(invX2, scale, roundingMode);
            int denom = 2 * i + 1;
            term =
                numer.divide(BigDecimal.valueOf(denom),
                             scale, roundingMode);
            if ((i % 2) != 0) {
                result = result.subtract(term);
            } else {
                result = result.add(term);
            }
            i++;
        } while (term.compareTo(BigDecimal.ZERO) != 0);
        return result;
    }
}
```

Note that all serializable classes, whether they implement the `Serializable` interface directly or indirectly, must declare a `private static final` field named `serialVersionUID` to guarantee serialization compatibility between versions. If no previous version of the class has been released, then the value of this field can be any `long` value, similar to the `227L` used by `Pi`, as long as the value is used consistently in future versions. If a previous version of the class has been released without an explicit `serialVersionUID` declaration, but serialization compatibility with that version is important, then the default implicitly computed value for the previous version must be used for the value of the new version's explicit declaration. The `serialver` tool can be run against the previous version to determine the default computed value for it.

The most interesting feature of this example is that the `Compute` implementation object never needs the `Pi` class's definition until a `Pi` object is passed in as an argument to the `executeTask` method. At that point, the code for the class is loaded by RMI into the `Compute` object's Java virtual machine, the `execute` method is invoked, and the task's code is executed. The result, which in the case of the `Pi` task is a `BigDecimal` object, is handed back to the calling client, where it is used to print the result of the computation.

The fact that the supplied `Task` object computes the value of `Pi` is irrelevant to the `ComputeEngine` object. You could also implement a task that, for example, generates a random prime number by using a probabilistic algorithm. That task would also be computationally intensive and therefore a good candidate for passing to the `ComputeEngine`, but it would require very different code. This code could also be downloaded when the `Task` object is passed to a `Compute` object. In just the way that the algorithm for computing π is brought in when needed, the code that generates the random prime number would be brought in when needed. The `Compute` object knows only that each object it receives implements the `execute` method. The `Compute` object does not know, and does not need to know, what the implementation does.

4 Compiling and Running the Program

Now that the code for the compute engine example has been written, it needs to be compiled and run.

Compiling the Programs

In this section, you learn how to compile the server and the client programs that make up the compute engine example.

Running the Programs

Finally, you run the server and client programs and consequently compute the value of π.

Compiling the Programs

In a real-world scenario in which a service such as the compute engine is deployed, a developer would likely create a Java Archive (JAR) file that contains the `Compute` and `Task` interfaces for server classes to implement and client programs to use. Next, a developer, perhaps the same developer of the interface JAR file, would write an implementation of the `Compute` interface and deploy that service on a machine available to clients. Developers of client programs can use the `Compute` and the `Task` interfaces, contained in the JAR file, and independently develop a task and client program that uses a `Compute` service.

In this section, you learn how to set up the JAR file, server classes, and client classes. You will see that the client's `Pi` class will be downloaded to the server at runtime. Also, the `Compute` and `Task` interfaces will be downloaded from the server to the registry at runtime.

This example separates the interfaces, remote object implementation, and client code into three packages:

- `compute` – `Compute` and `Task` interfaces
- `engine` – `ComputeEngine` implementation class
- `client` – `ComputePi` client code and `Pi` task implementation

First, you need to build the interface JAR file to provide to server and client developers.

Building a JAR File of Interface Classes

First, you need to compile the interface source files in the `compute` package and then build a JAR file that contains their class files. Assume that user `waldo` has written these interfaces and placed the source files in the directory `c:\home\waldo\src\compute` on Windows or the directory `/home/waldo/src/compute` on Solaris OS or Linux. Given these paths, you can use the following commands to compile the interfaces and create the JAR file:

Microsoft Windows:
```
cd c:\home\waldo\src
javac compute\Compute.java compute\Task.java
jar cvf compute.jar compute\*.class
```
Solaris OS or Linux:
```
cd /home/waldo/src
javac compute/Compute.java compute/Task.java
jar cvf compute.jar compute/*.class
```

The `jar` command displays the following output due to the `-v` option:

```
added manifest
adding: compute/Compute.class(in = 307) (out= 201)(deflated 34%)
adding: compute/Task.class(in = 217) (out= 149)(deflated 31%)
```

Now, you can distribute the `compute.jar` file to developers of server and client applications so that they can make use of the interfaces.

After you build either server-side or client-side classes with the `javac` compiler, if any of those classes will need to be dynamically downloaded by other Java virtual machines, you must ensure that their class files are placed in a network-accessible location. In this example, for Solaris OS or Linux this location is `/home/`*`user`*`/public_html/classes` because many web servers allow the accessing of a user's `public_html` directory through an HTTP URL constructed as `http://host/~`*`user`*`/`. If your web server does not support this convention, you could use a different location in the web server's hierarchy, or you could use a file URL instead. The file URLs take the form `file:/home/`*`user`*`/public_html/classes/` on Solaris OS or Linux and the form `file:/c:/home/`*`user`*`/public_html/classes/` on Windows. You may also select another type of URL, as appropriate.

The network accessibility of the class files enables the RMI runtime to download code when needed. Rather than defining its own protocol for code downloading, RMI uses URL protocols supported by the Java platform (for example, HTTP) to download code. Note that using a full, heavyweight web server to serve these class files is unnecessary. For example, a simple HTTP server that provides the functionality needed to make classes available for downloading in RMI through HTTP can be found at http://java.sun.com/javase/technologies/core/basic/rmi/class-server.zip.

Building the Server Classes

The `engine` package contains only one server-side implementation class, `ComputeEngine`, the implementation of the remote interface `Compute`.

Assume that user `ann`, the developer of the `ComputeEngine` class, has placed `ComputeEngine.java` in the directory `c:\home\ann\src\engine` on Windows or the directory `/home/ann/src/engine` on Solaris OS or Linux. She is deploying the class files for clients to download in a subdirectory of her `public_html` directory, `c:\home\ann\public_html\classes` on Windows or `/home/ann/public_html/classes` on Solaris OS or Linux. This location is accessible through some web servers as `http://`*`host`*`:`*`port`*`/~ann/classes/`.

The `ComputeEngine` class depends on the `Compute` and `Task` interfaces, which are contained in the `compute.jar` JAR file. Therefore, you need the `compute.jar` file in your class path when you build the server classes. Assume that the `compute.jar` file is located in the directory `c:\home\ann\public_html\classes` on Windows or the directory `/home/ann/public_html/classes` on Solaris OS or Linux. Given these paths, you can use the following commands to build the server classes:

Microsoft Windows:

```
cd c:\home\ann\src
javac -cp c:\home\ann\public_html\classes\compute.jar
    engine\ComputeEngine.java
```

Solaris OS or Linux:

```
cd /home/ann/src
javac -cp /home/ann/public_html/classes/compute.jar
    engine/ComputeEngine.java
```

The stub class for `ComputeEngine` implements the `Compute` interface, which refers to the `Task` interface. So, the class definitions for those two interfaces need to be network-accessible for the stub to be received by other Java virtual machines such as the registry's Java virtual machine. The client Java virtual machine will already have these interfaces in its class path, so it does not actually need to download their definitions. The `compute.jar` file under the `public_html` directory can serve this purpose.

Now, the compute engine is ready to deploy. You could do that now, or you could wait until after you have built the client.

Building the Client Classes

The `client` package contains two classes, `ComputePi`, the main client program, and `Pi`, the client's implementation of the `Task` interface.

Assume that user `jones`, the developer of the client classes, has placed `ComputePi.java` and `Pi.java` in the directory `c:\home\jones\src\client` on Windows or the directory `/home/jones/src/client` on Solaris OS or Linux. He is deploying the class files for the compute engine to download in a subdirectory of his `public_html` directory, `c:\home\jones\public_html\classes` on Windows or `/home/jones/public_html/classes` on Solaris OS or Linux. This location is accessible through some web servers as `http://`*`host`*`:`*`port`*`/~jones/classes/`.

The client classes depend on the `Compute` and `Task` interfaces, which are contained in the `compute.jar` JAR file. Therefore, you need the `compute.jar` file in your class path when you build the client classes. Assume that the `compute.jar` file is located in the directory `c:\home\jones\public_html\classes` on Windows or the directory `/home/jones/public_html/classes` on Solaris OS or Linux. Given these paths, you can use the following commands to build the client classes:

Microsoft Windows:
```
cd c:\home\jones\src
javac -cp c:\home\jones\public_html\classes\compute.jar
    client\ComputePi.java client\Pi.java
mkdir c:\home\jones\public_html\classes\client
cp client\Pi.class
    c:\home\jones\public_html\classes\client
```
Solaris OS or Linux:
```
cd /home/jones/src
javac -cp /home/jones/public_html/classes/compute.jar
    client/ComputePi.java client/Pi.java
mkdir /home/jones/public_html/classes/client
cp client/Pi.class
    /home/jones/public_html/classes/client
```

Only the `Pi` class needs to be placed in the directory `public_html\classes\client` because only the `Pi` class needs to be available for downloading to the compute engine's Java virtual machine. Now, you can run the server and then the client.

Running the Programs

A Note About Security

The server and client programs run with a security manager installed. When you run either program, you need to specify a security policy file so that the code is granted the security permissions it needs to run. Here is an example policy file to use with the server program:
```
grant codeBase "file:/home/ann/src/" {
    permission java.security.AllPermission;
};
```

Here is an example policy file to use with the client program:

```
grant codeBase "file:/home/jones/src/" {
    permission java.security.AllPermission;
};
```

For both example policy files, all permissions are granted to the classes in the program's local class path, because the local application code is trusted, but no permissions are granted to code downloaded from other locations. Therefore, the compute engine server restricts the tasks that it executes (whose code is not known to be trusted and might be hostile) from performing any operations that require security permissions. The example client's `Pi` task does not require any permissions to execute.

In this example, the policy file for the server program is named `server.policy`, and the policy file for the client program is named `client.policy`.

Starting the Server

Before starting the compute engine, you need to start the RMI registry. The RMI registry is a simple server-side bootstrap naming facility that enables remote clients to obtain a reference

to an initial remote object. It can be started with the `rmiregistry` command. Before you execute `rmiregistry`, you must make sure that the shell or window in which you will run `rmiregistry` either has no `CLASSPATH` environment variable set or has a `CLASSPATH` environment variable that does not include the path to any classes that you want downloaded to clients of your remote objects.

To start the registry on the server, execute the `rmiregistry` command. This command produces no output and is typically run in the background. For this example, the registry is started on the host `zaphod`.

Microsoft Windows (use `javaw` if `start` is not available):

```
start rmiregistry
```

Solaris OS or Linux:

```
rmiregistry &
```

By default, the registry runs on port 1099. To start the registry on a different port, specify the port number on the command line. Do not forget to unset your `CLASSPATH` environment variable.

Microsoft Windows:

```
start rmiregistry 2001
```

Solaris OS or Linux:

```
rmiregistry 2001 &
```

Once the registry is started, you can start the server. You need to make sure that both the `compute.jar` file and the remote object implementation class are in your class path. When you start the compute engine, you need to specify, using the `java.rmi.server.codebase` property, where the server's classes are network accessible. In this example, the server-side classes to be made available for downloading are the `Compute` and `Task` interfaces, which are available in the `compute.jar` file in the `public_html\classes` directory of user `ann`. The compute engine server is started on the host `zaphod`, the same host on which the registry was started.

Microsoft Windows:

```
java -cp c:\home\ann\src;c:\home\ann\public_html\classes\compute.jar
     -
Djava.rmi.server.codebase=file:/c:/home/ann/public_html/classes/compute.jar
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=server.policy
        engine.ComputeEngine
```

Solaris OS or Linux:

```
java -cp /home/ann/src:/home/ann/public_html/classes/compute.jar
     -Djava.rmi.server.codebase=http://zaphod/~ann/classes/compute.jar
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=server.policy
        engine.ComputeEngine
```

The above `java` command defines the following system properties:

- The `java.rmi.server.codebase` property specifies the location, a codebase URL, from which the definitions for classes originating *from* this server can be downloaded. If the codebase specifies a directory hierarchy (as opposed to a JAR file), you must include a trailing slash at the end of the codebase URL.
- The `java.rmi.server.hostname` property specifies the host name or address to put in the stubs for remote objects exported in this Java virtual machine. This value is the host name or address used by clients when they attempt to communicate remote method invocations. By default, the RMI implementation uses the server's IP address as indicated by the `java.net.InetAddress.getLocalHost` API. However, sometimes, this address is not appropriate for all clients and a fully qualified host name would be more effective. To ensure that RMI uses a host name (or IP address) for the server that is routable from all potential clients, set the `java.rmi.server.hostname` property.
- The `java.security.policy` property is used to specify the policy file that contains the permissions you intend to grant.

Starting the Client

Once the registry and the compute engine are running, you can start the client, specifying the following:

- The location where the client serves its classes (the `Pi` class) by using the `java.rmi.server.codebase` property
- The `java.security.policy` property, which is used to specify the security policy file that contains the permissions you intend to grant to various pieces of code
- As command-line arguments, the host name of the server (so that the client knows where to locate the `Compute` remote object) and the number of decimal places to use in the π calculation

Start the client on another host (a host named `ford`, for example) as follows:

Microsoft Windows:

```
java -cp c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar
     -Djava.rmi.server.codebase=file:/c:/home/jones/public_html/classes/
     -Djava.security.policy=client.policy
        client.ComputePi zaphod.east.sun.com 45
```

Solaris OS or Linux:

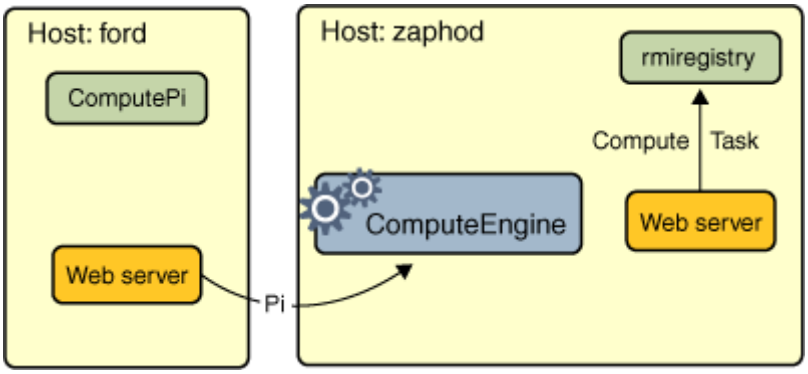
```
java -cp /home/jones/src:/home/jones/public_html/classes/compute.jar
     -Djava.rmi.server.codebase=http://ford/~jones/classes/
     -Djava.security.policy=client.policy
        client.ComputePi zaphod.east.sun.com 45
```

Note that the class path is set on the command line so that the interpreter can find the client classes and the JAR file containing the interfaces. Also note that the value of the `java.rmi.server.codebase` property, which specifies a directory hierarchy, ends with a trailing slash.

After you start the client, the following output is displayed:

```
3.14159265358979323846264338327950288419716939937
```

The following figure illustrates where the `rmiregistry`, the `ComputeEngine` server, and the `ComputePi` client obtain classes during program execution.



When the `ComputeEngine` server binds its remote object reference in the registry, the registry downloads the `Compute` and `Task` interfaces on which the stub class depends. These classes are downloaded from either the `ComputeEngine` server's web server or file system, depending on the type of codebase URL used when starting the server.

Because the `ComputePi` client has both the `Compute` and the `Task` interfaces available in its class path, it loads their definitions from its class path, not from the server's codebase.

Finally, the `Pi` class is loaded into the `ComputeEngine` server's Java virtual machine when the `Pi` object is passed in the `executeTask` remote call to the `ComputeEngine` object. The `Pi` class is loaded by the server from either the client's web server or file system, depending on the type of codebase URL used when starting the client.